IMAC2

Mathématiques pour l'informatique
Programmation orientée objet
Rapport

Les nombres rationnels

Élèves :
Axel Dona
Aurélien Droissart

Enseignant :
Vincent Nozick

4 janvier 2023

Table des matières

1 Tableau bilan : Programmation

Description	Élément demandé	Codé	Fonctionnel
Fonction de transformation d'un rationnel sous forme d'une fraction irréductible	OUI	OUI	OUI
Fonction de conversion d'un réel en rationnel	OUI	OUI	OUI
Valeur absolue	OUI	OUI	OUI
Partie entière	OUI	OUI	OUI
Carrée	OUI	OUI	OUI
Racine carrée	OUI	OUI	OUI
Puissance k-ième	OUI	OUI	OUI
Logarithme	OUI	OUI	OUI
Cosinus	OUI	NON	NON
Exponentiel	OUI	NON	NON
Produit d'un flottant avec un rationnel	OUI	OUI	OUI
Produit d'un nombre rationnel avec un flottant	OUI	OUI	OUI
Opérateurs +,-,*,/,=	OUI	OUI	OUI
Opérateurs +=,-=,-unaire,/	OUI	OUI	OUI
Opérateurs <,>,<=,>=,==,!=	OUI	OUI	OUI
Affichage «	OUI	OUI	OUI
Test unitaire +,-,*,/	OUI	OUI	OUI

2 Partie mathématique

2.1 Formalisation des rationnels

Un nombre rationnel est un nombre qui peut s'exprimer comme le quotient de deux entiers relatifs : $\frac{a}{b}$. a étant le numérateur, et b le dénominateur, un entier relatif non nul, et dans le cadre du projet strictement positif. Le signe de la négation est porté par le numérateur.

Il y a un nombre infini de façons d'écrire un nombre rationnel mais on préfère l'écrire sous forme irréductible, c'est à dire que le numérateur et dénominateur sont premiers entre eux. On l'appelle fraction irréductible. Pour le projet nous avons décidé de créer uniquement des fractions irréductibles, ainsi notre constructeur de nombre rationnels appelle directement la fonction irréductible.

Algorithm 1 Function irreductible

$gcd \leftarrow (m\_numerator, m\_denominator)$;
$m\_numerator \leftarrow m\_numerator/gcd$;
$m\_denominator \leftarrow m\_denominator/gcd$;

Algorithm 2 Constructor

$numerator \leftarrow m\_numerator$;
$denominator \leftarrow m\_denominator$;
$this->irreductible()$;

Nous avons aussi créé une fonction ***display*** pour nous permettre d'afficher nos nombres rationnels sous la forme $numerator/denominator$.

Et nous avons également surchargé l'opérateur « pour pouvoir directement afficher un rationnel avec $std :: cout << rn$;

2.2 Opération sur les rationnels

2.2.1 Opérateur de base

Dans le processus de création de notre classe, nous surchargeons les opérateurs binaires classiques pour pouvoir faire des opérations avec les nombres rationnels.

Nous avons donc surchargé le +, -, *, /, =. Ici pour les additions entre nombres rationnels :

Algorithm 3 Operator +

Input rationalNumber rn

rationalNumber temp ;

$temp.m\_numerator \leftarrow (m\_numerator * rn.m\_denominator) + (m\_denominator * rn.m\_numerator)$
$temp.m\_denominator \leftarrow m\_denominator * rn.m\_denominator$

return temp ;

Mais aussi pour les additions entre nombres réels et rationnels, qu'il a fallu coder deux fois pour que la commutativité de l'opération soit garantie :

Algorithm 4 Operator +

Input un réel x ;
$rn = convertRealToRatio(x, 20)$;

Return *this+rn ;

Pour faire des produits de nombre rationnels nous utilisons cette expression :

$$\frac{a}{b} * \frac{c}{d} = \frac{ac}{bd} \tag{1}$$

Et nous en déduisons le quotient, qui revient à multiplier par l'inverse :

$$\frac{a}{b} / \frac{c}{d} = \frac{ad}{bc} \tag{2}$$

2.2.2 Autres opérateurs

Nous avons exploré d'autres opérations comme :

$(\frac{a}{b})^2$ et $\sqrt{\frac{a}{b}}$

Peu de difficulté pour le carré d'un rationnel car il suffit de faire le carré du numérateur ainsi que du dénominateur puis rendre la fraction irréductible.

Pour la racine carré, on doit s'assurer que le numérateur ou le dénominateur ne soient pas négatif. Puis faire la racine carrée du numérateur et dénominateur et convertir le résultat du quotient des racines en un rationnel. On perd donc de la précision dans le quotient et la conversion d'un réel en rationnel.

On a également exploré :

$(\frac{a}{b})^k$ et $\ln \frac{a}{b}$

Il est très pratique d'utiliser le logarithme néparien avec les rationnels car une des propriété du logarithme est que : $\ln \frac{a}{b} = \ln a - lnb$
On transforme donc un quotient en négation et on ne pert pas en précision. La perte de précision peut être dûe par la fonction logarithme seulement.

Pour la puissance k-ième voici un aperçu du code :

Algorithm 5 Puissance k-ième

Input : k un entier.
IF $k > 0$
rationalNumber temp(pow(m_numerator,k),pow(m_denominator,k) ;
return temp.irreductible() ;
ELSE IF $k == 0$
return 1,0 ;
ELSE IF $k < 0$
rationalNumber temp(pow(m_numerator, -k), pow(m_denominator, -k)) ;
return temp.inverse().irreducible() ;

Pour les puissances positives, il n'y a aucun problème : on peut directement calculer le numérateur et dénominateur à la puissance k-ième, mais pour les puissances négatives il nous faut inverser le quotient pour retourner le nombre rationnel.

2.3 Conversion d'un réel en rationnel

Les rationnels nous permettent de faire des opérations plus précises que sur des flottants, car on manipule seulement des nombres entiers qui sont représentés sur 32 bits sans approximations en C++. Cependant, utiliser des rationnels à une limite dans le cas où on doit faire des opérations entre nombres à virgule et un rationnel, ou pour certaines opérations où on est contraint de convertir un nombre rationnel en flottant ou l'inverse.

Pour ce faire nous avons utilisé la fonction présente dans le sujet, et implémenté la fonction qui fait le changement inverse, ***ConvertRatioToReal***, qui effectue le quotient du numérateur et du dénominateur.

3 Partie programmation

3.1 Test unitaire

Nous avons effectué des tests unitaires pour vérifier que nos opérateurs étaient corrects. Pour ce faire nous avons créé des rationnels avec des valeurs génerées aléatoiremens, puis nous avons effectué le test entre l'opération que l'on a codé au préalable et le résultat qu'on devrait avoir.

Nous avons effectué ces tests pour vérifier l'addition, la soustraction, le produit et le quotient de nombres rationnels.

Dans notre recherche de confection des tests unitaires nous avons trouvé la classe std : :ratio et notre première idée était de comparer les opérations de notre classe avec la classe $std :: ratio$. Cependant, la classe $std :: ratio$ ne peut pas prendre en paramètre des variables et effectue les calculs directement à la compilation. Nous n'avons donc pas réussi à implémenter des tests de comparaison de notre classe avec la classe $std :: ratio$.

3.2 Problèmes rencontrés

Nous avons rencontré des problèmes pour créer la fonction ***cos***. Nous n'avons pas trouvé de solution pour faire une opération sur les rationnels qui ne soit consiste pas à convertir le rationnel en flottant, effectuer la fonction cosinus puis reconvertir en rationnel, ce qui donne des résultats très loin de la réalité.

Le même problème se pose aussi pour l'exponentiel.

Un autre problème rencontré est l'utilisation de GoogleTest, puisque l'un de nous était sur PC et l'autre sur Mac. Il a fallu jongler entre des paramètres différents. Cela n'a pas empêché l'utilisation de GoogleTest mais cela a tout de même pris du temps sur le projet.

4 Conclusion

En conclusion, nous avons réussi selon nous à implémenter la majorité des fonctionnalités du projet. Cela nous a permis de mieux comprendre la gestion de nombres rationnels.

À notre avis, cette librairie de nombres rationnels est plus simple d'utilisation que std : :ratio. La syntaxe est plus simple et nous pouvons faire plus d'opérations.

La piste d'amélioration du projet serait d'améliorer la fonction de conversion pour augmenter la précision des opérations des nombres rationnels.