DÉCEMBRE 2022
JANVIER 2023

RAPPORT LIBRATIO

Librairie de nombres rationnels

RÉDIGÉ PAR

Charline Le Pape & Juliette Jeannin

Charline Juliette	Demandé et fonctionnel	Demandé et pas fonctionnel	Demandé et pas codé	Pas demandé et fonctionnel	Pas demandé et pas fonctionnel
Somme	X				
Différence	X				
Produit	X				
Division	X				
Inverse	X				
Racine carrée				X	
Log népérien					X
Exponentielle					X
Puissance				X	
Cos					X
Sin					X
Tan					X
Conversion réel en rationnel	X				
Fonction pour rendre rationnel irréductible	X				
Moins unaire	X				
Valeur absolue	X				
Partie entière		X			
Produit réel et rationnel	X				
Produit rationnel et réel	X				
==	X				
!=	X				
<	X				
>	X				
<=	X				
>=	X				
Surcharge <<	X				
Classe en template				X	
Assert et exceptions					X
Constexpr				X	

Constructeur

Notre constructeur prend des entiers (int, long int...). Si un dénominateur null est rentré et que le numérateur vaut 1 alors nous avons fait le choix de renvoyer le rationel 1/0 correspondant à l'infini. Sinon c'est 0/1 qui est renvoyé pour un dénominateur nul. Le constructeur fait en sorte que le dénominateur soit positif et que le rationnel soit irréductible.

Nous avons eu l'idée de surcharger le constructeur pour pouvoir créer un rationnel à partir de nombre flottant. Nous l'avons codé mais à la compilation il nous est dit que le constructeur est déjà défini. Nous avons retiré cette partie pour ne pas laisser de code mort mais nous aurions voulu réussir implémenter ceci.

Convertisseur

Fonction qui convertit un réel en rationnel

Tout d'abord nous avons déclaré une variable qui contient le signe du réel.
Ensuite nous nous sommes aperçues de la limite du modèle : en forçant le nombre d'itération de l'algorithme, celui-ci peut donner des résultats incorrects. Le nombre d'itérations est donc plafonné à 10, après avoir fait plusieurs tests, pour tenter d'éviter cela.
Nous avons par la suite reproduit l'algorithme fourni en prenant soin de multiplier par le signe à chaque fois (partie entière et partie décimale).
Néanmoins reste toujours un problème de précision qui fait défaillir nos unit tests... Dans ceux-ci nous avons décidé de regarder l'égalité du résultat exact avec le résultat trouvé à 3 décimales près mais cela n'arrive presque jamais.
Pas de souci en revanche pour les 2-3 exemples que nous avons essayés.

Réponses aux questions

Les grands nombres comme les très petits nombres se représentent très mal avec les rationnels car ils sont encodés en deux entiers mais qui parfois sont tellement grands qu'il n'y a pas assez d'espace mémoire pour eux. De plus on effectue des opérations comme la multiplication, la division, le produit ou l'exponentielle avec eux ce qui fait vite atteindre des valeurs extrêmes que l'on ne peut plus stocker en l'état.

On peut donc d'abord tenter d'utiliser des long int et des long long int pour que nos opérations restent cohérentes plus longtemps mais le même problème finira par se poser.

On peut également imaginer la classe Rational avec le numérateur et le dénominateur en tant que unsigned int et ajouter un attribut "signe" supplémentaire qui porterait le signe du rationnel.

On peut également créer la classe Rational avec le numérateur et le dénominateur en nombre en virgule flottante et en ajoutant un attribut entier exposant qui stockerait la puissance de 10 par laquelle il faut multiplier pour obtenir le rationnel.

On pourrait aussi envisager une classe numérateur et une classe dénominateur.

Opérateurs

Soustraction de deux rationnels (surcharge opérateur -)

Nous utilisons la même méthode que l'addition sauf qu'on soustrait :

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

Moins unaire d'un rationnel (surcharge opérateur -)

Nous faisons la négation au numérateur : $\frac{a}{b} = -\frac{a}{b}$

Division de deux rationnels (surcharge opérateur /)

Nous avons choisi de faire le produit d'un rationnel par l'inverse de l'autre rationnel.

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

Produit d'un rationnel avec un réel (surcharge opérateur *)

On commence par convertir le réel en rationnel avec la fonction floatToRatio puis on effectue le produit de ces deux rationnels avec la surcharge de l'opérateur prévue à cet effet.

Opérateurs

Comparaisons == et !=

Pour la comparaison ==, on compare tout simplement les numérateurs entre eux et les dénominateurs entre eux . Si les deux sont identiques on retourne vrai. La != retourne vrai quand == est faux.

Comparaisons >=, >, <= et <

Pour la comparaison >= , nous mettons les deux rationnels concernés au même dénominateur afin de pouvoir comparer leurs numérateurs. On retourne true si le numérateur du premier et plus grand que celui du second, false sinon.

Les autres comparaisons(>, <= et <) découlent de la surchage de >= :

- rat1<rat2 retourne faux si rat1>=rat2 est vrai
- rat1<=rat2 retourne vrai si rat1<rat2 ou rat1==rat2 est vrai
- rat1> rat2 retourne vrai si rat1<= rat2 est faux

Méthodes

Inverse d'un rationnel

Nous échangeons le numérateur et le dénominateur en faisant attention à ce que le numérateur n'égale pas 0 par l'appel au constructeur.

$$\left(\frac{a}{b}\right)^{-1} = \frac{b}{a}$$

Puissance

Nous avons fait le choix de réutiliser la fonction std::pow plutôt que de faire des boucles for ou une fonction récursive pour favoriser la rapidité de notre programme. Nous utilisons std::pow pour le numérateur et pour le dénominateur et nous créons un nouveau rationnel qui a pour numérateur lui-même élevé à la puissance renseignée et pour dénominateur lui-même élevé à la puissance renseignée.

Méthodes

Racine carrée d'un rationnel

La première étape est de bien vérifier que le numérateur est positif, étant donné que la racine carrée d'un nombre négatif n'existe pas en réel.
Si c'est le cas deux options pour le calculer s'offre à nous :
On note a un entier et a. l'entier convertit en flottant.

Soit on calcule sqrt(a/b) en faisant sqrt(a) et sqrt(b), en les convertissant en rationnels et en faisant le rapport de ces deux rationnels : cette méthode est légèrement moins précise et plus couteuse comme on fait deux racines carrées mais on utilise plus les rationnels pour faire les calculs.

Soit on fait directement sqrt(a./b.) qu'on convertit ensuite en rationnel : on est plus précis et on ne fait qu'une seul racine carrée.
Au final c'est cette méthode qu'on a choisirait de garder pour la librairie de nombres rationnels étant donné les approximations déjà faites dans les autres fonctions qui nous font nous éloigner du résultat exact.

Logarithme d'un rationnel

Nous vérifions que le numérateur est strictement supérieur à 0 pour calculer le logarithme.
Puis comme dans le cas de la racine carré nous avons plusieurs possibilités.

Nous avons d'abord calculé le log en utilisant la propriété log(a/b) = log(a) - log(b).
On calcul le logarithme du numérateur puis celui du dénominateur, on les convertit en rationnels et on retourne leur différence.
Mais on fait deux fois un log ce qui est couteux et en plus on réalise une soustraction qui sur des petits nombres peut engendrer des erreurs de précision non négligeables.

Nous avons alors décidé de faire le calcul de log(a./b.) directement et de convertir le résultat en rationnel avec une seconde fonction.

Méthodes

Absolu d'un rationnel

Nous avons juste besoin de faire la valeur absolue du numérateur : |a/b|=|a|/b

Partie entière

Pour trouver la partie entière nous passons par trois cas :

- Si la valeur absolue du numérateur est inférieure au dénominateur -> il n'y a pas de partie entière.
- Si la valeur absolue du numérateur est égale au dénominateur -> la partie entière vaut 1
- Sinon on fait une boucle où on compte le nombre d'itérations jusqu'à ce que la valeur absolue du numérateur ne soit plus supérieure au dénominateur.

Avant de renvoyer la partie entière on vérifie si valeur absolue de numérateur = numérateur si oui on retourne la partie entière si non on retourne son opposé.

Alors oui, nous aurions pu faire directement (int)(a/b) pour avoir la partie entière mais bon il faut bien s'amuser de temps en temps et se challenger.

Fonction cosinus

Tout d'abord nous avons implémenté un cosinus qui reprend le développement limité du cosinus en 0 (ou formule de Taylor). Cette formule est intéressante car elle permet d'avoir une expression du cosinus sans utiliser la fonction std::cos prédéfinie.
Cependant ceci n'est valable que pour des rationnels proches de 0. Il y a donc une version complète du cosinus qui utilise le développement de Taylor près de 0 et une autre méthode sinon. Cette dernière consiste à scinder le rationnel en un rationnel qui correspond à sa partie entière et en un autre qui correspond à sa partie décimale pour utiliser la formule cos(a + b) = cos(a)cos(b) - sin(a)sin(b). Ainsi pour la partie décimale on utilise le cos de Taylor et pour la partie entière on utilise std::cos. Cela permet d'utiliser au plus les rationnels dans le processus mais est très peu précis, voire incohérent... Les unit test échouent...

La seconde solution est d'utiliser le cosinus prédéfini pour le rapport du numérateur et du dénominateur.

Méthodes

Fonction sinus

A la manière du cosinus, nous avons implémenté un sinus qui reprend la formule de Taylor pour les rationnels proches de 0 et use d'une autre méthode sinon. Cette dernière consiste aussi à scinder le rationnel en un rationnel qui correspond à sa partie entière et en un autre qui correspond à sa partie décimale pour cette fois utiliser la formule sin(a + b) = sin(a)cos(b) - cos(a)sin(b). Ainsi pour la partie décimale on utilise le sin de Taylor et pour la partie entière on utilise std::sin.
On utilise les rationnels mais c'est aussi très peu précis, voire incohérent... Les unit test échouent également...

Comme pour le cos, la seconde solution est d'utiliser le sin prédéfini pour le rapport du numérateur et du dénominateur.

Fonction tangente

Pour calculer la tangente nous réutilisons les fonctions sin et cos que nous avons faites et nous faisons :

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

Fonction exponentielle

A l'instar du cos et du sin, la fonction exponentielle utilise le développement limité de l'exponentielle pour les rationnels proches de 0 et emploie une autre méthode pour les autres : on scinde le rationnel en un rationnel de sa partie entière et en un autre de sa partie décimale et on renvoie le produit des deux comme exp(a+b) = exp(a)*exp(b). Ainsi pour la partie décimale on utilise l'exp de Taylor et pour la partie entière on utilise std::exp. Idem utilisation des rationnels mais peu de précision, voire incohérences... Mauvais résultats aux unit tests...

On a aussi codé la seconde solution qui est d'utiliser l'exp prédéfinie pour le rapport du numérateur et du dénominateur afin de gagner en précision.

Méthodes

Produit d'un réel avec un rationnel

On convertit le réel en rationnel et on retourne le résultat du produit des deux. Nous avons d'abord chercher à trouver un moyen de surcharger l'opérateur * pour pouvoir opérer le produit d'un rationnel avec un réel dans l'autre sens mais sans succès. Nous avons donc créé cette fonction à l'extérieur de la classe Rational pour faire le produit dans le sens voulu.

Autres remarques

Nos instanciations dans les exemples fonctionnent bien mais souvent nos unit tests s'interrompent pour cause d'exception en point flottant... Nous nous sommes dit que ce problème devait arriver pour cause d'une division par 0 mais comme quasiment tout repasse par notre constructeur qui nous envoie à l'infini ou nous laisse à 0, nous n'avons pas trouvé comment régler ce problème des plus déplaisant.

Ce problème peut aussi être lié aux problèmes de dépassement de capacité des entiers.

Nous avons alors tenté d'utiliser des assert et de gérer les exceptions pour trouver la source mais sans succès, les assert arrêtant directement la compilation et les try and catch soulevant le problème mais soit en laissant arriver l'exception en point flottant, soit en renvoyant une autre valeur mais qui engendrait un assert non vérifié dans les unit tests...

Notre choix s'est porté sur la gestion des exceptions pour que vous puissiez voir certains résultats de nos unit tests malgré les erreurs.