Ratio

Quentin Augey
Boulanger Elisa

2 janvier 2023

Résumé

Projet de programmation objet - mathématiques pour l'informatique IMAC2

1 Introduction

Dans ce rapport, nous présentons notre implémentation d'une librairie permettant de manipuler les nombres rationnels en C++. Ce projet a consisté en la création d'une classe mettant à disposition les calculs et opérations basiques que nous pouvons faire avec des nombres rationnels. Vous retrouverez ici, les difficultés rencontrées et notre réflexion sur ce sujet.

Lien git : https://github.com/Raito777/libRational

2 Partie programmation

2.1 Tableau bilan

Fonctions	Fait	Problèmes	Fonctionne ?
constructeur à partir d'un numérateur et dénominateur	oui	non	oui
constructeur à partir d'un nombre à virgule	oui	non	oui
constructeur à partir d'un nombre rationnel	oui	non	oui
somme, produit, division de deux rationnels	oui	non	oui
somme et soustraction, d'un rationnel avec un nombre à virgule flottante	oui	oui	oui
somme et soustraction d'un nombre à virgule flottante avec un rationnel	non	oui	oui
produit et division d'un nombre à virgule flottante avec un rationnel	non	oui	oui
produit et division d'un rationnel avec un nombre à virgule flottante	oui	non	oui
inverse d'un rationnel	oui	non	oui
cosinus, sinus, tangente	oui	non	oui
exponentielle et logarithme	oui	non	oui
Racine et puissance d'un nombre rationnel	oui	non	oui
fonction pour transformer le rationnel sous forme de fraction irréductible	oui	non	oui
le moins unaire	oui	non	oui
les opérateurs de comparaison : $==, !=, >, <, \leq$ et $\geq$	oui	non	oui
une fonction d'affichage sous forme de surcharge de l'opérateur $\ll$	oui	non	non
la valeur absolue	oui	non	oui
la partie entière d'un ratio et d'un float	oui	oui	oui
la conversion d'un réel en rationnel	oui	oui	oui
la conversion d'un rationnel en réel	oui	non	oui

Table 1 – Fonctions implémentées.

Nous avons rencontré des difficultés liées à l'utilisation de template pour la classe. En effet, nous avons dû refaire à plusieurs reprises certaines parties du code, car l'application de template n'était

pas encore très clair pour nous. Cela a été une grande partie du projet, car en utilisant des template, c'est quasiment toutes les méthodes et opérateurs qui ont été affectés. La difficulté principale était au niveau des opérateurs arithmetiques car il fallait que l'on puisse les utiliser quelque soit le type (int, long int...) et quel que soit l'ordre, ratio + ratio, ratio + valeur, valeur + ratio. Nous avons passé pas mal de temps à tester différentes méthodes et nous y sommes arrivés avec la notion de "friend" vue en cours. Nous avons aussi dû rajouter une exception prévenant l'utilisateur qu'il ne peut pas créer de nombre rationnel avec comme numérateur et dénominateur des nombres à virgule flottante. Avoir codé notre classe comme cela nous permet donc de créer des ratios de différents types et de faire des opérations entre eux sans aucun soucis.

2.2 Compilation et Debug

Au niveau de la compilation, nous nous sommes aidés des tps que nous avions fait sur la documentation avec doxygen et les tests unitaires avec googleTest. Notre premier objectif était d'avoir un cmake fonctionnel, nous permettant de commencer sereinement le projet. Nous avons eu un peu de mal avec Doxygen car celui-ci ne semblait pas compiler. Mais après un peu de recherche dans nos tps, nous avons réussi.

Enfin, pour ce projet, nous n'avons pas mis en place de debug "propre", mais après le tp concernant ce sujet, nous avons constaté que cela nous aurait été bien utile, notamment pour la fonction convertFLoatToRatio. Cela nous aurait permis de mieux voir l'évolution des valeurs étape par étape.

2.3 Tests unitaires

Pour la partie tests unitaires, nous nous sommes concentrés sur les fonctionnalités primaires de la librairie. Nous avons essayé de tester chaque fonctionnalité juste après leur implémentation, mais les tests des constructeurs, ont été créés tous en même temps.

Une des difficultés lors de nos tests était de savoir comment tester un opérateur sans réutiliser ce même opérateur deux fois. Nous avons donc comparé un nombre rationnel aléatoire créé avec la fonctionnalité à tester et un autre nombre correspondant au bon résultat. Tous nos tests ont utilisé des nombres aléatoires (sauf celui du constructeur par défaut). Pour nous aider nous nous sommes servis du tp sur googleTest et de celui sur les nombres aléatoires.

De plus, nous avons rencontré quelques problèmes dus à l'utilisation de template sur notre classe, car au final, chaque test se fait avec un type de valeur, et nous ne testons pas pour l'instant les autres types.

2.4 La partie entière

Pour la fonction partie entière, une idée est venue très rapidement. Celle de prendre la valeur en question et de lui enlever 1 à chaque passage jusqu'à que cette valeur soit inférieure à 1.

Algorithm 1: Partie entière

```
1 Function int_Part_
      Input: x ∈ R+ : le nombre dont on cherche la partie entière
2     int iterator = 0
3     float error = 2.0
4     if x < 1 then return 0
5     if x == 1 then return 1
6     while error > 1 do
7         iterator++
8         error = x − iterator
9     return iterator
```

Cette fonction a aussi été réalisé pour les rationnels.

2.5 Fraction irréductible

Pour mettre une fraction sous forme irréductible il faut calculer le PGCD du numérateur et du dénominateur de la fraction. Puis il faut se servir de ce pgcd et l'appliquer pour avoir la fraction irréductible.

Algorithm 2: Fraction irréductible

```
1  Function reduce
       Input: r : le rationnel à réduire
2      if le type de r est entier then
3          int pgcd = pgcd(numerateur, denominateur)
4          numerateur = numerateur / pgcd
5          denominateur = denominateur / pgcd
6      else
7          exception : un rationnel doit être composé d'entiers
8      if r < 0 then
9          on met le signe au numérateur
10     return r
```

3 Partie mathématiques

3.1 Formalisation de l'opérateur /

— Division de deux rationnels :

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

3.2 Autres opérations

— La fonction racine a été facile à implémenter car

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

— La fonction cosinus a demandé un peu plus de réflexion puis la fonction sinus et tangente ont découlé facilement. on s'était dit au départ qu'on pourrait essayer d'escroquer cette partie en partant d'un principe plutôt simple :

$$\cos\frac{a}{b} = \frac{\cos\frac{a}{b}}{1}$$

Puis on s'est rendu compte que ça ne marchait pas du coup on a attendu d'implémenter ConvertFloatToRatio et ConvertRatioToFloat afin de les implémenter : D'où

$$\cos\frac{a}{b} = ConvertFloatToRatio(\cos(ConvertRatioToFloat(\frac{a}{b})))$$

$$\sin\frac{a}{b} = ConvertFloatToRatio(\cos(ConvertRatioToFloat(\frac{a}{b})))$$

$$\tan\frac{a}{b} = ConvertFloatToRatio(\cos(ConvertRatioToFloat(\frac{a}{b})))$$

— La fonction puissance a été facile à implémenter puisque

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

3.3 Conversion d'un réel en rationnel

Question : D'après vous, à quel type de données s'adressent la puissance 1 de la ligne 8 et la somme de la ligne 12 ?
La puissance -1 à la ligne 8 s'adresse à

$$\frac{1}{x}$$

qui est un float.
La somme de la ligne 12 est une somme de rationnelles qui va donner le résultat final.

Comment faire pour modifier l'algorithme ConvertFloatToRatio afin qu'il puisse gérer les nombre négatifs ?
Au départ on s'était dit que dans les ratio on pouvait coder son signe afin que ce soit plus simple dans la fonction en mettant valeur absolue et de return à la fin le ratio * son signe. Notre pseudo code était :

Algorithm 3: Conversion d'un réel en rationnel version 1

```
1  Function convert_Float_to_Ratio
       Input: x ∈ R+ : un nombre réel à convertir en rationnel
              nb_iter ∈ N : le nombre d'appels récursifs restant
2      int sign = 1
3      if x < 0 then sign = −1
4      // première condition d'arrêt
5      if x == 0 then return 0/1
6      // seconde condition d'arrêt
7      if nb_iter == 0 then return 0/1
8      // appel récursif si x < 1
9      if abs(x) < 1 then
10         return ( convert_float_to_ratio(abs(1/x), nb_iter) )^−1
11     // appel récursif si x ≥ 1
12     if abs(x) ≥ 1 then
13         q = ⌊x⌋ // partie entière
14         return (q/1 + convert_float_to_ratio(abs(x) − q, nb_iter − 1)) ∗ sign
```

Cependant en implémentant la fonction on s'est rendu compte que cela ne fonctionnait pas comme prévu. En réfléchissant, on s'est dit que plutôt que de coder le signe et de mettre x en valeur absolue, il suffisait de dire qu'à chaque fois que x est inférieur à 0 alors cela retourne (-)la valeur de -x c'est-à-dire :

Algorithm 4: Conversion d'un réel en rationnel version 2

Function `convert_Float_to_Ratio`

Input: $x \in R^+$: un nombre réel à convertir en rationnel
`nb_iter` $\in N$: le nombre d'appels récursifs restant

// première condition d'arrêt
if $x == 0$ **then** return $\frac{0}{1}$
// seconde condition d'arrêt
if `nb_iter` $== 0$ **then** return $\frac{0}{1}$
if $x < 0$ **then** return $\Big(- \texttt{convert\_float\_to\_ratio}\Big(- x, \texttt{nb\_iter}\Big)\Big)$
// appel récursif si $x < 1$
if $x < 1$ **then**
    return $\Big(\texttt{convert\_float\_to\_ratio}\Big(\frac{1}{x}, \texttt{nb\_iter}\Big)\Big)^{-1}$
// appel récursif si $x \geq 1$
if $x \geq 1$ **then**
    $q = \lfloor x \rfloor$ // partie entière
    return $(\frac{q}{1} + \texttt{convert\_float\_to\_ratio}(x - q, \texttt{nb\_iter} - 1)) * sign$

Après observation et tests de notre fonction, on a remarqué que les grands et les très petits nombres se représentaient assez mal. En effet, ils se représentent mal car la précision de leur représentation binaire diminue à mesure que le nombre augmente ou diminue. En général, en binaire le nombre à virgule flottante est caractérisé par un certain nombre de bits pour la mantisse (la partie significative du nombre) et une autre partie pour stocker l'exposant. Plus on augmente moins il y a des bits pour la mantisse et donc la précision diminue et de même pour les nombres très petits.
On a donc cherché et trouvé une solution, celle-ci consiste à arrondir notre valeur à trois chiffres après la virgule lorsque x est inférieur à 1.

Ce qui nous donne notre algorithme final :

Algorithm 5: Conversion d'un réel en rationnel version 3

Function `convert_Float_to_Ratio`

Input: $x \in R^+$: un nombre réel à convertir en rationnel
`nb_iter` $\in N$: le nombre d'appels récursifs restant

// première condition d'arrêt
if $x == 0$ **then** return $\frac{0}{1}$
// seconde condition d'arrêt
if `nb_iter` $== 0$ **then** return $\frac{0}{1}$
if $x < 0$ **then** return $\Big(-$ `convert_float_to_ratio`$\Big(-x,$ `nb_iter`$\Big)\Big)$
// appel récursif si $x < 1$
if $x < 1$ **then**
    return $\Big($ `convert_float_to_ratio`$\Big(\frac{round(\frac{1}{x}*1000)}{1000},$ `nb_iter`$\Big)\Big)^{-1}$
// appel récursif si $x \geq 1$
if $x \geq 1$ **then**
    $q = \lfloor x \rfloor$ // partie entière
    return $(\frac{q}{1} +$ `convert_float_to_ratio`$(x - q,$ `nb_iter` $- 1)) * sign$

Lorsque les opérations entre rationnels s'enchaînent, le numérateur et le dénominateur peuvent prendre des valeurs très grandes, voir dépasser la limite de représentation des entiers en C++. Pour régler ce problème on peut envisager de :

— Utiliser des types de données plus grands que les entiers standard de C++, comme les types long long ou int128, qui permettent de représenter des nombres plus grands mais ont leurs propres limites de représentation.

— Utiliser une bibliothèque de calcul à virgule flottante de précision arbitraire, comme GMP (GNU Multiple Precision Arithmetic Library), qui permet de représenter des nombres avec une précision arbitrairement grande en utilisant des types de données spéciaux qui peuvent stocker des nombres très grands.

Cependant il ne faut pas oublier que ces solutions ont un coût en termes de complexité de calcul et de mémoire.

Après l'implémentation de cette fonction, nous avons aussi implémenter convertRatiotoFloat, qui est plus simple et qui nous a permis de formaliser les opérateurs : cos,sin,tan,exp.

3.4 Quand et pourquoi utiliser les nombres rationnels

Avant de commencer ce projet nous avons fait un petit brainstorming pour savoir quels seraient les fonctionnalités intéressantes utilisant les nombres rationnels.

Tout d'abord, un rationnel est utile lorsqu'une précision absolue est requise. En effet, comme vu en cours, les nombres à virgule flottante peuvent être limités et produire des erreurs d'approximation, tandis qu'un nombre rationnel, codé avec deux entiesr, peut avoir une précision parfaite.

Voici les exemples auxquels nous avons pensé mais que nous n'avons pas pu implémenter :

— Le calcul de pourcentage et de proportion :avec des nombres à virgule flottante, 30% de 100 peuvent donner un résultat approché comme 70.0000000002. Avec des nombres rationnels on obtiendrait directement 80. Idem pour des proportions comme 1/3, 2/3, 1/6 qui sont des nombres à virgule infinis, exprimés en rationnels, on obtient des proportions exactes.

— Pour certains calculs comme la moyenne de plusieurs valeurs (ou même plusieurs notes exprimées en rationnels) : avec des nombres flottants, on risque de perdre en précision à chaque opération, tandis qu'avec des nombres rationnels on obtiendra une valeur exacte.

— Pour créer des matrices : certaines matrices peuvent contenir des nombreuses petites valeurs. Grâce aux nombres rationnels, nous pourrions créer des matrices de rationnels contenant des valeurs exactes et ainsi ne pas amplifier les erreurs d'approximations lors des calculs.

En résumé, pour tous les calculs ayant besoin d'une précision absolue (et n'ayant pas de contraintes de performances) les nombres rationnels peuvent être utiles. Cela peut aller des exemples ci-dessus à d'autres utilisations comme par exemple dans des matrices, des polynômes, des calculs de distances entre deux points d'un plan, calculs de surfaces, des calculs avec des coefficients rationnels etc. Même si nous n'avons pas eu le temps d'implémenter ces fonctionnalités plus concrètes avec notre librairie, nous avons construit la base nous permettant de le faire.

3.5 Quand ne pas utiliser les nombres rationnels

Nous avons aussi retenu des points négatifs à l'utilisation de nombres rationnels.

— L'expression de grands nombres ou de nombres irrationnels (comme PI) sous forme de rationnel : comme évoqué précédemment, il est difficile d'exprimer de grands nombres sous forme de rationnel, car on perdrait en précision.

— Les performances : les nombres rationnels sont moins performants que les nombres à virgule flottante. Si on utilise des nombres rationnels dans des algorithmes nécessitant de nombreux calculs, le programme risque de durer plus longtemps.

— La mémoire : les nombres rationnels prennent plus de place en mémoire que les nombres à virgule flottante, car ils doivent stocker deux entiers au lieu d'un nombre à virgule.

— La facilité d'utilisation : les nombres rationnels ne sont souvent pas directement implémentés dans les langages, cela peut amener des difficultés d'utilisation.

4 Conclusion

Au cours de ce projet, nous avons développé une librairie de ratio en mettant en pratique les connaissances acquises lors des cours et tps de cette année. Malgré des difficultés sur certains aspects, nous avons reussi à les résoudre grâce à notre communication au sein du groupe. Globalement, nous avons apprécié ce sujet nous donnant une application directe et concrète des mathématiques en programmation.