BiliB — Rapport

Programmation objet × Mathématiques pour l'informatique — IMAC2

Emily-Rose STRICH & Quentin HUET

Tableau bilan

Voici le tableau récapitulatif des fonctionnalités de notre librairie :

	Demandé	Bonus
Codé et fonctionnel	Classe Ratio avec : - constructeurs (par défaut, copy, en donnant numérateur et dénominateur) - getters et setters - gestion du zéro Conversion de nombre réel en nombre rationnel, prenant en compte les réels négatifs Opérateurs (entre rationnels et rationnel avec un réel) - opérateur addition + - opérateur soustraction - - moins unaire - opérateur division / (rationnel et réel) - opérateur multiplication * (rationnel et réel) Opérateurs de comparaisons - opérateur égal == (ratio et réel) - opérateur non égal != (ratio et réel) - opérateurs >, <, ≥ et ≤ (dans les deux sens)	Notre classe Ratio est en template et utilise des fonctions en constexpr Opérateurs - opérateur modulo % - opérateur incrémentation ++ - opérateur décrémentation -- Fonctions - conversion de rationnel en réel - fonction reste Fonctions variadiques - addition - soustraction - produit - minimum - maximum

Opérations - cosinus - sinus - tangente - exponentielle - logarithme - puissance - racine carrée Fonctions - fonction réduction de la fraction - fonction valeur absolue - fonction partie entière - fonction inverse Affichage - opérateur out stream (<<)	Opérateurs d’assignation (entre rationnels et rationnel avec un réel) - opérateur assignation = - opérateur += - opérateur -= - opérateur *= - opérateur /= - opérateur %=
Codé, non fonctionnel	
Codé, pas testé	
Pas codé	- opérateur d’assignation = dans le sens “variableRéelle = nombreRationnel” - représentation de l’infini

Bibliothèque de rationnel

A) Structure de donnée pour un nombre rationnel

Notre classe Ratio possède deux attributs : le numérateur et le dénominateur. Ils sont tous deux instanciés en int par défaut mais la classe est en **template** donc il est possible d'utiliser d'autres types d'int. Par ailleurs, toutes les fonctions de notre classe sont en **constexpr**.

Le signe du nombre rationnel est toujours attribué au numérateur : si le dénominateur donné est inférieur à 0, alors notre nombre rationnel prend comme numérateur "moins le numérateur donné" (moins unaire). De cette manière, c'est toujours le numérateur qui indique le signe de nos nombres rationnels.

On récupère les numérateur et dénominateurs a et b. Si b est négatif :

$$ratioNumber = \frac{-a}{|b|}$$

Nos nombres rationnels sont toujours représentés par des fractions irréductibles grâce à notre fonction *reduce()* qui divise le numérateur et le dénominateur par leur PGCD. Cette fonction est en *void*, elle modifie directement le nombre rationnel.

Un nombre rationnel est par défaut initialisé à 0, qui est toujours noté $\frac{0}{1}$.

B) Opérateurs

Somme, produit et inverse

La somme de deux rationnels, le produit de deux rationnels et l'inverse d'un rationnel ont été implémentés comme donnés dans le sujet : $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$, $\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$, $(\frac{a}{b})^{-1} = \frac{b}{a}$

Pour la division entre rationnels, on utilise la formule suivante :

$$\frac{\frac{a}{b}}{\frac{c}{d}} = \frac{ad}{bc}$$

Fonctions mathématiques

Pour les fonctions complexes telles que racine carré, puissance, exponentielle, logarithme, cosinus, sinus et tangente, nous appliquons leur fonction dédiée de la STD sur la version reconverti en nombre du rationnel. Le résultat est ensuite converti à nouveau en rationnel.

Ainsi, on peut facilement appliquer toute opération mathématique directement avec notre type rationnel. Voici un exemple avec cosinus :

```
Function cosinus
  Input: ratioNumber : nombre rationnel qui subit l'opération
  return convertToRatio( std::sin( convertToNumber( ratioNumber )));
```

Moins unaire

Le moins unaire inverse simplement le numérateur : $-\left(\frac{a}{b}\right) = \frac{-a}{b}$.

Valeur absolue

Pour la fonction valeur absolue, on ne change pas le nombre s'il est positif et on renvoie son inverse s'il est négatif. On suit l'algorithme suivant (utilisant le moins unaire précédemment implémenté) :

```
Function absolute
  Input: ratioNumber : nombre rationnel qui subit l'opération
  if ratioNumber.numerator < 0
    return -ratioNumber
  else
    return ratioNumber
```

Partie entière et reste

La fonction partie entière (ou *floor*) renvoie simplement un entier égal au numérateur divisé par le dénominateur.

La fonction reste renvoie le numérateur modulo le dénominateur.

Modulo

La fonction modulo divise un nombre rationnel par un autre puis renvoie son reste. Si le nombre en face n'est pas un rationnel, il est converti.

Produit ratio*float et float*ratio

Pour multiplier un rationnel par un nombre réel, on convertit le nombre à virgule flottant en rationnel puis on multiplie les deux rationnels que nous avons.

La fonction de multiplication dans l'autre sens (nombre réel multiplié par un rationnel) renvoie à l'opération dans le sens "rationnel fois réel". On peut simplement faire ça car la multiplication est commutative.

Opérateurs de comparaisons

Pour les opérateurs booléens = et ≠ entre deux rationnels, on compare séparément l'égalité du numérateur ET du dénominateur.

Pour les comparaisons "supérieur" et "inférieur", on utilise les rationnels mis au même dénominateur et on compare les deux numérateurs. Pour cela, on multiplie chaque numérateur par le dénominateur de l'autre rationnel.

$$\frac{a}{b} < \frac{c}{d} \iff \frac{ad}{bd} < \frac{cb}{bd} \iff ad < cb$$

Opérateurs d'assignation

Nous avons ajouté à notre classe Ratio la surcharge des opérateurs d'assignation pour l'addition, la soustraction, la division, la multiplication, le modulo et évidemment l'assignation simple - le "=". Ces opérateurs utilisent leur version classique que nous avons codés avant (par exemple, += utilise le + qui est déjà surchargé dans notre classe ratio).

Ces opérateurs d'assignation sont disponibles pour deux rationnels, et pour un rationnel et une valeur, implémentés dans la classe. Sous la classe, il y a les versions pour le sens valeur - rationnel. Il ne manque que l'opérateur = pour l'assignation "valeur = rationnel", que nous n'avons pas réussi à implémenter.

Opérateurs d'incrémentation et de décrémentation

Nous avons aussi implémenté les opérateurs d'incrémentation et de décrémentation (++ et --). Ces opérateurs utilisent respectivement les opérateurs d'assignation += et -= pour ajouter ou soustraire 1 au nombre rationnel. Ces fonctions sont de type *void* pour qu'elles modifient le rationnel en lui-même, sans en renvoyer un nouveau, et qu'on puisse seulement écrire "++myRatio", par exemple.

Fonctions variadiques

La bibliothèque propose plusieurs fonctions **variadiques**.

Il y a les fonctions *addition*, *subtraction* et *product*, qui servent à obtenir un rationnel en donnant plusieurs arguments à la fonction. Cependant, elles ne sont pas très utiles car il suffit juste d'utiliser les opérateurs +, -, et x consécutivement. On laisse donc à l'utilisateur le choix quant à la façon de calculer ses sommes et produits.

Il y a également les fonctions *min* et *max*. Celles-ci sont plus utiles, puisqu'elles servent à comparer plusieurs rationnels d'un coup, et d'en obtenir le minimum et le maximum. Ces fonctions comparent tous les arguments un à un et conserve le plus petit ou le plus grand pour le renvoyer quand il n'y a plus d'arguments à traiter.

C) Conversion d’un réel en rationnel

Dans l’algorithme proposé, la puissance -1 et la somme (lignes 8 et 12) s’adressent au nombre rationnel que l’on crée.

Nous avons modifié l’algorithme pour qu’il puisse convertir les nombres réels négatifs. Avant toute opération, on teste si le nombre est négatif ; si c’est le cas, alors on applique la conversion sur le négatif du nombre réel (moins unaire). On retournera enfin le négatif du ratio obtenu.

Les nombres à virgule flottante ne sont parfois pas tout à fait égaux à 0 alors qu’ils devraient l’être : cela cause des problèmes pour une des conditions d’arrêt de l’algorithme qui est quand x est égal à 0. Pour pallier ce problème, nous avons remplacé le test “x == 0” par “x < 0.01”. Le nombre à virgule que nous avons choisi arbitrairement est satisfaisant d’après nos tests.

D) Analyse

En C++, les nombres sont codés sur un nombre fini de bits. Ainsi, certains nombres trop grand ou trop petits ne peuvent pas être gérés par l’ordinateur, qui va s’arrêter à une certaine limite et approximer le nombre.

```
Ratio enorme = Ratio(556565, 48556)*Ratio(23331, 5131313)*Ratio(3215, 454);
std::cout << enorme.convertToNumber() << std::endl;

// renvoie -1.2745
```

Les long int vont de -2 147 483 648 à 2 147 483 647, et les long double vont de $-3.4*10^{-4932}$ à $3.4*10^{4932}$.

Cette limite fait que des nombres à virgules très précis seront très rapidement mal représentés et approximés. La taille des variables reste donc un problème.

Les très grands nombres auront également du mal à être représentés car le ratio que l’ordinateur cherchera à créer aura des valeurs extrêmes, même si on s’assure à chaque opération que nos fractions sont irréductibles.

Au fur et à mesure des opérations, il est possible que le numérateur et le dénominateur puissent dépasser la limite de représentation des entiers en C++ alors que la fraction est déjà irréductible. Une solution pour résoudre ce problème pourrait être de décomposer le rationnel en une addition de plusieurs rationnels plus petits, à la manière d’une décomposition en éléments simples.

Programmation

A) Exemples d’utilisation

Dans notre dossier *examples*, le fichier *main.cpp* montre notre batterie d’exemples. En le compilant et en l’exécutant, on peut voir quelques résultats dans le terminal. Le fichier est séparé en plusieurs parties grâce à des commentaires :

- Constructeurs
- Getter et Setter
- Fonctions
- Opérateurs arithmétiques
- Opérateurs d’assignation
- Opérateurs de comparaison
- Min et Max
- Fonctions mathématiques

Dans le fichier, on retrouve la syntaxe pour implémenter un rationnel. Ensuite, chaque fonction est utilisée et expliquée entre guillemets dans un *std::cout*, pour que l’utilisateur sache à quoi elle correspond dans le fichier, ainsi que dans le terminal.

Les opérateurs arithmétiques et d’assignation sont présentés avec deux rationnels, avec un rationnel et une valeur, ou avec une valeur et un rationnel.

C’est aussi le cas des opérateurs de comparaison qui sont présentés avec l’opérateur ternaire “?”.

On y montre aussi que pour faire une opération comme la somme, on peut tout aussi bien utiliser l’opérateur + consécutivement, ou la fonction variadique *addition*.

Ensuite, grâce aux rationnels implémentés tout au long des exemples, on cherche le minimum et le maximum.

Enfin, il y a quelques exemples des fonctions mathématiques comme l’exponentielle.

Par ailleurs, tous les *std::cout* servent aussi à montrer comment afficher le rationnel, sans passer par les getters.

On utilise également des valeurs de types différents, comme int et float, pour montrer à l’utilisateur que c’est possible.

B) Tests unitaires

Pour les tests unitaires, notre CMake utilise la version de **GoogleTest** qui est donné avec notre projet.

Les tests unitaires sont rangés dans des fichiers différents, suivant les différentes parties de notre classe, indiquées en commentaires.

Nous avons décidé arbitrairement de tester nos fonctions avec des valeurs choisies plutôt que des nombres aléatoires.

Chaque fonction est testée — sauf les getters et setters — et dans les différentes configurations possibles. Par exemples, les opérateurs sont testés pour deux rationnels, pour un rationnel et une valeur, et pour une valeur et un rationnel. Ces valeurs sont prises avec des types différents, comme int, float ou double.

Lorsque la fonction testée doit renvoyer un rationnel, on compare son numérateur et son dénominateur avec celui attendu.

Pour les opérateurs de comparaison, on vérifie que l'hypothèse de comparaison est bien vraie ou fausse.

Cependant, pour les opérations mathématiques, comme il est difficile de comparer des nombres à virgule et qu'il ne sont jamais tout à fait égaux, on instancie une erreur. Il s'agit de l'écart entre la valeur obtenue par la fonction de la classe, et la valeur de la vraie fonction mathématique.

Par exemple :

```
long double error = ratio.exp().convertToNumber() - std::exp(3);
```

Elle est de type long double pour une grande précision et on utilise sa **valeur absolue**.

Pour tester la fonction, on vérifie que cette erreur est très petite. On la compare à un nombre choisi arbitrairement :

```
ASSERT_EQ (std::abs(error) < 0.000001, true);
```

C) Compilation

Notre projet requiert la version 20 de c++. On s'assure ainsi que toutes les fonctions même les plus récentes comme std::gcd fonctionne le plus souvent possible.

Nous avons choisi d'intégrer la librairie GoogleTest directement dans notre projet car il y avait des gros soucis d'installation avec les ordinateurs de la fac. La solution que nous avons choisi permet au projet d'être compilé sereinement sur n'importe quelle distribution linux proprement installée (avec cmake).

L'arborescence de notre projet se présente comme telle :

```
/Bilib
|   /rapport
|   |   ● Vous êtes ici
|   /libRatio
|   |   /src
|   |   |   Ratio.cpp
|   |   /include
|   |   |   Ratio.hpp // Contient toutes les fonctions de notre classe Ratio
|   |   CMakeLists.txt
|   /examples
|   |   /src
|   |   |   main.cpp // Contient tous les exemples de notre classe Ratio
|   |   CMakeLists.txt
|   /tests
|   |   /src // Contient tous les tests unitaires fait avec GoogleTest
|   |   |   arithmeticOperators.cpp
|   |   |   assignmentOperators.cpp
|   |   |   constructors.cpp
|   |   |   functions.cpp
|   |   |   mathsFunctions.cpp
|   |   |   relationalOperators.cpp
|   |   |   variadicsFunctions.cpp
|   |   CMakeLists.txt
|   CMakeList.txt
```

Dans notre projet, il reste le fichier *Ratio.cpp*, créé pour implémenter nos fonctions, avant d'avoir passé notre classe en t**emplate**. C'est un fichier vide, mais nous n'avons pas réussi à le supprimer sans avoir de problèmes pour la compilation, même en changeant le CMake.

D) Documentation

Notre bibliothèque est documentée à l'aide de **Doxygen**, le fichier *index.html* est accessible au chemin suivant : build/INTERFACE/doc/doc-doxygen/html/index.html

E) Gestionnaire de versions

Nous avons utilisé le gestionnaire de versions **Git**, et l'avons hébergé sur Github. Grâce à cela, nous avons pu avancer simultanément sur des branches séparées.

https://github.com/emilyrosest/BiliB

> Merci pour votre lecture :)