RATIO

Projet de Programmation Objet - Mathématiques pour l'informatique

Maxime Mathevet - Eric Thiberge

TABLEAU BILAN

ÉLÉMENTS DEMANDÉS	Pas fait	Fait mais ne fonctionne pas ou fait partiellement	Fait
readme.md			
Documenter (doxygen ?)			
Le dénominateur doit être positif			
$\frac{a}{b}$ doit être une fonction irréductible			
0 = 0/1			
Opérateurs à implémenter : +, -, ×, /, -1aire, ==, !=, >, <, <=, >= et << (affichage)			
∞ = 1/0			
Implémenter la méthode inverse			
opérations à implémenter : $\sqrt{\frac{a}{b}}$, $cos\left(\frac{a}{b}\right)$, $\left(\frac{a}{b}\right)^k$, $e^{\frac{a}{b}}$, $\lfloor x \rfloor$, ...		?	
Conversion d'un réel en rationnel		?	
Élaborer des tests pour savoir quand les nombres rationnels sont plus efficaces et quand ils ne le sont pas			
Batterie d'exemple pour montrer que la bibliothèque fonctionne.		?	
Coder la classe en template.			
Coder la classe en constexpr.			
Compilation du projet gérée avec cmake			

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

$$\frac{\frac{a}{b}}{\frac{c}{d}} = \frac{a \times d}{b \times c} \text{ avec } (b,\ c)\ \neq\ (0,\ 0)$$

Question : Méthodes cos, exp, puissance etc…

Pour l'opérateur puissance (pow( )), nous avons fait une fonction récursive avec la méthode :

```
Ratio.pow(k): // puissance k
     instancier ratio à 1/1 // (classe ratio)
     rendre irréductible le Ratio appelant (this)
     si k = 0:
          retourner ratio
     sinon:
          retourner this*pow(k-1)
```

Cela permet de ne pas utiliser une boucle `for` et d'obtenir une complexité moins élevée pour cette méthode : $o(x)$ contre $o(x^2)$.

Pour les méthodes `sqrt(), cos(), sin(), tan(), exp()` et `log()` nous avons testé deux méthodes :

- utiliser un développement limité, mais l'inconvénient est que cette méthode marche pour les nombres très proches de 0
- utiliser la fonction `convert_float_to_ratio().`

Dans les deux cas, ces opérateurs ne sont pas fonctionnels dans le cas de grands nombres.

Pour les méthodes sqrt( ) et log( ) , étant donné qu'elles sont définies sur $[\,0\,,\ +\infty\,]$ et $]\,0\,,\ +\infty\,]$, nous avons fait des exceptions pour éviter de calculer ces fonctions en dehors de leur domaine de définition.

Pour log( ) , nous avons utilisé la formule $\ln(\frac{a}{b})$ = ln(a) - ln(b)

Algorithm 1: Conversion d'un réel en rationnel

```
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      // première condition d'arrêt
3      if x == 0 then return 0/1
4      // seconde condition d'arrêt
5      if nb_iter == 0 then return 0/1
6      // appel récursif si x < 1
7      if x < 1 then
8          return ( convert_float_to_ratio(1/x, nb_iter) )^-1
9      // appel récursif si x > 1
10     if x > 1 then
11         q = ⌊x⌋ // partie entière
12         return q/1 + convert_float_to_ratio(x − q, nb_iter − 1)
```

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

Pour qu'il gère les nombres négatifs nous avons ajouté une condition au début de la fonction vérifiant si le nombre est négatif, et traite sa valeur absolue le cas échéant.

```
si x < 0:
      x = valeur absolue de x
      nouveau Ratio ratio = convert_float_to_ratio(x, nbIter)
      retourner -ratio
```

Elle finit par retourner l'opposé du nombre rationnel calculé.

Question pseudo code : D'après vous, à qui s'adresse la puissance – 1 de la ligne 8 ou bien la somme de la ligne 12 ?

A la ligne 8, la puissance -1 s'adresse au nombre rationnel fraîchement converti dont on effectue l'inverse.

A la ligne 12, la somme est l'opération + entre le nombre rationnel q/1 et le nombre rationnel venant d'être converti et anciennement float.

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

En effet, les grands nombres ainsi que les nombres avec une grande précision après la virgule sont les plus difficiles à représenter avec des rationnels en C++. La raison de cela est que l'écriture de tels nombres implique d'avoir de très grands nombres au numérateur ou au dénominateur.

Un nombre à virgule flottante tel que 14852.958645 par exemple devrait être noté : 2970591729/200000**,** or ici le numérateur dépasse la valeur maximale atteignable avec des `int.` (2 147 483 647 = $2^{31} - 1$ = 0111 1111 1111 1111 1111 1111 1111 1111_2) une fois cette valeur dépassée, le bit de poids le plus fort s'incrémente et passe à 1. Cependant ce bit sert à coder le signe du nombre et **1**011 0001 0000 1111 1010 0001 1111 0001 ne vaut pas 2 970 591 729 mais un nombre négatif : **-**1 324 375 567.

Cela finit donc par complètement fausser les calculs lors de la fonction `convert_float_to_ratio()` au bout de quelques itérations seulement et des opérations impliquant des rationnels avec des composantes importantes peuvent donner des résultats aberrants.

Question : Lorsque les opérations entre rationnels s'enchaînent, le numérateur et le dénominateur peuvent prendre des valeurs très grandes, voire dépasser la limite de représentation des entiers en C++. Voyez-vous des solutions ?

Il est premièrement possible d'utiliser des types tels que `long int` ou `long long int` pour permettre de repousser considérablement la limite de représentation des entiers en C++. Un `long long int` pourrait aller jusqu'à
9 223 372 036 854 775 807.

Une autre solution que nous imaginons serait de décomposer les nombres rationnels importants en sommes ou en produits de nombres rationnels plus petits ne dépassant pas une certaine valeur.

∞ = 1/0 :

Dans la fonction définissant l'opérateur << (chargé de l'affichage), nous avons traité les cas où le numérateur est égal à 0. Le compilateur affiche ∞ quand le dénominateur est positif et − ∞ lorsqu'il est négatif.

Implémenter la méthode inverse :

Grâce à notre traitement de ∞ = 1/0, nous n'avons pas eu à faire d'exception dans la méthode inverse.

Élaborer des tests pour savoir quand les nombres rationnels sont plus efficaces et quand ils ne le sont pas :

Le test que nous avons élaboré reprend la suite définie dans l'introduction du sujet. En utilisant notre bibliothèque, la suite fonctionne bien et le compilateur affiche une suite bien constante.

Batterie d'exemple pour montrer que la bibliothèque fonctionne :

Dans le fichier code/main.cpp est présente toute une série de tests unitaires permettant de tester le bon fonctionnement de ce qui a été implémenté.

Malheureusement, à cause de la manière dont nous avons codé les méthodes sqrt(), cos(), sin(), tan(), exp() et log() les six tests unitaires qui y sont associés ne réussissent pas.

Coder la classe en template :

La classe à été codée en template, il est donc possible d'instancier des nouveaux nombres rationnels avec `int`, des `long int` et des `long long int`.

Coder la classe en constexpr :

Nous avons défini toutes les fonctions avec un seul `return` en `constexpr` afin que celles-ci soient calculées à la compilation du programme.

Compilation du projet gérée avec cmake :

La compilation du projet est gérée avec cmake. La procédure à suivre pour exécuter le programme est inscrite tout en bas du fichier README.md trouvable sur le dépôt git du projet.

Problèmes majeurs rencontrés et réflexion sur le sujet :

Le plus gros problème auquel nous avons dû faire face est celui lié à

l'implémentation de `convert_float_to_ratio()`. Beaucoup de nos méthodes et de nos surcharges d'opérateurs reposaient ou reposent toujours sur le bon fonctionnement de cette fonction. Seulement, on sait que le résultat qu'elle renvoie peut être totalement erroné.

De plus, nous avons eu beaucoup de mal à coder la classe en template et à gérer les différents types. En particulier au moment de fractionner le code : beaucoup de fonctions donnaient des "références indéfinies" vers `convert_float_toratio()` (qui était définie dans un fichier différent dès le départ) mais seulement pour certains types. Exemple : `référence indéfinie vers Ratio<long> convert_float_to_ratio<long>(...)` mais pas pour les `int`... Il nous aura fallu du temps pour comprendre ce qui n'allait pas. C'est d'ailleurs ce problème là qui fait que la surcharge de l'opérateur `/` avec un réel n'est pas définie dans `ratioOperators.cpp` avec les autres.

En complément du sujet, il aurait été intéressant de laisser la possibilité d'instancier des nombres rationnels contenant des doubles. Cela aurait permis de représenter des angles en radians sous la forme $\frac{\pi}{2}$ par exemple.