Ratio - Projet de programmation objet - mathématiques pour l'informatique

Emma Veauvy, Grégoire Tinnes | IMAC2 | Lien GitHub

Partie programmation

Tableau bilan programmation

Tâche	Fait et fonctionnel
CMake	✅
Class rationnel	✅
Opérations sur rationnels	✅
Opérateurs de comparaison	✅
Valeur absolue	✅
Moins unaire	✅
Partie entière	✅
Fonction d’affichage	✅
Fonction conversion réel-rationnel	✅
Fonction fraction irréductible	✅
Fonction produit virgule flottante et ratio	✅
Fonction produit ratio et virgule flottante	✅
Tests unitaires	✅
Documentation	✅
Bonus : class en template	✅

Partie mathématiques

Questions

Question : Comment formaliseriez vous l'opérateur de division / ?

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

Lorsqu'on divise par une fraction, cela revient à multiplier par l'inverse de cette fraction.
Soit, dans cet exemple diviser avec c/d, c'est multiplier par d/c.
À noter que cette formalisation est très utile car la multiplication est plus rapide que la division lors du calcul de l'ordinateur : on gagne 4,5 nanosecondes.

Question : Autres opérateurs

$$\sqrt{\frac{a}{b}} = \frac{a}{b}^{\frac{1}{2}} = \frac{\sqrt{a}}{\sqrt{b}} = \frac{\sqrt{ab}}{b}$$

$$cos(\frac{a}{b}) : pas\ de\ formalisation\ possible$$

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

$$e^{\frac{a}{b}} = \frac{e^{a}}{b^{a}}$$

Excepté pour le cosinus, il est aisé de décomposer les écritures comme il est possible de le faire avec la racine, la puissance et l'exponentielle lorsqu'elles sont appliquées à une fraction.

Question : Comment le modifier pour qu'il gère également les nombres réels négatifs ?

Nous avons modifié deux des conditions :

- Dans le cas d'un x inférieur à 1, il faut maintenant vérifier dans un même temps qu'il soit supérieur à 0.
  On a alors : `if(x < 1 && x > 0)`
- Pour traiter le cas des x négatifs, nous avons modifié le cas x supérieur à 1 pour qu'il concerne également les x négatifs.
  On a alors : `if(x > 1 || x < 0)`

Question : D'une façon générale, on peut s'apercevoir que les grands nombres se représentent assez mal avec notre classe de rationnels. Voyez-vous une explication à ça ?

En effet, plus le nombre est grand, plus il est difficile de le représenter sous forme de fraction et un plus grand nombre d'itérations est nécessaire à notre fonction convert_float_to_ratio() pour être précise.
Également, les grands nombres ne sont pas très lisibles sous forme de fraction, on préfère généralement utiliser un nombre à virgule pour les représenter.
Aussi, l'utilisation de rationnels peut entraîner l'arrondissement de valeurs et donc provoque des erreurs de plus en plus grandes si la valeur est grande également.
Enfin, les dépassements peuvent être à l'origine de ce manque de précision.

Question : 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++. Voyez-vous des solutions ?

Il faudrait rendre irréductibles les fractions entre chaque opération de sorte que les valeurs du numérateur et du dénominateur ne prennent pas de trop grande valeur jusqu'à dépasser la limite de représentation des entiers.
Il est également possible d'utiliser en priorité des long int au lieu de simple int afin d'avoir une limite plus haute. Cette utilisation des types int ou long int est rendue facilement possible grâce à l'utilisation d'une classe en template.
En outre, afin d'éviter les dépassements, il faut normer et conditionner les données.

Réalisation du projet

Solutions face aux problèmes

- Au cours du projet, nous nous sommes rendus compte que l'on était amené à plusieurs reprises dans différentes méthodes (opérateurs +, -, >, <, =>, =<) à mettre deux rationnels au même dénominateur. Nous avons donc créé et implémenté après coup une fonction *commonDenom* qui met deux rationnels au même dénominateur. Le fait de créer une fonction à cet effet permet d'éviter la répétition et donc d'avoir un code plus court et clair.
  Pour nous la façon la plus simple a été de multiplier chacun des numérateurs par le dénominateur de l'autre fraction.
  Cela nous donne les 2 rationnels au même dénominateur, même si elles sont encore réductibles.

```
Function commonDenom

     input :  ratio1 : un rationnel

            ratio2 : un autre rationnel

//vérifier si les 2 ratios ont le même dénominateur

if ratio1.denom != ratio2.denom

     ratio2.num = ratio2.num * ratio1.denom

     ratio1.num = ratio1.num * ratio2.denom

     ratio2.denom = ratio2.denom * ratio1.denom

     ratio1.denom = ratio2.denom
```

- Dans la fonction *convert_float_to_ratio()*, dans l'application du pseudo-code donné dans l'énoncé, nous cherchions d'abord à appliquer une puissance -1 à un rationnel. Nous avons alors essayé d'utiliser la fonction pow sans succès. Finalement, nous avons fait une fonction *invert()* qui inverse la fraction, ce qui est équivalent à la puissance -1 et est plus simple.

- Pour la fonction *irreducible()*, nous avons fait le choix d'agir directement sur les attributs de l'instance plus que de retourner un Ratio car nous avons considéré qu'il n'y avait pas de cas où la forme non réduite était utile. Aussi, l'opération est non-destructive. En outre, cela permet de ne pas faire une copie supplémentaire du Ratio et donc d'économiser en nombre d'opérations.

```
Function irreducible

     input :  ratio : une fonction rationnel

//regarder si le ratio est irréductible, donc si c'est le cas

son pgcd est de 1

if pgcd(ratio) >1 then

     denom.ratio/=gcd;

     num.ratio/=gcd;
```

- Dans un certain nombres d'opérateurs, nous utilisons les fonctions *commonDenom()* ou *irreducible()* rendant impossible la mise en const de la fonction. Nous pourrions les modifier en créant un Ratio supplémentaire au sein de la fonction, lui affecter les valeurs du Ratio voulu et ainsi pouvoir utiliser les fonctions sur celui-ci. Cependant, nous sommes partis du principe que cela serait plus coûteux en opération.

- Pour la surcharge de l'opérateur * dans les cas d'une multiplication d'un ratio par un float et inversement, la difficulté était de bien séparer les deux fonctions afin de leur appliquer un comportement différent. Après différents essais qui n'aboutissaient pas, nous avons finalement trouvé une solution. Cela consiste à définir l'opération ratio * float en tant que membre de la classe Ratio, et l'opération ratio * float en dehors. Nous avons aussi choisi de retourner un résultat en float pour l'opération float * ratio par souci de logique. Enfin, après quelques recherches, nous avons réalisé qu'il était approprié d'utiliser la fonction faite auparavant *convert_float_to_ratio()* pour l'opération ratio * float pour faciliter le calcul et éviter une redondance, c'est pourquoi cet opérateur est défini après la fonction *convert_float_to_ratio()*.

- Une difficulté importante de ce projet a été de mettre en place le cmake, notamment par rapport à son organisation (cmakelists global, cmakelists par sous-répertoire, Doxygen) mais aussi dans le cadre du travail en groupe. En effet, le projet ne compilait d'abord que sur l'une de nos deux machines. Un des soucis était de nettoyer les fichiers de cache et autres fichiers créés automatiquement à l'exécution ainsi que de ne pas push sur le github le dossier build pour permettre une compilation sur nos deux machines.

Structure de données

Notre classe Ratio est composée de deux attributs de type T. C'est donc une classe en template.
Ces attributs sont :

- **T num**, qui correspond au numérateur
- **T denom**, qui correspond au dénominateur

Ces attributs sont privés (private). Il est possible de les lire ou de les modifier depuis les méthodes publiques suivantes : *getNum()*, *getDenom()*, *setNum()* et *setDenom()*.