iloMath

Guide et documentation

Version 0,95, 1er août 2017

Thomas Nguyen

PttNguyen.net

Table des matières

1 Généralités **5**

1.1 Avant-propos . . . 5
1.2 Fonctionnalités . . . 5
1.3 Histoire et auteur . . . 6
1.3.1 Système de numérotation des versions . . . 6
1.4 Licence . . . 6
1.5 Compilation et utilisation . . . 7
1.6 Informations supplémentaires à propos de la documentation . . . 8

2 Nombres **9**

2.1 Généralités . . . 9
2.1.1 Format d'affichage . . . 9
2.2 Types de nombres . . . 9
2.3 Caractéristiques communes . . . 9
2.3.1 Syntaxe, création, et affichage . . . 10
2.3.2 Opérateurs arithmétiques . . . 12
2.3.3 Opérateurs de comparaison . . . 13
2.4 Nombres naturels (NNumber) . . . 13
2.4.1 Description . . . 13
2.4.2 Documentation de la classe . . . 13
2.4.3 Exemple . . . 17
2.5 Nombres naturels légers (NNumberL) . . . 17
2.5.1 Description . . . 17
2.5.2 Documentation de la classe . . . 17
2.5.3 Exemple . . . 20
2.6 Nombres relatifs (Z) . . . 20
2.6.1 Description . . . 20
2.6.2 Documentation du patron de classes . . . 21
2.6.3 Exemple . . . 24
2.7 Nombres rationnels (QNumber) . . . 24
2.7.1 Description . . . 24
2.7.2 Documentation du patron de classes . . . 25
2.7.3 Exemple . . . 27
2.8 Nombres réels légers (RNumberL) . . . 28
2.8.1 Description . . . 28

2.8.2 Documentation de la classe 28
2.8.3 Exemple 30
2.9 Nombres complexes (C) 30
2.9.1 Description 30
2.9.2 Documentation du patron de classes 31
2.9.3 Exemple 33

3 Fonctions **34**
3.1 Généralités 34
3.2 Liste des fonctions mathématiques de la bibliothèque 34
3.2.1 Fonctions pour NNumber(L) 34
3.2.2 Fonctions pour Z 35
3.2.3 Fonctions pour QNumber 35
3.2.4 Fonctions pour RNumberL 37
3.2.5 Fonctions pour C 37
3.2.6 Exemples 38
3.3 Documentation du patron de classes Function 38
3.3.1 Description 38
3.3.2 Méthodes et fonctions associées 39
3.3.3 Exemples 40

4 Tenseurs, intervalles, représentation graphique **43**
4.1 Tenseurs (Tensor) 43
4.1.1 Description 43
4.1.2 Documentation du patron de classes 43
4.1.3 Exemples 45
4.2 Intervalles (Interval) 48
4.2.1 Description 48
4.2.2 Documentation du patron de classes 48
4.2.3 Exemples 49
4.3 Outils pour la représentation graphique 50
4.3.1 Boite de vue (ViewBox) 50
4.3.2 Données de graphique (DataPlot) 53
4.3.3 Données de graphique complexes (CDataPlot) 57

5 Autres fonctionnalités **60**
5.1 Couleurs 60
5.1.1 Couleurs RVBA avec entiers (RgbaI) 60
5.1.2 Couleurs RVBA avec flottants (RgbaF) 61
5.1.3 Couleurs TSLA avec flottants (Hsl) 62
5.1.4 Conversions entre couleurs de différents types 63

6 Divers **65**
6.1 Projets d'amélioration 65
6.2 Fin, contribuer 65

1. Généralités

1.1 Avant-propos

Le but de cet ouvrage est de proposer à l'utilisateur d'iloMath la possibilité de prendre connaissance des principales fonctionnalités de cette bibliothèque, et de fournir sa documentation.

Ce document peut se trouver sur le site officiel d'iloMath :

`https://iloMath.net/`

Vous pouvez me contacter pout toute question ou pour signaler des bogues par courriel via :

`dev@iloMath.net`

Merci pour votre intérêt pour iloMath.

1.2 Fonctionnalités

iloMath est une bibliothèque C++ proposant les fonctionnalités suivantes :

- Gestion de nombres naturels, relatifs, rationnels, et complexes rationnels arbitrairement grands et de manière exacte, tout en proposant des versions allégées de ces nombres pour la rapidité (en fait, des adaptations de `uint64_t` et `double` leur permettant d'être manipulées avec les nombres arbitrairement grands et exacts) ;
- Gestion des fonctions élémentaires réelles et complexes courantes : somme, différence, produit, divisions et composition de monômes, polynômes, exponentielle, logarithme, sinus, cosinus, tangente, arcsinus, arccosinus, arctangente, signe, argument, module, gamma/factorielle :
  - Gestion de plusieurs variables ;
  - Évaluation des fonctions et outils pour la représentation graphique des fonctions réelles ;
  - Outil pour la représentation graphique des fonctions complexes ;
  - Pour les nombres arbitrairement grands et exacts, les évaluations de fonctions autres que les polynômes ne sont pas encore possible ou mal supportés. Il est prévu d'améliorer et compléter ces implémentations dans des versions futures de cette bibliothèque.
- La gestion élémentaire de tenseurs d'ordre quelconque ayant comme composantes ces nombres ;
- Analyse syntaxique interne permettant d'initialiser une fonction ou un nombre de façon transparente à l'aide d'une chaine de caractères, en respectant des règles de syntaxe courantes et intuitives ;
- Gestion des bases entières de 2 à 65535 pour tous ces objets.

Il reste néanmoins avant tout un projet *personnel*. Ainsi, si vous écrivez du code de production/professionnel, il est préférable d'utiliser une bibliothèque plus connue et puissante,

par exemple GMP. Pour des projets personnels ou à petite ampleur, iloMath devrait cependant très bien convenir.

En particulier, la compatibilité ascendante ne sera pas assurée pour le moment.

Il aura néanmoins un but éducatif ; ses implémentations seront exposées et expliquées sur le site officiel, ce qui peut donner des exemples en C++ pour les débutants.

1.3 Histoire et auteur

L'auteur est Thomas Nguyen, un étudiant en mathématiques et sciences informatiques à l'Université de Genève.

Le développement de cette bibliothèque a commencé quelque part en 2014, et était en fait un projet de travail de maturité permettant simplement de mettre en pratique ses connaissances en mathématiques et en C++.

Ce projet fut donc ensuite repris en-dehors de ce cadre, simplement comme un passe-temps, et n'a pas d'enjeu particulier (si ce n'est qu'il pourra me servir pour d'autres projets personnels futurs). Il est mis à disposition de tous si ce projet peut être utile.

À la fin du travail de maturité, iloMath était en version $0,9\beta3$. Le code a depuis été bien remanié et d'autres fonctionnalités intéressantes listées plus haut ont été implémentées. La version actuelle est $0,95$ (v).

iloMath vient de « Math » et « -ilo », un suffixe en espéranto signifiant seul « outil » et permettant sinon de désigner l'outil correspondant à un verbe : kalkuli (calculer) $\rightarrow$ kalkulilo (calculatrice, « outil pour calculer »).

1.3.1 Système de numérotation des versions

Dans mes projets, les numéros de version peuvent être vus comme un nombre qui représente grossièrement le degré de perfection : plus ce dernier est proche de $1 = 100\%$, plus il est proche de la perfection. Un logiciel en version 1 est tout simplement un programme parfait, 100% terminé, qui ne sera *plus jamais* mis à jour.

Le système de numérotation donne la possibilité de convertir facilement ce numéro vers un système plus familier (1.0 / 1.1 / 2.0 / ...) : par exemple, $0,9$ correspond à une version 1.0, $0,91$ à une version 1.1, $0,95$ à une version 1.5, $0,99$ à une version 2.0, $0,999912$ à une version 4.1.2, etc.

Le nombre de 9 est le numéro de version principal, et les autres nombres composent éventuellement les sous-versions.

1.4 Licence

Pour le logiciel iloMath, auteur : Thomas Nguyen (PttNguyen.net). Version officielle du 1er août 2017.

Veuillez lire attentivement les termes et conditions suivants avant d'utiliser le logiciel. Son utilisation implique l'acceptation de ce présent contrat de licence. Si vous n'acceptez pas cette CLUF, n'utilisez pas le logiciel.

- Vous êtes autorisé à utiliser ce logiciel dans un contexte personnel, individuel, éducatif, et non commercial ;
- Vous êtes autorisé à distribuer des copies de ce programme ou de sa source, même altérées, selon les conditions suivantes :
  - L'auteur original (Thomas Nguyen) et un lien vers iloMath.net doivent être clairement

cités, mais pas d'une manière qui suggère qu'il vous soutient ou soutient la manière dont vous avez modifié le logiciel ;
  — La licence doit être la même que celle-ci et ne doit pas contenir de restrictions supplémentaires. En particulier, l'utilisation commerciale est interdite ; vous ne pouvez pas entre autres faire payer des copies de ce logiciel (altérées ou non) ou des logiciels se basant sur ces copies, demander des dons, ou l'intégrer dans un logiciel diffusant de la publicité ;
  — Les altérations doivent être mentionnées.
— Des exceptions peuvent être accordées par écrit par l'auteur contre certaines conditions ;
— Ce logiciel est fourni tel quel, sans la moindre garantie. L'auteur ne pourra être tenu pour responsable de tout dommage direct, indirect, secondaire ou accessoire (par exemple, pertes financières, perte de données, etc.), découlant de l'utilisation du logiciel ou de l'impossibilité d'utiliser celui-ci ;
— Cette licence peut être modifiée à tout moment par l'auteur. Seule sa version officielle la plus récente fait foi.

(c) Thomas Nguyen (PttNguyen.net), tous droits réservés.

1.5 Compilation et utilisation

La compilation est normalement facilement réalisable sous la plupart des Linux, et en voici la démarche.

Premièrement, iloMath utilise g++ comme compilateur, ainsi que make et cmake. Aucun autre compilateur n'a été testé. On peut en général les installer avec un terminal à l'aide de :

```
apt-get install cmake make g++
```

Ensuite, il faut aller dans le bon répertoire, c'est-à-dire là où il y a le fichier « `CMakeLists.txt`[1] », puis exécuter `cmake` et `make` de la manière suivante :

```
cd <chemin vers le dossier iloMath>
cmake CMakeLists.txt
make
```

Si tout va bien, la bibliothèque est créée sous `lib/libiloMath.a`. On peut ensuite par exemple exécuter les exemples de ce guide en créeant un code minimal contenant l'exemple dans un fichier `main.cpp`, et le compiler avec

```
g++ main.cpp -L<chemin vers le dossier iloMath/lib> -liloMath -I<chemin
    vers le dossier iloMath/include> -o Test
```

Ceci crée un programme appelé `Test`, qui exécute l'exemple donné. Pour des programmes plus compliqués, un moyen est de faire appel à make voire cmake, dont divers tutoriels sont disponibles sur le web. Sinon, voir le logiciel iloCalc du même auteur qui est entièrement basé sur iloMath et qui peut faire office d'exemple.

Le fichier à inclure est `iloMath.h`. Toutes les classes et fonctions d'iloMath sont dans l'espace de noms `ilm`.

Il devrait être possible de compiler sous d'autres systèmes d'exploitation, en utilisant leurs propres outils, la compilation croisée, ou encore des portages de g++, make et cmake. Par exemple, pour Windows, on peut compiler en utilisant MSYS, ou encore depuis Linux avec MinGW.

[1] Ne pas hésiter à l'adapter si besoin, par exemple pour avoir une compilation dynamique au lieu de statique.

1.6 Informations supplémentaires à propos de la documentation

Avant de commencer la documentation à proprement dite, quelques informations sont bonnes à savoir. Pour commencer, la bibliothèque utilise le C++11.

Dans les signatures des fonctions, les passages par références constantes sont simplement affichées comme des passages par copie pour des raison de simplicité. En général, les variables de type élémentaire sont passés par copie, et les autres par référence constante, sauf si on est certain que la variable doit de toute manière être copiée.

La bibliothèque et sa documentation utilisent les alias standards `(u)int8_t`, `(u)int16_t`, `(u)int32_t` et `(u)int64_t` pour désigner les nombres entiers. En général, `int8_t = char`, `int16_t = short`, `int32_t = int` et `int64_t = long int`, mais ceci peut varier selon la plateforme utilisée par la compilation.

Des constantes sont définies avec le préprocesseur (dans `iloMath/General.h`) et peuvent être modifiées par un développeur pour adapter le logiciel à ses besoins :

```
#define ILM_INTERNALBASE    10
#define ILM_USERBASE        10
#define ILM_FACLUTSIZE      256
#define ILM_DEFAULTITERS    0
#define ILM_MINKARATSUBA    128
```

- `ILM_INTERNALBASE` est utilisé pour indiquer quelle est la base interne, c'est-à-dire dans quelle base les nombres arbitrairement grands sont stockés. Il peut être utile de changer cette valeur si on travaille dans une base autre que 10, car ceci permet d'éviter des conversions de bases à la saisie et à l'affichage ;
- `ILM_USERBASE` est utilisé pour indiquer quelle est la base par défaut pour la saisie et l'affichage des nombres arbitrairement grands ;
- `ILM_FACLUTSIZE` : ceci indique le nombre de factorielles arbitrairement grandes précalculées au démarrage d'un programme utilisant iloMath. Ceci permet d'accélérer par exemple des calculs utilisant les séries de Taylor ;
- `ILM_DEFAULTITERS` : ceci indique le nombre d'itérations par défaut à utiliser pour les algoritmes de calcul numérique, par exemple combien de termes de Taylor doivent être calculés ;
- `ILM_MINKARATSUBA` : la multiplication des entiers arbitrairement grands utilisent l'algorithme de Karatsuba pour le calcul des grands nombres. Cette constante indique quelle longueur minimal les deux opérandes doivent avoir pour que l'algorithme soit utilisé.

Des énumérations sont également définies et ont par exemple pour rôle d'avoir des noms plus parlants (comme `POSITIVE` au lieu de faux pour représenter le signe).

- Signes (énumération anonyme) : `POSITIVE = 0`, `NEGATIVE = 1` ;
- Ouvert ou fermé pour les intervalles (énumération anonyme) : `OPEN = 0`, `CLOSED = 1` (en référence à l'électronique : un circuit fermé laisse passer le courant) ;
- Opérations (`Operation`) : `OPNONE`, `OPPLUS`, `OPMINUS`, `OPTIMES`, `OPDIV`, `OPPOW`, `OPFAC`, `OPCOMPOSE` ;
- Chiffres significatifs ou décimales, pour l'utilisation des fonctions `strApprox` (énumération anonyme) : `SIGNIFICANT = 0 DECIMAL = 1` ;
- Types (`Type`) : `ilm_NNumber`, `ilm_NNumberL`, `ilm_ZNumber`, `ilm_QNumber`, `ilm_RNumberL`, `ilm_CNumber`, `ilm_CNumberL`, `ilm_Operation`, `ilm_Function`, `ilm_Tensor` ;

2. Nombres

2.1 Généralités

iloMath gère nativement les nombres naturels, relatifs, rationnels, et complexes rationnels de manière exacte et de taille arbitrairement grande. Toutes les bases entre 2 et 65535 sont également gérées. Il existe également des versions allégées de ces nombres, qui perdent certains avantages (l'arbitrairement grand ou l'exactitude) au profit de la rapidité.

Les opérations usuelles (quatre opérations de base, modulo, puissance, etc.) sont implémentées.

Les nombres réels et complexes à coefficients réels arbitrairement grands ou exacts ne sont par contre pas directement gérés.

2.1.1 Format d'affichage

Les nombres en base 10 et inférieur sont représentés à l'aide des neuf chiffres. Les nombres dans une base comprise entre 11 et 16 sont représentés à l'aide des dix chiffres et des lettres majuscules A, B, C, D, E, F. Les nombres de base dès 17 sont représentés avec des chiffres ou groupes de chiffres de 0 à 9 espacés par des deux-points, en analogie au système heures-minutes-secondes ou à la notation des adresses MAC et Ipv6. Par exemple,

$$1001111010010101011_2 = 324779_{10} = 4F4AB_{16} = 4:244:171_{256}$$

Si une base est un paramètre d'une fonction, elle est toujours facultative, et vaut par défaut `ILM_USERBASE` .

2.2 Types de nombres

Voici la liste de tous les nombres gérés par iloMath.

- `NNumber` : nombres naturels arbitrairement grands et exacts ;
- `NNumberL` : nombres naturels allégés, basés sur le type `uint64_t` ;
- `Z<T>` : nombres relatifs dont la partie numérique est stockée dans le type `T = NNumber` ou `T = NNumberL` ;
- `QNumber` : nombres rationnels dont les numérateur et dénominateur sont stockés dans un `Z<NNumber>` ;
- `RNumberL` : nombres réels allégés, basés sur le type `long double` ;
- `C<T>` : nombres complexes pouvant être basés sur le type `T = RNumberL` ou `T = QNumber` pour le stockage des parties réelle et imaginaire.

2.3 Caractéristiques communes

Toutes les classes de nombres s'utilisent de la même manière, avec des variations propres à chaque ensemble de nombre qu'ils représentent.

2.3.1 Syntaxe, création, et affichage

Pour tous les types de nombre, un nombre peut être créé en le déclarant et en l'initialisant à l'aide d'un `std::string`. On peut aussi l'initialiser plus tard (il vaudra par défaut 0), en l'affectant plus tard un `std::string`. Voir les documentations qui suivent et les exemples pour voir concrètement comment utiliser ces nombres, et d'autres possibilités (comme la gestion d'autres bases).

Diagrammes de syntaxe

Pour l'initialisation, le `std::string` doit être syntaxiquement valide, et un contrôle est effectué avant chaque initialisation : une saisie invalide initialise un nombre indéfini (pas dans le sens comportement indéfini, mais dans le sens où il est attribué au nombre un état bien précis). Les diagrammes suivants représentent précisément la forme qu'une saisie doit avoir selon le type.

NNumber et NNumberL Si la base est ≤ 16, le diagramme syntaxique est :

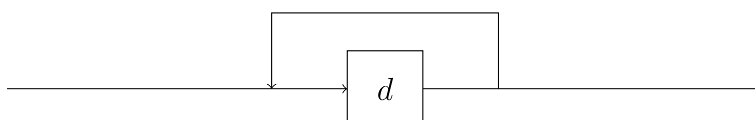

FIGURE 2.1 – d est un chiffre, soit un caractère qui est un chiffre entre 0 et 9 ou une lettre majuscule de A à F. Les chiffres représentés par ces caractères doivent être strictement inférieurs à la base.

Sinon, le diagramme est le suivant :

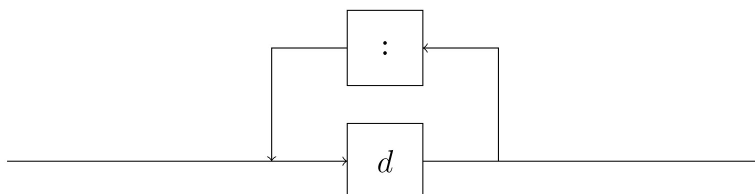

FIGURE 2.2 – d est ici un nombre entier naturel exprimé en base 10, strictement inférieur à la base.

S'il y a des zéros inutiles, ils seront automatiquement supprimés (la saisie est considérée comme valide quand même).

Exemples Les lignes sont des saisies et les colonnes sont des bases. La case indique si la saisie est valide (o) ou non (x) dans la base donnée.

	2	10	16	60
`10010011`	o	o	o	x
`64`	x	o	o	x
`3AF`	x	x	o	x
`EFG`	x	x	x	x
`22:59:5`	x	x	x	o
`101`	o	o	o	x

Z Le diagramme est :

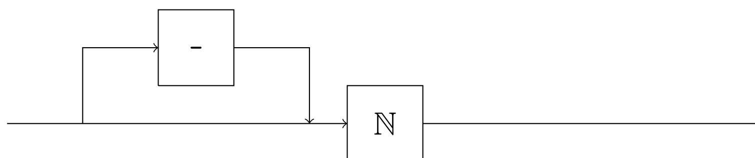

FIGURE 2.3 – - est le signe négatif, et est facultatif. ℕ est à remplacer par le diagramme syntaxique de NNumber associé à la base considérée.

Exemples En base 10, 123, -45 sont des saisies valides. 1-23, 4.5, +6789, abc et 12) ne le sont pas.

QNumber Pour les nombres rationnels, on peut entrer le nombre soit sous la forme d'une fraction, soit sous une représentation décimale, mais pas les deux.

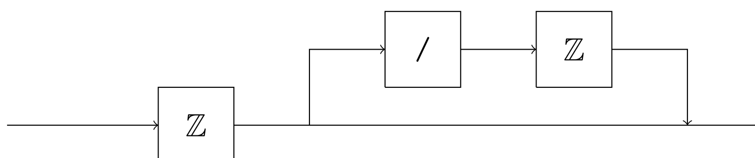

FIGURE 2.4 – Pour la saisie d'une fraction. / est une barre oblique symbolisant la barre de fraction. ℤ est à remplacer par le diagramme syntaxique de Z.

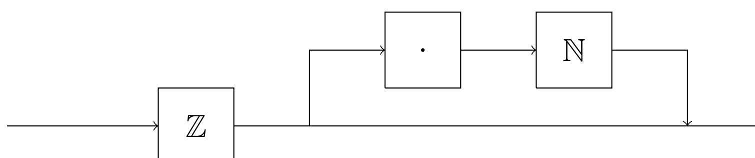

FIGURE 2.5 – Pour la saisie dans une écriture décimale. . est un point symbolisant le séparateur décimal. ℕ et ℤ sont à remplacer par les diagrammes syntaxiques de NNumber et Z respectivement.

Exemples En base 10, 123, -4.5 et 6/78 sont des saisies valides. 1-23, 5.-6, 1..2, 85/ et .12 ne le sont pas.

RNumberL Le nombre doit être saisi dans une syntaxe similaire à celle utilisée pour les nombres à virgule flottante en C++. Par exemple, les saisies 1234, 901.23 et -1.6e-19 sont valides, mais pas 1/2, 0.0.0 ou 1e.

C Pour les nombres complexes, la saisie est valide si elle respecte le diagramme suivant :

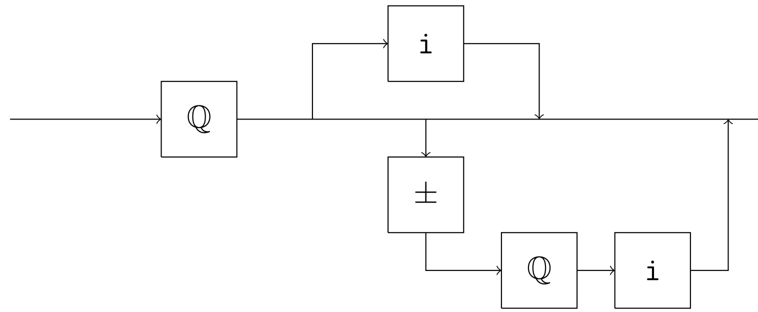

FIGURE 2.6 – `i` symbolise l'unité imaginaire. ± est un signe positif ou négatif séparant la partie réelle et imaginaire. ℚ est à remplacer par les diagrammes syntaxiques de `QNumber` ou par la validité de `RNumberL`.

Les saisies `-i` et `i` sont également valides.

Exemples En base 10, `1 + 2i`, `-4.5`, `6/78i` et `910.11 + 12/1314i` sont des saisies valides. `i - 3`, `i/5`, `i7`, `(1/11)i` et `13 + 17i/19` ne le sont pas.

Création et affichage de nombres

Pour créer un nombre, on l'initialise à l'aide d'un `std::string` contenant le nombre dans la base de l'utilisateur. Si on veut entrer un nombre dans une autre base, on l'initialise à l'aide d'un deuxième argument de type `uint16_t` spécifiant la base. Le nombre peut ensuite s'afficher comme n'importe quel nombre standard C++, à l'aide de l'opérateur de flux `<<`. Le nombre peut aussi être converti en un `std::string` à l'aide de la méthode commune `str`.

```
1 ilm::NNumber n0("23417"), n1("abcde"), n3("ABCDEF", 16);
2 std::cout << "n0 = " << n0 << std::endl;
3 std::cout << "n1 = " << n1.str() << std::endl;
4 std::cout << "n3 = " << n3 << " (en base 10)" << std::endl;
5 std::cout << "n3 = " << n3.str(16) << " (en base 16)" << std::endl;
```

```
x = 23417
y = undef
n3 = 11259375 (en base 10)
n3 = ABCDEF (en base 16)
```

2.3.2 Opérateurs arithmétiques

Les opérations usuelles ont été implémentées.

```
1 ilm::QNumber a("1/2"), b("-3/4");
2 std::cout << "a + b = " << a + b << std::endl;
3 std::cout << "a - b = " << a - b << std::endl;
4 std::cout << "a*b   = " << a*b   << std::endl;
5 std::cout << "a/b   = " << a/b   << std::endl;
```

```
a + b = -1/4
a - b = 5/4
a*b   = -3/8
a/b   = -2/3
```

2.3.3 Opérateurs de comparaison

Les opérateurs de comparaison ont également été implémentés pour chaque classe. À noter que pour les nombres complexes, seuls les opérateurs de comparaisons == et != sont disponibles.

```
1  ilm::QNumber x("-365/7"), y("31/24");
2
3  if (x == y) std::cout << "x est égal à y" << std::endl;
4  else std::cout << "x n'est pas égal à y" << std::endl;
5  if (x != y) std::cout << "x est différent de y" << std::endl;
6  else std::cout << "x n'est pas différent de y" << std::endl;
7  if (x < y) std::cout << "x est strictement inférieur à y" << std::endl;
8  else std::cout << "x n'est pas strictement inférieur à y" << std::endl;
9  if (x <= y) std::cout << "x est inférieur ou égal à y" << std::endl;
10 else std::cout << "x n'est pas inférieur ou égal à y" << std::endl;
11 if (x > y) std::cout << "x est strictement supérieur à y" << std::endl;
12 else std::cout << "x n'est pas strictement supérieur à y" << std::endl;
13 if (x >= y) std::cout << "x est supérieur ou égal à y" << std::endl;
14 else std::cout << "x n'est pas supérieur ou égal à y" << std::endl;
```

```
x n'est pas égal à y
x est différent de y
x est strictement inférieur à y
x est inférieur ou égal à y
x n'est pas strictement supérieur à y
x n'est pas supérieur ou égal à y
```

2.4 Nombres naturels (NNumber)

Les nombres naturels sont stockés grâce à la classe `NNumber`.

2.4.1 Description

Pour gérer des nombres arbitrairement grands, le principe est simple : les nombres sont stockés dans un `vector<uint32_t>`[1], où chaque case correspond à un chiffre. Le tableau dynamique permet d'élargir autant qu'on souhaite la quantité de chiffres stockés pour avoir un nombre aussi grand qu'on veut, dans la limite de la mémoire contigüe disponible.

Le code a été conçu pour faire coïncider les indices avec les exposants (petit-boutisme). Ainsi, le troisième chiffre (indice 3) du tableau dynamique correspond au chiffre des milliers.

Cette classe comporte également deux booléennes, qui stockent si un nombre est infini ou indéfini. Elles servent par exemple à gérer correctement les divisions par zéro ou les saisies invalides.

2.4.2 Documentation de la classe

On dira dans un `NNumber` que le i^e chiffre est celui qui correspond à la puissance i^e de la base considérée, on encore le i^e chiffre depuis la droite, en commençant par 0 (également l'indice au sens du C++).

[1] On pourrait se dire que c'est du gaspillage de place et que des `uint16_t`, voire `uint8_t` auraient pu être utilisées à la place. La raison de l'utilisation d'entiers 32 bits est d'une part le support des bases jusqu'à 65535, mais aussi le fait que lors de la multiplication (ou simplement l'addition), on serait souvent amené à des débordements d'entiers lors de la multiplication chiffre par chiffre.

Attributs

Ces attributs sont privés, mais toutefois documentés pour mieux comprendre la gestion des cas particuliers.

— std::vector<uint32_t> _n : tableau dynamique dont chaque case stocke un chiffre du nombre arbitrairement grand, exprimé dans la base interne. Les indices des cases coïncident avec les exposants correspondants aux chiffres ;
— bool _inf : indique si le nombre est infini (par exemple, après une division par zéro). Vrai pour oui et faux pour non ;
— bool _undef : indique si le nombre est indéfini (par exemple, suite à une opération comme $\frac{0}{0}$). Vrai pour oui et faux pour non.

On dira que le nombre est infini si _inf est vraie, et indéfini[2] si bool _undef est vrai. Si d'une manière ou d'une autre, les deux sont vraies, c'est le caractère indéfini qui prime. Dès que l'une des booléenne devient vraie, _n est remplacé par un std::vector<uint32_t> contenant juste un élément 0.

Constructeurs

— NNumber() : constructeur par défaut, initialise le nombre à zéro, et ses booléennes à faux ;
— NNumber(std::string, uint16_t = ILM_USERBASE) : crée un NNumber à partir d'un std::string. Crée un nombre indéfini si cette chaine est invalide. La base dans laquelle le nombre est exprimé peut être fournie comme paramètre (facultatif) ;
— NNumber(const char*, uint16_t = ILM_USERBASE) : comme ci-dessus ; pour gérer les initialisations avec l'expression entre guillemets ;
— NNumber(S a) avec S un type standard parmi uint8_t, uint16_t, uint32_t, uint64_t, int8_t, int16_t, int32_t, int64_t, bool : crée un NNumber à partir d'un entier standard. Crée un nombre indéfini si le nombre est négatif.

Accesseurs

— uint32_t size(uint16_t base = ILM_USERBASE) const : renvoie la taille du std::vector contenant les chiffres quand le nombre est exprimé dans la base donnée (argument facultatif) ;
— std::vector<uint32_t> n(uint16_t base = ILM_USERBASE) const : renvoie le std::vector contenant les chiffres du nombre quand il est exprimé dans la base donnée (argument facultatif) ;
— bool isInf() const : renvoie la valeur de _inf ;
— bool isUndef() const : renvoie la valeur de _undef ;
— uint32_t gdigit(uint32_t i, uint16_t b = ILM_USERBASE) const : renvoie le i^e chiffre ; cet indice est celui lorsque le nombre est exprimé dans la base b (argument facultatif). Si l'indice est plus grand ou égal au nombre de chiffres, 0 est renvoyé ;
— NNumber re() const : renvoie la partie réelle, soit le nombre lui-même ;
— NNumber im() const : renvoie la partie imaginaire, soit le nombre 0 ;
— std::string str(uint16_t = ILM_USERBASE) const : renvoie le nombre exprimé dans la base donnée (facultatif) sous la forme d'un std::string ;
— std::string strScientific(uint32_t s, uint16_t = ILM_USERBASE) const : renvoie le nombre exprimé en notation scientifique avec s chiffres significatifs dans la base donnée (facultatif) sous la forme d'un string ;
— Type type() const : renvoie la valeur ilm_NNumber de l'énumération des types.

[2]Ne pas confondre avec le comportement indéfini : là, le caractère indéfini du nombre est bien défini grâce à cette booléenne

Mutateurs

- `void reset()` : remet le nombre à zéro et les booléennes à faux ;
- `void setInf()` : appelle `reset()` et rend le nombre infini en affectant à `_inf` la valeur vraie.
- `void setUndef()` : appelle `reset()` et rend le nombre indéfini en affectant à `_undef` la valeur vraie ;
- `void sdigit(uint32_t d, uint32_t i, uint16_t b = ILM_USERBASE)` : remplace le i^e chiffre par d ; l'indice est celui lorsque le nombre est exprimé dans la base b (argument facultatif). Si $d \geq b$, ceci rend le nombre indéfini. Si l'indice est plus grand ou égal au nombre de chiffres, des zéros sont insérés entre la position choisie et le chiffre tout à gauche ;
- `void floor(uint32_t i, uint16_t b = ILM_USERBASE)` : le nombre devient le multiple de b^i inférieur le plus proche. Ceci équivaut à un arrondi vers le bas si b est la base est celle dans laquelle on travaille (argument facultatif). Ne fait rien si $i = 0$ et rend le nombre nul si b^i est supérieur au nombre ;
- `void ceil(uint32_t i, uint16_t b = ILM_USERBASE)` : comme `floor`, mais au supérieur ;
- `void round(uint32_t i, uint16_t b = ILM_USERBASE)` : comme `floor` et `ceil`, mais au multiple de b^i le plus proche ;
- `void shr(uint32_t i = 1, uint16_t b = ILM_USERBASE)` : tronque le nombre en enlevant i chiffres, après avoir exprimé le nombre dans la base b (facultatif). Équivalent à décaler vers i fois à droite le nombre s'il est écrit dans cette base, d'où le nom « shr » issu du langage assembleur ;
- `void shl(uint32_t i = 1, uint16_t b = ILM_USERBASE)` : le contraire de `shr` en ajoutant i zéros à la fin du nombre après l'avoir exprimé dans la base b. « Shl » est également issu du langage assembleur ;

Fonctions statiques

- `static bool isValid(std::string, uint16_t = ILM_USERBASE)` : renvoie si le `std::string` est un `NNumber` valide dans la base donnée (facultatif) ;
- `static NNumber zero()` : renvoie le nombre 0 dans le type `NNumber` ;
- `static NNumber one()` : renvoie le nombre 1 dans le type `NNumber` ;
- `static NNumber two()` : renvoie le nombre 2 dans le type `NNumber`.

Opérateurs/opérations arithmétiques

Opérateurs avec affectation :

- `NNumber& operator+=(NNumber y)` : ajoute y au nombre à l'aide d'un algorithme élémentaire ;
- `NNumber& operator-=(NNumber y)` : soustrait y au nombre à l'aide d'un algorithme élémentaire ;
- `NNumber& operator*=(NNumber y)` : multiplie le nombre par y à l'aide de l'algorithme de Karatsuba si les nombres sont suffisamment grands, classique sinon ;
- `NNumber& operator/=(NNumber y)` : divise le nombre par y à l'aide de la division euclidienne ;
- `NNumber& operator%=(NNumber y)` : le nombre devient le reste de la division par y ;
- `NNumber& operator++()` : incrémentation préfixée ;
- `NNumber& operator--()` : décrémentation préfixée ;
- `NNumber operator++(int)` : incrémentation postfixée ;
- `NNumber operator--(int)` : décrémentation postfixée.

Opérations sans affectation :

— `NNumber operator+(NNumber x, NNumber y)` : renvoie $x + y$;
— `NNumber operator-(NNumber x, NNumber y)` : renvoie $x - y$;
— `NNumber operator*(NNumber x, NNumber y)` : renvoie xy ;
— `NNumber operator/(NNumber x, NNumber y)` : renvoie $x \div y$;
— `NNumber operator%(NNumber x, NNumber y)` : renvoie $x \mod y$;

Toute opération faisant intervenir au moins un nombre indéfini donne un résultat indéfini, sans exception. Les autres cas particuliers sont les suivants : soient a et b des `NNumber` ni infinis, ni indéfinis, et ∞ un `NNumber` infini.

— Addition : $a + \infty = \infty$, $\infty + b = \infty$;
— Soustraction : $a-b$ donne un résultat indéfini si b est plus grand que a. Toute soustraction par ∞ également. Sinon, $\infty - b = \infty$;
— Multiplication : $0 \times \infty$ et $\infty \times 0$ donnent un résultat indéfini. Sinon, si au moins un nombre est infini, le résultat l'est également ;
— Division : $\infty \div \infty$ et $0 \div 0$ donnent un résultat indéfini, $a \div 0 = \infty$;
— Modulo : tout modulo avec ∞ comme opérande à gauche, ou 0 comme opérande de droite donnent un résultat indéfini, $a \mod \infty = a$.

La définition des constructeurs à partir d'entiers standards permettent d'utiliser les opérateurs avec des `NNumber` et ces entiers mélangés (par exemple, l'addition d'un `NNumber` avec un `int`).

La puissance, la factorielle, le *PGDC* et le *PPMC* sont également disponibles, voir le chapitre des fonctions pour leur documentation.

Opérateurs de comparaison

— `bool operator==(NNumber x, NNumber y)` : renvoie si $x = y$;
— `bool operator!=(NNumber x, NNumber y)` : renvoie si $x \neq y$;
— `bool operator<(NNumber x, NNumber y)` : renvoie si $x < y$;
— `bool operator<=(NNumber x, NNumber y)` : renvoie si $x \leq y$;
— `bool operator>(NNumber x, NNumber y)` : renvoie si $x > y$;
— `bool operator>=(NNumber x, NNumber y)` : renvoie si $x \geq y$.

Cas particuliers :

— Si au moins un nombre est indéfini, la comparaison est toujours fausse ;
— La comparaison de deux infinis est également toujours fausse.

Ici également, la définition des constructeurs à partir d'entiers standards permettent d'utiliser les opérateurs avec des `NNumber` et ces entiers mélangés.

Autres méthodes et fonctions associées

Autres méthodes :

— `bool isZero() const` : renvoie si le nombre est nul. Renvoie faux si le nombre est infini ou indéfini ;
— `bool isEven() const` : renvoie si le nombre est pair Renvoie faux si le nombre est infini ou indéfini ;
— `void split(NNumber& n1, NNumber& n0, uint32_t i, uint32_t base = ILM_USERBASE) const` : sépare le nombre en deux parties : n_0 contient les i chiffres depuis la gauche quand le nombre est écrit en base b (facultatif) (est nul si $i = 0$) et n_1 les autres chiffres (est nul si i est supérieur ou égal au nombre de chiffres) ;
— `NNumber rand() const` : renvoie un nombre aléatoire entre 0 et le nombre lui-même inclus. Renvoie un nombre indéfini si le nombre est infini ou indéfini.

Fonctions extérieures à la classe :

— `NNumber delta(NNumber, NNumber)` : renvoie l'écart entre deux nombres (équivalent à la valeur absolue d'une différence) ;
— `NNumber rand(NNumber, NNumber)` : renvoie un nombre aléatoire entre les deux nombres donnés ;
— `NNumber floor(NNumber, uint32_t, uint16_t)` : voir la méthode homologue ;
— `NNumber ceil(NNumber, uint32_t, uint16_t)` : voir la méthode homologue ;
— `NNumber round(NNumber, uint32_t, uint16_t)` : voir la méthode homologue ;
— `std::ostream& operator<<(std::ostream&, NNumber)` : permet par exemple d'afficher un `NNumber` en utilisant `std::cout` comme s'il s'agissait d'un type standard.

2.4.3 Exemple

Calcul d'un terme de Fibonacci.

```
1 uint32_t n(256);
2 ilm::NNumber u0(1), u1(1), un;
3 for (uint32_t i(2) ; i < n ; i++) {
4     un = u0 + u1;
5     u0 = u1;
6     u1 = un;
7 }
8 std::cout << "F(" << n << ") = " << un << std::endl;
```

```
F(256) = 141693817714056513234709965875411919657707794958199867
```

2.5 Nombres naturels légers (NNumberL)

2.5.1 Description

`NNumberL` est une version allégée de `NNumberL`. En effet, il ne stocke pas les nombres dans un tableau dynamique, mais dans un `uint64_t`, un type standard pouvant stocker tous les nombres entiers entre 0 et $2^{64} - 1$. De ce fait, cette nouvelle classe perd la propriété de l'arbitrairement grand et de la gestion des bases, au profit de la rapidité.

Autrement, le principe de fonctionnement est sensiblement le même, et la plupart des fonctions de `NNumber` sont reprises.

2.5.2 Documentation de la classe

Le i^e chiffre dans un `NNumberL` s'interprétera comme pour `NNumber`. On travaille toujours en base 10 avec cette classe.

Attributs

Ces attributs sont privés, mais toutefois documentés pour mieux comprendre la gestion des cas particuliers.

— `uint64_t _n` : stocke le nombre. Les indices des cases coïncident avec les exposants correspondants aux chiffres ;
— `bool _inf` et `bool _undef` : jouent le même rôle que pour `NNumber`. Dès que l'une des booléenne devient vraie, `_n` est mis à 0.

Constructeurs

- `NNumberL()` : constructeur par défaut, initialise le nombre à zéro, et ses booléennes à faux ;
- `NNumberL(std::string)` : crée un `NNumberL` à partir d'un `std::string`. Crée un nombre indéfini si cette chaine est invalide ;
- `NNumberL(const char*)` : comme ci-dessus ; pour gérer les initialisations avec l'expression entre guillemets ;
- `NNumberL(S a)` avec `S` un type standard parmi `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `int8_t`, `int16_t`, `int32_t`, `int64_t`, `bool` : crée un `NNumberL` à partir d'un entier standard. Crée un nombre indéfini si le nombre est négatif.

Accesseurs

- `uint32_t size() const` : renvoie la longueur du nombre ;
- `uint64_t n() const` : renvoie le `uint64_t` stockant le nombre ;
- `bool isInf() const` : renvoie la valeur de `_inf` ;
- `bool isUndef() const` : renvoie la valeur de `_undef` ;
- `uint32_t gdigit(uint32_t i) const` : renvoie le i^e chiffre ; si l'indice est plus grand ou égal au nombre de chiffres, 0 est renvoyé ;
- `NNumberL re() const` : renvoie la partie réelle, soit le nombre lui-même ;
- `NNumberL im() const` : renvoie la partie imaginaire, soit le nombre 0 ;
- `std::string str() const` : renvoie le nombre sous la forme d'un `std::string` ;
- `std::string strScientific(uint32_t s) const` : renvoie le nombre exprimé en notation scientifique avec s chiffres significatifs sous la forme d'un string ;
- `Type type() const` : renvoie la valeur `ilm_NNumberL` de l'énumération des types.

Mutateurs

- `void reset()` : remet le nombre à zéro et les booléennes à faux ;
- `void setInf()` : appelle `reset()` et rend le nombre infini en affectant à `_inf` la valeur vraie.
- `void setUndef()` : appelle `reset()` et rend le nombre indéfini en affectant à `_undef` la valeur vraie ;
- `void sdigit(uint32_t d, uint32_t i)` : remplace le i^e chiffre par d. Si $d \geq 10$, ceci rend le nombre indéfini. Si l'indice est plus grand ou égal au nombre de chiffres, des zéros sont insérés entre la position choisie et le chiffre tout à gauche ;
- `void floor(uint32_t i)` : le nombre devient le multiple de 10^i inférieur le plus proche (arrondi vers le bas en base 10). Ne fait rien si $i = 0$ et rend le nombre nul si 10^i est supérieur au nombre ;
- `void ceil(uint32_t i)` : comme `floor`, mais au supérieur ;
- `void round(uint32_t i)` : comme `floor` et `ceil`, mais au multiple de 10^i le plus proche ;
- `void shr(uint32_t i = 1)` : tronque le nombre en enlevant i chiffres. Équivalent à décaler vers i fois à droite le nombre, d'où le nom « shr » issu du langage assembleur ;
- `void shl(uint32_t i = 1)` : le contraire de `shr` en ajoutant i zéros à la fin du nombre. « Shl » est également issu du langage assembleur ;

Fonctions statiques

- `static bool isValid(std::string)` : renvoie si le `std::string` est un `NNumberL` valide ;

— `static NNumberL zero()` : renvoie le nombre 0 dans le type NNumber ;
— `static NNumberL one()` : renvoie le nombre 1 dans le type NNumber ;
— `static NNumberL two()` : renvoie le nombre 2 dans le type NNumber.

Opérateurs/opérations arithmétiques

Opérateurs avec affectation (les opérations `+ - * / %` et incrémentations sont implémentées en utilisant les opérateurs standards entre des `uint64_t`) :

— `NNumberL& operator+=(NNumberL y)` : ajoute y au nombre ;
— `NNumberL& operator-=(NNumberL y)` : soustrait y au nombre ;
— `NNumberL& operator*=(NNumberL y)` : multiplie le nombre par y ;
— `NNumberL& operator/=(NNumberL y)` : divise le nombre par y ;
— `NNumberL& operator%=(NNumberL y)` : le nombre devient le reste de la division par y ;
— `NNumberL& operator++()` : incrémentation préfixée ;
— `NNumberL& operator--()` : décrémentation préfixée ;
— `NNumberL operator++(int)` : incrémentation postfixée ;
— `NNumberL operator--(int)` : décrémentation postfixée.

Opérations sans affectation :

— `NNumberL operator+(NNumberL x, NNumberL y)` : renvoie $x + y$;
— `NNumberL operator-(NNumberL x, NNumberL y)` : renvoie $x - y$;
— `NNumberL operator*(NNumberL x, NNumberL y)` : renvoie xy ;
— `NNumberL operator/(NNumberL x, NNumberL y)` : renvoie $x \div y$;
— `NNumberL operator%(NNumberL x, NNumberL y)` : renvoie $x \mod y$;

Les cas particuliers sont les mêmes que pour `NNumber`, se référer à sa documentation pour en savoir plus. À noter également qu'aucun contrôle de débordement n'est effectué.

La puissance, la factorielle, le *PGDC* et le *PPMC* sont également disponibles, voir le chapitre des fonctions pour leur documentation.

La définition des constructeurs à partir d'entiers standards permettent d'utiliser les opérateurs avec des `NNumberL` et ces entiers mélangés (par exemple, l'addition d'un `NNumberL` avec un `int`).

Opérateurs de comparaison

— `bool operator==(NNumberL x, NNumberL y)` : renvoie si $x = y$;
— `bool operator!=(NNumberL x, NNumbeL y)` : renvoie si $x \neq y$;
— `bool operator<(NNumberL x, NNumberL y)` : renvoie si $x < y$;
— `bool operator<=(NNumberL x, NNumberL y)` : renvoie si $x \leq y$;
— `bool operator>(NNumberL x, NNumberL y)` : renvoie si $x > y$;
— `bool operator>=(NNumberL x, NNumberL y)` : renvoie si $x \geq y$.

Comme pour `NNumber`, si au moins un nombre est indéfini ou les deux sont infinis, la comparaison est toujours fausse.

Ici également, la définition des constructeurs à partir d'entiers standards permettent d'utiliser les opérateurs avec des `NNumberL` et ces entiers mélangés.

Autres méthodes et fonctions associées

Autres méthodes :

— `bool isZero() const` : renvoie si le nombre est nul. Renvoie faux si le nombre est infini ou indéfini ;
— `bool isEven() const` : renvoie si le nombre est pair Renvoie faux si le nombre est

infini ou indéfini ;
— `void split(NNumberL& x, NNumberL& n0, uint32_t i) const` : sépare le nombre en deux parties : n_0 contient les i chiffres depuis la gauche (est nul si $i = 0$) et x les autres chiffres (est nul si i est supérieur ou égal au nombre de chiffres) ;
— `NNumberL rand() const` : renvoie un nombre aléatoire entre 0 et le nombre lui-même inclus. Renvoie un nombre indéfini si le nombre est infini ou indéfini.

Fonctions extérieures à la classe :

— `NNumberL delta(NNumberL, NNumberL)` : renvoie l'écart entre deux nombres (équivalent à la valeur absolue d'une différence) ;
— `NNumberL rand(NNumberL, NNumberL)` : renvoie un nombre aléatoire entre les deux nombres donnés ;
— `NNumberL floor(NNumberL, uint32_t)` : voir la méthode homologue ;
— `NNumberL ceil(NNumberL, uint32_t)` : voir la méthode homologue ;
— `NNumberL round(NNumberL, uint32_t)` : voir la méthode homologue ;
— `std::ostream& operator<<(std::ostream&, NNumberL)` : permet par exemple d'afficher un `NNumberL` en utilisant `std::cout` comme s'il s'agissait d'un type standard.

2.5.3 Exemple

Opérations en vrac.

```
1  ilm::NNumberL x("8"), y("17");
2  std::cout << "x + y   = " << x + y << std::endl;
3  std::cout << "x - y   = " << x - y << std::endl;
4  std::cout << "y - x   = " << y - x << std::endl;
5  std::cout << "d(x, y) = " << ilm::delta(x, y) << std::endl;
6  std::cout << "x * y   = " << x * y << std::endl;
7  std::cout << "x / y   = " << x / y << std::endl;
8  std::cout << "y / x   = " << y / x << std::endl;
9  std::cout << "x % y   = " << x % y << std::endl;
10 std::cout << "x ^ y   = " << ilm::pow(x, y) << std::endl;
11 std::cout << "x!       = " << ilm::fac(x) << std::endl;
12 std::cout << "PGDC(x, y) = " << ilm::gcd(x, y) << std::endl;
13 std::cout << "PPMC(x, y) = " << ilm::lcm(x, y) << std::endl;
```

```
x + y   = 25
x - y   = undef
y - x   = 9
d(x, y) = 9
x * y   = 136
x / y   = 0
y / x   = 2
x % y   = 8
x ^ y   = 2251799813685248
x!       = 40320
PGDC(x, y) = 1
PPMC(x, y) = 136
```

2.6 Nombres relatifs (Z)

2.6.1 Description

Les nombres relatifs sont stockés grâce au patron de classes `Z<T>`, `T` étant un type de nombre naturel compatible.

Son principe est le suivant : il stocke la partie numérique du nombre dans un attribut de ce type `T`, et le signe dans un attribut de type `bool`. `Z<T>` reprend la plupart des membres de `NNumber(L)`, éventuellement adaptés pour gérer le signe, ce qui rend son utilisation similaire à ces classes.

2.6.2 Documentation du patron de classes

On a `T = NNumber` ou `NNumberL`. Pour `T = NNumberL`, à chaque fois que la base est un paramètre, il faut l'ignorer. Alias :

- `ZNumber  = Z<NNumber>` ;
- `ZNumberL = Z<NNumberL>`

Attributs

Ces attributs sont privés.

- `T _n` : stocke la partie numérique dans un `NNumber` ou un `NNumberL` ;
- `bool _sign` : stocke le signe du nombre. Vrai pour négatif et faux pour positif.

On n'a pas les booléennes `_inf` et `_undef` puisqu'ils sont déjà stockés dans `T`.

Constructeurs

- `Z()` : constructeur par défaut, initialise le nombre à zéro, et ses booléennes à faux ;
- `Z(std::string, uint16_t = ILM_USERBASE)` : crée un `Z` à partir d'un `std::string`. Crée un nombre indéfini si cette chaine est invalide. La base dans laquelle le nombre est exprimé peut être fournie comme paramètre (facultatif) ;
- `Z(const char*, uint16_t = ILM_USERBASE)` : comme ci-dessus ; pour gérer les initialisations avec l'expression entre guillemets ;
- `Z(S a)` avec `S` un type standard parmi `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `int8_t`, `int16_t`, `int32_t`, `int64_t`, `bool` : crée un `Z` à partir d'un entier standard.

Accesseurs

- `uint32_t size(uint16_t base = ILM_USERBASE) const` : renvoie la longueur du nombre quand il est exprimé dans la base donnée (argument facultatif) ;
- `T n() const` : renvoie la partie numérique dans le type `T` ;
- `bool isInf() const` : renvoie si le nombre est infini ;
- `bool isUndef() const` : renvoie si le nombre est indéfini ;
- `bool sign() const` : renvoie si le nombre est négatif ;
- `uint32_t gdigit(uint32_t i, uint16_t b = ILM_USERBASE) const` : renvoie le i^e chiffre ; cet indice est celui lorsque le nombre est exprimé dans la base b (argument facultatif). Si l'indice est plus grand ou égal au nombre de chiffres, 0 est renvoyé ;
- `Z re() const` : renvoie la partie réelle, soit le nombre lui-même ;
- `Z im() const` : renvoie la partie imaginaire, soit le nombre 0 ;
- `std::string str(uint16_t = ILM_USERBASE) const` : renvoie le nombre exprimé dans la base donnée (facultatif) sous la forme d'un `std::string` ;
- `std::string strScientific(uint32_t s, uint16_t = ILM_USERBASE) const` : renvoie le nombre exprimé en notation scientifique avec s chiffres significatifs dans la base donnée (facultatif) sous la forme d'un string ;
- `Type type() const` : renvoie la valeur `ilm_ZNumber` de l'énumération des types.

Mutateurs

- `void n(T)` : modifie la partie numérique ;
- `void reset()` : remet le nombre à zéro et les booléennes à faux ;
- `void setInf()` : appelle `reset()` et rend le nombre infini en affectant à `_inf` la valeur vraie.
- `void setUndef()` : appelle `reset()` et rend le nombre indéfini en affectant à `_undef` la valeur vraie ;
- `void sign(bool)` : définit le signe du nombre (négatif si la booléenne est vraie, positif sinon) ;
- `void sdigit(uint32_t d, uint32_t i, uint16_t b = ILM_USERBASE)` : remplace le i^e chiffre par d ; l'indice est celui lorsque le nombre est exprimé dans la base b (argument facultatif). Si $d \geq b$, ceci rend le nombre indéfini. Si l'indice est plus grand ou égal au nombre de chiffres, des zéros sont insérés entre la position choisie et le chiffre tout à gauche ;
- `void floor(uint32_t i, uint16_t b = ILM_USERBASE)` : le nombre devient le multiple de b^i inférieur le plus proche. Ceci équivaut à un arrondi vers le bas si b est la base est celle dans laquelle on travaille (argument facultatif). Ne fait rien si $i = 0$ et rend le nombre nul si b^i est supérieur au nombre ;
- `void ceil(uint32_t i, uint16_t b = ILM_USERBASE)` : comme `floor`, mais au supérieur ;
- `void round(uint32_t i, uint16_t b = ILM_USERBASE)` : comme `floor` et `ceil`, mais au multiple de b^i le plus proche ;
- `void shr(uint32_t i = 1, uint16_t b = ILM_USERBASE)` : tronque le nombre en enlevant i chiffres, après avoir exprimé le nombre dans la base b (facultatif). Équivalent à décaler vers i fois à droite le nombre s'il est écrit dans cette base, d'où le nom « shr » issu du langage assembleur ;
- `void shl(uint32_t i = 1, uint16_t b = ILM_USERBASE)` : le contraire de `shr` en ajoutant i zéros à la fin du nombre après l'avoir exprimé dans la base b. « Shl » est également issu du langage assembleur ;
- `void abs()` : enlève l'éventuel signe du nombre.

Fonctions statiques

- `static bool isValid(std::string, uint16_t = ILM_USERBASE)` : renvoie si le `std::string` est un `Z` valide dans la base donnée (facultatif) ;
- `static Z zero()` : renvoie le nombre 0 dans le type `Z` ;
- `static Z one()` : renvoie le nombre 1 dans le type `Z` ;
- `static Z two()` : renvoie le nombre 2 dans le type `Z`.

Opérateurs/opérations arithmétiques

Ces opérations ne sont en fait qu'une surcouche appelant les opérations homologues de `T` et appliquant la règle des signes. Opérateurs avec affectation :

- `Z& operator+=(Z y)` : ajoute y au nombre ;
- `Z& operator-=(Z y)` : soustrait y au nombre ;
- `Z& operator*=(Z y)` : multiplie le nombre par y ;
- `Z& operator/=(Z y)` : divise le nombre par y ;
- `Z& operator%=(Z y)` : le nombre devient le reste de la division par y ;
- `Z& operator++()` : incrémentation préfixée ;
- `Z& operator--()` : décrémentation préfixée ;
- `Z operator++(int)` : incrémentation postfixée ;

— `Z operator--(int)` : décrémentation postfixée.

Opérations sans affectation :

— `Z operator+(Z x, Z y)` : renvoie $x + y$;
— `Z operator-(Z x, Z y)` : renvoie $x - y$;
— `Z operator*(Z x, Z y)` : renvoie xy ;
— `Z operator/(Z x, Z y)` : renvoie $x \div y$;
— `Z operator%(Z x, Z y)` : renvoie $x \mod y$;
— `Z operator-()` : renvoie l'opposé du nombre.

En principe, les cas particuliers sont gérés par les opérateurs du type `T`, et en tenant compte des règles des signes (par exemple, $-1 \times \infty = -\infty$). Il y a toutefois quelques changements notables, et quelques nouvelles règles. Les changements par rapport à `T` sont :

— Soustraction : $a - b$ est maintenant bien défini si b est plus grand que a ou infini ;
— Division : diviser par zéro donne désormais toujours un nombre indéfini (au lieu de l'infini), même si le dividende n'est pas 0 ;
— Modulo : le résultat du modulo a toujours le signe de l'opérande de gauche.

La définition des constructeurs à partir d'entiers standards permettent d'utiliser les opérateurs avec des `Z` et ces entiers mélangés (par exemple, l'addition d'un `Z` avec un `int`).

Pour obtenir le PGDC ou le PPMC (sans signe), il faut utiliser l'accesseur `n` puis utiliser les fonctions `gcd` ou `lcm` du type `T`. La puissance et la factorielle sont également disponibles, voir le chapitre des fonctions pour leur documentation.

Opérateurs de comparaison

Comme pour les opérations, les opérateurs de comparaison sont une surcouche appelant les opérations homologues de `T` et appliquant la règle des signes. Les mêmes règles s'appliquent pour les cas particuliers. Opérateurs avec affectation :

— `bool operator==(Z x, Z y)` : renvoie si $x = y$;
— `bool operator!=(Z x, Z y)` : renvoie si $x \neq y$;
— `bool operator<(Z x, Z y)` : renvoie si $x < y$;
— `bool operator<=(Z x, Z y)` : renvoie si $x \leq y$;
— `bool operator>(Z x, Z y)` : renvoie si $x > y$;
— `bool operator>=(Z x, Z y)` : renvoie si $x \geq y$.

Ici également, la définition des constructeurs à partir d'entiers standards permettent d'utiliser les opérateurs avec des `Z` et ces entiers mélangés.

Autres méthodes et fonctions associées

Autres méthodes :

— `bool isZero() const` : renvoie si le nombre est nul. Renvoie faux si le nombre est infini ou indéfini ;
— `bool isEven() const` : renvoie si le nombre est pair Renvoie faux si le nombre est infini ou indéfini ;
— `Z rand() const` : renvoie un nombre aléatoire entre 0 et le nombre lui-même inclus. Renvoie un nombre indéfini si le nombre est infini ou indéfini.

Fonctions extérieures à la classe :

— `Z delta(Z x, Z y)` : renvoie $|x - y|$;
— `Z rand(Z, Z)` : renvoie un nombre aléatoire entre les deux nombres donnés ;
— `Z floor(Z, uint32_t)` : voir la méthode homologue ;
— `Z ceil(Z, uint32_t)` : voir la méthode homologue ;
— `Z round(Z, uint32_t)` : voir la méthode homologue ;

— `std::ostream& operator<<(std::ostream&, Z)` : permet par exemple d'afficher un `Z` en utilisant `std::cout` comme s'il s'agissait d'un type standard.

Pour utiliser `split`, il faut utiliser l'accesseur `n` puis utiliser la fonction `split` du type `T`.

2.6.3 Exemple

Exposition de diverses fonctions.

```
ilm::ZNumber z(-123456789);
z.floor(3, 10);
std::cout << "n = " << z.str() << std::endl;
z.ceil(5, 10);
std::cout << "n = " << z.str(10) << std::endl;
z.round(7, 10);
std::cout << "n = " << z.str(10) << std::endl;
std::cout << "n = " << z.str(16) << " (en base 16)" << std::endl;
z.round(5, 16);
std::cout << "n = " << z.str(16) << " (en base 16)" << std::endl;
std::cout << "n = " << z.str(10) << std::endl;
z.shl(5, 10);
std::cout << "n = " << z.str(10) << std::endl;
z.shr(10, 10);
std::cout << "n = " << z.str(10) << std::endl;
std::cout << "n = " << z.str(3) << " (en base 3)" << std::endl;
std::cout << "2e chiffre en base 10 : " << z.gdigit(1) << std::endl;
std::cout << "2e chiffre en base 3  : " << z.gdigit(1, 3) << std::endl;
z.sdigit(2, 10, 3);
std::cout << "n = " << z.str(3) << " (en base 3)" << std::endl;
std::cout << "n = " << z.str(10) << std::endl;
```

```
n = -123457000
n = -123400000
n = -120000000
n = -7270E00 (en base 16)
n = -7200000 (en base 16)
n = -119537664
n = -11953766400000
n = -1195
n = -1122021 (en base 3)
2e chiffre en base 10 : 9
2e chiffre en base 3  : 2
n = -20001122021 (en base 3)
n = -119293
```

2.7 Nombres rationnels (QNumber)

2.7.1 Description

Les nombres rationnels sont stockés grâce à la classe `QNumber`. Son principe est le suivant : il enregistre ces nombres sous la forme d'une fraction, dont le numérateur et le dénominateur sont ses attributs, de type `Z<NNumber>`. Les nombres décimaux sont indirectement gérés (possibilité de saisir et d'afficher ces nombres sous la forme décimale avec virgule), mais ne sont pas stockés sous cette forme.

Les `std::string` pour l'initialisation ou l'affectation peuvent autant être des fractions (avec barre de fraction) que des nombres décimaux (avec point comme virgule). Les fractions stockées sont toujours simplifiées.

2.7.2 Documentation du patron de classes

Le nombre est considéré comme indéfini si le numérateur ou le dénominateur est indéfini, si les deux sont infinis, ou si le dénominateur est nul. Il est considéré comme infini si le numérateur est infini et le dénominateur un nombre ni infini, ni indéfini, ni nul.

Attributs

Ces attributs sont privés.

- Z<NNumber> _num : stocke le numérateur ;
- Z<NNumber> _den : stocke le dénominateur.

Constructeurs

- QNumber() : constructeur par défaut, initialise le numérateur à 0 et le dénominateur à 1 ;
- QNumber(std::string, uint16_t = ILM_USERBASE) : crée un QNumber à partir d'un std::string. Crée un nombre indéfini si cette chaine est invalide. La base dans laquelle le nombre est exprimé peut être fournie comme paramètre (facultatif) ;
- QNumber(const char*, uint16_t = ILM_USERBASE) : comme ci-dessus ; pour gérer les initialisations avec l'expression entre guillemets ;
- QNumber(S a, S b = 1) avec S un type standard parmi uint8_t, uint16_t, uint32_t, uint64_t, int8_t, int16_t, int32_t, int64_t, bool : crée un QNumber à partir de deux entiers standards (numérateur et dénominateur). Si le dénominateur n'est pas spécifié, il vaut 1 ;
- QNumber(S a) avec S un type standard parmi float, double et long double : crée un QNumber à partir d'un nombre réel standard (précision maximale : 16 chiffres significatifs).

Accesseurs

- uint32_t size(uint16_t base = ILM_USERBASE) const : renvoie le nombre de chiffres avant la virgule quand le nombre est exprimé dans la base donnée (argument facultatif) ;
- Z<NNumber> num() const : renvoie le numérateur dans le type Z<NNumber> ;
- Z<NNumber> den() const : renvoie le dénominateur dans le type Z<NNumber> ;
- bool isInf() const : renvoie si le nombre est infini ;
- bool isUndef() const : renvoie si le nombre est indéfini ;
- bool sign() const : renvoie si le nombre est négatif ;
- bool isInt() const : renvoie si le nombre est entier ;
- Z<NNumber> intPart() const : renvoie la partie entière du nombre ;
- long double ld() const : renvoie le nombre sous la forme d'un long double ;
- QNumber re() const : renvoie la partie réelle, soit le nombre lui-même ;
- QNumber im() const : renvoie la partie imaginaire, soit le nombre 0 ;
- std::string str(uint16_t = ILM_USERBASE) const : renvoie le nombre exprimé dans la base donnée (facultatif) sous la forme d'un std::string ;
- std::string strScientific(uint32_t s, uint16_t = ILM_USERBASE) const : renvoie le nombre exprimé en notation scientifique avec s chiffres significatifs dans la base donnée (facultatif) sous la forme d'un string ;
- std::string strApprox(uint32_t d, bool t, bool f, uint16_t = ILM_USERBASE) const : renvoie le nombre approximé dans la base donnée (facultatif) avec d chiffres significatifs/après la virgule si t est vraie/fausse, en gardant les zéros inutiles si f est vraie ;
- Type type() const : renvoie la valeur ilm_QNumber de l'énumération des types.

Mutateurs

- void num(Z<NNumber>) : modifie le numérateur ;
- void den(Z<NNumber>) : modifie le dénominateur ;
- void reset() : remet le nombre à zéro ;
- void setInf() : rend le nombre infini (conserve le signe) ;
- void setUndef() : rend le nombre indéfini ;
- void sign(bool) : définit le signe du nombre (négatif si la booléenne est vraie, positif sinon) ;
- void reverse() : échange numérateur et dénominateur. Si le nombre est nul, il devient indéfini. S'il est infini, il devient nul. S'il est indéfini, il le reste.

Fonctions statiques

- static bool isValid(std::string, uint16_t = ILM_USERBASE) : renvoie si le std::string est un QNumber valide dans la base donnée (facultatif) ;
- static QNumber zero() : renvoie le nombre 0 dans le type QNumber ;
- static QNumber one() : renvoie le nombre 1 dans le type QNumber ;
- static QNumber two() : renvoie le nombre 2 dans le type QNumber.

Opérateurs/opérations arithmétiques

Ces opérations ne font finalement qu'appeler les opérations homologues de T et appliquer les règles avec les fractions. Opérateurs avec affectation :

- QNumber& operator+=(QNumber y) : ajoute y au nombre ;
- QNumber& operator-=(QNumber y) : soustrait y au nombre ;
- QNumber& operator*=(QNumber y) : multiplie le nombre par y ;
- QNumber& operator/=(QNumber y) : divise le nombre par y ;
- QNumber& operator++() : incrémentation préfixée ;
- QNumber& operator--() : décrémentation préfixée ;
- QNumber operator++(int) : incrémentation postfixée ;
- QNumber operator--(int) : décrémentation postfixée.

Opérations sans affectation :

- QNumber operator+(QNumber x, QNumber y) : renvoie $x + y$;
- QNumber operator-(QNumber x, QNumber y) : renvoie $x - y$;
- QNumber operator*(QNumber x, QNumber y) : renvoie xy ;
- QNumber operator/(QNumber x, QNumber y) : renvoie $x \div y$.
- QNumber operator-() : renvoie l'opposé du nombre.

En principe, les cas particuliers sont gérés par les opérateurs du type Z<NNumber>, et en tenant compte des règles avec les fractions (par exemple, $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$ est indéfini si les opérations intervenant dans le membre de droite donnent un nombre indéfini). Comme toujours, s'il y a au moins une opérande indéfinie, le résultat l'est également.

La puissance et la factorielle sont également disponibles, voir le chapitre des fonctions pour leur documentation.

La définition des constructeurs à partir de nombres standards permettent d'utiliser les opérateurs avec des QNumber et ces nombres (par exemple, l'addition d'un QNumber avec un double).

Opérateurs de comparaison

Comme pour les opérations, les opérateurs de comparaison sont une surcouche appelant les opérations homologues de QNumber. En effet, comparer $\frac{a}{b}$ et $\frac{c}{d}$ revient à comparer ad et bc, et donc ce sont les opérateurs de comparaison de Z<NNumber> qui sont responsables de la gestion des cas particuliers.

— bool operator==(QNumber x, QNumber y) : renvoie si $x = y$;
— bool operator!=(QNumber x, QNumber y) : renvoie si $x \neq y$;
— bool operator<(QNumber x, QNumber y) : renvoie si $x < y$;
— bool operator<=(QNumber x, QNumber y) : renvoie si $x \leq y$;
— bool operator>(QNumber x, QNumber y) : renvoie si $x > y$;
— bool operator>=(QNumber x, QNumber y) : renvoie si $x \geq y$.

Ici également, la définition des constructeurs à partir d'entiers standards permettent d'utiliser les opérateurs avec des QNumber et ces entiers mélangés.

Autres méthodes et fonctions associées

Autres méthodes :

— bool isZero() const : renvoie si le nombre est nul. Renvoie faux si le nombre est infini ou indéfini ;
— QNumber rand(uint32_t) const : renvoie un nombre aléatoire entre 0 et le nombre lui-même inclus. L'argument est le nombre de possibilités - 1 de nombres aléatoires, ces différents nombres étant séparés de manière équitable. Renvoie un nombre indéfini si le nombre est infini ou indéfini.

Fonctions extérieures à la classe :

— QNumber delta(QNumber x, QNumber y) : renvoie $|x - y|$;
— QNumber rand(QNumber, QNumber, uint32_t) : renvoie un nombre aléatoire entre les deux nombres donnés. Le troisième argument est le nombre de possibilités - 1 de nombres aléatoires, ces différents nombres étant séparés de manière équitable. Renvoie un nombre indéfini si le nombre est infini ou indéfini ;
— std::ostream& operator<<(std::ostream&, QNumber) : permet par exemple d'afficher un QNumber en utilisant std::cout comme s'il s'agissait d'un type standard.

2.7.3 Exemple

Celcul de e à l'aide de sa définition par les séries de Taylor.

```
1 ilm::QNumber e(0);
2 for (ilm::QNumber i(1) ; i < 50 ; i++) {
3     e += i/ilm::fac(i);
4 }
5 std::cout << "e = " << e << std::endl;
6 std::cout << "e = " << e.strApprox(50, ilm::SIGNIFICANT) << std::endl;
```

```
e = [Grosse fraction]
e = 2.7182818284590452353602874713526624977572470937
```

2.8 Nombres réels légers (RNumberL)

2.8.1 Description

Dans iloMath, les nombres réels ne sont pas gérés de manière formelle. À la place, l'utilisateur peut soit utiliser la gestion de manière exacte des nombres rationnels de tailles arbitraires avec `QNumber`, soit cette adaptation du type `long double` gérant de manière approximative mais rapide des nombres (pseudo-)réels : il s'agit de la classe `RNumberL`.

Il enregistre simplement les nombres dans un attribut de type `long double` ; comme pour `NNumberL`, il n'est pas possible non plus de manipuler des bases autres que 10.

2.8.2 Documentation de la classe

Attributs

Il n'y a qu'un attribut, celui qui stocke le nombre. Il est naturellement privé. La gestion des infinis et indéfinis ne se fait ici pas avec des booléennes, mais en utilisant la capacité de `long double` à les gérer.

— `long double _n` : stocke le nombre.

Constructeurs

— `RNumberL()` : constructeur par défaut, initialise le nombre à zéro ;
— `RNumberL(std::string)` : crée un `RNumberL` à partir d'un `std::string`. Ce dernier doit être dans une syntaxe similaire à celle utilisée pour initialiser un flottant habituel (par exemple, on peut entrer quelque chose comme `6.3` ou `12.345e-6`) ;
— `RNumberL(const char*)` : comme ci-dessus ; pour gérer les initialisations avec l'expression entre guillemets ;
— `RNumberL(S a)` avec `S` un type standard parmi `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `int8_t`, `int16_t`, `int32_t`, `int64_t`, `bool`, `float`, `double` et `long double` : crée un `RNumberL` à partir d'un nombre standard.

Accesseurs

— `long double n() const` : renvoie le `long double` contenant le nombre ;
— `bool isInf() const` : renvoie `std::isinf(_n)` ;
— `bool isUndef() const` : renvoie la valeur de `std::isnan(_n)` ;
— `bool sign() const` : renvoie si le nombre est négatif ;
— `bool isInt() const` : renvoie si le nombre est entier ;
— `T intPart() const` : renvoie la partie entière du nombre ;
— `long double ld() const` : renvoie le nombre sous la forme d'un long double ;
— `RNumberL re() const` : renvoie la partie réelle, soit le nombre lui-même ;
— `RNumberL im() const` : renvoie la partie imaginaire, soit le nombre 0 ;
— `std::string str() const` : renvoie le nombre exprimé tel que `_n` serait affiché, sauf s'il est indéfini (dans ce cas, affiche `undef`) ;
— `std::string strScientific(uint32_t s,) const` : renvoie le nombre exprimé en notation scientifique avec s chiffres significatifs sous la forme d'un string ;
— `std::string strApprox(uint32_t d, bool t, bool f) const` : renvoie le nombre approximé avec d chiffres significatifs/après la virgule si t est vraie/fausse, en gardant les zéros inutiles si f est vraie ;
— `Type type() const` : renvoie la valeur `ilm_RNumberL` de l'énumération des types.

Mutateurs

- `void reset()` : remet le nombre à zéro ;
- `void setInf()` : rend le nombre infini en affectant à `_n` la valeur `1.l/0.l` ;
- `void setUndef()` : rend le nombre indéfini en affectant à `_n` la valeur `std::nan("0")` ;
- `void sign(bool)` : définit le signe du nombre (négatif si la booléenne est vraie, positif sinon) ;
- `void floor()` : arrondit le nombre à l'unité vers le bas ;
- `void ceil()` : comme `floor`, mais vers le haut ;
- `void round()` : comme `floor` et `ceil`, mais à l'unité la plus proche ;
- `void abs()` : enlève l'éventuel signe du nombre ;
- `void reverse()` : inverse le nombre (`_n` devient `1.l/_n`).

Fonctions statiques

- `static bool isValid(std::string)` : renvoie si le `std::string` est un `RNumberL` valide ;
- `static RNumberL zero()` : renvoie le nombre 0 dans le type `RNumberL` ;
- `static RNumberL one()` : renvoie le nombre 1 dans le type `RNumberL` ;
- `static RNumberL two()` : renvoie le nombre 2 dans le type `RNumberL`.

Opérateurs/opérations arithmétiques

Opérateurs avec affectation (les opérations `+ - * / %` et incrémentations sont implémentées en utilisant les opérateurs standards entre des `long double`) :

- `RNumberL& operator+=(RNumberL y)` : ajoute y au nombre ;
- `RNumberL& operator-=(RNumberL y)` : soustrait y au nombre ;
- `RNumberL& operator*=(RNumberL y)` : multiplie le nombre par y ;
- `RNumberL& operator/=(RNumberL y)` : divise le nombre par y ;
- `RNumberL& operator++()` : incrémentation préfixée ;
- `RNumberL& operator--()` : décrémentation préfixée ;
- `RNumberL operator++(int)` : incrémentation postfixée ;
- `RNumberL operator--(int)` : décrémentation postfixée.

Opérations sans affectation :

- `RNumberL operator+(RNumberL x, RNumberL y)` : renvoie $x + y$;
- `RNumberL operator-(RNumberL x, RNumberL y)` : renvoie $x - y$;
- `RNumberL operator*(RNumberL x, RNumberL y)` : renvoie xy ;
- `RNumberL operator/(RNumberL x, RNumberL y)` : renvoie $x \div y$;
- `RNumberL operator-()` : renvoie l'opposé du nombre.

Comme les opérations se font avec les opérateurs et fonctions standards, les cas particuliers sont gérés par le type `long double`. La puissance et la factorielle sont également disponibles, voir le chapitre des fonctions pour leur documentation.

La définition des constructeurs à partir de nombres standards permettent d'utiliser les opérateurs avec des `RNumberL` et ces entiers mélangés (par exemple, l'addition d'un `RNumberL` avec un `int`).

Opérateurs de comparaison

Ils utilisent également directement la comparaison standard de deux `long double`. Voir la documentation de ce type pour en savoir plus sur les cas particuliers (on retrouvera par exemple la règle que si au moins un nombre est indéfini, la comparaison est toujours fausse).

— `bool operator==(RNumberL x, RNumberL y)` : renvoie si $x = y$;
— `bool operator!=(RNumberL x, RNumberL y)` : renvoie si $x \neq y$;
— `bool operator<(RNumberL x, RNumberL y)` : renvoie si $x < y$;
— `bool operator<=(RNumberL x, RNumberL y)` : renvoie si $x \leq y$;
— `bool operator>(RNumberL x, RNumberL y)` : renvoie si $x > y$;
— `bool operator>=(RNumberL x, RNumberL y)` : renvoie si $x \geq y$.

Ici également, la définition des constructeurs à partir d'entiers standards permettent d'utiliser les opérateurs avec des `RNumberL` et ces entiers mélangés.

Autres méthodes et fonctions associées

Autres méthodes :

— `bool isZero() const` : renvoie si le nombre est nul ;
— `RNumberL rand() const` : renvoie un nombre aléatoire entre 0 et le nombre lui-même inclus. Renvoie un nombre indéfini si le nombre est infini ou indéfini.

Fonctions extérieures à la classe :

— `RNumberL abs(RNumberL x)` : renvoie $|x|$;
— `RNumberL delta(RNumberL x, RNumberL y)` : renvoie $|x - y|$;
— `RNumberL rand(RNumberL, RNumberL)` : renvoie un nombre aléatoire entre les deux nombres donnés ;
— `RNumberL floor(RNumberL)` : voir la méthode homologue ;
— `RNumberL ceil(RNumberL)` : voir la méthode homologue ;
— `RNumberL round(RNumberL)` : voir la méthode homologue ;
— `std::ostream& operator<<(std::ostream&, RNumberL)` : permet par exemple d'afficher un `RNumberL` en utilisant `std::cout` comme s'il s'agissait d'un type standard.

2.8.3 Exemple

Génération de 50 nombres aléatoires entre 0 et 1.

```
1 ilm::RNumberL a(0), b(1), t, m;
2 for (uint32_t i(0) ; i < 50 ; i++) {
3     t = ilm::rand(a, b);
4     std::cout << t << " ";
5     m += t;
6 }
7 std::cout << std::endl << "M = " << m/50 << std::endl;
```

```
0.644866 0.0459583 0.419928 0.800837 0.664887 0.465803 0.596233 0.433505
    0.807783 0.838271 0.399345 0.836183 0.211681 0.46948 0.884415
    0.760906 0.731221 0.559812 0.977944 0.131404 0.292851 0.324607
    0.888915 0.109381 0.128667 0.424624 0.954446 0.331114 0.995305
    0.747099 0.966253 0.322781 0.777645 0.622828 0.379796 0.00130651
    0.889365 0.374967 0.609787 0.592289 0.481866 0.801516 0.168935
    0.128682 0.535287 0.324649 0.799962 0.323806 0.0450245 0.229694
M = 0.525078
```

2.9 Nombres complexes (C)

2.9.1 Description

Les nombres complexes sont stockés grâce au patron de classes `C<T>`, `T` étant un type de nombre rationnel ou réel compatible utilisé pour stocker les coefficients réel et imaginaire.

2.9.2 Documentation du patron de classes

On a T = ilm::QNumber ou RNumberL. Pour ce dernier, à chaque fois que la base est un paramètre, il faut l'ignorer. Alias :

- CQNumber = C<QNumber> ;
- CNumberL = C<RNumberL>

Le nombre est considéré comme indéfini si un des deux parties du nombre complexe est indéfinie. Il est considéré comme infini s'il n'est pas indéfini, et si l'une des deux parties est infinie (c'est donc un infini « ambigu », mais qui peut représenter *le* point à l'infini qu'on ajoute parfois au plan complexe pour par exemple avoir une bijection avec une sphère).

Attributs

Ces attributs sont privés.

- T _re : stocke la partie réelle ;
- T _im : stocke la partie imaginaire.

Constructeurs

- C() : constructeur par défaut, initialise les deux parties à zéro ;
- C(std::string, uint16_t = ILM_USERBASE) : crée un C à partir d'un std::string. Crée un nombre indéfini si cette chaine est invalide. La base dans laquelle le nombre est exprimé peut être fournie comme paramètre (facultatif) ;
- C(const char*, uint16_t = ILM_USERBASE) : comme ci-dessus ; pour gérer les initialisations avec l'expression entre guillemets ;
- C(S a, S b = 0) avec S un type standard parmi uint8_t, uint16_t, uint32_t, uint64_t, int8_t, int16_t, int32_t, int64_t, bool, float, double et long double : crée un C à partir de deux nombres standards (parties réelle et imaginaire ; pour les flottants, précision maximale : 16 chiffres significatifs) ;
- C(S a) avec S un type standard parmi std::complex<float>, std::complex<double> et std::complex<long double> : crée un C à partir d'un nombre réel standard (précision maximale : 16 chiffres significatifs).

Accesseurs

- T re() const : renvoie la partie réelle dans le type T ;
- T im() const : renvoie la partie imaginaire dans le type T ;
- bool isInf() const : renvoie si le nombre est infini ;
- bool isUndef() const : renvoie si le nombre est indéfini ;
- std::complex<long double> ld() const : renvoie le nombre sous la forme d'un std::complex<long double> ;
- std::string str(uint16_t = ILM_USERBASE) const : renvoie le nombre exprimé dans la base donnée (facultatif) sous la forme d'un std::string ;
- std::string strScientific(uint32_t s, uint16_t = ILM_USERBASE) const : renvoie le nombre avec les parties exprimées en notation scientifique avec s chiffres significatifs dans la base donnée (facultatif) sous la forme d'un string ;
- std::string strApprox(uint32_t d, bool t = DECIMAL, bool f = false, uint16_t = ILM_USERBASE) const : renvoie le nombre avec les parties approximées dans la base donnée (facultatif) avec d chiffres significatifs/après la virgule si t est vraie/fausse, en gardant les zéros inutiles si f est vraie ;
- Type type() const : renvoie la valeur ilm_CQNumber de l'énumération des types si T = ilm_QNumber et ilm_CNumberL sinon.

Mutateurs

- `void re(T)` : modifie la partie réelle ;
- `void im(T)` : modifie la partie imaginaire ;
- `void reset()` : remet le nombre à zéro ;
- `void setInf()` : rend le nombre infini en affectant aux deux parties une valeur infinie ;
- `void setUndef()` : rend le nombre indéfini ;
- `void sign(bool)` : rend le nombre indéfini en affectant aux deux parties une valeur indéfinie ;
- `void conjugate()` : conjugue le nombre en changeant le signe de la partie imaginaire ;
- `void reverse()` : inverse le nombre.

Fonctions statiques

- `static bool isValid(std::string, uint16_t = ILM_USERBASE)` : renvoie si le `std::string` est un `C` valide dans la base donnée (facultatif) ;
- `static C zero()` : renvoie le nombre 0 dans le type `C` ;
- `static C one()` : renvoie le nombre 1 dans le type `C` ;
- `static C two()` : renvoie le nombre 2 dans le type `C`.

Opérateurs/opérations arithmétiques

Ces opérations ne font finalement qu'appeler les opérations homologues de `T` et appliquer les règles avec les nombres complexes. Opérateurs avec affectation :

- `C& operator+=(C y)` : ajoute y au nombre ;
- `C& operator-=(C y)` : soustrait y au nombre ;
- `C& operator*=(C y)` : multiplie le nombre par y ;
- `C& operator/=(C y)` : divise le nombre par y ;
- `C& operator++()` : incrémentation préfixée ;
- `C& operator--()` : décrémentation préfixée ;
- `C operator++(int)` : incrémentation postfixée ;
- `C operator--(int)` : décrémentation postfixée.

Opérations sans affectation :

- `C operator+(C x, C y)` : renvoie $x + y$;
- `C operator-(C x, C y)` : renvoie $x - y$;
- `C operator*(C x, C y)` : renvoie xy ;
- `C operator/(C x, C y)` : renvoie $x \div y$.
- `C operator-()` : renvoie l'opposé du nombre.

En principe, les cas particuliers sont gérés par les opérateurs du type `T`, et en tenant compte des règles avec les nombres complexes. Comme toujours, s'il y a au moins une opérande indéfinie, le résultat l'est également. Pour l'addition et la soustraction, toute opération avec infini(s) donne désormais un nombre indéfini.

La définition des constructeurs à partir de nombres standards permettent d'utiliser les opérateurs avec des `C` et ces entiers mélangés (par exemple, l'addition d'un `C` avec un `double`).

Opérateurs de comparaison

Comme pour les opérations, les opérateurs de comparaison sont une surcouche appelant les opérations homologues de `T`. En effet, comparer $a + bi$ et $c + di$ revient à comparer a avec c et b avec d, et donc ce sont les opérateurs de comparaison de `T` qui sont responsables de la gestion des cas particuliers.

Comme il n'y a pas d'ordre total usuel dans $\mathbb{C}$, seuls les opérateurs d'égalité et d'inégalité sont implémentés.

— `bool operator==(C x, C y)` : renvoie si $x = y$;
— `bool operator!=(C x, C y)` : renvoie si $x \neq y$.

Ici également, la définition des constructeurs à partir d'entiers standards permettent d'utiliser les opérateurs avec des `C` et ces entiers mélangés.

Autres méthodes et fonctions associées

Autres méthodes :

— `bool isZero() const` : renvoie si le nombre est nul. Renvoie faux si le nombre est infini ou indéfini.

Fonctions extérieures à la classe :

— `std::ostream& operator<<(std::ostream&, C)` : permet par exemple d'afficher un `C` en utilisant `std::cout` comme s'il s'agissait d'un type standard.

2.9.3 Exemple

Affichage d'un nombre complexe en notation scientifique.

```
1 ilm::CQNumber a("1/2 + 3/4i"), n("-567"), r(ilm::pow(a, n));
2 std::cout << "a^n = " << r.strScientific(6) << std::endl;
```

```
a^n = -1.39203 x 10^25 + (3.39949 x 10^25)i
```

3. Fonctions

3.1 Généralités

Une autre fonction clé d'iloMath est la gestion de fonctions mathématiques usuelles. Par cette phrase, on entend d'une part une implémentation de fonctions simples comme sin, exp,... dans la `iloMath` qui prennent en paramètre un ou plusieurs nombre(s) d'un type parmi ceux qui ont été présentés dans le chapitre précédent et qui renvoient l'image de ces nombres, et d'autre part la possibilité de saisir des fonctions comme $\log(x^2) + 1$, les stocker, et les évaluer en des points.

Ce chapitre a pour but de documenter ces deux notions de fonctions mathématiques présentes dans `iloMath`. Une section listera et décrira les fonctions disponibles pour le développeur, et une autre documentera le patron de classes `Function<T>` qui s'occupe d'interpréter des saisies de fonctions, de les stocker, et de les évaluer.

Actuellement, les fonctions supportées sont les sommes, différences, produits, divisions, puissances, et composition de monômes, polynômes, valeur absolue et module, signe, l'exponentielle, logarithme, sinus, cosinus, tangente, arcsinus, arccosinus, arctangente, gamma et factorielle.

Exemples $x^3 + 1$, $\log(x)$, $x^2 + y^2$, $e^{\tan(x)} = \exp(\tan(x))$, x^x et $\Gamma(x) = (x-1)!$ sont supportés. $erf(x)$, $\int e^{-t^2} dt$, ($x = 1$ si $x > 1$, $x = 2$ sinon) ne le sont pas.

L'évaluation de fonctions contenant les fonctions comme l'exponentielle ou trigonométriques n'est actuellement pleinement possible qu'avec `RNumberL` ou `C<RNumberL>`. Pour `QNumber` et `C<QNumber>`, les implémentations sont pour le moment en général incomplètes ou difficilement utilisables, mis à part pour certains cas spécifiques. Il vaut donc mieux d'éviter de les utiliser pour le moment (mais, il est toutefois possible d'utiliser à la place des évaluations sans précision arbitraire pour ces types).

3.2 Liste des fonctions mathématiques de la bibliothèque

Pour les opérations élémentaires comme l'addition, voir la documentation des classes pour les nombres. Pour toutes ces fonctions, si au moins une des opérandes est indéfinie, le résultat l'est également.

3.2.1 Fonctions pour NNumber(L)

Ces fonctions existent pour les deux classes et s'utilisent de la même manière.

- `NNumber(L) sgn(NNumber(L) x)` : renvoie $sgn(x)$, soit 0 si x est nul et 1 sinon ;
- `NNumber(L) abs(NNumber(L) x)` : renvoie $abs(x)$, soit le nombre lui-même ;
- `NNumber(L) pow(NNumber(L) x, NNumber(L) y)` : renvoie x^y. Les cas particuliers sont traités dans cet ordre :
  - $x^0 = 1$ (y compris si $x = 0$ ou $x = \infty$) ;
  - $0^y = 0$ (y compris si $y = \infty$) ;
  - $1^y = 1$ (y compris si $y = \infty$) ;
  - $x^y = \infty$ si $x = \infty$ ou $y = \infty$;

— `NNumber(L) fac(NNumber(L) x)` : renvoie $x!$. On a $0! = 1$, $\infty! = \infty$;
— `NNumber(L) gcd(NNumber(L) x, NNumber(L) y)` : renvoie le plus grand diviseur commun (PGDC/greatest common divisor) de x et y. Si au moins un nombre est infini, le $PGDC$ est un nombre indéfini. Sinon, si un nombre est nul, le PGDC est l'autre nombre (même s'il vaut également 0) ;
— `NNumber(L) lcm(NNumber(L) x, NNumber(L) y)` : renvoie le plus petit multiple commun (PPMC/least common multiple) de x et y. Si au moins un nombre est infini, ou si un des deux nombres est nul, le $PPMC$ est un nombre indéfini. Sinon, si les deux nombres sont nuls, le PPMC également.

Aucun contrôle de débordement n'est effectué pour `NNumberL`, sauf pour la factorielle si $x > 20$ (dans ce cas, le résultat est un nombre infini).

3.2.2 Fonctions pour Z

— `Z<T> sgn(Z<T> x)` : renvoie $sgn(x)$, soit -1 si x est négatif, 0 si x est nul et 1 sinon ;
— `Z<T> abs(Z<T> x)` : renvoie $abs(x)$, soit x sans son signe ($-\infty$ devient ∞) ;
— `Z<T> pow(Z<T> x, Z<T> y)` : renvoie x^y. Les cas particuliers sont traités dans cet ordre :
  — Le résultat est indéfini si ($x < 0$ et $y = +\infty$), si ($|x| \neq 1$, $x \neq \pm\infty$, $y < 0$ et $y \neq -\infty$), ou (si $x = 0$ et $x = -\infty$) ;
  — $x^0 = 1$ (y compris si $x = 0$ ou $x = \pm\infty$) ;
  — $0^y = 0$ (y compris si $y = +\infty$) ;
  — $1^y = 1$ (y compris si $y = \pm\infty$) ;
  — $(-1)^{\pm\infty}$ est indéfini ; $(-1)^y$ vaut -1 si est impair et 1 sinon (y compris si $y < 0$) ;
  — $x^{-\infty} = 0$ (y compris si $x = \pm\infty$) ;
  — $(\pm\infty)^y = 0$ si $y < 0$ (y compris si $y = -\infty$) ;
  — $x^y = +\infty$ si $x = +\infty$ ou $y = +\infty$.
— `Z<T> fac(Z<T> x)` : renvoie $x!$. Le résultat est indéfini si x est négatif. Sinon, la fonction se comporte comme pour le type `T`.

3.2.3 Fonctions pour QNumber

Ces fonctions acceptent un argument facultatif de type `uint16_t` qui est le nombre d'itérations demandés pour le calcul (par exemple, nombre de termes de Taylor ou d'itérations pour la méthode de Newton). Plus ce paramètre est grand, plus le résultat est précis. Si ce nombre est 0, la fonction va convertir le ou les nombre(s) en `long double`, utiliser la bibliothèque standard et reconvertir le résultat en `QNumber`, au lieu d'utiliser une méthode implémentée (mais, les cas particuliers sont toujours traités avant).

L'argument de type `bool` n'est valable que si le nombre d'itérations n'est pas 0 et indique s'il faut supprimer des chiffres non significatifs entre chaque itération, ce qui peut faire gagner beaucoup de temps (mais pour le moment, cela n'a été implémenté que pour les puissances).

Ces paramètres sont facultatifs et valent respectivement `ILM_DEFAULTITERS` et `true` par défaut. Pour les fonctions dont les résultats exacts se calculent facilement, ces paramètres peuvent être passés, mais n'ont aucune influence et sont omis dans la liste ci-dessous : `sgn`, `abs`, `inv`, et `arg`.

En pratique, les algorithmes ne sont pas encore au point (vitesse, convergence) et il est déconseillé de les utiliser pour le moment, en indiquant 0 comme nombre d'itérations (la valeur par défaut de `ILM_DEFAULTITERS`). Certaines fonctions n'ont d'ailleurs pas encore leur propre implémentation et vont faire appel à la bibliothèque standard, quelles qu'en soient les paramètres. Ceci sera précisé si c'est le cas. Il est prévu d'améliorer et de compléter les implémentations existantes dans des versions futures d'`iloMath`.

- `QNumber sgn(QNumber x)` : renvoie $sgn(x)$, soit -1 si x est négatif, 0 si x est nul et 1 sinon ;
- `QNumber abs(QNumber x)` : renvoie $|x|$, soit x sans son signe ($-\infty$ devient ∞) ;
- `QNumber inv(QNumber x)` : renvoie $\frac{1}{x}$. Le résultat est indéfini si $x = 0$;
- `QNumber pow(QNumber x, QNumber y, uint16_t, bool)` : renvoie x^y. Les cas particuliers sont traités dans cet ordre :
  - Le résultat est indéfini si $x < 0$ et $y = +\infty$, ou si $x = 0$ et $y < 0$;
  - $x^0 = 1$ (y compris si $x = 0$ ou $x = \pm\infty$) ;
  - $0^y = 0$ (y compris si $y = +\infty$) ;
  - $1^y = 1$ (y compris si $y = \pm\infty$) ;
  - $(-1)^{\pm\infty}$ est indéfini ;
  - $x^{+\infty} = +\infty$ si $|x| > 1$ (y compris si $x = +\infty$) et 0 sinon, et inversément si $y = -\infty$. De plus, si $y = -\infty$ et $x \in [-1; 0]$, le résultat est indéfini ;
  - $(\pm\infty)^y = 0$ si $y < 0$ (y compris si $y = -\infty$) ;
  - $(+\infty)^y = +\infty$;
  - Si y est un entier, élève le numérateur et le dénominateur à la puissance $|y|$, après les avoir échangés si $y < 0$;
  - Sinon,
    - Si x est négatif, et le dénominateur de y pair, le résulat est indéfini ;
    - Sinon, la méthode de Newton est utilisée pour trouver la puissance, avec $(-x)^y = -x^y$ ($x > 0$). Si le nombre d'itérations demandé est 0, la fonction standard `std::pow` est utilisée ici à la place de la méthode de Newton.
- `QNumber fac(QNumber x, uint16_t, bool)` : renvoie $x!$. Si x est entier, la valeur est la même que celle renvoyée par la fonction homologue pour `Z<NNumber>`. Sinon, la fonction standard `std::tgamma(x + 1)` est utilisée à la place (pas encore d'implémentation spécifique pour le moment). Les deux derniers paramètres n'ont donc pour le moment aucune influence ;
- `QNumber gamma(QNumber x, uint16_t, bool)` : appelle `fac` avec $x - 1$;
- `QNumber exp(QNumber x, uint16_t, bool)` : renvoie e^x en utilisant sa définition par les séries de Taylor. Cas particuliers : si $e^{-\infty} = 0$ et $e^{+\infty} = +\infty$;
- `QNumber log(QNumber x, uint16_t, bool)` : renvoie $\log(x)$ en utilisant les séries de Taylor. Cas particuliers : le résultat est indéfini si $x < 0$ et $\log(0) = -\infty$;
- `QNumber sin(QNumber x, uint16_t, bool)` : renvoie $\sin(x)$ en utilisant les séries de Taylor. Cas particuliers : le résultat est indéfini si $x = \pm\infty$;
- `QNumber cos(QNumber x, uint16_t, bool)` : renvoie $\cos(x)$ en utilisant les séries de Taylor. Cas particuliers : le résultat est indéfini si $x = \pm\infty$;
- `QNumber tan(QNumber x, uint16_t, bool)` : renvoie $\tan(x)$ en utilisant sa définition possible $\tan(x) = \frac{\sin(x)}{\cos(x)}$. Aucun cas particulier n'est traité directement. Néanmoins, l'appel des fonctions `sin` et `cos` implique que le résultat est indéfini si $x = \pm\infty$ ou si $\cos(x) = 0$;
- `QNumber arcsin(QNumber x, uint16_t, bool)` : renvoie $\arcsin(x)$ en utilisant la fonction standard `std::asin(x)`. (pas encore d'implémentation spécifique pour le moment). Les deux derniers paramètres n'ont donc pour le moment aucune influence et pas de cas particulier n'a été programmé pour le moment (aucun contrôle effectué) ;
- `QNumber arccos(QNumber x, uint16_t, bool)` : renvoie $\arccos(x)$ en utilisant la fonction standard `std::acos(x)`. (pas encore d'implémentation spécifique pour le moment). Les deux derniers paramètres n'ont donc pour le moment aucune influence et pas de cas particulier n'a été programmé pour le moment (aucun contrôle effectué) ;
- `QNumber arctan(QNumber x, uint16_t, bool)` : renvoie $\arctan(x)$ en utilisant la fonction standard `std::atan(x)`. (pas encore d'implémentation spécifique pour le moment). Les deux derniers paramètres n'ont donc pour le moment aucune influence et pas de cas particulier n'a été programmé pour le moment (aucun contrôle effectué).

3.2.4 Fonctions pour RNumberL

Les mêmes fonctions que pour `QNumber` sont présentes et ont le même rôle. Cependant, pour des raisons de performance, aucun cas particulier n'est traité, et ils utilisent la bibliothèque standard (sauf `sgn`, `abs`, `inv`, et `arg` qui ont une implémentation comme celles ci-dessus). `gamma` utilise `std::tgamma` et `fac` utilise `std::tgamma(x + 1)`.

Les arguments pour le nombre d'itérations et la suppression des chiffres non significatifs entre les itérations sont également présents pour des raisons d'homogénéité, mais sont également facultatifs et n'ont aucune influence sur le résultat donné.

3.2.5 Fonctions pour C

Ici encore, les fonctions sont les mêmes que pour `QNumber`. Concernant la version `C<RNumberL>` du patron de classes, comme pour `RNumberL`, les fonctions utilisent la bibliothèque standard (cette fois toutes sauf `sgn` qui est implémentée comme ci-dessous, `inv`, et `gamma` du fait que `std::tgamma` n'existe pas pour les nombres complexes standards), et on fera les mêmes remarques que leurs homologues pour `RNumberL`.

On précisera que `abs` et `arg` utilisent les fonctions standards `std::abs` et `std::arg`, et que les nombres complexes standards sont ici `std::complex<long double>`.

`C<RNumberL> gamma(C<RNumberL> x)` est implémentée en utilisant l'algorithme de Lanczos, et donne un résultat à peu près en double précision.

Concernant `C<QNumber>`, on fera également les mêmes remarques que pour `QNumber`. Liste des fonctions :

- `C<T> sgn(C<T> x)` : renvoie $sgn(x) = \frac{x}{|x|}$, le résultat est indéfini si x est nul ou infini ;
- `C<QNumber> abs(C<QNumber> x)` : renvoie $|x|$, le module du nombre complexe, soit $\sqrt{a^2 + b^2}$ si $x = a + bi$. Si x est infini, $|x|$ également ;
- `C<T> inv(C<T> x)` : renvoie $\frac{1}{x}$. Le résultat est indéfini si $x = 0$;
- `C<QNumber> pow(C<QNumber> x, C<QNumber> y, uint16_t, bool)` : renvoie x^y. Les cas particuliers sont traités dans cet ordre :
  - Le résultat est indéfini si x ou y sont infinis ;
  - $x^0 = 1$ (y compris si $x = 0$) ;
  - $0^y = 0$ (sauf si y est un réel négatif, dans ce cas, le résultat est indéfini) ;
  - $1^y = 1$;
  - Si y est un entier, élève le numérateur et le dénominateur à la puissance $|y|$, après les avoir échangés si $y < 0$;
  - Sinon,
    - Si x et y, sont des nombres réels, renvoie le nombre tel qu'il est calculé par la fonction `pow` homologue de `QNumber` ;
    - Sinon, si l'exposant est entier, le calcul exact est effectué ;
    - Sinon, utilise la fonction standard `std::pow` (pas encore d'implémentation spécifique pour le moment).
- `C<QNumber> gamma(C<QNumber> x, uint16_t, bool)` : renvoie $x!$. Le résultat est infini si x l'est. Si x est réel, la valeur est la même que celle renvoyée par la fonction homologue pour `QNumber`. Sinon, la fonction homologue pour `C<RNumberL>` est utilisée à la place (pas encore d'implémentation arbitrairement précise pour le moment). Les deux derniers paramètres n'ont donc pour le moment aucune influence ;
- `C<T> fac(C<T> x, uint16_t, bool)` : appelle `gamma` avec $x + 1$;
- `C<QNumber> exp(C<QNumber> x, uint16_t, bool)` : renvoie e^x en utilisant sa définition par les séries de Taylor. Le résultat est indéfini si x est infini ;
- `C<QNumber> log(C<QNumber> x, uint16_t, bool)` : renvoie $\log(x)$ en utilisant les séries de Taylor. Cas particuliers : le résultat est infini si x est nul ou infini ;

— `C<QNumber> sin(C<QNumber> x, uint16_t, bool)` : renvoie $\sin(x)$ en utilisant les séries de Taylor. Cas particuliers : le résultat est indéfini si x est infini ;
— `C<QNumber> cos(C<QNumber> x, uint16_t, bool)` : renvoie $\cos(x)$ en utilisant les séries de Taylor. Cas particuliers : le résultat est indéfini si x est infini ;
— `C<QNumber> tan(C<QNumber> x, uint16_t, bool)` : renvoie $\tan(x)$ en utilisant sa définition possible $\tan(x) = \frac{\sin(x)}{\cos(x)}$. Aucun cas particulier n'est traité directement. Néanmoins, l'appel des fonctions `sin` et `cos` implique que le résultat est indéfini si x est infini et infini si $\cos(x) = 0$;
— `C<QNumber> arcsin(C<QNumber> x, uint16_t, bool)` : renvoie $\arcsin(x)$ en utilisant la fonction standard `std::asin(x)`. (pas encore d'implémentation spécifique pour le moment). Les deux derniers paramètres n'ont donc pour le moment aucune influence et pas de cas particulier n'a été programmé pour le moment (aucun contrôle effectué) ;
— `C<QNumber> arccos(C<QNumber> x, uint16_t, bool)` : renvoie $\arccos(x)$ en utilisant la fonction standard `std::acos(x)`. (pas encore d'implémentation spécifique pour le moment). Les deux derniers paramètres n'ont donc pour le moment aucune influence et pas de cas particulier n'a été programmé pour le moment (aucun contrôle effectué) ;
— `C<QNumber> arctan(C<QNumber> x, uint16_t, bool)` : renvoie $\arctan(x)$ en utilisant la fonction standard `std::atan(x)`. (pas encore d'implémentation spécifique pour le moment). Les deux derniers paramètres n'ont donc pour le moment aucune influence et pas de cas particulier n'a été programmé pour le moment (aucun contrôle effectué).

3.2.6 Exemples

```
1 ilm::QNumber  a("2"), n("1/2");
2 ilm::CNumberL x("1 + i");
3 std::cout << "a^n      = " << ilm::pow(a, n, 16).strApprox(32) << std::
    endl;
4 std::cout << "Gamma(x) = " << ilm::gamma(x) << std::endl;
```

```
a^n      = 1.41421356237309504880168872421
Gamma(x) = 0.498016 - 0.15495i
```

3.3 Documentation du patron de classes Function

3.3.1 Description

`Function<T>` est le patron de classes s'occupant de la manipulation (saisie, stockage et évaluation) de fonctions à nombre de variables de type compatible arbitraire. `T` doit être un type compatible : `QNumber`, `CQNumber`, `RNumberL` ou `CNumberL`.

Alias :

— `QFunction  = Function<QNumber>` ;
— `CQFunction = Function<C<QNumber>>` ;
— `RFunctionL = Function<RNumberL>` ;
— `CFunctionL = Function<C<RNumberL>>`.

Cette classe peut aussi faire office de gestion d'expressions, par exemple pour pouvoir stocker certains nombres réels comme $1 + 2^{\frac{1}{2}}$ ou i^i ou effectuer toutes sortes de calculs numériques.

Pour le moment, on ne peut que définir les fonctions et les évaluer. On ne peut pas encore effectuer d'opérations entre elles, ou les dériver. Ce patron de classes sera complété et amélioré dans les versions futures d'iloMath.

Syntaxe et exemples d'interprétation

Les règles pour la saisie d'une fonction. Dessiner un diagramme de syntaxe serait très compliqué, voici donc les règles générales de syntaxe pour entrer une fonction valide :

- Les espaces sont ignorées ;
- Les caractères de `0` à `9`, de `A` à `F`, et les deux points `:` désignent des chiffres ou des espaces entre les chiffres. L'ensemble des caractères des chiffres dépend de la base dans laquelle on travaille ;
- `i` symbolise toujours l'unité imaginaire ;
- Les variables sont les lettres de `E` à `Z` et de `a` à `z`, sauf `i` ;
- Les fonctions élémentaires particulières sont désignées par `sgn`, `abs`, `exp`, `log`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `gam` (gamma ; `fac` peut aussi être utilisé pour la factorielle) ;
- Les opérations disponibles par `+ - * / ^ !` ;
- Des parenthèses peuvent être utilisées ;
- Si on trouve une fonction élémentaire particulière quelque part, elle doit nécessairement être suivie par son argument entre parenthèses ;
- Il y a quelques autres règles de bon sens, par exemple, deux opérations ne peuvent pas se suivre sauf dans certains cas (par exemple, $x!! = (x!)!$), ou encore, les parenthèses doivent se fermer correctement.

Voici quelques exemples de saisies et d'interprétations :

- `log(x`, `1 ++ 2`, `3^-1`, `sin x` et `3^((2 + 1)` sont des saisies invalides ;
- `3 ^ ( -  1 )` donne la fonction constante 3^{-1} ;
- `exp(1)` donne la fonction constante $e^1 = e$;
- `sin(tan(x))` donne la fonction à une variable $\sin(\tan(x))$;
- `(x^2 + y^2)/r^2` donne la fonction à trois variables $\frac{x^2+y^2}{r^2}$
- `erf(x)` donne la fonction à quatre variables $erf(x) = erfx$;
- `xlog(y)` donne la fonction à deux variables $x \log(y)$;

3.3.2 Méthodes et fonctions associées

Constructeurs

- `Function()` : constructeur par défaut, crée la fonction constante 0 ;
- `Function(std::string, uint16_t = ILM_USERBASE)` : crée un `Function` à partir d'un `std::string`. Crée une fonction indéfinie si cette chaine est invalide. La base dans laquelle les nombres sont exprimés peut être fournie comme paramètre (facultatif) ;
- `Function(const char*, uint16_t = ILM_USERBASE)` : comme ci-dessus ; pour gérer les initialisations avec l'expression entre guillemets.

Accesseurs

- `std::string str(uint16_t = ILM_USERBASE) const` : renvoie la fonction avec les nombres exprimés dans la base donnée (facultatif) sous la forme d'un `std::string` ;
- `std::string strScientific(uint32_t s, uint16_t = ILM_USERBASE) const` : renvoie la fonction avec les nombres exprimés en notation scientifique avec s chiffres significatifs dans la base donnée (facultatif) sous la forme d'un string ;
- `std::string strApprox(uint32_t d, bool t = DECIMAL, bool f = false, uint16_t = ILM_USERBASE) const` : renvoie la fonction avec les nombres approximées dans la base donnée (facultatif) avec d chiffres significatifs/après la virgule si t est vraie/fausse, en gardant les zéros inutiles si f est vraie ;

— `T n() const` : si la fonction est constante, renvoie le nombre correspondant. Sinon, renvoie un nombre indéfini ;
— `std::vector<std::string> vars()` : renvoie un `std::vector` contenant toutes les variables sous la forme d'un `std::string` de la fonction ;
— `uint16_t nVars() const` : renvoie le nombre de variables de la fonction ;
— `bool isNumeric() const` : renvoie si la fonction (l'expression) est numérique, c'est-à-dire s'il n'y a aucune variable.

Mutateurs

— `void eval(T x = T::zero(), std::string var = "x", uint16_t = ILM_DEFAULTITERS, bool = true)` : évalue la fonction en remplaçant les variables représentées par `var` par `x`. Les deux derniers paramètres spécifient la précision du résultat en les passant aux fonctions documentées dans la section précédante.

Fonctions statiques

— `static T eval(Function f, T x, std::string var, uint16_t iters = ILM_DEFAULTITERS, bool shr = true)` : renvoie l'évaluation de f, voir le mutateur homologue.

Autres méthodes et fonctions associées

— `std::ostream& operator<<(std::ostream&, Function)` : permet par exemple d'afficher un `Function` en utilisant `std::cout` comme s'il s'agissait d'un type standard.

3.3.3 Exemples

Création et affichage de fonctions. Une saisie invalide donne une fonction indéfinie.

```
1  ilm::CQFunction f("(x^(log(exp(sin(x + 1)*tan(x))/(x*cos(x + cos(x))))^2)
       /(8x^4 + 3x^3 - x/2 + 1))^(1/4)"), g, h("erf(x)"), i("3^((2 + 1)");
2  g = "exp(x)";
3  std::cout << "f = " << f << std::endl;
4  std::cout << "g = " << g << std::endl;
5  std::cout << "h = " << h << std::endl;
6  std::cout << "i = " << i << std::endl;
```

```
f = (x^(log(exp(sin(x + 1)*tan(x))/(x*cos(x + cos(x))))^2)/(8*x^4 + 3*x^3
    - x/2 + 1))^1/4
g = exp(x)
h = e*r*f*x
i = undef
```

Idem, pour plusieurs bases.

```
1  ilm::CQFunction f("2x + 1"), g("1:0:0cos(x)^2:0", 64), h("x + 10y + 11z",
       2);
2  std::cout << "f = " << f << std::endl;
3  std::cout << "g = " << g << std::endl;
4  std::cout << "h = " << h << std::endl;
5  std::cout << "En base 2 : " << std::endl;
6  std::cout << "\tf = " << f.str(2) << std::endl;
7  std::cout << "\tg = " << g.str(2) << std::endl;
8  std::cout << "\th = " << h.str(2) << std::endl;
9  std::cout << "En base 10 : " << std::endl;
10 std::cout << "\tf = " << f << std::endl;
```

```
11 std::cout << "\tg = " << g.str() << std::endl;
12 std::cout << "\th = " << h.str(10) << std::endl;
13 std::cout << "En base 16 : " << std::endl;
14 std::cout << "\tf = " << f.str(16) << std::endl;
15 std::cout << "\tg = " << g.str(16) << std::endl;
16 std::cout << "\th = " << h.str(16) << std::endl;
```

```
f = 2*x + 1
g = 4096*cos(x)^128
h = x + 2*y + 3*z
En base 2 :
    f = 10*x + 1
    g = 1000000000000*cos(x)^10000000
    h = x + 10*y + 11*z
En base 10 :
    f = 2*x + 1
    g = 4096*cos(x)^128
    h = x + 2*y + 3*z
En base 16 :
    f = 2*x + 1
    g = 1000*cos(x)^80
    h = x + 2*y + 3*z
```

Utilisation de `strApprox` et `strScientific`.

```
1 ilm::QFunction f("0.5x + 2/3");
2 std::cout << "f = " << f.strApprox(3, ilm::SIGNIFICANT, true) << std::
      endl;
3 std::cout << "f = " << f.strScientific(3) << std::endl;
```

```
f = 0.500*x + 2.00/3.00
f = (5.00 x 10^(-1))*x + (2.00 x 10^0)/(3.00 x 10^0)
```

Informations sur la fonction.

```
1 ilm::CFunctionL f("sin(x^2 + y^2)cos(tan(z)^2)");
2 std::cout << "f est une fonction à " << f.nVars() << " variables : " << f
      .vars()[0];
3 for(uint32_t i(1) ; i < f.nVars() ; i++)
4     std::cout << " ; " << f.vars()[i];
```

```
f est une fonction à 3 variables : x ; y ; z
```

Évaluation d'un gros polynôme et d'une expression numérique.

```
1 ilm::CQFunction f("x^3y^2z - x^2y^2z + 4xy^2z - 3y^2z - x^3yz + 4xyz + 2x
      ^3z + 2x^2z - 2xz + 4z - 5x^3y^2 - 2x^2y^2 - 4xy^2 + 4x^3y + 4x^2y -
      3xy + x^3 + 3x^2 + x - 3");
2
3 std::cout << "f(x, y, z)   = " << f << std::endl;
4 f.eval("11", "x");
5 std::cout << "f(11, y, z)  = " << f << std::endl;
6 f.eval("-7", "y");
7 std::cout << "f(11, -7, z) = " << f << std::endl;
8 f.eval("5", "z");
9 std::cout << "f(11, -7, 5) = " << f << std::endl;
```

```
f(x, y, z)   = x^3*y^2*z - x^2*y^2*z + 4*x*y^2*z - 3*y^2*z - x^3*y*z + 4*
    x*y*z + 2*x^3*z + 2*x^2*z - 2*x*z + 4*z - 5*x^3*y^2 - 2*x^2*y^2 - 4*x
    *y^2 + 4*x^3*y + 4*x^2*y - 3*x*y + x^3 + 3*x^2 + x - 3
f(11, y, z)  = 1331*y^2*z - 121*y^2*z + 44*y^2*z - 3*y^2*z - 1331*y*z +
    44*y*z + 2662*z + 242*z - 22*z + 4*z - 6655*y^2 - 242*y^2 - 44*y^2 +
    5324*y + 484*y - 33*y + 1702
```

```
f(11, -7, z) = 65219*z - 5929*z + 2156*z - 147*z - -9317*z + -308*z +
    2662*z + 242*z - 22*z + 4*z + -378832
f(11, -7, 5) = -12862
```

```
1 ilm::CQFunction ne("(2^16/3^9 - 5^7*11)^2/13 + 17");
2 std::cout << "ne = " << ne << std::endl;
3 ne.eval();
4 std::cout << "    = " << ne << std::endl;
5 std::cout << "    = " << ne.strScientific(10) << std::endl;
```

```
ne = (2^16/3^9 - 5^7*11)^2/13 + 17
   = 286117650971648410990/5036466357
   = 5.680920524 x 10^10
```

4. Tenseurs, intervalles, représentation graphique

4.1 Tenseurs (Tensor)

4.1.1 Description

Ce patron de classes `Tensor<T>` (avec `T` une classe de nombres d'iloMath) permet de stocker des tenseurs d'ordre quelconque, soit des tableaux de dimensions arbitraires. Il stocke en fait le tenseur dans un `std::vector<T>` de dimension 1. Des conversions d'indices transparentes pour l'utilisateur sont faites afin qu'il ait l'impression de manipuler un véritable tenseur. Les dimensions du tenseur sont stockés dans un autre `std::vector`.

La conversion d'un indice n d'un élément du tableau dynamique vers la m-ième coordonnée x_m de cet élément, pour un tenseur d'ordre o, de dimensions $d_1 \times d_2 \times d_3 \times ... \times d_o$ est donnée par :

$$x_m = \lfloor \frac{n}{\prod_{i=1}^{m-1} d_i} \rfloor \mod d_m$$

Dans l'autre sens, un élément donné du tenseur aura informatiquement l'indice donné par :

$$n = \sum_{i=1}^{o} (x_i \prod_{j=1}^{i-1} d_j)$$

n est l'indice, o est l'ordre du tenseur, x_i une coordonnée, d_j une dimension du tenseur.

À noter que tous les indices et coordonnés s'inspirent du système du C++, et donc commencent toujours par 0 !

Contrairement aux autres classes, il n'est pas possible de créer un tenseur à partir d'un `std::string`. On doit l'initialiser avec un `vector<uint32_t>` contenant ses dimensions. En procédant ainsi, le tenseur est créé avec les dimensions renseignées et toutes ses composantes nulles.

Par contre, on peut afficher un `Tensor` comme d'habitude, à l'aide de l'opérateur de flux `<<`. Les tenseurs d'ordre 1 s'affichent en une colonne de composantes, ceux d'ordre 2 en une grille de composantes, et les autres sous forme décomposées en matrices.

L'utilisateur remplira ensuite le tenseur manuellement avec les mutateurs.

Les fonctionnalités sont pour le moment très élémentaires, mais seront complétées dans des versions futures d'`iloMath`. Il a surtout été créé comme outil pour la représentation graphique, et historiquement pour le stockage de polynômes.

4.1.2 Documentation du patron de classes

On a `T = QNumber`, `RNumberL`, ou leurs versions complexes. Alias :

— `QTensor  = Tensor<QNumber>`;

— CQTensor = Tensor<C<QNumber>>
— RTensorL = Tensor<RNumberL> ;
— CTensorL = Tensor<C<RNumberL>>

Les types T = NNumber(L) et T = Z<NNumber(L)> sont également supportés.

Attributs

Ces attributs sont privés.

— std::vector<T> _coeffs : stocke les coefficients du tenseur ;
— std::vector<uint32_t> _dimensions : stocke les dimensions du tenseur.

Constructeurs

— Tensor() : constructeur par défaut, crée un tenseur d'ordre 1 à une composante nulle ;
— Tensor(std::vector<uint32_t>) : créer un tenseur avec dimensions définies selon le std::vector et toutes les composantes nulles ;

Accesseurs

— uint32_t size() const : renvoie le nombre de composantes du tenseur ;
— std::vector<uint32_t> dims() const : renvoie les dimensions du tenseur ;
— uint32_t order() : renvoie l'ordre du tenseur ;
— T comp(uint32_t pos) const : renvoie la composante située à l'indice mentionnée dans le std::vector. Si l'indice est invalide, un nombre indéfini est renvoyé ;
— T comp(std::vector<uint32_t>) const : renvoie la composante située aux coordonnées mentionnées. Si la coordonnée est invalide, un nombre indéfini est renvoyé ;
— std::string str(uint16_t = ILM_USERBASE) const : renvoie le tenseur avec les composantes exprimées dans la base donnée (facultatif) sous la forme d'un std::string.

Mutateurs

— void comp(T c0, uint32_t pos) : modifie la composante à l'indice donnée dans le std::vector. Si l'indice est invalide, rien ne se passe ;
— void comp(T c0, std::vector<uint32_t> pos) : modifie la composante aux coordonnées données. Si les coordonnées sont invalides, rien ne se passe ;
— void addLine(uint32_t) : ajoute une « ligne » à la fin pour une dimension donnée (autrement dit, augmente cette dimension de 1). Si la dimension est supérieure à l'ordre, voir la remarque ci-dessous ;
— void delLine(uint32_t) : retire la « ligne » à la fin pour une dimension donnée (autrement dit, diminue cette dimension de 1). Si la dimension devient 0, ou si on diminue une dimension supérieure à l'ordre, le tenseur devient tel qu'il est après l'appel de reset ;
— void resize(std::vector<uint32_t>)) : redimensionne le tenseur en spécifiant ses nouvelles dimensions. Les dimensions qui augmentent auront des nouvelles composantes nulles ;
— void reset() : le tenseur devient d'ordre 1 à une composante nulle.

À noter que contrairement aux indices, les dimensions commencent à 1. Ajouter une « ligne » à la dimension 1 ajoute une ligne, et le faire à la dimension 2 ajoute une colonne, et ainsi de suite. Si le tenseur n'est par exemple que d'ordre 1 et qu'on ajoute une « ligne » à la dimension 3 (soit une profondeur), la taille des dimensions intermédiaires (ici une dimension : les colonnes) sera automatiquement créée et définie à 1 (et on aura 2 profondeurs).

Opérateurs/opérations arithmétiques

— `Tensor& operator+=(Tensor)` : ajoute les composantes du tenseur donné aux composantes correspondantes du tenseur. Si les dimensions des tenseurs sont différentes, ils sont redimensionnés de sorte que la nouvelle taille englobe toutes les dimensions des deux tenseurs avant l'addition (les nouvelles composantes sont alors nulles pour chaque tenseur) ;
— `Tensor& operator-=(Tensor)` : soustrait les composantes du tenseur donné aux composantes correspondantes du tenseur. Même remarque que pour l'addition si les tenseurs ne sont pas de la même taille ;
— `Tensor& operator*=(T)` : multiplie le tenseur par le scalaire donné.

Opérations sans affectation :

— `Tensor operator+(Tensor t1, Tensor t2)` : somme des deux tenseurs ;
— `Tensor operator-(Tensor t1, Tensor t2)` : différence des deux tenseurs ;
— `Tensor operator-()` : renvoie l'opposé du tenseur, soit celui avec toutes les composantes opposées ;
— `T operator*(Tensor)` : effectue un produit scalaire avec le tenseur donné ;

Autres méthodes et fonctions associées

Autres méthodes :

— `bool isValidPos(uint32_t pos) const` : renvoie si l'indice donnée est valide pour le tenseur ;
— `bool isValidPos(std::vector<uint32_t>) const` : renvoie si les coordonnées données sont valides pour le tenseur ;
— `uint32_t nDToOneD(std::vector<uint32_t>) const` : convertit des coordonnées en un indice ;
— `bool isValidPos(std::vector<uint32_t>) const` : Convertit un indice en coordonnées.

Fonctions extérieures à la classe :

— `std::ostream& operator<<(std::ostream&, Tensor)` : permet par exemple d'afficher un `Tensor` en utilisant `std::cout` comme s'il s'agissait d'un type standard.

4.1.3 Exemples

Manipulations de tenseurs.

```
1  std::vector<uint32_t> dims;
2  dims.push_back(3);
3  ilm::Tensor<ilm::NNumber> t1(dims);
4  std::cout << "t1 = " << std::endl << t1;
5  dims.push_back(4);
6  ilm::Tensor<ilm::NNumber> t2(dims);
7  std::cout << std::endl << "t2 = " << std::endl << t2;
8  dims.push_back(2);
9  ilm::Tensor<ilm::NNumber> t3(dims);
10 std::cout << std::endl << "t3 = " << std::endl << t3;
11 for (uint32_t i(0) ; i < t3.size() ; i++)
12     t3.comp(i*i, i);
13 std::cout << "t3_10 = " << t3 << std::endl;
14 std::cout << "t3_16 = " << t3.str(16) << std::endl;
15 std::cout << "t3_60 = " << t3.str(60);
```

```
t1 =
0
```

```
0
0

t2 =
0       0       0       0
0       0       0       0
0       0       0       0

t3 =

(i, j, 0) :

0       0       0       0
0       0       0       0
0       0       0       0

(i, j, 1) :

0       0       0       0
0       0       0       0
0       0       0       0
t3_10 =
(i, j, 0) :

0       9       36      81
1       16      49      100
4       25      64      121

(i, j, 1) :

144     225     324     441
169     256     361     484
196     289     400     529

t3_16 =
(i, j, 0) :

0       9       24      51
1       10      31      64
4       19      40      79

(i, j, 1) :

90      E1      144     1B9
A9      100     169     1E4
C4      121     190     211

t3_60 =
(i, j, 0) :

0       9       36      1:21
1       16      49      1:40
4       25      1:4     2:1

(i, j, 1) :

2:24    3:45    5:24    7:21
2:49    4:16    6:1     8:4
3:16    4:49    6:40    8:49
```

```
1 std::vector<uint32_t> dims{3, 4, 2};
2 ilm::Tensor<ilm::NNumberL> t(dims);
3 for (uint32_t i(0) ; i < t.size() ; i++)
```

```
4        t.comp(i*i, i);
5    std::vector<uint32_t> pos{1, 2, 1};
6    std::cout << "t = " << t << std::endl;
7    std::cout << "21e composante                       : " << t.comp(21) <<
         std::endl;
8    std::cout << "Composante à la position (1 ; 2 ; 1) : " << t.comp(pos) <<
         std::endl;
9    t.comp(1024, 21);
10   t.comp(65536, pos);
11   std::cout << "t = " << t;
```

```
t =
(i, j, 0) :

0       9       36      81
1       16      49      100
4       25      64      121

(i, j, 1) :

144     225     324     441
169     256     361     484
196     289     400     529

21e composante                       : 441
Composante à la position (1 ; 2 ; 1) : 361
t =
(i, j, 0) :

0       9       36      81
1       16      49      100
4       25      64      121

(i, j, 1) :

144     225     324     1024
169     256     65536   484
196     289     400     529
```

```
1    std::vector<uint32_t> dims{3, 4, 2};
2    ilm::Tensor<ilm::NNumberL> t(dims);
3    for (uint32_t i(0) ; i < t.size() ; i++)
4        t.comp(i*i, i);
5    dims[0] = 4; dims[1] = 2;
6    t.resize(dims);
7    std::cout << "t = " << t;
```

```
t =
(i, j, 0) :

0       9
1       16
4       25
0       0

(i, j, 1) :

144     225
169     256
196     289
0       0
```

Opérations.

```
std::vector<uint32_t> dims{2, 3};
ilm::Tensor<ilm::ZNumberL> t1(dims), t2(dims);
for (uint32_t i(0) ; i < t1.size() ; i++)
    t1.comp(i*i, i);
for (uint32_t i(0) ; i < t2.size() ; i++)
    t2.comp(i*i*i, i);
std::cout << "t1 + t2 = " << std::endl << t1 + t2 << std::endl;
std::cout << "t1 - t2 = " << std::endl << t1 - t2 << std::endl;
std::cout << "t1.t2   = " << t1*t2 << std::endl;
std::cout << "t1*2    = " << std::endl << t1*2 << std::endl;
```

```
t1 + t2 =
0       12      80
2       36      150

t1 - t2 =
0       -4      -48
0       -18     -100

t1.t2   = 4425
t1*2    =
0       8       32
2       18      50
```

4.2 Intervalles (Interval)

4.2.1 Description

Le stockage d'intevalles est également géré, par le patron de classes `Interval<T>`, `T` étant une classe de nombres d'iloMath pour stocker les extrémités. Il stocke simplement les valeurs des extrémités, et deux booléennes indiquant chacune si une extrémité est ouverte ou fermée, et propose des méthodes de bases modifiant ces valeurs.

Comme pour les tenseurs, il n'est pas possible de créer un intervalle à partir d'un `std::string`. On doit l'initialiser soit avec deux nombres ; les extrémités, soit avec ces deux nombres et deux booléennes indiquant la nature des extrémités. Pour les booléenes, on peut utiliser les constantes `OPEN` et `CLOSED` pour respectivement ouvert et fermé. Par défaut, l'intervalle est fermé des deux extrémités.

Il est toujours fait en sorte que la borne inférieure soit plus petite ou égale à la borne supérieure. Les cas du type $]a; a]$ (a est et n'est pas dans l'intervalle) sont toujours automatiquement évités en ouvrant les deux extrémités.

4.2.2 Documentation du patron de classes

On a `T = QNumber` ou `RNumberL`. Alias :

— `QInterval = Interval<QNumber>` ;

— `RIntervalL = Interval<RNumberL>`

Ce patron de classes devrait également accepter `T = NNumber(L)` et `Z<NNumber(L)>`, bien que ceci ne soit pas officiellement supporté.

Attributs

Ces attributs sont privés.

— `T _a, _b` : stocke respectivement la valeur des bornes inférieure et supérieure ;
— `bool _oa, _ob` : stocke respectivement la nature des bornes inférieure et supérieure.

Constructeurs

— `Interval()` : constructeur par défaut, crée l'intervalle $]0;0[=\emptyset$;
— `Interval(T, T, bool = CLOSED, bool = CLOSED) {` : créer un intervalle avec ses deux bornes de type `T` et leur nature de type `bool` (facultatif).

Accesseurs

— `T a() const` : renvoie la borne inférieure ;
— `T b() const` : renvoie la borne supérieure ;
— `bool oa() const` : renvoie la nature de la borne inférieure ;
— `bool ob() const` : renvoie la nature de la borne supérieure ;
— `std::string str(uint16_t = ILM_USERBASE) const` : renvoie l'intervalle avec les extrémités exprimées dans la base donnée (facultatif) sous la forme d'un `std::string`.

Mutateurs

— `void a(T)` : modifie la borne inférieure. Si le nouveau nombre est plus grand que la borne supérieure, les deux bornes sont échangées ;
— `void b(T)` : modifie la borne supérieure. Si le nouveau nombre est plus petit que la borne inférieure, les deux bornes sont échangées ;
— `void oa(bool)` : modifie la nature de la borne inférieure ;
— `void ob(bool)` : modifie la nature de la borne supérieure.

Autres méthodes et fonctions associées

Autres méthodes :

— `T delta() const` : renvoie la longueur de l'intervalle (borne supérieure moins borne inférieure) ;
— `T center() const` : renvoie le centre de l'intervalle (la moyenne des bornes).

Fonctions extérieures à la classe :

— `std::ostream& operator<<(std::ostream&, Interval)` : permet par exemple d'afficher un intervalle en utilisant `std::cout` comme s'il s'agissait d'un type standard.

4.2.3 Exemples

Création et affichage.

```
1 ilm::QInterval i("-64/3", "256/9", ilm::OPEN, ilm::CLOSED);
2 std::cout << "i = " << i << std::endl;
3 std::cout << "i = " << i.str(16) << std::endl;
```

```
i = ]-64/3 ; 256/9]
i = ]-40/3 ; 100/9]
```

Accesseurs et mutateurs.

```
1 ilm::RIntervalL i(-128, 2187);
2 std::cout << "Intervalle       : " << i << std::endl;
3 std::cout << "Borne inférieure : " << i.a() << std::endl;
4 std::cout << "Borne supérieure : " << i.b() << std::endl;
```

```
5 std::cout << "Nature de la borne inférieure : " << i.oa() << std::endl;
6 std::cout << "Nature de la borne supérieure : " << i.ob() << std::endl;
7 i.a(-8); i.b(8);
8 i.oa(ilm::OPEN); i.ob(ilm::OPEN);
9 std::cout << "Intervalle        : " << i << std::endl;
```

```
Intervalle        : [-128 ; 2187]
Borne inférieure : -128
Borne supérieure : 2187
Nature de la borne inférieure : 1
Nature de la borne supérieure : 1
Intervalle        : ]-8 ; 8[
```

4.3 Outils pour la représentation graphique

iloMath propose des outils pouvant faciliter la représentation graphique de fonctions lorsqu'une bibliothèque graphique tierce est utilisée.

4.3.1 Boite de vue (ViewBox)

Dans un premier temps, un outil pour gérer les délimitations lors de représentations graphiques également proposé, par le patron de classes `ViewBox<T>`, `T` étant un type compatible avec `Interval<T>`.

Lorsqu'on souhaite représenter graphiquement une fonction, il faut choisir des intervalles dans lesquelles on placera les points de cette fonction. Par exemple, on peut choisir de représenter une fonction à une variable sur l'intervalle $[-5; 5]$ pour les préimages x, et $[-8; 8]$ pour les images y. Pour une fonction à deux variables, on pourrait choisir deux intervalles pour les préimages de ces deux variables, et un troisième pour les images.

Le patron de classes `ViewBox<T>` permet le stockage de ces intervalles. On peut dire d'une certaine manière qu'il représente un produit cartésien d'intervalles.

Documentation du patron de classes

On a `T = QNumber` ou `RNumberL`. Alias :
- `QViewBox  = ViewBox<QNumber>`;
- `RViewBoxL = ViewBox<RNumberL>`

Attributs Cet attribut est privé.
- `std::vector<Interval<T>> _interval` : stocke les intervalles délimitant le champ de vision.

Constructeurs
- `ViewBox()` : constructeur par défaut, crée une boite de vue sans intervalle;
- `ViewBox(std::vector<Interval<T>>)` : crée la boite de vue délimitées par ces intervalles (dimension arbitraire);
- `ViewBox(Interval<T>)` : crée la boite de vue délimitées par un intervalle (1D);
- `ViewBox(Interval<T>, Interval<T>)` : crée la boite de vue délimitées par deux intervalles (2D);
- `ViewBox(Interval<T>, Interval<T>, Interval<T>)` : crée la boite de vue délimitées par trois intervalles (3D).

Une fois qu'une boite de vue a été crée, le seul moyen de changer sa dimension est d'appeler à nouveau un constructeur.

Accesseurs

— `uint32_t size() const` : renvoie le nombre d'intervalles ;
— `Interval<T> interval(uint32_t) const` : renvoie l'invervalle situé à l'indice donnée dans `_interval` ;
— `T a(uint32_t) const` : renvoie la borne inférieure de l'invervalle situé à l'indice donnée dans `_interval` ;
— `T b(uint32_t) const` : renvoie la borne supérieure de l'invervalle situé à l'indice donnée dans `_interval` ;
— `bool oa(uint32_t) const` : renvoie la nature de la borne inférieure de l'invervalle situé à l'indice donnée dans `_interval` ;
— `bool ob(uint32_t) const` : renvoie la nature de la borne supérieure de l'invervalle situé à l'indice donnée dans `_interval`.

Mutateurs

— `void interval(Interval<T>, uint32_t)` : modifie l'invervalle situé à l'indice donnée dans `_interval` ;
— `void a(T, uint32_t)` : modifie la borne inférieure de l'invervalle situé à l'indice donnée dans `_interval` ;
— `void b(T, uint32_t)` : modifie la borne supérieure de l'invervalle situé à l'indice donnée dans `_interval` ;
— `void oa(T, uint32_t)` : modifie la nature de la borne inférieure de l'invervalle situé à l'indice donnée dans `_interval` ;
— `void ob(T, uint32_t)` : modifie la nature de la borne supérieure de l'invervalle situé à l'indice donnée dans `_interval`.

Autres méthodes

— `void zoom(T factor, std::vector<T> x = std::vector<T>())` : effectue un zoom dans la boite de vue en réduisant la taille de tous les intervalles de `factor`, et en faisant en sorte que le point x dont les coordonnées sont contenues dans le `std::vector<T>` conserve sa position relative après le zoom (par exemple, si on zoom avec x le centre de la boite de vue, x sera également le centre après le zoom). Si le `std::vector<T>` a moins de coordonnées qu'il n'y a d'intervalles dans la vue, les coordonnées manquantes sont ajoutées et valent le centre de leurs intervalles respectifs. Si ce `std::vector<T>` est au contraire trop grand, les coordonnées en trop sont ignorées ;
— `void dezoom(T factor, std::vector<T> x = std::vector<T>())` : appelle `zoom` avec `1/factor` ;
— `std::vector<std::vector<T>> graduation()` : propose une graduation pouvant être utilisée pour afficher les grilles d'un graphique. Cette graduation ne fonctionne qu'en base 10 et chaque pas vaut à chaque fois un multiple d'une puissance de 10 (comme $0,01; 0,02; 0,03; ...$), la moitié d'une puissance de 10 (comme $5; 10; 15; ...$) ou un cinquième d'une puissance de 10 (comme $0,2; 0,4; 0,6; ...$). Chaque sous-`std::vector<T>` contient une graduation pour un intervalle donné, donc si la vue est en 3D, il y aura trois `std::vector<T>`.

Exemples

Comme pour les tenseurs et les intervalles, il n'est pas possible de créer une boite de vue à partir d'un `std::string` ; il faut utiliser les constructeurs documentés. Il n'est pas

possible non plus d'afficher directement une `ViewBox`. Pour cela, il faut récupérer les intervalles individuellement. Exemple pratique :

```
ilm::Interval<ilm::QNumber> ix(-5, 5), iy(-8, 8);
ilm::ViewBox<ilm::QNumber> vb(ix, iy);
std::cout << "Intervalle des préimages : " << vb.interval(0) << std::endl
    ;
std::cout << "Intervalle des images    : " << vb.interval(1) << std::endl
    ;
```

```
Intervalle des préimages : [-5 ; 5]
Intervalle des images    : [-8 ; 8]
```

Manipulations supplémentaires.

```
ilm::RIntervalL ix(-10, 10), iy(-50, 50), iz(0, 800);
ilm::RViewBoxL vb(ix, iy, iz);

std::cout << "Intervalle x        : " << vb.interval(0) << std::endl;
std::cout << "Intervalle y        : " << vb.interval(1) << std::endl;
std::cout << "Intervalle z        : " << vb.interval(2) << std::endl;
std::cout << "Intervalles         : " << vb.size() << std::endl;
std::cout << "Borne inférieure y : " << vb.a(1) << std::endl;
std::cout << "Borne supérieure y : " << vb.b(1) << std::endl;
std::cout << "Nature de la borne inférieure z : " << vb.oa(2) << std::
    endl;
std::cout << "Nature de la borne supérieure z : " << vb.ob(2) << std::
    endl;

vb.a(-8, 0); vb.b(16, 0);
vb.oa(ilm::OPEN, 0); vb.ob(ilm::OPEN, 0);

std::cout << "Intervalle x        : " << vb.interval(0) << std::endl;
```

```
Intervalle x        : [-10 ; 10]
Intervalle y        : [-50 ; 50]
Intervalle z        : [0 ; 800]
Intervalles         : 3
Borne inférieure y : -50
Borne supérieure y : 50
Nature de la borne inférieure z : 1
Nature de la borne supérieure z : 1
Intervalle x        : ]-8 ; 16[
```

Démonstration du zoom.

```
ilm::RIntervalL ix(-3, 5), iy(-4, 2);
ilm::RViewBoxL vb(ix, iy);
std::cout << vb.interval(0) << " x " << vb.interval(1);
vb.zoom(2, std::vector<ilm::RNumberL>{1, -2});
std::cout << " -> " << vb.interval(0) << " x " << vb.interval(1) << std::
    endl;
```

```
[-3 ; 5] x [-4 ; 2] -> [-1 ; 3] x [-3 ; 0]
```

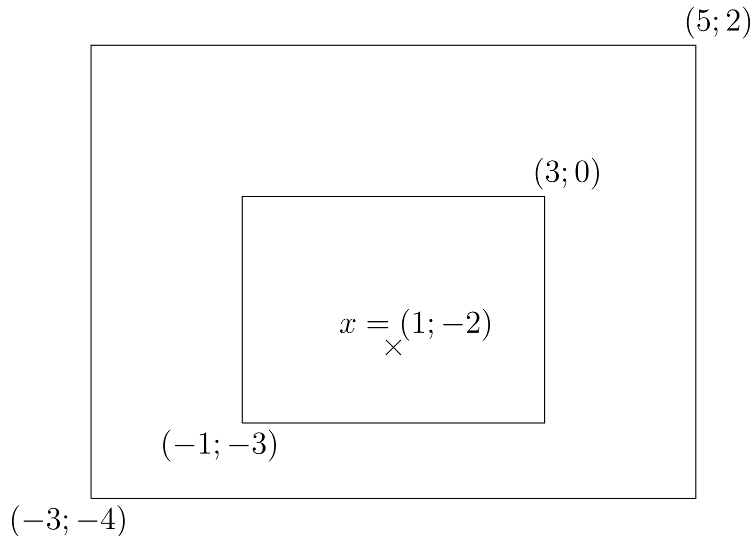

FIGURE 4.1 – Le zoom, concrètement. Comme prévu, les longueurs des intervalles ont bien été divisées par deux, et si on regarde $x = (1; -2)$, on constate que sur les deux rectangles, il se situe horizontalement au milieu, et aux $\frac{2}{3}$ verticalement.

4.3.2 Données de graphique (DataPlot)

Description

Le patron de classes `DataPlot<class S, class T>` permet de stocker des préimages de type `S` et images de type `T` de fonctions (les deux types pouvant être différents ; par exemple, on pourrait vouloir substituer des préimages réelles, mais avoir des images complexes).

Ce patron de classes consiste en le stockage des préimages dans un `std::vector<std::pair<std::vector<S>,std::string>>`, et les images dans un tenseur de coefficients de type `T`. Cela permet de gérer la représentation graphique de fonctions à nombre arbitraire de variables. Les exemples suivants expliquent un tel choix de stockage.

Dans le cas d'une fonction à une variable, une préimage correspond à une image. Les préimages sont stockées dans un `std::vector`, lié à l'aide de `std::pair` avec un `std::string` afin de pouvoir stocker également le nom de la variable correspondante. Cette paire lui-même est contenu dans un autre `std::vector`, tandis que les images seront stockées dans un tenseur d'ordre 1.

Dans le cas d'une fonction à deux variables, il faut deux préimages (par exemple, x et y) pour avoir une image (par exemple, z) ; $f(x; y) = z$. Pour la représentation graphique, il faut alors calculer les images de tous les couples de x et y possibles qui ont été choisis pour l'évaluation. Les préimages sont alors stockées dans deux `std::vector` (un contenant les x, et l'autre les y), chacun liés à un `std::string` contenant le nom de ces variables, et ces deux paires sont elle-mêmes contenues dans un autre `std::vector`, tandis que les images seront stockées dans un tenseur d'ordre 2, dont les dimensions sont le nombre de x et de y évalués.

Et le raisonnement est le même pour les fonctions à plus de variables.

La fonction qui sera évaluée est également stockée dans ce patron de classes, ainsi qu'une boite de vue qui sert de délimiteur du domaine visible. En effet, on peut très bien évaluer les images de x^3 pour $x \in [-5; 5]$, mais afficher seulement les points évalués en $x \in [-3; 4]$, et restreindre l'intervalle des images à $[-10; 10]$. Ainsi, toutes les valeurs pour $x \in [-5; 5]$ ont bien été calculées, mais on n'affiche que ce qui entre dans la boite de vue.

Une autre utilité, est l'existence d'une méthode permettant de convertir des points d'une

boite de vue vers une autre, tout en gardant exactement le même graphe se situant dans l'attribut de la boite de vue. Ceci est particulièrement utile pour convertir des points de la fonction en pixels, ce qui facilite considérablement la tâche de l'affichage du graphique. Pour cela, la transformation affine est utilisée :

$$x' = \frac{(x - x_a)(x'_b - x'_a)}{x_b - x_a} + x'_a$$
$$y' = \frac{(y - y_a)(y'_b - y'_a)}{y_b - y_a} + y'_a$$

x_a et x_b sont les extrémités de l'intervalle des points x à évaluer, y_a et y_b les extrémités de l'intervalle des images, et les variables avec prime les homologues en pixels.

Documentation du patron de classes

On a `T = QNumber` ou `RNumberL` et `S` également, mais il peut aussi être complexe.

Attributs Ces attributs sont privés.

- `Function<T> _f` : stocke la fonction à évaluer ;
- `std::vector<std::pair<std::vector<S>, std::string>> _pimgs` : stocke les pré-images, associés avec le nom de leur variable ;
- `Tensor<T> _imgs` : stocke les images ;
- `ViewBox<S> _vb` : stocke le champ de valeurs visibles.

Constructeurs

- `DataPlot()` : constructeur par défaut, crée un `DataPlot` qui ne contient aucune donnée.

Accesseurs

- `Function<T> f() const` : renvoie la fonction stockée ;
- `uint32_t size() const` : renvoie le nombre d'images ;
- `uint32_t nPimgs() const` : renvoie le nombre d'assemblages de préimages possibles, soit le nombre d'images qu'il y aura (utile si les images n'ont pas encore été calculées) ;
- `S pimg(uint32_t i, uint32_t j) const` : renvoie la j^e préimage (deuxième argument) dans le i^e `std::vector<S>` stocké dans `_pimgs` ;
- `T img(uint32_t) const` : renvoie la $n$ième image (au sens de `Tensor`) ;
- `std::vector<std::pair<std::vector<S>, std::string>> pimgs() const` : renvoie toutes les préimages et le nom des variables ;
- `Tensor<T> imgs() const` : renvoie toutes les images ;
- `ViewBox<S> vb() const` : renvoie la boite de vue stockée.

Mutateurs

- `void reset()` : supprime toutes les données du graphique ;
- `void f(Function<T>)` : modifie la fonction à évaluer ;
- `void vb(ViewBox<S> vb)` : modifie le champ de vision ;
- `void pimgs(std::vector<std::tuple<Interval<S>, S, std::string>>)` : génère les préimages à partir d'intervalles et le pas entre chaque préimage (qui sont équitablement espacés). Si, pour un intervalle, le pas est plus grand que la longueur de l'intervalle, le pas est forcé à la longueur de l'intervalle ;
- `void pimgs(std::vector<std::tuple<Interval<S>, uint32_t, std::string>>)` : idem, mais avec le nombre de préimages par intervalle au lieu du pas. Si ce nombre est plus petit que 2, il est forcé à 2 ;

— `void pimgs(Interval<S>, S, std::string = "x")` : idem, mais pour une seule variable (avec pas) ;
— `void pimgs(Interval<S>, uint32_t, std::string = "x")` : idem, mais pour une seule variable (avec nombre de points) ;
— `void plot()` : calcule les images en complétant `_imgs`.

Il est de la responsabilité de l'utilisateur d'entrer des préimages correspondant bien aux variables de la fonction, sans quoi, le comportement est indéfini.

Autres méthodes

— `static DataPlot convert(DataPlot<S, T> dp, ViewBox<S> vb0)` : convertit les points de sorte à ce que les nouvelles valeurs correspondent à la même position relative dans la boite de vue spécifiée, par rapport à celle stockée dans le `DataPlot` ;
— `static DataPlot complexConvert(DataPlot<S, T> dp, ViewBox<S> vb0)` : à utiliser à la place de `convert` si les images sont complexes, car ce dernier ne les convertit pas correctement.

Si les boites de vues ne sont pas de la même dimension on ne correspondent pas aux images, rien ne se passe (un `DataPlot` vide est renvoyé).

Exemples

Utilisation en pratique.

```
1  ilm::RFunctionL f("x^3 + xy^2");
2  ilm::RIntervalL ix(-3, 3), iy(-1.5, 1);
3  ilm::DataPlot<ilm::RNumberL, ilm::RNumberL> dp;
4
5  std::vector<std::tuple<ilm::RIntervalL, ilm::RNumberL, std::string>>
       vpisv;
6  vpisv.push_back(std::make_tuple(ix, 1.0, "x"));
7  vpisv.push_back(std::make_tuple(iy, 0.5, "y"));
8
9  dp.f(f);
10 dp.pimgs(vpisv);
11 dp.plot();
12
13 std::vector<ilm::RNumberL> pimgsx(std::get<0>(dp.pimgs()[0])),
14                            pimgsy(std::get<0>(dp.pimgs()[1]));
15
16 std::cout << "f(x ; y)     : " << f << std::endl;
17 std::cout << "Préimages x : {" << pimgsx[0];
18 for (uint32_t i(1) ; i < pimgsx.size() ; i++)
19     std::cout << " ; " << pimgsx[i];
20 std::cout << "}" << std::endl << "Préimages y : {" << pimgsy[0];
21 for (uint32_t i(1) ; i < pimgsy.size() ; i++)
22     std::cout << " ; " << pimgsy[i];
23 std::cout << "}" << std::endl << "Images      : " << std::endl << dp.imgs
       ();
```

```
f(x ; y)    : x^3 + x*y^2
Préimages x : {-3 ; -2 ; -1 ; 0 ; 1 ; 2 ; 3}
Préimages y : {-1.5 ; -1 ; -0.5 ; 0 ; 0.5 ; 1}
Images      :
-33.75   -30     -27.75   -27     -27.75   -30
-12.5    -10     -8.5     -8      -8.5     -10
-3.25    -2      -1.25    -1      -1.25    -2
0        0       0        0       0        0
3.25     2       1.25     1       1.25     2
12.5     10      8.5      8       8.5      10
33.75    30      27.75    27      27.75    30
```

Pour une seule variable, l'utilisation peut être grandement simplifiée.

```
1  ilm::QFunction f("x^3 - 2x^2 + 3x - 4");
2  ilm::QInterval intervalle(-5, 5);
3  ilm::DataPlot<ilm::QNumber, ilm::QNumber> dp;
4
5  dp.f(f);
6  dp.pimgs(intervalle, "2", "x");
7  dp.plot();
8
9  std::cout << "f(x)      : " << f << std::endl;
10 std::cout << "Préimages : {" << std::get<0>(dp.pimgs()[0])[0];
11 for (uint32_t i(1) ; i < std::get<0>(dp.pimgs()[0]).size() ; i++)
12     std::cout << " ; " << std::get<0>(dp.pimgs()[0])[i];
13 std::cout << "}" << std::endl << "Images    : " << std::endl << dp.imgs()
    ;
```

```
f(x)      : x^3 - 2*x^2 + 3*x - 4
Préimages : {-5 ; -3 ; -1 ; 1 ; 3 ; 5}
Images    :
-194
-58
-10
-2
14
86
```

Utilisation de la fonction `convert` pour convertir un graphique vers par exemple une zone de 640×480 pixels.

```
1  ilm::RFunctionL f("x^3 - 2x^2 + 3x - 4");
2  ilm::RIntervalL ix0(-5, 5), iy0(-300, 500), ix(0, 640), iy(0, 480);
3  ilm::DataPlot<ilm::RNumberL, ilm::RNumberL> dp, dp2;
4  ilm::RViewBoxL vb(ix0, iy0), vb2(ix, iy);
5
6  dp.f(f);
7  dp.pimgs(ix0, 5, "x");
8  dp.vb(vb); // On aurait pu faire ça aprés plot aussi
9  dp.plot();
10 dp2 = ilm::DataPlot<ilm::RNumberL, ilm::RNumberL>::convert(dp, vb2);
11
12 std::vector<ilm::RNumberL> pimgs(std::get<0>(dp.pimgs()[0])), pimgs2(std
    ::get<0>(dp2.pimgs()[0]));
13
14 std::cout << "f(x)      : " << f << std::endl;
15 std::cout << "Préimages : {" << pimgs[0];
16
17 for (uint32_t i(1) ; i < pimgs.size() ; i++)
18     std::cout << " ; " << pimgs[i];
19 std::cout << "} -> {" << pimgs2[0];
20 for (uint32_t i(1) ; i < pimgs2[0].size() ; i++)
21     std::cout << " ; " << pimgs2[i];
22 std::cout << "}" << std::endl << "Images      : " << std::endl << dp.imgs
    () << "->" << std::endl << dp2.imgs();
```

```
f(x)      : x^3 - 2*x^2 + 3*x - 4
Préimages : {-5 ; -2.5 ; 0 ; 2.5 ; 5} -> {0}
Images      :
-194
-39.625
-4
6.625
86
```

```
->
63.6
156.225
177.6
183.975
231.6
```

4.3.3 Données de graphique complexes (CDataPlot)

Ce dernier patron de classes était surtout destiné aux nombres réels, et une version spécifique aux nombres complexes a été implémentée. Elle reprend la plupart des fonctionnalités, adaptées à ces nombres.

Documentation du patron de classes

On a `T = QNumber` ou `RNumberL`, le type doit donc être celui des coefficients d'un nombre complexe et non un type complexe.

Attributs Ces attributs sont privés.

- `Function<T> _f` : stocke la fonction à évaluer ;
- `std::vector<std::tuple<std::vector<T>, std::vector<T>, std::string>> _pimgs` : stocke les préimages, associés avec le nom de leur variable ;
- `Tensor<T> _imgs` : stocke les images.

Ici, il y a deux ensembles de préimages par variable, donc chaque nouvelle variable ajoute deux dimensions au graphe. Elles ne stockent que les coefficients des nombres complexes évalués, et les coefficients correspondants composeront le nombre complexe lors de l'évaluation. Dans les images, les valeurs stockées le long d'une dimension paire (en commençant par 0) varient selon la partie réelle des préimages et celles stockées le long d'une dimension impaire selon la partie imaginaire.

Constructeurs

- `CDataPlot()` : constructeur par défaut, crée un `CDataPlot` qui ne contient aucune donnée.

Accesseurs

- `Function<T> f() const` : renvoie la fonction stockée ;
- `uint32_t size() const` : renvoie le nombre d'images ;
- `uint32_t nPimgs() const` : renvoie le nombre d'assemblages de préimages possibles, soit le nombre d'images qu'il y aura (utile si les images n'ont pas encore été calculées) ;
- `T img(uint32_t) const` : renvoie la $n$ième image (au sens de `Tensor`) ;
- `std::vector<std::tuple<std::vector<T>, std::vector<T>, std::string>> pimgs() const` : renvoie toutes les préimages et le nom des variables ;
- `Tensor<T> imgs() const` : renvoie toutes les images.

Mutateurs

- `void reset()` : supprime toutes les données du graphique ;
- `void f(Function<T>)` : modifie la fonction à évaluer ;
- `void vb(ViewBox<S> vb)` : modifie le champ de vision ;
- `void pimgs(std::vector<std::tuple<std::pair<Interval<T>, T>, std::pair<Interval<T>, T>, std::string>>)` : génère les préimages à partir d'in-

tervalles et le pas entre chaque coefficient de préimage (qui sont équitablement espacés). Si, pour un intervalle, le pas est plus grand que la longueur de l'intervalle, le pas est forcé à la longueur de l'intervalle ;

— `void pimgs(std::vector<std::tuple<std::pair<Interval<T>, uint32_t>, std::pair<Interval<T>, uint32_t>, std::string>>)` : idem, mais avec le nombre de préimages par intervalle au lieu du pas. Si un de ces nombres est plus petit que 2, il est forcé à 2 ;

— `void pimgs(std::pair<Interval<T>, T> is , std::pair<Interval<T>, T> isi, std::string = "x")` : idem, mais pour une seule variable (avec pas) ;

— `void pimgs(std::pair<Interval<T>, uint32_t> is, std::pair<Interval<T>, uint32_t> isi, std::string = "x")` : idem, mais pour une seule variable (avec nombre de points) ;

— `void plot()` : calcule les images en complétant `_imgs`.

Les deux intervalles $[a; b]$ et $[c; d]$ par variable indiquent qu'elle sera évaluée avec des valeurs situées dans le rectangle dont les coins opposés sont $a + bi$ et $c + di$. Il est de la responsabilité de l'utilisateur d'entrer des préimages correspondant bien aux variables de la fonction, sans quoi, le comportement est indéfini.

Autres méthodes La méthode `convert` est remplacée par `rima` qui décompose les parties Réelle et Imaginaire, le Module et l'Argument. et renvoie un tableau de quatre tenseurs de la même taille des images.

— `T minRe()`, `T minIm()`, `T minMod()` et `T minArg()` retournent l'image dont la partie réelle (respectivement imaginaire, le module et l'argument) est la plus petite ;

— Remplacer `min` par `max` des fonctions précédantes donnent la plus grande valeur à la place ;

— `std::array<T, 8> minMax()` : renvoie `minRe()`, `maxRe()`, `minIm()`, `maxIm()`, `minMod()`, `maxMod()`, `minArg()` et `maxArg()` dans cet ordre dans un tableau (ce qui permet d'éviter de parcourir 8 fois le tenseur pour trouver ces extrêmums) ;

— `std::array<Tensor<T>, 4> rima(T minRe, T maxRe, T minIm, T maxIm, T minMod, T maxMod, T minArg, T maxArg, T max = T::one())` : génère quatre tenseurs contenant une décomposition normalisée des images pour leur analyse. Dans l'ordre, le tableau contient le tenseur des parties réelles, imaginaires, les modules, et les argument. Ces valeurs varient entre 0 et la valeur `max` renseignée. Par exemple, l'image $1 + 2i$, si on fournit les paramètres `minRe = 0`, `maxRe = 4` et `max = 1`, aura une partie réelle normalisée de $0,25$ qui sera stockée à la même place dans le tenseur des parties réelles ;

— `std::array<Tensor<T>, 4> rima()` appelle la fonction ci-dessus avec les méthodes du même nom des argument (`minRe()` pour `minRe`, ...).

Exemple

Évaluation de $\sin(z)$ complexe.

```
1  ilm::CFunctionL f("sin(z)");
2  ilm::CDataPlot<ilm::RNumberL> cdp;
3
4  cdp.f(f);
5  cdp.pimgs(std::make_pair(ilm::RIntervalL(-4, 4), (uint32_t) 4),
6            std::make_pair(ilm::RIntervalL(-4, 4), (uint32_t) 3), "z");
7  cdp.plot();
8
9  std::cout << "f(z)                : " << f << std::endl;
10 std::cout << "Préimages z (Re) : {" << std::get<0>(cdp.pimgs()[0])[0];
11 for (uint32_t i(1) ; i < std::get<0>(cdp.pimgs()[0]).size() ; i++)
```

```
12        std::cout << " ; " << std::get<0>(cdp.pimgs()[0])[i];
13    std::cout << "}" << std::endl << "Préimages z (Im) : {" << std::get<1>(
          cdp.pimgs()[0])[0];
14    for (uint32_t i(1) ; i < std::get<1>(cdp.pimgs()[0]).size() ; i++)
15        std::cout << " ; " << std::get<1>(cdp.pimgs()[0])[i];
16    std::cout << "}" << std::endl << "Images        : " << std::endl << cdp.
          imgs();
```

```
f(z)        : sin(z)
Préimages z (Re) : {-4 ; -1.33333 ; 1.33333 ; 4}
Préimages z (Im) : {-4 ; 0 ; 4}
Images        :
20.6669 + 17.8379i      0.756802        20.6669 - 17.8379i
-26.5419 - 6.41961i     -0.971938       -26.5419 + 6.41961i
26.5419 - 6.41961i      0.971938        26.5419 + 6.41961i
-20.6669 + 17.8379i     -0.756802       -20.6669 - 17.8379i
```

Pour plus de variables, cela se complique énormément. Mais, une fois qu'on a compris le fonctionnement, généraliser à encore plus de variables est très simple.

```
1     ilm::CFunctionL f("x + yi");
2     ilm::RIntervalL ix(-4, 8), ixi(-10, 10), iy(-5, 3), iyi(-8, 8);
3     ilm::CDataPlot<ilm::RNumberL> cdp;
4
5     std::vector<std::tuple<std::pair<ilm::RIntervalL, ilm::RNumberL>, std::
          pair<ilm::RIntervalL, ilm::RNumberL>, std::string>> vpisv;
6     vpisv.push_back(std::make_tuple(std::make_pair(ix, "4"), std::make_pair(
          ixi, "5"), "x"));
7     vpisv.push_back(std::make_tuple(std::make_pair(iy, "1"), std::make_pair(
          ixi, "4"), "y"));
8
9     cdp.f(f);
10    cdp.pimgs(vpisv);
11    cdp.plot();
12
13    std::vector<ilm::RNumberL> pimgsx(std::get<0>(cdp.pimgs()[0])),
14                               pimgsxi(std::get<1>(cdp.pimgs()[0])),
15                               pimgsy(std::get<0>(cdp.pimgs()[1])),
16                               pimgsyi(std::get<1>(cdp.pimgs()[1]));
17
18    std::cout << "f(x ; y)      : " << f << std::endl;
19    std::cout << "Préimages x   : {" << pimgsx[0];
20    for (uint32_t i(1) ; i < pimgsx.size() ; i++)
21        std::cout << " ; " << pimgsx[i];
22    std::cout << "}" << std::endl << "Préimages xi : {" << pimgsxi[0];
23    for (uint32_t i(1) ; i < pimgsxi.size() ; i++)
24        std::cout << " ; " << pimgsxi[i];
25    std::cout << "}" << std::endl << "Préimages y  : {" << pimgsy[0];
26    for (uint32_t i(1) ; i < pimgsy.size() ; i++)
27        std::cout << " ; " << pimgsy[i];
28    std::cout << "}" << std::endl << "Préimages yi : {" << pimgsyi[0];
29    for (uint32_t i(1) ; i < pimgsyi.size() ; i++)
30        std::cout << " ; " << pimgsyi[i];
31    std::cout << "}" << std::endl << "Images        : " << std::endl << cdp.
          imgs();
```

```
f(x ; y)      : x + y*i
Préimages x  : {-4 ; 0 ; 4 ; 8}
Préimages xi : {-10 ; -5 ; 0 ; 5 ; 10}
Préimages y  : {-5 ; -4 ; -3 ; -2 ; -1 ; 0 ; 1 ; 2 ; 3}
Préimages yi : {-10 ; -6 ; -2 ; 2 ; 6 ; 10}
Images        : [trés gros tenseur]
```

5. Autres fonctionnalités

5.1 Couleurs

Comme fonctionnalité supplémentaire, `iloMath` supporte le stockage de couleurs `RVBA` (rouge, vert, bleu, alpha) et `TSLA` (teinte, saturation, luminosité, alpha) avec diverses bornes, et la conversion entre ces deux types de couleurs.

À noter que malgré leur nom, ils peuvent très bien être utilisés pour stocker d'autres types de couleurs si possible et si besoin, voire quelque chose d'autre qu'une couleur si la structure de donnée correspond bien.

5.1.1 Couleurs RVBA avec entiers (RgbaI)

Description

Cette classe permet de stocker une couleur par ses composantes rouge, vert, bleu, et transparence dans des valeurs entières de 0 à $2^n - 1$, avec n compris entre 1 et 32.

Documentation de la classe

Attributs Ces attributs sont privés.

- `std::array<uint64_t, 4> _sub` : stocke les quatre composantes de la couleur ;
- `uint8_t _bits` : stocke le nombre de bits utilisés par composante. Cela définit indirectement la valeur maximale d'une composante, qui est la puissance de deux correspondante, moins un.

Constructeur

- `RgbaI(uint64_t r = 0, uint64_t g = 0, uint64_t b = 0, uint64_t a = 0, uint8_t bits = 8)` : constructeur en fournissant la valeur des composantes et du nombre de bits utilisés par composante. Tous ces paramètres sont facultatifs. Si le nombre de bits par couleur est plus grand que 32 il est forcé à 32. S'il vaut 0, il est forcé à 1.

Le choix de `uint64_t` pour stocker les composantes et du maximum de 32 bits effectivment utilisés est dû au fait que cela évite des débordements qui fausseraient les résultats lors de certaines conversions. Sinon, il est vrai qu'utiliser un maximum de 16 bits dans des `uint32_t` aurait peut-être largement suffi... Mais dans tous les cas, qui peut le plus peut le moins ! De plus, comme mentionné avant, ça pourrait aussi être utilisé pour stocker d'autres choses. Et on ne sait jamais, peut-être que les photographes du futur auront besoin de 2^{128} couleurs :-°...

Accesseurs

- `uint64_t r() const` : renvoie la valeur de la composante pour le rouge (les fonctions `g()`, `b()` et `a()` font la même chose pour les composantes verte, bleue, et alpha) ;
- `uint64_t sub(uint8_t i) const` : renvoie la composante correspondante ; i est compris entre 0 et 3 inclus, et cela renvoie la composante respectives rouge, verte, bleue, et alpha. Si $i > 3$, 0 est retourné ;

— `uint64_t max() const` : renvoie la valeur maximale qu'une composante peut avoir, soit $2^n - 1$ avec n le nombre de bits par composante ;
— `std::string str() const` : renvoie la couleur sous la forme d'un `std::string`, avec les composantes séparées par une tabulation.

Mutateurs

— `void r(uint64_t s)` : modifie la composante rouge (comme pour les accesseurs, on a aussi `g`, `b` et `a`). Si s est plus grande que la valeur maximale possible, elle est forcée à cette valeur ;
— `void sub(uint64_t s, uint8_t i)` : modifie la i^e composante en lui donnant la valeur s. L'interprétation de l'indice est la même que pour l'accesseur. Si $i > 3$, rien ne se passe. Si s est plus grand que la valeur maximale, elle est forcée à cette valeur maximale ;
— `void convert(uint8_t bits)` : convertit la couleur vers la même, mais avec un nombre différent de bits.

Exemple

Création et conversions d'une couleur.

```
1 ilm::RgbaI rgbai(192, 255, 238);
2 std::cout << "Couleur (en 32 bits) : "<< rgbai << std::endl;
3 rgbai.convert(16); // 16 x 4 = 64
4 std::cout << "Couleur (en 64 bits) : "<< rgbai << std::endl;
5 rgbai.convert(4); // 4 x 4 = 16
6 std::cout << "Couleur (en 16 bits) : "<< rgbai << std::endl;
```

```
Couleur (en 32 bits) : 192      255     238     0
Couleur (en 64 bits) : 49344    65535   61166   0
Couleur (en 16 bits) : 11       15      14      0
```

5.1.2 Couleurs RVBA avec flottants (RgbaF)

Description

Comme la classe précédante, mais en stockant les composantes dans des flottants, ce qui peut être utile par exemple si on souhaite les avoir dans l'invervalle $[0; 1]$. Il s'utilise de la même manière que son homologue avec entiers.

Documentation de la classe

Attributs Ces attributs sont privés.

— `std::array<double, 4> _sub` : stocke les quatre composantes de la couleur ;
— `double _bits` : stocke la valeur maximale pouvant être stockée pour une composante.

Constructeur

— `RgbaF(double r = 0, double g = 0, double b = 0, double a = 0, double max = 1.)` : constructeur en fournissant la valeur des composantes et le maximum pour les composantes. Tous ces paramètres sont facultatifs. Si la valeur maximale n'est pas strictement positive, elle est forcée à 1. Si la valeur d'une composante est négative, elle est forcée à 0. Si elle est plus grande que le maximum donné, elle est forcée à cette valeur.

Accesseurs

— `double r() const` : renvoie la valeur de la composante pour le rouge (les fonctions `g()`, `b()` et `a()` font la même chose pour les composantes verte, bleue, et alpha) ;
— `double sub(uint8_t i) const` : renvoie la composante correspondante ; i est compris entre 0 et 3 inclus, et cela renvoie la composante respectives rouge, verte, bleue, et alpha. Si $i > 3$, 0 est retourné ;
— `double max() const` : renvoie la valeur maximale qu'une composante peut avoir ;
— `std::string str() const` : renvoie la couleur sous la forme d'un `std::string`, avec les composantes séparées par une tabulation.

Mutateurs

— `void r(double s)` : modifie la composante rouge (comme pour les accesseurs, on a aussi `g`, `b` et `a`). Si s est plus grande que la valeur maximale possible, elle est forcée à cette valeur ;
— `void sub(double s, uint8_t i)` : modifie la i^e composante en lui donnant la valeur s. L'interprétation de l'indice est la même que pour l'accesseur. Si $i > 3$, rien ne se passe. Si s est plus grand que la valeur maximale, elle est forcée à cette valeur maximale ;
— `void convert(double bits)` : convertit la couleur vers la même, mais avec un maximum différent.

Exemple

Création et conversions d'une couleur.

```
1 ilm::RgbaF rgbaf(0.75, 1., 0.93);
2 std::cout << "Couleur (max = 1)   : "<< rgbaf << std::endl;
3 rgbaf.convert(256.);
4 std::cout << "Couleur (max = 256) : "<< rgbaf << std::endl;
5 rgbaf.convert(16.);
6 std::cout << "Couleur (max = 16)  : "<< rgbaf << std::endl;
```

```
Couleur (max = 1)   : 0.75      1       0.93    0
Couleur (max = 256) : 192       256     238.08  0
Couleur (max = 16)  : 12        16      14.88   0
```

5.1.3 Couleurs TSLA avec flottants (Hsl)

Description

Cette classe a été conçue pour stocker des couleurs dont les composants désignent la teinte, saturation et luminosité à la place du rouge, vert et bleu. En particulier, il peut prendre des maximums différents pour les composantes (par exemple, 360 pour la teinte et 1 pour le reste).

Sinon, son utilisation est similaire.

HSL signifie Hue, Saturation and Lightness.

Documentation de la classe

Attributs Ces attributs sont privés.

— `std::array<double, 4> _sub` : stocke les quatre composantes de la couleur ;
— `std::array<double, 4> _max` : stocke les valeurs maximales pour les composantes correspondantes.

Constructeur

- `Hsl(double h = 0., double s = 0., double l = 0., double a = 0., double maxH = 360., double maxS = 1., double maxL = 1., double maxA = 1.)` : constructeur en fournissant la valeur des composantes et les maximums pour les composantes. Tous ces paramètres sont facultatifs. Si une valeur maximale n'est pas strictement positive, elle est forcée à 360 pour la teinte et 1 pour le reste. Si la valeur d'une composante est négative, elle est forcée à 0. Si elle est plus grande que son maximum correspondant, elle est forcée à cette valeur.

Accesseurs

- `double h() const` : renvoie la valeur de la composante pour la teinte (les fonctions `s()`, `l()` et `a()` font la même chose pour les composantes saturation, luminosité, et alpha) ;
- `double sub(uint8_t i) const` : renvoie la composante correspondante ; i est compris entre 0 et 3 inclus, et cela renvoie la composante respectives teinte, saturation, luminosité, et alpha. Si $i > 3$, 0 est retourné ;
- `double max(uint8_t i = 0) const` : renvoie la valeur maximale pour la composante choisie ; i s'interprète comme pour `sub`. Si $i > 3$, 0 est retourné ;
- `std::string str() const` : renvoie la couleur sous la forme d'un `std::string`, avec les composantes séparées par une tabulation.

Mutateurs

- `void h(double n)` : modifie la composante rouge (comme pour les accesseurs, on a aussi `s`, `l` et `a`). Si n est plus grande que la valeur maximale correspondante possible, elle est forcée à cette valeur ;
- `void sub(double n, uint8_t i)` : modifie la i^e composante en lui donnant la valeur s. L'interprétation de l'indice est la même que pour l'accesseur. Si $i > 3$, rien ne se passe. Si s est plus grand que la valeur maximale correspondante, elle est forcée à cette valeur maximale ;
- `void convert(double maxH, double maxS, double maxL, double maxA)` : convertit la couleur vers la même, mais avec des maximums différents.

Exemple

Création et conversions d'une couleur.

```
1 ilm::Hsl hsl(60., 0.5, 0.25);
2 std::cout << "Couleur (max = 360, 1, 1, 1)        : "<< hsl << std::endl;
3 hsl.convert(1., 1., 1., 1.);
4 std::cout << "Couleur (max = 1, 1, 1, 1)          : "<< hsl << std::endl;
5 hsl.convert(100., 100., 100., 100.);
6 std::cout << "Couleur (max = 100, 100, 100, 100)  : "<< hsl << std::endl;
```

```
Couleur (max = 360, 1, 1, 1)        : 60        0.5     0.25    0
Couleur (max = 1, 1, 1, 1)          : 0.166667  0.5     0.25    0
Couleur (max = 100, 100, 100, 100)  : 16.6667   50      25      0
```

5.1.4 Conversions entre couleurs de différents types

On peut librement convertir des couleurs d'un type vers un autre en utilisant ces convertisseurs donnés. Ils se nomment tous `convert` et effectuent la conversion par passage par référence (ce qui évite de devoir passer des maximums comme paramètres ; ils sont tenus compte automatiquement avec la couleur passée par référence).

Le premier argument est toujours la couleur à convertir, et le deuxième la couleur dans le nouveau type qui contiendra la couleur convertie.

- `void convert(RgbaI, RgbaF&)` : convertit une couleur RVBA aux composantes entières vers une couleur RVBA aux composantes flottantes ;
- `void convert(RgbaF, RgbaI&)` : conversion dans l'autre sens ;
- `void convert(RgbaF, Hsl&, uint8_t = 0)` : convertit une couleur RVBA aux composantes flottantes vers une couleur HSLA. Le troisième paramètre spécifie la manière de calculer la luminosité, voir ci-dessous ;
- `void convert(Hsl, RgbaF&, uint8_t = 0)` : conversion dans l'autre sens ;
- `void convert(RgbaI, Hsl&, uint8_t = 0)` : convertit une couleur RVBA aux composantes entières vers une couleur HSLA ;
- `void convert(Hsl, RgbaI&, uint8_t = 0)` : conversion dans l'autre sens.

Lorsqu'on convertit une couleur RVB vers une couleur HSL, il existe plusieurs manières de calculer la luminosité, ce qui peut par exemple permettre de tenir compte du fait que pour une certaine quantité de vert, une couleur paraitra plus claire que pour la même quantité de bleu. Ce qui peut aider à rendre plus uniforme ou représentatif certains graphiques qui représentent une grandeur par la luminosité. Quatre possibilités sont offertes pour le moment :

- Mode par défaut (si le paramètre est 0 ou aucun de ceux ci-dessous) : la luminosité est donnée par la moyenne des valeurs maximale et minimale des composantes ;
- Mode moyenne (si le paramètre est 1) : la luminosité est donnée par $\frac{1}{3}(r+g+b)$;
- Mode Rec 601 (si le paramètre est 2) : la luminosité est donnée par $0.299r + 0.587g + 0.114b$;
- Mode Rec 709 (si le paramètre est 3) : la luminosité est donnée par $0.2126r + 0.7152g + 0.0722b$.

Exemple

Conversions d'une couleur RVB vers une couleur TSL.

```
1 ilm::RgbaI rgbai(192, 255, 238);
2 ilm::Hsl hsl;
3 std::cout << "Couleur RVBA : "<< rgbai << std::endl;
4 ilm::convert(rgbai, hsl);
5 std::cout << "Couleur TSLA : "<< hsl << std::endl;
```

```
Couleur RVBA : 192      255      238      0
Couleur TSLA : 163.81   1        0.876471        0
```

6. Divers

6.1 Projets d'amélioration

Des améliorations sur plus ou moins le long terme de la bibliothèque sont prévues et seront effectuées durant mon temps libre. Les fonctionnalités suivantes semblent envisageables dans une prochaine petite version `iloMath 0,96`, dont la date de sortie est indéterminée :

- Écriture et amélioration des fonctions comme log ou sin avec les nombres arbitrairement grands et précis ;
- Support de préimages arbitraires (pour le moment, seules des préimages régulièrement espacés sur un intervalle peuvent être entrés ;
- Entrée de données numériques (du genre paires préimages-images) et interpolation (fonctions à une variable) ;
- Opérations entre fonctions, dérivées ;
- Gestion des matrices (il y a déjà les tenseurs, mais le support est extrêmement basique et limité) ;
  - Somme, différence, multiplication matricielle, déterminant ;
  - Inversion d'une matrice carrée ;
  - Diagonalisation d'une matrice carrée ;
- Implémentation de géométrie élémentaire : points, droites et segments, polygones, cercles, aire, périmètre, déterminer si deux figures géométriques se chevauchent.

Encore plus tard dans le futur, les fonctionnalités suivantes pourraient voir le jour. En tout cas, on ne passera pas à la version `iloMath 0,99` tant qu'elles n'auront pas été implémentées.

- Ajout d'autres fonctions comme $erf(x)$;
- Résolution d'équations simples ;
- Groupes cycliques, division polynomiale ;
- Courbes paramétriques sur le plan ;
- Intégration numérique.

D'autres idées n'ayant pas beaucoup à voir avec le reste pourraient aussi être implémentées comme la projection d'objets géométriques simples en 4D, Mandelbrot, automates cellulaires, etc. Mais, ça ne sera pas pour tout de suite.

6.2 Fin, contribuer

Et voilà, on arrive au bout. J'espère que cette bibliothèque vous sera utile pour vos projets. Merci encore pour votre intérêt pour iloMath.

Si vous l'avez apprécié et que vous souhaitez me remercier, vous pouvez m'écrire, signaler des bogues, ou me faire un don en cybermonnaie pour m'aider à payer l'hébergement et le nom de domaine du site :

- Bitcoin : `1EZ3g68314WFcxFVttgeqQpLe1z2Z8GL8g` ;
- Ethereum : `0x32de6b854b6a05448b4f25d4496990bece8a2862` / `iloMath.eth`.

Cela me motivera également pour fournir plus régulièrement de nouvelles fonctionnalités et améliorer encore le code existant !