UNIVERSITÉ DU QUÉBEC À MONTRÉAL

KPSILON :
CONCEPTION ET MISE EN ŒUVRE
DES FONDATIONS
D'UN SYSTÈME DE CALCUL FORMEL EN C++

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

THOMAS ROBERT DE MASSY

DÉCEMBRE 2024

REMERCIEMENTS

Je tiens avant tout à remercier mon directeur de recherche, Guy Tremblay, sans qui ce mémoire ne serait pas ce qu'il est. Sa rigueur et ses judicieux conseils m'ont, tout au long de ce projet, « *challengé* » en m'incitant à défendre mes choix et à explorer d'autres avenues, lorsque c'était préférable. Je le remercie aussi pour son précieux temps, et surtout pour sa grande écoute et sa patience. Son implication a perduré jusqu'à la toute fin et a été particulièrement impressionnante lors du sprint final de révision.

J'éprouve une profonde reconnaissance envers ma mère Brigitte et mon beau-père Gérald qui, lorsque j'étais âgé de treize ans et rêvant d'un ordinateur pour apprendre la programmation, m'ont pris au sérieux — sans prendre pour acquis que ce n'était encore qu'un caprice d'enfant. Merci d'avoir cru en moi et de m'avoir fait ce cadeau, même si vous n'en aviez pas vraiment les moyens ! Je n'ai jamais cessé de programmer depuis et ce ne sera pas pour bientôt.

Je veux aussi remercier mon ami Guillaume, pour l'accès qu'il m'a offert à son serveur de développement et, ainsi, me sauver de nombreuses heures d'attente avant la fin d'une recompilation complète de mon code source — 22 secondes *versus* 28 minutes, dans certaines situations ! Ouf, merci !

Je voudrais aussi remercier le professeur Étienne Gagnon, pour ses excellents cours de compilation et son élégant compilateur de compilateur *SableCC*. Dans ses deux cours, j'ai finalement compris comment mettre en œuvre un langage de programmation, un objectif que je souhaitais atteindre depuis de nombreuses années. Les discussions que nous avons eues sur le sujet m'ont suivi tout au long de ce projet. Et, oui, il avait raison sur deux points cruciaux, que j'ai fini par réaliser au cours de ce projet de maîtrise.

Et je m'excuse sincèrement à tous mes proches si je les ai « *assommés* » pendant ces longues années avec mes histoires de *garbage collector* et de calcul à précisions arbitraires — même si je risque d'en parler quand même encore un peu…

Je dédie ce mémoire à François.
J'aurais bien aimé savoir ce que tu en aurais pensé papa...

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

RÉSUMÉ

En mathématiques et en informatique, il existe deux principales méthodes de calcul, soit le *calcul numérique* et le *calcul symbolique*. Le calcul numérique repose sur l'utilisation d'algorithmes, issus de l'analyse numérique, permettant la résolution de problèmes mathématiques — par exemple, calcul d'intégrales et résolution d'équations différentielles. Or, pour de nombreux problèmes, les résultats produits, même lorsque précis, ne sont en fait que des *approximations*. Le calcul symbolique, quant à lui, permet de retourner des résultats exacts par manipulations symboliques d'expressions mathématiques — si celles-ci peuvent être résolues analytiquement. Ces manipulations symboliques peuvent être, entre autres, sous forme de simplification, de réduction de fraction, de changement de forme, de factorisation, de différentiation et d'intégration indéfinie.

Les logiciels utilisant le calcul symbolique sont appelés des *systèmes de calcul formel* (*computer algebra systems*). Dans ce mémoire, nous nous intéressons plus spécifiquement à la conception et à la mise en œuvre *des fondations* d'un tel système.

Pour ce faire, nous avons conçu et mis en œuvre un langage d'expressions arithmétiques et son interpréteur — à l'aide du compilateur de compilateur *ANTLR 4*. Ce langage supporte l'*arithmétique à précision arbitraire*, c.-à-d., à des précisions limitées uniquement par la mémoire disponible. Différents types multiprécision sont disponibles, et ce, au moyen de bibliothèques à licence libre de droits ou *LGPL*, tant des bibliothèques multiprécision en base 2 (*GMP*, *GNU MPFR*) qu'en base 10 (*Boost.Multiprecision*, *Mpdecimal*). Pour la gestion des allocations mémoire, nous avons utilisé le *ramasse-miettes Boehm-Demers-Weiser*, qui se charge, en arrière-plan, de libérer la mémoire allouée aux instances d'objets qui ne sont plus utilisés. Un environnement de développement a aussi été mis en place pour vérifier la qualité du code `C++`.

Pour valider notre travail, nous avons intégré à *Kpsilon* différentes fonctionnalités permettant de vérifier les résultats — entre autres, quant à la précision des résultats — et d'évaluer les performances ainsi que la consommation mémoire, notamment pour assurer que le ramasse-miettes fonctionne correctement et satisfait un certain nombre d'exigences que nous avions fixées et que nous décrivons.

Mots clés : système de calcul formel, calcul symbolique, arithmétique à précision arbitraire, langage de programmation, interpréteur, compilateur de compilateur, ramasse-miettes, *garbage collector*, `C++`.

INTRODUCTION

À l'avènement de l'informatique, à la fin des années 40, les premiers ordinateurs étaient utilisés essentiellement pour des calculs numériques. Grâce à la croissance exponentielle de la puissance des ordinateurs (Moore, 1965) — tant sur le plan de la vitesse des processeurs que de la quantité de mémoire — la précision de ces calculs numériques s'est améliorée — pensons entre autres aux prévisions météorologiques, qui sont devenues plus fiables. Il n'en reste pas moins que plusieurs de ces calculs numériques sont en fait des *approximations*, calculées en utilisant des algorithmes issus de l'*analyse numérique*[1].

L'amélioration de la puissance des ordinateurs a aussi permis le développement de logiciels pouvant utiliser des méthodes de calcul exactes et non numériques, logiciels appelés des ***systèmes de calcul formel***. Ces logiciels permettent de manipuler *symboliquement* des expressions mathématiques dans le but de retourner une réponse exacte, sans calculer le résultat sous une forme numérique approximative. Par manipulations symboliques d'expressions mathématiques, nous entendons : simplification (ex. réduction de fraction), changement de forme (ex. expansion de produits), factorisation, différentiation et intégration sans approximation (ex. calcul de primitive/intégrale indéfinie), etc. Le chapitre I présente et compare les méthodes de calcul numérique et symbolique, puis présente quelques logiciels numériques et systèmes de calcul formel existants.

Durant notre scolarité de baccalauréat et de maîtrise, nous avons implémenté deux prototypes de système de calcul formel en *Java* (à l'aide du *compilateur de compilateur SableCC*) dans le cadre des cours *INF5000 Théorie et construction des compilateurs* et *INF7641 Compilation*. Le premier prototype nous a permis de mieux identifier l'approche à utiliser pour implémenter le *calcul symbolique*, notamment, simplification d'expression, différentiation et intégration de fonctions simples. Le langage du deuxième prototype nous a permis d'obtenir un avant-goût de la syntaxe d'un langage, *Kpsilon*, que nous présenterons ultérieurement. Cependant, ces prototypes étaient lents, car plusieurs aspects avaient été implémentés naïvement, faute de temps — par exemple, la gestion de la mémoire.

La motivation initiale de notre projet de maîtrise était donc de développer, dans le langage `C++`, un système de calcul formel multiplateforme (*cross-platform*) et un langage associé inspiré du *pseudocode* — tous

[1] Soulignons que certaines équations mathématiques *ne peuvent pas* être résolues de façon analytique (i.e., algébrique), par ex., de nombreuses équations différentielles non linéaires fréquemment rencontrées en physique (Li *et al.*, 2007).

deux nommés *Kpsilon*[2]. Il s'agissait d'un projet personnel de longue date qui allait, sans nul doute, requérir plusieurs années de travail — ce qui s'est effectivement confirmé en cours de projet.

Dès le départ, les deux principales exigences que nous nous étions imposées ont grandement complexifié la mise en œuvre de ce logiciel : i) l'utilisation du langage `C++` ; ii) le fait qu'il puisse être compilé et exécuté sous les plateformes *Linux* et *Windows*[3]. En effet, l'utilisation de langages multiplateformes comme le *Python* ou le *Java* dans ce type de projet élimine d'emblée la nécessité d'adapter le code pour supporter différentes plateformes. De plus, puisque ces langages utilisent un ramasse-miettes (*garbage collector*), ils évitent de devoir mettre en place des mécanismes de gestion explicites de la mémoire. C'est donc peu dire que l'utilisation du `C++` a augmenté la complexité de ce projet. Néanmoins, pour des raisons de performance, nous tenions à utiliser un langage compilé en code machine natif (c.-à-d. non interprété et sans machine virtuelle exécutant du *bytecode*) tels que `C`, `C++` ou *Rust*. Pour des raisons de familiarité avec le langage, ainsi qu'une préférence envers la programmation orientée objet, nous avons donc choisi d'utiliser `C++`.

D'autres contraintes que nous voulions respecter étaient les suivantes :

- Nous voulions mettre en place un *écosystème d'outils de développement et de débogage*, car pour la suite à plus long terme de notre projet, il nous semblait important de s'assurer de la qualité et de la robustesse du code avant d'y intégrer d'autres fonctionnalités. Ceci nécessitait d'intégrer et d'utiliser plusieurs outils de vérification de code, de débogage, de profilage — comme nous le décrirons à la section 5.5.
- Pour valider le bon fonctionnement du ramasse-miettes et comparer les différentes bibliothèques multiprécision, nous voulions mettre un accent sur *l'intégration de tests de performance et de consommation mémoire*. En outre, nous voulions que ces tests puissent générer automatiquement les divers graphes de temps et de consommation mémoire — présentés aux chapitres VI et VII.
- Pour permettre les comparaisons de performance, il était aussi primordial qu'après toute modification majeure apportée au logiciel, il soit possible d'exécuter facilement une version antérieure, pour la comparer avec la version modifiée.
- Finalement, nous souhaitions que notre logiciel supporte les caractères accentués et *Unicode*, ainsi que l'affichage en couleurs, tant sous *Windows* que *Linux*.

[2]Dont le code source sera ouvert et distribué sous licence permissive *MIT*.

[3]Pour l'instant, le logiciel ne supporte pas *macOS*, faute d'accès à un *Mac*, mais des mesures ont été prises pour faciliter une future compatibilité avec cette plateforme. La liste des plateformes, architectures et compilateurs `C++` testés se trouve à l'annexe A.

Nous avons rapidement réalisé que, dans le cadre d'un projet de maîtrise — à temps partiel de surcroît —, la portée de notre projet devait être réévaluée et qu'il était plus réaliste de ne nous concentrer que sur certains éléments de base, et non sur l'ensemble du projet. Notre objectif est dès lors devenu de mettre en place des « *fondations solides* » sur lesquelles pourront s'appuyer les fonctionnalités qui seront ajoutées ultérieurement au logiciel. Donc, il nous faut dès maintenant préciser que *le volet « calcul symbolique » n'est pas implémenté* à l'heure actuelle, que le projet présenté dans ce mémoire a plutôt consisté à **concevoir et mettre en œuvre les fondations** qui permettront d'atteindre éventuellement l'objectif de développer un système complet de calcul formel en C++.

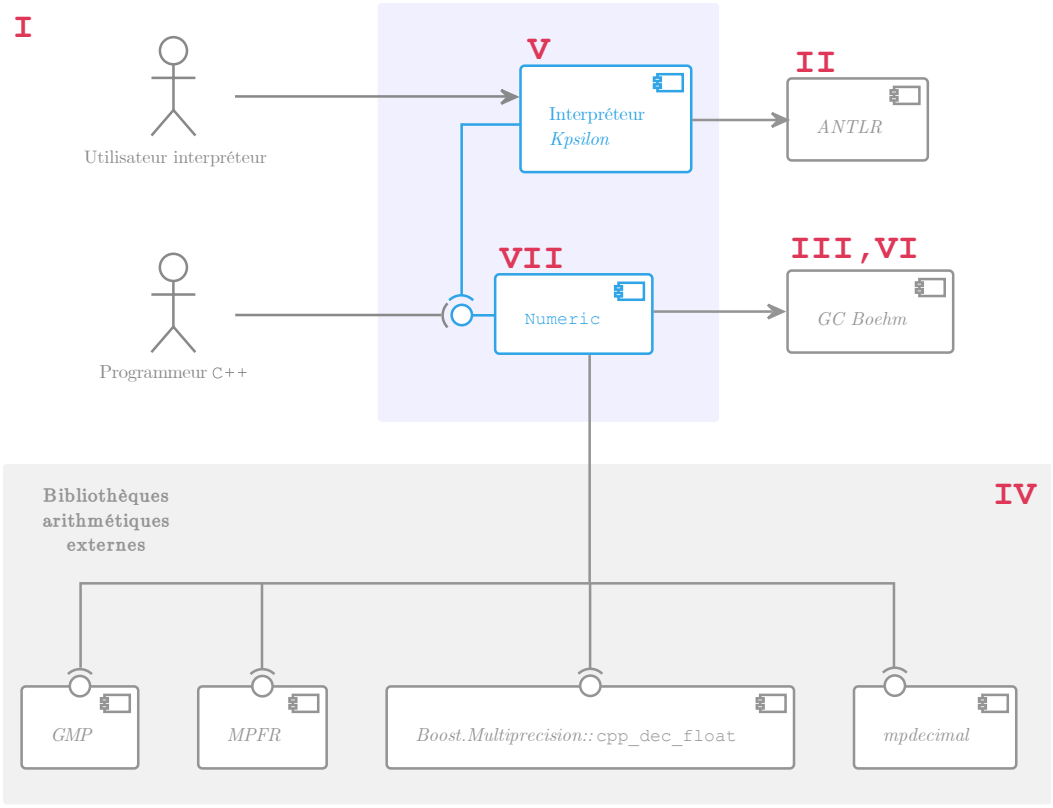


Figure I.1 Diagramme *UML* de composants du logiciel *Kpsilon*.

La figure I.1 présente une représentation *UML* simplifiée (diagramme de composants) des différents éléments qui seront discutés dans ce mémoire : les numéros de chapitres correspondants sont indiqués en rouge.

Pour faciliter la spécification et l'implémentation du langage *Kpsilon* ainsi que de son interpréteur, il était crucial de choisir un ***compilateur de compilateur*** C++ approprié. Notre choix s'est arrêté sur *ANTLR 4*. Le chapitre II explique le rôle d'un compilateur de compilateurs et en présente quelques-uns, alors que le chapitre V présente l'interpréteur *Kpsilon*, réalisé avec *ANTLR 4*.

C'est dans l'optique d'ériger les différents composants de l'interpréteur *Kpsilon* sur des bases solides qu'énormément de temps a été consacré à la mise en place d'un ***ramasse-miettes*** pour la gestion de la mémoire, et ce, au moyen du *Boehm-Demers-Weiser garbage collector* — plus communément appelé le *Boehm garbage collector*. Le chapitre III introduit le concept de ramasse-miettes et, comme nous le verrons au chapitre VI, qui explique le fonctionnement du ramasse-miettes *Boehm* et présente les principales difficultés rencontrées, ce point est de loin celui qui a été le plus problématique de notre projet !

Il était aussi nécessaire de créer l'infrastructure pour représenter et manipuler des valeurs numériques à des précisions arbitraires (c.-à-d., à des précisions limitées uniquement par la quantité de mémoire disponible) et permettre les calculs à ***arithmétique multiprécision***. Pour ce faire, plusieurs bibliothèques `C++` ont été testées et nous avons finalement retenu les suivantes : *GMP* (*MPIR*), *GNU MPFR*, *Boost.Multiprecision* et *Mpdecimal*. Comme expliqué au chapitre IV, elles ont chacune leurs avantages et désavantages — entre autres dû au fait que certaines utilisent une représentation en base 2 (binaire), alors que d'autres utilisent une représentation en base 10 (décimale). Pour permettre les calculs entre des valeurs de ces différents types, nous avons mis en œuvre un mécanisme de ***dispatch dynamique***[4]. L'architecture `C++` complète des types numériques ainsi qu'une introduction à la notion de *dispatch dynamique* sont présentées au chapitre VII. De plus, comme expliqué dans ce chapitre, cette partie du projet a aussi nécessité la mise en œuvre d'un ***préprocesseur C++ « maison »*** en *Python* (annexe C).

L'interpréteur *Kpsilon* mis en œuvre est relativement simple en ce qui concerne les « *calculs* » permis : il s'agit d'une calculatrice élémentaire supportant les quatre opérations arithmétiques de base, les expressions avec parenthèses et la constante π, et ce, pour différents types de nombres à précision arbitraire. Toutefois, pour faciliter la mise en œuvre des fondations et des fonctionnalités ultérieures, l'interpréteur supporte aussi de nombreuses commandes de débogage, notamment, pour le débogage de la gestion mémoire et pour la vérification du bon fonctionnement du ramasse-miettes. Des exemples de ces commandes sont présentés à la section 5.3 et la liste complète des commandes supportées se trouve en annexe A (tableau A.2).

[4] Il s'agit d'un anglicisme couramment utilisé qui pourrait être librement traduit en « *répartition dynamique* », « *aiguillage dynamique* » ou « *sélection dynamique* ».

CHAPITRE I
CALCUL NUMÉRIQUE ET CALCUL SYMBOLIQUE

Dans ce chapitre, nous présentons un aperçu des méthodes de calcul numérique et symbolique, dans le but de motiver les choix qui ont été faits pour développer les fondations d'un système de calcul formel. Ces choix et décisions portent notamment sur l'approche de la gestion mémoire à adopter, les types numériques de base à utiliser, et autres points dont nous discuterons dans les prochains chapitres. Bien que le *calcul symbolique* n'ait pas été mis en œuvre, tout a été mis en place dans l'optique qu'il sera implémenté par la suite. Nous expliquerons, entre autres, pourquoi l'utilisation d'un *ramasse-miettes* (*garbage collector* ou *GC*, chapitre III) pour la gestion mémoire et de nombres à précision arbitraire en base 10 (chapitre IV) est souhaitable.

Remarque: *Dans ce qui suit, il ne faut pas se laisser intimider par la présence de formules mathématiques dans certains exemples. Leur compréhension détaillée n'est pas nécessaire pour saisir l'essentiel du propos — elles ne sont présentes que pour illustrer les explications. Ce chapitre devrait donc, sans trop de difficulté, pouvoir être compris sans connaissance approfondie en calcul intégral ou numérique.*

1.1 Méthodes de calcul

Dans le domaine du calcul informatique, on peut classer la plupart des logiciels ou langages de programmation mathématiques en deux grandes catégories[5] (Malaquias et Lopes, 2007) : ***calcul numérique*** (*numerical computation*) ou ***calcul symbolique*** (*symbolic computation*). Comme précisé dans les sections suivantes, ces méthodes de calcul possèdent leurs avantages et désavantages respectifs. Un logiciel de calcul numérique utilise des algorithmes, étudiés en *analyse numérique* (une branche des mathématiques), pour obtenir une approximation des résultats, alors qu'un ***système de calcul formel*** (*Computer Algebra System* ou *CAS*) utilise le calcul symbolique pour manipuler, simplifier ou résoudre les expressions/équations mathématiques. Néanmoins, un logiciel (ou langage) peut appartenir aux deux catégories, s'il supporte les deux méthodes de calcul.

[5] Bien qu'il existe d'autres catégories spécialisées, par exemple, *analyse statistique* (ex. *R*), *simulation* (ex. *MathWorks Simulink*) et géométrie (ex. *GeoGebra*).

La différence entre ces deux méthodes peut être illustrée à l'aide d'un exemple. Ainsi, soit les équations suivantes avec une intégrale définie évaluée de façon numérique et de façon symbolique :

$$\int_0^{\frac{\pi}{4}} e \sin x \, dx \approx 0{,}796\,166\,314\,379\,486\,82 \tag{1.1}$$

$$\int_0^{\frac{\pi}{4}} e \sin x \, dx = \left(1 - \frac{1}{\sqrt{2}}\right) e \tag{1.2}$$

Ici, le calcul numérique (équation 1.1) ne retourne qu'une approximation, alors que le calcul symbolique (équation 1.2) retourne un résultat exact sous forme d'une expression mathématique. Le résultat de l'équation 1.2 est en fait un *nombre irrationnel* et *transcendant*[6] — c'est-à-dire qu'il possède un nombre infini de décimales (non périodiques)[7] et ne peut pas être représenté sous forme d'une *fraction décimale*[8]. La valeur exacte de l'intégrale est donc l'expression mathématique calculée symboliquement (équation 1.2) et non pas le résultat approximatif calculé numériquement (équation 1.1).

1.2 Calcul numérique

L'utilisation du calcul numérique dans les logiciels remonte au tout début de l'informatique. Sans en être les précurseurs, quelques bibliothèques importantes pour le langage *FORTRAN*, telles que *EISPACK* et *LINPACK* (pour l'algèbre linéaire), ont été populaires dans les années 70 et 80 ; elles ont été depuis supplantées par la bibliothèque *LAPACK* (*Linear Algebra PACKage*).

Le logiciel propriétaire *MathWorks MATLAB* (*MATrix LABoratory*) (Higham et Higham, 2017) développé par Moler à la fin des années 70, ainsi que les logiciels libres *GNU Octave* (un quasi-clone de *MATLAB*) et *Scilab*, sont considérés comme étant des logiciels de calcul numérique[9]. De plus, le langage de programmation multiparadigmes *Julia* (Bezanson *et al.*, 2017) peut aussi être classé dans cette catégorie. Plusieurs logiciels de calcul symbolique présentés à la section 1.3 supportent aussi le calcul numérique (ex. *Maplesoft Maple*),

[6] Ces affirmations ont été validées à l'aide de *Wolfram|Alpha*, par ex. :
`https://www.wolframalpha.com/input?i=is+%281-1%2Fsqrt%282%29%29e+transcendental?`

[7] Il ne peut pas être représenté sous forme d'une fraction d'entiers (*irrationnel*) et n'est racine d'aucun polynôme à coefficients entiers (*transcendant*).

[8] Un nombre appartenant à $\mathbb{D} = \left\{ \frac{a}{10^p}, a \in \mathbb{Z}, p \in \mathbb{N} \right\}$, le sous-ensemble des rationnels $\mathbb{Q}$ ayant un nombre fini de décimales.

[9] Quoique, comme nous le verrons à la section 1.3, *MATLAB* et *GNU Octave supportent aussi le calcul symbolique*.

mais ils sont généralement non optimisés pour cette tâche et lents. En outre, des bibliothèques de calcul numérique sont disponibles pour plusieurs langages de programmation généralistes comme *Python* et `C++`.

Comme mentionné brièvement à la section 1.1, le principal désavantage du calcul numérique réside dans le fait que le résultat retourné n'est souvent qu'une approximation. La majorité des logiciels et langages de calcul numérique utilisent des algorithmes étudiés dans une branche des mathématiques appelée l'*analyse numérique*. Or, ces algorithmes ne permettent d'approximer le résultat qu'à une certaine précision. Cette précision, bien qu'ajustable, s'obtient au détriment du temps d'exécution et, dans certains cas, de la consommation mémoire.

En contrepartie, ses avantages principaux sont sa simplicité en matière de programmation et sa rapidité d'exécution par rapport au calcul symbolique. Rapidité d'exécution, d'autant plus grande, lorsqu'un type de base (ex. un `double` 64 bits) est utilisé pour faire les calculs au niveau matériel, quand la précision des résultats est secondaire[10]. Néanmoins, comme on le constate dans le tableau 1.1 comparant les deux méthodes de calcul dans différents langages, pour certains problèmes le calcul symbolique peut s'avérer plus rapide — en particulier pour certains calculs à grandes précisions.

À titre d'exemple et pour expliquer comment est calculé le résultat numérique approximatif de l'équation 1.1, la *méthode de Simpson 1/3* (*Simpson's rule*), un algorithme permettant d'approximer les résultats d'intégrales définies de $f(x)$ sur l'intervalle $[a, b]$, est introduit pour clore cette section.

La *méthode de Simpson 1/3* est basée sur l'approximation suivante :

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right], h = \frac{b-a}{2} \tag{1.3}$$

Avec l'équation 1.3, le résultat n'est précis qu'à quelques décimales après la virgule. Pour obtenir une plus grande précision, une solution consiste à subdiviser l'intervalle $[a, b]$ en n sous-intervalles et à utiliser la *méthode de Simpson 1/3 composite*.

[10] Le manque de précision des types à virgule flottante de la *norme IEEE 754* sera expliqué au chapitre IV.

Formule pour le calcul (approximation) d'une intégrale définie avec la *méthode de Simpson 1/3 composite* pour n sous-intervalles :

$$\int_a^b f(x)\,dx \approx \frac{h}{6}\left[f(x_0) + 2\sum_{j=1}^{n-1} f(x_{2j}) + 4\sum_{j=1}^{n} f(x_{2j-1}) + f(x_{2n})\right] \tag{1.4}$$

où

- n est le nombre de sous-intervalles de $[a, b]$
- $h = \frac{b-a}{n}$
- $x_i = a + i\frac{h}{2}$ pour $i = 0, 1, ..., 2n-1, 2n$
- terme d'erreur : $err = -\frac{nh^5}{2880} f^{(4)}(\xi'), \xi' \in [a, b]$
- complexité temporelle : $\mathcal{O}(n)$
- complexité spatiale (consommation mémoire) : $\mathcal{O}(1)$

langage (*méthode*)	résultat	temps (*secs*)	erreur[11]
MapleSoft Maple (*symbolique*)	0,796 166 314 379 486 822 9	$\lesssim 1{,}6 \times 10^{-2}$	0
MapleSoft Maple (*numérique*)	0,796 166 314 379 486 824 5	$\approx 1{,}4 \times 10^{0}$	$3{,}4 \times 10^{-18}$
GNU Octave (*numérique*)	0,796 166 314 379 485 817 7	$\lesssim 1{,}0 \times 10^{-2}$	$1{,}3 \times 10^{-15}$
C++ (*numérique*)	0,796 166 314 379 486 705 9	$\approx 2{,}5 \times 10^{-5}$	$5{,}6 \times 10^{-16}$
Prototype 1 *Java/SableCC* (*symbolique*)	0,796 166 314 379 486 822 9	$\approx 3{,}4 \times 10^{-2}$	0
Prototype 1 *Java/SableCC* (*numérique*)	0,796 166 314 379 486 866 0	$\approx 2{,}6 \times 10^{0}$	$5{,}4 \times 10^{-17}$
Prototype 2 *Java/SableCC* (*numérique*)	0,796 166 314 379 486 866 0	$\approx 0{,}6 \times 10^{0}$	$5{,}4 \times 10^{-17}$

Tableau 1.1 Résultats de la méthode de Simpson composite pour $f(x) = e \sin x$ et $n = 2500$.

Pour $f(x) = e \sin x$ sur l'intervalle $[0, \frac{\pi}{4}]$ avec $n = 100$ sous-intervalles, le *résultat* de l'équation 1.4 est précis à 10 décimales après la virgule. Pour faire l'approximation numérique du résultat de l'équation 1.1 à une précision de 17 décimales, une valeur de $n \approx 2500$[12] est nécessaire et le calcul numérique prend environ 1,4 seconde avec *Maplesoft Maple* (tableau 1.1) sur un *Intel Core i7* 2,93*Ghz*, alors que le calcul symbolique ne prend que quelques millisecondes. Donc, pour obtenir une telle précision dans *Maple*, le calcul symbolique est plus performant que la méthode de Simpson 1/3 composite.

[11] Il s'agit ici de l'erreur relative, soit $err = \frac{|\Delta x|}{|x|}$, où x est la valeur attendue et $\Delta x = \textit{résultat} - x$.

[12] Cette valeur de n a été calculée avec *Maple* (fonction `ApproximateInt`), car la version gratuite de *Wolfram|Alpha* ne peut pas faire ce calcul pour un aussi grand n. De plus, *Maple* permet d'obtenir *symboliquement* la dérivée quatrième de $f(x)$ nécessaire pour calculer le terme d'erreur de l'équation 1.4 : $err = -\frac{nh^5}{2880} f^{(4)}(\xi') = -5{,}1 \times 10^{-18}$ si $\xi' = \frac{\pi}{4}$.

Mentionnons toutefois que la lenteur du calcul numérique est due au fait que *Maple* est un logiciel optimisé pour le calcul symbolique et non pour le calcul numérique. Les nombres qu'il utilise sont des *nombres à précision arbitraire*. Bien que plus lents, ces nombres permettent d'obtenir des précisions qui ne sont limitées que par la quantité de mémoire de l'ordinateur (cf. chapitre IV pour plus de détails). Les véritables logiciels de calcul numérique, ou même les langages généralistes comme le `C++`, utilisant des *nombres à virgule flottante IEEE 754 64 bits*, sont plus performants, bien que leur précision soit théoriquement limitée à 17 décimales[13] (cette limitation sera expliquée au chapitre IV).

Dans le tableau 1.1, on trouve aussi des résultats obtenus par les logiciels *GNU Octave* et `C++`. On constate qu'ils sont plus rapides que le calcul numérique de *Maple* (particulièrement le `C++`) et ne prennent que quelques microsecondes. Par contre, puisqu'ils utilisent des `double` *64 bits*, ils ne peuvent pas obtenir des précisions de plus de 15 (*GNU Octave*) ou 16 (`C++`) décimales, et ce, même à de grandes valeurs de n.

Notons que le tableau 1.1 présente aussi les résultats des prototypes *Java/SableCC* mentionnés en *introduction*. Dans le cas du premier prototype, les temps d'exécution, bien que deux fois plus lents, sont semblables aux temps de *Maple*. Par contre, le deuxième est environ deux fois plus rapide que *Maple* pour le calcul numérique. Ces différences de comportement des deux prototypes s'expliquent par le fait que le premier a été conçu avant tout pour le calcul symbolique, alors que le deuxième utilise un algorithme numérique. Cependant, les deux prototypes génèrent, avec $n = 2500$, une précision tronquée d'une décimale en comparaison avec celle de *Maple* ; ils doivent utiliser $n \approx 4000$ pour arriver à une précision identique.

Dans la majorité des cas, la rapidité des calculs que permet le calcul numérique, malgré son manque de précision, est justement le but recherché. En effet, rares sont les problèmes physiques et mathématiques où une précision de plus de 15 décimales après la virgule est nécessaire. La *NASA* (*National Aeronautics and Space Administration*) utilise une précision de 15 décimales pour la constante π pour ses missions spatiales[14] et, selon elle, 38 décimales de π sont suffisantes pour calculer la circonférence de l'univers (rayon de 46 milliards d'années lumière) à une précision du diamètre d'un atome d'hydrogène près. Pour cette raison, la majorité des logiciels de calcul numérique utilisent des nombres à virgule flottante *IEEE 754* de 64 bits.

[13] Cette valeur est une borne supérieure ; en pratique, elle est plutôt de 15 à 17 décimales — ce qui explique les précisions des résultats au tableau 1.1. Ce détail sera expliqué plus en détail au chapitre IV.

[14] `https://www.jpl.nasa.gov/edu/news/2016/3/16/how-many-decimals-of-pi-do-we-really-need/` (*archive*)

Cependant, quelques problèmes physiques et mathématiques requièrent de plus grandes précisions — c'est le cas, entre autres, des *oscillateurs anharmoniques* (80 décimales), de l'étude des systèmes atomiques de Coulomb à N *corps* (*n-body Coulomb atomic systems*, 120 décimales) et de l'analyse des *zéros de la fonction zêta de Riemann* (10 000 décimales) (Bailey et Borwein, 2015). De plus, d'un point de vue pédagogique, il peut être intéressant d'obtenir de plus grandes précisions ou même des résultats exacts sous forme d'expressions mathématiques — c'est ce que permet le calcul symbolique.

1.3 Systèmes de calcul formel et calcul symbolique

Un système de calcul formel (*Computer Algebra System ou CAS*) est un logiciel qui permet de manipuler, simplifier ou résoudre des équations et expressions mathématiques sous forme *symbolique*.

Par exemple, soit l'expression arithmétique $\frac{2\pi+2}{6}$ évaluée de deux façons :

$$\frac{2\pi+2}{6} \approx \frac{2\times 3{,}141\,592\,654+2}{6} \approx 1{,}380\,530\,884 \tag{1.5}$$

$$\frac{2\pi+2}{6} = \frac{\pi}{3}+\frac{1}{3} \tag{1.6}$$

Contrairement au calcul numérique (équation 1.5), le calcul symbolique (équation 1.6) ne calcule pas le résultat de façon approximative (sauf si explicitement demandé) ; il retourne plutôt une expression mathématique exacte. Les résultats calculés sont toujours exacts puisque seules des manipulations algébriques, des règles de différenciation/intégration, des équivalences mathématiques, etc., sont utilisées pour effectuer les calculs.

Un système de calcul symbolique est un excellent outil pédagogique puisqu'il peut aider à comprendre ou valider des concepts mathématiques (Koepf, 1999), et ce, encore plus si le logiciel affiche les étapes du calcul. Par exemple, les équations dans la figure 1.1 permettent d'expliquer comment l'équation 1.2 (p. 6) est résolue symboliquement : les règles d'intégration, les manipulations algébriques et les équivalences mathématiques utilisées sont affichées dans la colonne de droite.

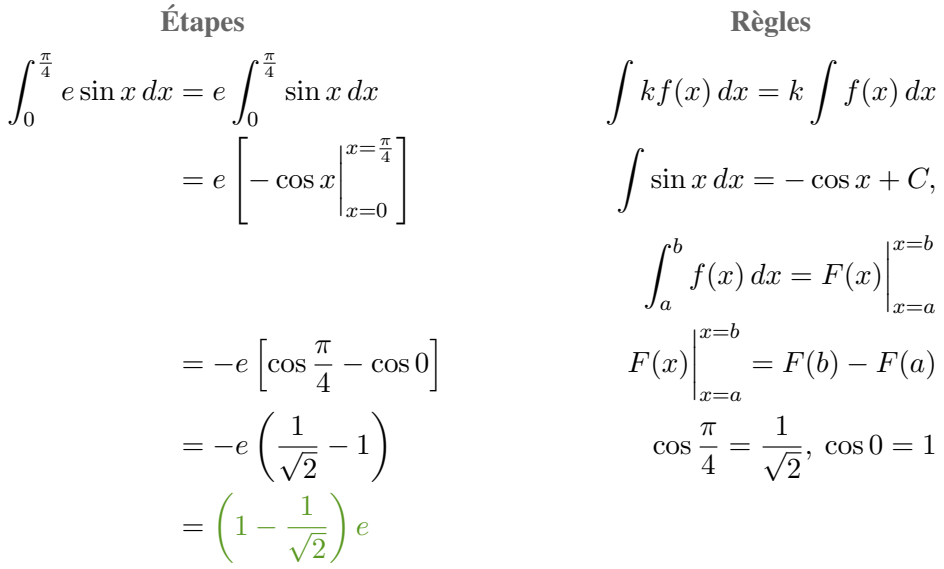


Figure 1.1 Étapes et règles pour résoudre symboliquement l'équation 1.2.

Les principaux désavantages du calcul symbolique sont qu'il est généralement plus demandant en temps d'exécution, et donc plus lent que le calcul numérique, et aussi plus complexe à mettre en œuvre. Cette lenteur est apparente lorsqu'il est utilisé dans un langage programmation de type *impératif*, c'est-à-dire lorsqu'il est utilisé de manière plus traditionnelle dans un code qui contient des boucles, des instructions conditionnelles et des branchements, etc. Cependant, comme mentionné à la section précédente, dans certains problèmes, le calcul symbolique s'avère plus rapide que le calcul numérique (tableau 1.1) — c'est le cas lorsqu'il n'est utilisé que pour résoudre/simplifier une équation/expression mathématique.

Schoonschip, développé par Veltman en 1963 (Weinzierl, 2002), est l'un des systèmes de calcul formel les plus anciens. Par la suite sont apparus *FORMAC* (*FORmula MAnipulation Compiler*), une extension de *FORTRAN IV*, développée par Sammet (Bond *et al.*, 1964), et *MATHLAB*, programmé en *Lisp* (Engelman, 1971). De plus, mentionnons que les logiciels *Macsyma* (Moses, 2012) (*project MAC's SYmbolic MAnipulator*), développé en 1968, et *REDUCE* (Hearn, 1968), font partie des systèmes de calcul formel les plus anciens toujours en opération (Hearn, 2008).

Plus récemment, à l'ère des interfaces graphiques, sont apparus des systèmes de calcul formel plus évolués et conviviaux. Pensons notamment au logiciel propriétaire *Maplesoft Maple* (anciennement *Waterloo Maple*) (Char *et al.*, 1983) développé par le *Symbolic Computation Group* (Univ. de Waterloo) et au

logiciel libre *Maxima* — tous deux créés en 1982 pour remplacer *Macsyma*. En 1988, Wolfram conçut le logiciel propriétaire *Wolfram Mathematica* (Wolfram, 1988) pour remplacer *SMP* (*Symbolic Mathematical Computation*) (Wolfram, 1985), un système de calcul formel qu'il avait créé en 1979 (commercialisé en 1981). *SMP* était critiqué à l'époque pour son manque de précision dû au fait qu'il utilisait des nombres à virgule flottante. *Wolfram|Alpha*, une interface web en langage naturel vers *Mathematica*, fut lancée en 2009.

D'autres systèmes de calcul formel valent aussi la peine d'être mentionnés. C'est le cas de *Derive*, un logiciel propriétaire lancé en 1988 (abandonné en 2007) par la compagnie *Soft Warehouse*, cette dernière ayant aussi développé *muMATH* à la fin des années 70. *Derive* a été acheté par *Texas Instruments* en 1999 pour être utilisé dans certaines calculatrices graphiques (*TI-89* et *TI-92*). Un autre logiciel historiquement important est *Axiom* (Jenks et Sutor, 1992) (anciennement *Scratchpad II*[15]), un logiciel programmé en *Lisp* et commercialisé dans les années 90. Mentionnons finalement *SageMath* (anciennement *SAGE* pour *System for Algebra and Geometry Experimentation*) créé en 2005 par Stein (Univ. de Washington) et sous licence *GPLv3*.

Certains logiciels de calcul numérique supportent aussi le calcul symbolique au moyen de modules externes. C'est le cas de *MathWorks MATLAB*, avec le *Symbolic Math Toolbox* (avec *MuPAD*), et de *GNU Octave*, avec le module *Symbolic* (avec la bibliothèque *SymPy* de *Python*). Certains langages de programmation généralistes, tels *Python* et `C++`, supportent aussi le calcul symbolique au moyen de bibliothèques externes[16].

Deux points additionnels que nous voulons souligner à propos du calcul symbolique, évoqués en début de chapitre, concernent la gestion de la mémoire et la représentation des nombres.

Bien que l'utilisation d'un *ramasse-miettes* (chapitre III) pour la gestion mémoire soit souhaitable dans un logiciel de calcul traditionnel ou numérique, elle l'est davantage dans un système de calcul formel. En effet, dans un logiciel de calcul symbolique, des structures complexes d'objets doivent être stockées en mémoire pour y placer des expressions mathématiques et non seulement des variables et les valeurs associées, donc il devient plus difficile de faire la trace de ce qui peut ou non être libéré. Pensons à l'équation $y = 2x$ ou la fonction $f(y) = 2xy$ qui doivent être stockées dans une structure de données en mémoire : lors de leur destruction, on ne doit pas supposer que la variable x n'est plus utilisée — donc, elle ne doit pas être détruite

[15] *Scratchpad II*, créé en 1977, est lui-même basé sur *Scratchpad* créé en 1965 chez *IBM* (jamais rendu public). *IBM* renomma *Scratchpad II* en *Axiom* en 1990 pour en faire un produit commercial. Il est maintenant distribué sous licence *BSD*.

[16] Entre autres, les bibliothèques *SymPy* (*Python*) et *GiNaC* (`C++`). Notons que la licence *GPL* utilisée par la bibliothèque *GiNaC* n'est malheureusement pas compatible avec la licence *MIT* qu'utilise *Kpsilon*.

automatiquement (ex : par le destructeur de l'objet qui l'utilisait).

Une telle subtilité concernant le stockage des expressions mathématiques fait en sorte qu'il devient difficile de faire soi-même la gestion de la mémoire dans un contexte de calcul symbolique. Un ramasse-miettes élimine ce problème, puisqu'il est en mesure de détecter automatiquement quand un objet n'est plus utilisé et de le détruire le cas échéant. Le tout se fait en arrière-plan, sans intervention du programmeur, ce qui facilite la gestion des allocations mémoire, et ce, sans avoir à suivre à tout moment la trace des utilisations des objets.

Finalement, comme nous l'expliquerons au chapitre IV, il est important de garantir la validité de la *dernière décimale* affichée à une certaine précision définie lors d'une approximation d'un résultat exact. Dans un contexte symbolique, il est donc préférable d'utiliser des nombres à précision arbitraire en base 10 (décimale) plutôt qu'en base 2 (binaire).

1.4 Avantages et désavantages des différentes méthodes de calcul : un exemple avec *GNU Octave*

Pour clore ce chapitre, nous présentons une expérience que nous avons effectuée, en utilisant *GNU Octave*, pour comparer les résultats et temps d'exécution de différentes méthodes de calcul. Il s'agit d'une série de tests d'une sommation d'une expression arithmétique utilisant des nombres irrationnels, soit $\sum_{k=1}^{n} \frac{ek}{\pi}$, en utilisant les méthodes de calcul numérique et symbolique :

1. Une simple boucle `for`[17] (calcul simple avec flottants).
2. La fonction `sum` (calcul numérique).
3. La fonction symbolique `symsum` avec le module *Symbolic* (calcul symbolique).
4. Une boucle `for`[17] avec le module *Symbolic* (calcul symbolique, effectué si $n \leq 100\,000$).
5. La fonction `sum` avec le module *Symbolic* (calcul symbolique, effectué si $n \leq 100\,000$).

[17] Pseudocode : **FOR** $k = 1$ **TO** n**;** $sum = sum + (e \times k)/\pi$**;** **END**, où sum, e et π sont symboliques.

Voici les résultats pour $n = 100\,000$, où les chiffres en rouge sont erronés :

- Résultat des calculs simple et numérique (*1.* et *2.*)

$$\sum_{k=1}^{100\,000} \frac{ek}{\pi} \approx 4\,326\,323\,159{,}960\,292\,816\,162\,109\,38 \tag{1.7}$$

- Résultat des calculs symboliques (*3.*, *4.* et *5.*)

$$\begin{aligned} \sum_{k=1}^{100\,000} \frac{ek}{\pi} &= \frac{5\,000\,050\,000e}{\pi} \\ &\approx 4\,326\,323\,159{,}960\,297\,049\,343\,234\,84 \end{aligned} \tag{1.8}$$

On constate (équation 1.7) que ni le calcul simple avec les primitives du langage (points flottants et boucle `for`), ni le calcul numérique au moyen de la fonction `sum` ne permettent d'obtenir une précision de plus de cinq décimales (, 96029). Ceci s'explique par le fait que *GNU Octave* utilise des `double` 64 bits (*IEEE 754*) par défaut et ne permet pas d'obtenir des résultats exacts (ce problème est expliqué en détail au chapitre IV). Les tests utilisant le calcul symbolique produisent eux des résultats exacts ($\frac{5\,000\,050\,000e}{\pi}$ dans l'équation 1.8), bien qu'une approximation survienne lors de l'affichage d'un nombre limité de chiffres : la présence du 4 à la fin s'explique parce que la valeur affichée est arrondie automatiquement (. . . 836 . . . ⇒ . . . 84).

```
Test Sum (k*exp(1))/pi, k=1..100000

1. Traditional computing (for)...done
     (0.000000 secs)
  ≈  4326323159.96029281616210938

2.  Numeric computing (sum)...done
     (0.008000 secs)
  ≈  4326323159.96029281616210938

3.  Symbolic computing (symsum)...done
     (1.008000 secs)
  =
  5000050000e
  -----------
       π
  ≈  4326323159.96029704934323484

4.  Symbolic computing (sum)...done
     (10861.848000 secs)
  =
  5000050000e
  ----------
       π
  ≈  4326323159.96029704934323484

5.  Symbolic computing (for)...done
     (12420.984000 secs)
  =
  5000050000e
  -----------
       π
  ≈  4326323159.96029704934323484
```

Capture 1.1 Calcul numérique vs. calcul symbolique avec $k = 0$ à 100 000.

Signalons dans la capture 1.1 que la version numérique, qui utilise la fonction `sum`, prend $8\ ms$ pour faire le calcul, alors que la version symbolique utilisant cette même fonction `sum` prend environ 3 heures. Ceci s'explique parce que *GNU Octave* est un logiciel de calcul numérique et que son module *Symbolic* utilise *Python* pour faire les calculs symboliques. La même raison explique que la version symbolique utilisant la boucle `for` demande environ $3\frac{1}{2}$ heures. Par contre, la fonction symbolique `symsum` est passablement rapide, bien que les versions simple et numérique soient plus rapides pour $n = 100\,000$.

Résultats des tests pour $n = 10\,000\,000$:

- Résultat des calculs simple et numérique (*1.* et *2.*)

$$\sum_{k=1}^{10\,000\,000} \frac{ek}{\pi} \approx 43\,262\,803\,297\,893{,}140\,625\,000\,000\,000\,00 \tag{1.9}$$

- Résultat du calcul symbolique (*3.*)

$$\begin{aligned} \sum_{k=1}^{10\,000\,000} \frac{ek}{\pi} &= \frac{50\,000\,005\,000\,000e}{\pi} \\ &\approx 43\,262\,803\,297\,893{,}151\,522\,214\,175\,571\,18 \end{aligned} \tag{1.10}$$

On constate aussi (équation 1.9) que plus la valeur de n augmente, plus il y a perte de précision dans le cas des méthodes simple et numérique. Encore une fois, ceci est dû au manque de précision des *nombres à virgule flottante 64 bits (IEEE 754)* utilisés par défaut par *GNU Octave*. Par contre, on constate que dans les cas symboliques (équation 1.10), la valeur de n n'affecte aucunement la précision[18].

[18] Pour $n > 100\,000$, le code *GNU Octave* n'exécute pas les deux derniers calculs symboliques (avec les fonctions `for` et `sum`), car ils demanderaient trop de temps de calcul.

```
Test Sum (k*exp(1))/pi, k=1..10000000

1. Traditional computing (for)...done (68.440000 secs)
  ≈  43262803297893.14062500000000000

2. Numeric computing (sum)...done (1.360000 secs)
  ≈  43262803297893.14062500000000000

3. Symbolic computing (symsum)...done (0.008000 secs)
  =
  50000005000000e
  ---------------
         π
  ≈  43262803297893.15152221417557118
```

Capture 1.2 Calcul numérique vs. calcul symbolique avec $k = 1$ à $10\,000\,000$.

Dans le cas de $n = 10\,000\,000$ (capture 1.2), la situation est inversée par rapport à $n = 100\,000$: la version symbolique (`symsum`) prend $8\ ms$ et est plus rapide[19] que les versions simple et numérique. La version simple (`for`) est la plus affectée et prend environ 68,5 secondes, alors que la version numérique (`sum`) prend environ 1,4 seconde.

Il est possible que la rapidité de la version symbolique puisse être expliquée par le fait que, puisque `symsum` est une fonction de calcul symbolique, elle peut simplifier la sommation avant de la calculer, comme ceci :

$$\sum_{k=1}^{n} \frac{ek}{\pi} = \frac{e}{\pi} \sum_{k=1}^{n} k = \frac{e}{\pi} \left(\frac{n(n+1)}{2} \right) = \frac{e}{\pi} 50\,000\,005\,000\,000, \text{ si } n = 10\,000\,000$$

Avec cette simplification, il devient clair que le temps de calcul ne dépend plus de la valeur de n[20].

En conclusion, ce dernier exemple illustre donc comment le calcul symbolique possède des avantages par rapport au calcul numérique pour la résolution de certains problèmes.

[19] Nous ne comprenons cependant pas pourquoi le test avec $n = 10\,000\,000$ est plus rapide que le test avec $n = 100\,000$, car l'inverse serait plus logique.

[20] Car $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$.

CHAPITRE II
COMPILATEURS DE COMPILATEUR ET *ANTLR 4*

Un de nos objectifs pour *Kpsilon* étant de développer un langage et son interpréteur, nous présentons dans ce chapitre l'outil *ANTLR 4*[21] que nous avons choisi d'utiliser, précédé d'une brève introduction à la compilation et aux compilateurs de compilateur.

2.1 Introduction à la compilation et aux compilateurs de compilateur

Bien qu'il soit possible d'implémenter un interpréteur ou un compilateur en partant de zéro[22], il est préférable de ne pas réinventer la roue et d'utiliser plutôt des outils pour rendre cette étape du développement moins ardue. Ce type d'outils se nomme un compilateur de compilateur ou générateur de compilateur (*compiler-compiler* ou *compiler generator*).

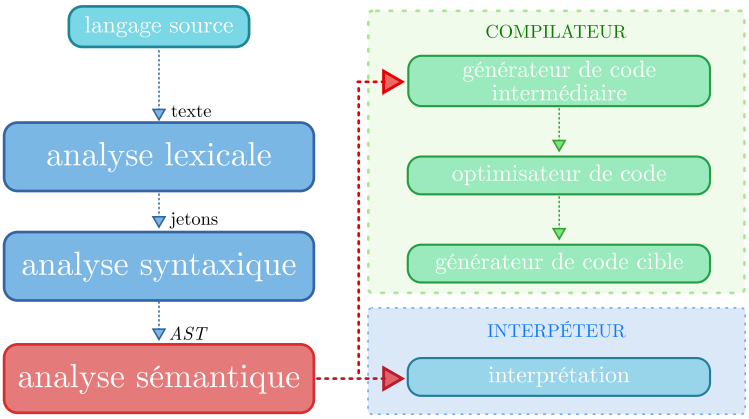


Figure 2.1 Phases d'un compilateur/interpréteur.

Le fonctionnement d'un compilateur ou d'un interpréteur peut être décomposé en différentes phases, telles que représentées dans la figure 2.1. Premièrement, une phase d'**analyse lexicale** (*lexical analysis*), recevant un contenu sous forme de texte, est effectuée pour identifier les **jetons** (*tokens*) valides (mots-clés, opérateurs, noms de fonctions/variables, etc.) du langage source. Deuxièmement, les jetons détectés sont envoyés vers la phase d'**analyse syntaxique** (*syntax analysis*) pour valider la syntaxe et produire en sortie un **arbre de**

[21] `https://www.antlr.org/` (*archive*)

[22] Le compilateur *gcc* utilisait le générateur d'analyseur syntaxique *GNU Bison* des les versions antérieures à 3.4, mais utilise un analyseur maison (*recursive-descent parser*) depuis 2004 (`https://gcc.gnu.org/gcc-4.1/changes.html` (*archive*)) dû à la complexité de la grammaire du langage `C++`, pour des raisons de performance, de facilité de maintenance et pour produire des messages d'erreurs plus précis. Les compilateurs *Clang* et *Rust* ont, quant à eux, toujours utilisé leur propre analyseur maison.

syntaxe abstraite (*Abstract Syntax Tree* ou *AST*). Un exemple d'arbre syntaxique se trouve dans la figure 2.4 (page 24). Puis, durant la phase d'**analyse sémantique**, les vérifications des définitions de types et de la validité sémantique des expressions sont effectuées.

Les phases d'analyses lexicale et syntaxique se font à partir d'un automate fini non déterministe (*AFN*, *Nondeterministic Finite Automaton* ou *NFA*) converti en automate fini déterministe (*AFD*, *Deterministic Finite Automaton* ou *DFA*) (Rabin et Scott, 1959). La théorie des automates finis a été introduite par des neurologues (McCulloch et Pitts, 1943) puis généralisée à l'informatique (Mealy, 1955; Moore, 1956). Les *AFN* et *AFD* sont des automates finis (*Finite-State Machine* ou *FSM*) qui permettent de décrire les jetons du langage source (un *langage rationnel* (Kleene, 1951)) au moyen d'expressions régulières. Un exemple d'une telle représentation est présenté dans le listing 2.2 (section 2.3). La différence majeure entre un *AFN* et un *AFD* est que dans un *AFD*, pour une suite donnée de caractères, pour chaque état, il n'y a qu'une seule transition possible (donc déterministe). Par contre, dans un *AFN*, pour certains états, plus d'une transition est possible (donc non déterministe). La conversion d'une expression régulière (*ER*) à un automate fini déterministe se fait dans l'ordre $ER \rightarrow AFN \rightarrow AFD$.

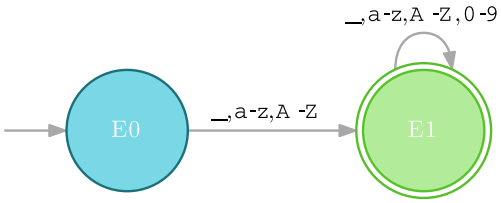


Figure 2.2 Exemple d'un *AFD* pour l'expression régulière d'un identifiant.

Par exemple, la figure 2.2 présente un *AFD* de l'expression régulière «`[_a-zA-Z][_a-zA-Z0-9]*`» représentant un identifiant de variable ne pouvant pas commencer par un chiffre (ex. `ma_variable_no1`).

Pour effectuer l'étape suivante d'analyse syntaxique, il existe différents types d'**analyseurs** (*parsers*) notamment *LR(k)* (Knuth, 1965) (*Left-to-right, Rightmost derivation in reverse*), *LL(k)* (*Left-to-right, Leftmost derivation*) (Stearns et Lewis, 1969; Rosenkrantz et Stearns, 1970), *LALR(k)* (*LookAhead LR*) (DeRemer, 1969), *GLR* (*Generalized LR*) (Tomita, 1984), *IELR(k)* (*Inadequacy Elimination LR)*) (Denny et Malloy, 2010) et *ALL(k)* (*Adaptive LL*) (Parr *et al.*, 2014).

La valeur de k définit le nombre de jetons que l'analyseur peut examiner pour déterminer la prochaine action (*lookahead tokens*). Les analyseurs avec $k = 1$ (ex. *LR(1)*) sont plus faciles à implémenter et plus rapides.

Une valeur de $k = *$ (ex. *LL(*)*) signifie qu'un nombre illimité de jetons est analysé, et ce, au détriment du temps d'exécution.

Les analyseurs de type *LL* sont plus simples à implémenter, mais moins performants — ils ont néanmoins été populaires par le passé, puisqu'ils permettent de définir facilement une grammaire pouvant être analysée par un analyseur de type descendant récursif (*recursive descent parser*) (Terry et Terry, 2005). Ceux de type *LR*, plus difficiles à implémenter mais plus puissants, peuvent traiter la majorité des grammaires (Tomassetti, 2017). Leur complexité temporelle est basse et normalement linéaire — ou $\mathcal{O}(n^3)$ dans les pires scénarios. L'analyseur *GLR* a l'avantage de pouvoir performer l'analyse en parallèle (*multithreading*) — bien que cela puisse engendrer des conflits (Vick, 2009). L'analyseur *ALL* combine la facilité d'implémentation du type *LL* avec la puissance de *GLR*, ce qui permet de supporter les grammaires non contextuelles sans récursivité gauche (*non-left-recursive context-free grammar*). Sa complexité est en théorie $\mathcal{O}(n^4)$, mais elle est linéaire en pratique avec la majorité des grammaires et surpasse les performances de *GLR* (Parr *et al.*, 2014).

Une distinction importante entre les analyseurs de types *LL* et *LR* est la façon dont l'arbre syntaxique résultant est construit. Le type *LL* le construit de la racine aux feuilles (*top-down*), alors que le type *LR* le fait en sens inverse (*bottom-up*). De plus, le type *LL* ne supporte pas la récursivité gauche (directe, indirecte ou invisible) dans la description du langage (dans une production) — ce qui limite l'expressivité des grammaires. Cependant, comme nous le verrons (section 2.3), le type *ALL(*)* permet malgré tout la récursivité gauche directe. Le listing 2.1 (page 21) présente un exemple des différents types de récursivités gauches.

Une grammaire algébrique (aussi appelée grammaire non contextuelle ou *Context-Free Grammar* (*CFG*)) est utilisée pour décrire la plupart des langages de programmation (Norvell, 2002). La forme de Backus-Naur (*Backus–Naur Form* ou *BNF*) a été introduite lors de la conception d'*Algol 58*, puis raffinée pour *Algol 60* (Backus, 1959; Backus *et al.*, 1963). C'est la forme la plus utilisée pour décrire les langages de programmation. Cependant, les compilateurs de compilateur présentés dans la section 2.2 utilisent plutôt la forme de Backus–Naur étendue (*Extended Backus–Naur Form* ou *EBNF*) (Wirth, 1977) pour décrire les grammaires (Gagnon, 1998; Corbett et al., 2021; Parr, 2013). La forme *EBNF* est une *BNF* à laquelle ont été ajoutés des opérateurs d'expressions régulières tels que `()`, `*`, `?` et `+`. Sans être plus « *puissante* »[23], une *EBNF* permet, avec ces extensions, d'exprimer une grammaire de façon plus concise.

[23] Donc toute grammaire pouvant être représentée par une *EBNF* peut être représentée par une *BNF* sans les opérateurs additionnels.

2.2 Choix du compilateur de compilateur

SableCC (Gagnon, 1998) est un compilateur de compilateur pour le langage *Java* qui utilise un analyseur de type *LALR(1)*. Contrairement à plusieurs autres, il a l'avantage de générer le squelette du code source pour visiter l'arbre syntaxique qu'il produit. Ceci est utile pour la mise en œuvre d'un interpréteur et permet une économie de temps au niveau de l'implémentation. Une version supportant le `C++` est disponible[24]. Cependant, selon É. Gagnon, elle n'est pas à jour et est quelque peu problématique. Bien que *SableCC* était le seul outil dont nous avions une bonne connaissance, puisque nous l'avons utilisé dans nos prototypes *Java* mentionnés en *introduction* et nos cours de compilation (baccalauréat et maîtrise), son support limité du `C++` a fait en sorte que nous l'avons mis de côté.

Les générateurs d'analyseurs lexicaux *Flex++* (version de *Flex* pour le `C++`, basée sur son prédécesseur *Lex*) ou *Re/Flex* en combinaison avec le générateur d'analyseurs syntaxiques *GNU Bison* (générant des analyseurs syntaxiques de types *LALR(1)*, *LR(1)*, *IELR(1)* et *GLR*) ont aussi été considérés. *GNU Bison* est compatible avec *Yacc* (*Yet Another Compiler-Compiler*) et supporte le `C++`. Cependant, à l'exception de *Re/Flex*, ces outils sont anciens et dépassés (Tomassetti, 2020). Leur support du `C++` est limité et ils sont compliqués à utiliser — l'implémentation du visiteur de l'arbre syntaxique est sous la responsabilité du programmeur. De plus, ils ne sont pas compatibles avec les *caractères Unicode*.

Notre choix s'est finalement arrêté sur *ANTLR* (*ANother Tool for Language Recognition*) un compilateur de compilateur créé comme successeur de *PCCTS* (*Purdue Compiler Construction Tool Set*) (Parr, 1996; Parr, 2013). Son analyseur syntaxique est de type *LL(*)* (Parr et Fisher, 2011) — *ALL(*)* depuis *ANTLR 4*. À l'instar de *SableCC*, il peut générer le code source pour visiter l'arbre syntaxique pour la mise en œuvre de l'interpréteur. De plus, depuis cette version, il supporte la génération du code de l'analyseur dans 20 langages, dont *Java*, *Python*, `C` et `C++`.

[24] `http://www.mare.ee/indrek/sablecc/` (*archive*)

```
exprA : NOMBRE
      | NOMBRE? exprA   /* Recursivité gauche invisible  ANTLR 3/4 : ERREUR */
      | exprB           /* Recursivité gauche indirecte  ANTLR 3/4 : ERREUR */
      | exprA '+' exprA /* Recursivité gauche directe     ANTLR 3   : ERREUR
                                                          ANTLR 4   : OK     */
;

exprB : exprA '*' NOMBRE;
```

Listing 2.1 Exemple d'une production avec récursivités gauches.

ANTLR 4 utilise un analyseur *ALL(*)* permettant la récursivité gauche directe (par réécriture automatique de la grammaire (Parr *et al.*, 2014)). Il permet donc de décrire des grammaires plus complexes que le pouvaient *ANTLR 3* et ses versions précédentes, lesquelles utilisaient un analyseur *LL(*)*.

L'exemple[25] dans le listing 2.1 illustre les trois formes de récursivité gauche : la récursivité gauche directe (*non-left direct recursion*) serait acceptée par *ANTLR 4*[26], alors que les récursivités gauches indirectes et invisibles (*non-left indirect and hidden recursions*) ne le seraient pas.

ANTLR supporte différents types de **prédicats sémantiques** (*semantic predicates*) (Parr, 2021, sect. *Semantic Predicates*) qui augmentent l'expressivité. Les prédicats sémantiques sont des expressions booléennes pouvant être ajoutées dans la description d'une grammaire pour analyser les jetons qui suivent (*lookahead*) et sélectionner l'action à effectuer en conséquence. Ils permettent donc d'élargir l'ensemble de la grammaire non ambigüe au-delà de l'ensemble supporté par *ALL(*)* et d'ainsi définir des langages plus expressifs.

Avec *ANTLR 4*, il est donc théoriquement possible de décrire toute grammaire pouvant être décrite par un analyseur de type *LALR(1)*. Aussi, sa licence *BSD*, son support du `C++` et des *caractères Unicode* sont trois autres avantages non négligeables.

Un exemple de description d'une grammaire pour une « *calculatrice élémentaire* » avec *ANTLR 4* se trouve

[25] Inspiré de `https://medium.com/@eichenroth/the-importance-of-left-recursion-in-grammars-608f849447f6` (*archive*)

[26] `https://github.com/antlr/antlr4/blob/0eb38a/doc/faq/general.md`

dans le listing 2.2[27] et la *EBNF*[28], sous forme de diagramme syntaxique (*railroad diagram*), correspondant à la production `expression` se trouve à la figure 2.3.

```
grammar Calculatrice;

/* Jetons */

PAR_GAUCHE: '(';
PAR_DROITE: ')';
MUL:        '*';
DIV:        '/';
PLUS:       '+';
MOINS:      '-';
NOMBRE:     [0-9]+;
ESPACES:    [ \r\n\t]+ -> skip;

entree : expression;

/* Production expression (liste d'alternatives) */
expression
   : NOMBRE                                                  # Nombre
   | MOINS expr_d=expression                                 # Negation
   | PAR_GAUCHE expr=expression PAR_DROITE                   # Parentheses
   | expr_g=expression operateur=(MUL|DIV)    expr_d=expression # MulOuDiv
   | expr_g=expression operateur=(PLUS|MOINS) expr_d=expression # AddOuSous
   ;
```

Listing 2.2 Exemple de la grammaire d'une « *calculatrice élémentaire* » avec *ANTLR 4*.

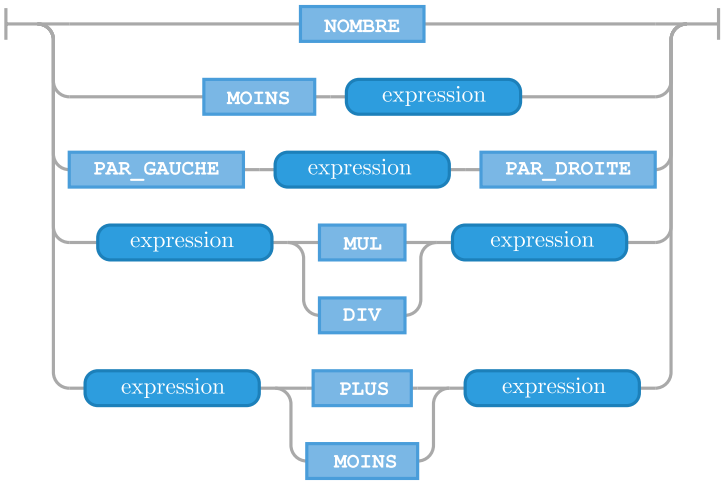


Figure 2.3 Diagramme syntaxique de la production `expression`.

[27] Cet exemple est basé sur `https://github.com/arothuis/antlr4-calculator/blob/d7b6405b87/src/main/antlr4/nl/arothuis/antlr4calculator/core/parser/Calculator.g4`

[28] Dans cet exemple, une simple *BNF* serait suffisante, puisqu'aucune extension *EBNF* n'est utilisée. Par contre, l'exemple dans le listing 2.1, bien qu'illégal, nécessite une *EBNF*, puisqu'il utilise l'opérateur `?` dans la récursivité gauche invisible.

```
Résultat: 1+3/256*(52--65) = 2.37109375

        +-----+     +-------+
        |  1  | --- |   +   |
        +-----+     +-------+
                      |
+---+     +-----+     +-------+
| 3 | --- |  /  | --- |   *   |
+---+     +-----+     +-------+
            |           |
        +-----+     +-------+     +----+
        | 256 |     |   -   | --- | 52 |
        +-----+     +-------+     +----+
                      |
                    +-------+
                    | nég.  |
                    +-------+
                      |
                    +-------+
                    |  65   |
                    +-------+
```

Capture 2.1 Résultat de l'exemple du listing 2.2 pour « `1 + 3 / 256 * (52 - -65)` ».

Dans cet exemple, la priorité des opérations de multiplication et division sur celles d'addition et soustraction est implicitement définie par l'ordre des déclarations des alternatives (Parr, 2013, sect. 5.4). On peut constater, en observant le résultat du calcul dans la capture 2.1, qu'il est *exact* et que les priorités des opérations mathématiques sont respectées. Dans la majorité des compilateurs de compilateurs[29], la définition des priorités est plus ardue et nécessite la déclaration de productions intermédiaires pour résoudre les ambiguïtés. Dans *ANTLR 4*, les ambiguïtés sont résolues en retenant la première alternative déclarée — ceci permet de déclarer implicitement les priorités des opérateurs. Notons cependant que ceci ne s'applique qu'aux opérateurs associatifs de gauche à droite et non aux opérateurs comme l'exponentiation, associatif de droite à gauche.

Les versions antérieures à *ANTLR 4* permettaient de générer automatiquement l'arbre syntaxique *AST*. Cette fonctionnalité a néanmoins été retirée d'*ANTLR 4* et ce dernier génère plutôt un arbre d'analyse (*parse tree*, *Concrete Syntax Tree* ou *CST*) (Stalla, 2020) — selon le concepteur d'*ANTLR*, la génération de l'arbre *AST* n'est pas vraiment utile dans les cas d'utilisation les plus fréquents d'*ANTLR*[30], puisqu'*ANTLR* génère le code source pour visiter automatiquement l'arbre *CST*. Contrairement à un arbre *AST* qui exclut toute

[29] *SableCC 4 (beta)* permet lui aussi de définir les priorités plus simplement en introduisant des mots clés pour les spécifier.

[30] `https://theantlrguy.atlassian.net/wiki/spaces/~admin/blog/2012/12/08/524353/Tree+rewriting+in+ANTLR+v4` (*archive*)

information superflue, un arbre *CST* représente de manière précise la dérivation du texte selon la définition de la grammaire. Cependant, l'ensemble des nœuds de l'arbre *AST* étant souvent un sous-ensemble de ceux de l'arbre *CST* (figure 2.4), il peut généralement être obtenu à partir de ce dernier en surchargeant (*overriding*) certaines méthodes de la classe générée par *ANTLR* pour visiter l'arbre *CST* (Parr, 2020, qst. *What if I need ASTs not parse trees for a compiler, for example?*) — ce qui est le cas pour l'exemple de la « *calculatrice élémentaire* » et a permis de générer l'arbre *AST* de la figure 2.4.

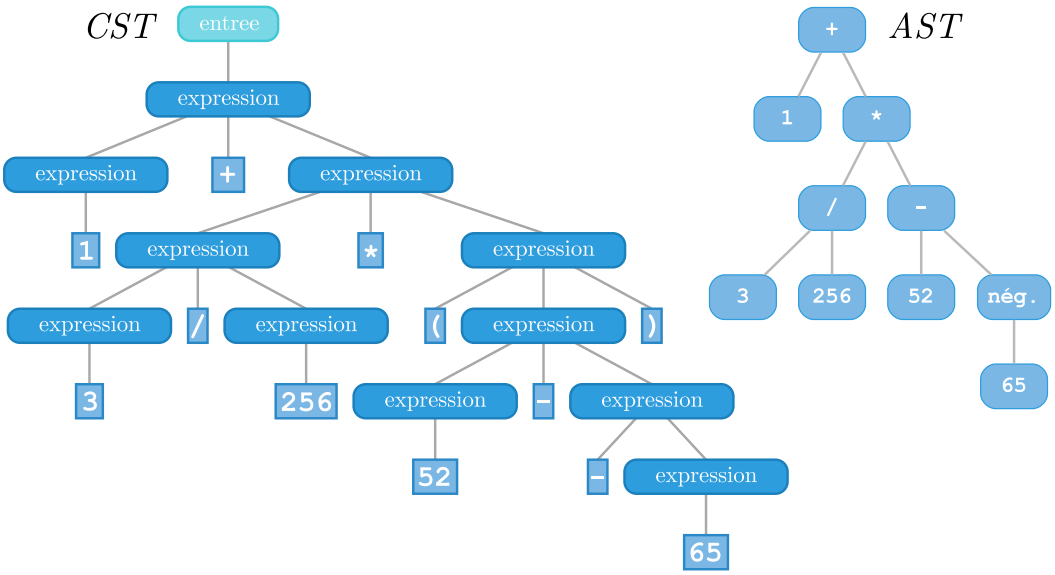


Figure 2.4 Arbres *CST* et *AST* de l'exemple du listing 2.2 pour « `1 + 3 / 256 * (52 - -65)` ».

CHAPITRE III
GESTION DE LA MÉMOIRE EN C++ ET RAMASSE-MIETTES BOEHM ET AL.

Le langage que nous comptons développer pour le système de calcul formel dont nous présentons les fondations sera un langage dynamique. Lors de la mise en œuvre d'un prototype *Java/SableCC* (voir l'*introduction*), le problème de la gestion mémoire ne se posait pas puisqu'il nous suffisait d'utiliser le *ramasse-miettes* fourni par la machine virtuelle de *Java*. Le choix d'utiliser le C++ pour notre projet de maîtrise nous complique toutefois la tâche : dans ce langage, la libération de la mémoire préalablement réservée doit se faire de façon explicite par le programmeur — à moins d'utiliser une bibliothèque spécialisée.

Trois options s'offraient à nous : une gestion explicite de l'allocation mémoire au moyen de pointeurs traditionnels, l'utilisation de pointeurs C++ intelligents (*smart pointers*) ou l'utilisation d'un ramasse-miettes au moyen d'une bibliothèque externe.

L'utilisation explicite de pointeurs traditionnels, malgré certains avantages — notamment, surcoûts moindres qu'une bibliothèque externe — nous semblait dangereuse et risquait d'occasionner des problèmes de fuites mémoire (*memory leaks*) difficiles à isoler, qui auraient pu entraver le bon déroulement du projet.

Les pointeurs intelligents, quant à eux, éliminent les problèmes de fuites mémoire sans réduire les performances. Cependant, bien qu'il soit légitime de les considérer comme une forme primitive de ramasse-miettes[31], ils ne sont pas conçus pour remplacer un ramasse-miettes dédié à une telle tâche. Un problème avec leur utilisation est qu'ils sont bloquants (*stop-the-world*), c'est-à-dire qu'ils auront tendance à bloquer le flux d'exécution naturel du logiciel (par exemple, au retour d'une fonction ou à la sortie d'une boucle) pour libérer la mémoire inutilisée. Pire encore, ce comportement peut provoquer un effet de cascade dans une situation d'imbrication de portées mémoire (*memory scopes*), c'est-à-dire, un espace mémoire qui n'est accessible qu'à l'intérieur d'une fonction, d'une boucle, etc., et qui est créé à l'entrée et détruit à la sortie d'une telle région de code.

Un ramasse-miettes est une fonctionnalité logicielle qui évite au programmeur de devoir lui-même gérer la mémoire en prenant en charge cette tâche de manière implicite en arrière-plan. Cette idée fut proposée initialement pour le langage *LISP* (McCarthy, 1960).

[31] https://www.bitesizedengineering.com/p/raii-and-smart-pointers-smarter-way (*archive*)

Lors de nos recherches, un article (Stover, 2003) sur le ramasse-miettes *Boehm* (*Boehm-Demers-Weiser garbage collector*) (Boehm et Demers, 1988) attira notre attention. Il s'agit d'un ramasse-miettes `C++` *conservateur* avec collection incrémentielle et générationnelle (*incremental and generational collection*) utilisant un *algorithme marquant et nettoyant* (*mark-and-sweep*). Il est utilisé dans de nombreux projets d'envergure tels que *Mono*[32], une ré-implémentation de *.NET* sous licence libre, le moteur de jeu *Unity*[33], le compilateur *Python Codon*[34] et *Mozilla* (comme détecteur de fuites mémoire[35]). Après analyse, il nous a semblé que ce projet était mature et digne d'intérêt, et nous avons décidé de l'utiliser.

Comme nous le verrons (section 3.2), le ramasse-miettes *Boehm*, lorsqu'il n'est pas utilisé en mode *incrémentiel*, est lui aussi de type *stop-the-world*. Par contre, son utilisation évite les collectes en cascades, puisque les collectes ne sont effectuées qu'à des intervalles déterminés et non à chaque clôture de portée active. De plus, il permet, au besoin, de désactiver temporairement les collectes à des moments stratégiques dans un objectif d'optimisation. D'où l'intérêt d'utiliser un véritable ramasse-miettes, plutôt qu'une approche basée sur les pointeurs intelligents.

Remarque: *Pour alléger le texte, nous utiliserons dorénavant l'abréviation usuelle GC (garbage collector), plutôt que ramasse-miettes.*

3.1 Fonctionnement et types de ramasses-miettes

Il existe différentes combinaisons possibles de concepts et d'algorithmes pouvant être utilisés lors de la conception d'un *GC*, parfois complémentaires, avec leurs avantages et inconvénients. Ici, nous ne nous concentrerons que sur les concepts et algorithmes qui ont été étudiés dans le cadre de notre projet en fonction de la bibliothèque choisie. Nous mentionnerons d'autres approches, sans les approfondir, uniquement pour préciser certains concepts.

Un *GC* peut être de type *précis* (*precise*, *exact* ou *accurate collector*) ou *conservateur* (voire partiellement

[32]Depuis *Mono* 3.1.1, un ramasse-miettes de type générationel simple est utilisé (*SGen*, un *GC précis*) par défaut, puisque dans certaines applications le *GC Boehm* peut parfois engendrer des fuites mémoire : `https://www.mono-project.com/docs/advanced/garbage-collector/sgen/` (*archive*).

[33]Puisque *Unity* utilise le langage *Mono*, il utilise aussi le *GC Boehm* : `https://docs.unity3d.com/Manual/performance-incremental-garbage-collection.html` (*archive*).

[34]Un compilateur de code *Python* en code machine : `https://docs.exaloop.io/codon/general/faq` (*archive*)

[35]`https://www.hboehm.info/gc/#users` (*archive*)

conservateur, *conservative* ou *partly/semi-conservative collector*). Un collecteur dit précis permet de détecter de façon systématique et sans ambiguïté tous les pointeurs d'allocations mémoire. Il a donc l'avantage d'éviter toutes possibilités de fuites mémoire. Inversement, un collecteur dit conservateur détecte ces pointeurs en analysant la mémoire *racine* (*root memory*), soit la pile (*stack*) ou les tas (*heaps*), pour y détecter des motifs qui s'apparentent à des adresses mémoire.

Contrairement à un *GC* de type précis, avec un *GC* conservateur, des faux négatifs (c.-à-d. que le *GC* ne détecte pas une allocation mémoire admissible à une collecte) sont possibles, ce qui fait que des fuites mémoire peuvent survenir. Il peut donc être délicat d'utiliser un tel *GC* dans certaines applications — par exemple, un serveur qui s'exécute pendant des mois. En pratique, de telles fuites mémoire sont rares, et il est possible de mettre en place certaines mesures pour les détecter. De plus, bien qu'ils soient théoriquement possibles, les faux positifs (c.-à-d. que le *GC* tente de libérer un espace mémoire non alloué) sont moins problématiques à l'ère des processeurs 64 bits *versus* 32 bits : avec un adressage 64 bits, une donnée quelconque a peu de chances d'être confondue avec un pointeur valide, puisque l'adressage mémoire valide ne représente qu'une infime partie de l'intervalle de valeurs possibles (Boehm, 2023).

Comme nous le verrons, malgré cet inconvénient, le *GC* utilisé dans le cadre de notre projet est de type conservateur, l'avantage majeur étant qu'il s'intègre facilement dans un code n'ayant pas été initialement conçu pour supporter la collecte des miettes (par exemple, un programme `C` ou `C++`)[36]. Ceci s'avère plus complexe dans le cas d'un *GC* de type précis.

Les collectes peuvent être de type *générationnel* ou *incrémentiel*. Une collecte dite générationnelle présuppose qu'une allocation mémoire récente risque d'être libérée plus rapidement qu'une allocation plus ancienne. Afin d'éviter de bloquer le flux d'exécution pendant une « *longue* » période (*stop-the-world*), la collecte générationnelle ne se fera que partiellement, en commençant par les dernières allocations détectées comme étant inaccessibles. Une collecte dite incrémentielle, tout comme une collecte générationnelle, ne se fera pas de façon complète, là aussi pour éviter de bloquer le flux d'exécution. Par contre, cette collecte ne se fera pas forcément selon l'ancienneté des allocations, mais sera plutôt répartie en plusieurs petites collectes ne nécessitant pas l'inspection de la totalité des allocations dans le tas. C'est donc dire que toute collecte générationnelle est incrémentielle, mais que l'inverse ne l'est pas forcément (Hungerecker et Schmitz, 2015).

[36] `https://www.iecc.com/gclist/GC-algorithms.html#Conservative%20collection` (*archive*)

Différents algorithmes peuvent être utilisés pour la détection des pointeurs admissibles à une collecte. Pensons aux algorithmes *mark-and-sweep*, *mark-and-don't-sweep* et *stop-and-copy*. L'algorithme *mark-and-sweep* est le plus ancien et le plus naïf. Ce dernier fut présenté par McCarthy dans son papier introduisant le *GC* du langage *LISP* (McCarthy, 1960). En gros, il consiste à effectuer la collecte en deux phases : une première dite de marquage (*mark*), où les objets accessibles sont marqués, suivie d'une deuxième dite de balayage (*sweep*), où les objets non marqués sont considérés comme non accessibles et donc admissibles à une collecte[37]. Son désavantage majeur est qu'il est de type *stop-the-world* et requiert un temps proportionnel à la taille de la mémoire (Zorn, 1990). De plus, il aura tendance à fragmenter la mémoire lors des libérations.

L'algorithme *stop-and-copy* (Cheney, 1970) a la cote depuis quelques années, puisque son temps d'exécution est indépendant de la taille de la mémoire et qu'il évite la fragmentation. Par contre, il est difficile à mettre en œuvre dans les langages tels que `C`/`C++` et dans un *GC* générationnel puisque, dans ce cas, les pointeurs ne peuvent être identifiés avec certitude et que l'algorithme *stop-and-copy* doit déplacer les objets — ce qui peut causer des problèmes[38]. De plus, il consomme deux fois plus de mémoire que les autres algorithmes.

L'algorithme *mark-and-don't-sweep*, quant à lui, est un compromis entre les deux précédents : il ne consomme pas autant de mémoire que l'algorithme *stop-and-copy*. mais il requiert une coopération entre l'allocateur et le collecteur, lequel peut entraver les collectes à certains moments. Ses avantages par rapport aux autres algorithmes sont donc mitigés.

3.2 Détails techniques et utilisation du ramasse-miettes *Boehm*

À l'origine, le *GC Boehm* n'était destiné qu'à être utilisé avec le langage `C` et non avec le `C++`. Il ne s'exécutait donc que lors d'appels aux opérations `malloc` et `free`. Cependant, une interface `C++` (*Ellis-Hull* `C++` *interface*, fichier *`gc_cpp.h`*) est maintenant disponible pour intercepter aussi les appels à `new` et `delete`, ce qui permet au *GC* de gérer les allocations mémoire dynamiques `C++`.

Il faut souligner qu'en `C++` certaines précautions doivent êtres prises par le programmeur pour s'assurer du bon fonctionnement du *GC*, ce qui fait que son utilisation en `C++` est plus complexe qu'en `C`, ce que nous

[37] `https://www.linkedin.com/pulse/mark-sweep-garbage-collection-algorithm-arpit-bhayani` (*archive*)

[38] `https://www.hboehm.info/gc/complexity.html` (*archive*)

avons pu constater, comme nous le décrirons ultérieurement.

> *The `C++` interface is implemented as a thin layer on the `C` interface. Unfortunately, this thin layer appears to be very sensitive to variations in `C++` implementations, particularly since it tries to replace the global `::new` operator, something that appears to not be well-standardized.* [...] *Usage of the collector from `C++` is also complicated by the fact that there are many "standard" ways to allocate memory in `C++`. The default `::new` operator, default `malloc`, and default STL allocators allocate memory that is not garbage collected, and is not normally "traced" by the collector. This means that any pointers in memory allocated by these default allocators will not be seen by the collector.*
>
> Documentation du *GC Boehm*

Un avantage du *GC Boehm* est qu'il est de type conservateur, donc plus approprié pour un projet `C++` qu'un *GC* précis, ces derniers ayant tendance à consommer beaucoup de mémoire lorsqu'utilisés dans des langages tels que `C++` et *Rust* (playX, 2020).

Par défaut, le *GC Boehm* utilise le mode générationnel dès son initialisation (`GC_init` ou `GC_INIT`[39]). Il est possible d'activer le mode incrémentiel au moyen de la fonction `GC_enable_incremental`, auquel cas il est conseillé d'activer ce mode dès le démarrage — avant ou tout juste après l'initialisation. Par contre, une fois ce mode en fonction, il ne peut pas être désactivé.

La bibliothèque du *GC Boehm* peut aussi être utilisée comme détecteur de fuites mémoire en ajoutant l'option `-DFIND_LEAK` (incompatible avec le mode incrémentiel) lors de sa compilation. Par exemple, la communauté de la Fondation *Mozilla* utilise cette fonctionnalité pour détecter les problèmes de fuites mémoire dans le navigateur *Firefox*[40]. Cette fonctionnalité n'a cependant pas été utilisée dans le cadre de notre projet, puisque des outils spécialisés nous semblaient plus appropriés (voir section 5.5).

L'interface `C++` du *GC Boehm* permet de spécifier, et ce de différentes manières, quelles allocations dynamiques et objets sont prises en charge par le *GC*. Avant tout, il est important de savoir que toutes les allocations standards faites par l'opérateur `new` (ou la fonction `malloc` du `C`) seront *non gérées par défaut*[41]. Pour qu'une allocation avec un `new` soit gérée, il est nécessaire d'ajouter un argument

[39] `GC_INIT` est une macro `C++` qui, en plus d'appeler `GC_init`, appelle plusieurs autres fonctions pour configurer le *GC*. C'est la méthode la plus sûre pour initialiser le *GC* lorsqu'utilisé dans un logiciel multiplateforme.

[40] `https://www.hboehm.info/gc/#users` (*archive*)

[41] Sauf si la bibliothèque du *GC* est compilée avec l'option `-DREDIRECT_MALLOC=GC_malloc` pour imposer par défaut la gestion de toutes les allocations mémoire, ce qui n'est généralement pas souhaitable.

supplémentaire (*placement* `new`) comme suit : `new (GC) classe`. Si un appel à `malloc` est requis, c'est plutôt `GC_malloc`[42] qu'il faut utiliser.

De plus, il est possible de déclarer une classe comme étant implicitement gérée par le *GC* lors de son instanciation en la faisant hériter de la classe `gc`. Dans ce cas, aucun argument supplémentaire n'est nécessaire lors d'un `new`. Cependant, le fait d'hériter de la classe `gc` ne fait pas que le destructeur de l'objet soit appelé lors de sa libération. C'est plutôt la classe `gc_cleanup` qu'il faut utiliser. Cette subtilité n'est pas mentionnée dans la documentation — la lecture du code source et des commentaires a été nécessaire pour comprendre ce « *détail* » important.

Il est bien sûr possible de libérer explicitement une allocation au moyen de `delete` (auquel cas, l'argument `(GC)` n'est pas nécessaire) ou `GC_free` (qui n'appelle pas le destructeur). À tout moment il est possible de déclencher une collecte totale ou partielle au moyen de `GC_gcollect` et `GC_collect_a_little`. Toutefois, dans la majorité des cas, il est préférable de laisser le *GC* gérer lui-même de façon implicite les occurrences des collectes (selon ses configurations).

Plusieurs *GC* offrent des fonctions dites de finalisation pour exécuter certaines tâches lorsqu'un objet est collecté. Une fonction de finalisation ne devrait pas remplacer un destructeur — ce sont deux notions différentes. Bien que le *GC Boehm* permet de les utiliser (via `GC_register_finalizer`), sa documentation[43] souligne qu'un logiciel bien programmé devrait rarement en avoir besoin. Dans le cadre de notre projet, les destructeurs `C++` ont été suffisants.

Comme mentionné précédemment (extrait *documentation du GC Boehm*, p. 29), il faut être prudent avec certaines structures `C++`, notamment les structures de données de la *STL* (*Standard Template Library*). Dans leur cas, il est primordial de les instancier avec la classe `gc_allocator`, sinon le *GC* n'a pas conscience des allocations mémoire que ces structures de données font à l'interne. Ce point sera expliqué plus en détail à la section 6.10, car il a été capital dans le cadre de notre projet.

[42] Ou une de ses variantes `GC_realloc`, `GC_malloc_atomic` ou `GC_malloc_uncollectable`

[43] `https://www.hboehm.info/gc/finalization.html` (*archive*)

CHAPITRE IV
BIBLIOTHÈQUES POUR ARITHMÉTIQUE MULTIPRÉCISION

Dans ce chapitre, nous présentons tout d'abord les types numériques de base « *standards* » (norme *IEEE 754*) et nous expliquons pourquoi, dans un système de calcul formel, l'utilisation d'une représentation décimale (base 10) est préférable à l'utilisation d'une réprésentation binaire (base 2). Nous présentons ensuite quelques bibliothèques pour arithmétique à précision arbitraire, avec leurs avantages et inconvénients.

4.1 Arithmétique à précision arbitraire en bases 2 et 10

Comme nous l'avons mentionné au chapitre I, un système de calcul formel est un logiciel qui n'effectue pas forcément les calculs à chaque opération arithmétique. Il procède plutôt par calcul symbolique, c'est-à-dire qu'il simplifie les expressions mathématiques dans leur forme la plus simple et les manipule tout en garantissant qu'il n'y ait aucune perte en précision[44]. Donc, les calculs ne sont effectués que pour manipuler les expressions mathématiques, ainsi que lorsqu'un résultat final est demandé. Toutefois, il est important qu'il soit en mesure de retourner un résultat à une précision arbitraire, lorsque requis.

Les types fondamentaux `float`, `double` et `long double` du `C++`, basés sur la norme *IEEE 754-1985* (*IEEE*, 1985), sont respectivement de taille 32, 64 et 128 bits[45]. Cette norme a été établie en 1985 par l'*Institute of Electrical and Electronics Engineers* (*IEEE*) pour standardiser l'implémentation et l'utilisation des types à virgule flottante. Son but était, entre autres, de régler certains problèmes de portabilité entre différentes architectures d'ordinateurs, appareils électroniques, compilateurs, etc. La norme spécifie un format de stockage en mémoire bien précis composé d'un bit de signe, d'un certain nombre de bits pour l'exposant et d'une série de bits pour la mantisse, normalisée et contenant les chiffres significatifs. Par exemple, un `double` 64 bits est composé d'un bit de signe, de 11 bits pour l'exposant et de 52 bits pour la mantisse. Le nombre de bits de la mantisse définit la précision maximale pouvant être atteinte. Par contre, bien que 52 bits soient explicitement stockés, la précision effective est toujours obtenue en y ajoutant un bit — donc dans ce cas-ci, 53 bits. Sans entrer dans les détails, cette norme impose aussi certaines règles à suivre dans le cas de valeurs spéciales (infini, zéro, *NaN* (*Not a Number*), dénormalisé, etc.).

[44] Par exemple, il simplifie $\frac{2\pi+2+6}{4}$ en $\frac{\pi}{2}+2$, mais ne calcule pas le résultat réel pour éviter la perte de précision — un logiciel de calcul traditionnel calculerait plutôt $\frac{2\pi+2+6}{4} = 3{,}570\,796\,326\,794\,896\,62$.

[45] Ces tailles peuvent toutefois varier d'une architecture ou d'un compilateur à l'autre.

Les précisions de base définies par la norme *IEEE 754-1985* ne sont pas adéquates pour obtenir les précisions qu'un système de calcul formel devrait permettre, à savoir, des précisions pouvant comporter un nombre de décimales limité seulement par la taille de la mémoire. Avec la mantisse d'un `double` 64 bits (53 bits effectifs), on ne peut représenter que 17 chiffres (décimaux) significatifs[46], car $\lceil log_{10}(2^{53}) + 1 \rceil = 17$ (Matula, 1968). Et avec la même formule, on obtient des précisions de 9 chiffres pour un `float` 32 bits (24 bits effectifs) et de 36 pour un `long double` 128 bits (113 bits effectifs).

Ces types de base sont stockés en mémoire sous forme binaire (base 2). Cette représentation est utilisée, car elle simplifie la conception matérielle des éléments logiques effectuant les calculs. Cependant, elle ne peut pas exprimer de façon précise certaines valeurs décimales (base 10) (Goldberg, 1991). Par exemple, le nombre $0{,}1$ ne peut pas être stocké en mémoire de façon exacte[47]. Bien que c'est ce nombre qui est affiché à l'écran par défaut dans certains langages (voir plus bas), en mémoire (dans le cas d'un `float`) c'est plutôt la valeur $0{,}100\,000\,001$ qui est stockée.

nombre	précisions		
	`float` (32 bits)	`double` (64 bits)	`long double` (128 bits)
0,1	0,100 000 001	0,100 000 000 000 000 01	0,100 000 000 000 000 000 001 355 252 715 606 881
0,2	0,200 000 003	0,200 000 000 000 000 01	0,200 000 000 000 000 000 002 710 505 431 213 761
0,3	0,300 000 012	0,299 999 999 999 999 99	0,300 000 000 000 000 000 010 842 021 724 855 044
0,4	0,400 000 006	0,400 000 000 000 000 02	0,400 000 000 000 000 000 005 421 010 862 427 522
0,5	0,5	0,5	0,5
0,6	0,600 000 024	0,599 999 999 999 999 98	0,600 000 000 000 000 000 021 684 043 449 710 089
0,7	0,699 999 988	0,699 999 999 999 999 96	0,699 999 999 999 999 999 989 157 978 275 144 956
0,8	0,800 000 012	0,800 000 000 000 000 04	0,800 000 000 000 000 000 010 842 021 724 855 044
0,9	0,899 999 976	0,900 000 000 000 000 02	0,899 999 999 999 999 999 978 315 956 550 289 911

Tableau 4.1 Valeurs en mémoire pour différents nombres décimaux à diverses précisions.

Le tableau 4.1 présente les valeurs effectivement stockées en mémoire de différents nombres à diverses

[46] Cette valeur n'est que théorique ; en pratique, elle est plutôt de 15 à 17 chiffres (Kahan, 1996) ; (*Wikipedia*).

[47] Pour qu'un nombre réel puisse être converti de façon exacte en base 2, il doit pouvoir être représenté sous forme d'une fraction ayant comme dénominateur une puissance de 2. L'ensemble de tels nombres peut être défini comme suit : $\left\{ \frac{\alpha}{\beta} \mid \alpha \in \mathbb{Z}, \beta \in \mathbb{N}^+, \exists \gamma \in \mathbb{N} : \beta = 2^\gamma \right\}$. Ce n'est pas le cas de $0{,}1 = \frac{1}{10}$.

précisions[48]. Par exemple, dans le cas des `double` 64 bits, la valeur pour `0.3`, bien que très proche, est différente de la valeur stockée en mémoire (`0,29999999999999999`).

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main() {
    cout << "float       : " << setprecision(9)  << (0.1f + 0.2f) << endl;
    cout << "double      : " << setprecision(17) << (0.1d + 0.2d) << endl;
    cout << "long double : " << setprecision(36) << (0.1L + 0.2L) << endl;
}
```

Listing 4.1 Calculs avec différents types flottants `C++` de base.

```
float       : 0.300000012
double      : 0.30000000000000004
long double : 0.300000000000000000010842021724855044
```

Capture 4.1 Sortie du code `C++` du listing 4.1.

Par défaut, les fonctions de sortie de la plupart des langages de programmation, dont celles du `C++`, arrondissent la valeur d'un nombre à virgule flottante[49] avant de l'afficher. De ce fait, les résultats affichés à l'écran peuvent donc ne pas représenter la valeur qui est réellement stockée en mémoire. Pour modifier ce comportement avec `std::cout` en `C++`, il est possible d'utiliser la fonction `std::setprecision` pour afficher une valeur à une certaine précision. Le code `C++` dans le listing 4.1 calcule $0{,}1 + 0{,}2$ en utilisant les trois types flottants de base et affiche les résultats à diverses précisions[50], comme illustré dans la capture 4.1. Dans ce cas, la précision exigée ne s'applique qu'une fois le résultat calculé et n'affecte que l'affichage — la valeur en mémoire reste inchangée.

[48] Ces résultats peuvent être validés à partir de ces convertisseurs : `https://www.h-schmidt.net/FloatConverter/IEEE754.html` (*archive*), `https://www.ultimatesolver.com/en/ieee-754` (*archive*) ou `https://www.exploringbinary.com/floating-point-converter/` (*archive*)

[49] Le `std::cout` du `C++` utilise l'approximation décimale la plus courte dite *round-trip guarantee*.

[50] Les précisions utilisées ont été obtenues en utilisant la formule $\lceil log_{10}(2^{taille\ mantisse}) + 1 \rceil$.

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

Capture 4.2 Résultat du calcul dans *Python*.

Python, quant à lui, utilise par défaut des `double` 64 bits, donc il affiche (capture 4.2) le même résultat que dans l'exemple `C++` (capture 4.1). Le fait qu'en mémoire $0{,}1 + 0{,}2 \neq 0{,}3$ explique que `(0.1 + 0.2) == 0.3` retourne `False`. Ce comportement est d'autant plus problématique qu'une telle comparaison naïve peut introduire un bogue difficile à détecter.

type	base	précision	résultat
`float` 32 bits	2	9	95 680,945 3
`double` 64 bits	2	17	99 999,997 764 825 821
`gmp_float`	2	1000	99 999,999 999 999 999 999 9[...]999 999 999 168
`mpfr_float`	2	1000	99 999,999 999 999 999 999 9[...]999 989 723 523
`cpp_bin_float`	2	1000	99 999,999 999 999 999 999 9[...]999 989 723 523
`cpp_dec_float`	10	1000	100 000
mpdecimal	10	1000	100 000

Tableau 4.2 Résultats avec différents types pour le calcul de $\sum_{i=1}^{10\,000\,000} 0{,}01$ [51].

Quoique dans de nombreux cas, y compris lors de calculs scientifiques, de telles approximations soient suffisantes, elles engendrent inévitablement des accumulations d'erreurs. On n'a qu'à penser au cas d'une application effectuant des calculs monétaires. Un simple calcul d'addition de cents ($0{,}01$ \$) produira un résultat erroné après un certain nombre d'opérations. Ce problème est illustré, dans le tableau 4.2, en utilisant différents types[52] pour calculer la somme de dix millions de cents : avec des `float` 32 bits, on obtient un résultat de $95\,680{,}95$ \$ et non $100\,000{,}00$ \$.

Dans le tableau, on constate aussi que seuls les types en base 10 retournent le résultat exact[53]. Il est donc

[51] Les types `gmp_float`, `mpfr_float` et `cpp_dec_float` (bibliothèque *Boost.Multiprecision*) et *mpdecimal* seront introduits aux sections 4.3 et 4.4. Une capture complète du test « *Radix 2 vs 10 tests* » se trouve dans la capture B.2 (annexe B).

[52] Ce test est exécuté par notre logiciel (« *Radix 2 vs 10 tests* »). Il permet de comparer les résultats, consommation mémoire et performances des différentes bibliothèques et ses propres types `Numeric` (section B.2 en annexe B et chapitre VII).

[53] Une solution naïve serait de faire les calculs en cents, en utilisant un type entier (ex. $101¢$), plutôt qu'en dollars (en virgule flottante). Cependant, s'il faut calculer des fractions de cents (par ex., taxes, taux d'intérêt), cette solution ne serait pas appropriée.

primordial qu'un système de calcul formel permette des calculs de très grande précision avec des nombres décimaux. C'est pour cette raison que la révision *IEEE 754-2008* (*IEEE*, 2008) nous est utile, car elle combine la norme *IEEE 754-1985* et la norme *IEEE 854-1987* (*IEEE*, 1987) pour l'arithmétique flottante à bases indépendantes. Il est donc maintenant possible d'utiliser une arithmétique en base 10, tout en respectant la norme *IEEE 754*.

Par contre, un désavantage de la représentation en base 10 est qu'elle n'est pas optimisée pour profiter de la représentation en base 2 utilisée par les ordinateurs au niveau matériel. Ceci fait en sorte que les calculs en base 10 sont typiquement plus lents que ceux utilisant la base 2.

4.2 Bibliothèques *GMP* (*MPIR*) et *MPFR* (base 2)

La bibliothèque *GMP* (*GNU Multiple Precision arithmetic library*)[54], créée il y a 30 ans, est en constante évolution et est utilisée dans de nombreux logiciels *CAS* réputés, tels *Wolfram Mathematica* et *Maplesoft Maple*[55]. Elle offre la possibilité de faire des calculs avec une précision limitée uniquement par la mémoire disponible. Qui plus est, sa licence *LGPLv3* (*GNU Lesser General Public License v3*) est plus permissive que la licence *GPLv3* (*GNU General Public License v3*) et autorise son utilisation dans un logiciel utilisant une autre licence, si chargée dynamiquement.

GMP implémente aussi des optimisations de calculs de division et de racine carrée particulièrement intéressantes (Zimmermann, 2000). Cependant, elle ne fournit pas certaines fonctions mathématiques de base — notamment, les fonctions trigonométriques. Il existe aussi un projet similaire, la bibliothèque *GNU MPFR* (*GNU Multiple Precision Floating-point Reliable library*) (Fousse *et al.*, 2007). Cette dernière est basée sur *GMP*, mais elle inclut de nombreuses fonctions mathématiques, y compris les fonctions trigonométriques.

Ces deux bibliothèques utilisent la base 2 (binaire) pour stocker les nombres en mémoire, mais *GMP* ne suit que partiellement la norme *IEEE 754*. Il est donc préférable, non seulement pour les avantages cités précédemment, d'utiliser la bibliothèque *MPFR* plutôt que *GMP* :

---

[54] Sous *Windows*, c'est plutôt la bibliothèque *MPIR* (*Multiple Precision Integers and Rationals*) qui est habituellement utilisée : `http://www.mpir.org/` (archivé). Cependant, au moment de la rédaction du mémoire, nous avons réalisé que *MPIR* est maintenant considérée obsolète, donc notre projet devra être ajusté en conséquence.

[55] `https://library.wolfram.com/infocenter/Conferences/7518/Macalester_talk.txt` (*archive*) et `https://www.maplesoft.com/support/help/AddOns/view.aspx?path=GMP` (*archive*)

> *Note that the mpf functions are not intended as a smooth extension to IEEE P754 arithmetic. In particular results obtained on one computer often differ from the results on a computer with a different word size. New projects should consider using the GMP extension library MPFR (`https://www.mpfr.org`) instead. MPFR provides well-defined precision and accurate rounding, and thereby naturally extends IEEE P754.*
>
> Manuel de la bibliothèque *GMP* (Granlund et al., 2020, ch. 7 *Floating_Point-Function*)

4.3 Bibliothèque *Boost.Multiprecision* (bases 2 et 10)

La bibliothèque *Boost.Multiprecision* (Maddock et Kormanyos, 2022, ch. 1 *Boost.Multiprecision*) offre une interface `C++` vers plusieurs bibliothèques arithmétiques `C` externes — dont *GMP* et *MPFR*. Basée sur la bibliothèque *e_float* (Kormanyos, 2011), elle supporte les entiers ($\mathbb{Z}$), réels ($\mathbb{R}$), complexes ($\mathbb{C}$) ainsi que rationnels ($\mathbb{Q}$) — dans les bases 2 (binaire) et 10 (décimale). De plus, elle permet de supporter les fonctions trigonométriques avec la bibliothèque *GMP*.

Sa licence *Boost Software License*, de type *BSD* ou *MIT*, est permissive. Toutefois, l'utilisation de bibliothèques externes telles que *GMP* et *MPFR* requiert de respecter leur licence respective. C'est pourquoi *Boost.Multiprecision* supporte une version interne (moins performante), pour chacun des types, si le projet n'est pas en mesure de respecter les licences des bibliothèques externes ou pour éviter certaines dépendances. C'est le cas du type `cpp_dec_float` qui utilise la base 10, dont nous traiterons ci-dessous.

type *Boost.Multiprecision*	bibliothèque, ensemble, base, précision
☐ `gmp_int`	*GMP, entiers ($\mathbb{Z}$), base 2, « illimitée »*
☐ `cpp_int`	*Boost, entiers ($\mathbb{Z}$), base 2, « illimitée »*
☑ `gmp_float`	*GMP, réels ($\mathbb{R}$), base 2, « illimitée »*
☑ `mpfr_float`	*MPFR, réels ($\mathbb{R}$), base 2, « illimitée »*
☐ `cpp_bin_float`	*Boost, réels ($\mathbb{R}$), base 2, non testée*
☑ `cpp_dec_float`	*Boost, réels ($\mathbb{R}$), base 10, limitée*

Tableau 4.3 Liste des différents types de la bibliothèque *Boost.Multiprecision*.

Le tableau 4.3 dresse une liste non exhaustive des types de *Boost.Multiprecision* les plus intéressants pour notre projet. Les types que nous avons utilisés dans les classes `Numeric` (chapitre VII) y sont cochés.

Le type `boost::multiprecision::cpp_dec_float` est particulièrement intéressant car il utilise la base 10 (décimale). Par contre, il a certains désavantages : il est lent, a une précision limitée et, puisque construit autour des *templates* C++, est délicat à utiliser. Son implémentation actuelle n'est pas optimisée pour les grandes précisions — il devient lent à grandes précisions et consomme beaucoup de mémoire (sur la pile par défaut) (Maddock et Kormanyos, 2022, sect. `cpp_dec_float`). Comme nous le verrons à la sous-section 7.2.3, il nous a causé de nombreux problèmes en début de projet et il a finalement été remplacé par la bibliothèque *mpdecimal*.

4.4 Bibliothèque *mpdecimal* (base 10)

Mpdecimal (*libmpdec*) est une bibliothèque C — pour laquelle une *API* C++ via *libmpdec++* (Krah, 2021) et la classe `Decimal` est disponible — implémentant en totalité la spécification générale d'arithmétique décimale (*General Decimal Arithmetic Specification*) (Cowlishaw, 2009). Afin de supporter la base 10, elle respecte la norme *IEEE 754-2008* (*IEEE*, 2008), avec toutefois quelques restrictions. De plus, elle est multiplateforme (*cross-platform*) et sa licence de type *BSD* est permissive. C'est cette bibliothèque que *Python* utilise pour l'arithmétique à précision arbitraire en base 10 dans sa classe `Decimal`[56].

Contrairement à `cpp_dec_float` (section 4.3), elle est performante et optimisée pour les grandes précisions. De plus, elle a l'avantage d'être rapide pour manipuler des nombres représentables de façon exacte avec un nombre fini de décimales (par exemple, le nombre $0{,}01$ *versus* la constante π), même à de grandes précisions. Elle est donc performante pour calculer $1/2 + 2$, même si la précision demandée est d'un million de décimales. Ceci n'est pas le cas pour toutes les bibliothèques présentées précédemment et constitue un net avantage. Par contre, pour calculer $1/3 + \pi$ à de grandes précisions, elle est plus lente que les bibliothèques en base 2 *GMP* et *MPFR* (section 4.2).

Comme nous l'expliquerons plus en détail au chapitre VII, pour toutes ces raisons, c'est cette bibliothèque que notre projet utilise par défaut (plutôt que `cpp_dec_float`). Les complications qui sont survenues lors de son intégration dans le code C++ du projet y seront aussi discutées.

[56] *Python* utilisait, dans les versions précédant la version 3.3, sa propre bibliothèque `decimal.py`. Cette dernière était peu performante, ce que nous montrerons ultérieurement.

4.5 Avantages et inconvénients des différents types flottants : un tableau comparatif

En guise de conclusion à cette partie, nous présentons, dans le tableau 4.4, un résumé des avantages et inconvénients des différents types et bibliothèques discutés dans le présent chapitre.

type	avantages	inconvénients
`double` 64 bits **base 2**	• Aucune licence. • Supporte les fonctions trigonométriques. • Calculs rapides (au niveau matériel). • Création rapide. • Affichage rapide. • Respecte la norme *IEEE 754*.	• Représentation en mémoire non exacte (ex. 0,1). • Précision limitée à 17 décimales (page 32). • Non fiable dans les calculs décimaux (ex. calculs financiers).
`gmp_float` (*GMP/MPIR*) **base 2**	• Plus rapide que les types en base 10 pour les grandes précisions.	• Représentation en mémoire non exacte (ex. 0,1). • Licence restrictive (*LGPL*). • Ne supporte pas les fonctions trigonométriques. • Création lente. • Affichage lent. • Ne respecte que partiellement la norme *IEEE 754*. • Non fiable dans les calculs décimaux (ex. calculs financiers).
`mpfr_float` (*MPFR*) **base 2**	• Plus rapide que les types en base 10 pour les grandes précisions. • Supporte les fonctions trigonométriques. • Respecte la norme *IEEE 754*.	• Représentation en mémoire non exacte (ex. 0,1). • Licence restrictive (*LGPL*). • Création lente. • Affichage lent. • Non fiable dans les calculs décimaux (ex. calculs financiers).
`cpp_bin_float` (*Boost*) **base 2**	• Licence non restrictive. • Aucune bibliothèque à installer. • Supporte les fonctions trigonométriques.	• Représentation en mémoire non exacte (ex. 0,1). • Non fiable dans les calculs décimaux (ex. calculs financiers). • Jusqu'à deux fois plus lente que les bibliothèques *GMP* et *MPFR*. • Nous n'avons pas vérifié son support de la norme *IEEE 754* pour l'instant.
`cpp_dec_float` (*Boost*) **base 10**	• Représentation en mémoire exacte. • Licence non restrictive. • Aucune bibliothèque à installer. • Supporte les fonctions trigonométriques. • Creation rapide. • Affichage rapide. • Fiabilité des calculs décimaux (ex. calculs financiers).	• Compilation lente (*templates* C++). • Complexe à manipuler. • Calculs à grandes précisions lents. • Précision maximale limitée. • Ne respecte pas la norme *IEEE 754*.
`Decimal` (*mpdecimal*) **base 10**	• Représentation en mémoire exacte. • Licence non restrictive. • Calculs rapides des nombres à précision exacte et petite (0,01 *vs*. $1/3$ et π). • Création rapide. • Affichage rapide. • Respecte la norme *IEEE 754-2008*. • Fiabilité des calculs décimaux (ex. calculs financiers).	• Utilisation en C++ peut être compliquée (chapitre VII). • Ne supporte pas les fonctions trigonométriques. • Calculs plus lents que les types en base 2 pour les grandes précisions ($1/3$ et π *vs*. 0,01).

Tableau 4.4 Avantages et inconvénients des bibliothèques pour arithmétique multiprécision et `double`.

Dans les chapitres qui suivent, nous allons maintenant présenter nos propres contributions : l'interpréteur et le langage *Kpsilon* (chapitre V), notre approche pour la gestion de la mémoire avec le ramasse-miette *Boehm* (chapitre VI), ainsi que les types `Numeric` multiprécision (chapitre VII).

CHAPITRE V
INTERPRÉTEUR ET LANGAGE *KPSILON*

Ce chapitre vise à introduire le langage *Kpsilon* et son interpréteur. Quelques aspects de son implémentation sont aussi abordés. Ceci dit, ce chapitre ne se veut pas un manuel d'instructions, mais plutôt une introduction à l'utilisation et au fonctionnement de l'interpréteur ; un manuel plus détaillé se trouve en annexe A.

L'interpréteur permet à un utilisateur d'évaluer ou exécuter des expressions ou commandes supportées par le langage *Kpsilon*. Pour le moment, le langage et son interpréteur sont limités et ne supportent pas encore le *calcul symbolique* (chapitre I) : comme expliqué dans l'*introduction*, dû à la complexité du projet, nous avons été contraints d'en restreindre sa portée et de nous concentrer sur les principaux éléments de ses fondations. La mise en place d'outils de développement et de débogage (section 5.5 et annexe C), l'intégration du *compilateur de compilateur ANTLR 4* (chapitre II), la gestion de la mémoire au moyen du *GC Boehm* (chapitres III et VI) et la mise en place des différents composants (chapitre VII) pour permettre l'arithmétique multiprécision en bases 2 (binaire) et 10 (décimale) avec diverses bibliothèques externes (chapitre IV) sont les points qui ont particulièrement retenu notre attention.

Le langage et son interpréteur consistent, pour l'instant, en une calculatrice élémentaire supportant les opérations arithmétiques de base (incluant les multiplications implicites, exemple « `2x` » ou « `2(1+3)` » : cf. sous-section 5.4.2), les variables, la constante π (« `pi` »), ainsi que diverses commandes permettant de modifier les configurations, d'inspecter la mémoire et de déboguer (annexe A, section A.5).

Remarque: *La présence d'une barre horizontale ondulée (~~~~~) dans les captures d'écran de ce chapitre indique que certaines lignes moins pertinentes ont été omises.*

5.1 Aperçu de l'utilisation de l'interpréteur et du langage *Kpsilon*

Pour utiliser l'interpréteur *Kpsilon*, il suffit de fournir à l'invite « `kpsilon>` » des expressions ou commandes supportées par le langage *Kpsilon*. On peut aussi fournir des entrées multiples en les séparant avec « `;` ».

type `Numeric`	base arithmétique	identifiant interpréteur	bibliothèque arithmétique	précision maximale[57]
`RealBinaryGMP`	2	***b1***	*GMP (Boost `gmp_float`)*	$4{,}2 \times 10^9$
`RealBinaryMPFR`	2	***b2***	*MPFR (Boost `mpfr_float`)*	$4{,}2 \times 10^9$
`RealBinaryFloat`	2	***b3***	*N/A (`double` 64 bits)*	15 à 17
`RealDecimalCPPDec`	10	***d1***	*Boost `cpp_dec_float`*	$2{,}1 \times 10^6$
`RealDecimalMPDec`	10	***d2***	*mpdecimal*	$1{,}8 \times 10^{19}$

Tableau 5.1 Liste des types `Numeric` disponibles dans *Kpsilon*.

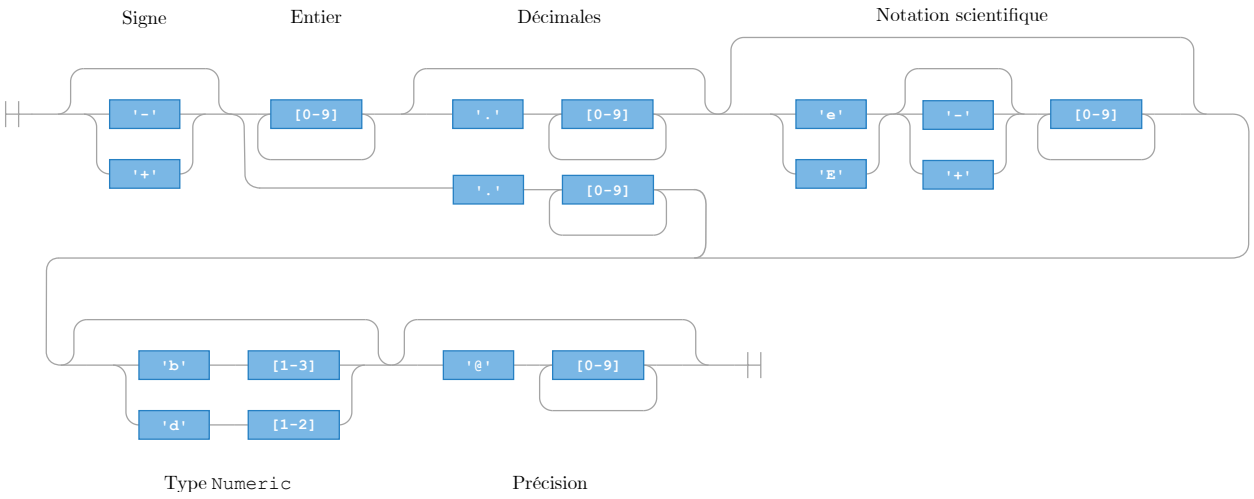


Figure 5.1 Diagramme syntaxique des valeurs des types `Numeric`.

Les différents types *`Numeric`* (section 5.2) supportés par l'interpréteur sont énumérés dans le tableau 5.1 ; les valeurs possibles de ces types sont décrites par le diagramme syntaxique de la figure 5.1. Dans le tableau 5.1, la colonne « *identifiant interpréteur* » indique l'identifiant à utiliser pour spécifier le type `Numeric` désiré. Par exemple, l'expression « `0.01`***`b2`***`@20` » dénote un nombre de type ***`RealBinaryMPFR`*** (« ***`b2`*** ») ayant une valeur de 0,01 à une précision de 20 décimales (« `@20` »). Une description complète du langage *Kpsilon* se trouve à l'annexe A.

L'interpréteur supporte les définitions de variables et les *caractères Unicode*. Il est donc possible d'utiliser directement des expressions telles que « σ = 0.2 » et « $\varphi = 2\pi + 2\sigma$ », plutôt que « `sigma = 0.2` » et « `phi = 2pi + 2sigma` ».

[57] C'est-à-dire, la précision maximale en nombre de chiffres, incluant la partie entière.

Pour le moment, l'interpréteur ne permet que l'évaluation d'expressions arithmétiques de base[58] et, au besoin, d'en affecter le résultat à une variable. Par contre, plusieurs commandes permettant d'analyser le comportement du *GC* et la consommation mémoire, de configurer et déboguer l'interpréteur, etc., ont été implémentées (cf. section 5.3). Ces commandes facilitent le développement du logiciel et nous ont été utiles pour valider son bon fonctionnement.

• **`set type`** *`type`*	Définit le type *`Numeric`* à utiliser par défaut.
• **`set prec`** *`précision`*	Définit la précision à utiliser par défaut.
• **`set disp`** *`précision`*	Définit le nombre de décimales affichées à l'écran.
• **`set mul`** *`mode`*	Configure le mode (`default`, `fixed` ou `hybrid`) de multiplications implicites (sous-section 5.4.2).
• **`ephm{on|off}`**	Active/déactive les objets éphémères (section 7.3).
• **`list`**	Affiche la liste des variables actuellement définies.
• **`mem`**	Affiche des informations sur l'état de la mémoire.
• **`dump [`***`objet`***`][`***`taille`***`]`**	Affiche le contenu mémoire de *`objet`*. Le champ *`objet`* peut être une adresse mémoire, un nom de variable ou une expression arithmétique. Le champ *`taille`* permet de définir le nombre d'octets à afficher.
• **`gc[full]`**	Force une collecte du *GC* (partielle ou complète).
• **`gcreset`**	Détruit et collecte tous les objets détectés par le *GC*.
• **`dumpgc`**	Affiche tous les objets détectés par le *GC*.
• **`showtree`**	Génère l'arbre syntaxique (*CST*, section 2.3) de la dernière entrée à l'invite de l'interpréteur.
• **`trace{on|off}`**	Active/déactive la fonctionnalité de « `TRACE` » pour déboguer le code `C++` (section A.5).
• **`trace{`***`niveau`***`}`**	Configure le niveau (1 à 5) de « `TRACE` » (section A.5).

Tableau 5.2 Liste (non-exhaustive) des commandes de l'interpréteur *Kpsilon*.

Le tableau 5.2 présente une liste, non-exhaustive, des commandes supportées par l'interpréteur : seules quelques commandes utiles à la compréhension du présent chapitre sont indiquées ; une liste complète ainsi que plusieurs exemples se trouvent à la *section A.5* du manuel du langage *Kpsilon* (annexe A).

[58] À savoir : addition, soustraction, multiplication et division sur des nombres à précision arbitraire en bases 2 et 10, ainsi que négation, parenthèses, et la constante π (`pi`).

Un exemple illustrant la commande « `showtree` » est présenté dans la capture 5.1, pour l'expression « `-0.4 / -3 + (1 + 1323) / 4213231312` ». Quant à la figure 5.2, elle présente l'arbre syntaxique (*CST*) généré pour cette expression, représenté de façon graphique.

```
kpsilon> -0.4 / -3 + (1 + 1323) / 4213231312

 Timings         : 0.003334s real, 0.000000s CPU (n/a%)
 Memory          : adr 0x7f84c72fa500, used 19922944, free 4770160640 (bytes)
 Data memory     : object 42248 bytes, total data 7498264 bytes in 10 objects
 Numeric type    : RealDecimalMPDec, prec:100000/100000, mem:42248
 Result          : 0.133333647581448842 44...
 Output timings : 0.002375s real, 0.010000s CPU (421.1%)

kpsilon> showtree

 Found tokens for "-0.4 / -3 + (1 + 1323) / 4213231312" :

   [@0,0:0='-',<'-'>,1:0]
   [@1,1:3='0.4',<NUM>,1:1]
   [@2,5:5='/',<'/'>,1:5]
   [@3,7:7='-',<'-'>,1:7]
   [@4,8:8='3',<NUM>,1:8]
   [@5,10:10='+',<'+'>,1:10]
   [@6,12:12='(',<'('>,1:12]
   [@7,13:13='1',<NUM>,1:13]
   [@8,15:15='+',<'+'>,1:15]
   [@9,17:20='1323',<NUM>,1:17]
   [@10,21:21=')',<')'>,1:21]
   [@11,23:23='/',<'/'>,1:23]
   [@12,25:34='4213231312',<NUM>,1:25]
   [@13,35:35='\ n',<NL>,1:35]
   [@14,36:35='<EOF>',<EOF>,2:0]

 Generating concrete syntax tree (CST) and open in inspector.
```

Capture 5.1 Commande « `showtree` » pour l'expression « `-0.4/-3+(1+1323)/4213231312` ».

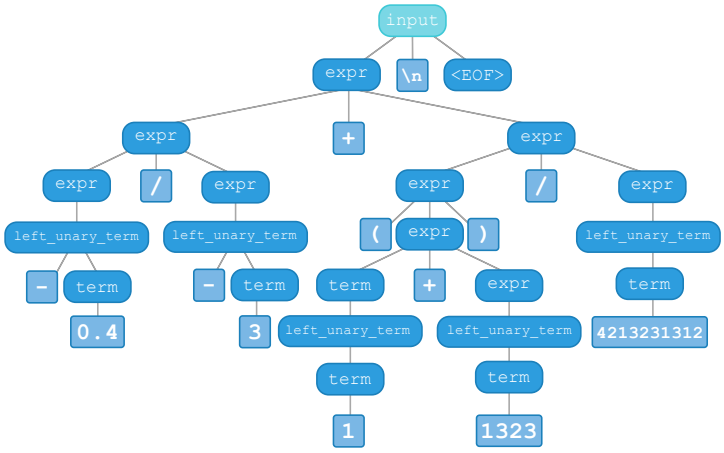


Figure 5.2 Représentation graphique de l'arbre syntaxique (*CST*) généré par la commande « `showtree` ».

5.2 Architecture de l'interpréteur *Kpsilon*

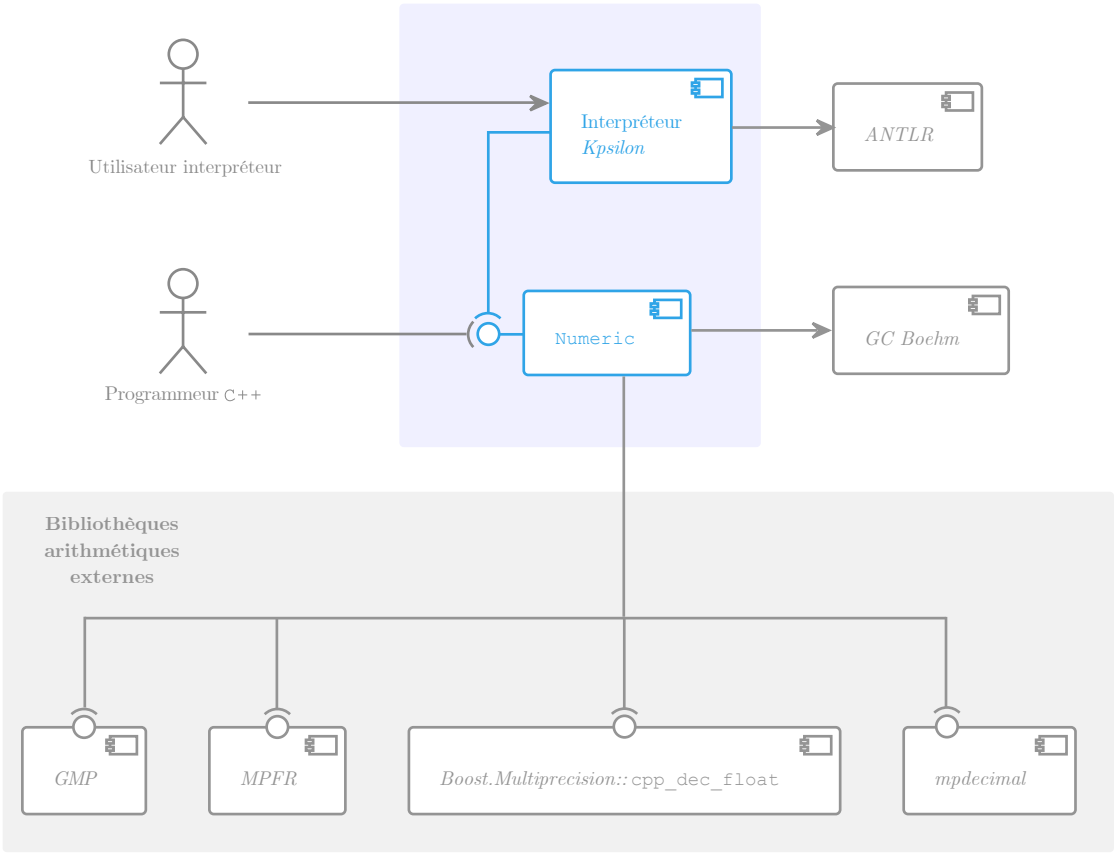


Figure 5.3 Diagramme *UML* de composants du logiciel *Kpsilon*.

Une vue d'ensemble des principaux composants du logiciel *Kpsilon* (en bleu) et des bibliothèques externes qu'il utilise (en gris) est présentée à la figure 5.3.

5.2.1 L'interpréteur

L'interpréteur *Kpsilon* est implémenté au moyen du *compilateur de compilateur ANTLR 4*. Comme mentionné précédemment (chapitre II), ce dernier permet, en plus de générer le code C++ des *analyseurs lexicaux* et *syntaxiques*, de générer le code du visiteur d'arbres syntaxiques, ce qui facilite la mise en œuvre.

5.2.2 La classe `Numeric`

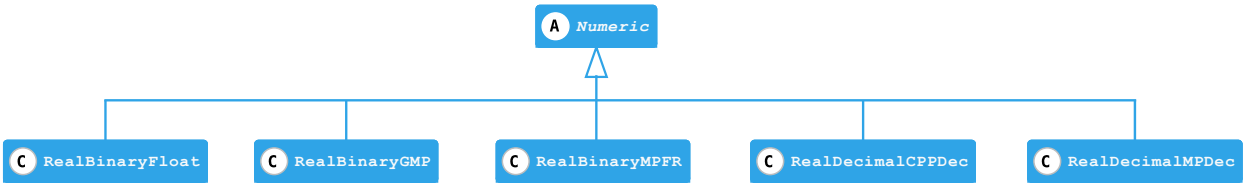


Figure 5.4 Diagramme *UML* des sous-classes `Numeric`.

Tel qu'expliqué au chapitre I, puisque *Kpsilon* sera avant tout destiné au *calcul symbolique*, il doit être en mesure de produire des résultats de façon *exacte jusqu'à une certaine précision*. Pour cette raison, l'interpréteur utilise par défaut une arithmétique à précision arbitraire en *base 10* (décimale), et ce, au moyen de la bibliothèque *Mpdecimal* (type `RealDecimalMPDec`)[59] tout en supportant aussi une autre bibliothèque décimale, *Boost* `cpp_dec_float` (type `RealDecimalCPPDec`). Néanmoins, l'interpréteur supporte aussi des types en *base 2* (binaire) à des fins d'optimisation — les calculs en base 2 sont généralement plus performants que ceux en base 10[60]. Ces différents types, implémentés par différentes bibliothèques, sont représentés par des sous-classes de la classe `Numeric` ; la figure 5.4 donne un aperçu de ces sous-classes. Ce choix d'implémentation sera expliqué plus en détail au chapitre VII.

5.2.3 Le *GC Boehm*

Le *GC Boehm* est utilisé pour gérer les allocations mémoires des objets du langage *Kpsilon*, c'est-à-dire libérer la mémoire allouée pour les objets `Numeric` qui ne sont plus utilisés. La gestion mémoire et l'interaction entre le *GC* et les objets du langage seront expliquées en détail au chapitre VII.

5.3 Commandes de gestion mémoire et de débogage

Plusieurs commandes sont disponibles pour analyser le contenu de la mémoire et vérifier le bon fonctionnement de l'interpréteur *Kpsilon* (voir la *liste partielle* à la section 5.1 ou complète à l'*Annexe A*).

[59] Comme expliqué précédemment, avec une arithmétique à précision arbitraire en base 2, la valeur 0,3 dans l'interpréteur *Kpsilon* (`0.3b2`) serait égale à 0,299 999 999 999 999 99 à une précision de 17 décimales.

[60] Supporter plusieurs bibliothèques arithmétiques permet aussi d'utiliser l'interpréteur dans les cas où des bibliothèques arithmétiques ne seraient pas installées ou supportées sous le système d'exploitation utilisé.

```
1  kpsilon> -0.4 / -3 + (1 + 1323) / 4213231312
2
3   Timings         : 0.000642s real, 0.000000s CPU (n/a%)
4   Memory          : adr 0x7fa507396b40, used 20193280, free 32663875584 (bytes)
5   Data memory     : object 41640 bytes, total data 373968 bytes in 9 objects
6   Numeric type    : RealBinaryGMP, prec:100000/100000, mem:41640
7   Result          : 0.13333364758144884244...
8   Output timings  : 0.029609s real, 0.030000s CPU (101.3%)
9
10 kpsilon> dumpgc
11
12 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
13  Number of managed allocated memory::Object : 10
14
15    0x7fa507396c40 -> 4213231312 (RealBinaryGMP, prec:100000/100000, mem:41640)
16    0x7fa507396cc0 -> 1324 (RealBinaryGMP, prec:100000/100000, mem:41640)
17    0x7fa507396d40 -> -0.4 (RealBinaryGMP, prec:100000/100000, mem:41640)
18    0x7fa507396dc0 -> 1323 (RealBinaryGMP, prec:100000/100000, mem:41640)
19    0x7fa507396ec0 -> 1 (RealBinaryGMP, prec:100000/100000, mem:41640)
20    0x7fa507396b80 -> 0 (RealBinaryGMP, prec:100000/100000, mem:41640)
21    0x7fa507396fc0 -> 0 (RealBinaryGMP, prec:100000/100000, mem:41640)
22    0x7fa507396e00 -> -3 (RealBinaryGMP, prec:100000/100000, mem:41640)
23    0x7fa507396f00 -> 0.1333336476 (RealBinaryGMP, prec:100000/100000, mem:41640)
24    0x7fa507396b40 -> 0.1333336476 (RealBinaryGMP, prec:100000/100000, mem:41640)
25
26  Total memory used by the GC allocators : 416400 bytes
27
28 kpsilon> gc
29
30  GC called, Numeric objects: 10->3, allocators memory: 415520->124656 bytes.
31
32 kpsilon> dumpgc
33
34 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
35  Number of managed allocated memory::Object : 3
36
37    0x7fa507396d40 -> -0.4 (RealBinaryGMP, prec:100000/100000, mem:41640)
38    0x7fa507396fc0 -> 0 (RealBinaryGMP, prec:100000/100000, mem:41640)
39    0x7fa507396b40 -> 0.1333336476 (RealBinaryGMP, prec:100000/100000, mem:41640)
40
41  Total memory used by the GC allocators : 124920 bytes
```

Capture 5.2 Exemple d'utilisation des commandes « `gc` » et « `dumpgc` ».

À titre d'exemple, la capture 5.2 présente la sortie d'un examen de l'état de la mémoire au moyen des commandes « *dumpgc* » (afficher les objets détectés par le *GC*) et « *gc* » (forcer une collecte du *GC*). On constate qu'après l'évaluation de l'expression « `-0.4/-3+(1+1323)/4213231312` » (*ligne 1*), l'affichage de la liste des objets détectés pas le *GC* avec la commande « `dumpgc` » (*lignes 15* à *24*) contient plusieurs objets `Numeric` intermédiaires. Dans cette liste, l'objet en mauve contient la valeur du résultat de l'évaluation de l'expression et ne sera pas détruit lors d'une collecte ; les objets en bleu sont possiblement utilisés et pourraient ne pas être détruits ; tous les autres objets de la liste ne sont plus en utilisation et seront assurément

détruits. Après une collecte forcée par la commande « `gc` » (*ligne 28*), on voit que la liste des objets détectés est réduite après l'exécution de « `dumpgc` » (*lignes 37* à *39*).

Notons que plusieurs commandes de débogage ne sont pas énumérées dans la liste partielle des commandes (tableau 5.2), mais sont présentées dans le manuel du langage *Kpsilon* (annexe A). Il est aussi possible d'activer, au moment de la compilation, d'autres fonctionnalités permettant de déboguer l'interpréteur. La fonctionnalité de « `TRACE` »[61] est particulièrement utile pour déboguer l'interpréteur au niveau du code `C++` et est présentée à la *section* A.5.

5.4 Autres caractéristiques de *Kpsilon* : Calculs à précision mixte et multiplications implicites

5.4.1 Calculs à précisions mixtes

Dans une opération arithmétique binaire (avec deux opérandes), si deux valeurs `Numeric` n'utilisent pas la même précision, la précision du résultat sera celle de la valeur ayant la plus petite précision. De plus, si les types diffèrent, le résultat sera du type le moins précis des deux. Une telle approche est nécessaire pour éviter de créer des valeurs inutilement — pour ne pas dire « *faussement* » — précises.

Plus spécifiquement, les types `Numeric` sont ordonnés en précision ou « *qualité* » comme suit — c'est-à-dire qu'un `double` 64 bits est moins précis qu'un type multiprécision, et qu'un type en base 2 est moins précis qu'un type en base 10 :

$$T_{ord} : \textbf{\texttt{b3}}\ (\texttt{Float}) \prec \textbf{\texttt{b1}}\ (\texttt{GMP}) \prec \textbf{\texttt{b2}}\ (\texttt{MPFR}) \prec \textbf{\texttt{d1}}\ (\texttt{CPPDec}) \prec \textbf{\texttt{d2}}\ (\texttt{MPDec})$$

Cet ordre est basé sur la précision, mais aussi sur la « *qualité* » des bibliothèques multiprécision utilisées par les types : tel que mentionné au chapitre IV, *MPFR* est préférable à *GMP* (plus ancienne), alors que *mpdecimal* est préférable aux *Boost* `cpp_dec_float` (plus lents).

La figure 5.5 présente une règle qui décrit le type et la précision du résultat lorsque des valeurs de type et précision différentes sont combinées : min_{ord} est le minimum qui correspond à l'ordre T_{ord} sur la « *qualité* » des types, alors que min_N est le minimum usuel sur les entiers.

[61] La fonctionnalité « `TRACE` » doit être activée lors de la compilation du logiciel. Pour activer les « `TRACE`s » dans l'interpréteur, on utilise les commandes « *`trace{on|off}`* » et « *`trace{niveau}`* ».

Soit v_1, *resp.* v_2, *de type* `Numeric` t_1, *resp.* t_2, *avec précision* p_1, *resp.* p_2 :

$v_1 : t_1@p_1, v_2 : t_2@p_2$

Soit une opération binaire $\odot$

Alors

$v_1 \odot v_2 : min_{Tord}(t_1, t_2)@min_{\mathbb{N}}(p_1, p_2)$

Figure 5.5 Règle décrivant le type et la précision du résultat pour des valeurs de type et précision différentes.

```
1  kpsilon> res = 1b1@10 / 3d2@100
2
3   Timings         : 0.005745s real, 0.000000s CPU (n/a%)
4   Memory          : adr 0x7fe77f2f7d40, used 19730432, free 5060972544 (bytes)
5   Data memory     : object 112 bytes, total data 200 bytes in 7 objects
6   Numeric type    : RealBinaryGMP, prec:10/10, mem:112
7   Result          : 0.3333333333
8   Output timings  : 0.000421s real, 0.000000s CPU (n/a%)
9
10 kpsilon> list
11
12  Variables list :
13
14  1) _   : 0.3333333333 (kpsilon::numeric::RealBinaryGMP, prec:10/10, mem:112)
15           @ 0x7fe77f2f7d40
16  2) res : 0.3333333333 (kpsilon::numeric::RealBinaryGMP, prec:10/10, mem:112)
17           @ 0x7fe77f2f7d40
18
19  Total memory (bytes)         : 112
20  Maximum number of variables : 329406144173384850
```

Capture 5.3 Exemple de l'opération arithmétique $1 \div 3$ entre différents types `Numeric` dans l'interpréteur.

Un exemple d'un calcul avec les types *RealBinaryGMP* et *RealDecimalMPDec* est présenté dans la capture 5.3. On peut observer que le résultat de l'expression « `res = 1b1@10 / 3d2@100` » (ligne 1) est un *RealBinaryGMP* à une *précision de 10 décimales* (en vert, ligne 6) — soit de *type* et *précision* fixés par la règle présentée dans la figure 5.5.

Pour éviter de perdre de la précision lors de telles opérations mixtes, il est malgré tout souhaitable d'effectuer les opérations arithmétiques en utilisant la plus grande des précisions, quitte à convertir le résultat à la plus petite précision une fois le calcul terminé. L'interpréteur *Kpsilon* effectue donc une première phase pour identifier les précisions *min*/*max*, et ensuite effectue la(les) opération(s) arithmétique(s) à la précision *max*

lors d'une deuxième phase, pour finalement convertir le résultat à la précision *min* appropriée.

```
kpsilon> 4.6@1000 + 1e-8@1 + 4.6@1000

 Timings         : 0.000642s real, 0.000000s CPU (n/a%)
 Memory          : adr 0x7f8b3faeb000, used 21684224, free 32413343744 (bytes)
 Data memory     : object 136 bytes, total data 6437318 bytes in 7 objects
 Numeric type    : RealDecimalMPDec, prec:1/1, mem:136
 Result          : 8
 Hex value       : 0x08
 Output timings : 0.001028s real, 0.000000s CPU (n/a%)
```

Capture 5.4 Évaluation d'une expression arithmétique **sans** identification de précisions *min*/*max*.

```
kpsilon> 4.6@1000 + 1e-8@1 + 4.6@1000

 Timings         : 0.000511s real, 0.000000s CPU (n/a%)
 Memory          : adr 0x7f7df5515c00, used 20799488, free 32413782016 (bytes)
 Data memory     : object 592 bytes, total data 3234719 bytes in 7 objects
 Numeric type    : RealDecimalMPDec, prec:1/1, mem:592
 Result          : 9
 Hex value       : 0x09
 Output timings : 0.001124s real, 0.010000s CPU (889.9%)
```

Capture 5.5 Évaluation d'une expression arithmétique **avec** identification de précisions *min*/*max*.

Les captures 5.4 et 5.5 présentent les résultats de l'évaluation de l'expression « `4.6@1000 + 1e-8@1 + 4.6@1000` » dans l'interpréteur *Kpsilon* sans *vs.* avec le calcul des précisions *min*/*max*. Notons que l'interpréteur n'arrondit pas les valeurs décimales au moment de leur création ou lors d'une conversion de précision ; celles-ci sont plutôt tronquées pour préserver la valeur exacte de la dernière décimale[62]. Puisque, dans cette expression, la valeur 1×10^{-8} utilise une précision d'une décimale, sans détection de précisions *min*/*max* et afin de retourner un résultat ayant la plus petite des précisions, la valeur $4{,}6$ est convertie en 4 (précision d'une décimale, incluant la partie entière, avec troncation) au moment d'effectuer les opérations d'addition. Dans ce cas, le résultat de l'évaluation de l'expression est 8 et non 9 comme c'est le cas en activant la détection de précisions *min*/*max*. Or, sans limitation de précision, le résultat de cette expression est $9{,}200\,000\,01$, donc un résultat de 9 est plus approprié que 8.

[62] Nous prévoyons ajouter une option pour sélectionner le comportement par défaut désiré, i.e., arrondissement ou troncation.

5.4.2 Multiplications implicites et le paradoxe de *PEMDAS*

Le *paradoxe de PEMDAS* (*Parenthèse, Exposant, Multiplication/Division, Addition/Soustraction*) pose la question suivante : est-ce que $6/2(1+2) = 1$ ou 9 ? Or, il semble qu'il n'y a pas de « *bonne réponse* », car aucune convention n'existe pour la priorité des opérations des multiplications implicites (Linkletter, 2019). Néanmoins, la réponse 9 semble être celle privilégiée dans les calculatrices et logiciels récents, c'est-à-dire, $6/2(1+2) = 6/2 \times (1+2) = 9$.

```
1  kpsilon> x = 2
2   2
3
4  kpsilon> 1/(2x)
5   0.25
6
7  kpsilon> set mul default
8   Implicit multiplications set to default mode.
9
10 kpsilon> 6/2(1 + 2)
11  1
12
13 kpsilon> 1/2x
14  0.25
15
16 kpsilon> set mul fixed
17  Implicit multiplications set to fixed mode.
18
19 kpsilon> 6/2(1 + 2)
20  9
21
22 kpsilon> 1/2x
23  1
24
25 kpsilon> set mul hybrid
26  Implicit multiplications set to hybrid mode.
27
28 kpsilon> 6/2(1 + 2)
29  9
30
31 kpsilon> 1/2x
32  0.25
```

Capture 5.6 Modes de multiplications implicites.

Initialement, l'interpréteur *Kpsilon* retournait le résultat $6/2(1+2) = 1$, puisque c'est l'interprétation naïve de ce calcul — comme si les parenthèses, pcq. collées sur le 2 qui précède, avaient priorité. Une option pour choisir le mode d'ordre des multiplications implicites a par la suite été ajoutée (commande « `set mul default|fixed|hybrid` ») — la *ligne 11* de la capture 5.6 affiche la réponse par défaut (mode « `default` ») du calcul $6/2(1+2)$, alors que la *ligne 20* affiche le résultat corrigé (mode « `fixed` »).

Bien que la réponse $6/2(1+2) = 9$ soit celle couramment acceptée, examinons l'expression $1/2x$. Pour $x = 2$, en mode « `fixed` » le résultat est $1/2x = (1/2)x = 1$, tel qu'indiqué à la *ligne 23*. Or, en mathématiques, on aurait plutôt tendance à interpréter cette expression comme $\frac{1}{2x} = \frac{1}{4} = 0{,}25$. Pour cette raison, un mode hybride a aussi été ajouté (mode « `hybrid` ») (*ligne 32*).

5.5 Autres outils utilisés pour le développement de *Kpsilon*

Comme mentionné en *introduction*, plusieurs outils de vérification de code, de débogage, de profilage, etc. ont été intégrés au projet et utilisés tout au long du développement du logiciel *Kpsilon* :

- La bibliothèque *Catch2* permet de mettre en place des tests unitaires. En outre, nous avons utilisé à plusieurs reprises une approche *TFD* (*Test-First Development*).
- Par défaut, le débogueur de *Microsoft Visual Studio* n'affiche pas les valeurs numériques des objets des types multiprécision. Toutefois, nous avons pris soin de construire les classes des types multiprécision de façon à ce qu'elles soient compatibles avec le *framework Natvis*. Ce dernier permet de choisir la valeur affichée dans la fenêtre d'inspection du débogueur, ce qui facilite le débogage.
- L'outil d'analyse statique *clang-tidy* (un *linter*) permet d'analyser le code source pour valider le respect des bonnes pratiques `C++` dans *Windows* et *Linux*, par exemple, valider le respect des « *`C++` Core Guidelines* » (Stroustrup et Sutter, 2024).
- L'*analyseur statique* de *Clang* permet la détection de bogues potentiels au moment de la compilation.
- Le logiciel *Valgrind*, avec ses différents outils de débogage et de profilage — *Memcheck*, *Cachegrind*, *Callgrind*, *Massif*, *Helgrind*, *DRD* et *DHAT* — permet la détection de fuites ou de bogue au niveau de la gestion mémoire, le profilage, le débogage multitâche (*multithreading*), etc. Malheureusement, son outil *Memcheck* permettant la détection de fuites mémoire n'est pas compatible avec le *GC Boehm* — il est donc inutile pour valider le bon fonctionnement du *GC*. Par contre, son outil de profilage *Callgrind* nous a été utile pour isoler des goulots d'étranglement (*bottleneck*) dans le code.
- Les analyseurs dynamiques *AddressSanitizer*, *MemorySanitizer* et *ThreadSanitizer* de *Clang* permettent la détection de bogues ou fuites mémoire au moment de l'exécution. *AddressSanitizer* est particulièrement utile, car contrairement à *Valgrind*, il est compatible avec le *GC Boehm* — il a été utilisé tout au long du développement pour détecter les fuites mémoire et valider le bon fonctionnement du *GC*.
- Le profileur (*profiler*) *GNU gprof* permet de détecter les goulots d'étranglement (*bottleneck*) pour régler les problèmes de performance.
- L'analyseur de couverture (*coverage analyzer*) *gcov* (ainsi que *gcovr* qui génère des rapports) est utile pour détecter les parties de code qui ne sont jamais utilisées (*dead code*) ou vérifier la couverture des tests unitaires.

CHAPITRE VI
GESTION DE LA MÉMOIRE AVEC LE RAMASSE-MIETTES BOEHM ET AL.

Dans ce chapitre, nous présentons les principales étapes d'intégration du ramasse-miettes (*garbage collector* ou *GC*) *Boehm-Demers-Weiser* à notre projet, ainsi que les principales difficultés rencontrées. Bien que nous avions présumé que son intégration serait simple — pour ne pas dire « *plug and play* » — plusieurs problèmes sont survenus. Obtenir le comportement désiré de la part du *GC* a donc exigé des mois de recherche et d'expérimentation.

Plus spécifiquement, nous avions comme exigences que le *GC* respecte les conditions suivantes :

1. Se comporter de façon prévisible et sans fuites mémoire ;
2. Se comporter de façon identique (ou similaire) sous les différentes plateformes supportées ;
3. Permettre le lancement implicite *vs.* explicite des collectes ;
4. Supporter le contrôle dynamique des intervalles des collectes ;
5. Pouvoir être désactivé temporairement, et ce, sans engendrer de fuite mémoire ;
6. Autoriser la destruction explicite des objets au moyen de l'opérateur `delete` ;
7. Détecter implicitement la quantité exacte de mémoire gérée ;
8. Supporter la programmation parallèle via le *multithreading*.

Comme nous allons l'expliquer, le respect de ces conditions n'a été acquis qu'après avoir surmonté plusieurs embûches ; à vrai dire, seulement après avoir compris et ajouté des *allocateurs mémoire*, introduits à la section 6.9 et présentés plus en détails au chapitre VII.

6.1 Évaluation du comportement du *GC* et graphes de consommation mémoire

Dans ce chapitre, la consommation mémoire du logiciel et les temps d'exécutions sont analysés, en fonction de différentes configurations du *GC*, dans le but de vérifier le bon fonctionnement du *GC* et de valider le respect des conditions énumérées ci-dessus. Tous les graphes de ce chapitre ont été générés par la fonctionnalité *« tests du GC »* (*Garbage collector tests*) du logiciel. Le listing 6.1 contient une description simplifiée[63] du fonctionnement de ce test, sous forme de pseudocode `C++`. En gros, le test crée dynamiquement des objets

[63] Ce code comporte environ 700 lignes, sans compter les fonctions de lecture mémoire, d'affichage à l'écran et de génération des graphes. Son fonctionnement détaillé est expliqué à la section B.1.

inaccessibles — auxquels ne réfère aucun pointeur. Puisque ces objets sont inaccessibles, seul le *GC* est en mesure de les détruire s'il fonctionne correctement. Le test génère aussi un graphe de consommation mémoire et de mesure de temps d'exécution, ce qui permet de vérifier le bon fonctionnement du *GC*.

```
Numeric* pObj;

auto temps_depart = temps_courant();

// Première phase de génération d'objets accessibles
creer_objets_accessibles(NOMBRE_OBJS_ACCESSIBLES);

// Phase de génération d'objets inaccessibles
for (int no = 0; no < NOMBRE_OBJS_INACCESSIBLES; no++) {
    /* Le même pointeur pObj est utilisé, donc il ne
       pointe plus vers les objets Numeric précédents;
       ces derniers sont donc inaccessibles! */

    pObj = Numeric::create(TYPE, PREC, VAL);

    analyser_memoire_libre();
}

// Deuxième phase de génération d'objets accessibles
creer_objets_accessibles(NOMBRE_OBJS_ACCESSIBLES);

auto temps_total = temps_courant() - temps_depart;
```

Listing 6.1 Pseudocode C++ du test du *GC*.

Plus spécifiquement, tel qu'illustré dans le pseudocode, ce test procède à deux phases de création d'objets accessibles (`NOMBRE_OBJS_ACCESSIBLES`), où la mémoire est explicitement libérée et le *GC* n'est aucunement sollicité. Entre ces deux phases s'insère une phase de création d'une certaine quantité d'objets `Numeric` inaccessibles (`NOMBRE_OBJS_INACCESSIBLES`) d'un certain type (`TYPE`), ayant comme valeur π (`VAL`) à une certaine précision (`PREC`). Si le *GC* libère correctement la mémoire allouée durant la phase intermédiaire de création d'objets *inaccessibles*, la mémoire utilisée lors de la deuxième phase de création d'objets accessibles devrait être (approximativement) la même que lors de la première phase. Dans le cas contraire, on serait en présence d'un problème de fuites mémoire[64].

[64] Par exemple, un tel problème survient lors de la 2^e phase de création d'objets accessibles dans la figure 6.14.

Remarque: *Des versions détaillées des graphes se trouvent en annexe D et peuvent être visualisées en cliquant sur les graphes. Ces versions contiennent de l'information sur les configurations du logiciel lors de l'exécution et permettent de visualiser les collectes implicites vs.* explicites *du GC. De plus, elles incluent un code QR qui contient des informations sur les configurations*[65] *de compilation et d'exécution — les instructions pour décoder ces codes QR se trouvent à l'annexe D. Pour simplifier la navigation entre les versions simplifiées et détaillées des graphes, il suffit de cliquer sur les graphes pour se déplacer d'une version à l'autre.*

6.2 Principales configurations analysées

L'*API (Application Programming Interface) du GC Boehm* contient plusieurs fonctions permettant de configurer le comportement du *GC* — toutes les fonctions mentionnées dans ce chapitre dont le nom débute par le préfixe «`GC_`» font partie de cette *API*. Bien que plusieurs autres fonctions de l'*API* et différentes options de compilations de la bibliothèque du *GC* ont été testées, seules celles intéressantes dans le cadre de notre projet seront présentées.

Voici une liste des différentes configurations du *GC* analysées :

- ***Valeur du diviseur d'espace libre* :** La fonction `GC_set_free_space_divisor` permet de modifier cette valeur pour ajuster les intervalles de collectes du *GC*, c'est-à-dire, qu'une grande valeur de diviseur déclenchera une libération de la mémoire inutilisée plus fréquemment qu'une petite valeur ;
- ***Seuil de consommation mémoire* :** Bien que cette fonctionnalité ne soit pas fournie par l'*API* du *GC*, nous l'avons ajoutée à notre logiciel par mesure de sécurité. Elle permet de forcer une collecte lorsqu'un seuil prédéfini de consommation mémoire est franchi ;
- ***Mode incrémentiel* :** La fonction `GC_enable_incremental` permet d'activer le mode *incrémentiel* du *GC* (section 3.1) ;
- ***Taille du tas* :** Le *GC* utilise son propre tas (*heap*) pour surveiller la mémoire qu'il doit gérer. Les fonctions `GC_expand_hp` et `GC_set_max_heap_size` permettent de modifier la taille de ce tas.

Tout au long du projet, nous avons pris soin que toute modification au comportement du logiciel soit activée au moyen d'options de compilation, pour toujours pouvoir revenir à un comportement antérieur. Toutefois,

[65] En observant les résultats des différents tests présentés dans ce mémoire, on pourra remarquer que certains logiciels et plateformes utilisés sont maintenant obsolètes. Ceci n'était pas le cas en début de projet, mais pour des questions monétaires et par manque de temps, nous avons choisi de ne pas mettre à jour notre environnement de développement en cours de route. La liste des plateformes, architectures et compilateurs `C++` testés se trouve à l'annexe A.

nous avons réalisé lors de la rédaction du mémoire que ce n'était pas toujours possible. Lorsqu'une ancienne version a été utilisée pour générer un graphe, l'icône 🕘 est affichée à gauche de sa légende.

6.3 Lecture de consommation mémoire sous *Linux*

Remarque: *Dans le graphe de consommation mémoire de la figure 6.1 — et les autres graphes du présent chapitre et du chapitre VII —, les deux zones avec un arrière-plan hachuré représentent deux périodes de créations d'objets* `Numeric` *accessibles (un pointeur vers un tel objet existe en tout temps). Après la première période de création d'objets accessibles, une longue période de création d'objets* `Numeric` *inaccessibles (axe horizontal, c.-à-d. de* 1 *à* 10 000 *objets dans la figure 6.1) est effectuée. La consommation mémoire des objets non détruits par le GC peut être observée sur l'axe vertical.*

Lors de l'exécution du test du *GC* pour générer les graphes de consommation mémoire et de temps d'exécution, nous avons constaté que la gestion de la mémoire se faisait de façon différente d'une plateforme à l'autre. Plus spécifiquement, sous *Linux*, il est impossible d'obtenir avec exactitude la quantité de mémoire consommée par un processus et, par le fait même, la mémoire libre. Par contre, dans *Windows*, bien qu'un problème similaire soit aussi présent, nous ne constatons pas d'erreur substantielle lors des lectures de consommation mémoire.

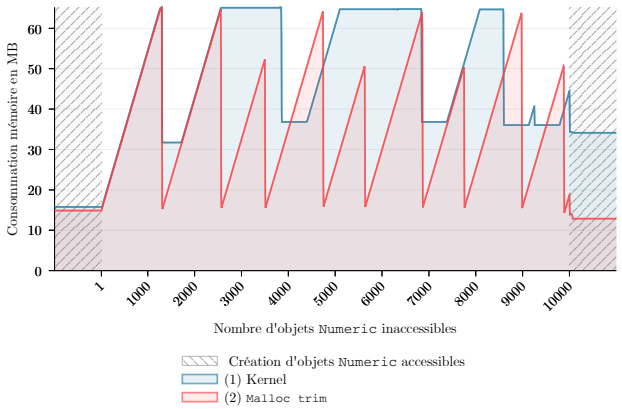


Figure 6.1 Lecture de consommation mémoire dans *Linux* (*détails*).

Sous *Linux*, ce problème peut être constaté en observant la courbe de la mémoire consommée par *Kpsilon* rapportée par le *kernel* (courbe 1 en bleu), présentée à la figure 6.1. Des plateaux de quantité de mémoire consommée sont présents après les collectes du *GC* (la *version détaillée* affiche les collectes). Pourtant, ces plateaux sont trompeurs, car la mémoire est bel et bien libérée (par un appel à `free` en `C` ou à `delete` en `C++`), donc disponible ; le *kernel* n'en a simplement pas « *conscience* ».

Sans décrire le fonctionnement de la gestion mémoire sous *Linux*, puisqu'il s'agit d'un sujet complexe, précisons que dans certaines situations il arrive que le *kernel* ne soit pas en mesure de détecter la quantité réelle de mémoire libre. Ce problème bien documenté a été soulevé à maintes reprises (Roche, 2022) et peut faussement donner l'impression qu'un *GC* ne fonctionne pas correctement — il est la cause de critiques de certains langages de programmation, dont *Ruby* (Lai, 2019). Une solution à ce problème, mentionnée dans ces références, consiste alors à faire un appel à la fonction `C malloc_trim` avant une lecture de consommation mémoire (ou à d'autres moments opportuns). Ceci permet, lorsque possible, de libérer la mémoire du point de vue du *kernel* (quantité de mémoire « *non coupée* » (*untrimmed*) sur le dessus du tas). Le résultat de la lecture de la mémoire consommée en utilisant `malloc_trim` (avant chaque lecture) est illustré dans la courbe 2 (en rouge) de la figure 6.1. C'est donc la lecture avec `malloc_trim` qui est utilisée, sous *Linux*, pour générer la totalité des graphes de ce mémoire.

6.4 Utilisation du ramasse-miettes *Boehm* et complications

6.4.1 Utilisation du *GC Boehm*

En cours de route, nous avons réalisé que nous avions mal compris le fonctionnement de la bibliothèque du *GC Boehm*. Nous avions cru, à tort, qu'il serait impossible que le *GC* appelle implicitement les destructeurs des objets au moment de leur collecte : nous faisions hériter les objets de la classe `gc` pour activer leur prise en charge par le *GC*, et la classe `gc` n'appelle pas implicitement les destructeurs. Après lecture des *commentaires du code source*, nous avons compris que pour appeler implicitement les destructeurs, c'est la classe `gc_cleanup` qui doit être utilisée, classe qui n'est pas mentionnée dans la documentation de la bibliothèque. C'est donc cette classe qui a été utilisée pour nos expérimentations.

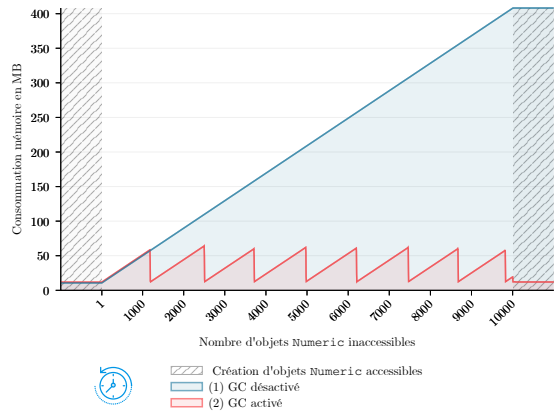


Figure 6.2 Le *GC* dans *Linux* (*détails*).

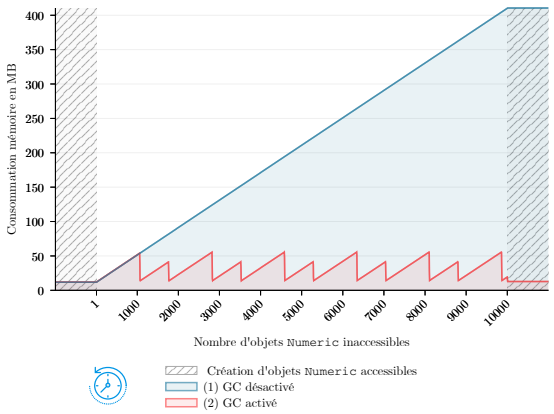


Figure 6.3 Le *GC* dans *Windows* (*détails*).

Les figures 6.2 et 6.3 permettent de visualiser la consommation mémoire lors de l'exécution, sous *Linux* et *Windows*, du test de création d'objets inaccessibles (section B.1) avec et sans *GC* : sans *GC* (en bleu), la consommation mémoire augmente de façon linéaire et occasionne inévitablement un problème de fuites mémoire. Par opposition, en activant le *GC* (en rouge), on observe que ce dernier fonctionne comme prévu en libérant la mémoire à intervalles réguliers.

Si ce test était exécuté de manière à créer un plus grand nombre d'objets inaccessibles, le logiciel planterait rapidement. Pire encore, le système d'exploitation pourrait devenir non réactif, ou même planter, par manque de mémoire. Ce dernier dénouement est fréquent sous *Windows*, alors que *Linux* semble mieux gérer la consommation mémoire des processus et aura tendance à terminer l'exécution du processus fautif.

6.4.2 Désactivation temporaire du *GC*

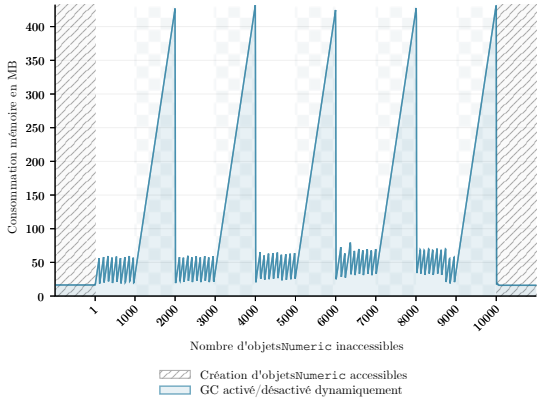


Figure 6.4 Activation/désactivation dynamique du *GC* dans *Linux* (*détails*).

On peut constater dans les zones avec un arrière-plan en échiquier de la figure 6.4, où le *GC* est dynamiquement désactivé (au début de la zone) puis réactivé (à la fin), qu'il est possible de désactiver temporairement le *GC* tout en évitant les fuites mémoire. Ce résultat permet de conclure que la *condition 5* est respectée. Cette condition est importante, car l'expérience nous a montré qu'il est parfois utile de désactiver temporairement le *GC* à des fins d'optimisations. Citons, à titre d'exemple, un projet de session où nous avions désactivé le *GC* de *Java* lors de calculs mathématiques intensifs pour en diminuer significativement les temps de calcul.

6.4.3 Détection de la mémoire utilisée par les objets `Numeric`

Il n'est pas possible de faire en sorte que le *GC* prenne en charge les ressources mémoire allouées par les objets internes des bibliothèques d'arithmétique multiprécision puisque ces objets lui sont invisibles. Ces dernières sont donc libérées explicitement par les destructeurs des objets *`Numeric`*[66], eux-mêmes appelés implicitement lors de la collecte des miettes. Ceci fait en sorte que le *GC* n'est pas « *conscient* » de la quantité réelle de mémoire utilisée par les objets `Numeric` — les collectes risquent ainsi de ne pas être lancées à des intervalles adéquats. Nous présenterons une solution à ce problème à la section 6.9.

6.4.4 Comportement du *GC* sous différentes plateformes

Un autre problème avec le *GC Boehm* est qu'il se comporte différemment de version en version et de plateforme en plateforme. En effet, sa forte dépendance aux spécificités du système le rend parfois « *capricieux* ».

Premièrement, sous *Windows*, il était au départ impossible de le faire fonctionner correctement dans ses versions les plus récentes. La version 7.4.18 du *GC* devait être utilisée sur ce système d'exploitation, la version 8.0.4 ne fonctionnant pas comme elle le devrait, provoquant à tout coup une erreur de segmentation[67]. Heureusement, après plusieurs mois d'expérimentations de potentielles solutions, il a été possible de régler ce problème et d'utiliser la version 8.0.4 du *GC* dans *Windows* : il s'agissait de *linker* statiquement et non dynamiquement certaines bibliothèques — c.-à-d. toutes les bibliothèques utilisées par le logiciel, hormis celles de *GMP*/*MPIR* et *MPFR*, puisque leur licence *LGPLv3* ne le permet pas avec la licence *MIT* utilisée par *Kpsilon*.

Deuxièmement, de plateforme en plateforme, les collectes ne sont pas réparties de façon similaire. Les figures 6.5 et 6.6 exposent le comportement du *GC* sous différentes plateformes — avec une précision d'un million de décimales. On constate, dans la section *temps d'exécution* de la figure 6.6, que le même test exécuté sous *Windows* (courbes (2)) est une fois et demie plus lent que sous *Linux* (courbes (1)). Cette dernière plateforme est pourtant installée sur le même ordinateur de façon native sur une deuxième partition. Ce problème sera discuté en détail à la section 6.5 lors de l'introduction de la notion de valeur du diviseur du *GC*.

[66] Cette classe `C++` du système de calcul formel sert à encapsuler les bibliothèques multiprécision `C` (section 7.2).

[67] Cette erreur de segmentation semble être causée par l'inclusion de la bibliothèque dynamique *ANTLR*.

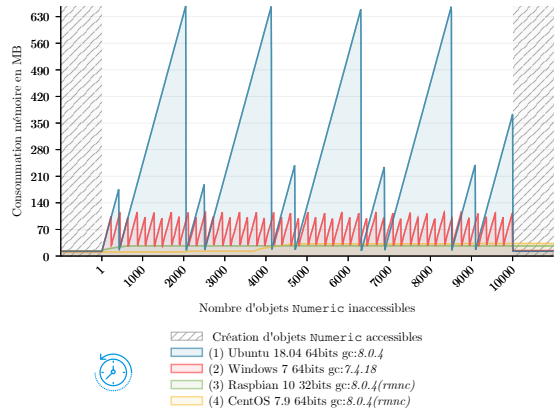


Figure 6.5 Comportement du *GC* sous différentes plateformes (ancienne version, *détails*).

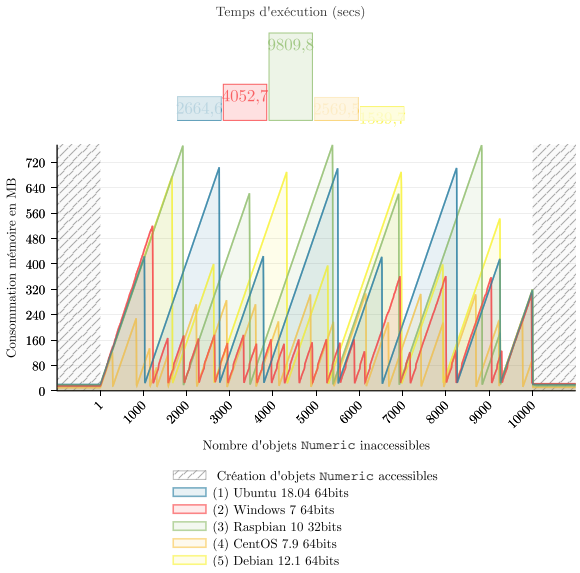


Figure 6.6 Comportement du *GC* sous différentes plateformes (nouvelle version, *détails*).

Troisièmement, pour une raison toujours incomprise, sur les ports *Linux CentOS 7.9* et *Raspbian 10*, la version 8.0.4 de la bibliothèque du *GC* utilisée sans modification provoque des erreurs de segmentation (*segmentation faults*) à la sortie du logiciel[68]. Après plusieurs semaines consacrées à ce bogue, nous avons compris qu'il n'est pas causé par un problème dans le code du projet, mais par une tentative du *GC* de libérer une partie de la mémoire allouée par la bibliothèque *Boost.Multiprecision* à laquelle il ne devrait pas accéder. Notre solution, temporaire et non idéale, a alors consisté à attraper toutes les erreurs de segmentation à la sortie de la fonction `main` dans le code `C++` et à les ignorer : puisque l'exécution du logiciel est terminée et que cette erreur survient en tout dernier lieu, elle peut être ignorée.

6.5 Valeur du diviseur d'espace libre (*free space divisor*)

Comme mentionné en début de chapitre, il est important de pouvoir contrôler explicitement les intervalles des collectes du *GC*. Pour ce faire, une solution consiste à modifier la fréquence des collectes du *GC Boehm* au moyen de la fonction `GC_set_free_space_divisor`.

[68] En fait, dans *Raspbian*, seul le code des tests unitaires *Catch2* provoque une erreur de segmentation.

La description de cette fonction provenant du commentaire associé dans le code source du *GC* est la suivante :

> *We try to make sure that we allocate at least N/`GC_free_space_divisor` bytes between collections, where N is twice the number of traced bytes, plus the number of untraced bytes (bytes in "atomic" objects), plus a rough estimate of the root set size. N approximates GC tracing work per GC. The initial value is `GC_FREE_SPACE_DIVISOR`. Increasing its value will use less space but more collection time. Decreasing it will appreciably decrease collection time at the expense of space.*
>
> Commentaire de la fonction `GC_set_free_space_divisor`, fichier `gc.h`, l. 315

Donc, modifier la valeur du diviseur d'espace libre (un entier non nul) fait en sorte que le *GC* effectue plus ou moins de collectes durant un certain laps de temps : avec une grande valeur de diviseur, le *GC* effectue une plus grande quantité de collectes — quoiqu'une valeur supérieure à 1000 semble avoir peu d'impact. Par défaut, la valeur de ce diviseur est de 3.

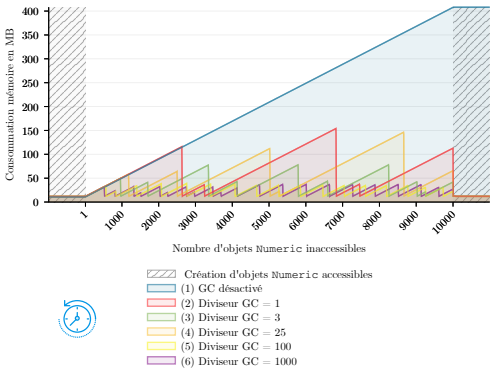


Figure 6.7 Diviseur dans *Linux* (*détails*).

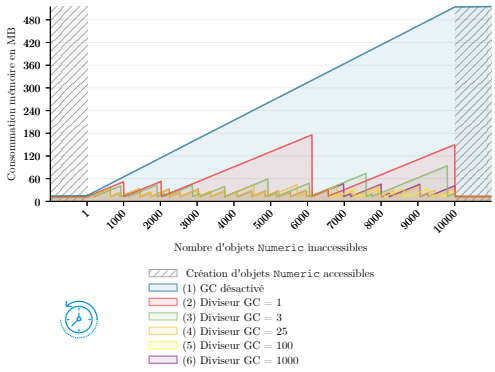


Figure 6.8 Diviseur dans *Windows* (*détails*).

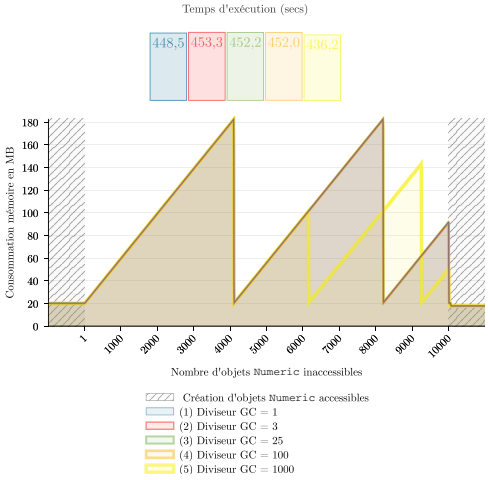


Figure 6.9 Diviseur dans *Linux* (nouvelle version, *détails*).

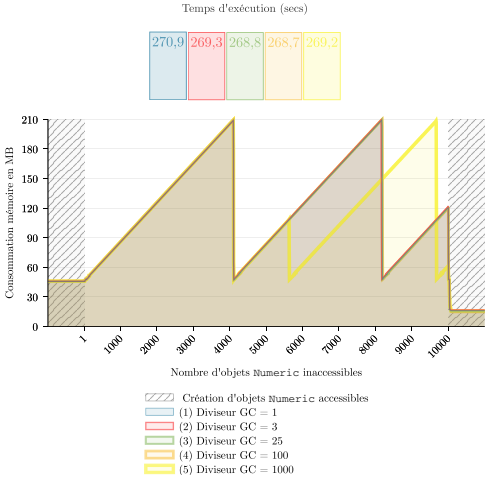


Figure 6.10 Diviseur dans *Windows* (nouvelle version, *détails*).

On constate un problème dans les figures 6.7 et 6.8 : les collectes ne sont pas effectuées de façon identique de plateforme en plateforme. Ce détail fait en sorte que la *condition 2* n'est, pour l'instant, pas respectée ; elle le deviendra, par contre, avec la solution proposée à la section 7.4.

En outre, la modification de la valeur du diviseur ne fonctionne plus correctement dans la version actuelle du projet, ce qu'on peut constater dans les figures 6.9 et 6.10 — c'est-à-dire que la modification de la valeur du diviseur semble avoir peu ou pas d'effet.

Puisque les graphes des tests précédents montrent que l'utilisation du diviseur du *GC* semble avoir peu d'effet sur le contrôle des collectes, nous avons donc cherché une autre solution, laquelle sera présentée au *chapitre suivant* (section 7.4).

6.6 Seuil de consommation mémoire (*memory threshold*)

Puisqu'il est difficile d'ajuster le comportement du *GC* en cours d'exécution et qu'une consommation mémoire excessive risque de faire planter le logiciel ou l'ordinateur, nous avons mis en place une solution préventive s'exécutant implicitement du point de vue du programmeur : surveiller en tout temps la consommation mémoire et forcer une collecte si un certain seuil est atteint.

Plusieurs modes de spécification du seuil ont été implémentés :

0) Désactivé ;

1) La mémoire libre est plus petite qu'un pourcentage de la mémoire totale ;

2) La mémoire libre est plus petite qu'un pourcentage de la mémoire libre plus la mémoire consommée par le logiciel ;

3) La mémoire consommée par le logiciel est plus grande qu'un pourcentage de la mémoire libre ;

4) La mémoire consommée par le logiciel est plus grande qu'un pourcentage de la mémoire totale ;

5) La mémoire consommée par le logiciel est plus grande qu'une quantité spécifiée de mégaoctets.

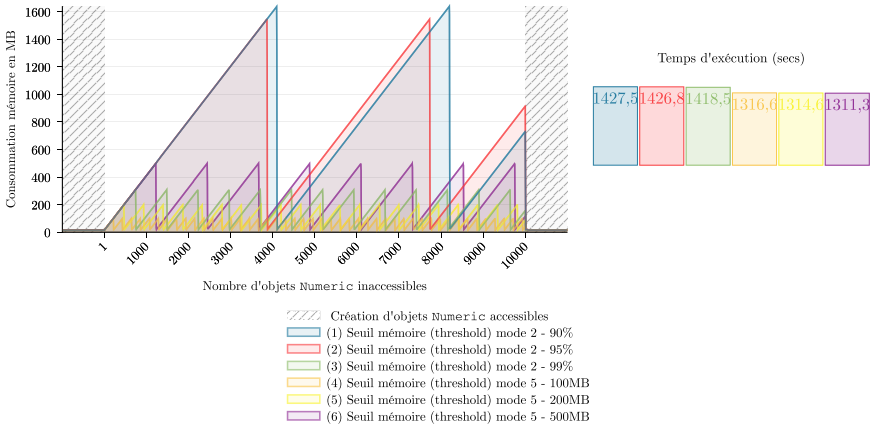


Figure 6.11 Seuils de consommation mémoire dans *Linux* (*détails*).

Selon nos expérimentations, c'est le mode 2 qui semble être le plus efficace et c'est ce dernier qui est utilisé par défaut. La figure 6.11 permet de visualiser le comportement du *GC* avec différents pourcentages ou quantités de mégaoctets dans les modes 2 et 5.

Ce seuil de consommation mémoire se veut une fonctionnalité de sûreté (*fail-safe*) et est utilisé en tout temps par le logiciel. Cependant, avec la solution introduite à la section 7.4, cette protection est maintenant rarement utile : elle ne s'active qu'en cas de bogue et évite le plantage de l'ordinateur.

6.7 Mode incrémentiel (*incremental mode*)

Le *GC Boehm* peut aussi fonctionner en mode incrémentiel (`GC_enable_incremental`). La fréquence des collectes dans ce mode peut être ajustée avec la fonction `GC_set_full_freq`. Bien que dans ce mode les collectes sont appelées plus fréquemment, on peut constater dans les figures 6.12 et 6.13 que cela ne semble pas affecter les performances.

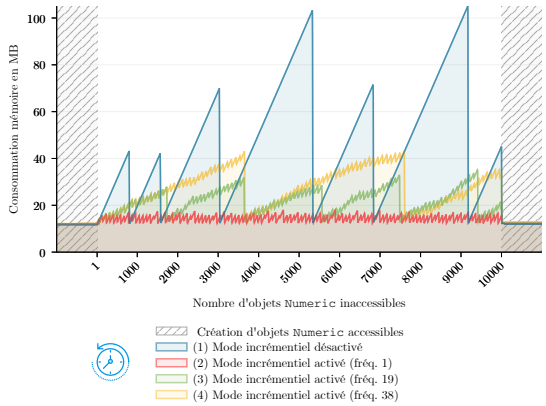


Figure 6.12 Mode incrémentiel *Linux* (*détails*).

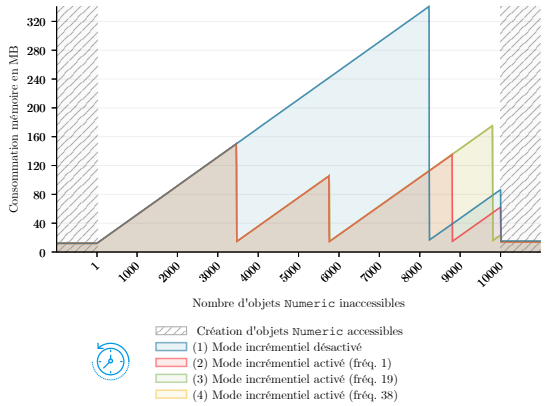


Figure 6.13 Mode incrémentiel *Windows* (*détails*).

Par contre, ce mode ne convient pas à toutes les structures mémoire, tel qu'indiqué dans le commentaire de la fonction `GC_enable_incremental`, et ne peut être désactivé une fois activé :

> *Enable incremental/generational collection. Not advisable unless dirty bits are available or most heap objects are pointer-free (atomic) or immutable. [...] For best performance, should be called as early as possible. [...]*
>
> Commentaire de la fonction `GC_enable_incremental`, fichier `gc.h`, l. 830

Cependant, dans la structure mémoire courante de notre logiciel, tous les pointeurs contenus dans les objets `Numeric` sont libérés par les destructeurs de ces derniers, donc le point « *unless [...] most heap objects are pointer-free* » n'est pas problématique. Il semblait alors possible d'utiliser ce mode et il aurait pu être intéressant pour rendre le langage moins gourmand en mémoire. Toutefois, il se comporte différemment de plateforme en plateforme, comme on le constate dans les figures 6.12 (*Linux*) et 6.13 (*Windows*).

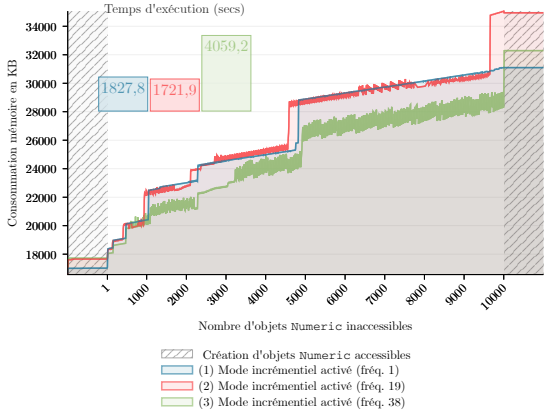


Figure 6.14 Problème de fuites mémoire du mode incrémentiel[69] (*détails*).

Pire encore, avec des objets `Numeric` à grandes précisions, le mode incrémentiel engendre des fuites mémoire, comme on le constate, dans la figure 6.14, lors de la deuxième phase de création d'objets `Numeric` accessibles.

À cause des désavantages exposés dans cette section, bien que toujours présent, le mode incrémentiel n'est pas utilisé dans notre logiciel.

[69] Utilise π à une précision de 100 000 décimales, contrairement à celle de 10 000 utilisée dans les figures 6.12 et 6.13.

6.8 Taille du tas (*heap size*)

Le *GC Boehm* maintient son propre tas (*heap*) pour gérer les allocations des espaces mémoire qu'il doit monitorer. Il semblait intéressant de changer la taille de ce tas de façon à modifier le comportement du *GC*. Les fonctions `GC_expand_hp` et `GC_set_max_heap_size` permettent d'ajuster la taille de ce tas.

Néanmoins, quelques expérimentations ont permis de conclure que la modification de cette taille affecte peu le comportement du *GC*. Cette fonctionnalité demeure cependant disponible dans le logiciel, car elle pourrait s'avérer intéressante à des fins d'optimisation — c'est-à-dire qu'elle pourrait éviter des ajustements de taille de tas inutiles.

6.9 Allocateurs mémoire personnalisés (*custom memory allocators*)

Après ces diverses expérimentations, puisque les *conditions 2*, *4* et *7* n'étaient toujours pas respectées, il nous fallait trouver une solution pour contrôler les intervalles des collectes et faire en sorte que le *GC* se comporte de façon similaire sur toutes les plateformes. Nous avons alors compris l'importance des allocateurs mémoire personnalisés (*custom memory allocators*) des différentes bibliothèques arithmétiques :

- ***GMP*** et ***MPFR*** (Granlund et al., 2020, sect. *Custom Allocation*) ;
- ***Mpdecimal*** (Krah, 2008, sect. *Custom Allocation Functions*) ;
- ***Boost.Multiprecision*** **`cpp_dec_float`**
  (Maddock et Kormanyos, 2022, sect. *Floating-Point Numbers/*`cpp_dec_float`).

En `C++`, les allocateurs mémoire personnalisés sont des classes permettant de redéfinir les méthodes d'allocation/déallocation mémoire par défaut (c.-à-d., `new`/`delete` et `malloc`/`free`) des conteneurs (*containers*) de la *STL* dans le but d'optimiser la gestion mémoire (vikas2gqb5, 2024). Ce sont des classes définies par le programmeur contenant au minimum les méthodes `allocate` et `deallocate` — elles doivent être passées au *template* des conteneurs de la *STL* lors de leur instanciation. Toutes les bibliothèques multiprécision utilisées par notre projet permettent la redéfinition de leurs allocateurs mémoire et permettent ainsi de modifier leur gestion de la mémoire.

Comme indiqué à la section 6.4, il n'était pas possible de faire en sorte que le *GC* prenne en charge les ressources mémoire allouées par les objets des bibliothèques d'arithmétique multiprécision puisque leurs

allocations mémoire ne sont pas visibles et pas gérées par le *GC*. Or, les allocateurs mémoire personnalisés règlent ce problème en permettant de redéfinir les fonctions d'allocation et de désallocation mémoire des bibliothèques arithmétiques — ce qui permet de rendre ces allocations mémoire visibles au *GC*.

En théorie, s'il est possible de contrôler la quantité d'allocations mémoire visibles par le *GC*, il est envisageable de parvenir à modifier les intervalles des collectes — comme nous l'avions infructueusement tenté en modifiant la valeur du *diviseur d'espace libre*. Puisque l'implémentation des allocateurs mémoire est tributaire des décisions prises lors de leur ajout aux classes `Numeric`, elle sera présentée au chapitre VII. Toutefois, il est dès maintenant possible d'affirmer qu'en utilisant les allocateurs mémoire personnalisés, les *conditions 2*, *4* et *7* sont finalement respectées. Une explication de ce résultat sera présentée à la section 7.4 (figures 7.14, 7.15 et 7.16).

6.10 Bilan : utilisation du *GC Boehm* et respect des conditions attendues du *GC*

Pour clore ce chapitre, voici nos conclusions quant à l'utilisation du *GC Boehm* :

- Le *GC Boehm* n'appelle les destructeurs des objets qu'il détruit que s'ils héritent de la classe `gc_cleanup` et non de la classe `gc`.

- Seules les *conditions 2*, *4* et *7*, énumérées en début de chapitre, n'étaient pas respectées avant la solution mentionnée à la section 6.9. C'est-à-dire que le *GC* ne se comportait pas de manière identique de plateforme en plateforme, qu'il était impossible de contrôler les intervalles de collectes et de calculer précisément la quantité de mémoire utilisée par les objets `Numeric`. Pour combler ces lacunes, une autre solution a été mise en place, soit l'ajout des *allocateurs mémoire*, décrite au chapitre VII. La *condition 8* (support *multithreads*), bien que respectée en utilisant le *GC* tel quel, a cependant dû être réévaluée lors de la mise en œuvre des allocateurs mémoire. Ce dernier point sera expliqué en détail à la section 7.5 du prochain chapitre.

- La compréhension du fonctionnement du *GC Boehm* et de son utilisation a été complexe, notamment parce que la documentation disponible ne couvre pas tous les détails et qu'il est souvent nécessaire de recourir aux *commentaires* du code source pour comprendre certains points. Mentionnons toutefois quelques points importants spécifiés dans la documentation en ligne du *GC Boehm* :

  – Il est recommandé de ne pas combiner des allocations standards avec des allocations gérées par le *GC* :

  > *It is usually best not to mix garbage-collected allocation with the system* `malloc-free`*. If you do, you need to be careful not to store pointers to the garbage-collected heap in memory allocated with the system* `malloc`*.*
  > *In the case of* `C++`*, you need to be especially careful not to store pointers to the garbage-collected heap in areas that are not traced by the collector. The collector includes some alternate interfaces to make that easier.*
  >
  > *Using the Garbage Collector: A simple example*

  Par contre, il nous semble impossible d'éviter une telle situation dans tout projet le moindrement complexe. Outre les problèmes d'erreurs de segmentation sur certaines plateformes, pouvant êtres expliquées par ce détail, le *GC* ne semble éprouver aucun problème à gérer la mémoire.

  – Des précautions particulières additionnelles doivent être prises lors de l'utilisation de la *Standard Template Library* (*STL*). Selon la documentation du *GC Boehm*, certaines structures de données, telles que celles fournies par la *STL*, sont problématiques[70]. Pour cette raison, il est nécessaire d'instancier explicitement les conteneurs de la *STL* (*STL containers*) avec la classe `gc_allocator` pour que le *GC* soit en mesure de gérer correctement leur contenu.

[70] `https://hboehm.info/gc/gcinterface.html` (*archive*)

CHAPITRE VII
CLASSE ET SOUS-CLASSES POUR LA MISE EN OEUVRE DE L'ARITHMÉTIQUE MULTIPRÉCISION

Dans ce chapitre, nous décrivons la *hiérarchie des objets pour la mise en œuvre du langage Kpsilon* ainsi que les différentes sous-classes `Numeric` et leur implémentation. Il y est aussi question de la notion d'*objets éphémères*, du *support multithreads*, des *allocateurs mémoire* et des méthodes de *dispatch dynamique* des sous-classes `Numeric`. Finalement, nous présentons et analysons les résultats de quelques *comparaisons* de *performances* et de *précisions* entre les différents types `Numeric`, ainsi qu'avec d'autres logiciels.

La figure 7.1 présente la légende et des exemples de différents éléments des diagrammes *UML* de classes utilisés dans ce chapitre. Et en guise de rappel, la figure 7.2 présente le diagramme de classes pour les différents types numériques de *Kpsilon*.

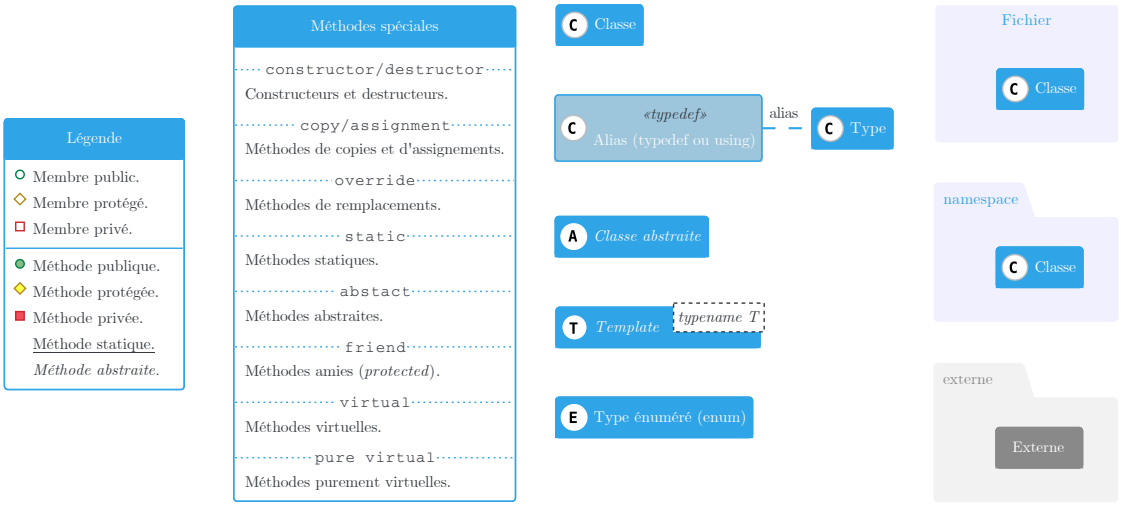


Figure 7.1 Légende et éléments des diagrammes *UML* de classes.

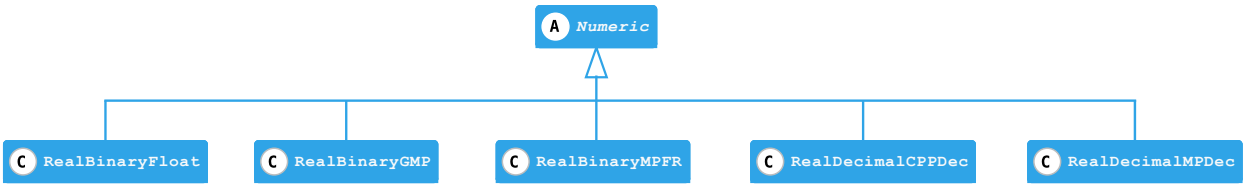


Figure 7.2 Diagramme *UML* des sous-classes `Numeric` (rappel).

7.1 Choix conceptuels et hiérarchie des classes d'implémentation

Conceptuellement, les instances des types numériques du langage *Kpsilon* (figure 7.2) sont des *valeurs* (Maclennan, 1982), puisqu'il s'agit de nombres, immuables et sans identité. La mise en œuvre C++ de ces valeurs numériques repose évidemment sur l'utilisation d'objets — donc des *« value objects »*[71].

Toutes les classes des types numériques du langage *Kpsilon* sont des descendantes de la *classe abstraite* `Numeric`. Ce choix de conception a été pris pour permettre d'effectuer des opérations mathématiques entre différents types, et ce, tant dans le code C++ pour l'implémentation du langage et interpréteur *Kpsilon* que pour un utilisateur de l'interpréteur.

Pour l'instant, seuls des types pour valeurs à virgule flottante ont été implémentés. Toutefois, grâce à la mise en œuvre basée sur le *dispatch dynamique* (section 7.6), d'autres types numériques pourront facilement être ajoutés, par exemple, des entiers, des rationnels et des nombres complexes.

```
Numeric& n1 = *Numeric::create(ObjType::RealBinaryGMP,     10,   "1");
Numeric& n2 = *Numeric::create(ObjType::RealDecimalMPDec, 100,   "3");

Numeric& res = n1 / n2; // res = 1/3 = 0.3333333333
```

Listing 7.1 Exemple C++ du calcul de $1 \div 3$ entre différents types `Numeric`.

Le listing 7.1 présente du code C++ utilisant des `Numeric` pour calculer 1 en base 2 (`RealBinaryGMP` ou `b1`) à une précision de 10 décimales divisée par 3 en base 10 (`RealDecimalMPDec` ou `d2`) à une précision de 100 décimales. Le résultat de cette opération est $0{,}333\,333\,333\,3$ ($0{,}\overline{3}$) en base 2 (`RealBinaryGMP` ou `b1`) à une précision de 10 décimales. (Voir aussi la capture 5.3 (chapitre V) pour un exemple dans l'interpréteur *Kpsilon*.)

[71] Pour le moment, l'interpréteur peut créer plus d'une copie d'une même valeur ; une optimisation sera ajoutée ultérieurement pour produire des valeurs uniques.

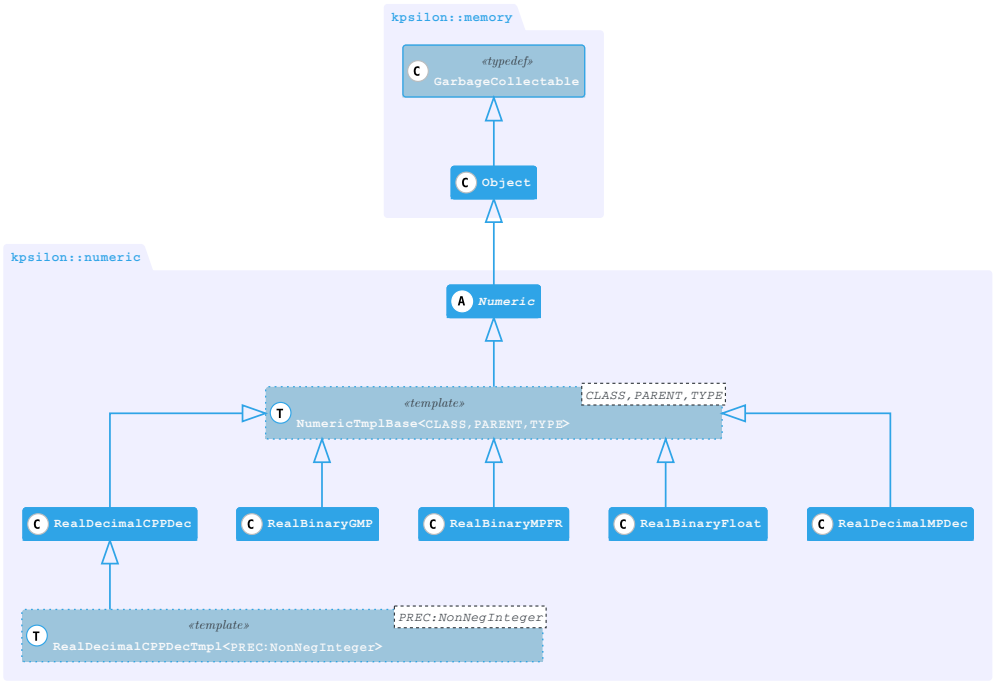


Figure 7.3 Diagramme de classes (simplifié) pour l'implémentation des `Numeric`.

La figure 7.3 présente un diagramme des classes d'implémentation (simplifié) pour la classe `Numeric` et ses sous-classes. Les entités `Numeric` étant des nombres, il est naturel de vouloir utiliser des définitions d'opérateurs (*operator overloading*). L'idée est donc d'utiliser le *polymorphisme* du `C++` pour manipuler, lors des calculs avec des opérateurs arithmétiques, des références d'objets `Numeric` et d'implémenter ces opérateurs par l'intermédiaire de méthodes de ***dispatch dynamique*** (*dynamic dispatch*, section 7.6).

Bien qu'en `C++` il soit recommandé de retourner une copie d'objet, et non une référence ou un pointeur dans les méthodes de définition d'opérateur (CalTech, 2007), cette approche n'est pas compatible avec le polymorphisme. C'est-à-dire que pour appeler les méthodes spécifiques (*method overriding*) à chacune des classes enfants (*sous-classes* `Numeric`) à partir d'un parent commun (super-classe `Numeric`), il est nécessaire de définir ces méthodes comme étant *virtuelles* — sans quoi, ce sont toujours les versions des méthodes du parent qui sont appelées. Or, pour utiliser les méthodes virtuelles d'un objet `Numeric` reçu en paramètre par les méthodes de définition d'opérateur, l'objet `Numeric` doit être reçu sous forme de pointeur ou de référence et non par copie, car le *dispatch dynamique* des méthodes virtuelles ne fonctionne que par pointeurs ou références. De plus, il est impossible de passer une copie d'un objet `Numeric` en paramètre, puisque la classe `Numeric` est une classe abstraite et qu'il est impossible d'instancier ce type de classe.

De plus, dans une optique d'optimisation, retourner une référence ou un pointeur s'avère plus performant. En effet, retourner une copie d'objet requiert la création d'un nouvel objet, ce qui consomme une certaine quantité de cycles *CPU*. Néanmoins, pour pouvoir retourner une référence ou un pointeur vers l'objet résultant, ce dernier doit être créé dynamiquement par la méthode de définition d'opérateur. Cette approche ne respecte pas l'idiome (ou patron) de programmation *RAII* (*Resource Acquisition Is Initialization*) (Stroustrup, 1994; Stroustrup, 2013) — c'est-à-dire que l'objet ayant créé le nouvel objet devrait être responsable de le détruire lors de sa propre destruction. Toutefois, puisque tous les objets `Numeric` sont gérés et implicitement libérés par le *GC Boehm*, la non-adoption d'un patron *RAII* n'est aucunement problématique.

Cependant, lors de manipulations d'objets `Numeric`, il est nécessaire de toujours garder à l'esprit que les valeurs retournées par les opérateurs arithmétiques sont des références de type `Numeric`. En effet, leur utilisation est délicate dans certaines situations et peut être problématique si le programmeur (`C++`) ne saisit pas cette subtilité. Pour cette raison, une petite « *API* » a été ajoutée pour faciliter leur utilisation[72]. Il faut aussi mentionner que la manipulation par références d'objets créés dynamiquement peut engendrer, lors d'un enchaînement d'opérations arithmétiques, un problème de consommation excessive de mémoire — ce problème sera décrit lorsque nous introduirons les ***objets éphémères*** (section 7.3).

Deux versions des sous-classes `Numeric` ont été implémentées. La première version nécessitait la redéfinition de plusieurs méthodes communes lors de l'ajout d'un nouveau type `Numeric` — i.e., lors de l'ajout d'une nouvelle sous-classe. Pour éviter ce problème, la deuxième version — discutée dans le présent chapitre — utilise une classe *template* intermédiaire, ***`NumericTmplBase`***, pour regrouper les méthodes communes.

[72] Bien que cette « *API* » ne soit pas utilisée dans le code `C++` du logiciel *Kpsilon*, elle est utilisée (méthode `generate`) dans le pseudocode présenté dans le listing 7.4 (section 7.6).

7.1.1 Classe `Object` et classe abstraite `Numeric`

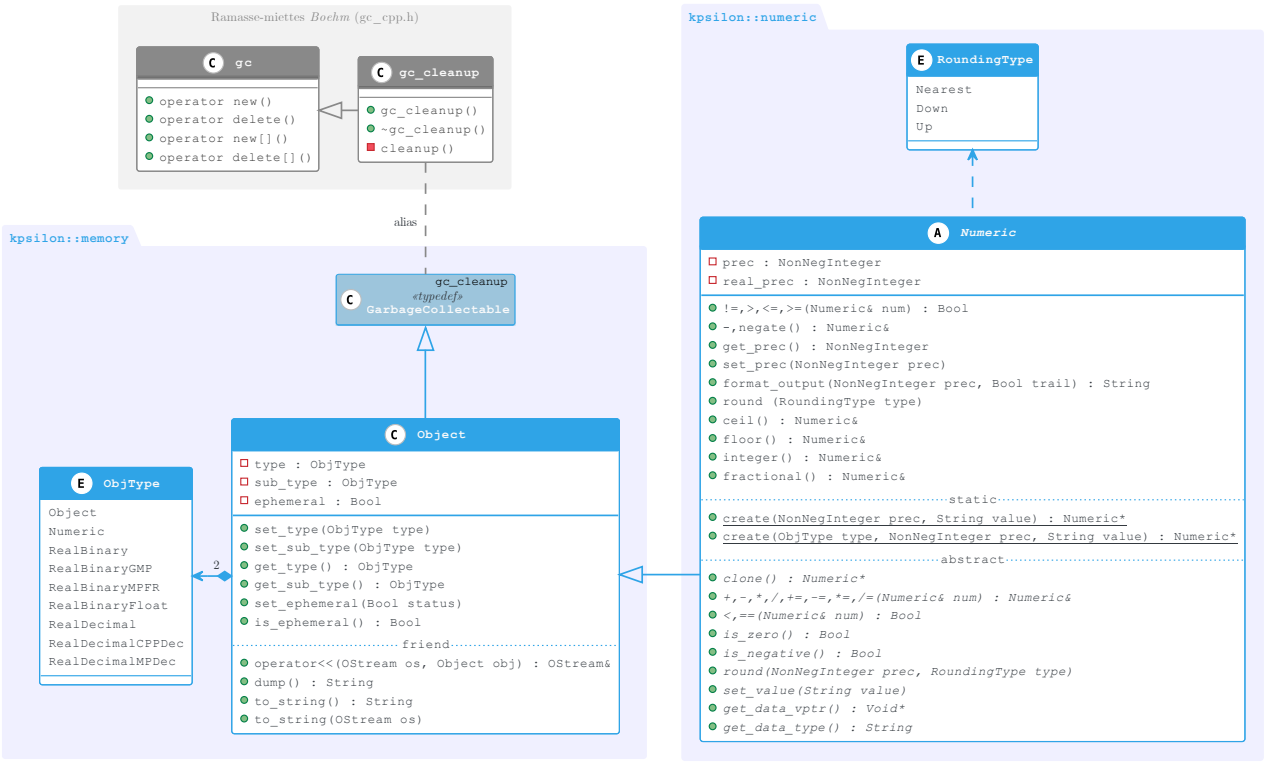


Figure 7.4 Diagramme *UML* de la classe `Numeric` et de leurs ancêtres.

La figure 7.4 présente un diagramme *UML* des classes `Object` et `Numeric`.

Tous les objets du langage *Kpsilon* sont des descendants de la classe **`Object`**. Cette dernière hérite de la classe **`GarbageCollectable`** (un alias de la classe *`gc_cleanup`*, chapitre VI) et tous ses descendants sont donc pris en charge par le *GC*. La classe `Numeric` est la classe mère de toutes les classes des types numériques du langage *Kpsilon* — elle est descendante directe de la classe `Object`. Il faut noter que dans le code `C++` de l'implémentation du langage symbolique (ou toute autre fonctionnalité de notre logiciel), le *GC* ne prend en charge que les objets héritant de la classe `Object` ; toutes les autres structures de données sont manipulées par des pointeurs traditionnels (lors d'allocations/désallocations locales) ou intelligents (chapitre III).

Un patron de conception de type « *factory method* » (Gamma *et al.*, 1995) est utilisé pour la création des objets `Numeric`. Ce patron est couramment utilisé pour manipuler des sous-classes (enfants) au moyen de leur super-classe (parent) commune : il s'agit de définir une fonction de création qui, par une forme de *dispatch*, retourne un pointeur (de type super-classe) vers l'instance de la sous-classe désirée. Pour ce faire, la

classe `Numeric` contient la méthode *`create(type,prec,val)`* qui est responsable de la création de toutes les valeurs de type `Numeric`. Le résultat réfère à un objet de type « `type` » (un des types `Numeric` de l'*enum* *`ObjType`* de la figure 7.4) ayant une valeur numérique « `val` » et une précision « `prec` ».

7.1.2 Classe *template* `NumericTmplBase<CLASS,PARENT,TYPE>`

Pour simplifier l'ajout de nouveaux types `Numeric`, une classe intermédiaire a été ajoutée sous forme du *template* **`NumericTmplBase<CLASS,PARENT,TYPE>`**. Son rôle est de définir les méthodes communes aux différents types pour éviter de les redéfinir à chaque ajout d'un nouveau type `Numeric`.

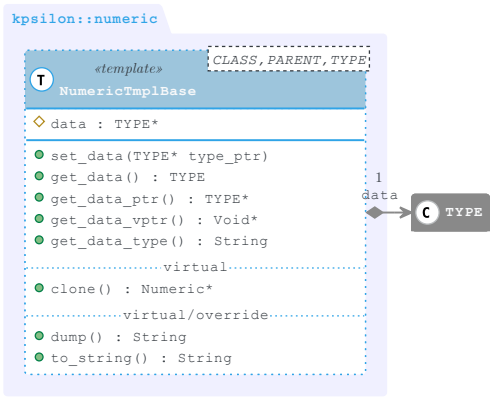


Figure 7.5 Diagramme *UML* de la classe *template* `NumericTmplBase`.

La figure 7.5 présente le diagramme *UML* de la classe *template* `NumericTmplBase<CLASS,PARENT,TYPE>`. Un *template* doit être utilisé, car le nouveau type `Numeric` doit contenir et manipuler des objets de type `TYPE` — ce dernier identifie la bibliothèque d'arithmétique multiprécision utilisée par le nouveau type. L'identifiant `CLASS`, quant à lui, indique le nom de la classe du nouveau type `Numeric` et l'identifiant `PARENT` est la classe dont il hérite (`Numeric` ou *`RealDecimalCPPDec`*).

7.2 Sous-classes `Numeric`

Les classes décrites ci-dessus sont communes à tous les types `Numeric` de *Kpsilon*. Nous allons maintenant décrire les différents sous-types `Numeric` disponibles dans la mise en œuvre actuelle.

7.2.1 Classes `RealBinaryGMP` (`b1`) et `RealBinaryMPFR` (`b2`)

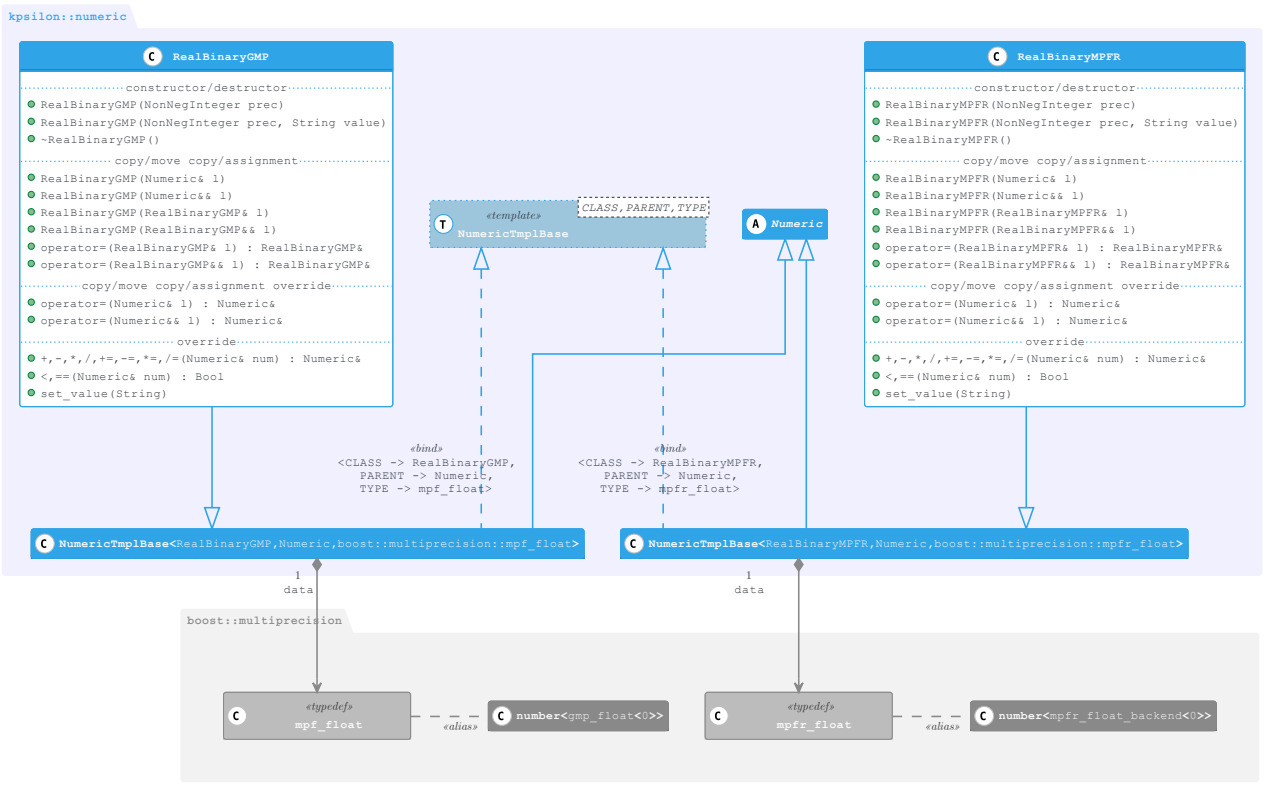


Figure 7.6 Diagramme *UML* des classes `RealBinaryGMP` et `RealBinaryMPFR` et de leurs ancêtres et descendants.

La figure 7.6 présente le diagramme *UML* des classes `RealBinaryGMP` et `RealBinaryMPFR`. Ces classes utilisent respectivement les types *`gmp_float`* (*GMP* ou *MPIR*[73]) et *`mpfr_float`* (*GNU MPFR*) présentés à la section 4.2. Ce sont des types en base 2 ayant une précision maximale de 4,2 milliards de décimales.

Ces deux classes sont quasiment identiques, donc il est possible de modifier la classe `RealBinaryGMP` et d'utiliser un script pour générer automatiquement la classe `RealBinaryMPFR`. Un *template* aurait pu être utilisé, mais il n'aurait que complexifié le code. Cependant, comme expliqué à la section 7.7, le type `RealBinaryMPFR` ne se comporte pas comme prévu, à savoir qu'il est plus lent que le type `RealBinaryGMP` dans certaines situations (jusqu'à 100 fois). Pourtant, ils sont pratiquement identiques — un détail doit nous avoir échappé, mais nous n'avons pas eu le temps d'explorer plus à fond.

[73] Sous *Windows*, c'est plutôt la bibliothèque compatible *MPIR* (*Multiple Precision Integers and Rationals*) qui est habituellement utilisée : `http://www.mpir.org/` (archivé).

Comme mentionné précédemment (section 4.2), tout nouveau projet devrait utiliser la bibliothèque *MPFR* plutôt que *GMP*. C'est pour cette raison que le type `RealBinaryMPFR` est le type binaire utilisé par défaut.

Bien que plus lent que le type `RealBinaryFloat`, discuté ci-dessous, les types `RealBinaryGMP` et `RealBinaryMPFR` ont l'avantage de supporter de grandes précisions en base 2, et sont généralement (hormis le problème avec `RealBinaryMPFR`) plus rapides que les types `RealDecimalCPPDec` et `RealDecimalMPDec` (base 10, section 7.7).

7.2.2 Classe `RealBinaryFloat` (b3)

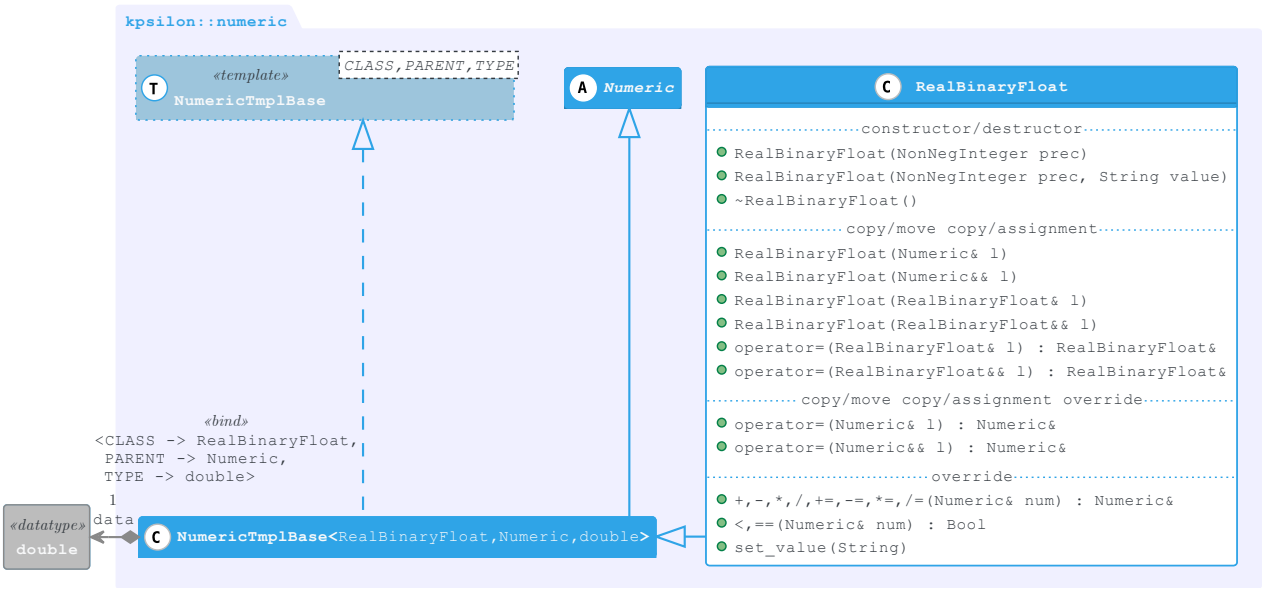


Figure 7.7 Diagramme *UML* de la classe `RealBinaryFloat` et de ses ancêtres et descendants.

La figure 7.7 présente le diagramme *UML* de la classe `RealBinaryFloat`. Cette classe utilise des `double` 64 bits (base 2). Elle est limitée à des précisions d'au plus *17 décimales* [46], mais elle a l'avantage d'être optimisée au niveau matériel et d'être la plus rapide de toutes les sous-classes `Numeric` (section 7.7).

Cette classe pourra être utilisée lorsque la rapidité d'exécution prévaut sur la précision des résultats. Par exemple, l'*interpréteur Kpsilon* pourrait utiliser implicitement ce type s'il détecte que la précision des résultats n'est pas importante — pensons à une variable d'itération dans une boucle `for`[74].

[74] Quoique dans le cas d'un itérateur de type `i++`, ce sera un entier de type `long` (64 bits) qui sera utilisé ; cependant, pour l'instant, les types entiers n'ont pas été implémentés.

En fin de chapitre, nous verrons, dans le tableau 7.2 de comparaisons de performances, que le type `RealBinaryFloat` est plus lent que les types `double` 64 bits de *Ruby* (`Float`) et de *Python* (`float`). Cette différence est causée par les choix d'implémentation de la hiérarchie des sous-classes `Numeric`. Il s'agit d'un compromis qui a été fait puisque *Kpsilon* vise à être un système de calcul formel (section 1.3), dont le but premier est de produire des résultats exacts à une précision bien définie.

7.2.3 Classes `RealDecimalCPPDec` (`d1`) et `RealDecimalCPPDecTmpl<PREC>`

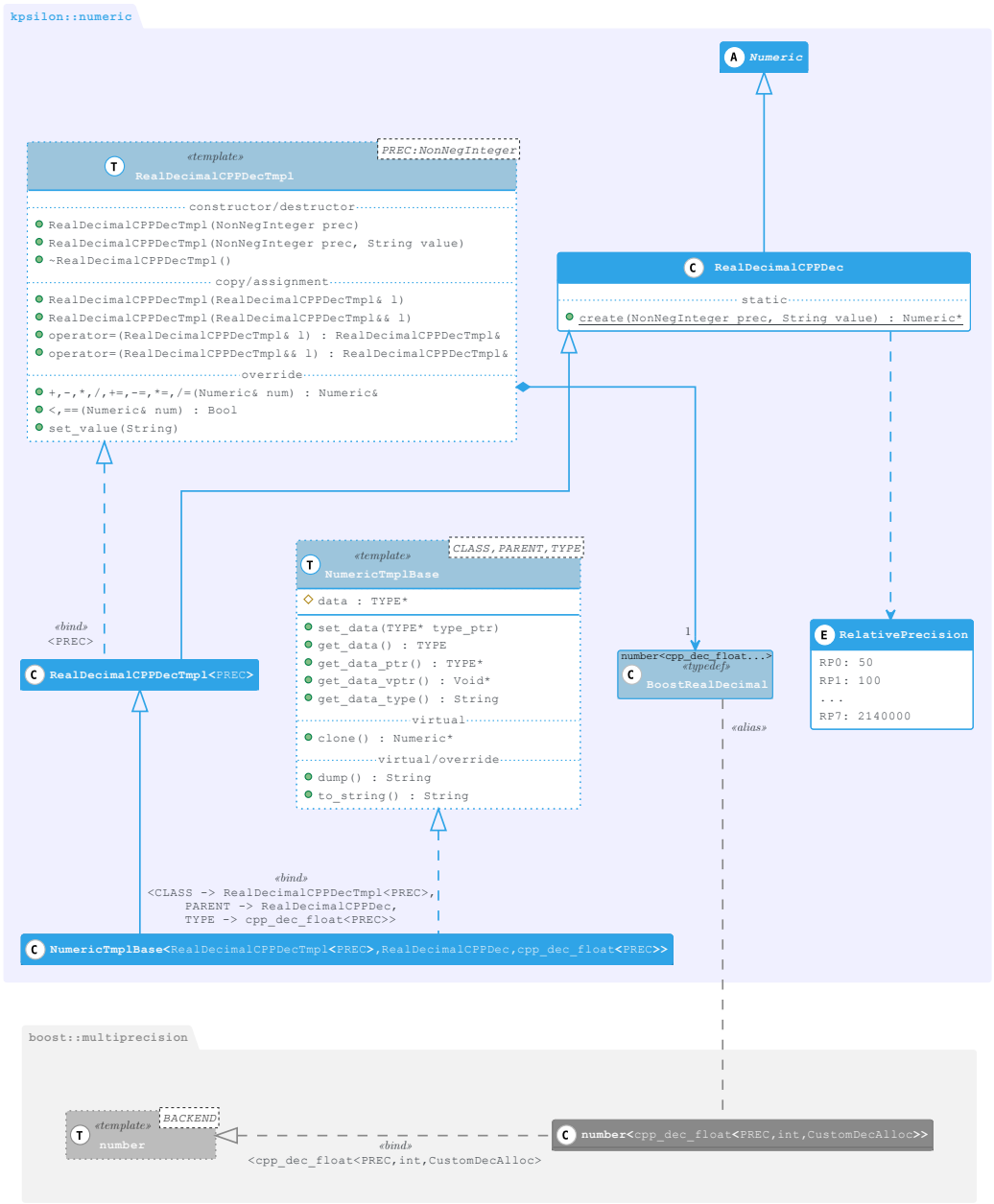


Figure 7.8 Diagramme *UML* de la classe `RealDecimalCPPDec` et de ses ancêtres et descendants.

La figure 7.8 présente le diagramme *UML* de la classe `RealDecimalCPPDec`. Ce type a été le plus complexe à implémenter. Contrairement aux autres types de la bibliothèque *Boost.Multiprecision*, la précision des *`cpp_dec_float`* ne peut pas être définie dynamiquement. Le type `cpp_dec_float` est en fait une classe *template* qui doit être instanciée au moment de la compilation du code `C++`, et ce, pour chacune des précisions désirées. Or, il était impensable d'instancier une *template* pour chacune des précisions possibles.

Notre solution est de n'instancier qu'un certain nombre de *templates* pour des précisions *relatives*, `RP0` à `RP7`[75]. Ces différentes précisions sont spécifiées par les valeurs énumérées (`enum`) `RelativePrecision`. Le choix de ces précisions relatives est basée sur la *suite de Fibonacci*. En effet, il semble que la suite de Fibonacci soit appropriée pour l'approximation de valeurs croissantes lorsque les petites valeurs sont plus sujettes d'être choisies (Brilliant.org, 2017; Cohn, 2022, *Fibonacci heap*).

Pour éviter la répétition de code lors de l'instanciation des différents *templates* `RealDecimalCPPDecTmpl <PREC>` définissant les précisions relatives et leur utilisation dans la méthode de *dispatch dynamique* (section 7.6) de la classe `RealDecimalCPPDec`, nous avons mis en place une forme d'automatisation de génération de code `C++` pour chacune des précisions — en l'occurrence, une forme de boucle pour itérer sur les éléments d'une liste contenant les valeurs des différentes précisions relatives.

Bien que le langage du *préprocesseur `C++`* soit *Turing-complet* et qu'il est théoriquement possible d'ajouter des boucles au code du préprocesseur `C++` (Vandevoorde *et al.*, 2017), leur implémentation est complexe et rend le code difficile à comprendre. Nous avons tout d'abord expérimenté avec la bibliothèque *Boost.Preprocessor*, qui supporte les boucles au niveau du préprocesseur `C++`, mais le code généré est quasi impossible à comprendre et déboguer (comporte de nombreuses classes *templates* ayant des noms cryptiques).

Après diverses recherches, nous avons choisi d'implémenter notre propre ***préprocesseur `C++` « maison »*** en *Python* (annexe C). Ce préprocesseur est exécuté avant la phase de compilation pour générer le code désiré à partir de fichiers contenant certaines commandes en commentaires. Un exemple de son utilisation sera présenté dans le pseudocode de l'implémentation d'une méthode de *dispatch dynamique* (listing 7.4, section 7.6).

Comme on le verra à la section 7.7, le type `RealDecimalCPPDec` a une très grande précision (max. 2,1

[75] Il est possible de demander une « *granularité* » plus fine de ces précisions relatives, soit `RP0` à `RP15`, avec une option de compilation.

millions décimales), mais il est toutefois le plus lent de tous les `Numeric`. En fait, il n'est pas recommandé d'utiliser le type *Boost.Multiprecision* `cpp_dec_float` à de grandes précisions :

> *Normally* `cpp_dec_float` *allocates no memory : all of the space required for its digits are allocated directly within the class. As a result care should be taken not to use the class with too high a digit count as stack space requirements can grow out of control.* ***If that represents a problem then providing an allocator as the final template parameter causes*** `cpp_dec_float` ***to dynamically allocate the memory it needs*** *: this significantly reduces the size of* `cpp_dec_float` *and increases the viable upper limit on the number of digits at the expense of performance. However, please bear in mind that arithmetic operations rapidly become very expensive as the digit count grows :* ***the current implementation really isn't optimized or designed for large digit counts***.
>
> *Boost.Multiprecision 1.79* (Maddock et Kormanyos, 2022, sect. `cpp_dec_float`)

C'est pour cette raison que nous avons ajouté le prochain type, `RealDecimalMPDec`, et qu'il est le type en base 10 par défaut.

7.2.4 Classe `RealDecimalMPDec` (d2)

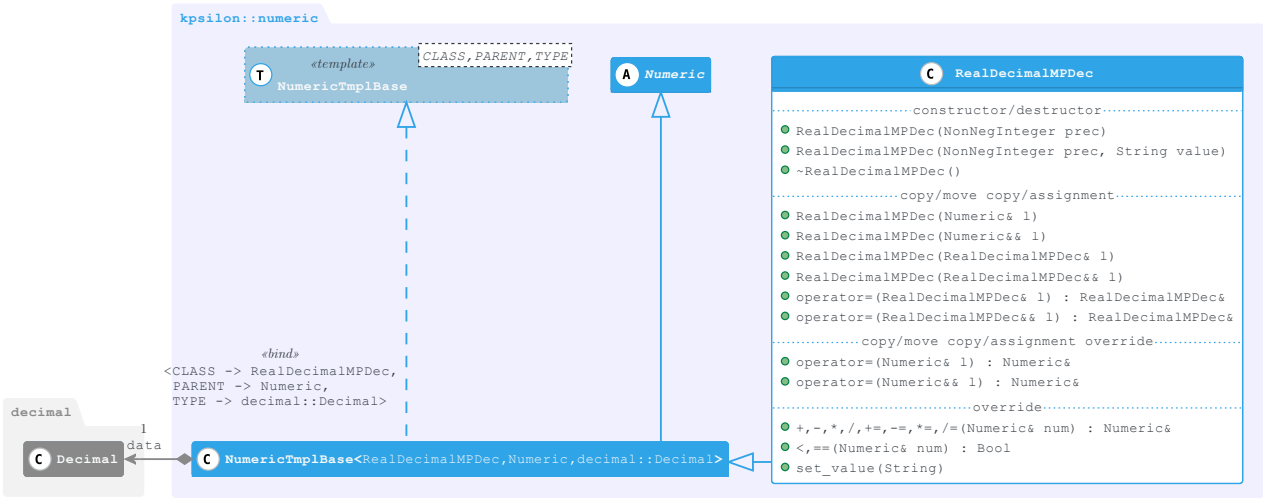


Figure 7.9 Diagramme *UML* de la classe `RealDecimalMPDec` et de ses ancêtres et descendants.

La figure 7.9 présente le diagramme *UML* de la classe `RealDecimalMPDec`. Ce type utilise la bibliothèque à précisions arbitraires en base 10 *mpdecimal* (section 4.4) et, tel que mentionné plus haut, est nettement plus rapide.

7.3 Objets éphémères

L'utilisation du polymorphisme et le retour de références d'objets `Numeric` dans les méthodes des opérateurs arithmétiques causent un problème de consommation mémoire. En effet, puisque ces objets sont créés dynamiquement et ne sont détruits qu'implicitement par le *GC*, les enchaînements d'opérations arithmétiques peuvent consommer inutilement de la mémoire pouvant être libérée. Or, les objets `Numeric` intermédiaires retournés par les méthodes des opérateurs arithmétiques ont souvent une « *durée de vie* » limitée et peuvent sans problème être détruits explicitement une fois utilisés.

Par exemple, pour effectuer le calcul $res = val_1 + val_2 + val_3$, l'*interpréteur Kpsilon* évaluera le résultat en trois étapes, soit : $num_1 = val_1 + val_2$, $num_2 = num_1 + val_3$ et $res = num_2$. Les objets `Numeric` retournés num_1 et num_2 sont des objets intermédiaires qui peuvent être détruits explicitement après utilisation. C'est pour éviter cette destruction explicite et éviter les fuites mémoire qu'en `C++` il est recommandé de retourner une copie d'objet et non une référence ou un pointeur dans les méthodes de définition d'opérateurs (CalTech, 2007). Sans l'ajout d'un dispositif supplémentaire pour libérer ces objets intermédiaires, *Kpsilon* consomme trop de mémoire et ses performances en sont réduites.

Notre solution à ce problème a été d'introduire la notion d'**objets éphémères** (*ephemeral objects*) dans la classe `Object`. Bien qu'il aurait été possible de détruire explicitement ces objets dans l'*interpréteur Kpsilon*, cette phase a été directement implémentée dans le code `C++` dans les méthodes de *dispatch dynamique* des objets `Numeric`. Les méthodes de *dispatch dynamique* peuvent détecter si un objet `Numeric` est éphémère au moyen de la méthode `Object::is_ephemeral` et le détruire au besoin[76].

La capture 7.1 présente un exemple d'examen de la mémoire au moyen des commandes « `dumpgc` » (affiche les objets détectés par le *GC*), « `gcreset` » (vide la mémoire monitorée par le *GC*) et « `ephmoff` » désactive les objets éphémères). Après avoir désactivé les objets éphémères avec « `emphoff` » (*ligne 30*), on constate (*lignes 48*, *50* et *51*) que trois objets éphémères (« `ephm` », en rouge) auraient pu être libérés. Ce n'est pas le cas lorsque les objets éphémères sont activés (par défaut), comme on le constate dans la liste affichée par « `dumpgc` » (*lignes 15* à *22*) — ils n'ont pas besoin d'être libérés par le *GC* car ils ont déjà été détruits.

[76] Au lieu de détruire un objet éphémère, il serait possible de le réutiliser pour éviter d'en créer un autre et optimiser le code. Cette optimisation n'a pas encore été faite, mais le sera éventuellement.

```
kpsilon> -0.4 / -3 + (1 + 1323) / 4213231312

 Timings         : 0.004727s real, 0.000000s CPU (n/a%)
 Memory          : adr 0x7f3c6bdd8bc0, used 20025344, free 5182664704 (bytes)
 Data memory     : object 42248 bytes, total data 3485908 bytes in 7 objects
 Numeric type    : RealDecimalMPDec, prec:100000/100000, mem:42248
 Result          : 0.13333364758144884244...
 Output timings : 0.005522s real, 0.000000s CPU (n/a%)

kpsilon> dumpgc

 Number of managed allocated memory::Object : 8

   0x7f3c6bdd8c40 -> 1323 (RealDecimalMPDec, prec:100000/100000, mem:136)
   0x7f3c6bdd8cc0 -> 1 (RealDecimalMPDec, prec:100000/100000, mem:136)
   0x7f3c6bdd8d40 -> 0 (RealDecimalMPDec, prec:100000/100000, mem:42248)
   0x7f3c6bdd8dc0 -> -0.4 (RealDecimalMPDec, prec:100000/100000, mem:136)
   0x7f3c6bdd8f40 -> -3 (RealDecimalMPDec, prec:100000/100000, mem:136)
   0x7f3c6bdd8fc0 -> 0 (RealDecimalMPDec, prec:100000/100000, mem:136)
   0x7f3c6bdd8b40 -> 4213231312 (RealDecimalMPDec, prec:100000/100000, mem:136)
   0x7f3c6bdd8bc0 -> 0.1333336476 (RealDecimalMPDec, prec:100000/100000, mem:42248)

 Total memory used by the GC allocators : 85312 bytes

kpsilon> gcreset

 8 Objects were deleted.

kpsilon> ephmoff

 Ephemeral objects are now disabled.

kpsilon> -0.4 / -3 + (1 + 1323) / 4213231312

 Timings         : 0.007690s real, 0.000000s CPU (n/a%)
 Memory          : adr 0x7f3c6bdd8780, used 20070400, free 5184167936 (bytes)
 Data memory     : object 42248 bytes, total data 4307452 bytes in 10 objects
 Numeric type    : RealDecimalMPDec, prec:100000/100000, mem:42248
 Result          : 0.13333364758144884244...
 Output timings : 0.004601s real, 0.000000s CPU (n/a%)

kpsilon> dumpgc

 Number of managed allocated memory::Object : 11

   0x7f3c6bdd8800 -> 0.0000003142 (RealDecimalMPDec, prec:100000/100000, mem:42248, ephm)
   0x7f3c6bdd8880 -> 4213231312 (RealDecimalMPDec, prec:100000/100000, mem:136)
   0x7f3c6bdd8cc0 -> 0.1333333333 (RealDecimalMPDec, prec:100000/100000, mem:42248, ephm)
   0x7f3c6bdd8900 -> 1324 (RealDecimalMPDec, prec:100000/100000, mem:136, ephm)
   0x7f3c6bdd8980 -> 1323 (RealDecimalMPDec, prec:100000/100000, mem:136)
   0x7f3c6bdd8dc0 -> -3 (RealDecimalMPDec, prec:100000/100000, mem:136)
   0x7f3c6bdd8a00 -> 1 (RealDecimalMPDec, prec:100000/100000, mem:136)
   0x7f3c6bdd8fc0 -> -0.4 (RealDecimalMPDec, prec:100000/100000, mem:136)
   0x7f3c6bdd8700 -> 0 (RealDecimalMPDec, prec:100000/100000, mem:42248)
   0x7f3c6bdd8780 -> 0.1333336476 (RealDecimalMPDec, prec:100000/100000, mem:42248)
   0x7f3c6bdd8bc0 -> 0 (RealDecimalMPDec, prec:100000/100000, mem:136)

 Total memory used by the GC allocators : 169944 bytes
```

Capture 7.1 Inspection de la mémoire avec et sans objets éphémères.

Remarque: *Dans les graphiques qui suivent, les classes sont identifiées à l'aide des couleurs suivantes :*

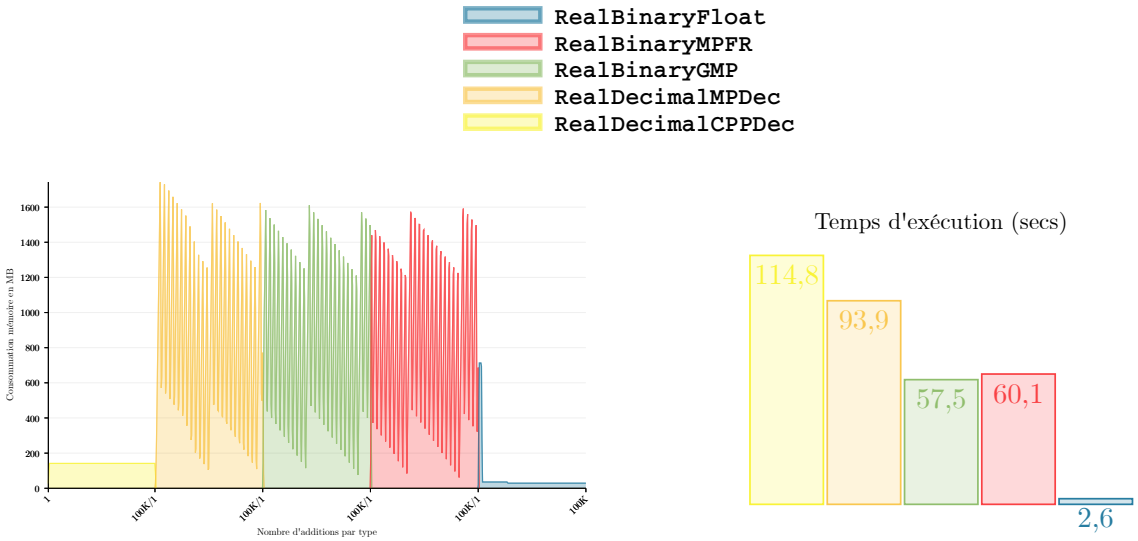


Figure 7.10 Consommation mémoire **sans** objets éphémères[77] (*détails*).

Figure 7.11 Temps **sans** objets éphémères (*détails*).

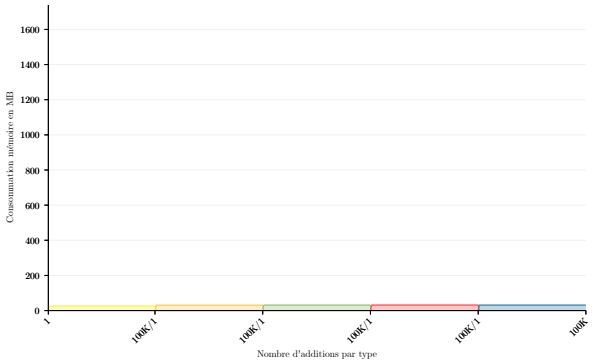


Figure 7.12 Consommation mémoire **avec** objets éphémères (*détails*).

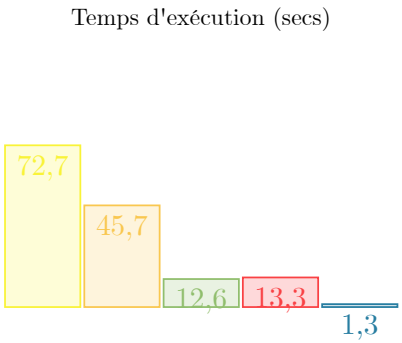


Figure 7.13 Temps **avec** objets éphémères (*détails*).

La figure 7.10 permet de visualiser le problème de consommation de mémoire excessive sans l'utilisation d'objets éphémères, figure générée par le *test des différents types* du logiciel (*Radix 2 vs 10 tests*, section B.2). Ce test effectue 100 000 additions successives de la valeur π à une précision d'un million de décimales $\left(\sum_{i=1}^{100\,000} \pi_{1M}\right)$. La valeur π est utilisée, car elle évite les différences de consommation mémoire entre les divers types `Numeric` — comme par exemple la valeur $0{,}1$, qui consomme beaucoup moins de mémoire dans

[77] Dans ce graphe, les `RealDecimalCPPDec` semblent consommer peu de mémoire, ce n'est cependant pas le cas. En effet, par défaut, les `cpp_dec_float` allouent la mémoire sur la pile (*stack*) et non sur le tas (*heap*). Ces allocations sont donc invisibles aux fonctions de lecture mémoire du logiciel. Toutefois, nous verrons (figure 7.17) que ce n'est pas le cas avec l'introduction des *allocateurs mémoire* à la section 7.4 — puisque ceux-ci allouent la mémoire sur le tas.

les types décimaux que dans les types binaires. On observe, dans la figure 7.10, que sans objets éphémères, le test peut consommer jusqu'à 1,6 gigaoctet. Par contre, dans la figure 7.12, avec les objets éphémères, il ne consomme au maximum qu'une trentaine de mégaoctets. De plus, dans les figures 7.11 et 7.13, on peut voir que les performances de la sommation sont meilleures avec l'utilisation des objets éphémères — de deux à quatre fois et demie plus rapide.

Un exemple de l'utilisation des objets éphémères sera présenté dans l'implémentation des méthodes de *dispatch dynamique* (listing 7.4, section 7.6).

7.4 Allocateurs mémoire des sous-classes `Numeric`

```
// Variable statique commune à toutes les instances
static Integer NumericClass::total_allocs_taille;

Void* NumericClass::custom_allocate(Integer taille) {
    total_allocs_taille += taille;

    if ((total_allocs_taille + taille) / ALLOCATEURS_DIVISEUR
        < allocateur_freq) {
        return GC_MALLOC(taille); // Ou GC_MALLOC_UNCOLLECTABLE(taille)
    } else {
        return malloc(taille);
    }
}

Void NumericClass::custom_free(Void* pointeur,Integer taille) {
    total_allocs_taille -= taille;

    /* La fonction GC_is_heap_ptr permet de détecter si le GC gère
       l'allocation */
    if (GC_is_heap_ptr(pointeur)) {
        GC_FREE(pointeur);
    } else {
        free(pointer);
    }
}
```

Listing 7.2 Pseudocode `C++` des allocateurs mémoire personnalisés d'une sous-classe `Numeric`.

Comme mentionné à la section 6.9, pour que le comportement du *GC respecte les conditions* énumérées au chapitre chapitre VI, notre solution a consisté à définir des ***allocateurs mémoire personnalisés*** (*custom memory allocators*). Toutes les bibliothèques pour arithmétique multiprécision utilisées supportent la définition de tels allocateurs, et donc ils ont été ajoutés à toutes les sous-classes `Numeric`. Le listing 7.2 présente une implémentation (simplifiée) en pseudocode `C++` des fonctions d'allocateurs mémoire à fournir aux différentes sous-classes `Numeric`[78].

La variable `allocateur_freq` prend une valeur réelle dans l'intervalle $[0, 1]$ et représente le pourcentage d'allocation gérées par le *GC*. La constante `ALLOCATEURS_DIVISEUR` doit être définie pour chacune des sous-classes `Numeric` et permet d'ajuster le comportement de l'allocateur en fonction de la mémoire requise par la bibliothèque multiprécision utilisée par la sous-classe `Numeric`.

Lorsque la taille totale des allocations mémoire (`total_allocs_taille`) est plus petite qu'un certain pourcentage `allocateur_freq`, c'est la fonction *`GC_MALLOC`* (ou *`GC_MALLOC_UNCOLLECTABLE`*) qui est utilisée pour faire en sorte que le *GC* gère l'allocation (elle lui est visible). Dans le cas contraire, c'est un simple `malloc` qui est utilisé (l'allocation n'est pas visible par le *GC*). Ces différentes fonctions personnalisées d'allocation mémoire sont implicitement appelées par les bibliothèques multiprécisions lors de la création ou la destruction de valeurs numériques — plutôt que d'utiliser des `malloc`/`free` standards.

7.4.1 Variation du pourcentage d'allocations visibles et contrôle des intervalles de collectes

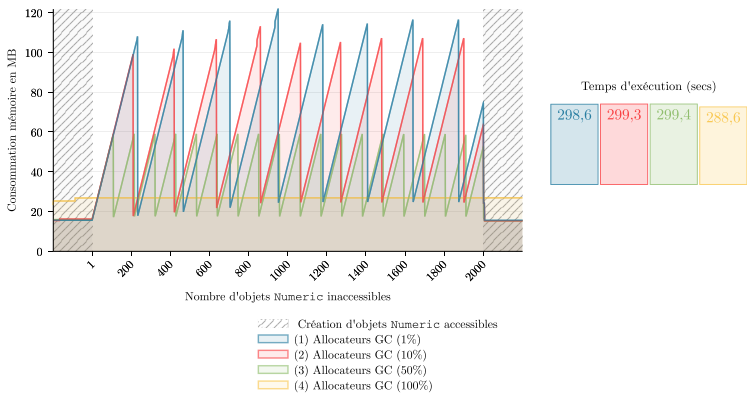


Figure 7.14 Différents pourcentages des allocations visibles par le *GC* dans *Linux* (*détails*).

On constate dans la figure 7.14 qu'il est possible de contrôler les intervalles des collectes du *GC* en variant le pourcentage des allocations mémoire visibles par le *GC*.

[78] Une troisième fonction doit être fournie, soit *`custom_reallocate`*, mais elle a été omise pour simplifier la présentation.

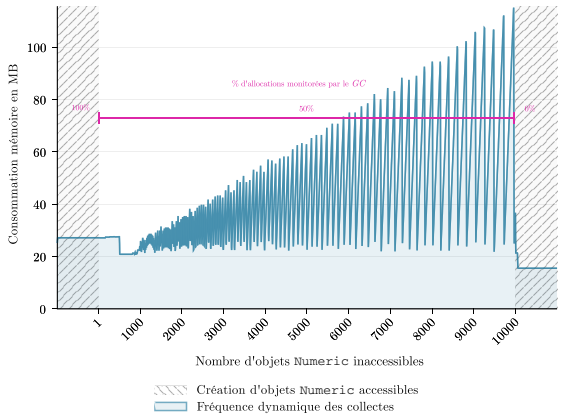


Figure 7.15 Pourcentage dynamique des allocations visibles par le *GC* dans *Linux* (*détails*).

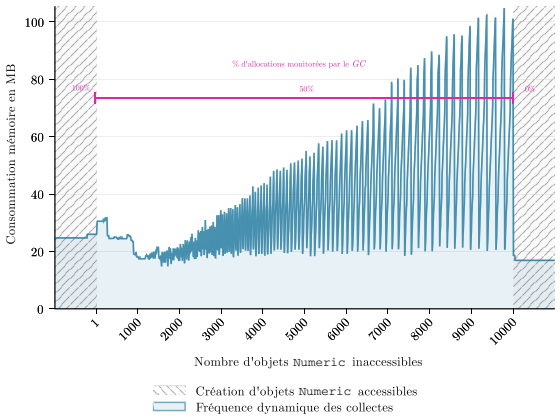


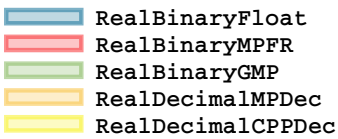
Figure 7.16 Pourcentage dynamique des allocations visibles par le *GC* dans *Windows* (*détails*).

Dans les figures 7.15 et 7.16, le pourcentage des allocations mémoire non visibles par le *GC* est incrémenté linéairement tout au long du test. En l'occurrence, en début de test, 100% des allocations mémoire sont visibles par le *GC* (il gère toutes les allocations) et ce pourcentage est graduellement diminué pour atteindre 0% en fin de test (le *GC* ne gère aucune des allocations).

En comparant les figures 7.15 et 7.16, on constate que, non seulement le comportement du *GC* peut être modifié avec précision, mais que ce dernier se comporte de façon quasi identique de plateforme en plateforme (*Linux versus Windows*). Ces figures présentent donc les deux graphes qui nous ont semblé les plus importants dans notre parcours de recherche : après un an et demi de lectures, codage, expérimentations et tests, **le *GC* se comporte (enfin !) comme prévu**, et **son comportement respecte toutes les *conditions*** énumérées au chapitre VI,

7.4.2 Variation du pourcentage d'allocations visibles et objets éphémères

Remarque: *Dans les graphiques qui suivent, les classes sont identifiées à l'aide des couleurs suivantes :*



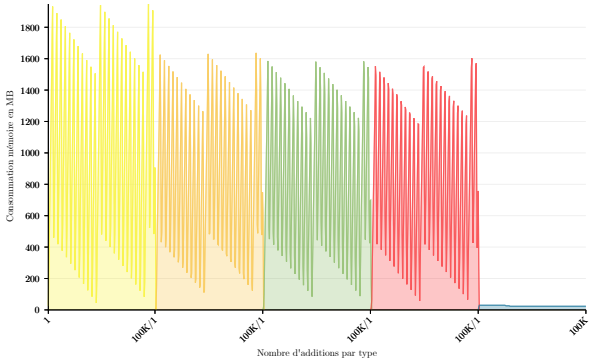


Figure 7.17 Consommation **sans** objets éphémères et **avec** allocateurs dynamiques 0% (*détails*).

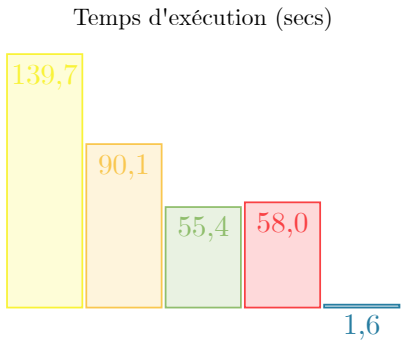


Figure 7.18 Temps **sans** objets éphémères et **avec** allocateurs dynamiques 0% (*détails*).

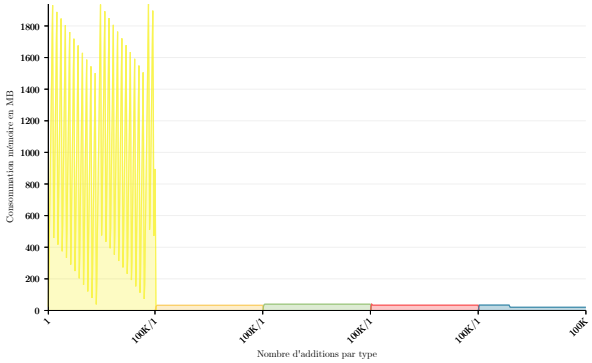


Figure 7.19 Consommation **sans** objets éphémères et **avec** allocateurs dynamiques 50% (*détails*).

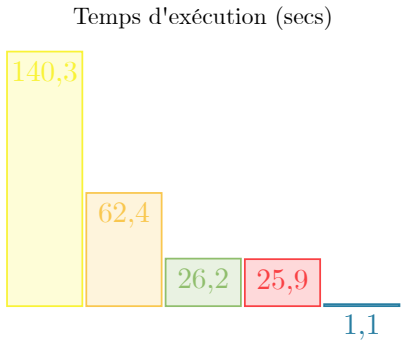


Figure 7.20 Temps **sans** objets éphémères et **avec** allocateurs dynamiques 50% (*détails*).

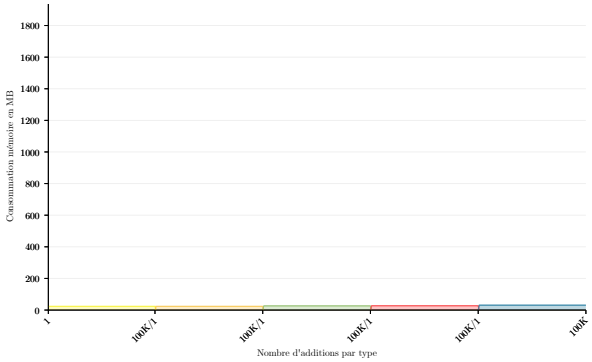


Figure 7.21 Consommation **avec** objets éphémères et **avec** allocateurs dynamiques 50% (*détails*).

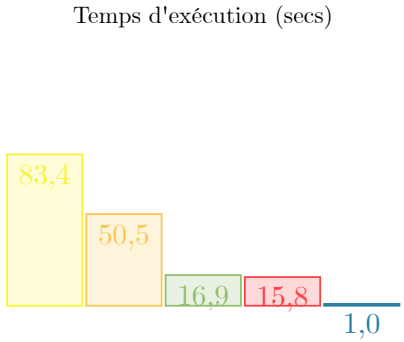


Figure 7.22 Temps **avec** objets éphémères et **avec** allocateurs dynamiques 50% (*détails*).

En comparant les figures 7.17[79] et 7.19, on constate que si le *GC* gère un certain pourcentage (50%) des

[79]En observant les résultats du type `RealDecimalCPPDec` dans la figure 7.19, on constate que les allocateurs ne semblent pas fonctionner correctement. Il s'agit d'un bogue dans l'implémentation des `Numeric`, qui fait que la modification du pourcentage d'allocations gérées par le *GC* n'a pas d'effet. Ce bogue n'a été identifié qu'au moment de la rédaction du mémoire et, faute de temps, n'a pas encore été corrigé.

allocations mémoire, les *objets éphémères* ne semblent plus nécessaires. Néanmoins, en comparant le temps dans les figures 7.20 (sans objets éphémères) et 7.22 (avec objets éphémères), on constate que les performances sont significativement supérieures lorsque les objets éphémères sont utilisés.

L'utilisation des allocateurs mémoire dans le type `RealDecimalMPDec` est plus complexe que dans les autres types. Par défaut, les allocations mémoire faites pour stocker un nombre par la classe `Decimal` de l'*API* `C++` de la bibliothèque *mpdecimal* (*mpdecimal++*) sont statiques, et non dynamiques. Or, puisque ces allocations sont statiques, les allocateurs mémoire personnalisés ne les détectent pas. Ce problème est lié au fait que nous utilisons l'*API* `C++` et non les fonctions `C` de la bibliothèque *mpdecimal*. Pour l'instant, la solution utilisée consiste à appeler explicitement la fonction `C` `mpd_set_dynamic_data` lors de la création d'un objet `RealDecimalMPDec` pour forcer une allocation dynamique (Krah, 2008, sect. *Advanced Memory Handling* et *Attributes of a Decimal*).

7.5 Support *multithreads* et classe *template* `ThreadsDataAccessor<TYPE>`

Il nous semblait important de prendre en considération le support du *multithreading*, et ce en tout début de projet[80]. En effet, l'ajout du *multithreading* peut s'avérer problématique s'il n'est pas prévu dès le début de la conception d'un langage. Il est bien connu que ce fût le cas dans certains langages de programmation, tels que *Ruby* et *Python* (Schiessl, 2014; ThePythonWiki, 2020).

Bien que toutes *thread-safes*, aucune des bibliothèques multiprécision utilisées n'est en mesure de profiter des *microprocesseurs multi-cœur* (*multi-core processors*) via le *multithreading* pour accélérer les temps de calculs à grandes précisions. Cependant, il serait intéressant de subdiviser les calculs complexes, dans l'interpréteur *Kpsilon*, pour les effectuer en parallèle — par exemple, pour $0{,}4 \div 3 + 1324 \div 4213231312$, on pourrait calculer $0{,}4 \div 3$ dans un premier *thread* et $1324 \div 4213231312$ dans un deuxième *thread*, puis effectuer l'addition.

Or, l'ajout des allocateurs mémoire rendait notre code non *thread-safe*. En effet, depuis leur ajout, les sous-classes `Numeric` tiennent à jour des compteurs de consommation mémoire et de nombre d'objets (`Numeric` ou autres) créés (par exemple, les variables statiques `s_mem_size` et `s_alloc_count` dans le listing 7.3.). Ces compteurs doivent impérativement être *thread-safes* pour permettre la création d'objets en parallèle.

[80] Le *GC Boehm* supporte le *multithreading* si l'option `-DGC_THREADS` (ou `-DGC_LINUX_THREADS`) est définie lors de sa compilation.

Pour éviter de modifier tous les allocateurs mémoire des sous-classes `Numeric`, ainsi que d'ajouter plusieurs macros de préprocesseur `C++`, la classe *template* `ThreadsDataAccessor<TYPE>` a été ajoutée. Cette classe est responsable de gérer implicitement les barrières d'accès aux variables statiques privées et partagées (au moyen de la fonction *RAII* `std::scoped_lock` de `C++17`). Le fait que les fonctions d'allocateurs mémoire utilisées par les bibliothèques multiprécisions sont des fonctions statiques implique que ces fonctions doivent manipuler des variables statiques partagées par toutes les instances de la sous-classe `Numeric`, ce qui a compliqué l'implémentation de la classe `ThreadsDataAccessor`.

Pour stocker les informations des différents *threads*, des conteneurs `unordered_map` sont utilisés par la classe `ThreadsDataAccessor` — ils sont, dans la plupart des cas, plus optimaux que des `map` ($\mathcal{O}(1)$ en moyenne et $\mathcal{O}(n)$ dans le pire cas, *versus* $\mathcal{O}(\log n)$). Ils sont aussi *thread-safes*, à tout le moins, en lecture[81].

Le listing 7.3 présente la partie du code `C++` du logiciel *Kpsilon* responsable de déclarer les compteurs de type `ThreadsDataAccessor` pour les allocateurs mémoire des types `Numeric` dans la classe *template* `NumericTmplBase<CLASS,PARENT,TYPE>`.

Par défaut, une instance de la classe `ThreadsDataAccessor` est privée, mais elle peut être déclarée partagée en passant la valeur `false` comme argument au constructeur de la classe. Une instance `ThreadsDataAccessor` privée est un type de base ou un objet statique (ou global) qui est unique (local) à un *thread* — donc similaire au mécanisme *thread local storage* présent dans certains langages de programmation[82]. Le compteur `s_mem_size` doit être privé, car la quantité de mémoire allouée par un *thread*, lors de la création d'un objet `Numeric`, ne doit pas entrer en conflit avec la quantité de mémoire allouée par un autre type. Une instance `ThreadsDataAccessor` partagée est, quant à elle, identique du point de vue de tous les *threads*. Par contre, la classe `ThreadsDataAccessor` est responsable de verrouiller implicitement les *mutex* (`std::scoped_lock`), lors d'accès de lecture et écriture, pour éviter les *situations de compétition* (*race conditions*) entre les différents *threads*.

[81] `https://en.cppreference.com/w/cpp/container#Thread_safety` (*archive*).

[82] L'utilisation de *thread local storage* permettrait de remplacer l'utilisation de la fonction `std::scoped_lock` dans le cas des objets `ThreadsDataAccessor` privés. Par manque de temps, cette solution n'a pas été explorée, mais elle serait envisageable puisque le *GC Boehm* supporte ce mécanisme (Boehm et Demers, 1988, sect. *Garbage collector scalability* (archive)).

```cpp
template<typename C, typename P, typename T>
class NumericTmplBase : public P {
protected:
    T* m_data_{nullptr}; // Donnée bibliothèque multiprécision

// Compteur de nombre d'instances du type Numeric
#ifdef ENABLE_THREADS_DATA_ACCESSORS
    inline static memory::ThreadsDataAccessor<size_t>
                  s_this_inst_count{false}; // PARTAGÉE
#else
    inline static size_t s_this_inst_count = 0;
#endif

#ifdef USE_CUSTOM_ALLOCATORS
    // Mémoire consommée par le Numeric actuel (this)
    size_t m_mem_size_ = 0;

#   ifdef ENABLE_THREADS_DATA_ACCESSORS
    /* Taille de la mémoire alouée par les Numeric d'un même
       type (PRIVÉE) */
    inline static memory::ThreadsDataAccessor<size_t> s_mem_size;
    /* Nombre d'allocations mémoire détectées par l'allocateur
       d'un type Numeric (PARTAGÉE) */
    inline static memory::ThreadsDataAccessor<size_t> s_alloc_count{false};
    /* Nombre d'allocations gérées par le GC (PARTAGÉE) */
    inline static memory::ThreadsDataAccessor<size_t> s_managed_count{false};
#   else
    inline static size_t s_mem_size      = 0;
    inline static size_t s_alloc_count   = 0;
    inline static size_t s_managed_count = 0;
#   endif // #ifdef ENABLE_THREADS_DATA_ACCESSORS
#endif // #ifdef USE_CUSTOM_ALLOCATORS

public:
// [...]
}
```

Listing 7.3 Extrait du fichier `src/numeric_v2/numeric_tmpl_base.ipp`.

Les différents tests que nous avons effectués démontrent que l'ajout de la classe `ThreadsDataAccessor` affecte peu les performances des sous-classes `Numeric`[83].

[83] Exemple : un test créant 10 000 objets `RealBinaryGMP` à une précision d'un million de décimales prend 1 495,6 secs (sans `ThreadsDataAccessor`) *vs.* 1 502,5 secs (avec `ThreadsDataAccessor`) sous *Debian 12.1* 64 bits (x86-64 40x3GHz/31GB).

7.6 *Dispatch dynamique*

Puisqu'il est possible d'effectuer des opérations arithmétiques entre les différents types d'objets `Numeric`, et ce, tant dans l'interpréteur *Kpsilon* que dans le code `C++`, il était nécessaire de mettre en place une forme de *dispatch multiple* (*multiple dispatch*). Le *dispatch multiple* permet de spécialiser une méthode pour plus d'un type de ses paramètres (formels) et facilite ainsi le polymorphisme — ce qui s'avère particulièrement important pour la classe `RealDecimalCPPDec`, laquelle est composée d'une multitude d'instances de la classe *template* `RealDecimalCPPDecTmpl<PREC>`.

Certains langages de programmation supportent par défaut le *dispatch multiple* — pensons aux langages *Lisp*, via le *CLOS* (*Common Lisp Object System*), et *Julia*. Malheureusement, le langage `C++` ne possède pas de fonctionnalité de *dispatch multiple*, bien que certaines solutions ont été proposées, dont les multi-méthodes — proposées par Stroustrup (concepteur du `C++`) : « *As of 2021, `C++` natively supports only single dispatch, though adding multi-methods (multiple dispatch) was proposed by Bjarne Stroustrup (and collaborators) in 2007* (Stroustrup *et al.*, 2007) ».

Les bibliothèques *YOMM2* et *omm* (*Open Multi-Methods*), fortement inspirées de la proposition de Stroustrup *et al.* (2007), permettent de supporter les multi-méthodes en `C++`. Nous avons fait quelques expérimentations, mais aucune de ces deux bibliothèques ne nous satisfaisait. La bibliothèque *Multimethods*, plus simple et moins lourde que les deux précédentes, a davantage été testée ; encore une fois, les résultats ont été décevants.

Une autre solution possible était d'utiliser un *patron de conception de type « visiteur »* (*visitor pattern*) (Gamma *et al.*, 1995). Bien que cette approche populaire soit la plus recommandée pour ce type de problème, nous avons constaté lors de nos expérimentations qu'elle aussi rendait le code complexe. De plus, nous craignions la complexité d'implémentation/maintenance et la surcharge (*overhead*) engendrées par l'utilisation d'un tel type de patron (Aakashattri111, 2023).

Suite à des expérimentations, nous avons choisi d'utiliser le *dispatch simple* et le polymorphisme du `C++` pour mettre en place les méthodes de *dispatch dynamique* (*dynamic dispatch*). Ce choix explique la décision d'utiliser la classe `Numeric` comme mère de toutes les classes des types multiprécision afin de les rendre polymorphiques. Puisque, hormis l'opération de division, le code de toutes les opérations est identique, le ***préprocesseur `C++` « maison »*** (annexe C) mis en place pour la classe `RealDecimalCPPDec` (sous-section 7.2.3) a été utilisé pour éviter la répétition de code. Le listing 7.4 présente une version simplifiée d'une méthode de *dispatch dynamique*.

```cpp
/*=== PP_REPEAT_BEGIN(PP_OP__, ['+', '-', '*', '/']) ===*/
template <unsigned P>
Numeric& RealDecimalCPPDecTmpl<P>::operator/*=== PP_EXPR(PP_OP__) ===*/(const Numeric& right) {
    NumericType res;
    size_t      prec = get_result_prec(get_prec(), right.get_prec());

    // Dispatch dynamique
    switch(right.get_type()) {

        case ObjType::RealDecimalCPPDec : {

            if (right.get_prec() == get_prec()) { // Les précisions sont identiques

                res = Numeric::generate(ObjType::RealDecimalCPPDec, prec);

                res.set_data(new BoostRealDecimal<P>(get_data() /*=== PP_EXPR(PP_OP__) ===*/ right.get_data()));
            } else { // Les précisions sont différentes

                switch (right.get_prec()) {

                /*=== PP_REPEAT_BEGIN(PP_PREC__, [50, 100, 500, 1000, 10000, 100000, 1000000, 2140000]) ===*/
                    case /*=== PP_EXPR(PP_PREC__) ===*/: {

                        // Faire la conversion de précision et effectuer l'opération
                        // NOTE: Cette partie est optimisée dans la version finale du code!
                        if (right.get_prec() > get_prec()) {
                            res = *this /*=== PP_EXPR(PP_OP__) ===*/ *(new RealDecimalCPPDecTmpl<P>(right));
                        } else {
                            res = *(new RealDecimalCPPDecTmpl</*=== PP_EXPR(PP_PREC__) ===*/>(get_data())) \
                                  /*=== PP_EXPR(PP_OP__) ===*/ right;
                        }

                        break;
                    }
                /=== PP_REPEAT_END ===*/

                    default: throw except::illegal_precision_except(); // Erreur! Précision invalide
                }
            }
        }
```

```
        // Faire la conversion de type et effectuer l'opération
        case ObjType::RealBinaryGMP    : { res = Numeric::generate(ObjType::RealBinaryGMP,   prec, get_data())  \
                                                 /*=== PP_EXPR(PP_OP__) ===*/ right;
                                                 break;
                                       }
        case ObjType::RealBinaryMPFR   : { res = Numeric::generate(ObjType::RealBinaryMPFR,  prec, get_data())  \
                                                 /*=== PP_EXPR(PP_OP__) ===*/ right;
                                                 break;
                                       }
        case ObjType::RealBinaryFloat  : { res = Numeric::generate(ObjType::RealBinaryFloat, prec, get_data())  \
                                                 /*=== PP_EXPR(PP_OP__) ===*/ right;
                                                 break;
                                       }
        case ObjType::RealDecimalMPDec : { res = *this /*=== PP_EXPR(PP_OP__) ===*/                            \
                                                 Numeric::generate(ObjType::RealDecimalCPPDec, prec, get_data());
                                                 break;
                                       }
        default : { throw except::bad_numeric_type_except(); } // Erreur! Type invalide
    }

    // Destruction automatique des objets éphémères s'ils sont activés
    if (configs.mem.ephemeral_objects) { // Est-ce que les objets éphémères sont activés?

        res.set_ephemeral(true); // Le résultat est un objet éphémère

        if (right.is_ephemeral()) { delete &right; }
        // Le "delete this" est légal et nécessaire pour détruire l'objet éphémère gauche
        // Voir https://stackoverflow.com/questions/11756693/self-delete-an-object-in-c
        if (this->is_ephemeral()) { delete this; }
    }

    return res;
}
/=== PP_REPEAT_END ===*/

// Déclarations explicites des différents templates au moment de la compilation
/*=== PP_REPEAT_BEGIN(PP_PREC__, [50, 100, 500, 1000, 10000, 100000, 1000000, 2140000]) ===*/
template class RealDecimalCPPDecTmpl</*=== PP_EXPR(PP_PREC__) ===*/>;
/=== PP_REPEAT_END ===*/
```

Listing 7.4 Pseudocode C++ de la méthode de *dispatch dynamique* (simplifiée) de la classe `RealDecimalCPPDec`.

Dans la méthode de *dispatch dynamique* de la classe `RealDecimalCPPDec` du listing 7.4, les commandes du préprocesseur « *maison* » se trouvent dans les commentaires délimités par `/*=== ... ===*/`. La syntaxe et l'utilisation de ces commandes sont décrites en détails en annexe C. Mentionnons simplement que les commandes `PP_REPEAT_BEGIN(ITERATEUR, LISTE)` et `PP_REPEAT_END` sont équivalentes à un « `for ITERATEUR in LISTE:` » en *Python*. Le champ `ITERATEUR` est le nom de l'objet courant et le champ `LISTE` doit être remplacé par n'importe quel objet *Python* de type `List`. La commande `PP_EXPR(EXPRESSION)` permet d'ajouter au code `C++` le résultat de l'expression *Python* `EXPRESSION`.

Une fois traité par le préprocesseur « *maison* », le code du listing 7.4 génère un fichier contenant les implémentations des quatre opérations arithmétiques et les `case` pour « *dispatcher* » les différents *templates* `RealDecimalCPPDecTmpl<PREC>` et autres sous-classes `Numeric`.

En utilisant cette approche, l'implémentation des différentes méthodes de *dispatch dynamique* est simplifiée et évite la répétition de code. Cette stratégie requiert cependant l'ajout de classes intermédiaires et l'utilisation des classes abstraites et méthodes virtuelles du `C++` qui ajoutent une certaine surcharge (*overhead*). Heureusement, comme on le verra à la section 7.7, les performances du logiciel sont plus que satisfaisantes.

Pour réduire cette surcharge, les méthodes de *dispatch dynamique* évitent d'utiliser les fonctions `C++` de *casting* telles que `dynamic_cast<TYPE>(object)` et `typeid(object).name()`[84] pour identifier le type d'objet `Numeric` passé en paramètres. En l'occurrence, tous les objets du langage *Kpsilon* sont « *conscients* » de leur type et peuvent le retourner au moyen de la méthode `Object::get_type` — ceci évite de devoir « *downcaster* » les objets `Numeric` pour les identifier.

7.7 Bilan : comparaisons des types `Numeric` avec différents langages/logiciels

Dans cette section, nous comparons les temps d'exécution du calcul de l'expression «$0{,}4 \div 3 + 1324 \div 4\,213\,231\,312$»[85] à une précision d'un million de décimales, avec *Kpsilon* et avec les langages/logiciels *Python 2.7.12*, *Python 3.5.2*, *Python 3.11.2*, *Ruby MRI 2.3.1*, *GNU bc 1.06.95* et *GNU Octave 4.2.2* (section 1.2).

[84] Il faut éviter la fonction `dynamic_cast<TYPE>(object)`, car elle est plutôt lente (« *downcasting* »). La fonction `typeid(object).name()`, bien que plus rapide, retourne des résultats différents selon la plateforme, donc est problématique.

[85] Le résultat possède une partie décimale sans périodicité jusqu'à un million de décimales. C'est l'expression $1324 \div 4\,213\,231\,312$ qui est responsable de cette propriété. Le résultat possède bien sûr un *développement décimal périodique*, puisque c'est un rationnel, mais la période est de 12 391 856 décimales — affirmation que nous avons validée à l'aide d'un script *Python* « *maison* ». Nous avons souvent utilisé cette expression dans nos expérimentations, car certaines bibliothèques multiprécision sont plus efficaces que d'autres avec des nombres avec une partie décimale périodique (ex. $0{,}1\overline{3}$) ou finie (ex. $0{,}1$).

Nous comparons aussi les différents types `Numeric` entre eux. Le tableau 7.1 présente les types et langages comparés.

type	logiciel/ langage	base	bibliothèque arithmétique
`Float`	*Ruby MRI 2.3.1*	2	*N/A* (`double` *64 bits*)
`float`	*Python 3.5.2*	2	*N/A* (`double` *64 bits*)
`float`	*Python 3.11.2*	2	*N/A* (`double` *64 bits*)
`RealBinaryFloat`	*Kpsilon*	2	*N/A* (`double` *64 bits*)
`Binary`	*GNU bc 1.06.95*	2	*« maison »* (`number.c`)
`Binary`	*Python 3.5.2*	2	*mpmath* (*GMP*)
`Binary`	*Python 3.11.2*	2	*mpmath* (*GMP*)
`Binary`	*GNU Octave 4.2.2*	2	*SymPy* / *mpmath* (*GMP*)
`RealBinaryGMP`	*Kpsilon*	2	`gmp_float` (*GMP*)
`RealBinaryMPFR`	*Kpsilon*	2	`mpfr_float` (*MPFR*)
`Decimal`	*Python 2.7.12*	10	`decimal.py`
`Decimal`	*Python 3.5.2*	10	*cdecimal* (*mpdecimal*)
`Decimal`	*Python 3.11.2*	10	*cdecimal* (*mpdecimal*)
`RealDecimalCPPDec`	*Kpsilon*	10	`cpp_dec_float` (*Boost*)
`RealDecimalMPDec`	*Kpsilon*	10	*mpdecimal*

Tableau 7.1 Liste des types testés de différents langages/logiciels.

Les figures qui suivent ont été générées dans *Linux*[86] par un script *Python*. Lorsque cela s'avère pertinent (c.-à-d. si les variations de temps sont significatives), les temps indiqués sont les temps moyens (*moy*) de 100 exécutions ; de plus, les temps minimums (*min*), maximums (*max*) et les écarts type (σ) sont aussi indiqués (cf. ci-dessous, tableau 7.2).

[86] *Linux Ubuntu 16.04.7 LTS* 64 bits (x86-64 2x2.26GHz/8GB).

7.7.1 Comparaisons des performances

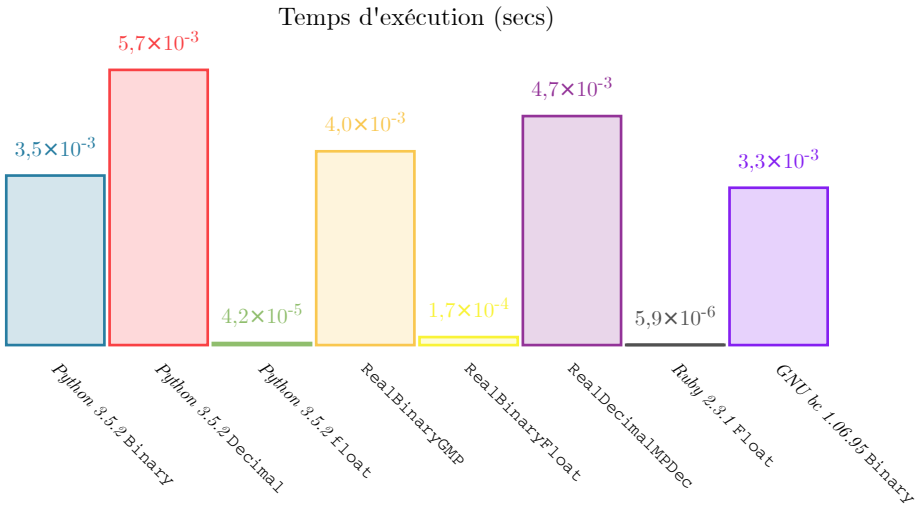


Figure 7.23 Comparaison des temps (`Numeric` avec allocateurs mémoire).

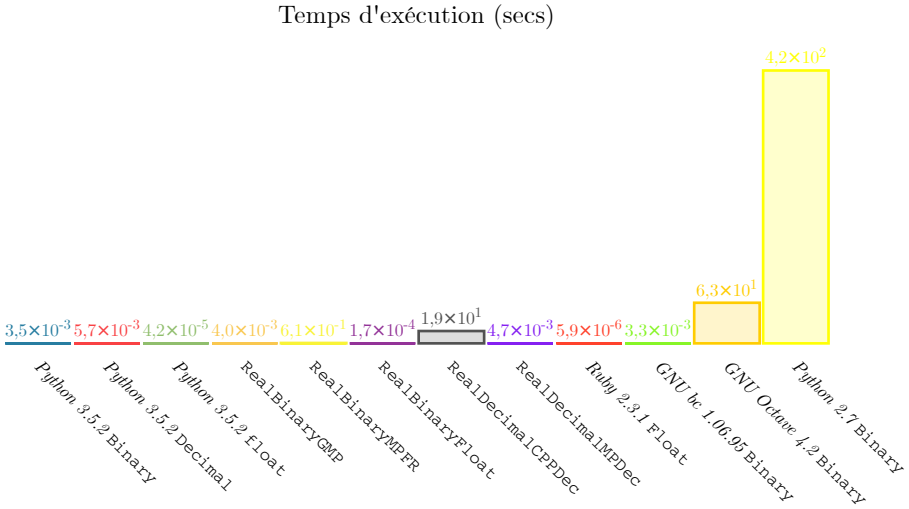


Figure 7.24 Comparaison des temps des types lents (`Numeric` avec allocateurs mémoire).

Dans la figure 7.23[87], on remarque que les calculs avec le type `RealDecimalMPDec` sont plus rapides qu'avec leur équivalent en *Python 3.5.2* (`Decimal`) ; les types `RealBinaryGMP` et son équivalent *Python* (`Binary`) ont, quant à eux, des performances assez semblables. Notons que seul le type `Float` 64 bits a été testé dans *Ruby MRI* et que les calculs avec le type `RealBinaryFloat` s'effectuent plus lentement qu'avec les `Float` de *Ruby* ainsi qu'avec les `float` de *Python*.

[87] Les types `RealDecimalCPPDec` et `RealBinaryMPFR` ont été omis, car trop lents.

Comme prévu, le type `RealDecimalCPPDec` est le plus lent de tous les types `Numeric` : cf. figure 7.24. Cette figure permet aussi de constater que le type `RealBinaryMPFR` est beaucoup plus lent que le type `RealBinaryGMP`. Or, ils devraient être relativement identiques, ce qui laisse présager la présence d'un bogue avec ce type. Ce problème peut possiblement s'expliquer par la remarque suivante dans la documentation de la bibliothèque *MPFR* :

> *Avoid using MPFR from `C++`, or make sure your `C++` interface does not perform unnecessary allocations or copies. Slowdowns of up to a factor 15 have been observed on some applications with a `C++` interface.*
>
> The *MPFR* documentation, sect. *4.8 Getting the best efficiency out of MPFR* (*archive*).

De plus, dans la figure 7.24, on constate aussi que *GNU Octave* est, de loin, le plus lent de tous (en faisant abstraction de la lenteur des versions de *Python* antérieures à la version 3.3). La bibliothèque *mpdecimal* n'est utilisée par *Python* que depuis *Python 3.3* ; dans ses versions antérieures, *Python* utilisait la bibliothèque *`decimal.py`*[88]. Cette dernière est codée en *Python* et est donc beaucoup plus lente que *mpdecimal* (codée en `C`) — comme on peut le constater dans la figure 7.24.

type	base	moyenne	minimum	maximum	écart type
`Float` *Ruby MRI 2.3*	2	$5{,}9 \times 10^{-6}$	$5{,}1 \times 10^{-6}$	$7{,}1 \times 10^{-6}$	$4{,}5 \times 10^{-7}$
`float` *Python 3.5*	2	$4{,}2 \times 10^{-5}$	$3{,}9 \times 10^{-5}$	$9{,}5 \times 10^{-5}$	$7{,}5 \times 10^{-6}$
`RealBinaryFloat`	2	$1{,}7 \times 10^{-4}$	$1{,}5 \times 10^{-4}$	$2{,}6 \times 10^{-4}$	$1{,}8 \times 10^{-5}$
`Binary` *GNU bc 1.06*	2	$3{,}3 \times 10^{-3}$	$2{,}6 \times 10^{-3}$	$1{,}9 \times 10^{-2}$	$1{,}6 \times 10^{-3}$
`Binary` *Python 3.5*	2	$3{,}5 \times 10^{-3}$	$3{,}4 \times 10^{-3}$	$5{,}8 \times 10^{-3}$	$2{,}5 \times 10^{-4}$
`RealBinaryGMP`	2	$4{,}0 \times 10^{-3}$	$3{,}8 \times 10^{-3}$	$4{,}5 \times 10^{-3}$	$1{,}2 \times 10^{-4}$
`RealDecimalMPDec`	10	$4{,}7 \times 10^{-3}$	$4{,}6 \times 10^{-3}$	$5{,}1 \times 10^{-3}$	$1{,}3 \times 10^{-4}$
`Decimal` *Python 3.5*	10	$5{,}7 \times 10^{-3}$	$5{,}1 \times 10^{-3}$	$6{,}1 \times 10^{-3}$	$2{,}9 \times 10^{-4}$
`RealBinaryMPFR`	2	$6{,}1 \times 10^{-1}$	*N/A*	*N/A*	*N/A*
`RealDecimalCPPDec`	10	$1{,}9 \times 10^{1}$	*N/A*	*N/A*	*N/A*
`Binary` *GNU Octave 4.2*	2	$6{,}3 \times 10^{1}$	*N/A*	*N/A*	*N/A*
`Decimal` *Python 2.7*	10	$4{,}2 \times 10^{2}$	*N/A*	*N/A*	*N/A*

Tableau 7.2 Temps en secondes du calcul $0{,}4 \div 3 + 1324 \div 4213231312$ (un million de décimales).

[88] `https://www.bytereef.org/mpdecimal/index.html` (*archive*)

Les résultats précédents sont présentés en ordre de performance dans le tableau 7.2. On peut noter que les types utilisant des `double` 64 bits (bases en rouge), bien que plus performants, ne permettent qu'une précision maximale de 15 à 17 décimales (section 4.1). On peut aussi noter que les performances de plusieurs types `Numeric` se classent bien en comparaison avec les autres logiciels/langages ; seuls les types `RealBinaryMPFR` et `RealDecimalCPPDec` sont nettement plus lents — lenteur possiblement causée par un bogue dans le cas du type `RealBinaryMPFR`.

Notons que, bien que les performances du logiciel *GNU bc* (*basic calculator*, calculatrice à arithmétique multiprécision) soient excellentes et qu'il retourne un résultat avec une précision exacte, il s'agit d'un logiciel limité et difficile d'utilisation, avec une syntaxe proche du langage `C`. Il n'est généralement utilisé qu'à l'intérieur de scripts *Bash* ou équivalents.

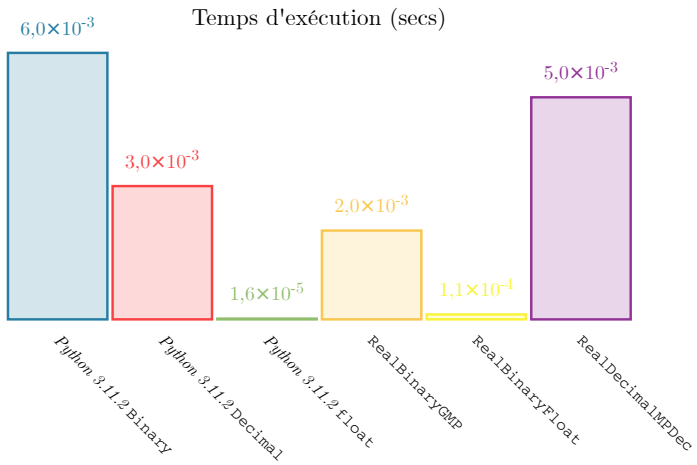


Figure 7.25 Temps `Numeric` *vs. Python 3.11.2* à un million de décimales (moy. 100 exécutions).

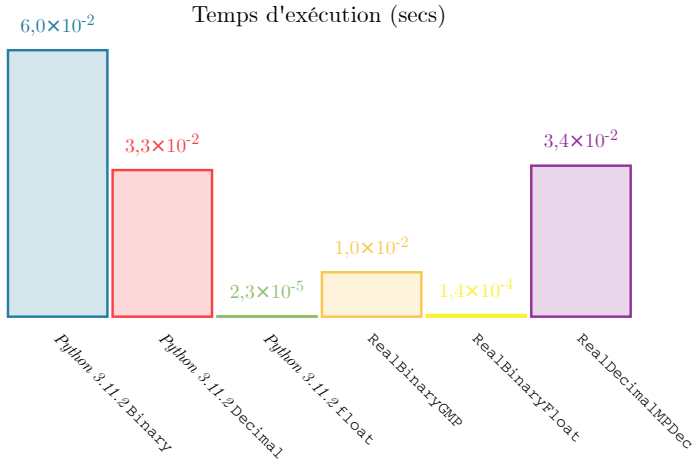


Figure 7.26 Temps `Numeric` *vs. Python 3.11.2* à 10 millions de décimales (moy. 10 exécutions).

Puisque les versions testées de certains logiciels ne sont plus à jour, comme c'est le cas de *Python 3.5.2*, on peut présumer qu'une version plus récente soit plus performante. *Python 3.11.2* a donc été testé sur un autre système plus moderne et se comporte de façon similaire (figure 7.25), et ce, même à de plus grandes précisions (figure 7.26)[89] ; quoique le type `Binary` de *Python 3.11.2* semble, dans les deux précisions, plus lent (en comparaison avec notre logiciel) que celui de *Python 3.5.2*.

7.7.2 Comparaisons des précisions

```
$ ./calc --test --tool="kpsilon_bin,kpsilon_bin2,kpsilon_dec,
                        kpsilon_dec2,python_bin,python_dec,
                        bc,octave" --prec=1000000 --dps=20
                        0.4/3+1324/4213231312

python bin   :  0.1333336475814488424
python dec   :  0.1333336475814488424
kpsilon bin  :  0.1333336475814488424
kpsilon bin2 :  0.1333336475814488424
kpsilon dec  :  0.1333336475814488424
kpsilon dec2 :  0.1333336475814488424
bc           :  0.1333336475814488424
octave       :  0.1333336475814488424

TEST PASSED
```

Capture 7.1 Résultats des tests des différents types (**sans** types `double` 64 bits).

```
$ ./calc --test --tool="kpsilon_bin,kpsilon_bin2,kpsilon_bin3,kpsilon_dec,
                        kpsilon_dec2,python_bin,python_dec,python,ruby,
                        bc,octave" --prec=1000000 --dps=20
                        0.4/3+1324/4213231312

python bin   :  0.1333336475814488424
python dec   :  0.1333336475814488424
python       :  0.13333364758144883
kpsilon bin  :  0.1333336475814488424
kpsilon bin2 :  0.1333336475814488424
kpsilon bin3 :  0.13333364758144883
kpsilon dec  :  0.1333336475814488424
kpsilon dec2 :  0.1333336475814488424
ruby         :  0.13333364758144883
bc           :  0.1333336475814488424
octave       :  0.1333336475814488424

[...]
DIFF 3 (python float) vs 10 (bc) add 4... at position 19
[...]
DIFF 6 (kpsilon binary3) vs 8 (kpsilon decimal2) add 4... at position 19
[...]
DIFF 9 (ruby) vs 11 (octave) add 4... at position 19
```

Capture 7.2 Résultats des tests des différents types (**avec** types `double` 64 bits).

[89] Ces deux graphes ont été générés sous *Debian 12.1* 64 bits (x86-64 40x3GHz/31GB).

Les captures 7.1 et 7.2[90] présentent les sorties d'un script qui teste tous les résultats en s'assurant qu'ils soient exacts jusqu'à la précision désirée — un million de décimales. On remarque dans la capture 7.2 que les types utilisant des `double` 64 bits ne passent pas le test et qu'ils diffèrent du résultat exact à partir de la 19^e décimale ; la capture 7.1, quant à elle, ne présente aucun type utilisant des `double` 64 bits et passe donc tous les tests. Précisons que seuls 20 chiffres sont visibles (option `--dps=20`, soit 20 chiffres incluant la partie entière) pour simplifier l'affichage à l'écran ; néanmoins, il y a bien un million de décimales (option `--prec=1000000`) qui sont calculées et vérifiées.

À la vue de ces résultats, on peut en conclure que, hormis la faible performance des types `RealBinaryMPFR` et `RealDecimalCPPDec`, la structure des sous-classes `Numeric`, le *dispatch dynamique* et l'interpréteur *Kpsilon* se comportent de manière satisfaisante.

[90] La commande `calc` utilisée dans ces captures est un script *Bash* qui filtre la sortie pour la simplifier.

CONCLUSION

Ce projet de recherche avait comme *objectif initial* de mettre en œuvre un système de calcul formel en langage `C++` — nommé *Kpsilon*. Toutefois, comme nous l'avons rapidement réalisé, cet objectif était trop ambitieux dans le cadre d'une maîtrise à temps partiel. Nous avons donc dû réévaluer notre objectif et restreindre la portée de notre projet. En l'occurrence, nous avons choisi d'axer nos recherches et travaux sur *les fondations* qui permettront, éventuellement, la mise en œuvre d'un tel système.

Bien qu'en constante réévaluation, notre *objectif révisé* comportait : la mise en place d'un environnement de développement fonctionnant sous *Linux* et *Windows* ; la conception et mise en œuvre du *langage Kpsilon* et de son *interpréteur* au moyen du compilateur de compilateur *ANTLR 4* ; l'intégration d'un mécanisme, utilisant le *ramasse-miettes Boehm*, pour une *gestion implicite de la mémoire* ; le support de l'*arithmétique à précision arbitraire en bases 2 et 10* en utilisant *diverses bibliothèques de calcul multiprécision* ; la conception de *fonctionnalités de tests* pour valider le bon fonctionnement du *GC*. De plus, l'ensemble de ces composants devait supporter le *multithreading*, dans l'optique que le langage *Kpsilon* supportera éventuellement la création de *threads*.

Comme nous croyons l'avoir montré dans les chapitres V, VI et VII, nous avons été en mesure de mettre en œuvre les points énumérés ci-dessus et, en ce sens, nous pensons avoir atteint notre objectif révisé.

Concernant l'intégration du ramasse-miettes *Boehm*, grâce à nos expérimentations, nous avons réalisé que tous les problèmes ne pouvaient pas être réglés en modifiant ses diverses options — comme nous l'avons cru pendant trop longtemps. En effet, les options par défaut peuvent faire qu'il fonctionne correctement dans la majorité des situations. Il est toutefois primordial de bien comprendre son fonctionnement, d'utiliser les mécanismes appropriés de configuration — allocateurs mémoire (section 7.4), utilisation de classe la `gc_cleanup`, instanciation des structures de données avec `gc_allocator` — et de suivre les recommandations énumérées à la section 6.10. Sans contredit, parvenir à cette conclusion a été la principale difficulté de notre projet.

Tout au long du développement de *Kpsilon*, nous avons validé son fonctionnement de diverses façons : fonctionnalités de tests (sections B.1 et B.2), commandes de débogage de l'interpréteur (section A.5), tests unitaires (section A.5), outils d'analyses statique et dynamique de code (section 5.5). Les tests et

expérimentations que nous avons faits nous ont permis de vérifier le bon fonctionnement de *Kpsilon* et de son ramasse-miettes, de comparer les performances et consommations mémoire, et de valider les résultats des calculs multiprécisions. Concernant ses performances, nous sommes plus que satisfait des résultats obtenus en les comparant avec celles d'autres logiciels (sous-section 7.7.1, tableau 7.2).

Une autre difficulté importante de notre projet fut de concevoir un logiciel qui soit vraiment multiplateforme. Entre autres, nous avons constaté que le *GC Boehm* se comporte différemment de plateforme en plateforme, et ce, même entre distributions *Linux*. De plus, en `C++`, les différences entre plateformes concernant la gestion mémoire, l'accès au système de fichiers, l'affichage au terminal ou tout autre type d'accès aux ressources système, doivent être prises en compte.

Bien que notre logiciel ne puisse pas encore effectuer des calculs et des manipulations symboliques, l'arbre syntaxique produit par *ANTLR 4* permettrait déjà de représenter des formules mathématiques sous forme symbolique. Il suffirait ensuite d'utiliser le visiteur d'arbres syntaxiques fourni par *ANTLR* pour incorporer le code permettant d'évaluer l'expression. C'est cette approche que nous avions utilisée dans notre premier prototype *Java*/*SableCC* (mentionné en *introduction*). Toutefois, pour des raisons d'optimisation, il serait préférable de convertir cet arbre en une *représentation intermédiaire* — une représentation simplifiée facilitant les manipulations et les optimisations.

Pour conclure, mentionnons que la description complète du langage *Kpsilon*, ainsi que des exemples d'utilisation de l'interpréteur et de résultats des fonctionnalités de tests, se trouvent à l'annexe B. Un accès au dépôt *Git* du projet *Kpsilon* (incluant les codes sources des différents exemples présentés dans ce mémoire), sa documentation *Doxygen* et un accès *SSH* vers une machine virtuelle *Linux* sur laquelle il est possible de tester le logiciel, pourra aussi être fourni sur demande.

ANNEXE A
LANGAGE ET INTERPRÉTEUR *KPSILON*

CONTENU

A.1 Plateformes, architectures et compilateurs testés

Le projet *Kpsilon* supporte les architectures *IA-32*, *x86-64* et *ARM* sur un système possédant minimalement 4 GB de mémoire *RAM*. La compilation a été testée avec les compilateurs *gcc++ 8.3*, *gcc++ 9.4* et *gcc++ 12.2*, *Clang++ 9.0*, *Clang++ 14.0* et *MS Visual Studio 2019* (*cl*) sous les plateformes suivantes[91] :

- *Microsoft Windows 7 Pro*[92] 64 bits (*x86-64*) ;
- *Linux Ubuntu 16.04 LTS* et *18.04.5 LTS* 64 bits (*x86-64*) ;
- *Linux Debian 12.1* 64 bits (*x86-64*) ;
- *Linux Raspbian 10* 32 bits (*ARM*) ;
- *Linux CentOS 7.9.2009* 64 bits (*x86-64*).

[91] Soulignons qu'en cours de projet, l'environnement de développement est devenu dépassé (certains systèmes d'exploitation et logiciels sont devenus obsolètes), mais il n'a pas été mis à jour, pour des raisons financières et par manque de temps.

[92] Bien qu'elles n'ont pas été testées, la compilation et l'exécution du logiciel devraient fonctionner tel quel sous *Windows 10/11*.

A.2 Interface utilisateur du logiciel *Kpsilon*

- **Menu principal**

```
GC:ucalloc(50%)-thrds GC_ver:8.0.4 GC_div:3 GC_thrsh:2(5%) rel_mem:yes
GC_heap_max:unlimited GC_heap_expand:0 GC_heap_size:4788224
prec:100000(100000) dec_prec:20... radix:10 lib:MPDEC ver:2.5 numeric_ver:2
process_id:758645 thread_id:0x8ed80d86f9717291 threads:1
Object:1 RealDecimal:1(64) RealBinary:0(0) ephem_objs:yes
used_mem:16060416(kernel) free_mem:32648278016 sys_mem:33619996672

VERBOSE MODE

-- Main menu --

 1) Configurations
 2) Interpreter
 3) Garbage collector tests
 4) Radix 2 vs 10 tests
 5) Laboratory
 6) Show GC status
 7) Collect all & reset GC
 8) Exit

Selection: 2
```

- **1) Configurations** Menu des configurations

```
-- System configurations --

 1) Global                      verbose:on colors:on
 2) Numeric types               RealDecimalMPDec (base10) 20.../100000
 3) Memory                      GC:on/3 trsh:2/5%
 4) Tools                       record:off redirect:latex/pdf
 5) Activate options hints      (explain every options in details)
 6) About
 7) Return to main menu

Selection: 7
```

- **2) Interpreter** Interpréteur *Kpsilon*

```
-- RealDecimalMPDec @ 100000 interpreter --

Type                    help  List all available commands
                        quit  Exit

Press                  [TAB]  Auto complete the input
                 [SHIFT-TAB]  Select the next input hint
           [PAGEUP/PAGEDOWN]  Navigate the hints list
                    [UP/DOWN]  Browse history
        [CTRL-S] or [CTRL-R]  Search history (forward or reverse)
                    [CTRL-C]  Exit


All automatic garbage collections will emit a [GC] message!


Warming up interpreter...done (0.056284 seconds).

kpsilon> set prec 30

 Internal precision changed to 30 decimals.
```

```
kpsilon> set disp 30

 Display precision changed to 30 decimals.

kpsilon> 1/2π

 0.159154943091895335768883763373

kpsilon> (1d1 / 137.035999206d2) * 1b@15

 0.007297352562787

kpsilon>
```

- **3) Garbage collector tests**

  Un exemple de son utilisation se trouve à la section B.1 de l'annexe B.

- **4) Radix 2 vs 10 tests**

  Un exemple de son utilisation se trouve à la section B.2 de l'annexe B.

- **5) Laboratory**

```
-- Laboratory (temporary experimentations) --


1) Voluntary memory leaks test
2) Voluntary uninitialized memory tests
3) Trigger a segmentation fault
4) Misc experimentations
5) Multithreaded GC test
6) Multithreaded Numerics V2 test
7) Return to main menu

Selection: 7
```

- **6) Show GC status**

```
Version           : 8.0.4
Mode              : Normal
Divisor           : 3
Threshold         : 2/5%
Custom allocators : dec: 50%, bin: 50%
Maximum heap size : unlimited
Heap expand       : 0 bytes
Current heap size : 786432 bytes
Incremental mode  : disabled
Ephemeral objects : enabled

Memory            : used 12021760, free 32653086720, total 33619996672 (bytes)

Number of managed allocated memory::Object : 0
```

- **7) Collect all & reset GC**

  Force une collecte complète et réinitialise le *GC*.

- **8) Exit**

  Quitte le logiciel.

A.3 Types numériques multiprécision (`Numeric`)

Comme expliqué au chapitre VII, l'interpréteur *Kpsilon* supporte différents types de valeurs à précision arbitraire, en base 2 et 10. Le tableau A.1 présente les types *`Numeric`* qu'il supporte, alors que les valeurs possibles de ces types sont décrites par le diagramme syntaxique de la figure A.1.

type `Numeric`	base arithmétique	identifiant interpréteur	bibliothèque arithmétique	précision maximale[93]
`RealBinaryGMP`	2	***b1***	*GMP (Boost `gmp_float`)*	$4{,}2 \times 10^9$
`RealBinaryMPFR`	2	***b2***	*MPFR (Boost `mpfr_float`)*	$4{,}2 \times 10^9$
`RealBinaryFloat`	2	***b3***	*N/A (`double` 64 bits)*	15 à 17
`RealDecimalCPPDec`	10	***d1***	*Boost `cpp_dec_float`*	$2{,}1 \times 10^6$
`RealDecimalMPDec`	10	***d2***	*mpdecimal*	$1{,}8 \times 10^{19}$

Tableau A.1 Liste des types `Numeric` disponibles dans *Kpsilon*.

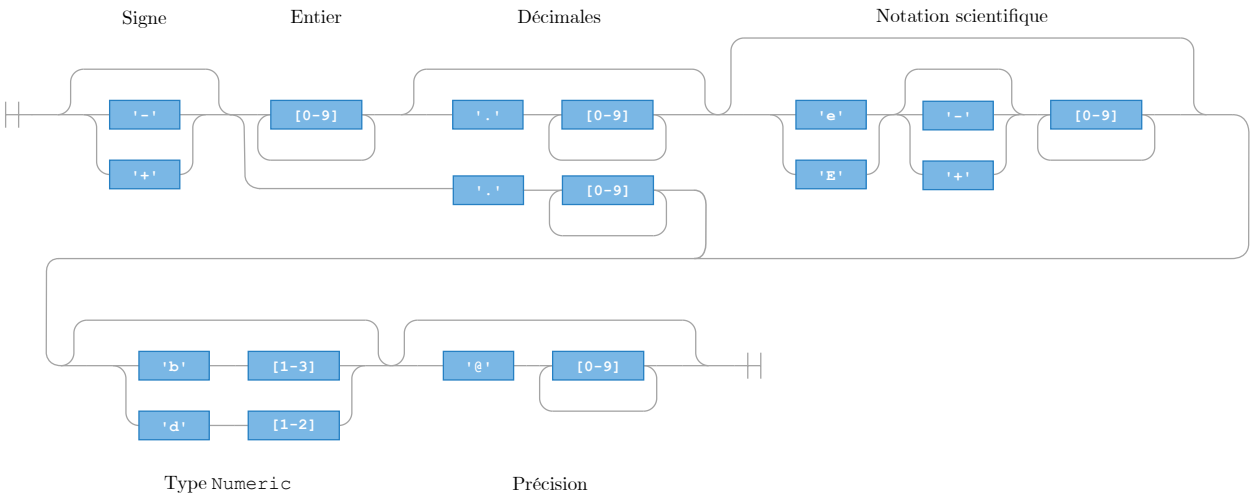


Figure A.1 Diagramme syntaxique des valeurs des types `Numeric`.

[93] C'est-à-dire, la précision maximale en nombre de chiffres (incluant la partie entière).

- **Valeur du dernier résultat retourné**

  _ (tiret bas, *underscore*)

- **Constante** π

  `pi` ou π

  Remarque: *Le code Unicode de* π *est* `1D70B` *(voir la remarque à « nom de variable (`var`) »).*

- **Valeur numérique** (`num`)

  `([+|-]({0-9}+[.{0-9}+] | .{0-9}+)[e[+-]{0-9}+] | 0x{0-9a-f}+)`

  Remarque: *La notation scientifique peut être représentée avec « `e[+-]{0-9}+` » et une une valeur hexadécimale ou une adresse mémoire par « `0x{0-9a-f}+` ».*

- **Définition/conversion de type et précision**

  `(num | (expr))({bd}{1-3} | bin | dec)[@{0-9}+]`

- **Nom de variable** (`var`)

  `([_+]{a-zA-Z}[{_0-9a-zA-Z}+] | lettre_grec)`

  Remarque: *Les codes Unicode de l'alphabet grec se trouvent entre* `U+0001D6E2` *et* `U+0001D714`[94]*. Sous Linux, pour utiliser un code Unicode il faut appuyer sur* `CTRL+SHIFT+u` *(relâcher) code* `RETOUR` *ou copier/coller directement le caractère. À l'invite, utilisez la commande* `syms` *pour afficher la liste des codes Unicode des lettres grecques.*

- **Affectation de variable** (`affect`)

  `var = expr`

- **Commande** (`cmd`)

  `cmd`

  Voir le tableau A.2 pour la liste des commandes supportées.

[94] Liste des codes *Unicode* : `https://www.unicode.org/charts/PDF/U1D400.pdf`.

- **Lecture/déréférencement d'adresse mémoire d'une valeur `Numeric` (*ref*)**

  Retourne l'adresse mémoire d'une valeur `Numeric` :

  `&(`*num* `|` *var* `|` `(`*expr*`))`

  Déréférencement d'une valeur `Numeric` à partir d'une adresse mémoire :

  `@(`*var* `|` `(`*expr*`))`

  Exemple :

```
kpsilon> x = 3

 Timings         : 0.000382s real, 0.000000s CPU (n/a%)
 Memory          : adr 0x7f6e16877b40, used 22736896, free 32644300800 (bytes)
 Data memory     : object 136 bytes, total data 42112 bytes in 8 objects
 Numeric type    : RealDecimalMPDec, prec:100000/100000, mem:136
 Result          : 3
 Hex value       : 0x03
 Output timings : 0.004680s real, 0.010000s CPU (213.7%)

kpsilon> y = &x

 Memory Address : 0x7f6e16877b40 (kpsilon::numeric::RealDecimalMPDec,
 ↪  prec:100000/100000, mem:136)

kpsilon> @y

 Timings         : 0.001375s real, 0.000000s CPU (n/a%)
 Memory          : adr 0x7f6e16877b40, used 22839296, free 32642846720 (bytes)
 Data memory     : object 136 bytes, total data 42112 bytes in 14 objects
 Numeric type    : RealDecimalMPDec, prec:100000/100000, mem:136
 Result          : 3
 Hex value       : 0x03
 Output timings : 0.002853s real, 0.000000s CPU (n/a%)
```

  Bogue: *Au moment de la rédaction, nous avons réalisé que l'ajout des modes de multiplications implicites a brisé cette fonctionnalité — ce bogue sera fixé ultérieurement. Pour le moment, pour la faire fonctionner, il faut ajouter cet argument à la commande de compilation :*

  `-compargs="-DDONT_APPLY_TEMP_IMPLICIT_MUL_BUG_FIX"`

- **Expression (*expr*)**

  `[{+-}][(]`*num* `|` *var* `|` `(_|pi|π)` `|` *ref* `|` *expr*`)[`*expr* `|` `{+-*/}` *expr*`][)]`

- **Entrée complexe (*stm*)**

  `(`*expr* `|` *affect* `|` cmd`)[;` *stm*`]`

A.5 Commandes supportées et exemples (`cmd`)

commande(s)	description	type
`help`	Affiche le manuel d'instructions.	MISC
`quit`	Quitte l'interpréteur.	MISC
`cls`	Efface l'écran.	MISC
`verbose`	Active/désactive l'affichage détaillé.	MISC
`syms`	Affiche la liste de l'alphabet grec (*Unicode*).	MISC
`delete`	Détruit une variable ou un objet en mémoire.	MEM
`remove`	Détruit une variable sans détruire son contenu mémoire.	MEM
`list`	Affiche la liste des variables définies par l'utilisateur.	MEM
`clear`	Détruit toutes les variables définies par l'utilisateur.	MEM
`set type`	Spécifie le type `Numeric` à utiliser par défaut.	MUMERIC
`set prec`	Spécifie la précision à utiliser par défaut.	MUMERIC
`set disp`	Spécifie la précision à afficher par défaut.	MUMERIC
`set div`	Spécifie la valeur du *diviseur du GC*.	MEM
`set freq`	Configure la fréquence des *allocateurs mémoire*.	MEM/NUMERIC
`set mul`	Configure le mode de multiplications implicites par défaut.	NUMERIC
`ephmon et ephmoff`	Active/désactive les *objets éphémères*.	MEM/NUMERIC
`gc et gcfull`	Force une collecte du *GC*.	MEM
`gcreset`	Détruit tous les objets monitorés par le *GC*.	MEM
`gcon et gcoff`	Active/désactive le *GC*.	MEM
`peek et poke`	Lecture et écriture en mémoire.	MEM/DEBUG
`callstack`	Affiche la pile d'exécution (*call stack*).	DEBUG
`mem`	Affiche des informations et l'état de la mémoire.	MEM/DEBUG
`showtree`	Génère l'*arbre syntaxique (CST)* d'une expression.	DEBUG
`tests`	Exécute les tests unitaires *Catch2*.	DEBUG
`dump`	Affiche le contenu d'une adresse mémoire ou d'un objet.	MEM/DEBUG
`dumpgc et dumpgcall`	Affiche l'état du *GC*.	MEM/DEBUG
`dumpheap[s]`	Affiche le contenu mémoire du (des) tas (*heap(s)*).	MEM/DEBUG
`dumpstack`	Affiche le contenu mémoire de la pile (*stack*).	MEM/DEBUG
`dumpbin`	Affiche le contenu mémoire du binaire.	MEM/DEBUG
`traceon et traceoff`	Active, désactive la fonctionnalité de `TRACE`.	DEBUG
`trace`	Active/Sélectionne le niveau de `TRACE`.	DEBUG
`source`	Affiche le code source d'une fonction ou d'un fichier.	DEBUG
`disasm`	Désassemble le code d'une fonction ou d'une adresse mémoire.	DEBUG

Tableau A.2 Liste complète des commandes supportées par l'interpréteur *Kpsilon*.

Important: *Les exemples des différentes commandes ont été générés à partir d'un binaire compilé en mode Debug. Ceci explique la lenteur des calculs lorsque les temps sont affichés dans les exemples. De plus, la présence d'une barre horizontale ondulée (〰〰〰) dans les listings des exemples indique que certaines lignes moins pertinentes ont été omises.*

- **help**

  Affiche le manuel d'instructions.

  Exemple :

```
kpsilon> help

 Expresions

  num          = [+|-] {{0-9}+ [.{0-9}+] | {.{0-9}+} | 0x{0-9a-f}+}
  var          = [_+] {a-zA-Z} [{_0-9a-zA-Z}+] | greek_letter
                 NOTE: Greek letters Unicode U+0001D6E2 to U+0001D714
                       See: https://www.unicode.org/charts/PDF/U1D400.pdf

                       Usage: CTRL+SHIFT+u (release) code ENTER or copy/paste
                       Use the syms command to display a valid Unicodes chart
  expr         = [() [+|-]{num|var|_|expr} [expr | {+|-|*|/} [+|-]expr] [)]
  Constant π   = pi | π (CTRL+SHIFT+u (release) 1D70B ENTER)
  Last answer = _


 Inputs

  {expr|cmd} [; {expr|cmd}]      Expression or command to evaluate
  var=expr                       Affect a result to variable var
  num[d|b][1-9][@prec]           Choose between Decimal and Binary types
  (expr)[d|b][1-9][@prec]        Convert between Decimal and Binary types
  &{num|var|(expr)}              Return the memory address of the input object
  @{addr|var|(expr)}             Derefence a memory address to a Numeric object


 Commands (cmd)

  help                           Show this menu
  syms                           Display a list of greek letters Unicode codes
  verbose [on|off]               Enable/disable verbose mode
  delete {addr|var|(expr)}       Delete a variable or the object at addr
  remove var                     Remove a variable without deleting it's data
  list                           Show the list of affected variables
  clear                          Clear the variables memory
  set type {d[ec]|b[in]}[1-9]    Choose default Numeric type
  set prec prec                  Change the default precision
  set disp prec                  Change the display precision
  set div val                    Change the GC divisor value
  set {dec|bin} freq val         Change the frequency (%) of the GC allocators
  set mul {default|fixed|hybrid} Change the implicit multiplications mode
  ephmon|ephmoff                 Enable/disable ephemeral objects
  gc|gcfull                      Force a GC collection (normal or full)
  gcreset                        Delete all Objects and reset the GC
  gcon|gcoff                     Enable/disable the GC
  peek {addr|var|(expr)} [size]  Read size (default 8) bytes from memory
  poke {addr|var|(expr)}
       {data|var|(expr)} [size]  Write data of size (default 8) bytes to memory
  dump [addr|var|(expr)] [size]  Dump memory of ans/address/variable/expression
```

```
dumpgc|dumpgcall                Dump the GC status (normal or detailed)
dumpheap[s]|dumpstack|dumpbin   Dump specific memory sections
callstack                       Dump the callstack
mem                             Dump the memory state
source {file[:function]|function}
       [lines]                  Show a source code listing
disasm {addr|file[:function]|function}
       [lines]                  Dump a disassembled code listing
showtree                        Show last input in the parser tree inspector
tests                           Execute all unit tests
traceon|traceoff|trace{1-5}     Change the TRACE levels
cls                             Clear screen
quit                            Exit
```

Press

```
[TAB]                           Auto complete the input
[SHIFT-TAB]                     Select the next input hint
[PAGEUP/PAGEDOWN]               Navigate the hints list
[UP/DOWN]                       Browse history
[CTRL-S] or [CTRL-R]            Search history (forward or reverse)
[CTRL-C]                        Exit
```

Examples

```
kpsilon> set prec 2000
kpsilon> set disp 2000
kpsilon> -0.4 / -3 + (1 + 1323) / 4213231312
kpsilon> (1d1 / 137.035999206d) * 1b@15
kpsilon> set prec 1000
kpsilon> set type dec
kpsilon> _ / 2pi
kpsilon> gc
kpsilon> dumpgc
kpsilon> x = 1/3; dump (peek (&x + 0xa0))
kpsilon> quit
```

- **syms**

  Affiche la liste des *caractères Unicode* (alphabet grec) supportés comme noms de variables.

  Exemple :

```
kpsilon> syms

  Usage: CTRL+SHIFT+u (release) code ENTER or copy/paste

  names       capital   small  codes

  alpha       Α         α      1D6E2  1D6FC
  beta        Β         β      1D6E3  1D6FD
  gamma       Γ         γ      1D6E4  1D6FE
  delta       Δ         δ      1D6E5  1D6FF
  epsilon     ε         ϵ      1D6E6  1D700
  zeta        Ζ         ζ      1D6E7  1D701
  eta         Η         η      1D6E8  1D702
  theta       Θ         θ      1D6E9  1D703
  iota        Ι         ι      1D6EA  1D704
  kappa       Κ         κ      1D6EB  1D705
  lambda      Λ         λ      1D6EC  1D706
  mu          Μ         μ      1D6ED  1D707
  nu          Ν         ν      1D6EE  1D708
  xi          Ξ         ξ      1D6EF  1D709
  omicron     Ο         o      1D6F0  1D70A
  pi (const)  Π         π      1D6F1  1D70B
  rho         Ρ         ρ      1D6F2  1D70C
  sigma       Σ         σ      1D6F4  1D70E
  tau         Τ         τ      1D6F5  1D70F
  upsilon     Υ         υ      1D6F6  1D710
  phi         Φ         φ      1D6F7  1D711
  chi         Χ         χ      1D6F8  1D712
  psi         Ψ         ψ      1D6F9  1D713
  omega       Ω         ω      1D6FA  1D714
```

- **verbose [on | off]**

  Active/désactive l'affichage détaillé.

  Exemple :

```
kpsilon> verbose on

 Verbose mode is enabled.

kpsilon> 1/3

 Timings        : 0.079363s real, 0.070000s CPU (88.2%)
 Memory         : adr 0x7f9d0a93ea80, used 226648064, free 5060227072 (bytes)
 Data memory    : object 21312 bytes, total data 84224 bytes in 7 objects
 Numeric type   : RealDecimalMPDec, prec:100000/100000, mem:21312
 Result         : 0.33333333333333333333...
 Output timings : 0.162052s real, 0.100000s CPU (61.7%)

kpsilon> verbose off

 Verbose mode is disabled.

kpsilon> 1/3

 0.33333333333333333333...
```

- **delete** (*addr* | *var* | (*expr*))

  Détruit une variable ou un objet (par adresse mémoire, nom ou une adresse calculée par une expression).

  Exemple :

```
kpsilon> x=1;y=2;z=3

 3

kpsilon> list

 Variables list :

 1) _ : 3 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)
        @ 0x7fafccc026c0
 2) x : 1 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)
        @ 0x7fafccc02b40
 3) y : 2 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)
        @ 0x7fafccc02900
 4) z : 3 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)
        @ 0x7fafccc026c0

 Total memory (bytes)         : 768
 Maximum number of variables : 329406144173384850

kpsilon> delete 0x7fafccc02900

 Numeric object @ 0x7fafccc02900 deleted
 Object infos : 2 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)

kpsilon> delete x

 Variable x and Numeric object @ 0x7fafccc02b40 deleted
 Object infos : 1 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)

kpsilon> delete (0x7fafccc02600 + 0xc0)

 Numeric object @ 0x7fafccc026c0 deleted
 Object infos : 3 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)
```

- **remove *var***

  Détruit une variable sans détruire son contenu mémoire.

  Exemple :

```
kpsilon> x=1/3

 0.33333333333333333333...

kpsilon> list

 Variables list :

 1) _ : 0.33333333333333333333 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000,
 ↪  mem:21312)
        @ 0x7f018ed24a80
 2) x : 0.33333333333333333333 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000,
 ↪  mem:21312)
        @ 0x7f018ed24a80

 Total memory (bytes)         : 21312
 Maximum number of variables : 329406144173384850

kpsilon> remove x

 Variable x removed

kpsilon> list

 Variables list :

 1) _ : 0.33333333333333333333 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000,
 ↪  mem:21312)
        @ 0x7f018ed24a80

 Total memory (bytes)         : 21312
 Maximum number of variables : 329406144173384850
```

- **list**

  Affiche la liste des variables définies par l'utilisateur.

  Exemple :

```
kpsilon> x=1;y=2;z=3

 3

kpsilon> list

 Variables list :

 1) _ : 3 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)
        @ 0x7fafccc026c0
 2) x : 1 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)
        @ 0x7fafccc02b40
 3) y : 2 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)
        @ 0x7fafccc02900
 4) z : 3 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)
        @ 0x7fafccc026c0

 Total memory (bytes)         : 768
 Maximum number of variables : 329406144173384850
```

- **clear**

  Détruit toutes les variables définies par l'utilisateur.

  Exemple :

```
kpsilon> x=1;y=2;z=3

 3

kpsilon> list

 Variables list :

 1) _ : 3 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)
        @ 0x7f993e4666c0
 2) x : 1 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)
        @ 0x7f993e466b40
 3) y : 2 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)
        @ 0x7f993e466900
 4) z : 3 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)
        @ 0x7f993e4666c0

 Total memory (bytes)         : 768
 Maximum number of variables : 329406144173384850

kpsilon> clear

 Memory cleared.

kpsilon> list

 Variables list :

 1) _ : 3 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)
        @ 0x7f993e4666c0

 Total memory (bytes)         : 256
 Maximum number of variables : 329406144173384850
```

- **set type (d[ec] | b[in])[*1-9*]**

  Spécifie le type *Numeric* à utiliser par défaut.

  La liste des types *Numeric* supportés se trouve dans le tableau A.1.

  Exemple :

```
kpsilon> set type b1

 Default Numeric type set to RealBinaryGMP.

kpsilon> 1/3

 Timings        : 0.604220s real, 0.230000s CPU (38.1%)
 Memory         : adr 0x7ff2a9f09a80, used 260534272, free 4786962432 (bytes)
 Data memory    : object 41760 bytes, total data 124656 bytes in 7 objects
 Numeric type   : RealBinaryGMP, prec:100000/100000, mem:41760
 Result         : 0.33333333333333333333...
 Output timings : 0.482087s real, 0.150000s CPU (31.1%)

kpsilon> set type b2

 Default Numeric type set to RealBinaryMPFR.

kpsilon> 1/3

 Timings        : 0.790753s real, 0.250000s CPU (31.6%)
 Memory         : adr 0x7ff2a9f09780, used 307785728, free 4789710848 (bytes)
 Data memory    : object 41752 bytes, total data 124632 bytes in 11 objects
 Numeric type   : RealBinaryMPFR, prec:100000/100000, mem:41752
 Result         : 0.33333333333333333333...
 Output timings : 0.415709s real, 0.090000s CPU (21.6%)

kpsilon> set type b3

 Default Numeric type set to RealBinaryFloat.

kpsilon> 1/3

 Timings        : 0.243914s real, 0.110000s CPU (45.1%)
 Memory         : adr 0x7ff2a9f09480, used 316968960, free 4765204480 (bytes)
 Data memory    : object 192 bytes, total data 0 bytes in 15 objects
 Numeric type   : RealBinaryFloat, prec:100000/17, mem:192
 Result         : 0.33333333333333331
 Output timings : 0.100562s real, 0.030000s CPU (29.8%)

kpsilon> set type d1

 Default Numeric type set to RealDecimalCPPDec.

kpsilon> 1/3

 Timings        : 3.307495s real, 1.730000s CPU (52.3%)
 Memory         : adr 0x7ff2a9f09180, used 558542848, free 4572368896 (bytes)
 Data memory    : object 100240 bytes, total data 0 bytes in 19 objects
 Numeric type   : RealDecimalCPPDec, prec:100000/100000, mem:100024
 Result         : 0.33333333333333333333...
 Output timings : 0.085639s real, 0.030000s CPU (35.0%)

kpsilon> set type d2

 Default Numeric type set to RealDecimalMPDec.

kpsilon> 1/3
```

```
 Timings        : 0.229054s real, 0.100000s CPU (43.7%)
 Memory         : adr 0x7ff290c29e40, used 559026176, free 4570767360 (bytes)
 Data memory    : object 21312 bytes, total data 84224 bytes in 23 objects
 Numeric type   : RealDecimalMPDec, prec:100000/100000, mem:21312
 Result         : 0.33333333333333333333...
 Output timings : 0.099913s real, 0.030000s CPU (30.0%)

kpsilon> set type bin

 Default Numeric type set to RealBinaryFloat.

kpsilon> 1/3

 Timings        : 0.233148s real, 0.120000s CPU (51.5%)
 Memory         : adr 0x7ff290c29b40, used 559026176, free 4567048192 (bytes)
 Data memory    : object 192 bytes, total data 0 bytes in 27 objects
 Numeric type   : RealBinaryFloat, prec:100000/17, mem:192
 Result         : 0.33333333333333331
 Output timings : 0.119188s real, 0.040000s CPU (33.6%)

kpsilon> set type dec

 Default Numeric type set to RealDecimalMPDec.

kpsilon> 1/3

 Timings        : 0.181876s real, 0.100000s CPU (55.0%)
 Memory         : adr 0x7ff290c29840, used 559071232, free 4562337792 (bytes)
 Data memory    : object 21312 bytes, total data 147392 bytes in 31 objects
 Numeric type   : RealDecimalMPDec, prec:100000/100000, mem:21312
 Result         : 0.33333333333333333333...
 Output timings : 0.112467s real, 0.040000s CPU (35.6%)
```

- **set prec *prec***

  Spécifie la précision à utiliser par défaut.

  Exemple :

```
kpsilon> set prec 10

 Internal precision changed to 10 decimals.

kpsilon> 1/3

 Timings        : 0.059989s real, 0.040000s CPU (66.7%)
 Memory         : adr 0x7f40dd723180, used 238878720, free 4802908160 (bytes)
 Data memory    : object 256 bytes, total data 42112 bytes in 19 objects
 Numeric type   : RealDecimalMPDec, prec:10/10, mem:256
 Result         : 0.3333333333
 Output timings : 0.022254s real, 0.020000s CPU (89.9%)

kpsilon> set prec 100

 Internal precision changed to 100 decimals.

kpsilon> 1/3

 Timings        : 0.065316s real, 0.050000s CPU (76.6%)
 Memory         : adr 0x7f40c4426e40, used 238878720, free 4817772544 (bytes)
 Data memory    : object 280 bytes, total data 42160 bytes in 23 objects
 Numeric type   : RealDecimalMPDec, prec:100/100, mem:280
 Result         :
 ↪  0.3333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
 ↪  333333333333333333
 Output timings : 0.020485s real, 0.020000s CPU (97.6%)
```

- **set disp *prec***

  Spécifie la précision à afficher à l'écran.

  Exemple :

```
kpsilon> set prec 100

 Internal precision changed to 100 decimals.

kpsilon> set disp 100

 Display precision changed to 100 decimals.

kpsilon> 1/3

 Timings        : 0.067405s real, 0.050000s CPU (74.2%)
 Memory         : adr 0x7fded6f2ba80, used 226697216, free 4899647488 (bytes)
 Data memory    : object 280 bytes, total data 42160 bytes in 7 objects
 Numeric type   : RealDecimalMPDec, prec:100/100, mem:280
 Result         :
 ↪  0.3333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
 ↪  333333333333333333
 Output timings : 0.089578s real, 0.080000s CPU (89.3%)

kpsilon> set disp 10

 Display precision changed to 10 decimals.

kpsilon> 1/3

 Timings        : 0.064398s real, 0.050000s CPU (77.6%)
 Memory         : adr 0x7fded6f2b780, used 239292416, free 4890497024 (bytes)
 Data memory    : object 280 bytes, total data 42232 bytes in 11 objects
 Numeric type   : RealDecimalMPDec, prec:100/100, mem:280
 Result         : 0.3333333333...
 Output timings : 0.028291s real, 0.030000s CPU (106.0%)
```

- **`set div val`**

  Spécifie la valeur du *diviseur du GC* (un nombre entier).

  Exemple :

```
kpsilon> set div 1
 GC divisor value changed to 1.
kpsilon> set div 100
 GC divisor value changed to 100.
kpsilon> set div 1000
 GC divisor value changed to 1000.
```

- **`set (dec | bin) freq val`**

  Configure la fréquence (un pourcentage représenté par un réel entre 0 et 1) des *allocateurs mémoire* des types `Numeric`.

  Exemple :

```
kpsilon> set bin freq 0.5
 Binary allocators frequency set to 0.5.
kpsilon> set dec freq 1
 Decimal allocators frequency set to 1.
kpsilon> set bin freq 0
 Binary allocators frequency set to 0.
```

- **`set mul (default | fixed | hybrid)`**

  Configure le mode de multiplications implicites par défaut (sous-section 5.4.2).

  Exemple :

```
kpsilon> x = 2
 2
kpsilon> 1/(2x)
 0.25
kpsilon> set mul default
 Implicit multiplications set to default mode.
kpsilon> 6/2(1 + 2)
 1
kpsilon> 1/2x
 0.25
kpsilon> set mul fixed
 Implicit multiplications set to fixed mode.
kpsilon> 6/2(1 + 2)
 9
kpsilon> 1/2x
 1
kpsilon> set mul hybrid
 Implicit multiplications set to hybrid mode.
kpsilon> 6/2(1 + 2)
 9
kpsilon> 1/2x
 0.25
```

- **ephmon** et **ephmoff**

  Active/désactive les *objets éphémères*.

  Exemple :

```
kpsilon> ephmoff

 Ephemeral objects are now disabled.

kpsilon> -0.4 / -3 + (1 + 1323) / 4213231312

 0.13333364758144884244...

kpsilon> dumpgc

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 Number of managed allocated memory::Object : 11

   0x7f9301fb7000 -> 0.1333336476 (RealDecimalMPDec, prec:100000/100000, mem:21312)
   0x7f9301fb7540 -> -3 (RealDecimalMPDec, prec:100000/100000, mem:256)
   0x7f9301fb7180 -> 4213231312 (RealDecimalMPDec, prec:100000/100000, mem:256)
   0x7f9301fb76c0 -> 0 (RealDecimalMPDec, prec:100000/100000, mem:256)
   0x7f9301fb7300 -> 1323 (RealDecimalMPDec, prec:100000/100000, mem:256)
   0x7f9301fb70c0 -> 0.0000003142 (RealDecimalMPDec, prec:100000/100000, mem:21312, ephm)
   0x7f9301fb7480 -> 0.1333333333 (RealDecimalMPDec, prec:100000/100000, mem:21312, ephm)
   0x7f9301fb7240 -> 1324 (RealDecimalMPDec, prec:100000/100000, mem:256)
   0x7f9301fb7600 -> -0.4 (RealDecimalMPDec, prec:100000/100000, mem:256)
   0x7f9301fb7780 -> 0 (RealDecimalMPDec, prec:100000/100000, mem:21312)
   0x7f9301fb73c0 -> 1 (RealDecimalMPDec, prec:100000/100000, mem:256)

 Total memory used by the GC allocators : 87040 bytes

kpsilon> gcreset

 11 Objects were deleted.

kpsilon> ephmon

 Ephemeral objects are now enabled.

kpsilon> -0.4 / -3 + (1 + 1323) / 4213231312

 0.13333364758144884244...

kpsilon> dumpgc

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 Number of managed allocated memory::Object : 9

   0x7f9301fb7cc0 -> 0.1333336476 (RealDecimalMPDec, prec:100000/100000, mem:21312)
   0x7f9301fb7e40 -> 4213231312 (RealDecimalMPDec, prec:100000/100000, mem:256)
   0x7f9301fb7a80 -> 1324 (RealDecimalMPDec, prec:100000/100000, mem:256)
   0x7f9301fb7840 -> 1323 (RealDecimalMPDec, prec:100000/100000, mem:256)
   0x7f9301fb7c00 -> 1 (RealDecimalMPDec, prec:100000/100000, mem:256)
   0x7f9301fb79c0 -> 0 (RealDecimalMPDec, prec:100000/100000, mem:21312)
   0x7f9301fb7d80 -> -3 (RealDecimalMPDec, prec:100000/100000, mem:256)
   0x7f9301fb7b40 -> -0.4 (RealDecimalMPDec, prec:100000/100000, mem:256)
   0x7f9301fb7f00 -> 0 (RealDecimalMPDec, prec:100000/100000, mem:256)

 Total memory used by the GC allocators : 44416 bytes
```

- **gc** et **gcfull**

  Force une collecte du *GC*.

  Remarque: *`gcfull` force une collecte complète.*

  Exemple :

```
kpsilon> gcreset

 3 Objects were deleted.

kpsilon> -0.4 / -3 + (1 + 1323) / 4213231312

 0.13333364758144884244...

kpsilon> dumpgc

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 Number of managed allocated memory::Object : 9

   0x7f89db09ccc0 -> -3 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f89db09ce40 -> 1 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f89db09ca80 -> 1324 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f89db09cc00 -> 0 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f89db09c840 -> 0.1333336476 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f89db09c9c0 -> 4213231312 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f89db09cd80 -> 0 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f89db09cb40 -> 1323 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f89db09cf00 -> -0.4 (RealBinaryGMP, prec:100000/100000, mem:41760)

 Total memory used by the GC allocators : 375840 bytes

kpsilon> gc

 GC called, Numeric objects: 9->3, allocators memory: 373968->124656 bytes.

kpsilon> dumpgc

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 Number of managed allocated memory::Object : 3

   0x7f89db09ccc0 -> -3 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f89db09c840 -> 0.1333336476 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f89db09cf00 -> -0.4 (RealBinaryGMP, prec:100000/100000, mem:41760)

 Total memory used by the GC allocators : 125280 bytes
```

- **gcreset**

  Détruit tous les objets monitorés par le *GC*.

  Exemple :

```
kpsilon> -0.4 / -3 + (1 + 1323) / 4213231312

 0.13333364758144884244...

kpsilon> dumpgc

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 Number of managed allocated memory::Object : 9

   0x7f5f3f599540 -> 0 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f5f3f599900 -> 1323 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f5f3f599840 -> 1324 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f5f3f599c00 -> -3 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f5f3f5999c0 -> 1 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f5f3f599600 -> 0.1333336476 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f5f3f599b40 -> 0 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f5f3f599f00 -> -0.4 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f5f3f599780 -> 4213231312 (RealBinaryGMP, prec:100000/100000, mem:41760)

 Total memory used by the GC allocators : 375840 bytes

kpsilon> gcreset

 9 Objects were deleted.

kpsilon> dumpgc

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 Number of managed allocated memory::Object : 0
```

- **gcon** et **gcoff**

  Active/désactive le *GC*.

  Exemple :

```
kpsilon> gcoff

 The garbage collector is now disabled.

kpsilon> -0.4 / -3 + (1 + 1323) / 4213231312

 Timings         : 0.234283s real, 0.180000s CPU (76.8%)
 Memory          : adr 0x7fa35fc6d780, used 249806848, free 4935524352 (bytes)
 Data memory     : object 415512 bytes, total data 3737728 bytes in 8 objects
 Numeric type    : RealBinaryGMP, prec:1000000/1000000, mem:415512
 Result          : 0.13333364758144884244...
 Output timings : 1.543173s real, 1.530000s CPU (99.1%)

kpsilon> gc

 GC disabled!

kpsilon> gcon

 The garbage collector is now enabled.

kpsilon> -0.4 / -3 + (1 + 1323) / 4213231312

 Timings         : 0.205003s real, 0.160000s CPU (78.0%)
 Memory          : adr 0x7fa35fc6d180, used 265596928, free 4922232832 (bytes)
 Data memory     : object 415512 bytes, total data 7060160 bytes in 16 objects
 Numeric type    : [GC]RealBinaryGMP, prec:1000000/1000000, mem:415512
 Result          : [GC]0.13333364758144884244...
[GC]Output timings : 1.623868s real, 1.480000s CPU (91.1%)

kpsilon> gc

 GC called, Numeric objects: 6->4, allocators memory: 2907120->2076512 bytes.
```

  Remarque: *Notons la présence des* [GC] *dans le listing de l'exemple : ils indiquent une collecte du GC lorsque ce dernier est activé.*

- **peek** et **poke**

  Lecture (peek) et écriture (poke) en mémoire de « *size* » octets (8 par défaut).

  Important: *Ces commandes ne seront pas activées dans la version Release : puisqu'elles permettent de modifier la mémoire du logiciel, elles sont dangereuses ! Elles ne doivent donc être utilisées que dans la version Debug.*

  Utilisation :

  peek (*addr* | *var* | (*expr*) [*size*]

  poke (*addr* | *var* | (*expr*)) (*data* | *var* | (*expr*)) [*size*]

  Exemple :

```
kpsilon> x=1

 1

kpsilon> dump x

 Mem Value  : 1.00000000000000000000
 Formated   : 1
 Hex value  : 0x01
 Mem addr   : 0x7fdecfbe9f00
 Name       : kpsilon::numeric::RealDecimalMPDec
 Type       : numeric::RealDecimalMPDec [0x10 offset:0x0a]
 Sub Type   : numeric::Real [0x07 offset:0x0b]
 Precision  : 20 (real:20) [0x0000000000000014 offset:0x88/0x0000000000000014 offset:0x90]
 Ephemeral  : no [0x00 offset:0x09]
 Is address : no [0x00 offset:0x98]
 Size       : 256 (obj:160 + dataobj:96 + data:0) bytes
 Data addr  : 0x7fdecfbe2eb0 [offset:0xa0]
 Data type  : decimal::Decimal
 Allocators : yes (50%)
 Data size  : 0 (expected:10) bytes

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

kpsilon> dumpgc

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 Number of managed allocated memory::Object : 5

   0x7fdecfbe9900 -> 1 (RealDecimalMPDec, prec:20/20, mem:256)
   0x7fdeebdf1f00 -> 1 (RealDecimalMPDec, prec:20/20, mem:256)
   0x7fdecfbe9f00 -> 1 (RealDecimalMPDec, prec:20/20, mem:256)
   0x7fdeebdf13c0 -> 1 (RealDecimalMPDec, prec:20/20, mem:256)
   0x7fdecfbe93c0 -> 0 (RealDecimalMPDec, prec:20/20, mem:256)

 Total memory used by the GC allocators : 1280 bytes

kpsilon> ephm_offset = &x + 0x09

 Memory Address : 0x7fdecfbe9f09 (kpsilon::numeric::RealDecimalMPDec, prec:20/20, mem:256)

kpsilon> peek ephm_offset 1

 0
```

```
kpsilon> poke ephm_offset 0x01 1

 1

kpsilon> peek ephm_offset 1

 1

kpsilon> dump x

 Mem Value  : 1.00000000000000000000
 Formated   : 1
 Hex value  : 0x01
 Mem addr   : 0x7fdecfbe9f00
 Name       : kpsilon::numeric::RealDecimalMPDec
 Type       : numeric::RealDecimalMPDec [0x10 offset:0x0a]
 Sub Type   : numeric::Real [0x07 offset:0x0b]
 Precision  : 20 (real:20) [0x0000000000000014 offset:0x88/0x0000000000000014 offset:0x90]
 Ephemeral  : yes [0x01 offset:0x09]
 Is address : no [0x00 offset:0x98]
 Size       : 256 (obj:160 + dataobj:96 + data:0) bytes
 Data addr  : 0x7fdecfbe2eb0 [offset:0xa0]
 Data type  : decimal::Decimal
 Allocators : yes (50%)
 Data size  : 0 (expected:10) bytes

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

kpsilon> dumpgc

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 Number of managed allocated memory::Object : 17

   0x7fdecfbe9240 -> 0x7fdecfbe9f09 (RealDecimalMPDec, prec:20/20, mem:256)
   0x7fdecfbe9000 -> 9 (RealDecimalMPDec, prec:20/20, mem:256)
   0x7fdeebdf1600 -> 1 (RealDecimalMPDec, prec:8/8, mem:256)
   0x7fdeebdf1c00 -> 1 (RealDecimalMPDec, prec:8/8, mem:256)
   0x7fdecfbe9c00 -> 1 (RealDecimalMPDec, prec:8/8, mem:256)
   0x7fdeebdf10c0 -> 1 (RealDecimalMPDec, prec:8/8, mem:256)
   0x7fdecfbe90c0 -> 0x7fdecfbe9f00 (RealDecimalMPDec, prec:20/20, mem:256)
   0x7fdeebdf16c0 -> 0 (RealDecimalMPDec, prec:8/8, mem:256)
   0x7fdecfbe96c0 -> 0 (RealDecimalMPDec, prec:8/8, mem:256)
   0x7fdecfbe9300 -> 1 (RealDecimalMPDec, prec:20/20, mem:256)
   0x7fdecfbe9900 -> 1 (RealDecimalMPDec, prec:20/20, mem:256)
   0x7fdeebdf1f00 -> 1 (RealDecimalMPDec, prec:20/20, mem:256)
   0x7fdecfbe9f00 -> 1 (RealDecimalMPDec, prec:20/20, mem:256, ephm)
   0x7fdeebdf13c0 -> 1 (RealDecimalMPDec, prec:20/20, mem:256)
   0x7fdecfbe93c0 -> 0 (RealDecimalMPDec, prec:20/20, mem:256)
   0x7fdeebdf19c0 -> 140594944843529 (RealDecimalMPDec, prec:20/20, mem:256)
   0x7fdeebdf1d80 -> 1 (RealDecimalMPDec, prec:20/20, mem:256, ephm)

 Total memory used by the GC allocators : 4352 bytes
```

Explications: *Cet exemple modifie la mémoire de la variable « x » (un objet `Numeric`) pour la transformer en* objet éphémère *(`ephem`) en remplaçant la valeur 0 par un 1 dans l'espace mémoire de son flag « Ephemeral ». Ce flag (un booléen d'un octet) se trouve à l'offset `0x09` relativement à l'adresse mémoire d'un objet `Numeric` (`0x7fdecfbe9f00` dans cet exemple).*

- **dump** (*addr* | *var* | (expr)) [*size*]

Affiche le contenu d'une adresse mémoire ou d'un objet (avec informations).

Exemple :

```
kpsilon> x=1/3

 Timings        : 0.155240s real, 0.130000s CPU (83.7%)
 Memory         : adr 0x7f2b16732300, used 265531392, free 4723576832 (bytes)
 Data memory    : object 240 bytes, total data 499424 bytes in 37 objects
 Numeric type   : RealBinaryGMP, prec:20/20, mem:240
 Result         : 0.33333333333333333333
 Output timings : 0.022940s real, 0.020000s CPU (87.2%)

kpsilon> dump x

 Mem Value   : 0.33333333333333333333
 Formated    : 0.33333333333333333333
 Mem addr    : 0x7f2b16732300
 Name        : kpsilon::numeric::RealBinaryGMP
 Type        : numeric::RealBinaryGMP [0x0a offset:0x0a]
 Sub Type    : numeric::Real [0x07 offset:0x0b]
 Precision   : 20 (real:20) [0x0000000000000014 offset:0x88/0x0000000000000014 offset:0x90]
 Ephemeral   : no [0x00 offset:0x09]
 Is address  : no [0x00 offset:0x98]
 Size        : 240 (obj:160 + dataobj:48 + data:32) bytes
 Data addr   : 0x7f2b166b6de0 [offset:0xa0]
 Data type   :
 ↪  boost::multiprecision::number<boost::multiprecision::backends::gmp_float<0>,
 ↪  (boost::multiprecision::expression_template_option)1>
 Allocators : yes (50%)
 Data size   : 32 (expected:32) bytes

 DEBUG informations [offset:0x30-0x87] :

   Created    : 2 secs ago (Sun Aug 18 22:28:14 2024)
   Modified   : 2 secs ago (Sun Aug 18 22:28:14 2024)
   Accessed   : 2 secs ago (Sun Aug 18 22:28:14 2024)
   Used mem   : 265527296 bytes
   Free mem   : 4723847168 bytes
   Stacktrace : 0x0000007742dc kpsilon::memory::Object::Object(kpsilon::memory::ObjType
 ↪  const&, kpsilon::memory::ObjType const&)() at memory/object.hpp:168
                0x000000812b86
 ↪  kpsilon::numeric::NumericTmplBase<kpsilon::numeric::RealBinaryGMP,
 ↪  kpsilon::numeric::Numeric,
 ↪  boost::multiprecision::number<boost::multiprecision::backends::gmp_float<0u>,
 ↪  (boost::multiprecision::expression_template_option)1>
 ↪  >::NumericTmplBase(kpsilon::memory::ObjType const&, kpsilon::memory::ObjType
 ↪  const&)() at numeric_v2/numeric_tmpl_base.ipp:134
                0x0000007dfa50 kpsilon::numeric::RealBinaryGMP::RealBinaryGMP(unsigned
 ↪  long const&)() at numeric_v2/real_binary_gmp.cpp:81
                0x0000008cbdd2
 ↪  kpsilon::numeric::RealBinaryGMP::operator/(kpsilon::numeric::Numeric const&)
 ↪  const() at numeric_v2/real_binary_gmp_dispatch.preprocessed.cpp:1113
                0x00000059565a kpsilon::KpsilonVisitor::visitMultOrDiv(kpsilon::KpsilonP⌋
 ↪  arser::MultOrDivContext*)() at
 ↪  interpreter/kpsilon_visitor.cpp:1765
```

```
                0x0000005fcc17
 ↪  antlr4::tree::AbstractParseTreeVisitor::visitChildren(antlr4::tree::ParseTree*)()
 ↪  at antlr4-runtime/tree/AbstractParseTreeVisitor.h:43
                0x000000518751
 ↪  kpsilon::KpsilonVisitor::visitInput(kpsilon::KpsilonParser::InputContext*)() at
 ↪  interpreter/kpsilon_visitor.cpp:180
```

```
                0x00000064b6d3 kpsilon::demos::Demo::calculator(unsigned long, bool)()
 ↪  at lab/demo_calc.cpp:2053
                0x0000004e4bc3 main() at main.cpp:1143
                0x7f2b2cdce840 __libc_start_main
                0x0000004e0d59 _start

  Interpreter history :

    x=1/3 at line 1 (2 secs ago)

NATVIS DEBUG string : 0.33333333333333333333 (kpsilon::numeric::RealBinaryGMP,
 ↪  prec:20/20, mem:240, ephm) [offset:0x10-0x2f]

Memory dump :

0x7f2b16732300 40 cf c2 01 00 00 00 00 00 0a 07 00 00 00 00 @...............
0x7f2b16732310 80 8d 6b 16 2b 7f 00 00 53 00 00 00 00 00 00 00 ..k.+...S.......
0x7f2b16732320 53 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 S...............
0x7f2b16732330 00 a0 d3 0f 00 00 00 00 00 30 90 19 01 00 00 00 .........0......
0x7f2b16732340 00 5c 74 16 2b 7f 00 00 40 5e 74 16 2b 7f 00 00 .\ t.+...@^t.+...
0x7f2b16732350 00 60 74 16 2b 7f 00 00 30 90 6d 16 2b 7f 00 00 .`t.+...0.m.+...
0x7f2b16732360 60 90 6d 16 2b 7f 00 00 60 90 6d 16 2b 7f 00 00 `.m.+...`.m.+...
0x7f2b16732370 be ad c2 66 00 00 00 00 be ad c2 66 00 00 00 00 ...f.......f....
0x7f2b16732380 be ad c2 66 00 00 00 00 14 00 00 00 00 00 00 00 ...f............
0x7f2b16732390 14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x7f2b167323a0 e0 6d 6b 16 2b 7f 00 00 20 00 00 00 00 00 00 00 .mk.+... .......

kpsilon> dump 0x7f2b166b6de0 20

Value      : 0.33333333333333333333
Data type  :
 ↪  boost::multiprecision::number<boost::multiprecision::backends::gmp_float<0>,
 ↪  (boost::multiprecision::expression_template_option)1>
Mem addr   : 0x7f2b166b6de0
Owner addr : 0x7f2b16732300
Owner name : kpsilon::numeric::RealBinaryGMP
Allocators : yes (50%)
Mem size   : 32 bytes

Memory dump :

0x7f2b166b6de0 03 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00 ................
0x7f2b166b6df0 70 c5 d8 02 00 00 00 00 14 00 00 00 00 00 00 00 p...............
0x7f2b166b6e00 6d 6f 76 65 5f 63 75 72 73 6f 72 5f 6f 6e 65 5f move_cursor_one_
0x7f2b166b6e10 73 75 62 77 6f 72 64 5f 72 69 67 68 74 00 00 00 subword_right...
0x7f2b166b6e20 e6 da 15 2f 2b 7f 00 00 00 00 00 00 00 00 00 00 .../+...........
0x7f2b166b6e30 0b 00 00 00 00 00 00 00 80 ca a1 2f 2b 7f 00 00 ........../+...
0x7f2b166b6e40 6b 69 6c 6c 5f 74 6f 5f 65 6e 64 5f 6f 66 5f 77 kill_to_end_of_w
0x7f2b166b6e50 6f 72 64 00 00 00 00 00 00 00 00 00 00 00 00 00 ord.............
0x7f2b166b6e60 20 57 6f 16 2b 7f 00 00 60 6c 72 16 2b 7f 00 00  Wo.+...`lr.+...
0x7f2b166b6e70 88 51 ab 30 95 f9 c8 c7 00 00 00 00 00 00 00 00 .Q.0............
0x7f2b166b6e80 49 14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 I...............
0x7f2b166b6e90 10 21 d8 02 00 00 00 00 a0 86 01 00 00 00 00 00 .!..............
0x7f2b166b6ea0 03 00 00 00 01 00 00 00 01 00 00 00 00 00 00 00 ................
0x7f2b166b6eb0 30 bf d1 02 00 00 00 00 14 00 00 00 00 00 00 00 0...............
0x7f2b166b6ec0 00 6e 71 16 2b 7f 00 00 00 00 00 00 00 00 00 00 .nq.+...........
0x7f2b166b6ed0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x7f2b166b6ee0 30 78 30 30 30 30 30 30 34 65 30 64 35 39 20 5f 0x0000004e0d59 _
0x7f2b166b6ef0 73 74 61 72 74 00 00 00 00 00 00 00 00 00 00 00 start...........
0x7f2b166b6f00 32 30 32 34 2d 30 38 2d 31 38 20 32 32 3a 32 38 2024-08-18 22:28
0x7f2b166b6f10 3a 34 31 2e 33 38 38 00 00 00 00 00 00 00 00 00 :41.388.........
```

- **dumpgc** et **dumpgcall**

  Affiche l'état du *GC*.

  Remarque: *La commande « `dumpgcall` » affiche toutes les informations accessibles au GC.*

  Exemple (dumpgc) :

```
kpsilon> -0.4 / -3 + (1 + 1323) / 4213231312

 Timings         : 0.292176s real, 0.280000s CPU (95.8%)
 Memory          : adr 0x7f7502229840, used 260743168, free 4833083392 (bytes)
 Data memory     : object 41760 bytes, total data 332416 bytes in 8 objects
 Numeric type    : RealBinaryGMP, prec:100000/100000, mem:41760
 Result          : 0.13333364758144884244...
 Output timings : 0.195427s real, 0.140000s CPU (71.6%)

kpsilon> dumpgc

 Dump the garbage collector status:

 Version           : 8.0.4
 Mode              : Normal
 Divisor           : 3
 Threshold         : 2/5%
 Custom allocators : dec: 50%, bin: 50%
 Maximum heap size : unlimited
 Heap expand       : 0 bytes
 Current heap size : 7901184 bytes
 Incremental mode  : disabled
 Ephemeral objects : enabled

 Memory            : used 273678336, free 4819898368, total 8298164224 (bytes)

 Number of managed allocated memory::Object : 9

   0x7f7502229cc0 -> -3 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f7502229e40 -> 1 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f7502229a80 -> 1324 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f7502229c00 -> 0 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f7502229840 -> 0.1333336476 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f75022299c0 -> 4213231312 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f7502229d80 -> 0 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f7502229b40 -> 1323 (RealBinaryGMP, prec:100000/100000, mem:41760)
   0x7f7502229f00 -> -0.4 (RealBinaryGMP, prec:100000/100000, mem:41760)

 Total memory used by the GC allocators : 375840 bytes
```

  Exemple (dumpgcall, extrait) :

```
kpsilon> dumpgcall

 Dump the garbage collector status:

 Version           : 8.0.4
 Mode              : Normal
 Divisor           : 3
 Threshold         : 2/5%
 Custom allocators : dec: 50%, bin: 50%
 Maximum heap size : unlimited
 Heap expand       : 0 bytes
 Current heap size : 7901184 bytes
 Incremental mode  : disabled
 Ephemeral objects : enabled
```

```
Memory            : used 273678336, free 4817592320, total 8298164224 (bytes)

Number of managed allocated memory::Object : 9

  0x7f7502229cc0 -> -3 (RealBinaryGMP, prec:100000/100000, mem:41760)
  0x7f7502229e40 -> 1 (RealBinaryGMP, prec:100000/100000, mem:41760)
  0x7f7502229a80 -> 1324 (RealBinaryGMP, prec:100000/100000, mem:41760)
  0x7f7502229c00 -> 0 (RealBinaryGMP, prec:100000/100000, mem:41760)
  0x7f7502229840 -> 0.1333336476 (RealBinaryGMP, prec:100000/100000, mem:41760)
  0x7f75022299c0 -> 4213231312 (RealBinaryGMP, prec:100000/100000, mem:41760)
  0x7f7502229d80 -> 0 (RealBinaryGMP, prec:100000/100000, mem:41760)
  0x7f7502229b40 -> 1323 (RealBinaryGMP, prec:100000/100000, mem:41760)
  0x7f7502229f00 -> -0.4 (RealBinaryGMP, prec:100000/100000, mem:41760)

Total memory used by the GC allocators : 375840 bytes


Raw managed memory dump :

  Disappearing (short) links:
  Disappearing long links:
  Finalizers:
  Finalizable object: 0x7f7502229cc0
```

~~~~~~

```
  Finalizable object: 0x7f7502229f00


Last collection stats :

  ***GC Dump collection #4
  Time since GC init: 4393 msecs

  ***Static roots:
  From 0x1f7f000 to 0x1f9da80 (temporary)
```

~~~~~~

```
  From 0x7f74ff1ac000 to 0x7f74ff1acbc0 (temporary)
  GC_root_size: 2761576

  ***Heap sections:
  Total heap size: 7901184 (0 unmapped)
  Section 0 from 0x7f7502252000 to 0x7f7502292000 0/64 blacklisted
```

~~~~~~

```
  Section 8 from 0x7f74e602d000 to 0x7f74e62c7000 0/666 blacklisted

  ***Free blocks:
  Free list 2 (total size 8192):
          0x7f74e6049000 size 8192 not black listed
  Free list 60 (total size 6590464):
          0x7f74e604c000 size 6590464 not black listed
  GC_large_free_bytes: 6598656

  ***Blocks in use:
  kind(0=ptrfree,1=normal,2=unc.),size_in_bytes,#_marks_set,#objs
  2,3936,1,1
```

~~~~~~

```
  2,4704,1,1
  blocks= 304, bytes= 1302528


Regions :
```

```
***Section from 0x7f7502252000 to 0x7f7502292000
        0x7f7502252000        used for blocks of size 0x40 bytes

        0x7f7502291000        used for blocks of size 0x60 bytess

***Section from 0x7f74e6631000 to 0x7f74e6695000
        0x7f74e6631000 Missing header!!((nil))

        0x7f74e604c000        free block of size 0x649000 bytes
```

- **dumpheap[s], dumpstack** et **dumpbin**

  Affiche le contenu du (des) tas (*heap(s)*), de la pile (*stack*) ou de la mémoire du binaire.

  Exemple :

```
kpsilon> dumpheap

 Found 34 heaps but only the main heap(s) will be dumped.

 Using heap 32 from 0x7ff68a1d8000 to 0x7ff68a3a6000 size 1892352 bytes.

 Partial heap 32 dump from 0x7ff68a270c00 to 0x7ff68a271000 size 1024 :

 0x7ff68a270c00 40 0e 27 8a f6 7f 00 00 00 00 00 00 00 00 00 00 @.'.............

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 0x7ff68a270ca0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
 0x7ff68a270cb0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
 Numeric detected @ 0x7ff68a270cc0
 3.1415926536 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:42368)
 0x7ff68a270cc0 f0 1e c5 01 00 00 00 00 00 10 07 00 00 00 00 00 ................
 0x7ff68a270cd0 a0 a2 27 8a f6 7f 00 00 5a 00 00 00 00 00 00 00 ..'.....Z.......
 0x7ff68a270ce0 5a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 Z...............
 0x7ff68a270cf0 00 80 79 0c 00 00 00 00 00 20 9c 1a 01 00 00 00 ..y...... ......
 0x7ff68a270d00 00 a4 f1 70 f6 7f 00 00 c0 a6 f1 70 f6 7f 00 00 ...p.......p....
 0x7ff68a270d10 00 a8 f1 70 f6 7f 00 00 00 00 00 00 00 00 00 00 ...p............
 0x7ff68a270d20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
 0x7ff68a270d30 17 b2 c2 66 00 00 00 00 17 b2 c2 66 00 00 00 00 ...f.......f....
 0x7ff68a270d40 26 b2 c2 66 00 00 00 00 a0 86 01 00 00 00 00 00 &..f............
 0x7ff68a270d50 a0 86 01 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
 0x7ff68a270d60 60 6e f5 70 f6 7f 00 00 80 a4 00 00 00 00 00 00 `n.p............
 0x7ff68a270d70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
 Numeric detected @ 0x7ff68a270d80
 1 (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 0x7ff68a270ff0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................


kpsilon> dumpstack

 Stack top memory address    : 0x7ffcf7002000
 Stack bottom memory address : 0x7ffcf704b000
 Total stack size            : 299008 bytes
 Stack pointer address       : 0x7ffcf70449e0
 Used stack size             : 26144 bytes

 Used stack memory dump from 0x7ffcf70449e0 to 0x7ffcf704b000 size 26144 bytes :

 0x7ffcf70449e0 cd 2c c2 01 00 00 00 00 2a 00 00 00 00 00 00 00 .,......*.......
 0x7ffcf70449f0 54 00 00 00 00 00 00 00 00 54 1c 31 e0 bd 32 29 T........T.1..2)
 0x7ffcf7044a00 b0 2c c2 01 00 00 00 00 2e 00 00 00 00 00 00 00 .,..............
 0x7ffcf7044a10 70 4a 04 f7 fc 7f 00 00 a0 4a 04 f7 fc 7f 00 00 pJ.......J......
 0x7ffcf7044a20 67 5d 04 f7 fc 7f 00 00 e0 49 04 f7 fc 7f 00 00 g].......I......
 0x7ffcf7044a30 bf 69 04 f7 fc 7f 00 00 68 5d 04 f7 fc 7f 00 00 .i......h]......
 0x7ffcf7044a40 c8 5a 04 f7 fc 7f 00 00 67 5d 04 f7 fc 7f 00 00 .Z......g]......
 0x7ffcf7044a50 e0 49 04 f7 fc 7f 00 00 bf 69 04 f7 fc 7f 00 00 .I.......i......

 [aborted]


kpsilon> dumpbin
```

```
Executable binary memory dump from 0x000000400000 to 0x000001d7f000 size 26734592 bytes :

0x000000400000 7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00 .ELF............
0x000000400010 02 00 3e 00 01 00 00 00 30 0d 4e 00 00 00 00 00 ..>.....0.N.....
0x000000400020 40 00 00 00 00 00 00 00 38 ed bf 10 00 00 00 00 @.......8.......
0x000000400030 00 00 00 00 40 00 38 00 0a 00 40 00 28 00 25 00 ....@.8...@.(.%.
0x000000400040 06 00 00 00 05 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0x000000400050 40 00 40 00 00 00 00 00 40 00 40 00 00 00 00 00 @.@.....@.@.....

[aborted]
```

- **`callstack`**

  Affiche la pile d'exécution (*call stack*).

  Exemple :

```
kpsilon> callstack

Call stack:

 1) 0x000000646c16 kpsilon::demos::Demo::process_basic_commands(kpsilon::io::Console&,
↪   std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >
↪   const&, std::__cxx11::basic_string<char, std::char_traits<char>,
↪   std::allocator<char> > const&, bool const&, int&) in file lab/demo_calc.cpp at line
↪   1210
 2) 0x00000064b0e3 kpsilon::demos::Demo::calculator(unsigned long, bool) in file
↪   lab/demo_calc.cpp at line 1916
 3) 0x0000004e4bc3 main in file main.cpp at line 1143
 4) 0x7f9c1cb8d840 __libc_start_main
 5) 0x0000004e0d59 _start
```

- **mem**

  Affiche des informations et l'état de la mémoire.

  Exemple (extrait) **:**

```
kpsilon> mem

 Memory                       : used 226541568, free 4926656512, total 8298164224 bytes
 Mtrim/Minfo/MCinfo           : 226476032/142128/278447 bytes
 Objects                      : 3 Object(s)
 GC enabled                   : yes
 Total allocs memory          : 42240 bytes
 Total GC memory              : 42240 bytes
 Ephemeral objects            : enabled

 Executable binary            : 0x000000400000-0x000001d7f000 (26734592 bytes)

 Stack top memory address     : 0x7ffec3068000
 Stack bottom memory address  : 0x7ffec30b0000
 Total stack size             : 294912 bytes
 Stack pointer address        : 0x7ffec30aa8a0
 Used stack size              : 22368 bytes

 Heaps

  0x000001f82000-0x000001f9e000 (114688 bytes)
  0x000002491000-0x0000024d5000 (278528 bytes)
  0x7f94fe238000-0x7f9507140000 (149979136 bytes)
  0x7f9507147000-0x7f950714f000 (32768 bytes)
  0x7f9507156000-0x7f95071ab000 (348160 bytes)
  0x7f9507283000-0x7f95072a6000 (143360 bytes)
  0x7f95072ef000-0x7f9507304000 (86016 bytes)
  0x7f9507309000-0x7f9507312000 (36864 bytes)
  0x7f9507314000-0x7f950731a000 (24576 bytes)
  0x7f9507320000-0x7f9507324000 (16384 bytes)
```

```
  0x7f95086a5000-0x7f9508704000 (389120 bytes)
  0x7f9508855000-0x7f95089f8000 (1716224 bytes)
  0x7f9517b58000-0x7f9517ba4000 (311296 bytes)
  0x7f9518905000-0x7f9518909000 (16384 bytes)
  0x7f9518b22000-0x7f9518b26000 (16384 bytes)
  0x7f9519428000-0x7f951942b000 (12288 bytes)
  0x7f9519869000-0x7f9519abd000 (2441216 bytes)
  0x7f951b117000-0x7f951b2e5000 (1892352 bytes)
  0x7f951b2e5000-0x7f951b316000 (200704 bytes)
  0x7f951b318000-0x7f951b319000 (4096 bytes)

 GC Heap min                  : 0x7f951b1afc00
 GC Heap max                  : 0x7f951b1aff00

 Default radix                : 10 (decimal)
 Default precision            : 100000
 Shown precision              : 20
 Default binary type          : RealBinaryMPFR
 Default decimal type         : RealDecimalMPDec
```

- **source** (*file*[:*function*] | *function*) [*lines*]

  Affiche le code source d'une fonction ou d'un fichier.

  Exemple :

```
kpsilon> source visitNumber

File: src/interpreter/kpsilon_visitor.cpp

   1888    antlrcpp::Any KpsilonVisitor::visitNumber(KpsilonParser::NumberContext *ctx) {
   1889        if (m_detect_min_max_prec_ && m_detect_prec_mode_) {
   1890
   1891            return 0;
   1892        }
   1893
   1894        numerics::Numeric* num;
   1895        auto               value_str = ctx->getText();
   1896
   1897    #if (defined DO_TRACE) && (DO_TRACE >= 2)
   1898        std::string org_value_str = value_str;
   1899    #endif
   1900
   1901        // If the negate_children status is enabled, the number value should be
 ↪  negated!
   1902        if (this->negate_children) {
   1903            value_str = "-" + value_str;
   1904        }
   1905
   1906        if (configs.numerics.default_radix == 2) {
   1907            num = numerics::Numeric::create(mem::ObjType::RealBinary,
 ↪  this->m_prec_, value_str);
   1908        } else {
   1909            num = numerics::Numeric::create(mem::ObjType::RealDecimal,
 ↪  this->m_prec_, value_str);
   1910        }
   1911
   1912        TRACE2_CALL("<%s> Interpreter visitNumber() (%s) called and returned %s @
 ↪  %s", std::string(m_trace_tree_level, '=').c_str(), org_value_str.c_str(),
 ↪  num->dump(true, 20).c_str(), HEX_ADDR_CSTR(num))
   1913
   1914        m_int_stack_.push_back(num);
   1915
   1916        return 0;
   1917    }
```

- **disasm** (*addr* | *file*[:*function*] | *function*) [*lines*]

  Désassemble le code d'une fonction ou d'une adresse mémoire.

  Exemple (extrait) **:**

```
kpsilon> disasm visitNumber

  kpsilon`kpsilon::KpsilonVisitor::visitNumber at kpsilon_visitor.cpp:1888
     1887
     1888  antlrcpp::Any KpsilonVisitor::visitNumber(KpsilonParser::NumberContext *ctx) {
     1889        if (m_detect_min_max_prec_ && m_detect_prec_mode_) {
  kpsilon`kpsilon::KpsilonVisitor::visitNumber:
  kpsilon[0x59c890] <+0>:     55                                  pushq  %rbp
  kpsilon[0x59c891] <+1>:     48 89 e5                            movq   %rsp, %rbp
  kpsilon[0x59c894] <+4>:     41 57                               pushq  %r15
  kpsilon[0x59c896] <+6>:     41 56                               pushq  %r14
  kpsilon[0x59c898] <+8>:     41 55                               pushq  %r13
  kpsilon[0x59c89a] <+10>:    41 54                               pushq  %r12
  kpsilon[0x59c89c] <+12>:    53                                  pushq  %rbx
  kpsilon[0x59c89d] <+13>:    48 81 ec f8 05 00 00                subq   $0x5f8, %rsp
  ↪           ; imm = 0x5F8
  kpsilon[0x59c8a4] <+20>:    49 89 fc                            movq   %rdi, %r12
  kpsilon[0x59c8a7] <+23>:    48 89 f3                            movq   %rsi, %rbx
  kpsilon[0x59c8aa] <+26>:    48 89 d6                            movq   %rdx, %rsi
  kpsilon`kpsilon::KpsilonVisitor::visitNumber + 29 at kpsilon_visitor.cpp:1889
     1888  antlrcpp::Any KpsilonVisitor::visitNumber(KpsilonParser::NumberContext *ctx) {
     1889        if (m_detect_min_max_prec_ && m_detect_prec_mode_) {
     1890
  kpsilon[0x59c8ad] <+29>:    80 7b 3a 00                         cmpb   $0x0, 0x3a(%rbx)
  kpsilon[0x59c8b1] <+33>:    74 06                               je     0x59c8b9
  ↪           ; <+41> at kpsilon_visitor.cpp:1895
```

```
  kpsilon[0x59f9f5] <+12645>: 74 18                               je     0x59fa0f
  ↪  kpsilon`kpsilon::KpsilonVisitor::visitNumber + 12998 at kpsilon_visitor.cpp:1919
     1918        return 0;
     1919    }
     1920
  kpsilon[0x59fb56] <+12998>: 4c 89 e0                            movq   %r12, %rax
  kpsilon[0x59fb59] <+13001>: 48 81 c4 f8 05 00 00                addq   $0x5f8, %rsp
  ↪           ; imm = 0x5F8
  kpsilon[0x59fb60] <+13008>: 5b                                  popq   %rbx
  kpsilon[0x59fb61] <+13009>: 41 5c                               popq   %r12
  kpsilon[0x59fb63] <+13011>: 41 5d                               popq   %r13
  kpsilon[0x59fb65] <+13013>: 41 5e                               popq   %r14
  kpsilon[0x59fb67] <+13015>: 41 5f                               popq   %r15
  kpsilon[0x59fb69] <+13017>: 5d                                  popq   %rbp
  kpsilon[0x59fb6a] <+13018>: c3                                  retq
  kpsilon[0x59fb6b] <+13019>: 48 89 c3                            movq   %rax, %rbx
  kpsilon[0x59fb6e] <+13022>: e9 ab dd ff ff                      jmp    0x59d91e
  ↪           ; <+4238> [inlined] std::__cxx11::basic_string<char,
  ↪  std::char_traits<char>, std::allocator<char> >::_M_data() const at
  ↪  basic_string.h:226
```

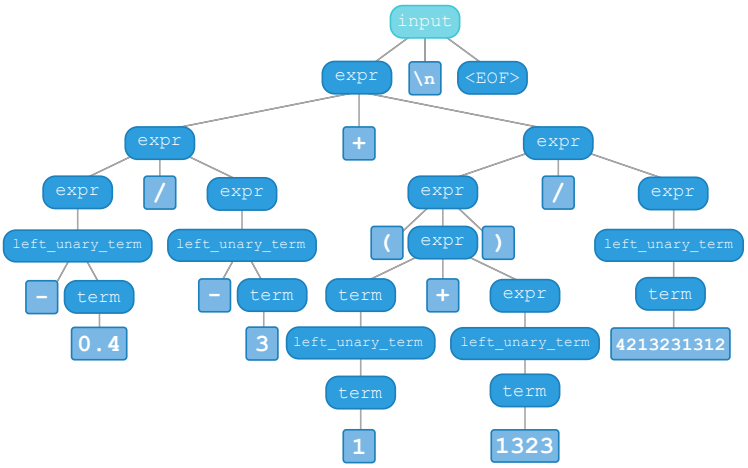
- **showtree**

  Génère l'*arbre syntaxique (CST)* de la dernière expression fournie à l'invite.

  Exemple :

```
kpsilon> -0.4 / -3 + (1 + 1323) / 4213231312

 Timings         : 0.003334s real, 0.000000s CPU (n/a%)
 Memory          : adr 0x7f84c72fa500, used 19922944, free 4770160640 (bytes)
 Data memory     : object 42248 bytes, total data 7498264 bytes in 10 objects
 Numeric type    : RealDecimalMPDec, prec:100000/100000, mem:42248
 Result          : 0.13333364758144884244...
 Output timings : 0.002375s real, 0.010000s CPU (421.1%)

kpsilon> showtree

 Found tokens for "-0.4 / -3 + (1 + 1323) / 4213231312" :

   [@0,0:0='-',<'-'>,1:0]
   [@1,1:3='0.4',<NUM>,1:1]
   [@2,5:5='/',<'/'>,1:5]
   [@3,7:7='-',<'-'>,1:7]
   [@4,8:8='3',<NUM>,1:8]
   [@5,10:10='+',<'+'>,1:10]
   [@6,12:12='(',<'('>,1:12]
   [@7,13:13='1',<NUM>,1:13]
   [@8,15:15='+',<'+'>,1:15]
   [@9,17:20='1323',<NUM>,1:17]
   [@10,21:21=')',<')'>,1:21]
   [@11,23:23='/',<'/'>,1:23]
   [@12,25:34='4213231312',<NUM>,1:25]
   [@13,35:35='\n',<NL>,1:35]
   [@14,36:35='<EOF>',<EOF>,2:0]

 Generating concrete syntax tree (CST) and open in inspector.
```

  Résultat :



- `tests`

Exécute les tests unitaires *Catch2*.

Exemple :

```
kpsilon> tests

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
tests is a Catch v2.13.9 host application.
Run with -? for options

-------------------------------------------------------------------------------
Commutative operations on different types
  4.6d1@1 + 1e-8d2@1 == 4.6d2@1 + 1e-8d1@1 (C++)
-------------------------------------------------------------------------------
/home/user/kpsilon/tests/numeric/tests_real.cpp:255
...............................................................................

/home/user/kpsilon/tests/numeric/tests_real.cpp:265: FAILED:
  REQUIRE( res1.to_string() == res2.to_string() )
with expansion:
  "4.6" == "5.00000001"

-------------------------------------------------------------------------------
Commutative operations on different types
  4.6d2@1 + 1e-8b1@1 == 4.6b1@1 + 1e-8d2@1 (C++)
-------------------------------------------------------------------------------
/home/user/kpsilon/tests/numeric/tests_real.cpp:288
...............................................................................

/home/user/kpsilon/tests/numeric/tests_real.cpp:298: FAILED:
  REQUIRE( res1.to_string() == res2.to_string() )
with expansion:
  "5" == "4.6"

===============================================================================
test cases: 105 | 104 passed | 1 failed
assertions: 555 | 553 passed | 2 failed
```

Remarque: *Pour l'instant, il est normal que deux tests échouent, car ils ont été mis en place pour régler un bogue sur lequel nous travaillons encore.*

- **traceon, traceoff** et **trace{1-5}**

  Active, désactive ou sélectionne le niveau (1 à 5) de TRACE.

  Important: *Pour utiliser la fonctionnalité TRACE, elle doit être activée au moment de la compilation du logiciel.*

  Exemple (trace2) :

```
kpsilon> trace2

 TRACE level set to 2.

kpsilon> 1/3


TRACE_CALL => Interpreter visitInput() called, input = 1/3<EOF>
     |===> kpsilon::demos::Demo::calculator(unsigned long, bool) at demo_calc.cpp:2053
 ↪   -> virtual antlrcpp::Any
 ↪   kpsilon::KpsilonVisitor::visitInput(kpsilon::KpsilonParser::InputContext*) at
 ↪   kpsilon_visitor.cpp:118

TRACE_CALL ==> Interpreter visitMultOrDiv() (/) called
     |===>
 ↪   kpsilon::KpsilonParser::MultOrDivContext::accept(antlr4::tree::ParseTreeVisitor*)
 ↪   at KpsilonParser.cpp:1717 -> virtual antlrcpp::Any
 ↪   kpsilon::KpsilonVisitor::visitMultOrDiv(kpsilon::KpsilonParser::MultOrDivContext*)
 ↪   at kpsilon_visitor.cpp:1726

TRACE_CALL <===> Interpreter visitNumber() (1) called and returned 1
     (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256) @ 0x7f0fc0c39e40
     |===>
 ↪   kpsilon::KpsilonParser::NumberContext::accept(antlr4::tree::ParseTreeVisitor*) at
 ↪   KpsilonParser.cpp:2465 -> virtual antlrcpp::Any
 ↪   kpsilon::KpsilonVisitor::visitNumber(kpsilon::KpsilonParser::NumberContext*) at
 ↪   kpsilon_visitor.cpp:1912

TRACE_CALL <===> Interpreter visitNumber() (3) called and returned 3
     (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:256) @ 0x7f0fc0c39b40
     |===>
 ↪   kpsilon::KpsilonParser::NumberContext::accept(antlr4::tree::ParseTreeVisitor*) at
 ↪   KpsilonParser.cpp:2465 -> virtual antlrcpp::Any
 ↪   kpsilon::KpsilonVisitor::visitNumber(kpsilon::KpsilonParser::NumberContext*) at
 ↪   kpsilon_visitor.cpp:1912

TRACE <== Interpreter visitMultOrDiv() (1 / 3) returned 0.33333333333333333333
     (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:21312, ephm) @
     0x7f0fc0c39a80
     |======> virtual antlrcpp::Any
 ↪   kpsilon::KpsilonVisitor::visitMultOrDiv(kpsilon::KpsilonParser::MultOrDivContext*)
 ↪   at kpsilon_visitor.cpp:1768

TRACE <= Interpreter visitInput() returned 0.33333333333333333333
     (kpsilon::numeric::RealDecimalMPDec, prec:100000/100000, mem:21312, ephm) @
     0x7f0fc0c39a80
     |======> virtual antlrcpp::Any
 ↪   kpsilon::KpsilonVisitor::visitInput(kpsilon::KpsilonParser::InputContext*) at
 ↪   kpsilon_visitor.cpp:180
 Timings        : 0.256367s real, 0.100000s CPU (39.0%)

TRACE_CALL Interpreter get_result() called, result = 0.3333333333... @ 0x7f0fc0c39a80
     |===> kpsilon::demos::Demo::calculator(unsigned long, bool) at demo_calc.cpp:2337
 ↪   -> kpsilon::numeric::Numeric& kpsilon::KpsilonVisitor::get_result() at
 ↪   kpsilon_visitor.cpp:93
 Memory         : adr 0x7f0fc0c39a80, used 226471936, free 4742451200 (bytes)
```

```
 Data memory     : object 21312 bytes, total data 84224 bytes in 7 objects
 Numeric type    : RealDecimalMPDec, prec:100000/100000, mem:21312
 Result          : 0.33333333333333333333...
 Output timings : 0.119749s real, 0.090000s CPU (75.2%)
```

Exemple (`trace5`, extrait) :

```
kpsilon> trace5

 TRACE level set to 5.

kpsilon> 0.5


TRACE_CALL => Interpreter visitInput() called, input = 0.5<EOF>
     |===> kpsilon::demos::Demo::calculator(unsigned long, bool) at demo_calc.cpp:2053
 ↪   -> virtual antlrcpp::Any
 ↪   kpsilon::KpsilonVisitor::visitInput(kpsilon::KpsilonParser::InputContext*) at
 ↪   kpsilon_visitor.cpp:118

TRACE ThreadsDataAccessor set called with id 0x000000000000 and real thread id
      0x895a29211928c3b5 (val:5 @ 0x7efe3ec91ec0)
      |======> void kpsilon::memory::ThreadsDataAccessor<T>::set(const thread_id&, T*)
 ↪   [with T = long unsigned int; kpsilon::memory::ThreadsDataAccessor<T>::thread_id =
 ↪   long unsigned int] at threads_data_accessor.ipp:294


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

TRACE ThreadsDataAccessor set called with id 0x000000000000 and real thread id
      0x895a29211928c3b5 (val:1 @ 0x7efe3ec91ec0)
      |======> void kpsilon::memory::ThreadsDataAccessor<T>::set(const thread_id&, T*)
 ↪   [with T = long unsigned int; kpsilon::memory::ThreadsDataAccessor<T>::thread_id =
 ↪   long unsigned int] at threads_data_accessor.ipp:294

TRACE_CALL boost::multiprecision::gmp_float alloc 41552 bytes (unmanaged by GC @
      0x00000264d040 tid:0x895a29211928c3b5)
      |===> __gmpf_init2 at :__gmpf_init2 in /usr/lib/x86_64-linux-gnu/libgmp.so.10 ->
 ↪   static void* kpsilon::numeric::RealBinaryGMP::custom_gmp_allocate_func(size_t) at
 ↪   real_binary_gmp_allocators.cpp:112


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


TRACE_CALL boost::multiprecision::gmp_float alloc 22 bytes (unmanaged by GC @
      0x000002694620 tid:0x895a29211928c3b5)
      |===> __gmpf_get_str at :__gmpf_get_str in /usr/lib/x86_64-linux-gnu/libgmp.so.10
 ↪   -> static void* kpsilon::numeric::RealBinaryGMP::custom_gmp_allocate_func(size_t)
 ↪   at real_binary_gmp_allocators.cpp:112


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 Timings         : 0.800075s real, 0.360000s CPU (45.0%)

TRACE_CALL Interpreter get_result() called, result = 0.5 @ 0x7efe3ec6de40
     |===> kpsilon::demos::Demo::calculator(unsigned long, bool) at demo_calc.cpp:2337
 ↪   -> kpsilon::numeric::Numeric& kpsilon::KpsilonVisitor::get_result() at
 ↪   kpsilon_visitor.cpp:93
 Memory          : adr 0x7efe3ec6de40, used 241504256, free 4567441408 (bytes)
 Data memory     : object 41760 bytes, total data 41552 bytes in 5 objects
 Numeric type    :
```

```
ThreadsDataAccessor set called with id 0x000000000000 and real thread id
      0x895a29211928c3b5 (val:110 @ 0x7efe3ec91ec0)
      |======> void kpsilon::memory::ThreadsDataAccessor<T>::set(const thread_id&, T*)
  ↪   [with T = long unsigned int; kpsilon::memory::ThreadsDataAccessor<T>::thread_id =
  ↪   long unsigned int] at threads_data_accessor.ipp:294

TRACE_CALL boost::multiprecision::gmp_float alloc 100002 bytes (unmanaged by GC @
      0x000002694620 tid:0x895a29211928c3b5)
      |===> __gmpf_get_str at :__gmpf_get_str in /usr/lib/x86_64-linux-gnu/libgmp.so.10
  ↪   -> static void* kpsilon::numeric::RealBinaryGMP::custom_gmp_allocate_func(size_t)
  ↪   at real_binary_gmp_allocators.cpp:112


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

RealBinaryGMP, prec:100000/100000, mem:41760

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

TRACE_CALL boost::multiprecision::gmp_float dealloc 100147 bytes (unmanaged by GC @
      0x0000026accd0 tid:0x895a29211928c3b5)
      |===> __gmp_tmp_reentrant_free at :__gmp_tmp_reentrant_free in
  ↪   /usr/lib/x86_64-linux-gnu/libgmp.so.10 -> static void
  ↪   kpsilon::numeric::RealBinaryGMP::custom_gmp_free_func(void*, size_t) at
  ↪   real_binary_gmp_allocators.cpp:241

 Result         :

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

TRACE_CALL boost::multiprecision::gmp_float dealloc 2 bytes (unmanaged by GC @
      0x000002694620 tid:0x895a29211928c3b5)
      |===> std::ostream& boost::multiprecision::operator<<
  ↪   <boost::multiprecision::backends::gmp_float<0u>,
  ↪   (boost::multiprecision::expression_template_option)1>(std::ostream&,
  ↪   boost::multiprecision::number<boost::multiprecision::backends::gmp_float<0u>,
  ↪   (boost::multiprecision::expression_template_option)1> const&) at
  ↪   :expression_template_option)1> const&) in
  ↪   /home/user/kpsilon/build/debug_g++-9_trace5_gcalloc_uncollect_gcthreads_v2/kpsilon
  ↪   -> static void kpsilon::numeric::RealBinaryGMP::custom_gmp_free_func(void*, size_t)
  ↪   at real_binary_gmp_allocators.cpp:241
0.5


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 Output timings : 11.036923s real, 4.450000s CPU (40.3%)
```

- **cls**

  Efface l'écran.

- **quit**

  Quitte l'interpréteur.

ANNEXE B
FONCTIONNALITÉS DE TESTS DU LOGICIEL *KPSILON*

CONTENU

B.1 Test de création d'objets inaccessibles « *3) Garbage collector tests* »

Le test de création d'objets inaccessibles analyse la consommation mémoire du logiciel et les temps d'exécutions. Il permet de vérifier le bon fonctionnement du logiciel et du *GC*, et ce, en fonction de différentes versions du logiciel et configurations du *GC*. Il a été utilisé pour générer les principaux graphes contenus dans ce mémoire (chapitres VI et VII). Le listing B.1 contient une description simplifiée[95] du fonctionnement de ce test, sous forme de pseudocode `C++`. Ce test crée dynamiquement des objets *inaccessibles* — auxquels ne réfère aucun pointeur. Puisque ces objets sont inaccessibles, seul le *GC* est en mesure de les détruire s'il fonctionne correctement. Le test génère aussi un graphe de consommation mémoire et de mesure de temps d'exécution.

Dans le graphe de consommation mémoire généré par ce test, les deux zones avec un arrière-plan hachuré représentent deux périodes de créations d'objets `Numeric` accessibles (un pointeur vers un tel objet existe en tout temps). Après la première période de création d'objets accessibles, une longue période de création d'objets `Numeric` inaccessibles (axe horizontal, c.-à-d. de 1 à 10 000 objets dans les figures B.1 et B.2) est effectuée. La consommation mémoire des objets non détruits par le *GC* peut être observée sur l'axe vertical. Finalement, pour interpréter la légende des graphes générés, se référer au tableau D.1 à l'annexe D

La capture B.1 présent un exemple d'une exécution de ce test et les graphes qu'il génère sont présentés aux figures B.1, B.2 et B.3.

[95] Ce code compte environ 700 lignes, sans les fonctions de lecture mémoire, d'affichage à l'écran et de génération des graphes.

```
Numeric* pObj;

auto temps_depart = temps_courant();

// Première phase de génération d'objets accessibles
creer_objets_accessibles(NOMBRE_OBJS_ACCESSIBLES);

// Phase de génération d'objets inaccessibles
for (int no = 0; no < NOMBRE_OBJS_INACCESSIBLES; no++) {
    /* Le mêeme pointeur pObj est utilisé, donc il ne
       pointe plus vers les objets Numeric précédents;
       ces derniers sont donc inaccessibles! */

    pObj = Numeric::create(TYPE, PREC, VAL);

    analyser_memoire_libre();
}

// Deuxième phase de génération d'objets accessibles
creer_objets_accessibles(NOMBRE_OBJS_ACCESSIBLES);

auto temps_total = temps_courant() - temps_depart;
```

Listing B.1 Pseudocode C++ du test du test de création d'objets inaccessibles.

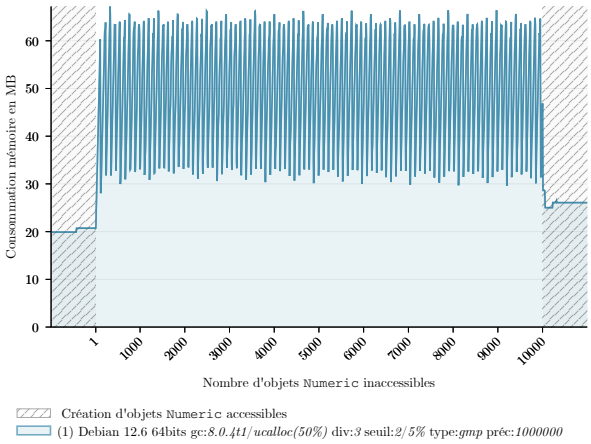


Figure B.1 Graphe généré par le test de création d'objets inaccessibles dans *Linux* (*détails*).

```
-- Extensive garbage collector testing of the RealBinaryGMP Numeric type --


RECORDING TIMINGS & MEMORY GRAPHS

Press [ENTER]  to accept default values
      [CTRL-D] to get more details about the active question
      [CTRL-C] to exit
      [ESC]    during the test to abort

  GC test mode                                          [up/down] : GC
  Total unreachable Numeric objects to create [100-1000000000] : 10000
  Force garbage collector reset                           [Y/N] : N
  Monitor garbage collection events                       [Y/N] : Y
  Memory levels based on total system memory              [Y/N] : N
  Levels output rate modulo exponent (10^M)             M=[0-1] : 1
  Number of pre/post record collects data padding    [0-10000] : 100
  Record collect events                                   [Y/N] : Y
  Generate timings plots                                  [Y/N] : Y
  Level memory limit (bytes, 0 = no limit)         [0-sys_mem] : 50000000


  Generating 100 pre-test record collects data padding...done.
[                                                  ==========================================+]
 used:21729280 (alloc:15910784, del:13701056), free:32636055552

  Press [ESC] to abort!

  Creating 10000 RealBinaryGMP objects with a precision of 1000000 digits...
  Creating unreachable Numerics...
[                                                  ==================================++]
 used:53432320 (alloc:5284320, del:6091936), free:32614178816, objs:10000
  done.
 1689.961169s wall, 1308.590000s user + 52.440000s system = 1361.030000s CPU (80.5%)


  Generating 100 post-test record collects data padding...done.
[                                                  ==================================+]
 used:27459584 (alloc:91920, del:74960), free:32642678784


 Consumed memory in MB
    +       + +  ++  + +++ +  ++  + ++  + +++ +  ++  + +++ +  ++  + ++
 60 |      +++:+++++++++:++:++:++++++++++::++:+++++++++:++:++:++++++:
    |     +:::+:+:::::::+:::::::+:::::::::::::::+:::::::+::::::+::::::
 50 |     ::::::::::::::+:::::::::::::::+:::::::+:::::::+:+::::::+::::::+:
    |     :::+:::::::+:::::::::+::::+:+:::::::+:::::::+:::::::+::::::+::+
 40 |     :+::::::+:::::::+:+:::::::::::::::+:::::::+:::::::+::::::::::::
    |     ::::::+:::::::+:::::::::+::::::+:::::::::::::::+:::::::+:::::
 30 |     ::::+:::::::::::::::+:::::::::::::::+:::::::::::::::+::::::+
    |     :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::+++++++
 20 |+++++:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
    |:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
 10 |:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
    |:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
  0 +---------------------------------------------------------------------+
          1   1000  2000  3000  4000 5000  6000  7000  8000  9000  10000
                    Number of unreachable Numeric objects


  Saving ../../output/plots/test_manual_gc_test/test_gc plot...done.
  Saving timings pie chart and plots...done.
  Saving compound plot(s)...done.
```

Capture B.1 Test de création d'objets inaccessibles.

Il est possible d'exécuter ce test plus d'une fois, avec des configurations différentes ou sur un autre système, pour ajouter les nouveaux résultats à la figure B.1. Une exemple d'une deuxième passe du test, utilisant le type `RealDecimalMPDec` au lieu du type `RealBinaryGMP` utilisé lors de la première passe, est présenté à la figure B.2.

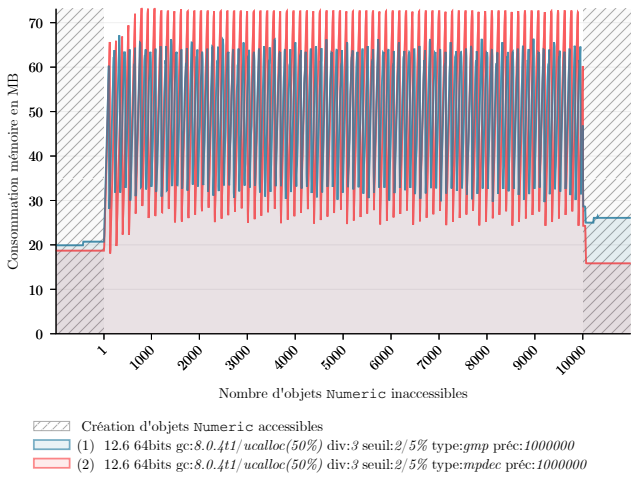


Figure B.2 Résultat d'une deuxième passe du test dans *Linux* (*détails*).

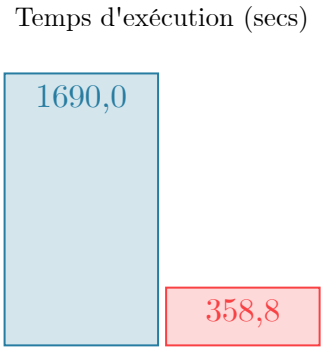


Figure B.3 Temps en secondes de la deuxième passe du test (*détails*).

En observant les temps dans la figure B.3, on constate que le type `RealBinaryGMP` est passablement plus lent que le type `RealDecimalMPDec`. Ceci s'explique par le fait que, comme mentionné au tableau 4.4 du chapitre IV, la création d'un `gmp_float` est lente — et ce test ne fait que des créations d'objets et non des calculs.

B.2 Test des différents types `Numeric` et bibliothèques multiprécision « *4) Radix 2 vs 10 tests* »

Ce test permet de comparer la consommation mémoire, les performances et la précision des résultats des différents types `Numeric` et des bibliothèques multiprécision. Il a été utilisé pour générer une grande partie des graphes du chapitre VII.

La capture B.2 présent un exemple d'une exécution de ce test et les graphes qu'il génère sont présentés aux figures B.4 et B.5.

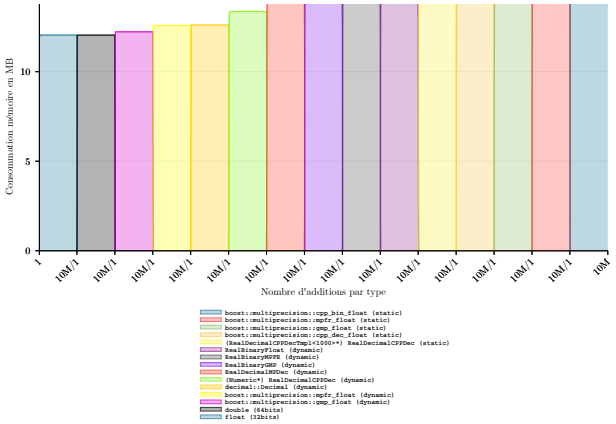


Figure B.4 Consommation mémoire (*details*).

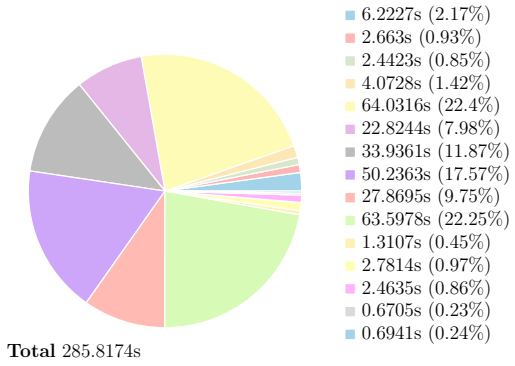


Figure B.5 Temps d'exécutions (*details*).

Remarque: *Dans le mémoire, les graphes en pointes de tarte (pie chart) générés par ce test (cf. figure B.5) ont été convertis manuellement en graphe en bâtons (bar chart) — ce dernier type de graphe est plus approprié pour comparer des temps d'exécution.*

```
-- Radix 2 vs 10 tests --

Comparing IEEE 754, RealBinary, RealDecimal and boost::multiprecision types.


RECORDING TIMINGS & MEMORY GRAPHS

Press [ENTER]  to accept default values
      [CTRL-C] to exit
      [ESC]    during the test to abort or [s] to skip

  Format Numeric outputs                            [Y/N] : N
  Output memory statistics                          [Y/N] : N
  Enter number of operations to perform  [1000-1000000000] : 10000000
  Arithmetic operation                          [up/down] : ADD
  Value to use                                     [expr] : 0.01

Expected final result = 100000

Using value 0.01 in 10000000 additions.

Calculations will be using a precision of 1000 decimals.


Press [s] to skip or [ESC] to abort!


Computing float (32bits)...

  Relative timing vs float : 1 (0.694129s/0.694129s)
  Relative timing vs gmp   : n/a
  Execution time           : 0.694857s real, 0.250000s CPU (36.0%)

  Result float (32bits) : 95680.9453


Computing double (64bits)...

  Relative timing vs float : 0.966021 (0.670544s/0.694129s)
  Relative timing vs gmp   : n/a
  Execution time           : 0.678741s real, 0.240000s CPU (35.4%)

  Result double (64bits) : 99999.997764825821


Computing boost::multiprecision::gmp_float (dynamic)...

  Relative timing vs float : 3.54917 (2.46358s/0.694129s)
  Relative timing vs gmp   : 1 (2.46358s/2.46358s)
  Execution time           : 2.464533s real, 2.060000s CPU (83.6%)

  Result boost::multiprecision::gmp_float (dynamic) :
   ↪ 99999.999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
   ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
   ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
   ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
   ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
   ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
   ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
   ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
   ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
   ↪ 99999999999999999999999999999999999999999999999999999999999999999999999999999999999168


Computing boost::multiprecision::mpfr_float (dynamic)...
```

```

  Relative timing vs float : 4.00711 (2.78145s/0.694129s)
  Relative timing vs gmp   : 1.12903 (2.78145s/2.46358s)
  Execution time           : 2.782531s real, 2.360000s CPU (84.8%)

  Result boost::multiprecision::mpfr_float (dynamic) :
  ↪ 99999.999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 9999999999999999999999999999999999999999999999999999999999999999999999999999989723523


Computing decimal::Decimal (dynamic)...

  Relative timing vs float : 1.88838 (1.31078s/0.694129s)
  Relative timing vs gmp   : 0.532063 (1.31078s/2.46358s)
  Execution time           : 1.311491s real, 0.890000s CPU (67.9%)

  Result decimal::Decimal (dynamic) : 100000


Computing (Numeric*) RealDecimalCPPDec (dynamic)...

  Relative timing vs float : 91.6225 (63.5979s/0.694129s)
  Relative timing vs gmp   : 25.8152 (63.5979s/2.46358s)
  Execution time           : 63.599020s real, 63.090000s CPU (99.2%)

  Result (Numeric*) RealDecimalCPPDec (dynamic) : 100000


Computing RealDecimalMPDec (dynamic)...

  Relative timing vs float : 40.1504 (27.8696s/0.694129s)
  Relative timing vs gmp   : 11.3126 (27.8696s/2.46358s)
  Execution time           : 27.870290s real, 27.400000s CPU (98.3%)

  Result RealDecimalMPDec (dynamic) : 100000


Computing RealBinaryGMP (dynamic)...

  Relative timing vs float : 72.3732 (50.2363s/0.694129s)
  Relative timing vs gmp   : 20.3916 (50.2363s/2.46358s)
  Execution time           : 50.237053s real, 49.740000s CPU (99.0%)

  Result RealBinaryGMP (dynamic) :
  ↪ 99999.999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
  ↪ 99999999999999999999999999999999999999999999999999999999999999999999999999999999999


Computing RealBinaryMPFR (dynamic)...

  Relative timing vs float : 48.8902 (33.9361s/0.694129s)
```

```
  Relative timing vs gmp   : 13.7751 (33.9361s/2.46358s)
  Execution time           : 33.936860s real, 33.420000s CPU (98.5%)

  Result RealBinaryMPFR (dynamic) :
↪  99999.999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  99999999999999999999999999999999999999999999999999999999999999999999999999999897235


Computing RealBinaryFloat (dynamic)...

  Relative timing vs float : 32.8821 (22.8244s/0.694129s)
  Relative timing vs gmp   : 9.26474 (22.8244s/2.46358s)
  Execution time           : 22.825140s real, 22.290000s CPU (97.7%)

  Result RealBinaryFloat (dynamic) : 99999.99998630968912039


Computing (RealDecimalCPPDecTmpl<1000>*) RealDecimalCPPDec (static)...

  Relative timing vs float : 92.2473 (64.0316s/0.694129s)
  Relative timing vs gmp   : 25.9913 (64.0316s/2.46358s)
  Execution time           : 64.032738s real, 63.530000s CPU (99.2%)

  Result (RealDecimalCPPDecTmpl<1000>*) RealDecimalCPPDec (static) : 100000


Computing boost::multiprecision::cpp_dec_float (static)...

  Relative timing vs float : 5.86761 (4.07288s/0.694129s)
  Relative timing vs gmp   : 1.65324 (4.07288s/2.46358s)
  Execution time           : 4.074275s real, 3.650000s CPU (89.6%)

  Result boost::multiprecision::cpp_dec_float (static) : 100000


Computing boost::multiprecision::gmp_float (static)...

  Relative timing vs float : 3.51858 (2.44235s/0.694129s)
  Relative timing vs gmp   : 0.991381 (2.44235s/2.46358s)
  Execution time           : 2.444906s real, 2.020000s CPU (82.6%)

  Result boost::multiprecision::gmp_float (static) :
↪  99999.999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999⌋
↪  99999999999999999999999999999999999999999999999999999999999999999999999999999999999168


Computing boost::multiprecision::mpfr_float (static)...

  Relative timing vs float : 3.83657 (2.66308s/0.694129s)
  Relative timing vs gmp   : 1.08098 (2.66308s/2.46358s)
  Execution time           : 2.664729s real, 2.240000s CPU (84.1%)
```

```

  Result boost::multiprecision::mpfr_float (static) :
    99999.999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    9999999999999999999999999999999999999999999999999999999999999999999999999999989723523


Computing boost::multiprecision::cpp_bin_float (static)...

  Relative timing vs float : 8.96483 (6.22275s/0.694129s)
  Relative timing vs gmp   : 2.5259 (6.22275s/2.46358s)
  Execution time           : 6.225735s real, 5.800000s CPU (93.2%)

  Result boost::multiprecision::cpp_bin_float (static) :
    99999.999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
    9999999999999999999999999999999999999999999999999999999999999999999999999999989723523
```

Capture B.2 Test des différentes bibliothèques et types `Numeric`.

ANNEXE C

PRÉPROCESSEUR C++ `PREPROCESSOR.PY`

CONTENU

C.1 Introduction

Le préprocesseur « *maison* » a été créé afin de permettre d'utiliser des boucles pour générer le code `C++` des *templates* `RealDecimalCPPDec<PREC>` des différentes précisions relatives. Bien qu'il soit possible de faire des boucles avec le préprocesseur `C++`, puisque ce dernier est *Turing-complet*, la syntaxe nécessaire pour introduire des boucles et le code généré sont complexes, cryptiques et difficiles à déboguer. Il est aussi possible d'utiliser des bibliothèques externes[96] pour simplifier cette tâche, mais encore une fois, le code généré est compliqué à déboguer. Un préprocesseur « *maison* » a donc été implémenté en *Python* et a finalement été utilisé dans plusieurs autres situations.

Le script *Python* contenant le code du préprocesseur se nomme `preprocessor.py`. Ce script est exécuté, au besoin, par *CMake* (fichier `CMakeLists.txt`) avant la compilation. Dans les faits, une fois que *CMake* a généré les « *makefiles* », c'est `make` (ou `msbuild` dans *Windows*) qui exécute le préprocesseur, si une modification est détectée sur un des fichiers `.template.?pp` ou sur tout autre fichier `.hpp` en mode détection étendue (voir plus loin).

Le code source du préprocesseur est complexe, car ce dernier fonctionne de manière itérative en appliquant plusieurs passes de « *préprocessing* » pour simplifier son implémentation. S'il devait être recodé en partant de zéro (il le sera probablement, car il contient quelques bogues difficiles à corriger), *ANTLR* serait utilisé pour *parser* les fichiers *template*, car nous avons réalisé qu'il permet de générer du code *Python*. De plus, le

[96] Par exemple, la bibliothèque *Boost.Preprocessor* a été testée en début de projet. Elle supporte les boucles au niveau du préprocesseur `C++`, mais le code généré est quasi impossible à comprendre et déboguer — il est composé de nombreuses classes *templates* ayant des noms cryptiques.

préprocesseur peut être un peu capricieux et son fonctionnement peut sembler étrange, car à la base il n'était conçu que pour supporter des boucles, donc tout est orienté boucles !

Le préprocesseur permet d'utiliser à l'intérieur de ses commandes des valeurs de `#define` se trouvant ailleurs dans le code `C++` — y compris dans un fichier externe `.hpp`. De plus, il est possible de lui passer en paramètres des valeurs de `#define` pour modifier son comportement de l'extérieur (voir l'option `-defines=`).

C.2 Modes de détection des modifications aux fichiers `.template.?pp`

Il est possible d'utiliser différents modes de détection de modification(s) de fichier(s) avant de lancer le préprocesseur en passant un *flag* à *CMake* ou une option à la commande « `compile` » :

- **Détection désactivée**

  Le préprocesseur ne sera jamais exécuté.

  Flag CMake **:** `-DPREPROCESS_SOURCES=OFF`
  Option « compile » : `--nopp`

- **Détection de base** (mode par défaut)

  Dans ce mode, seuls les fichiers `.template.?pp` dont la date de modification a été altérée (il est aussi possible d'utiliser un mode de calcul de la somme *MD5* du *template* ; ce mode est désactivé par défaut) sont précompilés. Ce mode de détection est le plus rapide, car les `#define` `C++` contenus dans les fichiers *headers* du projet ne sont pas vérifiés. C'est-à-dire que le préprocesseur n'aura pas à *parser* le projet en entier pour les détecter. Par contre, si une valeur de `#define` est utilisée par une commande du préprocesseur, elle ne sera pas détectée si modifiée et le *template* ne sera pas re-précompilé. Il est donc nécessaire d'utiliser le mode détection étendue pour mettre à jour le fichier `.preprocessed.?pp` en fonction de cette nouvelle valeur.

  Flag CMake **:** `-DPREPROCESS_SOURCES=AUTO` (optionel)
  Option « compile » : `n/a`

- **Détection étendue**

Ce mode détecte si une ou des valeurs de `#define` C++ utilisées dans les commandes du préprocesseur ont été modifiées dans le projet (fichiers *headers*). Ce mode est lent, car le préprocesseur doit *parser* en entier le code C++ du projet. Dans ce mode, tous les fichiers *templates* sont vérifiés en tout temps.

Flag CMake : `-DPREPROCESS_SOURCES=ON`

Option « compile » : `--fullppdetect`

Une précompilation complète du projet peut être lancée avec la commande « `compile preprocess` ».

Remarque: *Une précompilation complète est effectuée après un réinitialisation du projet (« `compile reset` »).*

C.3 Utilisation du préprocesseur

```
preprocessor.py [-hsvVML] [--source_path=path] [--build_dir=path]
                          [--defines="define1=val[;define2=val...]"
                          file1 [file2 ...]
```

Options :

- **`-h`** (*help*)

  Affiche la liste des options.

- **`-s`** (*silent*)

  Exécute le préprocesseur en mode silencieux.

- **`-v`** (*verbose*)

  Affiche des informations en lien avec la précompilation, incluant les *TAGs* détectés, les substitutions de `#define` (*`Macros expansion:`*) et les commentaires de précompilation (*`Comment:`*).

  À titre d'exemple, voici les informations affichées lors du « *parsing* » du code *« Utilisation des `__INDEX?__` dans des boucles imbriquées »* :

```
Preprocessing file exemple1.template.hpp...
- PP_EXPR(ARRAY_SIZE) tag detected at line 3
- PP_EVAL(PP_COUNT__ = 0) tag detected at line 5
```

```
- PP_REPEAT_BEGIN(PP_IND__, range(0,ARRAY_SIZE)) tag detected at line 6
Macros expansion: PP_REPEAT_BEGIN(PP_IND__, range(0,ARRAY_SIZE)) -> PP_REPEAT_BEGIN(PP_IND__,
range(0,4))
- PP_REPEAT_BEGIN(PP_VAL__, range(10,25,3)) tag detected at line 7
- PP_IF(PP_VAL__ % 2 == 0) tag detected at line 8
Comment: Is the value even?
- PP_EXPR(__INDEX1__ * PP_VAL__ + PP_COUNT__) tag detected at line 9
- PP_EXPR(__INDEX1__ * PP_VAL__ + PP_COUNT__) tag detected at line 10
PP_ENDIF tag detected at line 11
- PP_EVAL(PP_COUNT__ += 1) tag detected at line 12
PP_REPEAT_END tag detected at line 13
PP_REPEAT_END tag detected at line 14
NOTE: Additional tags were detected in pass 3 but were not shown in the terminal.
NOTE: Additional tags were detected in pass 4 but were not shown in the terminal.
NOTE: Additional tags were detected in pass 5 but were not shown in the terminal.
done (exemple1.preprocessed.hpp).
```

Remarque: *Il n'affiche que l'information du dernier TAG détecté lorsqu'une ligne en contient plus d'un ; il spécifie néanmoins que d'autres TAGs ont été détectés.*

Cette option est utilisée par défaut dans le fichier `CMakeLists.txt`

- **`-V`** (*more verbose*)

  Affiche des informations supplémentaires de débogage, incluant les différentes phases de précompilation.

- **`-M`** (*monitor changes*)

  Utilise une base de données locale pour détecter si un fichier *template* (ou un `#define` `C++` qu'il utilise) est modifié. La précompilation n'est pas exécutée si aucune modification n'est détectée. Il est possible d'utiliser deux modes de détection :

  - Changement de la date du fichier *template* et modfifications des `#define`
  - Calcul de la somme *MD5* du fichier *template* et modifications des `#define`.

  Par défaut, c'est la date de modification qui est utilisée (plus rapide).

- **`-L`** (*use lock file*)

  Un fichier de verrouillage (*lock file*) est utilisé par *CMake* pour éviter de relancer le préprocesseur à toutes les étapes de la compilation. Cette option n'est normalement pas utilisée si le préprocesseur est

utilisé directement en ligne de commande, seul *CMake* l'utilise.

- **`--source_path=`***`path`*

  Permet de spécifier le répertoire des fichiers sources du projet.

- **`--build_dir=`***`path`*

  Permet de spécifier le répertoire de compilation (*build directory*).

- **`--defines="`***`define1=val[;define2=val...]`***`"`**

  Permet de définir de l'extérieur des `#define` `C++`. Si ces `#define` sont présents dans le code `C++`, ils seront écrasés. Le code *CMake* utilise cette option pour configurer certaines options du projet (dont la granularité des valeurs possibles des précisions dans les *templates* des `RealDecimalDecCPP<PREC>`).

C.4 Syntaxe du langage du précompilateur

- **Introduction**

  Les commandes du préprocesseur (*TAGs*) sont de la forme « **`/*=== TAG[(params)] [comment] ===*/`** » ou « **`{*=== TAG[(params)] [comment] ===*}`** ». Le préprocesseur supporte ces deux syntaxes pour permettre d'ajouter des *TAGs* dans les commentaires `C++` de type « **`/*...*/`** » sans que l'*IDE* affiche une erreur.

  Les paramètres des *TAGs* du préprocesseur sont en réalité du code *Python* et peuvent donc être générés par n'importe quel code *Python* valide. Ils peuvent contenir des `#define` `C++` définis ailleurs dans le code ou passés en arguments (voir option *`--defines=`*) pour changer le comportement du préprocesseur de l'extérieur — exemple, à partir de *CMake*.

  Il est possible d'ajouter un commentaire à chacun des *TAGs*, le préprocesseur va afficher ce commentaire s'il est lancé avec les options « **`-v`** » ou « **`-V`** ».

  Il n'est pas obligatoire d'utiliser `PP_EXPR` pour convertir les `#define` générés par les *TAGs* du préprocesseur, ils sont automatiquement convertis en valeur par le préprocesseur comme les `#define` `C++`. Il est donc possible de les utiliser directement dans le code `C++` contenu dans les blocs des *TAGs* du préprocesseur. Dans certains cas, c'est même la meilleure option (voir exemple

« *PP_REPEAT using a list (version 2)* » dans la section sur les boucles).

- **Suppression de ligne (`PP_DISCARD`)**

  Utilisation :

  ```
  /*=== PP_DISCARD [comment] ===*/
  ```

  Ce *TAG* est pratique pour retirer une ligne de code qui se trouve dans le fichier `.template.?pp`, mais qu'on veut retirer du fichier `.preprocessed.?pp`. Il est utilisé à plusieurs endroits dans le code du projet pour retirer des commentaires.

  Fichier d'entrée (*template*) :

  ```
  int val = 3;
  // Cette ligne sera retirée! /*=== PP_DISCARD ===*/
  std::cout << val;
  ```

  Fichier de sortie (*preprocessed*) :

  ```
  int val = 3;
  std::cout << val;
  ```

  Bogue: *Ne fonctionne pas si utilisé sur une ligne contenant un `#define` utilisé par le préprocesseur.*

- **Expression (`PP_EXPR`)**

  Utilisation :

  ```
  /*=== PP_EXPR(expression_python) [comment] ===*/
  ```

  Évaluation d'une expression *Python* (une chaîne de caractères qui est valide dans la fonction `eval` de *Python*) — ne peut donc pas être une affectation ou autre code complexe (sentence). Les expressions valides sont de la forme : « `1 + 3` », « `MY_CPP_DEFINE` », « `MY_CPP_DEFINE / 3 - 1` », etc. Le *TAG* est remplacé par le résultat de l'évaluation dans le fichier généré.

  Fichier d'entrée (*template*) :

  ```
  #define MY_DEF_VAL 999

  /* Calcule MY_DEF_VAL * 2
  ```

```
   Devrait donner : {*=== PP_EXPR(MY_DEF_VAL * 2) ===*}.

   Remarque : MY_DEF_VAL vaut {*=== PP_EXPR(MY_DEF_VAL) ===*}.
*/

int val = /*=== PP_EXPR(MY_DEF_VAL * 2) ===*/;
```

Fichier de sortie (*preprocessed*) :

```
#define MY_DEF_VAL 999

/* Calcule MY_DEF_VAL * 2

   Devrait donner : 1998.

   Remarque : MY_DEF_VAL vaut 999.
*/

int val = 1998;
```

- **If...then...else (`PP_IF` ... `PP_ELSE` ... `PP_ENDIF`)**

  Utilisation :

```
/*=== PP_IF(expression) [comment] ===*/
// code C++ à générer (ou code du préprocesseur à interpréter) si True
[/*=== PP_ELSE [comment] ===*/]
// code C++ à générer (ou code du préprocesseur à interpréter) si False
/*=== PP_ENDIF [comment] ===*/
```

  Permet la sélection du code à générer. L'argument `expression` peut être tout code *Python* (valide dans sa fonction `eval`) qui retourne `True` ou `False`. Le `PP_ELSE` est optionnel et les `PP_IF` imbriqués sont supportés.

  Bogue: *Il y a un bogue si un `PP_IF` ne se trouve pas dans une boucle. Puisque le préprocesseur avait été créé avant tout pour introduire la possibilité d'utiliser des boucles, le code est construit autour des boucles et il faut absolument la présence d'une boucle en tout temps. Ce bogue serait simple à corriger, mais pour l'instant il est nécessaire d'ajouter une boucle factice (`DUMMY`) quand aucune boucle n'est présente (ex. en rouge dans l'exemple suivant).*

Fichier d'entrée (*template*) :

```
#define MY_DEF_VAL 999

/* MY_DEF_VAL vaut 99 ou 999?

   Remarque: MY_DEF_VAL vaut {*=== PP_EXPR(MY_DEF_VAL) ===*}.
*/

/*=== PP_REPEAT_BEGIN(DUMMY, [1]) Bogue: boucle DUMMY ===*/
    /*=== PP_IF(MY_DEF_VAL == 99) IF principal ===*/
std::cout << "MY_DEF_VAL vaut 99" << std::endl;
    /*=== PP_ELSE ===*/
        /*=== PP_IF(MY_DEF_VAL == 999) IF imbriqué ===*/
std::cout << "MY_DEF_VAL vaut 999" << std::endl;
        /*=== PP_ELSE ===*/
std::cout << "MY_DEF_VAL ne vaut pas 99 ou 999" << std::endl;
        /*=== PP_ENDIF ===*/
    /*=== PP_ENDIF ===*/
/*=== PP_REPEAT_END Bogue: boucle DUMMY ===*/
```

Fichier de sortie (*preprocessed*) :

```
#define MY_DEF_VAL 999

/* MY_DEF_VAL vaut 99 ou 999?

   Remarque: MY_DEF_VAL vaut 999.
*/

std::cout << "MY_DEF_VAL vaut 999" << std::endl;
```

- **Boucles (`PP_REPEAT_BEGIN` ... `PP_REPEAT_END`)**

  Utilisation :

  ```
  /*=== PP_REPEAT_BEGIN(iterateur_id, list | range) [comment] ===*/
  // code C++ ou du préprocesseur à répéter
  /*=== PP_REPEAT_END [comment] ===*/
  ```

  Répète la génération de code un certain nombre de fois. Les arguments `list` et `range` peuvent être du code *Python* qui retourne un `list` ou un `range`. Les variables `__INDEX__` ou `__INDEX?__` sont automatiquement générées. Ces variables contiennent un compteur d'itération et peuvent être utiles pour la manipulation d'array `C++`. Dans le cas de boucles imbriquées, le compteur de la boucle

extérieur est nommé `__INDEX__` ou `__INDEX1__`, celui de la première boucle imbriquée se nomme `__INDEX2__`, etc. (voir l'exemple « *Utilisation des `__INDEX?__` dans des boucles imbriquées* »).

Important: *Les `#define` dans le bloc de la boucle vont être automatiquement remplacés par leur valeur lors de la première phase ! Même ceux non utilisés par les commandes du préprocesseur — ce détail devrait être corrigé, car il peut être problématique dans certains cas.*

Fichier d'entrée (*template*) :

```
#define ARRAY_SIZE 4

int my_array[/*=== PP_EXPR(ARRAY_SIZE) ===*/];

/*
  PP_REPEAT utilisant un "range"
*/
/*=== PP_REPEAT_BEGIN(PP_MY_IND__, range(0,ARRAY_SIZE)) ===*/
    my_array[/*=== PP_EXPR(PP_MY_IND__) ===*/] =
        /*=== PP_EXPR(PP_MY_IND__ * 2) ===*/;
/*=== PP_REPEAT_END ===*/

/*
  PP_REPEAT utilisant un "list" (version 1)
*/
/*=== PP_REPEAT_BEGIN(PP_MY_VALS__, [['a'],['b'],['c']]) ===*/
    my_array[/*=== PP_EXPR(__INDEX__) ===*/] =
        /*=== PP_EXPR(PP_MY_VALS__[0]) ===*/;
/*=== PP_REPEAT_END ===*/

/*
  PP_REPEAT utilisant un "list" (version 2)
*/
/*=== PP_REPEAT_BEGIN(PP_MY_VALS__, ['a','b','c']) ===*/
    my_array[/*=== PP_EXPR(__INDEX__) ===*/] = PP_MY_VALS__;
/*=== PP_REPEAT_END ===*/
```

Fichier de sortie (*preprocessed*) :

```
#define ARRAY_SIZE 4

int my_array[4];

/*
  PP_REPEAT utilisant un "range"
*/
my_array[0] = 0;
my_array[1] = 2;
```

```
my_array[2] = 4;
my_array[3] = 6;

/*
  PP_REPEAT utilisant un "list" (version 1)
*/
my_array[0] = a;
my_array[1] = b;
my_array[2] = c;

/*
  PP_REPEAT utilisant un "list" (version 2)
*/
my_array[0] = a;
my_array[1] = b;
my_array[2] = c;
```

Important: *Dans le cas des deux exemples utilisant des* `list` *contenant des chaînes de caractères, il est important de noter que pour utiliser directement les* `list` *de chaînes de caractères dans la « version 2 », il ne faut pas utiliser* `PP_EXPR`*, car cela causerait une erreur d'interprétation Python (c.-à-d. que l'expression «* `eval(a)` *» est illégale, car la variable* `a` *n'est pas définie) !*

- **Évaluation de code *Python* (`PP_EVAL`)**

  Utilisation :

  ```
  /*=== PP_EVAL(code_python) [comment] ===*/
  ```

  Ce *TAG* est le plus puissant de tous les *TAGs* — il permet d'exécuter directement du code *Python*. Une mémoire privée lui est allouée — celle-ci est persistante tout le long de l'exécution du préprocesseur. Il est donc possible d'affecter des objets *Python* et de les réutiliser plus loin dans un autre *TAG* (voir le *template* en exemple). Contrairement à `PP_EXPR`, aucun remplacement n'est effectué dans le fichier généré, car `PP_EVAL` ne retourne aucune valeur, le *TAG* est simplement retiré après avoir exécuté le code *Python*. Bien que performant, c'est le seul *TAG* que nous n'avons pas encore eu besoin d'utiliser dans le projet.

  Remarque: *Il serait intéressant d'ajouter des blocs* `PP_EVAL_BEGIN` *...* `PP_EVAL_END` *pour ajouter du code Python multiligne ! Pour l'instant, il est possible d'inclure un fichier Python complexe avec la commande* `PP_EVAL(import MyExtModule; MyExtModule.myfunc();)` *(voir l'exemple « Exemple d'un appel à un code Python externe »).*

  Fichier d'entrée (*template*) :

```c
#define MY_DEF_VAL 999

/* Calcule 1 / 3 + MY_DEF_VAL

   Devrait donner {*=== PP_EXPR(1/3 + MY_DEF_VAL.0) ===*}.

   Remarque: MY_DEF_VAL vaut {*=== PP_EXPR(MY_DEF_VAL) ===*}.
*/

/*=== PP_EVAL(PP_MY_VAR__ = 1/3; PP_MY_VAR__ += MY_DEF_VAL.0) ===*/
float val2 = /*=== PP_EXPR(PP_MY_VAR__) ===*/;
```

Fichier de sortie (*preprocessed*) :

```c
#define MY_DEF_VAL 999

/* Calcule 1 / 3 + MY_DEF_VAL

   Devrait donner 999.3333333333334.

   Remarque: MY_DEF_VAL vaut 999.
*/

float val2 = 999.3333333333334;
```

- **Exemples complexes**

  Utilisation des `__INDEX?__` dans des boucles imbriquées :

  Fichier d'entrée (*template*) :

```c
#define ARRAY_SIZE 4

int my_array[/*=== PP_EXPR(ARRAY_SIZE) ===*/];

/*=== PP_EVAL(PP_COUNT__ = 0) ===*/
/*=== PP_REPEAT_BEGIN(PP_IND__, range(0,ARRAY_SIZE)) ===*/
    /*=== PP_REPEAT_BEGIN(PP_VAL__, range(10,25,3)) ===*/
        /*=== PP_IF(PP_VAL__ % 2 == 0) Is the value even? ===*/
            // my_arrayDEF___INDEX2__[DEF_PP_IND__] = DEF___INDEX1__ *
 ↪  DEF_PP_VAL__ + DEF_PP_COUNT__ = /*=== PP_EXPR(__INDEX1__) ===*/ *
 ↪  /*=== PP_EXPR(PP_VAL__) ===*/ + /*=== PP_EXPR(PP_COUNT__) ===*/ =
 ↪  /*=== PP_EXPR(__INDEX1__ * PP_VAL__ + PP_COUNT__) ===*/
            my_array/*=== PP_EXPR(__INDEX2__) ===*/[/*===
 ↪  PP_EXPR(PP_IND__) ===*/] = /*=== PP_EXPR(__INDEX1__ * PP_VAL__ +
 ↪  PP_COUNT__) ===*/;
```

```
        /*=== PP_ENDIF ===*/
        /*=== PP_EVAL(PP_COUNT__ += 1) ===*/
    /*=== PP_REPEAT_END ===*/
/*=== PP_REPEAT_END ===*/
```

Fichier de sortie (*preprocessed*) :

```
#define ARRAY_SIZE 4

int my_array[4];

// my_arrayDEF___INDEX2__[DEF_PP_IND__] = DEF___INDEX1__ * DEF_PP_VAL__
 ↪  + DEF_PP_COUNT__ = 0 * 10 + 20 = 0
my_array0[0] = 0;
// my_arrayDEF___INDEX2__[DEF_PP_IND__] = DEF___INDEX1__ * DEF_PP_VAL__
 ↪  + DEF_PP_COUNT__ = 0 * 16 + 20 = 2
my_array2[0] = 2;
// my_arrayDEF___INDEX2__[DEF_PP_IND__] = DEF___INDEX1__ * DEF_PP_VAL__
 ↪  + DEF_PP_COUNT__ = 0 * 22 + 20 = 4
my_array4[0] = 4;
// my_arrayDEF___INDEX2__[DEF_PP_IND__] = DEF___INDEX1__ * DEF_PP_VAL__
 ↪  + DEF_PP_COUNT__ = 1 * 10 + 20 = 15
my_array0[1] = 15;
// my_arrayDEF___INDEX2__[DEF_PP_IND__] = DEF___INDEX1__ * DEF_PP_VAL__
 ↪  + DEF_PP_COUNT__ = 1 * 16 + 20 = 23
my_array2[1] = 23;
// my_arrayDEF___INDEX2__[DEF_PP_IND__] = DEF___INDEX1__ * DEF_PP_VAL__
 ↪  + DEF_PP_COUNT__ = 1 * 22 + 20 = 31
my_array4[1] = 31;
// my_arrayDEF___INDEX2__[DEF_PP_IND__] = DEF___INDEX1__ * DEF_PP_VAL__
 ↪  + DEF_PP_COUNT__ = 2 * 10 + 20 = 30
my_array0[2] = 30;
// my_arrayDEF___INDEX2__[DEF_PP_IND__] = DEF___INDEX1__ * DEF_PP_VAL__
 ↪  + DEF_PP_COUNT__ = 2 * 16 + 20 = 44
my_array2[2] = 44;
// my_arrayDEF___INDEX2__[DEF_PP_IND__] = DEF___INDEX1__ * DEF_PP_VAL__
 ↪  + DEF_PP_COUNT__ = 2 * 22 + 20 = 58
my_array4[2] = 58;
// my_arrayDEF___INDEX2__[DEF_PP_IND__] = DEF___INDEX1__ * DEF_PP_VAL__
 ↪  + DEF_PP_COUNT__ = 3 * 10 + 20 = 45
my_array0[3] = 45;
// my_arrayDEF___INDEX2__[DEF_PP_IND__] = DEF___INDEX1__ * DEF_PP_VAL__
 ↪  + DEF_PP_COUNT__= 3 * 16 + 20 = 65
my_array2[3] = 65;
// my_arrayDEF___INDEX2__[DEF_PP_IND__] = DEF___INDEX1__ * DEF_PP_VAL__
 ↪  + DEF_PP_COUNT__ = 3 * 22 + 20 = 85
my_array4[3] = 85;
```

Remarque: *Si des* `DEF_` *sont ajoutés devant les noms de* `#define`*, de variables Python et des*

*itérateurs, c'est qu'ils seraient remplacés automatiquement par leur valeur puisqu'ils se trouvent dans une boucle ! Voir la note « **Important:** » dans la section des boucles.*

Exemple d'un appel à un code *Python* externe :

Fichier d'entrée (*template*) :

```
/* Teste un appel à un module Python externe

   Fichier MyExtCode.py :
     def mul2(val):
       return val * 2
*/

/*=== PP_EVAL(import MyExtCode; PP_MY_VAR__ = MyExtCode.mul2(8);) ===*/

int my_ext_val = /*=== PP_EXPR(PP_MY_VAR__) ===*/;
```

Fichier de sortie (*preprocessed*) :

```
/* Teste un appel à un module Python externe

   Fichier MyExtCode.py :
     def mul2(val):
       return val * 2
*/

int my_ext_val = 16;
```

Bogue: *Il y a un bogue dans le parsing quand le dernier caractère du code* `PP_EVAL` *est une parenthèse fermante ! Il faut ajouter un « ; » après la parenthèse (en rouge dans l'exemple).*

ANNEXE D
GRAPHES DÉTAILLÉS

Pour interpréter les légendes des graphes détaillés, consultez le tableau D.1 (page 163). Les informations contenues dans les légendes indiquent les configurations de base ; pour obtenir plus de détails, il faut décoder le code *QR* dans le coin droit dans le bas du graphe. Ce code *QR* contient un objet *JSON* (listing D.1) contenant, pour chacune des courbes du graphe, les configurations du compilateur `C++`, la date de compilation, l'identifiant du *commit Git*, les configurations du logiciel lors de la compilation et de l'exécution, etc.

```
{
    "name": "Nom du graphe",
    "1": {
        "name": "Nom de la courbe du test",
        "configs": {
            "date": "Date du test",
            "git": "Identifiant du commit",
            "checksum": "Hachage SHA-256 de l'exécutable",
            "system": "Informations sur le système",
            "compiler": "Nom et version du compilateur C++",
            "compiler_flags": "Opions du compilateurs",
            "linker_flags": "Options du linker",
            "boost": "Version de la bibliothèque Boost",
            "defines": "Macro C++ importantes",
            "configs": "Configurations importantes"
        }
    }
}
```

Listing D.1 Exemple des données *JSON* pour une courbe d'un graphe.

Puisqu'il est nécessaire d'encoder une grande quantité d'informations dans le code *QR*, il utilise la capacité maximale possible pour un tel code[97] et l'objet *JSON* y est compressé avec l'outil *zbar*[98] et encodé en *base45*[99]. Il ne peut donc pas être décodé directement, il doit être décodé et décompressé manuellement (listing D.2).

[97] Soit un code *QR 40-L* ayant une capacité maximale d'environ 2953 octets (`https://en.wikipedia.org/wiki/QR_code#Information_capacity`).

[98] La version de *zbar* utilisée doit être minimalement 7.15 ; les versions précédentes n'utilisaient pas le même format d'encodage.

[99] L'encodage recommandé pour des données binaires dans un code *QR* (`https://datatracker.ietf.org/doc/html/rfc9285#name-when-to-use-and-not-use-bas`).

```
sudo apt install zbar-tools zpaq jq
pip install codext

zbarimg -q --raw qrcode.png | base45 --decode > qrcode.zpaq
zpaq x qrcode.zpaq
cat temp_qr_data.txt | jq -M
```

Listing D.2 Décodage manuel des codes *QR*[100].

Il est aussi possible de décoder automatiquement tous les codes *QR* présents dans le mémoire, sous *Linux*, avec la commande «`./compile qrdecode`» dans le dépôt du projet si le fichier `memoire.pdf` est placé dans le répertoire `doc/`[101].

Certains graphes ont été générés par une ancienne version du logiciel et sont identifiés par la présence de l'icône 🕘 à gauche de leur légende.

[100] Le code *QR* doit avoir été extrait du document avec une grande résolution. Une solution simple consiste à utiliser le logiciel vectoriel *Inkspace* (gratuit) pour ouvrir une page du mémoire et sélectionner (*CTRL-bouton gauche*) le code *QR* à extraire. Il est possible de l'exporter en format *PNG* en sélectionnant `File/Export PNG Image...` dans le menu et ensuite cliquer sur l'onglet `Selection`, puis choisir une valeur de *dpi* de 2000 (ou choisir une taille d'image d'un minimum de 600 par 600 pixels) avant de cliquer sur le bouton `Export`. Il est à noter que, vu la grande résolution du code *QR* et sa petite taille, il est peu probable qu'il puisse être décodé s'il est extrait d'une version imprimée du document.

[101] Certaines dépendances doivent cependant être préalablement installées (voir le fichier `doc/qrdecode.readme` dans le dépôt du projet) .

section	champs	descriptions
Ramasse-miettes	**gc:***version* **[(**rm**|**rmnc**)]** **[/**alloc**|**ucalloc**(***pourcent***%|**dyn**)]** **|** non	• *version* *Version du GC (ex. 8.0.4)* • rm *Mode redirect* malloc • rmnc *Mode redirect* malloc *sans collections* • alloc *Allocateurs mémoire avec collections* • ucalloc *Allocateurs mémoire sans collections* • *pourcent* *Pourcentage des collectes non monitorées (ex. 50%)* • dyn *Ajustement dynamique des collectes non monitorées* • non *GC désactivé*
Options mémoire	**opt:***malloc_trim*,*mode_mem*,*mode_test*	• *malloc_trim* *Utilisation de* malloc_trim *pour libérer la mémoire* ML (libère la mémoire avec malloc_trim), MD (comportement par défaut) ou NA (non applicable sous *Windows*). • *mode_mem* *Mode de lecture mémoire utilisée* U0 (kernel), U1 (malloc_trim), U2 (mallinfo), U3 (mallocinfo), U4 (gcalloc), U5 (gcallocobjs) ou NA (non applicable sous *Windows*). • *mode_test* *Mode de création d'objets dans le test du GC* GC (objets gérés par le *GC*), GCV (vecteurs d'objets gérés par le *GC*), SP (pointeurs intelligents) ou SPV (vecteurs de pointeurs intelligents).
Diviseur du *GC*	**div:***valeur*	• *valeur* *Valeur du diviseur du GC (ex. 3, voir section 6.5)*
Seuil de consommation	**seuil:***mode***/***config*	• *mode* *Mode de seuil de consommation mémoire (0/désactivé à 5, voir section 6.6)* • *config* *Configuration de mode de seuil (% ou MB pour le mode 5)*
Taille du tas (*heap*)	**tas:***min***/***max* MB	• *min* *Taille minimale (en MB, voir section 6.8)* • *max* *Taille maximale (en MB)*
Mode incrémentiel	**inc:**oui**|**non**[,**freq**:***fréquence***]**	• oui**|**non *Mode incrémentiel activé/désactivé (voir section 6.7)* • *fréquence* *Fréquence du mode incrémentiel*
Objets éphémères	**ephm:**oui**|**non	• oui**|**non *Objets éphémères activés/désactivés (voir section 7.3)*
Type arithmétique	**type:***nom_ver***/**2**|**10**[**v1**]**	• *nom_ver* *Nom et version de la bibliothèque arithmétique (ex. gmp6.1)* • 2**|**10 *Base 2 ou 10* • v1 *Utilise l'ancienne version des objets* Numeric
Précision	**préc:***valeur*	• *valeur* *Précision arithmétique utilisée (ex. 10000)*
Compilateur	**cmpl:***nom_ver*	• *nom_ver* *Nom et version du compilateur C++ (ex. gcc-v9.4)*

Tableau D.1 Guide d'interprétation des légendes des graphes détaillés.

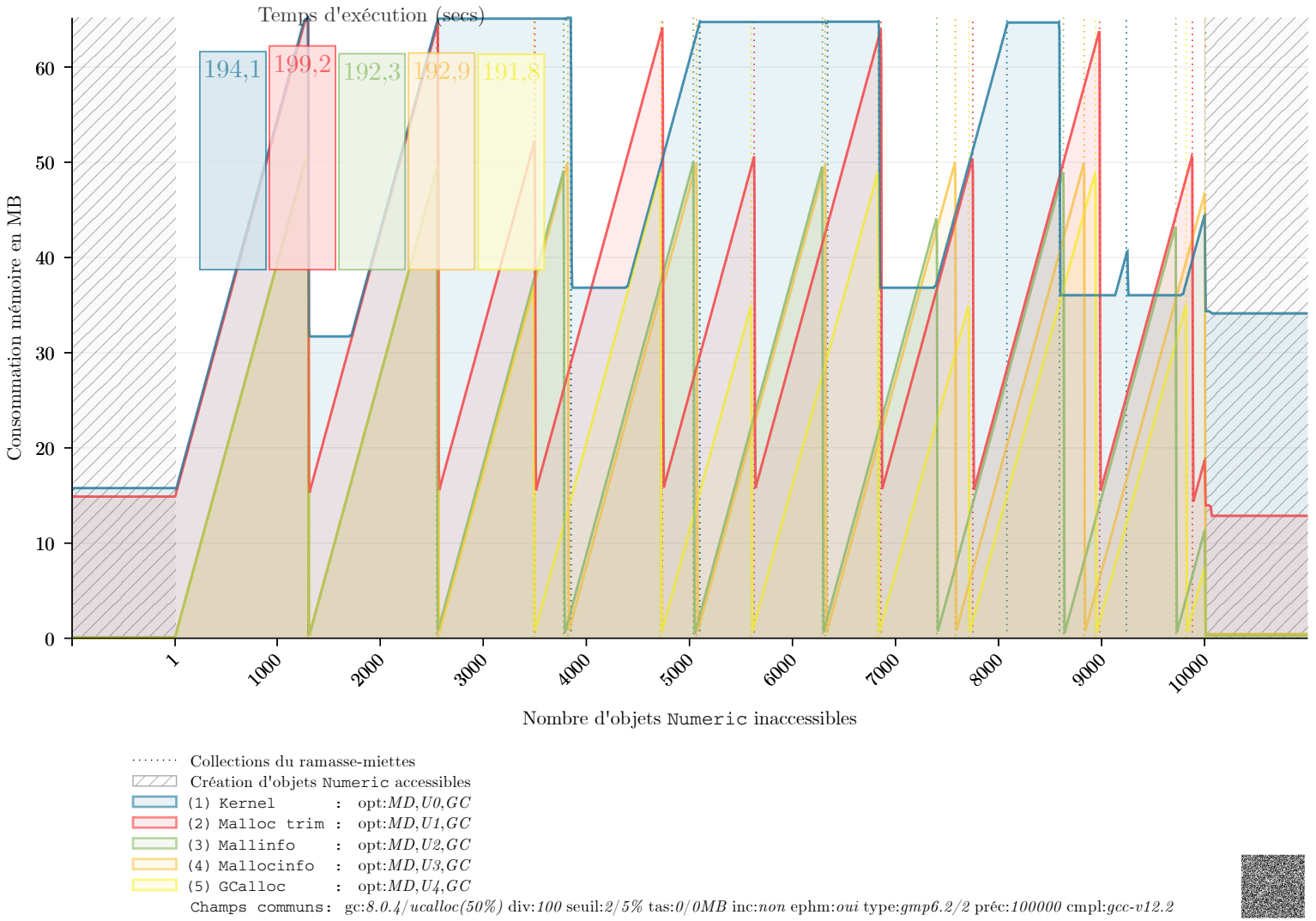


Figure 6.1 Lecture mémoire libre dans *Linux*[102] (*retour*).

[102]Généré sous *Debian 12.1* 64 bits (x86-64 40x3GHz/31GB).

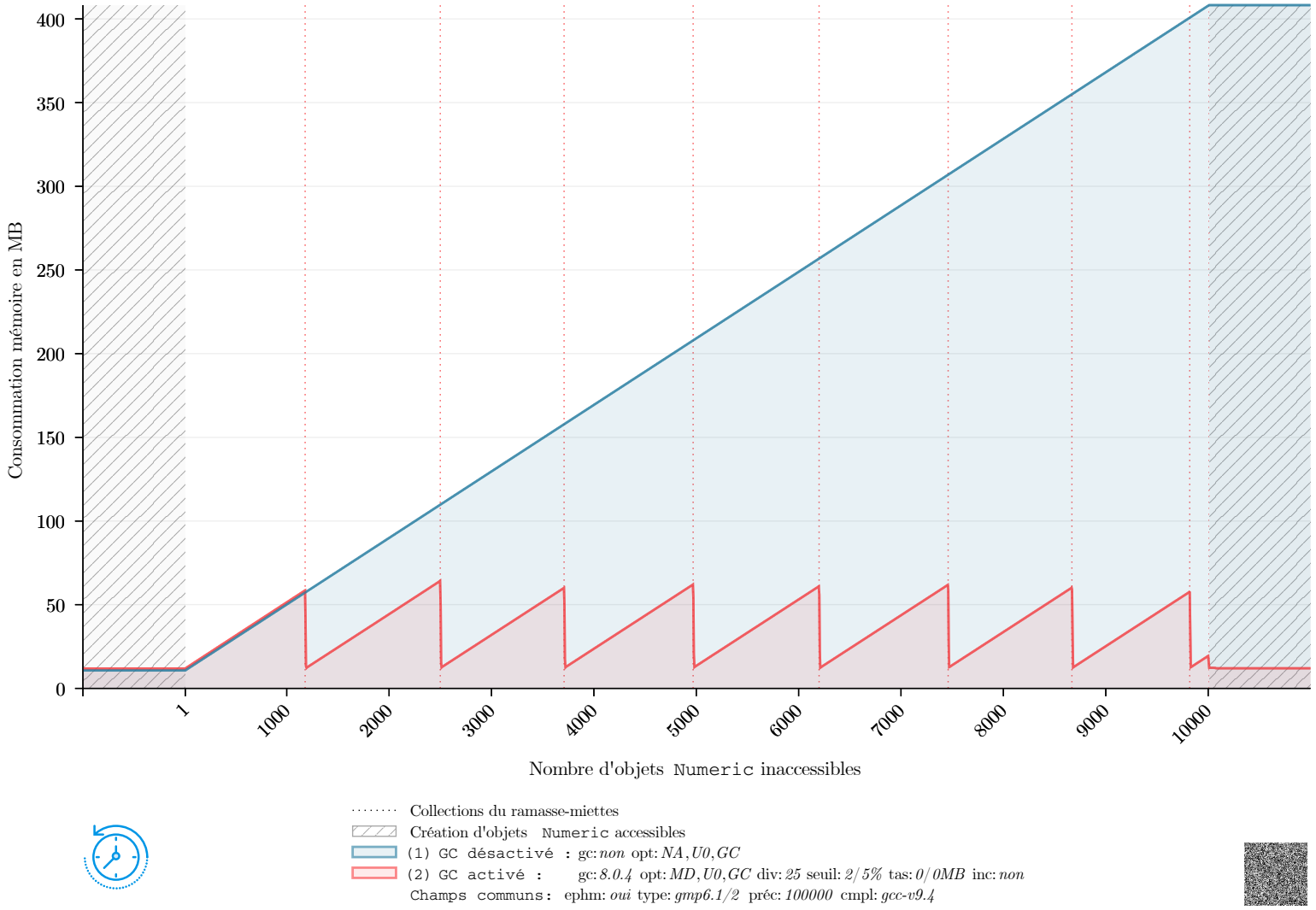


Figure 6.2 Comportement avec et sans *GC* dans *Ubuntu 16.04*[103] (*retour*).

[103] Généré sous *Linux Ubuntu 16.04.7 LTS* 64 bits (x86-64 2x2.26GHz/8GB).

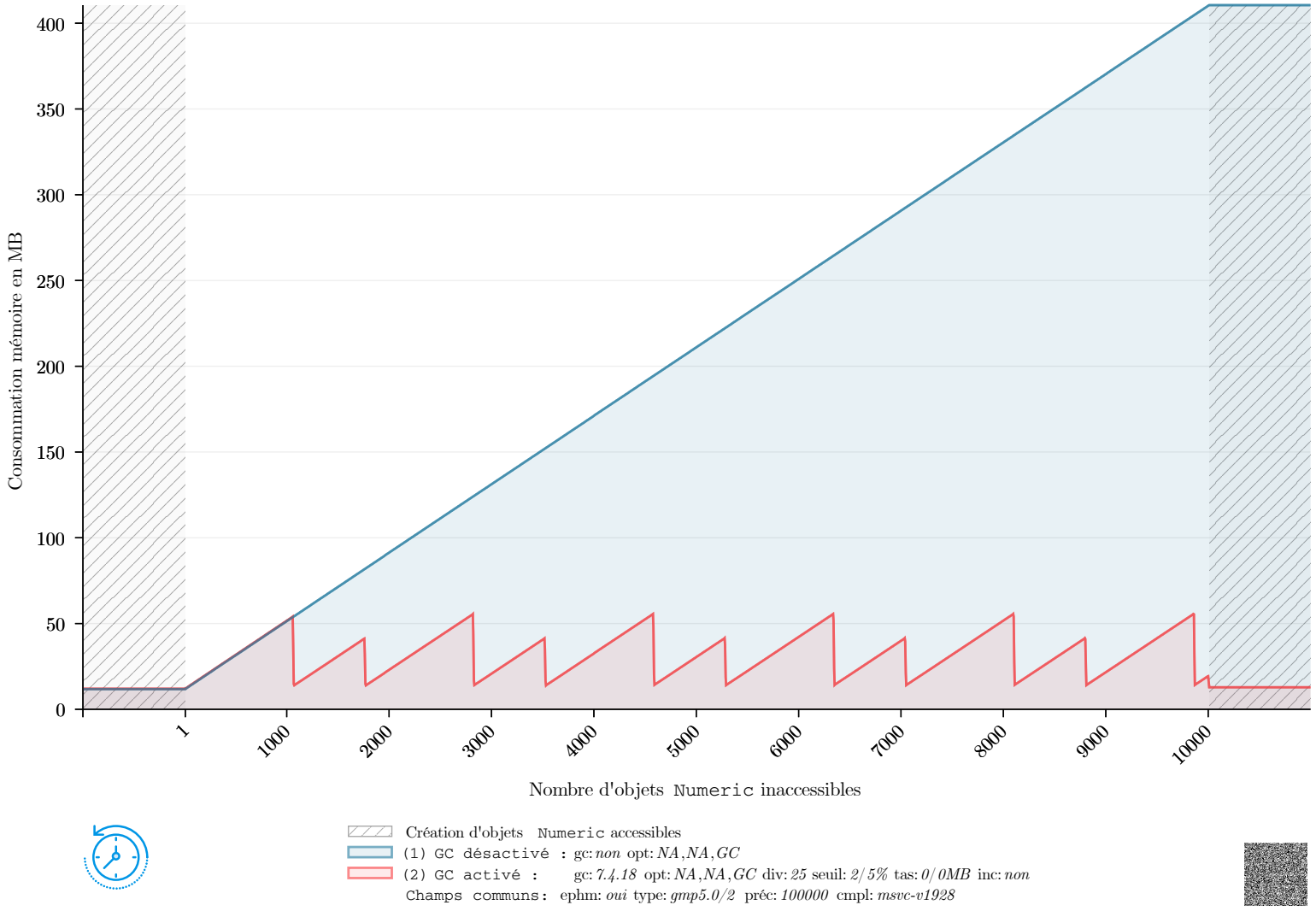


Figure 6.3 Comportement avec et sans *GC* dans *Windows 7*[104] (*retour*).

[104]Généré sous *Microsoft Windows 7 Professional* 64 bits (x86-64 8x2.94GHz/12.0GB). À noter que la version 7.4.18 du *GC* ne permet pas de détecter les collectes.

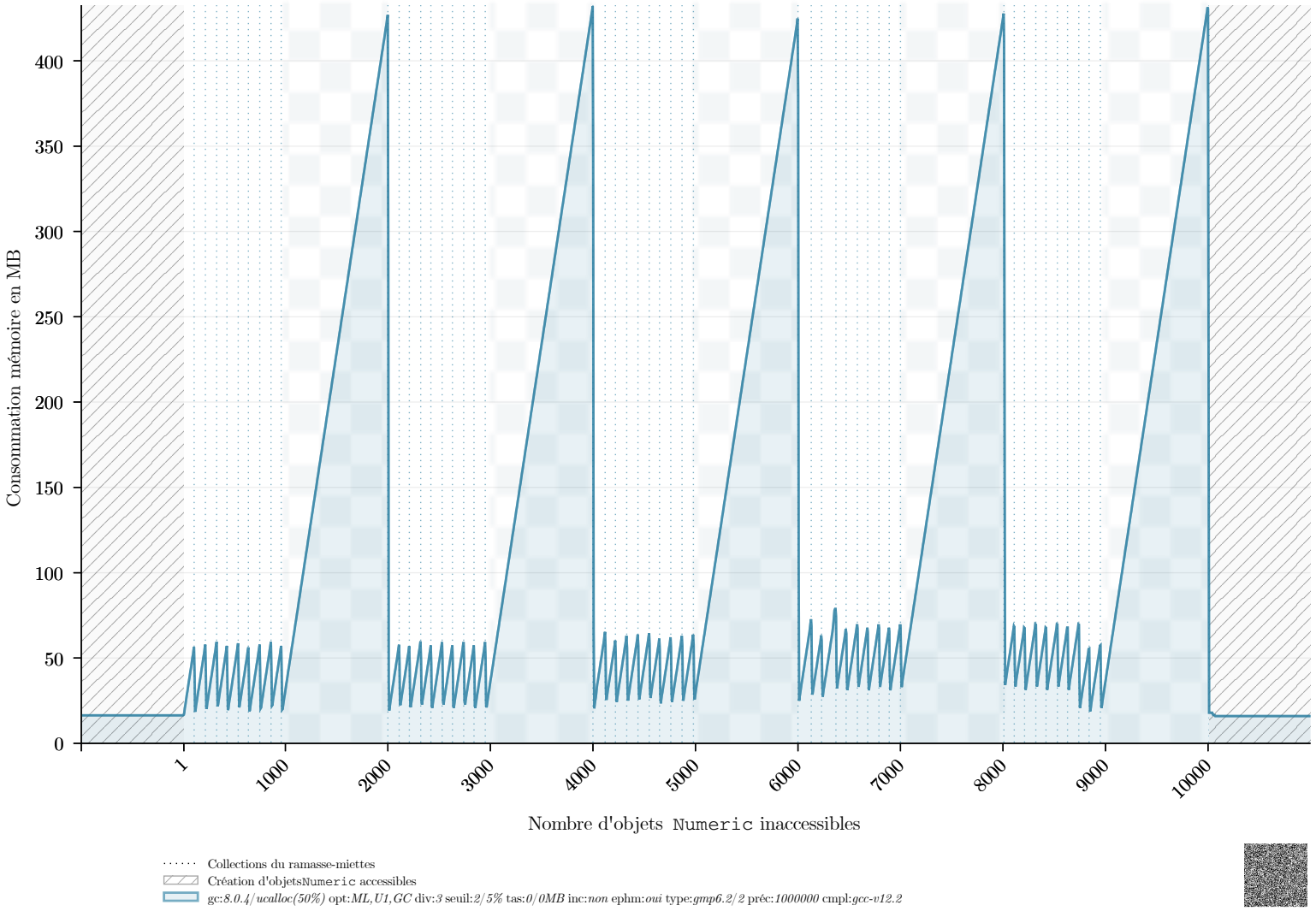


Figure 6.4 Activation/désactivation dynamique du *GC* dans *Debian 12.1*[105] (*retour*).

[105] Les zones avec un arrière-plan en échiquier représentent les périodes où le *GC* est désactivé temporairement. Généré sous *Debian 12.1* 64 bits (x86-64 40x3GHz/31GB).

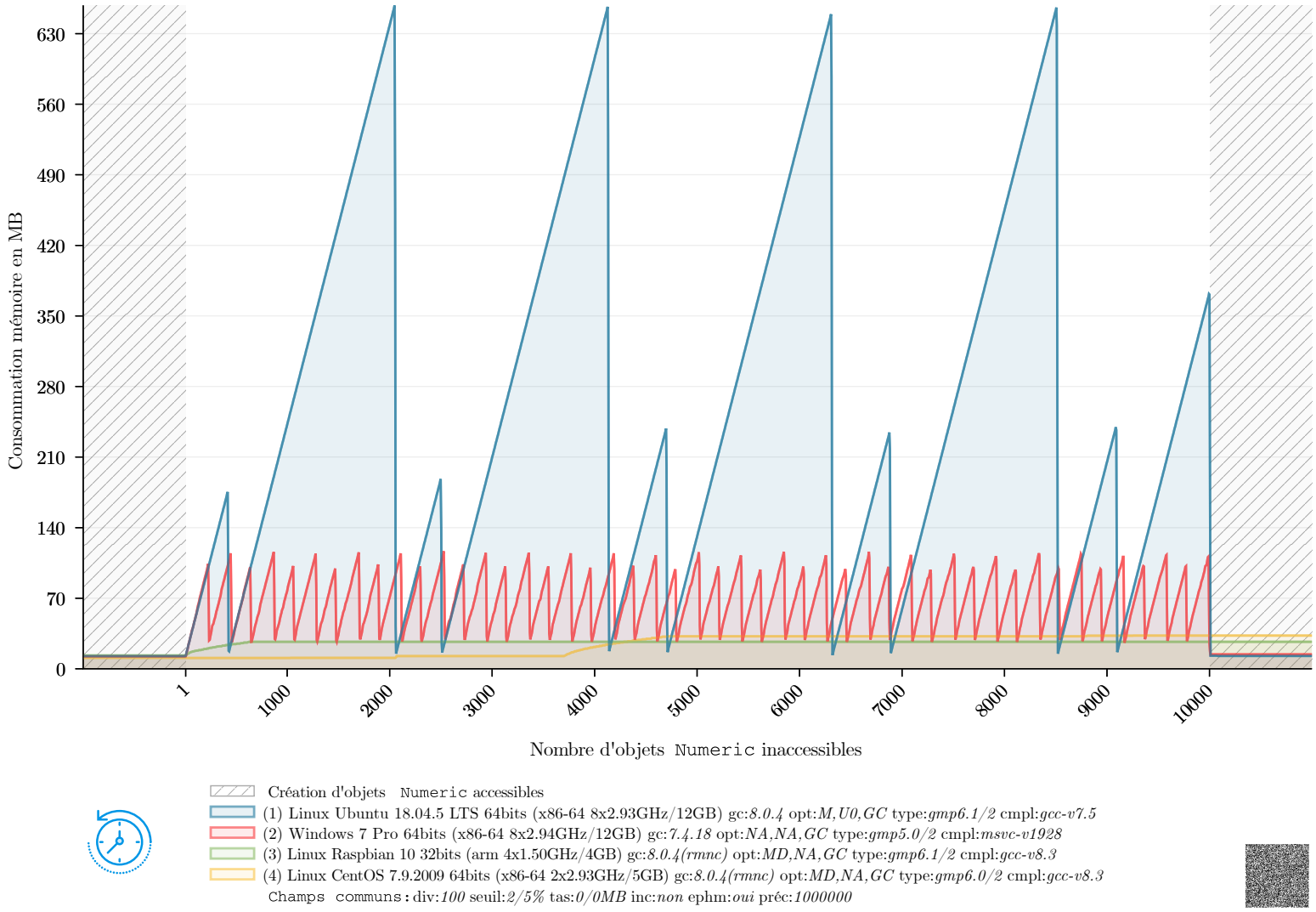


Figure 6.5 Comportement du *GC* sous différentes plateformes (ancienne version)[106] (*retour*).

[106] À noter que l'affichage des collectes du *GC* a été désactivé pour alléger le graphe (courbes 3 et 4) et que la plateforme *Debian* n'est pas supportée par cette ancienne version.

Figure 6.6 Comportement du *GC* sous différentes plateformes (*retour*).

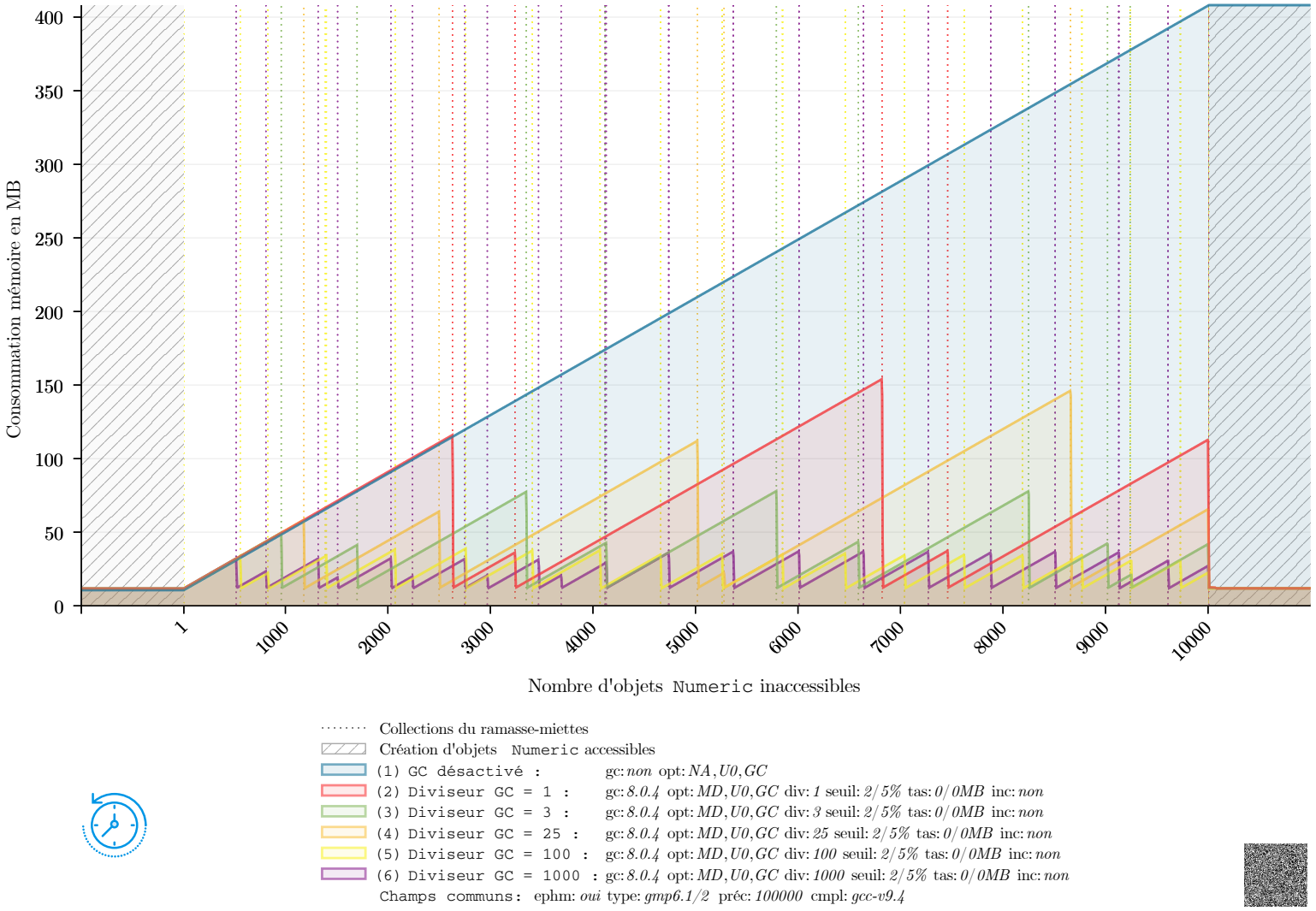


Figure 6.7 Différentes valeurs du diviseur du *GC* dans *Ubuntu 16.04*[107] (*retour*).

[107] Généré sous *Linux Ubuntu 16.04.7 LTS* 64 bits (x86-64 2x2.26GHz/8GB).

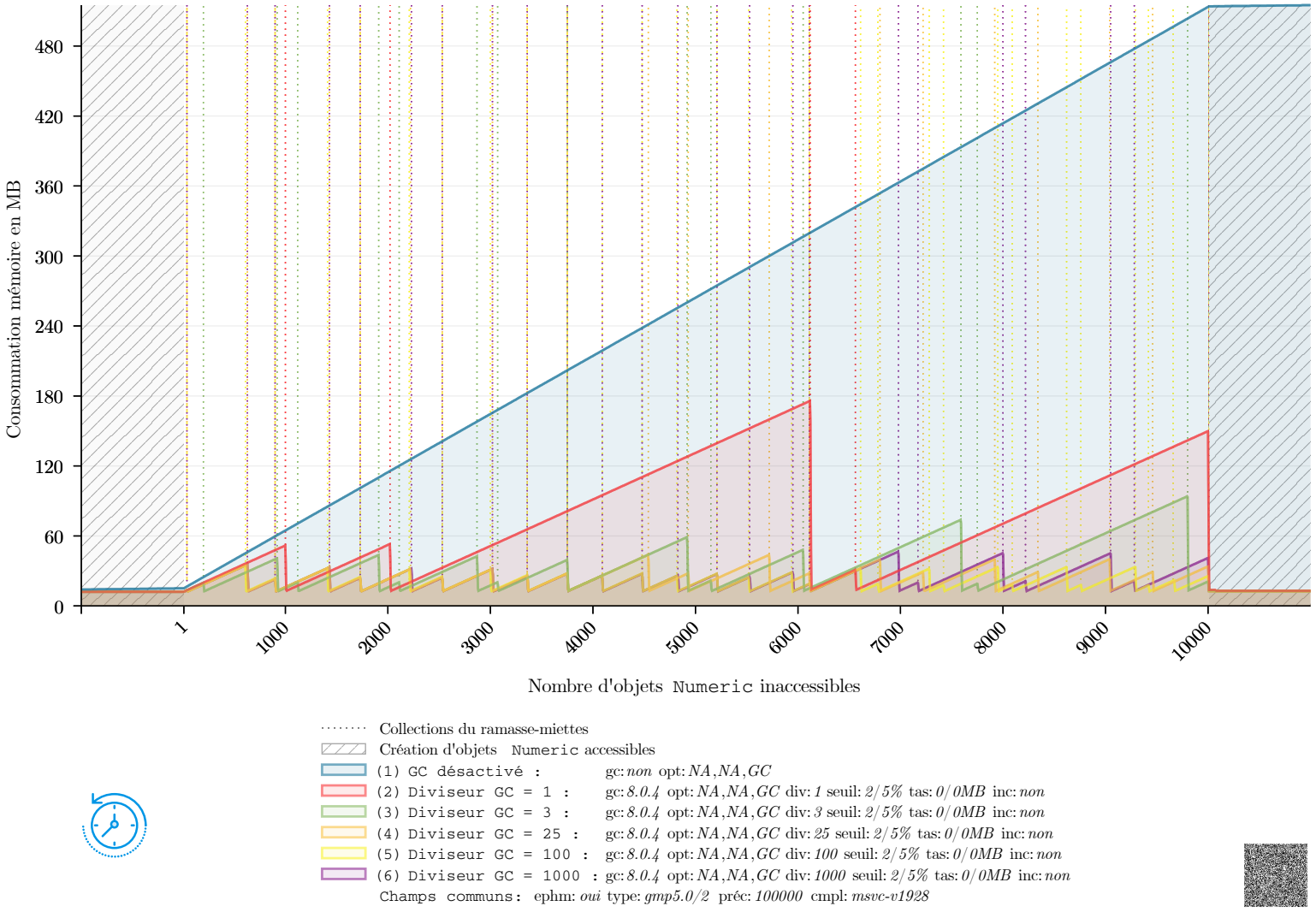


Figure 6.8 Différentes valeurs du diviseur du *GC* dans *Windows 7*[108] (*retour*).

[108] Généré sous *Microsoft Windows 7 Professional* 64 bits (x86-64 8x2.94GHz/12.0GB).

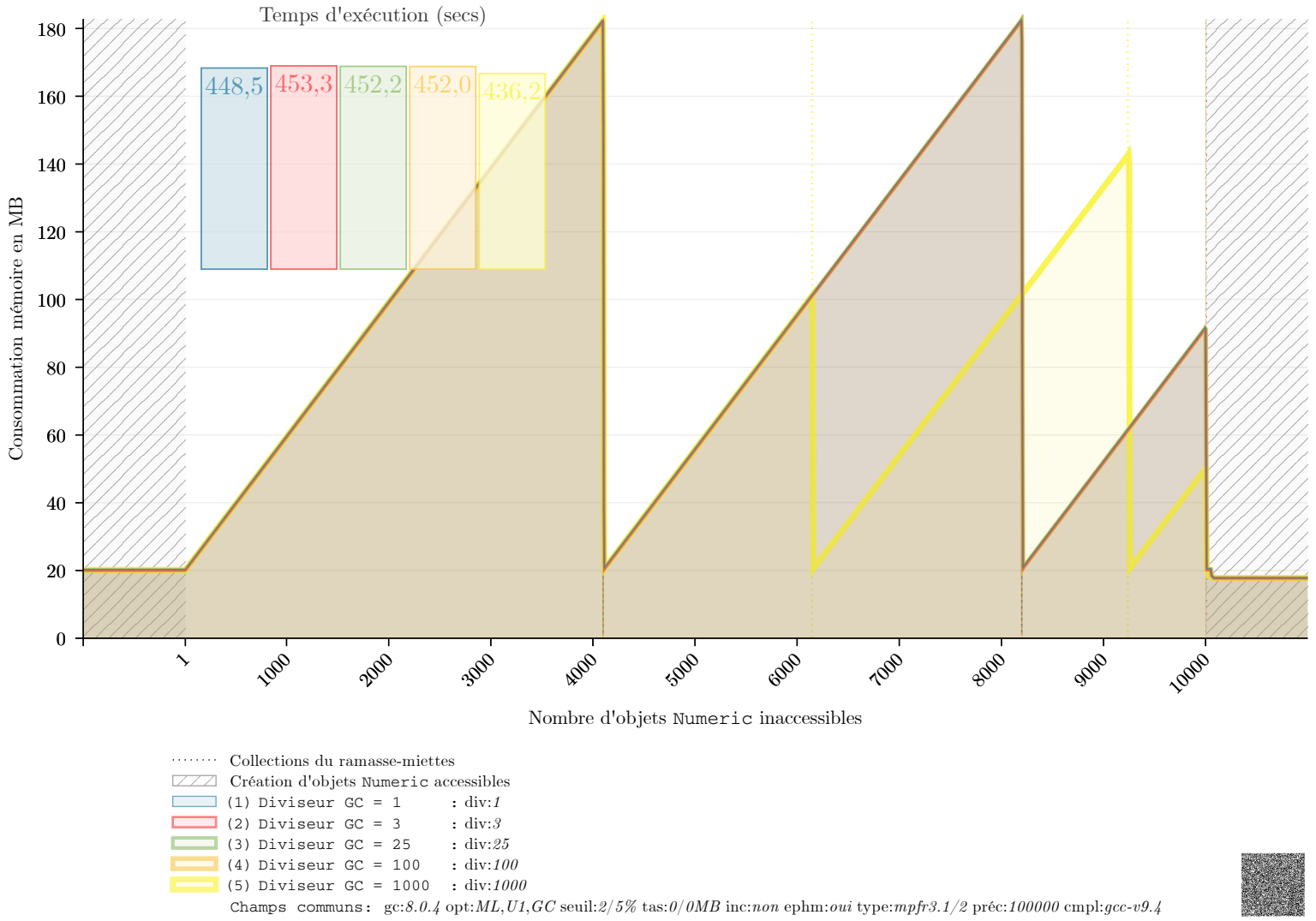


Figure 6.9 Différentes valeurs du diviseur du *GC* dans *Ubuntu 16.04*[109] (nouvelle version, *retour*).

[109] Généré sous *Linux Ubuntu 16.04.7 LTS* 64 bits (x86-64 2x2.26GHz/8GB).

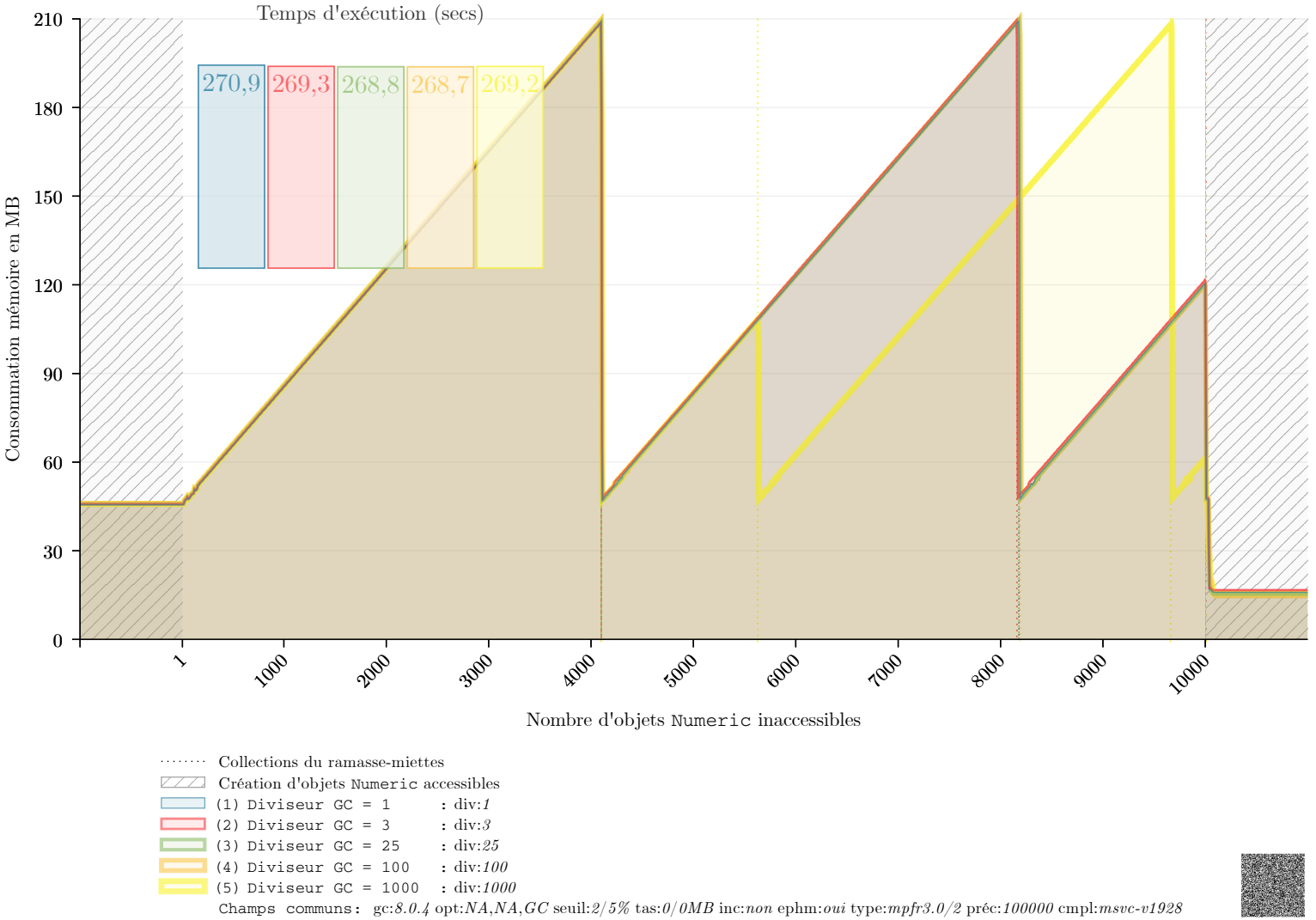


Figure 6.10 Différentes valeurs du diviseur du *GC* dans *Windows 7*[110] (nouvelle version, *retour*).

[110] Généré sous *Microsoft Windows 7 Professional* 64 bits (x86-64 8x2.94GHz/12.0GB).

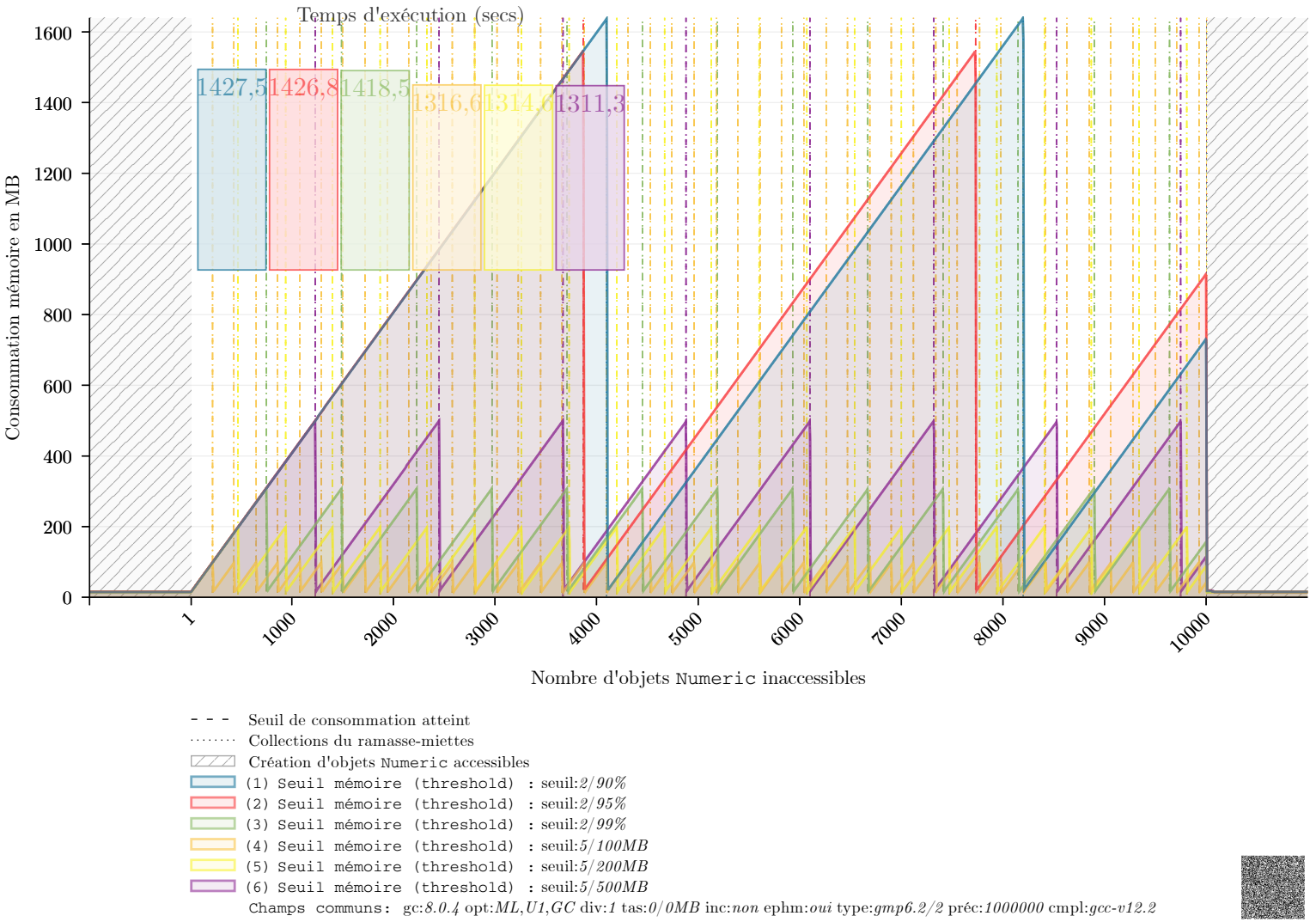


Figure 6.11 Différentes seuils de consommation mémoire dans *Debian 12.1*[111] (*retour*).

[111] Généré sous *Debian 12.1* 64 bits (x86-64 40x3GHz/31GB).

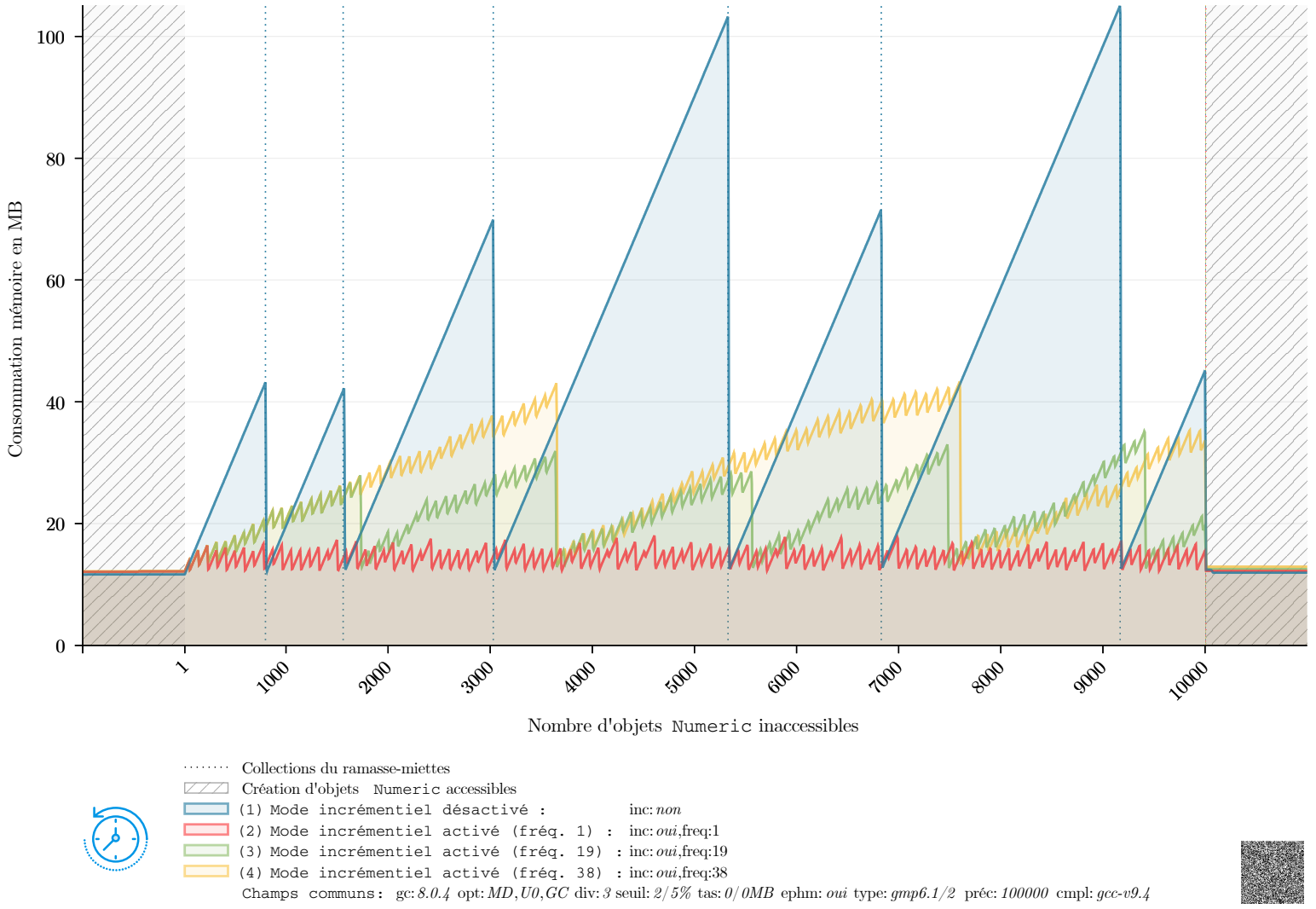


Figure 6.12 Différentes fréquences du mode incrémentiel du *GC* dans *Ubuntu 16.04*[112] (*retour*).

[112]Généré sous *Linux Ubuntu 16.04.7 LTS* 64 bits (x86-64 2x2.26GHz/8GB).

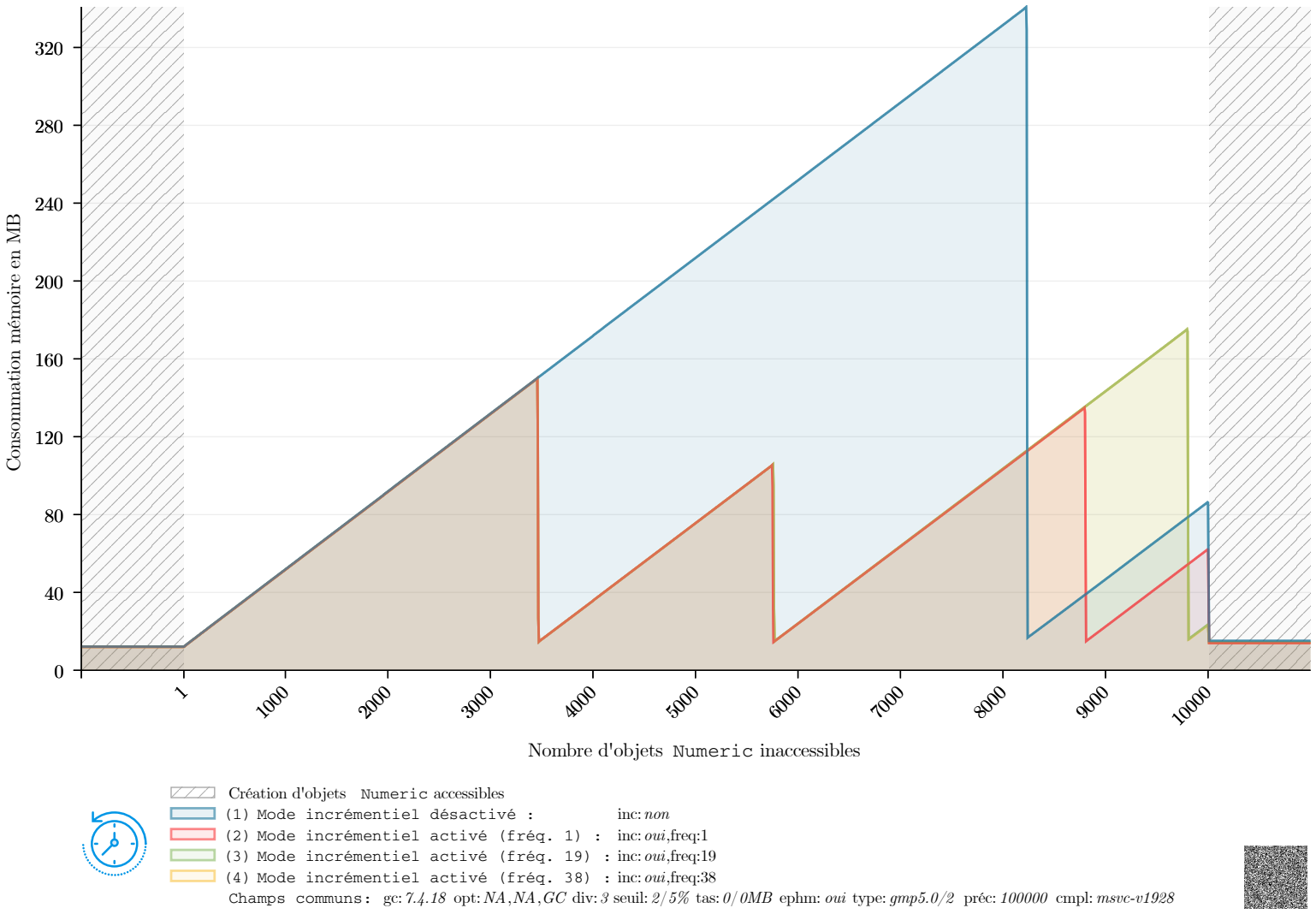


Figure 6.13 Différentes fréquences du mode incrémentiel du *GC* dans *Windows 7*[113] (*retour*).

[113] Généré sous *Microsoft Windows 7 Professional* 64 bits (x86-64 8x2.94GHz/12.0GB).

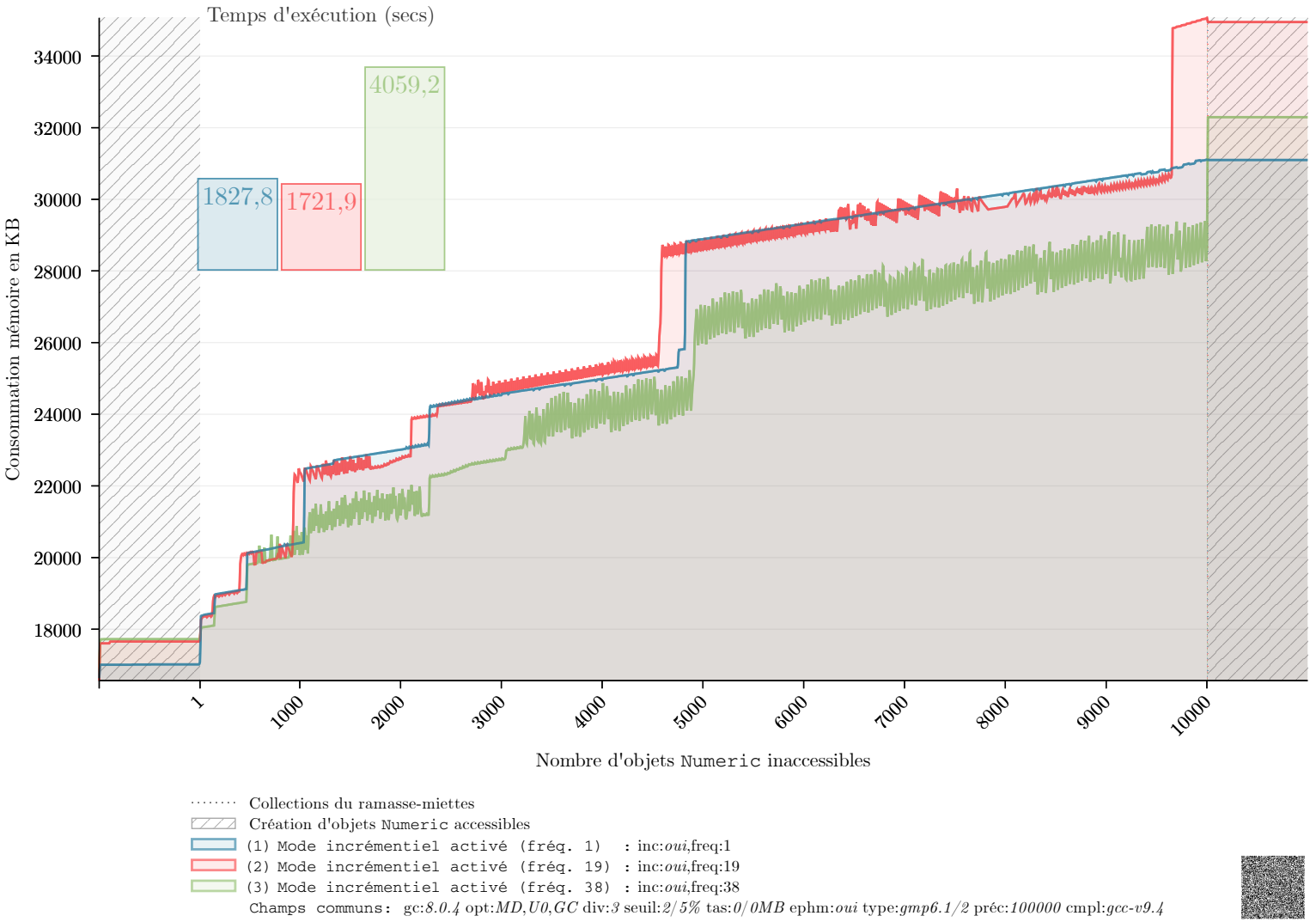


Figure 6.14 Problème de fuites mémoire du mode incrémentiel du *GC* dans *Ubuntu 16.04*[69,114] (*retour*).

[114] Généré sous *Linux Ubuntu 16.04.7 LTS* 64 bits (x86-64 2x2.26GHz/8GB).

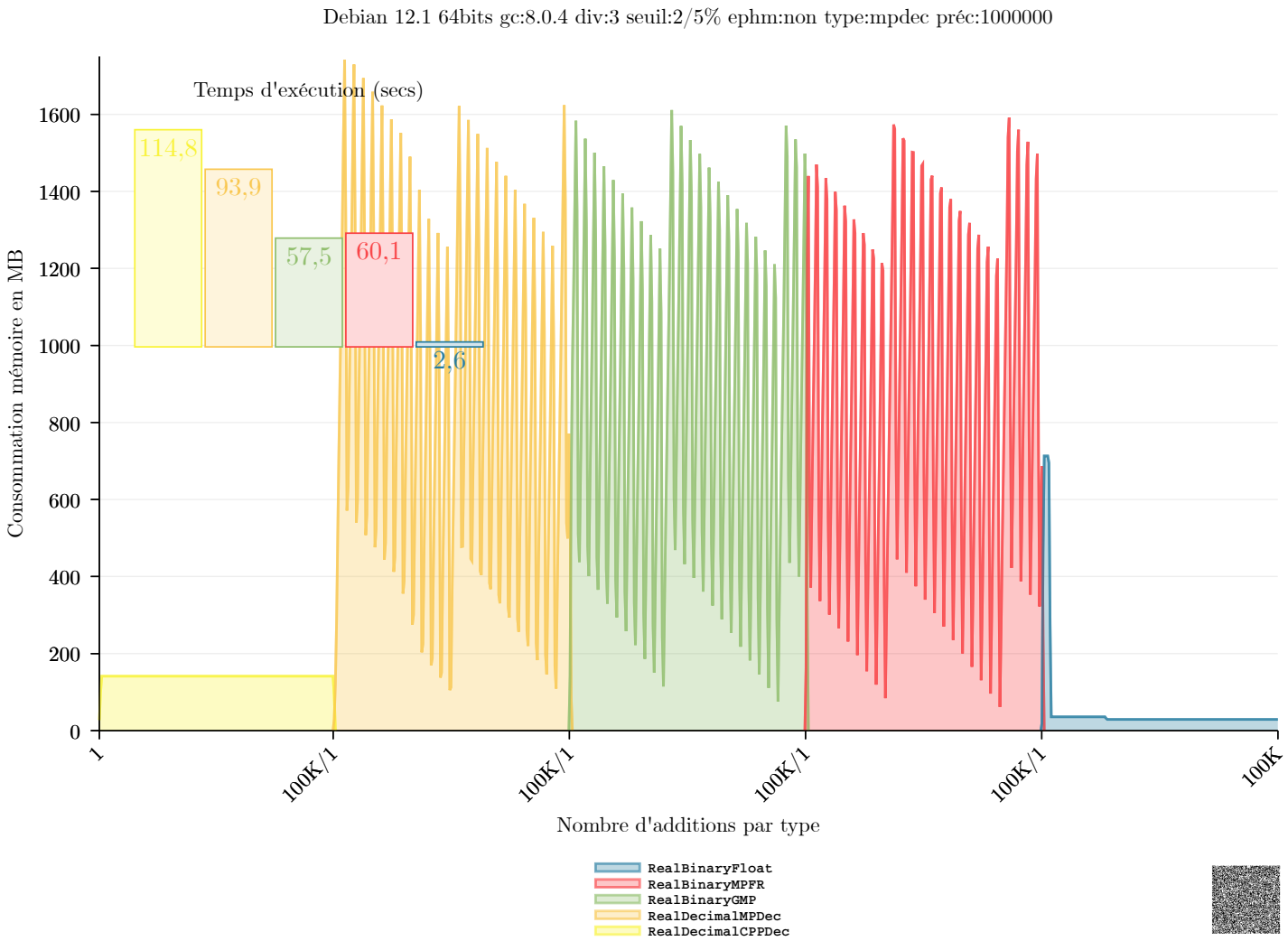


Figure 7.10 Résultats sans objets éphémères dans *Debian 12.1*[115] (*retour*).

[115] Généré sous *Debian 12.1* 64 bits (x86-64 40x3GHz/31GB).

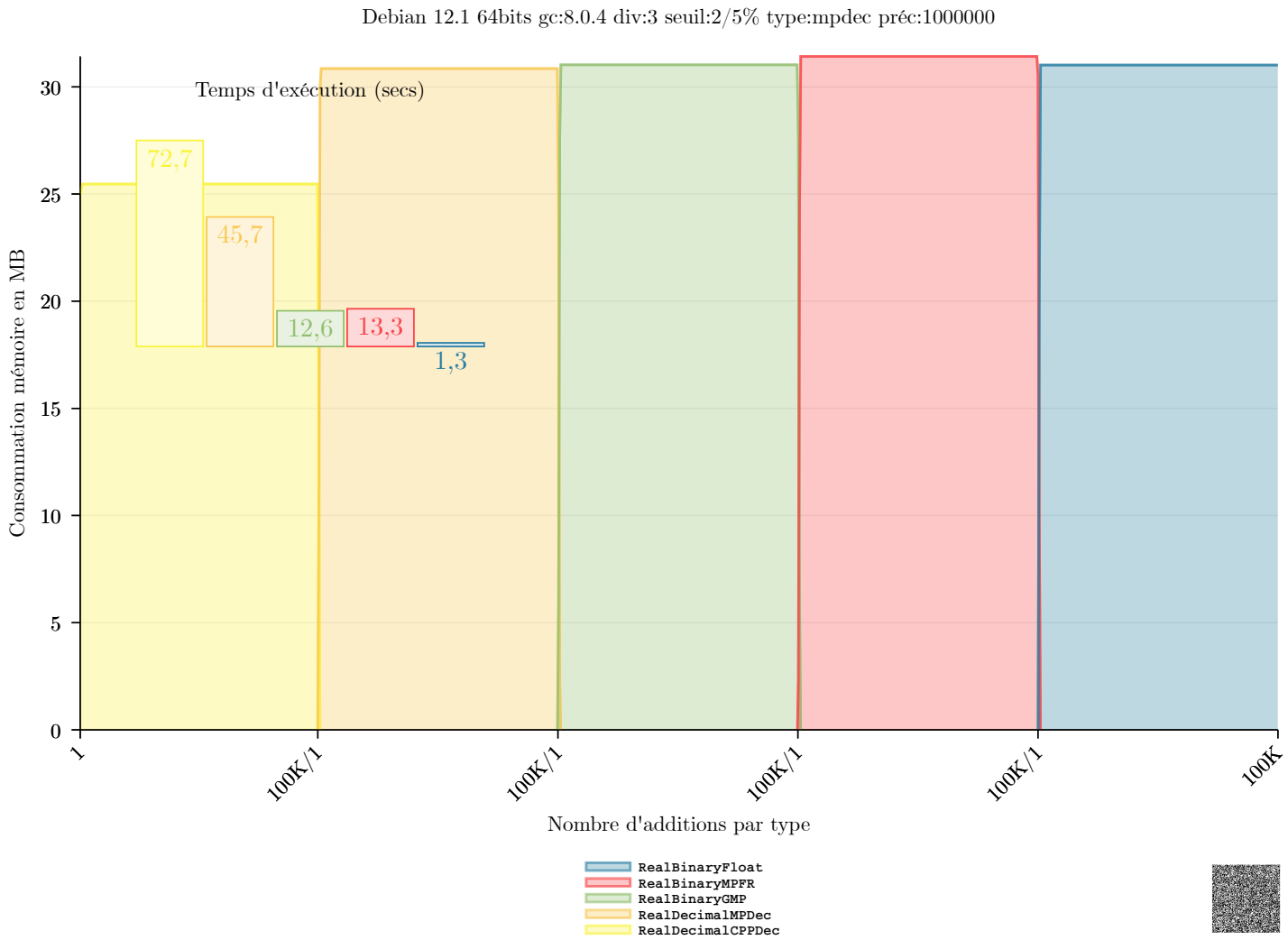


Figure 7.12 Résultats avec objets éphémères dans *Debian 12.1*[116] (*retour*).

[116] Généré sous *Debian 12.1* 64 bits (x86-64 40x3GHz/31GB).

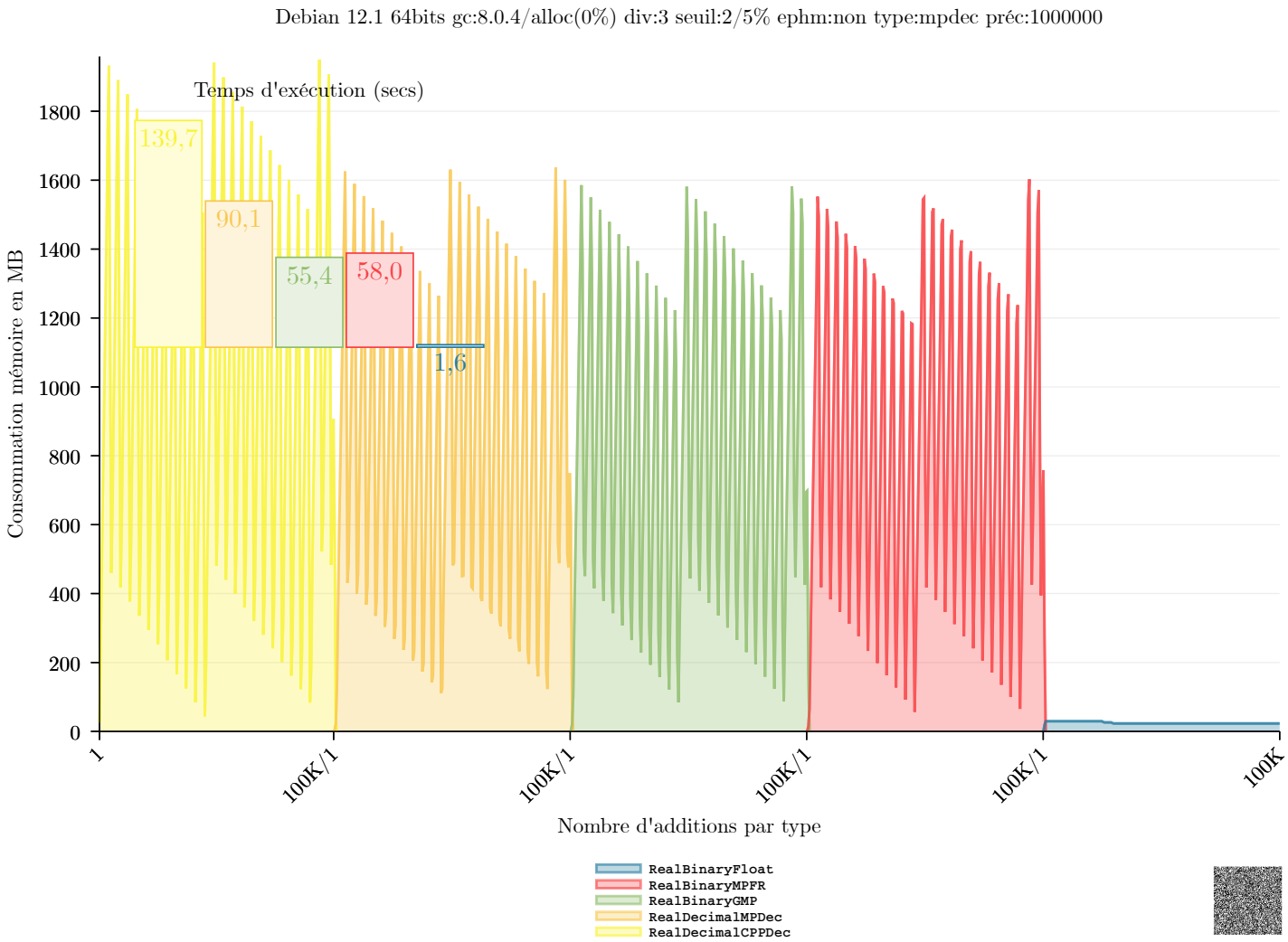


Figure 7.17 Résultats sans objets éphémères et allocateurs dynamiques 0% dans *Debian 12.1*[117] (*retour*).

[117] Généré sous *Debian 12.1* 64 bits (x86-64 40x3GHz/31GB).

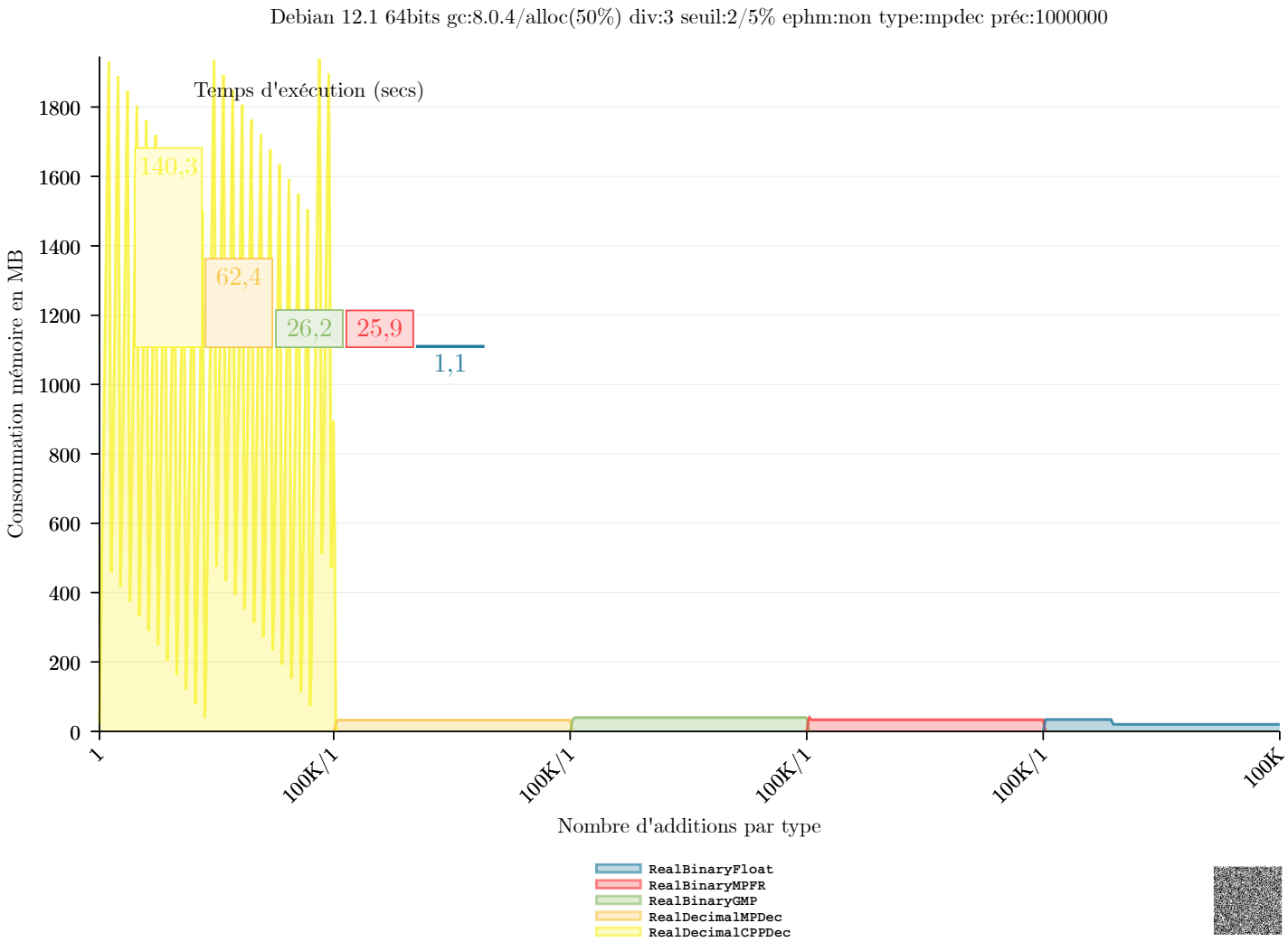


Figure 7.19 Résultats sans objets éphémères et allocateurs dynamiques 50% dans *Debian 12.1*[118] (*retour*).

[118] Généré sous *Debian 12.1* 64 bits (x86-64 40x3GHz/31GB).

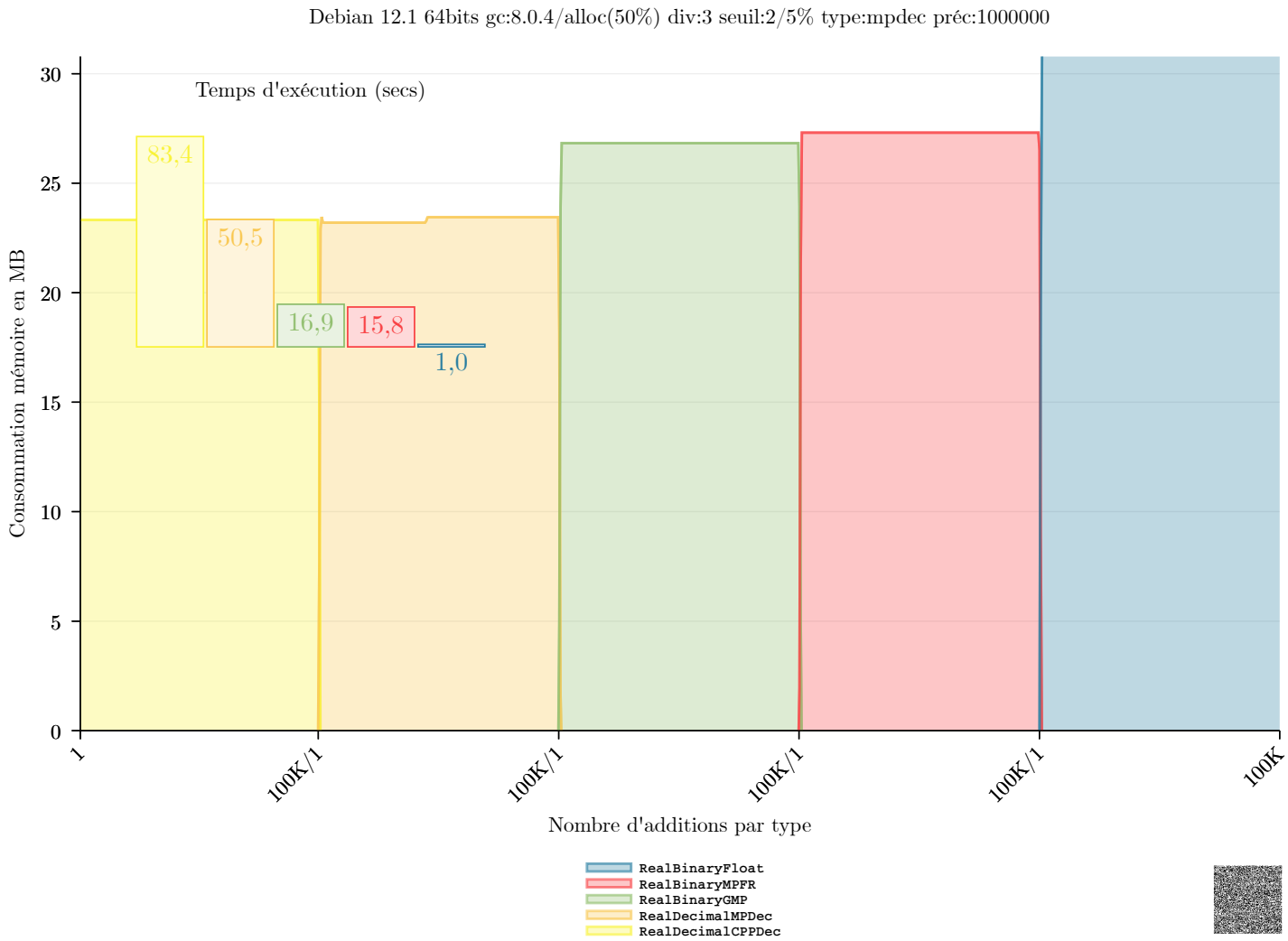


Figure 7.21 Résultats avec objets éphémères et allocateurs dynamiques 50% dans *Debian 12.1*[119] (*retour*).

[119]Généré sous *Debian 12.1* 64 bits (x86-64 40x3GHz/31GB).

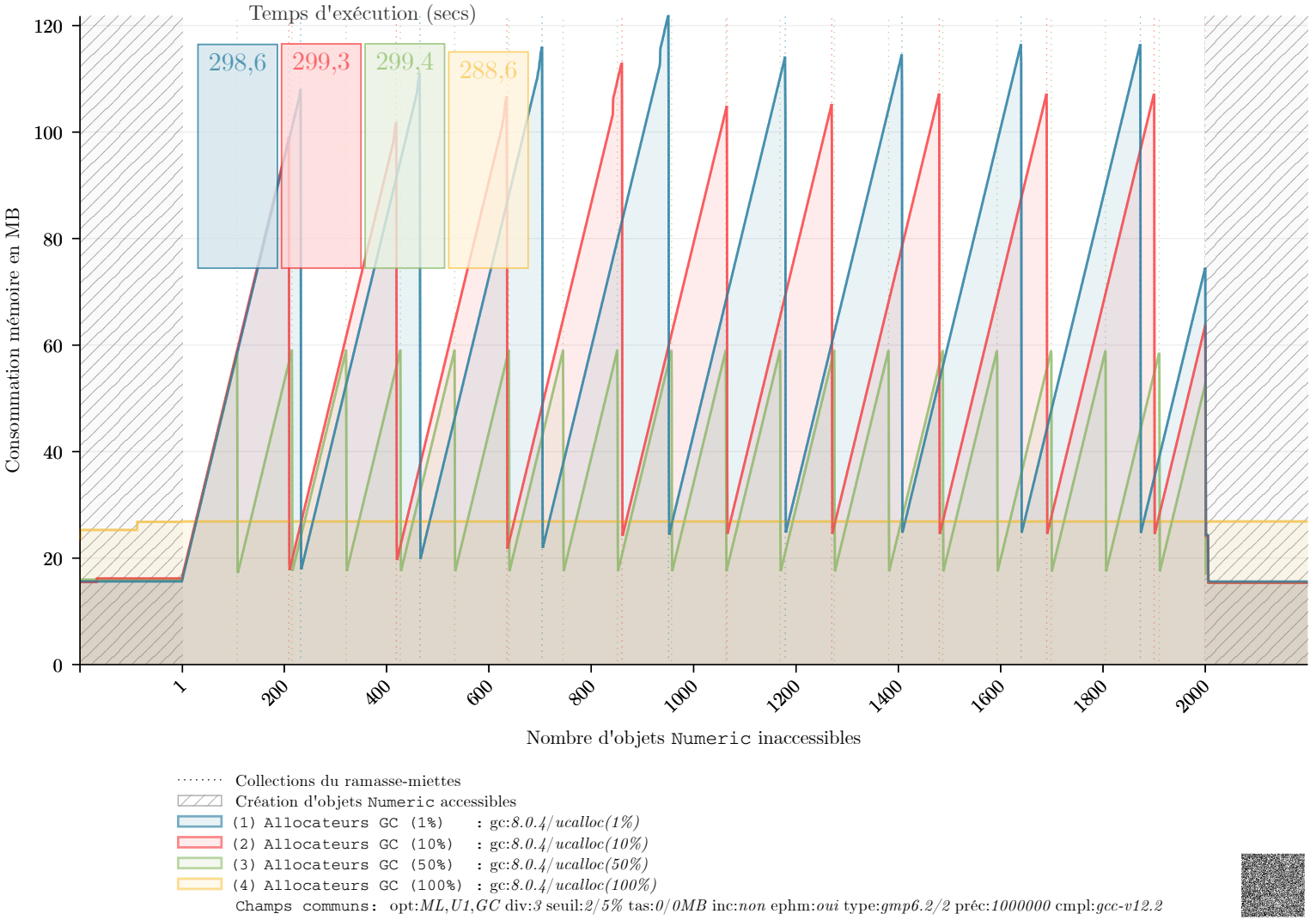


Figure 7.14 Différents pourcentages des allocations monitorées par le *GC* dans *Debian 12.1*[120] (*retour*).

[120] À noter que l'affichage des collectes du *GC* a été désactivé dans le cas de la courbe 4 pour alléger le graphe. Généré sous *Debian 12.1* 64 bits (x86-64 40x3GHz/31GB).

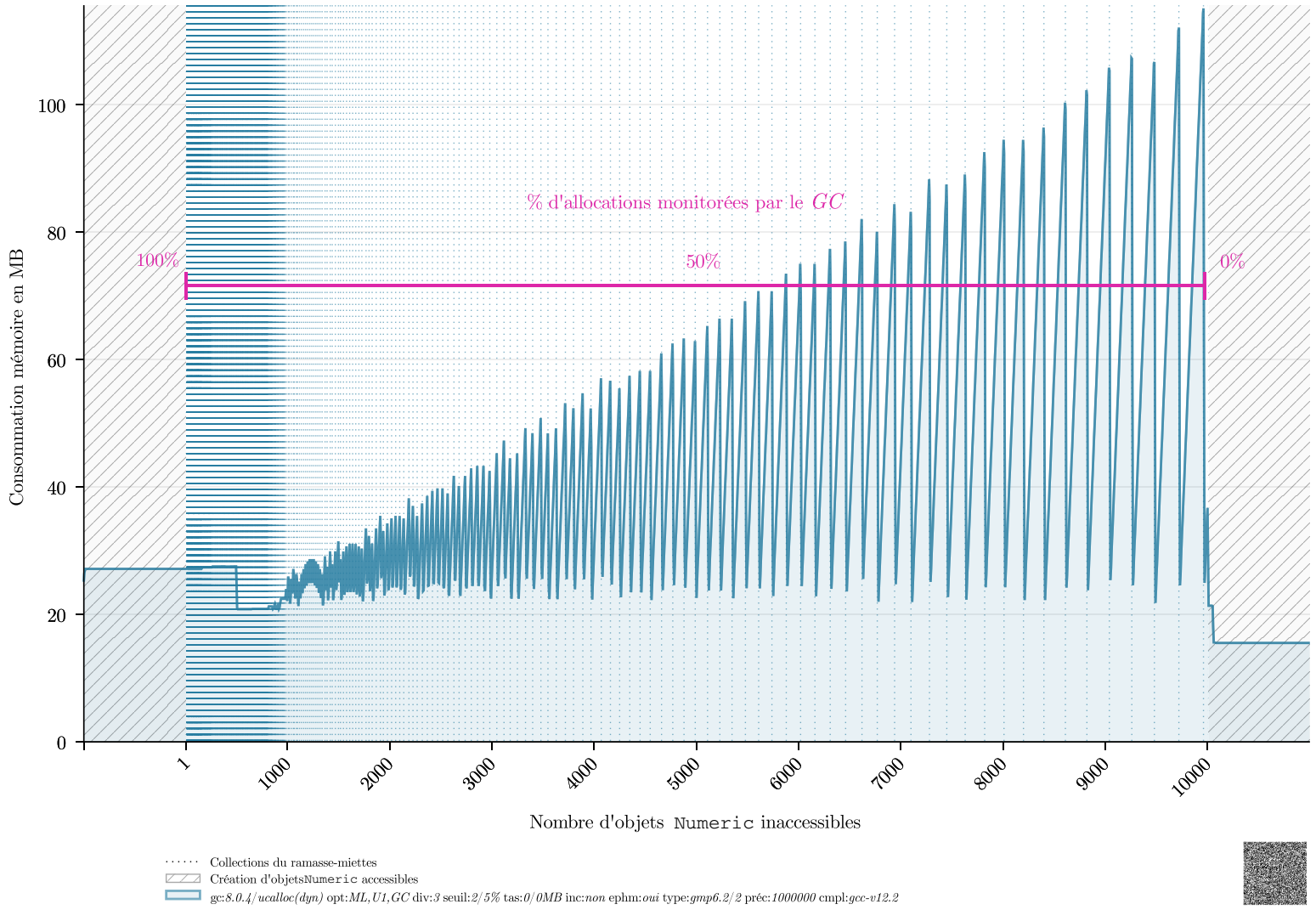


Figure 7.15 Pourcentage dynamique des allocations monitorées par le *GC* dans *Debian 12.1*[121] (*retour*).

[121] Généré sous *Debian 12.1* 64 bits (x86-64 40x3GHz/31GB).

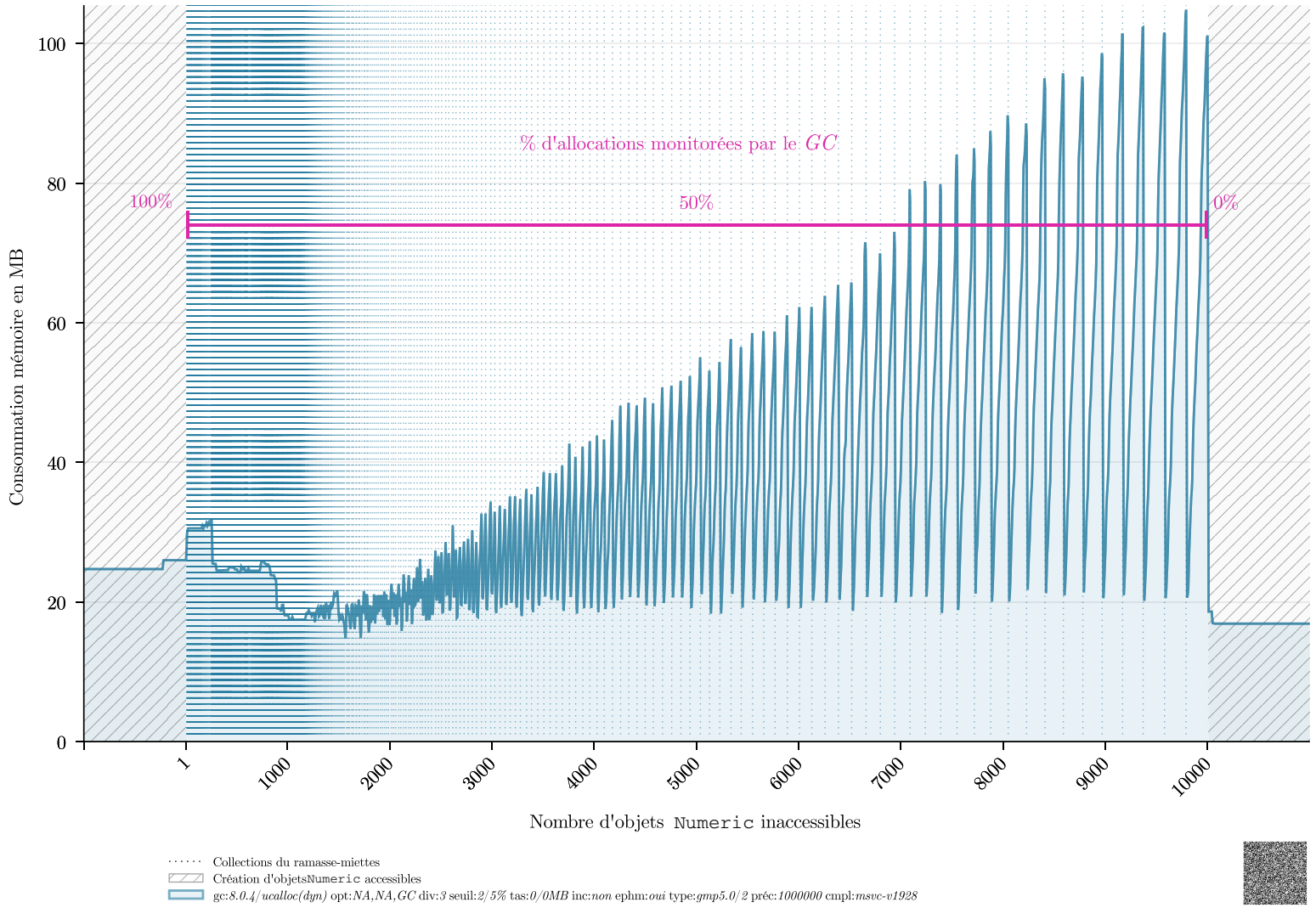


Figure 7.16 Pourcentage dynamique des allocations monitorées par le *GC* dans *Windows 7*[122] (*retour*).

[122]Généré sous *Microsoft Windows 7 Professional* 64 bits (x86-64 8x2.94GHz/12.0GB).

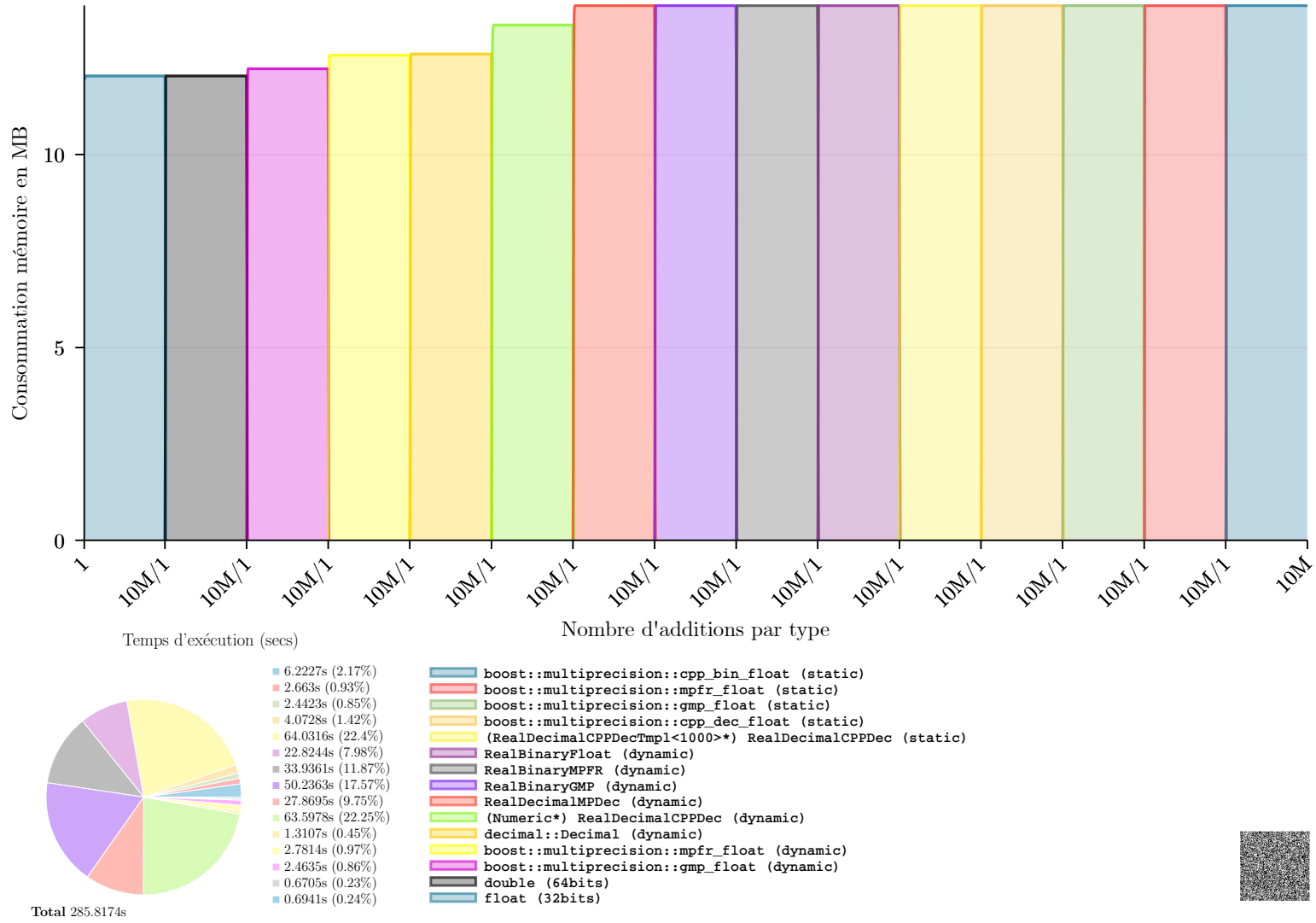


Figure B.4 Résultats du test des différentes bibliothèques et types `Numeric` (*retour*).

[122] Généré sous *Linux Ubuntu 16.04.7 LTS* 64 bits (x86-64 2x2.26GHz/8GB).

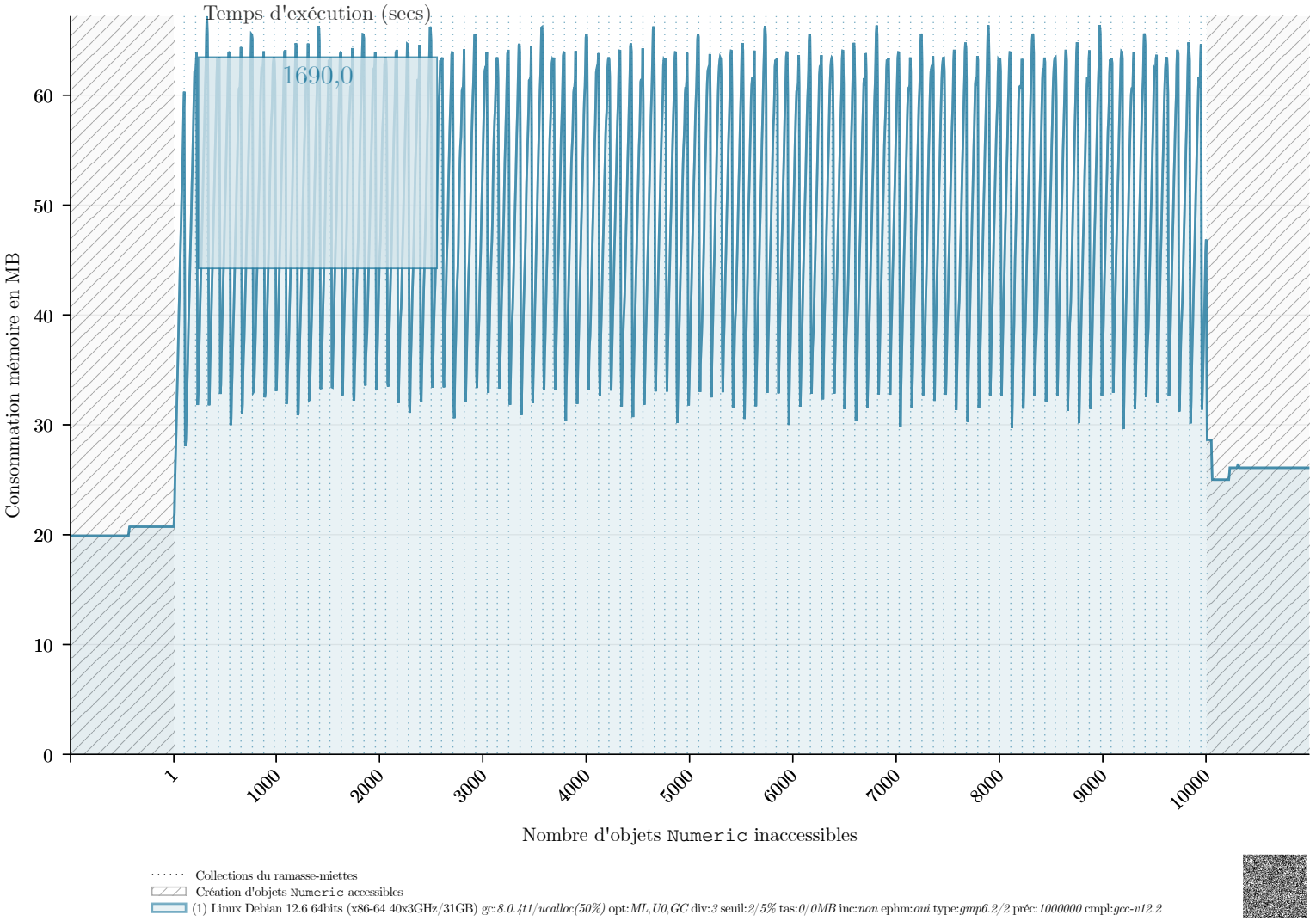


Figure B.1 Résultat du test de création d'objets inaccessibles dans *Linux*[123] (*retour*).

[123] Généré sous *Debian 12.1* 64 bits (x86-64 40x3GHz/31GB).

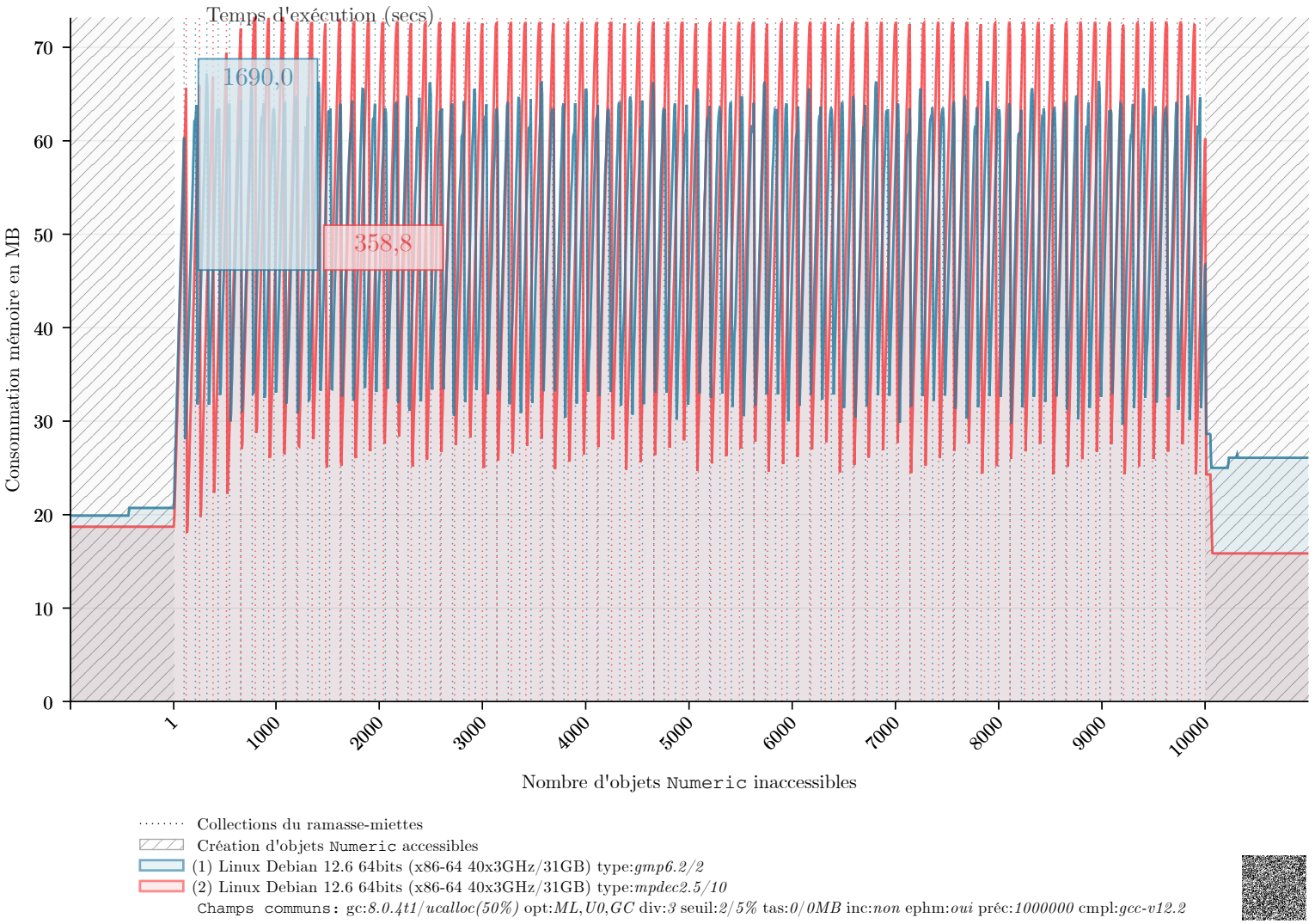


Figure B.2 Résultat d'une deuxième passe du test de création d'objets inaccessibles dans *Linux*[124] (*retour*).

[124]Généré sous *Debian 12.1* 64 bits (x86-64 40x3GHz/31GB).

GLOSSAIRE

Remarque: *La plupart des entrées de ce glossaire sont des extraits provenant des versions française ou anglaise de Wikipedia. Les extraits provenant de la version anglaise (indiqués par «Wikipedia (traduction)») ont été traduits par le service de traduction automatique DeepL puis ont été légèrement révisés.*

Algorithme arrêt et copie (***stop-and-copy algorithm***) « *Dans cet algorithme (algorithme de Cheney), le tas (heap) est divisé en deux moitiés égales, dont une seule est utilisée à tout moment. Le ramassage des miettes s'effectue en copiant les objets vivants d'un semi-espace (l'espace de départ) vers l'autre (l'espace d'arrivée), qui devient alors le nouveau tas. L'ensemble de l'ancien tas est alors jeté en une seule fois.* » Wikipedia (traduction)

Algorithme marquant et nettoyant (***mark-and-sweep algorithm***) « *Un ramasse-miettes utilisant un algorithme marquant et nettoyant conserve un bit ou deux avec chaque objet pour enregistrer s'il est blanc ou noir. L'ensemble gris est conservé dans une liste séparée ou en utilisant un autre bit. Lorsque l'arbre de référence est parcouru pendant un cycle de collecte (la phase de "marquage"), ces bits sont manipulés par le ramasse-miettes. Un dernier nettoyage des zones de mémoire libère ensuite les objets blancs.* » Wikipedia (traduction)

Algorithme marquant et non nettoyant (***mark-and-don't-sweep algorithm***) « *Un algorithme marquant et non nettoyant, comme l'algorithme marquant et nettoyant, conserve un bit avec chaque objet pour enregistrer s'il est blanc ou noir; l'ensemble gris est conservé dans une liste séparée ou à l'aide d'un autre bit. Il existe deux différences essentielles. Premièrement, les notions de blanc et de noir n'ont pas la même signification que pour l'algorithm marquant et nettoyant. [...] Deuxièmement, l'interprétation du bit noir/blanc peut changer. [...] Aucune phase de "balayage" n'est nécessaire.* » Wikipedia (traduction, extrait)

Analyse numérique (***numerical analysis***) « *L'analyse numérique est une discipline à l'interface des mathématiques et de l'informatique. Elle s'intéresse tant aux fondements qu'à la mise en pratique des méthodes permettant de résoudre, par des calculs purement numériques, des problèmes d'analyse mathématique.* » Wikipedia

Analyse syntaxique ascendante (***bottom-up parser***) « *En informatique, l'analyse syntaxique révèle la structure grammaticale d'un texte. C'est la première étape dans l'étude de son sens. À l'inverse de l'analyse descendante, l'analyse ascendante reconnaît d'abord les plus petites unités du texte (les*

unités lexicales) analysé avant d'en reconnaître la structure grammaticale en le confrontant à des règles de syntaxe. » Wikipedia

Analyse syntaxique descendante (***top-down parser***) « *En informatique, l'analyse syntaxique descendante est une stratégie d'analyse où l'on examine d'abord le niveau le plus élevé de l'arbre d'analyse et où l'on descend l'arbre d'analyse en utilisant les règles de réécriture d'une grammaire formelle. Les analyseurs LL sont un type d'analyseur qui utilise une stratégie d'analyse descendante.* » Wikipedia *(traduction)*

Classe abstraite (***abstract class***) « *En programmation orientée objet (POO), une classe abstraite est une classe si et seulement si elle n'est pas instanciable. Elle sert de base à d'autres classes dérivées (héritées).* » Wikipedia

Langage rationnel (***regular language***) « *En théorie des langages, les langages rationnels ou langages réguliers ou encore langages reconnaissables peuvent être décrits de plusieurs façons équivalentes :*

- *Ce sont les langages décrits par les expressions régulières ou rationnelles, d'où le nom de langages réguliers.*
- *Ce sont les langages obtenus, à partir des lettres et de l'ensemble vide, par les opérations rationnelles, à savoir l'union, le produit et l'étoile de Kleene, d'où le nom de langages rationnels.*
- *Ce sont les langages reconnus par des automates finis, d'où le nom de langages reconnaissables.*

Les langages rationnels ont de très nombreuses applications, à la fois théoriques et pratiques. Ils sont utilisés en informatique (par exemple en compilation), en linguistique (par exemple pour décrire la morphologie d'une langue), ils interviennent dans les traitements de texte, ou dans des commandes spécifiques comme grep du système Unix. » Wikipedia

Multithreading « *Un processeur est dit multithread s'il est capable d'exécuter efficacement plusieurs threads simultanément. Contrairement aux systèmes multiprocesseurs (tels les systèmes multi-cœur), les threads doivent partager les ressources d'un unique cœur : les unités de traitement, le cache processeur et le translation lookaside buffer ; certaines parties sont néanmoins dupliquées : chaque thread dispose de ses propres registres et de son propre pointeur d'instruction. Là où les systèmes multiprocesseurs incluent plusieurs unités de traitement complètes, le multithreading a pour but d'augmenter l'utilisation d'un seul cœur en tirant profit des propriétés des threads et du parallélisme au niveau des instructions. Comme les deux techniques sont complémentaires, elles sont parfois combinées dans des systèmes comprenant de multiples processeurs multithreads ou des processeurs avec de multiples cœurs multithreads.* » Wikipedia

Méthode/fonction virtuelle (***virtual function***) « *En programmation orientée objet, une fonction virtuelle est une fonction définie dans une classe (méthode) qui est destinée à être redéfinie dans les classes qui en héritent. Dans la plupart des langages, soit toutes les méthodes sont automatiquement virtuelles (Java, Swift...), soit le mot clé virtual est utilisé pour indiquer que la méthode d'une classe est virtuelle (C++, Delphi, Free Pascal...).* » Wikipedia

Nombre irrationnel (***irrational number***) « *Un nombre irrationnel est un nombre réel qui n'est pas rationnel, c'est-à-dire qu'il ne peut pas s'écrire sous la forme d'une fraction $\frac{a}{b}$, où a et b sont deux entiers relatifs (avec b non nul). Les nombres irrationnels peuvent être caractérisés de manière équivalente comme étant les nombres réels dont le développement décimal n'est pas périodique ou dont le développement en fraction continue est infini.* » Wikipedia

Nombre transcendant (***transcendental number***) « *En mathématiques, un nombre transcendant sur les rationnels est un nombre réel ou complexe qui n'est racine d'aucun polynôme non nul $a_0 + a_1X + a_2X^2 + \cdots + a_nX^n$, où n est un entier naturel et les coefficients a_i sont des rationnels non tous nuls, ou encore (en multipliant ces $n + 1$ rationnels par un dénominateur commun) qui n'est racine d'aucun polynôme non nul à coefficients entiers. Un nombre réel ou complexe est donc transcendant si et seulement s'il n'est pas algébrique. Comme tout nombre rationnel est algébrique, tout nombre transcendant est donc un nombre irrationnel. La réciproque est fausse : par exemple $\sqrt{2}$ est irrationnel mais n'est pas transcendant, puisqu'il est solution de l'équation polynomiale $x^2 - 2 = 0$.* » Wikipedia

Objets racine (***roots objects***) « *Les objets racine du ramasse-miettes (ou simplement, les racines (roots)) sont le point de départ à partir duquel nous commençons notre analyse des traces. Ces objets sont garantis accessibles, car ils sont généralement pointés par les variables globales ou locales de la pile.* » Writing a Mark-Sweep Garbage Collector - Dmitry Soshnikov (traduction)

Polymorphisme (***polymorphism***) « *En informatique et en théorie des types, le polymorphisme est le concept consistant à fournir une interface unique à des entités pouvant avoir différents types. Par exemple, des opérations telles que la multiplication peuvent ainsi être étendues à des scalaires aux vecteurs ou aux matrices, l'addition, des scalaires aux fonctions ou aux chaînes de caractères, etc.* » Wikipedia

Programmation impérative (***imperative programming***) « *En informatique, la programmation impérative est un paradigme de programmation qui décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme. Ce type de programmation est le plus répandu parmi*

l'ensemble des langages de programmation existants, et se différencie de la programmation déclarative (dont la programmation logique ou encore la programmation fonctionnelle sont des sous-ensembles). [...] La plupart des langages de haut niveau comporte cinq types d'instructions principales :

- *la séquence d'instructions*
- *l'affectation (assignation)*
- *l'instruction conditionnelle*
- *la boucle*
- *les branchements* » Wikipedia (extrait)

RAII (***R**esource **A**cquisition **I**s **I**nitialization*) « *En programmation orientée objet, resource acquisition is initialization, abrégé RAII et traduisible littéralement par l'acquisition d'une ressource est une initialisation, est une technique utilisée dans plusieurs langages orientés objet, comme `C++`, `D` et Ada. Cette technique, inventée par Bjarne Stroustrup1, permet de s'assurer, lors de l'acquisition d'une ressource, que celle-ci sera bien libérée en liant cette acquisition à la durée de vie d'un objet : la ressource est acquise durant l'initialisation de l'objet, il est alors possible de l'utiliser tant que la ressource est disponible et elle est libérée au moment de la destruction de l'objet, cette destruction étant garantie même en cas d'erreur. [...] La technique RAII aide à l'écriture de code plus résistant aux exceptions : pour libérer une ressource avant de permettre à l'exception de se propager, on peut écrire un destructeur approprié plutôt que de disséminer et multiplier les instructions de nettoyage entre les blocs de prise en compte des exceptions.* » Wikipedia (extrait)

Ramasse-miettes (***garbage collector***) « *Un ramasse-miettes, ou récupérateur de mémoire, ou glaneur de cellules (en anglais garbage collector, abrégé en GC), est un sous-système informatique de gestion automatique de la mémoire. Il est responsable du recyclage de la mémoire préalablement allouée puis inutilisée.* » Wikipedia

Ramasse-miettes conservateurs (***conservative collectors***) « *Les ramasse-miettes conservateurs supposent que tout motif binaire en mémoire peut être un pointeur si, interprété comme un pointeur, il pointe vers un objet alloué. Les ramasse-miettes conservateurs peuvent produire des faux positifs, lorsque la mémoire inutilisée n'est pas libérée en raison d'une identification incorrecte des pointeurs.* » Wikipedia (traduction)

Ramasse-miettes générationnels (***generational collectors***) « *Il a été observé empiriquement que dans de nombreux programmes, les objets les plus récemment créés sont aussi ceux qui ont le plus de chances de devenir rapidement inaccessibles. Un ramasses-miettes générationnel divise les objets en générations et,*

sur la plupart des cycles, ne placera que les objets d'un sous-ensemble de générations dans l'ensemble blanc initial (condamné). » Wikipedia (traduction)

Ramasse-miettes incrémentiels (***incremental collectors***) « *Les ramasse-miettes incrémentiels effectuent le cycle de collecte des déchets en phases distinctes, l'exécution du programme étant autorisée entre chaque phase (et parfois pendant certaines phases).* » Wikipedia (traduction)

Ramasse-miettes précis (***precise collectors***) « *Certains ramasse-miettes peuvent identifier correctement tous les pointeurs (références) d'un objet; ils sont appelés ramasse-miettes précis (ou exacts), le contraire étant un ramasse-miettes conservateur ou partiellement conservateur.* » Wikipedia (traduction)

Thread « *Un thread ou fil (traduction normalisés par ISO/CEI 2382-7:20001 (autres appellations connues : processus léger, fil d'exécution, fil d'instruction, processus allégé, filet d'exécution, exétron, tâche, voire unité d'exécution ou unité de traitement) est similaire à un processus car tous deux représentent l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Du point de vue de l'utilisateur, ces exécutions semblent se dérouler en parallèle. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les threads d'un même processus se partagent sa mémoire virtuelle. En revanche, tous les threads possèdent leur propre pile d'exécution.* » Wikipedia

Thread Local Storage (***TLS***) « *Le Thread Local Storage (TLS), ou mémoire locale de thread, est un type de mémoire spécifique et locale à un thread. Ce mécanisme est parfois requis parce que tous les threads d'un même processus partagent le même espace d'adressage. Donc, les données situées dans une variable statique ou globale sont exactement au même emplacement mémoire pour tous les threads, et correspondent donc à la même entité. [...] Cependant, il est parfois utile que deux threads puissent référencer la même variable "globale" tout en possédant chacun une copie distincte, donc à des adresses mémoire différentes. Ceci rend la variable "locale" au thread, tout en ayant une syntaxe d'utilisation identique à celle d'une variable globale. Un exemple trivial d'une telle variable est, par exemple, la variable* `errno` *du langage* `C`.* » Wikipedia (extrait)

Thread safety « *La thread safety d'un code (qu'on appelle alors "code thread-safe") est la propriété de celui-ci associée au fait qu'il est capable de fonctionner correctement lorsqu'il est exécuté simultanément au sein du même espace d'adressage par plusieurs threads.* » Wikipedia

BIBLIOGRAPHIE

Aakashattri111 (2023). Visitor Method Design Patterns in C++. Récupéré le 2024-02-19 de https://www.geeksforgeeks.org/visitor-method-design-patterns-in-c/#disadvantages-of-visitor-design-patterns-in-c

Backus, J. (1959). The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich *ACM-GAMM* Conference. Récupéré le 2023-04-19 de https://www.softwarepreservation.org/projects/ALGOL/paper/

Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., van Wijngaarden, A. et Woodger, M. (1963). Revised Report on the Algorithmic Language *ALGOL 60*. *The Computer Journal*, *5*(4), 349–367. http://dx.doi.org/10.1093/comjnl/5.4.349

Bailey, D. H. et Borwein, J. M. (2015). High-Precision Arithmetic in Mathematical Physics. *Mathematics*, *3*(2), 337–367. http://dx.doi.org/10.3390/math3020337

Bezanson, J., Edelman, A., Karpinski, S. et Shah, V. B. (2017). *Julia* : A Fresh Approach to Numerical Computing. *SIAM Review*, *59*(1), 65–98. http://dx.doi.org/10.1137/141000671

Boehm, H.-J. (2023). *FAQ* : A Garbage Collector for C and C++. Récupéré le 2023-07-01 de https://www.hboehm.info/gc/faq.html

Boehm, H.-J. et Demers, A. J. (1988). A Garbage Collector for C and C++. Récupéré le 2021-06-11 de https://www.hboehm.info/gc/

Bond, E., Auslander, M., Grisoff, S., Kenney, R., Myszewski, M., Sammet, J., Tobey, R. et Zilles, S. (1964). *FORMAC* an Experimental Formula Manipulation Compiler. Dans *Proceedings of the 1964 19th ACM national conference*, ACM '64, 112.101–112.1019. Association for Computing Machinery. http://dx.doi.org/10.1145/800257.808916

Brilliant.org (2017). Fibonacci Heap | Brilliant Math & Science Wiki. Récupéré le 2023-12-18 de https://brilliant.org/wiki/fibonacci-heap/

CalTech (2007). C++ Operator Overloading Guidelines. *California Institute of Technology*. Récupéré le 2023-12-08 de http://courses.cms.caltech.edu/cs11/material/cpp/donnie/cpp-ops.html

Char, B., Geddes, K. et Gonnet, G. (1983). The *Maple* Symbolic Computation System. *ACM SIGSAM Bulletin*, *17*(3-4), 31–42. http://dx.doi.org/10.1145/1089338.1089344

Cheney, C. J. (1970). A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, *13*(11), 677–678. http://dx.doi.org/10.1145/362790.362798

Cohn, M. (2022). Agile Estimation : Why the Fibonacci Sequence Works. Récupéré le 2024-01-02 de https://www.mountaingoatsoftware.com/blog/why-the-fibonacci-sequence-works-well-for-estimating

Corbett, R. et al. (2021). *GNU Bison* 3.8.1 Documentation. Récupéré le 2023-06-17 de https://www.gnu.org/software/bison/manual/html_node/index.html

Cowlishaw, M. (2009). General Decimal Arithmetic Specification. Récupéré le 2022-10-15 de http://speleotrove.com/decimal/

Denny, J. E. et Malloy, B. A. (2010). The *IELR*(1) Algorithm for Generating Minimal *LR*(1) Parser Tables for Non-*LR*(1) Grammars with Conflict Resolution. *Science of Computer Programming*, *75*(11), 943–979. http://dx.doi.org/10.1016/j.scico.2009.08.001

DeRemer, F. (1969). *Practical Translators for LR(k) Languages*. (Thèse de doctorat). Massachusetts Institute of Technology. http://dx.doi.org/10.5555/888578

IEEE (1985). *IEEE 754-1985* Standard for Binary Floating-Point Arithmetic. http://dx.doi.org/10.1109/ieeestd.1985.82928

IEEE (1987). *IEEE 854-1987* Standard for Radix-Independent Floating-Point Arithmetic. *ANSI/IEEE Std 854-1987*, 1–19. http://dx.doi.org/10.1109/IEEESTD.1987.81037

IEEE (2008). *IEEE 754-2008* Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, 1–70. http://dx.doi.org/10.1109/IEEESTD.2008.4610935

Engelman, C. (1971). The Legacy of *MATHLAB 68*. Dans *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, *SYMSAC* '71, 29–41. Association for Computing Machinery. http://dx.doi.org/10.1145/800204.806265

Fousse, L., Hanrot, G., Lefèvre, V., Pelissier, P. et Zimmermann, P. (2007). *MPFR* : A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Transactions on Mathematical Software*, *33*(2). http://dx.doi.org/10.1145/1236463

Gagnon, E. (1998). *SableCC, An Object-Oriented Compiler Framework*. (Mémoire de maîtrise). McGill University. Récupéré le 2023-09-03 de https://escholarship.mcgill.ca/concern/theses/0c483m29g

Gamma, E., Vlissides, J., Helm, R. et Johnson, R. (1995). *Design Patterns : Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional.

Goldberg, D. (1991). What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys (CSUR)*, *23*(1), 5–48. http://dx.doi.org/10.1145/103162.103163

Granlund, T. et al. (2020). *GNU MP* 6.2.1 Documentation. Récupéré le 2023-08-31 de https://gmplib.org/manual/

Hearn, A. C. (1968). *REDUCE* : A User-Oriented Interactive System for Algebraic Simplification. Dans *Symposium on Interactive Systems for Experimental Applied Mathematics : Proceedings of the Association for Computing Machinery Inc. Symposium*, 79–90. Association for Computing Machinery. http://dx.doi.org/10.1145/2402536.2402544

Hearn, A. C. (2008). *REDUCE* : The First Forty Years. Récupéré le 2023-07-23 de http://www.reduce-algebra.com/about.php

Higham, D. J. et Higham, N. J. (2017). *MATLAB Guide* (3ième éd.). Society for Industrial and Applied Mathematics.

Hungerecker, S. et Schmitz, M. (2015). Concepts of Programming Languages. CoPL'15. Récupéré le 2023-09-03 de https://www.isp.uni-luebeck.de/

Jenks, R. D. et Sutor, R. (1992). *AXIOM : The Scientific Computation System*. Springer-Verlag.

Kahan, W. (1996). Lecture Notes on the Status of *IEEE* Standard 754 for Binary Floating-Point Arithmetic. Récupéré le 2023-07-22 de https://people.eecs.berkeley.edu/~wkahan/ieee754status/

Kleene, S. C. (1951). Representation of Events in Nerve Nets and Finite Automata. http://dx.doi.org/10.1515/9781400882618-002

Knuth, D. E. (1965). On the Translation of Languages from Left to Right. *Information and Control*, *8*(6), 607–639. http://dx.doi.org/10.1016/S0019-9958(65)90426-2

Koepf, W. (1999). Numeric Versus Symbolic Computation. *International Journal of Computer Algebra in Mathematics Education*, *4*, 179–203. http://dx.doi.org/10.1007/978-1-4613-0297-1_14

Kormanyos, C. (2011). Algorithm 910 : A Portable C++ Multiple-Precision System for Special-Function Calculations. *ACM Transactions on Mathematical Software*, *37*(4), 45 :1–45 :27. http://dx.doi.org/10.1145/1916461.1916469

Krah, S. (2008). *Mpdecimal* Library Documentation. Récupéré le 2023-02-26 de https://www.bytereef.org/mpdecimal/doc/libmpdec/index.html

Krah, S. (2021). *Mpdecimal* C++ *API* Documentation. Récupéré le 2023-02-26 de https://www.bytereef.org/mpdecimal/doc/libmpdec++/index.html

Lai, H. (2019). What Causes Ruby Memory Bloat ? Récupéré le 2023-12-10 de https://www.joyfulbikeshedding.com/blog/2019-03-14-what-causes-ruby-memory-bloat.html

Li, Y., Geng, F. et Cui, M. (2007). The Analytical Solution of a System of Nonlinear Differential Equations. *International Journal of Mathematical Analysis (Ruse)*, *1*, 451–462. Récupéré le 2024-04-11 de https://www.m-hikari.com/ijma/ijma-password-2007/ijma-password9-12-2007/

Linkletter, D. (2019). The *PEMDAS* Paradox. Récupéré le 2023-08-29 de https://plus.maths.org/content/pemdas-paradox

Maclennan, B. (1982). Values and Objects in Programming Languages. *ACM SIGPLAN Notices*, *17*(12), 70–79. http://dx.doi.org/10.1145/988164.988172

Maddock, J. et Kormanyos, C. (2022). *Boost.Multiprecision* 1.79 Documentation. Récupéré le 2023-08-15 de https://www.boost.org/doc/libs/1_79_0/libs/multiprecision/doc/html/index.html

Malaquias, J. R. et Lopes, C. R. (2007). Implementing a Computer Algebra System in Haskell. *Applied Mathematics and Computation*, *192*(1), 120–134. http://dx.doi.org/10.1016/j.amc.2007.02.126

Matula, D. W. (1968). In-and-Out Conversions. *Communications of the ACM*, *11*(1), 47–50. http://dx.doi.org/10.1145/362851.362887

McCarthy, J. (1960). Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, *3*(4), 184–195. http://dx.doi.org/10.1145/367177.367199

McCulloch, W. S. et Pitts, W. (1943). A Logical Calculus of the Ideas Immanent in Nervous Activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133. http://dx.doi.org/10.1007/BF02478259

Mealy, G. H. (1955). A Method for Synthesizing Sequential Circuits. *Journal of Symbolic Logic*, *22*(3), 334–335. http://dx.doi.org/10.2307/2963669

Moore, E. F. (1956). Gedanken-Experiments on Sequential Machines. *Automata studies*, *34*, 129–153. http://dx.doi.org/10.1515/9781400882618-006

Moore, G. E. (1965). Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, *11*(3), 33–35. http://dx.doi.org/10.1109/N-SSC.2006.4785860

Moses, J. (2012). Macsyma : A Personal History. *Journal of Symbolic Computation*, *47*(2), 123–130. http://dx.doi.org/10.1016/j.jsc.2010.08.018

Norvell, T. (2002). A Short Introduction to Regular Expressions and Context Free Grammars. Récupéré le 2023-04-01 de https://www.engr.mun.ca/~theo/Courses/

Parr, T. (1996). *Language Translation Using PCCTS and C++ : a Reference Guide* (repr. éd.). Automata Publ. Récupéré le 2023-03-31 de https://www.researchgate.net/publication/2505121_Language_Translation_Using_PCCTS_and_C_A_Reference_Guide

Parr, T. (2013). *The Definitive ANTLR 4 Reference* (2ième éd.). Pragmatic Bookshelf.

Parr, T. (2020). *ANTLR 4* Frequently-Asked Questions (*FAQ*). Récupéré le 2023-06-01 de https://github.com/antlr/antlr4/blob/0eb38a0/doc/faq/index.md

Parr, T. (2021). *ANTLR 4* Documentation. Récupéré le 2023-07-28 de https://github.com/antlr/antlr4/blob/0eb38a0/doc/index.md

Parr, T. et Fisher, K. (2011). *LL(*)* : The Foundation of the *ANTLR* Parser Generator. *ACM SIGPLAN Notices*, *46*(6), 425–436. http://dx.doi.org/10.1145/1993316.1993548

Parr, T., Harwell, S. et Fisher, K. (2014). Adaptive *LL*(*) Parsing : The Power of Dynamic Analysis. *ACM SIGPLAN Notices*, *49*(10), 579–598. http://dx.doi.org/10.1145/2714064.2660202

playX (2020). Conservative vs Precise GC : What to choose ? Récupéré le 2024-04-19 de https://medium.com/@gtashnik11/conservative-vs-precise-gc-what-to-choose-318d069d994c

Rabin, M. et Scott, D. (1959). Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*, *3*, 114–125. http://dx.doi.org/10.1147/rd.32.0114

Roche, X. (2022). When Allocators are Hoarding your Precious Memory. Récupéré le 2023-10-07 de https://algolia.com/blog/engineering/when-allocators-are-hoarding-your-precious-memory/

Rosenkrantz, D. J. et Stearns, R. E. (1970). Properties of Deterministic Top-Down Grammars. *Information and Control*, *17*(3), 226–256. http://dx.doi.org/10.1016/S0019-9958(70)90446-8

Schiessl, C. (2014). Multithreading in the *MRI Ruby* Interpreter. Récupéré le 2023-12-06 de https://web.archive.org/web/20231206071507/https://bugfactory.io/blog/multithreading-in-the-mri-ruby-interpreter/

Stalla, A. (2020). Migrating from *ANTLR2* to *ANTLR4*. Récupéré le 2023-03-24 de https://tomassetti.me/migrating-from-antlr2-to-antlr4/

Stearns, R. E. et Lewis, P. M. (1969). Property Grammars and Table Machines. *Information and Control*, *14*(6), 524–549. http://dx.doi.org/10.1016/S0019-9958(69)90312-X

Stover, G. M. (2003). The *Boehm* Collector for C and C++. Récupéré le 2021-04-20 de https://www.drdobbs.com/the-boehm-collector-for-c-and-c/184401632

Stroustrup, B. (1994). *The Design and Evolution of C++* (1st éd.). Addison-Wesley Professional.

Stroustrup, B. (2013). *The C++ Programming Language* (4th éd.). Addison-Wesley Professional.

Stroustrup, B., Pirkelbauer, P. et Solodkyy, Y. (2007). Open Multi-Methods for C++. 6th International Conference on Generative Programming and Component Engineering. http://dx.doi.org/10.1145/1289971.1289993

Stroustrup, B. et Sutter, H. (2024). C++ Core Guidelines. Récupéré le 2024-06-08 de https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.html

Terry, P. D. et Terry, P. (2005). *Compiling with C# and Java*. Pearson/Addison-Wesley.

ThePythonWiki (2020). The *Python* Wiki : GlobalInterpreterLock. Récupéré le 2024-03-30 de https://wiki.python.org/moin/GlobalInterpreterLock

Tomassetti, G. (2017). A Guide to Parsing : Algorithms and Terminology. Récupéré le 2023-07-03 de https://tomassetti.me/guide-parsing-algorithms-terminology/

Tomassetti, G. (2020). Why You Should Not Use (*F*)*Lex*, *Yacc* and *Bison*. Récupéré le 2023-06-07 de https://tomassetti.me/why-you-should-not-use-flex-yacc-and-bison/

Tomita, M. (1984). *LR* Parsers for Natural Languages. Dans *10th International Conference on Computational Linguistics and 22nd Annual Meeting of the Association for Computational Linguistics*, 354–357. Association for Computational Linguistics. http://dx.doi.org/10.3115/980491.980564

Vandevoorde, D., Josuttis, N. M. et Gregor, D. (2017). *C++ Templates : The Complete Guide (2nd Edition)*. Addison-Wesley Professional.

Vick, P. (2009). *LL* vs. *LR* vs. *GLR*. Récupéré le 2023-03-29 de https://www.panopticoncentral.net/2009/03/13/ll-vs-lr-vs-glr/

vikas2gqb5 (2024). How to Create Custom Memory Allocator in C++ ? Récupéré le 2024-07-19 de https://www.geeksforgeeks.org/how-to-create-custom-memory-allocator-in-cpp/

Weinzierl, S. (2002). Computer Algebra in Particle Physics. http://dx.doi.org/10.48550/arXiv.hep-ph/0209234

Wirth, N. (1977). What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions ? *Communications of the ACM*, *20*(11), 822–823. http://dx.doi.org/10.1145/359863.359883

Wolfram, S. (1985). Symbolic Mathematical Computation. *Communications of the ACM*, *28*(4), 390–394. http://dx.doi.org/10.1145/3341.3347

Wolfram, S. (1988). *Mathematica : A System for Doing Mathematics by Computer*. Addison-Wesley Longman Publishing Co., Inc.

Zimmermann, P. (2000). A Proof of *GMP* Fast Division and Square Root Implementations. Récupéré le 2021-07-21 de https://homepages.loria.fr/PZimmermann/papers/

Zorn, B. (1990). Comparing Mark-and-Sweep and Stop-and-Copy Garbage Collection. Dans *Proceedings of the 1990 ACM conference on LISP and functional programming*, *LFP* '90, 87–98. Association for Computing Machinery. http://dx.doi.org/10.1145/91556.91597