Projet de Programmation Objet - Mathématiques pour l’Informatique

Partie programmation

Tableau bilan

ÉLÉMENTS DEMANDÉS, CODÉS ET QUI FONCTIONNENT	
CONSTRUCTEURS	
RationalNumber(*long int a*, *unsigned long int b*)	Constructeur à partir de deux entiers
RationalNumber(const RationalNumber *&val*)	Constructeur par copie
RationalNumber(const *double x*)	Constructeur à partir d’un réel. Il marche de paire avec la fonction de conversion que l’on détaillera plus loin
~RationalNumber()=default;	Destructeur
MÉTHODES	
RationalNumber **operator+**(const RationalNumber *&val*) const	Somme deux rationnels
RationalNumber **operator-** (const RationalNumber *&val*) const	Soustrait le nombre rationnel donné en paramètre au nombre rationnel actuel
RationalNumber **operator-**() const	Renvoie l’opposé d’un rationnel après avoir copié ce dernier. Le rationnel n’est pas modifié directement
RationalNumber **operator***(const RationalNumber *&val*) const	Multiplie deux rationnels
RationalNumber **inverse**() const	Renvoie l’inverse d’un rationnel
RationalNumber **abs**() const	Renvoie la valeur absolue d’un rationnel
int **integerPart**() const	Renvoie la partie entière d’un rationnel. Soit le résultat de la division euclidienne du numérateur par le dénominateur
RationalNumber **operator/** (const RationalNumber *&val*) const	Divise le nombre rationnel actuel par le nombre rationnel donné en paramètre
bool **operator <** (const RationalNumber *&val*) const bool **operator >** (const RationalNumber *&val*) const bool **operator ≥** (const RationalNumber *&val*) const	Comparent deux nombres rationnels et renvoient True si la comparaison est vraie, False sinon

bool **operator ≤** (const RationalNumber &*val*) const bool **operator==**(const RationalNumber &*val*) const bool **operator ≠** (const RationalNumber &*val*) const	
RationalNumber **irreducible**()	Simplifie la fraction pour la rendre irréductible. Divise le numérateur et le dénominateur par leur PGCD
int **pgcd**() const	Renvoie le pgcd du numérateur et du dénominateur pour l'utiliser dans la fonction **irreducible**()
RationalNumber **powRatio**(*int n*) const	Renvoie la copie d'un nombre rationnel à la puissance n
FONCTIONS	
std::ostream& **operator<<** (std::ostream& *stream*, const RationalNumber& *val*)	Surcharge de << qui permet l"affichage des rationnels
RationalNumber **convertDoubleToRatio**(*double x*, *int iter*)	Algorithme de conversion d'un réel en rationnel
RationalNumber **operator*** (const T& *x*, RationalNumber *ratio*)	Fonction template qui permet de multiplier un rationnel avec un réel, int etc…
RationalNumber **cosRatio**(RationalNumber *ratio*)	Renvoie le cosinus sous forme rationnelle d'un rationnel
RationalNumber **sinRatio**(RationalNumber *ratio*)	Renvoie le sinus sous forme rationnelle d'un rationnel
RationalNumber **sqrtRatio**(RationalNumber *ratio*)	Renvoie la racine carrée sous forme rationnelle d'un rationnel
RationalNumber **expRatio**(RationalNumber *ratio*)	Renvoie l'exponentielle sous forme rationnelle d'un rationnel
TESTS UNITAIRES	
TEST (RationalConstructors, intConstructor)	Teste le constructeur à partir de 2 entiers
TEST (RationalConstructors, copyConstructor)	Teste le constructeur par copie.
TEST (RationalConstructors,doubleConstructor)	Teste le constructeur à partir d'un réel. Il teste par la même occasion la fonction de conversion **convertDoubleToRatio()** puisque le constructeur l'appelle pour générer le nombre rationnel
TEST (RationalArithmetic, plus)	Teste l'opérateur +
TEST (RationalArithmetic, minus)	Teste l'opérateur -
TEST (RationalArithmetic, multiply)	Teste l'opérateur *
TEST (RationalArithmetic, inverse)	Teste l'opération inverse
TEST (RationalArithmetic, abs)	Teste l'opération valeur absolu
TEST (RationalArithmetic, divide)	Teste l'opération diviser
TEST (RationalArithmetic, pow)	Teste l'opération puissance
TEST (RationalArithmetic, cos)	Teste l'opération cosinus

TEST (RationalArithmetic, sin)	Teste l’opération sinus
TEST (RationalArithmetic, sqrt)	Teste l’opération racine
TEST (RationalArithmetic, exp)	Teste l’opération exponentielle
TEST (RationalComparators, strictInf)	Teste le comparateur <
TEST (RationalComparators, strictSup)	Teste le comparateur >
TEST (RationalComparators, inf)	Teste le comparateur ≤
TEST (RationalComparators, sup)	Teste le comparateur ≥
TEST (RationalComparators, equals)	Teste le comparateur ==
TEST (RationalComparators, different)	Teste le comparateur ≠
ÉLÉMENTS DEMANDÉS, CODÉS ET QUI NE FONCTIONNENT PAS	
LA GESTION D’EXCEPTIONS	
assert ((denominator != 0) && "division by 0")	Dans le constructeur à partir de deux entiers, empêche l’utilisateur de rentrer la valeur 0 au dénominateur
assert ((*ratio*.numerator >= 0) && "error: sqrtRatio : value should be positive")	Dans la fonction racine carrée, empêche l’utilisateur de demander la racine d’un nombre négatif
if (numerator == 0) **throw** std :: string (" inverse this number will lead to a division by zero ")	Dans la fonction d’inversion, prévient l’utilisateur qu’il essaye d’inverser 0 et que cela conduit à avoir 0 au dénominateur et donc une division par 0

La fonction de conversion

D’un point de vue algorithmique, la fonction de conversion est celle qui nous a posé le plus de difficultés.

Après analyse, nous avons rapidement compris que nous aurions besoin d’implémenter d’abord les fonctions d’opération usuelles sur les rationnels. Notamment l’opération + et la fonction inverse.

Gérer les négatifs

Pendant un long moment, nous ne savions pas comment gérer les nombres négatifs avec cette fonction. Nous avons pensé à stocker le signe dans une variable mais nous n’arrivions pas à implémenter cette solution correctement. C’est après plusieurs sessions de travail que nous avons compris que la solution tenait en une ligne : utiliser le moins unaire pour renvoyer l’opposé de la fonction de conversion de la valeur absolue du réel en paramètre.

```
if (x<0)
    {
        return -convertDoubleToRatio(std::abs(x),iter);}
```

La précision

Une fois le problème de gestion des négatifs résolu, nous avons fait face à une autre difficulté. Plus d'une fois sur deux, notre constructeur pour les réels ne passait pas son test unitaire. Les fractions retournées avaient pour numérateur et dénominateur des entiers extrêmement longs, dû à l'enchaînement des opérations entre rationnels dans l'algorithme de conversion. En pratique, la diminution du nombre d'itérations a réglé ce problème dans une grande majorité des cas, mais nous reviendrons dessus dans l'analyse mathématique.

Les types

En règle générale, les soucis auxquels nous avons fait face concernaient les types des variables manipulées. Nous avons pris le parti de coder le dénominateur en long unsigned int, afin de laisser au numérateur seul la gestion du signe, ce qui perturbait le bon fonctionnement de la plupart de nos fonctions. En effet, lors des opérations entre long int et unsigned long int, le long int est casté en unsigned long int, ce qui induit constamment une erreur dans les tests unitaires lorsque les rationels ne sont pas tous les deux positifs. Nous avons donc casté le dénominateur en long int chaque fois que nous effectuons ce genre d'opérations.

Nous avons aussi dû faire attention à la division numérateur/dénominateur, qui renvoie un entier alors que nous avons besoin de comparer deux réels, notamment dans les tests unitaires. Il faut donc caster le numérateur en double pour éviter tout problème.

La gestion d'exceptions

La gestion d'exception a été codée en dernier, et par manque de temps nous n'avons pas pu approfondir la réflexion pour savoir ce qui posait problème. Depuis que les lignes ont été codées, le programme réagit bien lorsque l'on fait les erreurs concernées par la gestion d'exceptions. Au moment de le lancer, il s'arrête si l'utilisateur tente de construire un rationnel avec un 0 au dénominateur, ou bien demande une racine d'un nombre négatif. Cependant, il ne renvoie pas le message d'erreur clair qu'il devrait renvoyer. Il s'arrête également complètement lorsque l'utilisateur demande l'inversion de 0 alors qu'il devrait juste renvoyer un avertissement.

Partie mathématique

Opérations sur les rationnels

L'opérateur de division

Lorsque l'on travaille avec des nombres rationnels, "diviser" correspond à "multiplier par l'inverse".

$\frac{a}{b} / \frac{c}{d} = \frac{a}{b} * (\frac{c}{d})^{-1} = \frac{a}{b} * \frac{d}{c}$

L’opération racine carré

Même en utilisant les règles de calcul de la racine, il nous a semblé nécessaire de repasser par les réels pour calculer la racine d’un rationnel. En effet, prendre la racine du numérateur ne garantit pas qu’il reste entier.

$\sqrt{\frac{a}{b}} = \frac{\sqrt{a}}{\sqrt{b}}$

La méthode que nous avons utilisée consiste à calculer les racines respectives du numérateur et du dénominateur pour obtenir deux réels que l’on divise pour obtenir le résultat de la fraction. Enfin, on applique l’algorithme de conversion pour revenir à un nombre rationnel.

Une autre idée aurait pu être de d’abord convertir les racines respectives du numérateur et du dénominateur en nombre rationnel pour ensuite appliquer l’opérateur division des rationnels.

L’opération cosinus

Nous n’avons pas trouvé d’autre méthode que de convertir le nombre rationnel en réel, d’y appliquer le cosinus, puis de convertir le nombre en rationnel.

$\cos \frac{a}{b} = \cos c = d = \frac{e}{f}$

a, b, e, f des entiers et c, d réels

L’opération sinus

Le raisonnement est le même que pour le cosinus.

$\cos \frac{a}{b} = \cos c = d = \frac{e}{f}$

a, b, e, f des entiers et c, d réels

L’opération puissance

En utilisant les règles de calcul de la puissance, on se ramène à calculer les puissances du numérateur et du dénominateur séparément. Or, la puissance d’un entier donne un entier.

$(\frac{a}{b})^k = \frac{a^k}{b^k} = \frac{c}{d}$

a, b, c, d des entiers

Nous avons considéré que k était un entier et nous n’avons pas traité le cas des puissances réelles ou rationnelles.

L’opération exponentielle

Le raisonnement est le même que pour le cosinus.

$e^{\frac{a}{b}} = e^c = d = \frac{e}{f}$

a, b, e, f des entiers et c, d réels

Les tests unitaires

Le défi principal lors de l'implémentation des tests unitaires était d'éviter au maximum la reconversion des rationnels en réels afin de ne pas perdre en précision. Nous avons eu du mal à trouver des solutions sans faire appel aux réels à nouveau. Les quelques tests qui utilisent complètement les rationnels sont les suivants, et voici les propriétés utilisées :

La fonction inverse :

$$si\ \ num1 = \frac{a}{b}\ \ et\ \ num2 = num1^{-1} = \frac{b}{a}\ \ alors\ \ num1 * num2 = 1$$

Les comparateurs (qui fonctionnent tous sur le même principe) :

$$si\ \ \frac{a}{b} > \frac{c}{d}\ \ alors\ \ ad > bc$$

Et la fonction division, qui utilise l'inverse:

$$si\ \ num1 = \frac{a}{b}\ \ et\ \ num2 = \frac{c}{d}\ \ alors\ \ num1/num2 = num1 * num2^{-1}$$

Le reste des tests se base sur la presque égalité des résultats de l'opération effectuée avec nos méthodes et l'opération effectuée après reconversion en réels. Nous nous demandons donc si certains tests prouvent vraiment le bon fonctionnement de nos méthodes ou si nous ne faisons que nous répéter, mais nous n'avons pas trouvé d'autre solution.

Pour certaines méthodes, comme le cosinus ou la racine carrée, nous passons par les réels et revenons aux rationnels. Ceci entraine deux conversions et donc plus d'erreurs.

Dans le cas de la fonction puissance, le test unitaire ne fonctionne quasiment jamais. Les méthodes utilisées nous semblent pourtant correctes. Nous pensons donc que l'erreur vient du fait que le numérateur et le dénominateur se retrouvent rapidement très grands et qu'il doit donc y avoir des erreurs d'approximation. De plus, dans le test unitaire, pour comparer les résultats, nous convertissons tout en réel, ce qui ajoute encore des approximations.

Une solution à ce problème de conversion dans les tests unitaires aurait été de surcharger *ASSERT_EQ()* pour que cette fonction puisse prendre des rationnels en paramètre. Cependant, nous ne savons pas si c'est possible et nous n'avons pas eu le temps d'explorer cette piste.

Les limites mathématiques de notre bibliothèque

La représentation des grands nombres

Instinctivement, nous avons pensé que la représentation d'un grand nombre en rationnel impliquait un numérateur supérieur à ce nombre. Ce qui signifierait une croissance rapide du numérateur si le nombre de base est très très grand. Cependant, nous avons vite trouvé des contre-exemples à cette théorie.

Peut-être alors que le fait qu'un nombre soit grand multiplie les chances d'imprécisions dans les calculs lors de la conversion. Bien qu'elles soient minimes, lors de la conversion d'un

grand nombre elles s'accumulent et finissent par ne plus être négligeables.

Ceci étant dit, nous n'avons pas de réponse exacte à donner à cette question. 🤷‍♀️🤷

L'enchaînement des opérations

Comme indiqué dans le sujet, nous avons bien remarqué que lorsque les opérations entre rationnels s'enchaînent, le numérateur et le dénominateur peuvent prendre des valeurs très grandes. Il faudrait alors pouvoir séparer en deux la fraction résultat, et retourner la somme d'un entier (représenté par une fraction dont le dénominateur est 1) et d'une fraction réduite. Par exemple :

$\frac{6}{5} + \frac{7}{8} = \frac{83}{40} = \frac{2}{1} + \frac{3}{40}$

En généralisant, cela revient à effectuer la division euclidienne du numérateur par le dénominateur. On l'écrit sous la forme :

$$numerateur = q * dénominateur + r$$

avec q le quotient de la division et r le reste. On peut ensuite séparer la fraction, et reprendre les calculs avec la forme simplifiée :

$\frac{num}{denom} = \frac{q*denom+r}{denom} = \frac{q}{1} + \frac{r}{denom}$

L'exemple de 0.3

Certains réels ne sont pas correctement convertis. Prenons l'exemple de 0.3 :

RationalNumber(0.3) renvoie :

$\frac{2632434403619972241}{8774566888274794589}$

Alors que :

$0.3 = \frac{3}{10}$

Dans un premier temps, nous avons pensé que l'erreur venait du fait d'utiliser des *float*, nous avons donc utilisé des *double,* mais cela n'a rien changé. En utilisant le débugueur, nous avons remarqué que dès l'appel de la fonction 0.3 s'affichait comme 0.29999999…

En reprenant notre cours, nous avons converti nous même 0.3 en binaire :

0 01111110 0 1001 1001 1001 1001 …

Le motif 1001 se répète à l'infini.

Finalement, c'est à ce moment précis que le but de ce projet a refait surface. Après avoir passé du temps sur tout le reste, nous avions presque oublié la problématique : certains nombres comme 0.3 ne peuvent pas être représentés de manière exacte avec les *float*, *double*, *long double* etc. C'est pourquoi passer par une bibliothèque rationnelle qui représenterait 0.3 comme le couple (3,10) permettrait d'être exacte.

Une solution à notre problème serait de changer l'algorithme de conversion pour qu'il interprète chaque nombre comme une suite de chiffres. Puis, en fonction de la position de

chaque chiffre dans cette suite, par rapport à la virgule, il pourrait en déduire le nombre de dixièmes, centièmes, millièmes, etc. puis faire l’addition.

Par exemple,

$1.236 = \frac{12}{10} + \frac{3}{100} + \frac{6}{1000} = \frac{6}{5} + \frac{3}{100} + \frac{3}{500} = \frac{600}{500} + \frac{15}{500} + \frac{3}{500} = \frac{600+15+3}{500} = \frac{618}{500} = \frac{309}{250}$

Les nombres irrationnels

$\sqrt{2},\ \pi,\ \phi$

Il est important de noter aussi que certains nombres ne sont pas rationnels. Dans ces cas là, notre bibliothèque n’est pas utile.