RAPPORT PROJET RATIONNEL

COURTE INTRODUCTION

Dans ce projet, on se propose de représenter les nombres réels sous formes de nombres rationnels plutôt qu'avec des nombres à virgule flottante. Il convient donc d'implémenter notre propre bibliothèque pour ces nombres rationnels, contenant nos différents éléments permettant de les manipuler et de les utiliser.

PARTIE PROGRAMMATION

Nous proposons de commencer en dressant un petit tableau bilan pour la partie programmation du projet. Il liste les fonctionnalités sur lesquelles nous avons travaillé tout au long du projet. Nous retrouvons ainsi l'état de ces fonctionnalités, à savoir si elles sont demandées, codées, fonctionnelles, etc…

TABLEAU BILAN

OUI	NON

Fonctionnalité	Demandée	Codée	Fonctionnelle
Somme de deux rationnels, entre rationnel et flottant et inversement			
Différence entre deux rationnels, entre rationnel et flottant et inversement			
Produit de deux rationnels			
Inverse d'un rationnel			
Moins unaire			
Valeur absolue d'un rationnel			
Partie entière d'un rationnel			
Produit entre réel et rationnel			
Produit entre rationnel et réel			
Division entre deux rationnels			
Division entre réel et rationnel			
Division entre rationnel et réel			

Fonctionnalité	Demandée	Codée	Fonctionnelle
Modulo entre deux rationnels			fonctionne presque mieux qu'avec des réels
Racine carrée d'un rationnel			
Exponentielle d'un rationnel			
Fonctions trigonométriques (cos, sin, tan et leur réciproque) sur un rationnel			
PGCD entre deux rationnels			
Logarithme naturel d'un rationnel			
Puissance d'un rationnel			
Conversion d'un réel en rationnel			
Conversion d'un rationnel en réel			
Affichage (surcharge opérateur <<)			
Opérateurs de comparaison			
Expression et gestion du "Zéro" et de l' "Infini"			
Classes (Rationnel)	Conseillée		
Fonctions template	Conseillée		
Usage de la STL	Conseillée		
Exceptions	Conseillée		
Asserts	Conseillée		
Tests unitaires			PGCD
Getters			
Constructeurs			
Destructeur			
Simplification d'un rationnel			
Signe d'un rationnel			

Nous avons donc pris certaines libertés en ajoutant des éléments non demandés que nous jugions intéressants et pertinents de traiter.

PROBLÈMES RENCONTRÉS - SOLUTIONS - REMARQUES

Ici, nous expliquons les problèmes rencontrés avec leurs solutions (ou non, et également émettons des remarques autour du projet et du code.

L'opération de produit entre rationnel et flottant (impact sur division aussi) :

Lors de la mise en place de l'opérateur **"*"** entre ces deux types, nous obtenions une troncature sur le rationnel résultant, et ducoup nous perdions en précision.

En fait, c'est parce que le numérateur et le dénominateur étaient définis comme des ***long*** dans notre classe. L'opération entre le numérateur et le flottant nécessaire causait cette troncature. Il fallait donc bien préciser ici de mettre le numérateur et le dénominateur en flottant, dans le cas de cet opérateur. De plus, il fallait également prendre en compte le signe pour éviter les erreurs, ce pourquoi nous avons ajouté un attribut ***mSign*** dans la classe Ratio.
Ensuite, quand nous sommes passés en ***template***, le problème n'avait plus de raison d'être et l'opération marchait.

La fonction trigonométrique tangente :

Cette fonction n'est pas vraiment un problème mais nous tenions à soulever l'expression de cette dernière. Dans la plupart des cas, nous utilisons les fonctions de la STL dans notre projet : directement faites, précises, faciles à obtenir et à manipuler.

Nous voulions alors expérimenter la relation ***tan = sin/cos*** pour définir notre fonction tangente (sin et cos provenant de la STL). Au final, en comparant cette relation et ***tan()*** de la STL, nous perdions en précision. Nous nous sommes donc cantonnés à utiliser cette fonction *tan()*, dommage car nous voulions essayer autre chose que la STL cette fois-là.

L'opérateur modulo :

Suite à l'implémentation de cet opérateur, nous l'avons considéré comme "**trop efficace**" ! Prenons l'exemple suivant avec différents types :
(int % réel) : $2 \,\%\, 0.6667 = 0.6667$
mais
(version rationnelle) : $\frac{2}{1} \,\%\, \frac{2}{3} = 0$
C'est ici l'un des cas où l'utilisation de la forme rationnelle est meilleure.
Par ailleurs, nous obtenons une erreur à cause de cette particularité lors du test unitaire de cette opération, que nous ne savons pas trop comment gérer.

Utilisation de *static* :

Nous utilisons des fonctions membres déclarées ***static*** dans le projet. Ceci pour imiter le comportement habituel des fonctions qu'elles remplacent. Exemple : std::sqrt(2), Ratio::sqrt(r). Tout simplement.

Test unitaire du PGCD entre deux nombres rationnels :

Le test unitaire pour cette fonction ne marche pas à défaut de trouver une fonction ***gcd()*** qui marche bien entre deux flottants pour nos tests. Le calcul du PGCD fait intervenir le reste de la division (**%**), nous avons vu le cas particulier de cet opérateur précédemment : il pourrait alors y avoir un lien quant à l'échec du test unitaire. Quoi qu'il en soit, le test ne marche pas au moment où nous parlons.

PARTIE MATHÉMATIQUES

Dans cette partie, nous nous intéressons aux différents concepts que nous comptions programmer. C'est une étape de formalisation. Nous allons retrouver nos différentes idées/pensées, abouties ou non. En gros, ce sont toutes nos recherches, qui nous ont permis (ou presque, voire pas du tout) de réaliser les concepts. Le sujet comporte quelques questions auxquelles nous allons répondre, voire étendre pour la réflexion.

LES OPÉRATEURS

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

Ici, nous pouvons utiliser cette phrase que l'on connaît tous (du moins) :

"Diviser, c'est multiplier par l'inverse !"

Et ainsi, formaliser de cette manière :

- division de deux rationnels : $\frac{a}{b} / \frac{c}{d} = \frac{ad}{bc}$

On revient donc à faire une multiplication entre un rationnel et l'inverse de l'autre rationnel (que l'on pourrait obtenir facilement en créant une opération effectuant l'inverse d'un rationnel !).

Et si nous voulons utiliser l'opérateur modulo % dans le projet ?

- modulo de deux rationnels : $\frac{a}{b} \% \frac{c}{d} = \frac{ad \% bc}{bd}$

Question : *Vous pourrez également explorer les opérations suivantes :*

$$\sqrt{\frac{a}{b}}, \quad cos\left(\frac{a}{b}\right), \quad \left(\frac{a}{b}\right)^k, \quad e^{\frac{a}{b}}, \quad \text{etc.}$$

Voici quelques formalisations auxquelles nous avons pensé pour différentes opérations :

- racine carrée : $\sqrt{\frac{a}{b}} = \frac{\sqrt{a}}{\sqrt{b}}$
- puissance d'un rationnel : $\left(\frac{a}{b}\right)^k = \frac{a^k}{b^k}$
- exponentielle d'un rationnel : $e^{\frac{a}{b}} = \sqrt[b]{e^a} = \left(e^a\right)^{\frac{1}{b}}$

Fonctions trigonométriques (en partant du cosinus) :

- cosinus d'un rationnel : $cos\left(\frac{a}{b}\right) \simeq 1 - \frac{\left(\frac{a}{b}\right)^2}{2}$ (avec $\frac{a}{b}$ très petit)
- sinus d'un rationnel : $sin\left(\frac{a}{b}\right) = cos\left(\frac{\pi}{2} - \frac{a}{b}\right)$
- tangente d'un rationnel : $tan\left(\frac{a}{b}\right) = \frac{sin\left(\frac{a}{b}\right)}{cos\left(\frac{a}{b}\right)}$

Remarque : pour ces fonctions trigonométriques (mais également pour la fonction exponentielle par exemple), nous pourrions utiliser les **développements limités** pour les approximer (il faudrait alors s'élever à un ordre assez grand pour davantage de précision !). Ou encore les **formules d'Euler** mais il faudrait alors intégrer les nombres complexes au projet (une *library* ?).

- valeur absolue d'un rationnel : $abs\left(\frac{a}{b}\right) = \frac{abs(a)}{b}$
- partie entière (sous forme rationnelle) : $ent\left(\frac{a}{b}\right) = \frac{ent\left(\frac{a}{b}\right)}{1}$
- PGCD entre deux rationnels : $gcd\left(\frac{a}{b}, \frac{c}{d}\right) = \frac{gcd(ad,bc)}{bd}$
- Logarithme naturel d'un rationnel : $log\left(\frac{a}{b}\right) = log(a) - log(b)$

Nous remarquons que ces fonctions particulières sont déjà définies dans la STL (*<numeric>, <cmath>*), alors pourquoi ne pas directement les utiliser sachant qu'elles sont plus précises par rapport à nos approximations (surtout si nous utilisons des développements limités, approximations, etc...). Mais un problème viendrait à nous, car ces fonctions renverraient un résultat à virgule flottante. Il nous faut alors convertir ce résultat en nombre rationnel → algorithme !? C'est ce que nous allons traiter par la suite.

CONVERSION D'UN RÉEL EN RATIONNEL

Voici le pseudo-code d'un algorithme permettant la conversion d'un réel en rationnel.

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 réels négatifs ?*

Pour permettre à ce pseudo code de gérer les nombres négatifs, on va d'abord faire en sorte qu'il convertisse le réel sans se préoccuper de son signe.
Pour ce faire, les conditions et les appels récursifs se feront sur l'absolu du réel mais la valeur renvoyée se fera avec son signe. Ainsi, dès le premier appel récursif, on ne s'occupe plus que d'un réel positif qui n'aura aucun impact sur le résultat donné.

Exemple :

```
if ((roundf(fabs(number) * 1000) / 1000) < 1) {
        return Ratio((convertFloatToRatio(1/fabs(number),
nb_iter).invert()),sign(number));
    }
```

La fonction sign() renvoie 1 si le chiffre est supérieur ou égal à 0 et -1 sinon.

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

En calculant à la main, on remarque qu'à partir d'un certain nombre d'itérations, la précision entre le calcul permis à l'ordinateur et la valeur exacte va diverger de plus en plus, dû à la précision permise selon le type utilisé.
De ce fait, cette différence de calcul va se cumuler et se répercuter sur le rationnel représenté, provoquant une erreur de plus en plus grande à partir de ce seuil d'itérations.
À l'inverse, si les itérations sont peu nombreuses, notre nombre ne sera pas assez précis, voire incomplet.

On va donc rechercher un équilibre entre le nombre recherché et le nombre créé. Nous avons donc déterminé que selon la tranche du nombre recherché, le rationnel serait plus ou moins précis. Privilégiant des petites valeurs aux très grandes quant il s'agit de les convertir. Nous avons également ajouté un arrondi pour les tests d'égalité à 1 pour éviter d'obtenir des valeurs aberrantes d'une itération à une autre. N'oublions pas que selon le type choisi l'impact de l'erreur sera + ou - importante.

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 ?*

- (Ne pas utiliser des rationnels…)

- Séparer ce nombre en deux représentations distinctes liées sous forme de produit par exemple. Il faudrait malheureusement pour cela adapter les opérations pour ne pas briser la règle de priorité d'opérations, ou utiliser un ordre de grandeur. Si besoin, augmenter à trois éléments ou plus.

- Arrondir le réel à un rationnel simplement représenté mais avec une précision par rapport à ce dernier raisonnable (s'il existe).

CONCLUSION

Ce projet est intéressant car il nous a poussé à implémenter et à se servir des différentes notions vues en cours. La création d'une *library* propre, accompagnée d'une documentation, est satisfaisant et ce pareillement pour l'environnement *Cmake*, bien que parfois un peu complexe à la mise en place.

Au niveau de l'environnement de travail, Théo était sous une distribution de Linux (Ubuntu 20.04.5 LTS) et Lilou sous macOS (M1).

La mise en commun des informations ont été faites lors des séances de cours dédiées au projet et des échanges se sont effectués par l'intermédiaire de l'application Discord. Pour ce qui est du code, nous utilisions GitHub pour facilement voir nos avancées respectives et les étapes (*cf.* lien en fin de page).

Lilou s'est consacrée aux tests unitaires et à la structuration du projet en template (tri des fonctions, cas particuliers, etc…) tout en aidant parfois Théo sur la compréhension de certaines erreurs, qui lui s'est occupé principalement des fonctionnalités (opérateurs, fonctions, etc…).

Lien GitHub: https://github.com/MissBidule/ProgMaths_ALIDOR_COUARDT