RAPPORT FINAL PROJET MATH-INFO & PROG

I. INTRODUCTION

Dans ce rapport, Aurore Lafaurie et Sarah N'GOTTA vont vous présenter leur travail de recherche et de programmation sur les nombres rationnels. Dans une première partie, vous trouverez une présentation de la classe de rationnels, les surcharges d'opérateurs associées et des recherches approfondies sur le sujet... Dans une deuxième partie basée sur l'analyse, elles traiteront des questions posées dans le sujet. Enfin, une partie succincte concernant la partie programmation, elles présenteront un tableau des tâches réalisées.

II. Nombre Rationnel

A. Classe

Nous avons codé une classe Ratio à partir des consignes du sujet. Nous avons modifié quelques points.

Les attributs de notre classe sont donc :

- le numérateur comme integer (int),
- le dénominateur comme integer non signé (unsigned int)

Nous avons aussi envisagé d'ajouter un booléen permettant de savoir si le rationnel créé était approximé ou exact. Afin de coller au sujet, tous les rationnels générés ont un PGCD égal à 1 avec des méthodes de vérification.

B. Opérateurs

1. Somme

$\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{cd}$

WARNING : ne pas avoir de dénominateur égal à 0

2. Soustraction

$\frac{a}{b} + (-\frac{c}{d}) = \frac{ad+(-c)b}{cd}$

WARNING : ne pas avoir de dénominateur égal à 0, on utilise l'opérateur rationnel + et le moins unaire -

3. Moins Unaire

$-(\frac{a}{b}) = \frac{-a}{b}$

WARNING : ne pas avoir de dénominateur égal à 0

4. Produit

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

WARNING : ne pas avoir de dénominateur égal à 0

5. Inverse

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

WARNING : ne pas avoir de dénominateur égal à 0

6. Division

$\frac{\frac{a}{b}}{\frac{c}{d}} = \frac{a}{b} * \frac{d}{c} = \frac{a}{b} * inverse(\frac{c}{d}) = \frac{ad}{bc}$

WARNING : ne pas avoir de dénominateur égal à 0, on va utiliser l'opérateur d'inverse vu précédemment. Dans les tests unitaires, on testera deux méthodes, utiliser l'opérateur inverse surchargé pour les ratios et le produit classique en faisant l'inversion nous-même. Une exception est donc renvoyée si le ratio à diviser est divisé par le rationnel 0. L'opérateur puissance acceptant les puissances négatives, on peut également calculer un inverse, une division avec la fonction puissance (^-1).

7. Valeur Absolue

$\frac{a}{b} = \frac{a}{b}$ ou $\frac{-a}{b} = \frac{a}{b}$

WARNING : ne pas avoir de dénominateur égal à 0, on utilise l'opérateur abs de base prévu dans la librairie cmath.

8. Partie Entière

$\frac{a}{b} = int(\frac{a}{b})$

WARNING : ne pas avoir de dénominateur égal à 0, bien penser à faire un static_cast< int > du dénominateur afin de pouvoir faire l'opération.

9. Puissance

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

WARNING : ne pas avoir de dénominateur égal à 0, cet opérateur est surchargé pour des int comme pour des rationnels en exposant. Pour cela, il a fallu décomposer le rationnel en partie entière et décimale puis appliquer de manière successive.

10. Racine Carrée

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

WARNING : ne pas avoir de dénominateur égal à 0, on utilise l'opérateur sqrt de base prévu dans la librairie cmath.

Notre piste d'analyse au sujet de l'opérateur racine carrée pour des nombres rationnels est que cet opérateur ne pourra pas fonctionner pour :

- des nombres entiers qui mis sous racine carrée ne donne pas en résultat un nombre entier (ex : ne fonctionne pas pour sqrt(2) mais fonctionne pour sqrt(9)=3). Lorsque ça ne fonctionne pas, on ne pourra jamais obtenir de fraction entière, seulement des approximations.

- une autre approche, serait de dire que l'opérateur ne fonctionnera pas sur les nombres premiers et nombres entiers décomposables en nombres premiers (ex : ne fonctionnera pas pour 2, 3 et 6. En effet, 2 et 3 sont des nombres premiers et 6=2*3, elles se décomposent en produit de nombres premiers, passé sous racine, le résultat ne sera pas un nombre entier).

Pour le moment la fonction d'approximation n'est pas implémentée. On aura donc un message d'erreur si le calcul comprend les exceptions précédentes.

Une autre façon de calculer la racine carrée d'un rationnel serait d'utiliser l'opérateur rationnel de puissance. Ainsi : $\sqrt{\frac{a}{b}} = (\frac{a}{b})^{\frac{1}{2}}$

Nous avons tout de même implémenté l'opérateur ainsi que réalisé les tests unitaires mais ces derniers ne fonctionnent que pour des racines de nombres entiers non premiers et donc la racine est un entier.

11. Fonction d'affichage (<<)

```
std::ostream& operator<<(std::ostream& os, const Ratio &r) ;
```

On va surcharger un opérateur d'affichage de la librairie std ostream. Le paramètre os renvoie à ce qui va venir avant les <<, la surcharge << fait le lien entre le mot d'appel "cout" et notre choix d'affichage de rationnel.Surcharger << nous permet de choisir la façon dont on va afficher nos rationnels à savoir : a/b ainsi que l'équivalent en nombre réel. Cette surcharge d'opérateur ne sera pas implémentée à l'intérieur de la classe car elle n'est pas une méthode propre à la classe, ce n'est pas un opérateur de type Ratio.

12. Cos, Sin & Tan

Les seules valeurs rationnelles exactes que puissent prendre le cosinus ou le sinus d'un angle rationnel sont 0 , 1 , -1, $\frac{1}{2}$, $-\frac{1}{2}$. Pour les autres cas, il s'agit d'approximations.

	0	$\frac{\Pi}{6}$	$\frac{\Pi}{4}$	$\frac{\Pi}{3}$	$\frac{\Pi}{2}$	Π
sin	0	$\frac{1}{2}$	-	-	1	0
cos	1	-	-	$\frac{1}{2}$	0	-1

13. Exponentielle et Logarithme Népérien

Les fonction exponentielle et logarithme sont définies dans R et R*+, elles prennent donc en paramètre n'importe quel paramètre réel (positif non-nul pour la fonction ln), y compris des rationnels puisque R=QuQ'.

Pour la fonction exponentielle, elle est définie dans tout le domaine des réels. Il n'y a donc pas d'exceptions à préciser pour les paramètres qui lui sont passés.

Pour la fonction logarithme népérien, il faut faire attention, les paramètres passés à la fonction ne doivent pas être négatifs ou nuls. La fonction n'est pas définie pour ces valeurs-ci. Il faudrait donc un message d'erreur. Pour une valeur infinie, la fonction ln vaut l'infini. En 1, la fonction ln vaut 0.

Les résultats de exp et de ln ne sont pas tous rationnels. Il faudrait donc approximer les résultats. Les résultats rationnels entiers dont nous sommes certains sont 1 pour exponentielle et 0 pour logarithme népérien. De plus, après quelques recherches nous avons pu trouver qu'une exponentielle d'un nombre rationnel est un nombre irrationnel. Nous avons donc commencé à coder une version simplifiée mais fonctionnelle.

Une méthode très simple est de passer par des réels pour ensuite calculer et reconvertir.

La réciproque de la fonction exp est la fonction ln et inversement. Peut-être serait-il possible de coder la fonction exp et de s'en servir pour coder la fonction ln en précisant les exceptions qui les différencient.

*En plus des opérateurs explicités ci-dessus, nous avons également surchargé des opérateurs d'assignation comme +=, -=, *= et des opérateurs de type comparaison (/=, <, >, <=, >=) qui ont été templaté pour être utilisés avec autre chose que des rationnels.*

C. Approximation et Conversion

1. Nombre approximé ou non

Une bonne façon d'améliorer notre code serait de créer des fonctions d'approximation. L'une pourrait nous dire si le résultat d'un calcul est approximatif ou non et une autre qui pourrait approximer un résultat plutôt que de renvoyer une erreur. Par exemple, si l'on prend le cas de l'opérateur racine carré. Une telle fonction pourrait renvoyer un résultat approximatif de $\sqrt{2}$ en convertissant le résultat en float au lieu de renvoyer une erreur.

2. Conversion réél-ratio

a) Pour les nombres réels positifs ou nuls

En prenant l'algorithme donné dans le sujet, ce cas était déjà traité. Afin de l'étendre, nous avons décidé d'utiliser des templates afin de pouvoir reconnaitre si la valeur passait était un nombre entier ou décimal (cela permettait ainsi de gagner du temps à la compilation). Par exemple, pour un nombre entier pas besoin de faire tout l'algorithme, il suffisait juste de créer un ratio avec la valeur comme numérateur.

b) Pour les nombres réels négatifs

Pour ce qui est de la partie négative des nombres, l'algorithme n'a pas été très compliqué à changer. En effet, il a juste fallu rajouter une condition mentionnant que si la valeur était négative il fallait alors calculer la conversion avec la valeur absolue de la valeur et appliquer le moins unitaire au dernier moment.
Pseudo code :

```
Si valeur < 0:
Alors - (conversionReelVersRatio(abs(valeur)));
```

c) Nombre d'itérations déterminé

Pour ce qui est du nombre d'itérations sur l'algorithme, nous avons décidé de partir sur 6, car après plusieurs tests, ce chiffre a semblé concluant pour calculer les 9 premières décimales de PI et ainsi obtenir le PI de notre classe Ratio. Cela nous ait donc semblé être une bonne idée de rester sur le même chiffre pour garder le même ordre de précision.

D. Pour aller plus loin... Templetage et constexpr

Nous n'avons pas utilisé le constexpr car nous nous sommes retrouvées face à des problèmes à la compilation. Pour ce qui est de l'utilisation du template par contre, nous l'avons employé dans de nombreuses méthodes et notamment celle de conversion d'un réel vers un rationnel afin de pouvoir convertir un entier ou un nombre décimal.

III. Analyse

A. Les questions du sujet

1. Dans quels cas les nombres rationnels sont-ils efficaces ? Dans quels cas ne le sont-ils pas ?

Les nombres rationnels sont efficaces si les nombres ne sont pas trop grands ou trop petits. En effet, les int et les unsigned int ne sont pas codés à l'infini et sont donc limités. Ils sont aussi efficaces quand on doit représenter certains réels avec un nombre de décimales infini (par exemple $\frac{1}{3}$).

2. D'une façon générale, on peut s'apercevoir que les grands nombres (et le très petit nombre) se représentent assez mal avec notre classe de rationnels. Voyez-vous une explication à ça ?

Comme dit plus tôt, les très grands nombres ainsi que les très petits seront compliqués à gérer en l'état puisque les entiers sont codés sur un nombre de bits limité. On se retrouverait donc dans certains cas avec des numérateurs et des dénominateurs tellement grands qu'il serait impossible de les coder en integer et unsigned integer.

3. 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 ?

Pour éviter d'être limité dans la représentation d'entiers, on pourrait choisir d'utiliser des long int par exemple. Cela permettrait d'avoir plus de possibilités mais cela serait aussi largement plus demandeur en terme de mémoire. Il faudrait donc que le type de variable soit adaptatif à la situation.

Une autre solution aurait été de garder la partie entière d'un côté et de ne travailler qu'avec la partie décimale. Cela résoudrait une part du problème (pour les très grands nombres). Il faudrait pour cela avoir un attribut conservant la partie entière. Cela n'a pas été implémenté mais cela aurait pu être une solution.

Pour ce qui est des très petits nombres, il aurait aussi été envisageable de faire une opération afin de multiplier par une puissance de 10 avant la conversion et diviser par la même puissance de 10 pour l'utilisateur. Ainsi, l'implémentation est cachée à l'utilisateur.

4. 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 ainsi que la somme de la ligne 12 s'applique à des objets de notre classe de rationnels. Cela nous permet ainsi d'avoir le bon type de retour.

IV. Programmation

Cliquer ici pour aller sur la page Git du projet !

Demandé	**Pas demandé**	**Pas demandé**
codé et fonctionnel	codé et fonctionnel	pas codé ou pas fonctionnel
[Ratio] * Numérateur (int) (represented as a) Dénominateur (uint) (represented as b) * ratios irreductibles * 0=0/1 * infini=1/0 * PI-> constexpr 103993/33102	* puissance * racine carrée * cosinus/sinus * variadics (dans pow) * exponentielle * ln * coder en template	* class en constexpr
[Opérateurs] * Somme * Produit * Inverse * Division * Moins unaire * Valeur Absolue * Partie entière * Produit reel-ratio/ratio-reel * Opérateurs de comparaison (==, !=, >; <; <=, >=) * Opérateur d'affichage << * Conversion reel to ratio (aussi pour les nombres négatifs)		
* Tests unitaires fonctionnels * exemples * documentation * cmake * readme * classes * usage stl * exceptions * asserts * espaces de nommage		

Comme le montre le tableau, tout ce qui était demandé dans le sujet a été réalisé et est codé et fonctionnel. Pour que notre librairie soit facilement utilisable, nous avons réalisé un fichier d'exemple sur l'utilisation de cette dernière ainsi qu'une documentation Doxygen.