CREPIN Olivia
FLEITZ Maëlle

# Rapport projet Maths Informatique Programmation
# Les rationnels

| **Éléments demandés codés** | **Fonctionne** |
|---|---|
| dénominateur positif | Oui |
| fonction de passage en fraction irréductible | Oui |
| conversion d'un réel en rationnel | Oui |
| nombre 0 noté (0/1) | Oui |
| nombre infini noté (1/0) | Oui |
| moins unaire | Oui |
| valeur absolue | Oui |
| partie entière | Oui |
| opérateurs de calcul entre deux rationnels (+, -, *, /) | Oui |
| inverse d'un rationnel | Oui |
| produit d'un nombre à virgule flottante avec un rationnel | Oui |
| produit d'un rationnel avec un nombre à virgule flottante | Oui |
| opérateurs de comparaison | Oui |
| fonction d'affichage : surcharge de << | Oui |
| Mise en place de tests unitaires | Oui |
| fonctions template des opérateurs | Oui |
| exceptions | Oui |
| asserts | Oui |
| constexpr | Oui |
| documentation Doxygen | Oui |
| CMake | Oui |
| **Éléments demandés non codés** | **Aucun** |
| **Éléments non demandés** | **Fonctionne** |
| autre opérations (+, -, /,*) d'un nombre à virgule flottante avec un rationnel | Oui |
| autre opérations (+, -, /,*) d'un rationnel avec un nombre à virgule flottante | Oui |
| opérateurs d'assignation (+=,-=,*=,/=) | Oui |
| opérateurs d'incrémentation (++, --) | Oui |
| opérateur de calcul : % | Oui |
| logarithme (base e et base 10) | Oui |
| fonctions trigonométriques (cos,sin,tan) | Oui |

## Partie Mathématiques

**Formalisation de l'opérateur division /**
Nous avons décidé de formaliser cette opérateur, suivant sa définition, comme étant :

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

On a également ajouté deux conditions d'arrêt dans le cas ou l'un des deux numérateurs, de la première partie de l'égalité, est égal à 0 (on retourne 0/1), si les deux rationnels sont égaux on retourne 1/1 directement.

**Formalisation de l'opérateur racine carrée**
Dans notre code, l'opération de racine carrée se fait par une fonction nommée square_root et non par le symbole √ puisqu'il n'existe pas en programmation. C'est également plus compliqué à afficher dans la console puisque le symbole ne fait pas partie des symboles ascii classiques.
Pour effectuer une racine carrée d'un nombre rationnel, nous nous sommes basés sur la propriété suivante :

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

Nous avons ensuite écrit cet algorithme:

```
var numerateur_en_ratio, denominateur_en_ratio, sqrt_en_ratio

initialiser numerateur_en_ratio à sqrt(this.numerateur) /1
initialiser denominateur_en_ratio à 1/sqrt(this.denominateur)
initialiser sqrt_en_ratio numerateur_en_ratio*denominateur_en_ratio

simplifier sqrt_en_ratio

retourner sqrt_en_ratio
```

Dans cet algorithme, on sépare en deux rationnels le numérateur et le dénominateur de notre rationnel d'origine. Individuellement, on fait la racine carrée des deux parties, ce qui nous donne *sqrt(numerateur)/1* et *1/sqrt(denominateur)*. On multiplie les deux rationnels ensembles.
Nous avons mis un assert pour éviter le cas où le Ratio est négatif.

**Formalisation de l'opérateur cosinus, sinus et tangente**
Formaliser ces opérateurs de trigonométrie s'est avéré plus complexe que prévu. Nous avons fait beaucoup de recherches pour trouver une solution qui permettait de ne pas passer par un float mais les solutions trouvées étaient très complexes pour nous. Ainsi, nous avons fait une version simplifiée de nos cosinus, sinus (et donc nécessairement tangente)

qui convertit notre rationnel en float puis le float en un rationnel à la fin de l'opération. Le problème est que l'on perd en précision en faisant cela. En convertissant un type ratio en un float et inversement, on fait à chaque fois des arrondis.

Nous avons mis un assert pour éviter le cas où l'on fait tan(pi/2) qui est impossible à faire.

Nous avions aussi pensé à formaliser l'opérateur cosinus à partir de la série de Taylor. Cependant, cette série utilise les factorielles ce qui augmente énormément la complexité de l'algorithme. Nous avions ensuite envisagé d'utiliser les exponentielles, notamment avec la formule d'Euler mais l'utilisation de nombres complexes nous aurait rendu la tâche plus difficile.

**Formalisation de l'opérateur absolue**
Pour la fonction absolue, nous avons d'abord codé une première version sans utiliser la librairie c++ <cmath>. Toutefois, quand nous avons fait toutes les autres méthodes et fonctions, nous nous sommes rendu compte que nous utilisions systématiquement <cmath>. Pour plus de cohérence dans la globalité de notre code, nous avons alors changé les deux seuls opérateurs qui n'utilisaient pas la <cmath> : absolue et puissance.

Sans <cmath> voici ce que nous avions fait:
Pour rendre une valeur absolue d'un nombre rationnel, on a d'abord fait en sorte que le dénominateur soit forcément positif (avec une fonction à part : (denom_positif()). A partir de là on a tout simplement fait en sorte que si le numérateur était négatif on le passe en positif.

```
Si denominateur<0
    numerateur *= -1
    denominateur *= -1
Si numerateur <0
    retourner rationnel(-numerateur, denominateur)
Sinon
    retourner rationnel(numerateur, denominateur)
```

Avec <cmath> nous nous sommes basé sur la formule qui suit:

$$\left|\frac{a}{b}\right| = \frac{|a|}{|b|}$$

Ainsi on fait l'absolue séparément sur le numérateur et sur le dénominateur.

**Formalisation de l'opérateur puissance**
Pour la puissance nous avons décidé d'utiliser une fonction récursive qui s'appelle tant que la puissance *n* n'est pas décrémentée à zéro. Cette fonction ne marche que pour les entiers positifs (0 inclus), car les puissances négatives causerait une erreur *out_of_range*, les valeurs étant hors de limite de la fonction.

```
Si la puissance n == 0
    retourner rationnel de 1
Sinon
    retourner rationnel * puissance(rationnel, n-1)
```

Comme pour la fonction absolue nous avons implémenté deux façons différentes de coder l'opérateur puissance avec ou sans <cmath>. Nous avons donc gardé la version avec cmath pour rester cohérent. De plus, ça demande un peu moins d'effort au processeur d'utiliser <cmath> que de faire une fonction récursive.

Ainsi avec <cmath> on fait la puissance sur le numérateur et le dénominateur séparément.

$$(\frac{a}{b})^{\mathrm{n}} = \frac{a^{\mathrm{n}}}{b^{\mathrm{n}}}$$

**Formalisation de l'opérateur logarithme**
Pour le logarithme nous nous sommes basé sur la formule qui suit :
$log(a/b) = log(a) - log(b)$
Nous avons ajouté également un assert puisque le domaine de définition de log(x) est ]0, +∞[
Pour le logarithme on a codé deux versions, pour le log en base e et le log en base 10.

**Formalisation de l'opérateur exponentielle**
L'exponentielle d'un rationnel peut s'écrire sous la forme :

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

Nous sommes donc parties de cette forme pour coder l'opérateur. Au lieu de s'occuper d'une exponentielle d'un rationnel, nous avons individualisé chaque membre de la fraction pour qu'on ait à calculer une exponentielle d'un entier et une puissance d'un nombre à virgule flottante. Le soucis avec cette fonction est que les nombres augmentent très vite et viennent rapidement dépasser la limite des *int*.

**Conversion d'un réel à un rationnel**
En ce qui concerne l'algorithme, nous avons pu remarquer que la **ligne 8** permet de générer le dominateur du rationnel. En effet, on peut voir que la puissance -1 crée un rationnel équivalent à $\frac{1}{rationnel}$.
Par exemple, si nous prenons le nombre 0.5, en suivant l'algorithme nous accédons à la ligne 8 (car 0.5 < 1). Par récursivité, on retourne alors $\frac{1}{convertFloatToRatio(\frac{1}{0.5}, nbIter)}$ soit $\frac{1}{\frac{2}{1}}$ donc $\frac{1}{2}$.

Pour la **ligne 12,** la somme correspond à l'addition de toutes les parties calculées (numérateur et dénominateur) pour former le rationnel.

L'algorithme ne prenait en charge que les **réels positifs**. Nous avons remarqué que cela était dû aux deux conditions à la ligne 7 et 10 : $(x < 1)$ et $(x \geq 1)$. Ces deux conditions ne prennent pas en compte les valeurs négatives, elles sont donc hors limite. Pour pallier ce problème, nous avons rajouté un deuxième argument à ces deux conditions.

Pour la condition de la ligne 7, nous avons choisi de mettre $(x > -1)$ && $(x < 1)$. Ce qui est important dans cette condition est que le réel $x$ doit avoir une valeur à virgule flottante autour de 0 pour pouvoir former le dénominateur, il ne doit donc pas devenir plus petit que -1 et plus grand que 1.

Pour la condition de la ligne 10 nous avons opté pour $(x \geq 1 \,||\, x \leq -1)$. C'est dans cette condition que l'on s'occupe des valeurs plus petites ou égales -1, pour avoir la partie du numérateur négative.

**Analyse**

Précédemment nous avons mentionné que les **grands nombres et très petits nombres se représentent mal**. Une des explications est liée à l'implémentation de notre classe rationnel. En effet, nous avons défini les numérateurs et dénominateurs par des $int$**,** soit, codé sur 32 bits. Ces nombres doivent donc dépasser rapidement cette limite d'implémentation car ils prennent beaucoup d'espace et de bits. Pour régler cela, nous pourrions passer les types du numérateur et dénominateur en template afin de générer des `long int` ou `long long int` et travailler sur des tailles de rationnels beaucoup plus grandes.

De plus, avec l'algorithme donné, on ne crée qu'un seul rationnel. Or, il est impossible de représenter, et des petites valeurs, et des grandes valeurs de manière la plus exacte possible en créant qu'un seul rationnel. Un seul rationnel ne permet pas assez de précision pour ces nombres, car pour les nombres grands, il faudrait un numérateur très grand alors que pour les nombres petits, il faudrait un dénominateur très grand. À la place, une solution envisageable serait de générer une somme de rationnels, un rationnel représentant la partie entière, un rationnel représentant la partie décimale. Sinon on risquerait de se faire bloquer par cette partie entière qui est trop grande. Par exemple, si le nombre que l'on souhaite convertir est très grand avec décimales et qu'avec cet algorithme, on obtient un dénominateur grand, le numérateur sera encore plus grand et c'est à cause de ce dernier que l'on dépassera la limite. En séparant la partie entière et la partie décimale, on évite de bloquer au numérateur.

Une deuxième solution serait de diminuer le nombre d'itérations, cela causerait certes une perte de précision mais cela éviterait de dépasser la limite de représentation des entiers.

**Détails de la structure du projet**

Notre librairie pour les rationnels prend la forme d'une grande classe Ratio définie dans ratio.hpp et implémentée dans ratio.cpp. Cette classe est composée de deux attributs privés qui sont les entiers m_num et m_denom, le numérateur et le dénominateur de notre rationnel. Cette classe est composée d'un grand nombre de méthodes: des getter, des setter pour voir et accéder aux deux attributs; des surcharges d'opérateurs, les opérateurs absolu et racine_carree sous formes de méthodes ainsi que des méthodes permettant certaines manipulations du ratio (simplification, conversion…).

En dehors de notre classe Ratio, nous avons implémentés la surcharge des opérateurs de comparaison et les fonctions des opérateurs inverse, puissance, exponentielle puisque c'est plus logique d'appeler ces fonctions avec notre ratio en paramètre plutôt que d'appeler la fonction sur notre ratio (exemple: puissance(ratio,2) plutôt que ratio.puissance(2)). On garde des conventions utilisées dans d'autres librairies pour plus de cohérence.

En dessous encore, nous avons placé tous les opérateurs qui pouvaient utiliser des templates.

Le projet se présente sous la forme de plusieurs sous dossiers, dont le dossier de notre librairie "/lib_Ratio", celui de nos exemples "/my_examples", et nos tests unitaires "/test_units". Notre librairie est entièrement en anglais et formalisée en snake case.

**Problèmes rencontrés**

Le premier problème que nous avons rencontré était de trouver le **nombre d'itérations nécessaires** pour convertir les nombres à virgule flottantes en rationnel. Nous avons choisi alors 9, car nous avons pu observer que les nombres très grands et très petits perdaient en précision avec un faible nombre d'itérations. Nous avons donc décidé de chercher quel nombre d'itérations serait le plus efficace en faisant le calcul d'erreur, c'est-à-dire la différence entre leur valeur réelle et celle que l'on obtient avec notre algorithme. Nos numérateurs et nos dénominateurs sont tous des int, or si la précision est trop grande on se retrouve avec des rationnels possédant des numérateurs très grands sur des dénominateurs tout aussi grands. Toutefois, il ne faut pas dépasser la taille maximale d'un int.

| | | | | |
|---|---|---|---|---|
| int | ≥ 16 bits (processeur 16 bits) | -32 768 | $-(2^{15})$ | +32 767 |
| | ≥ 32 bits (processeur 32 bits)[4] | -2 147 483 648 | $-(2^{31})$ | +2 147 483 647 |

Comme nous pouvons le voir sur le tableau ci-dessous, un int pour un processeur 32 bit les bornes sont ±2 147 483 647 dans ce nombre on compte un total de 10 chiffres or si on autorisait une précision de 10 chiffres cela voudrait dire que notre Int puisse aller jusqu'à 9 999 999 999, ce qui n'est pas possible, ainsi en choisissant 9 chiffres de précisions on est sûr de ne pas dépasser la limite de taille d'un int lors de son initialisation.

Cependant, nous avons pu observer que le nombre d'itérations dépendait aussi de la précision que l'on souhaitait avoir et du nombre en lui-même et de son type. Pour que l'algorithme soit le plus efficace possible il faudrait calculer la précision nécessaire pour chaque nombre et en déduire par la suite le nombre d'itérations.

Aussi, nous avons remarqué que la création du rationnel du double 151.62 ne fonctionnait plus à partir de la 5e itération. Les valeurs données étaient erronées, puis on obtenait une erreur déclenchée par l'assert, car il dépassait la limite de représentation dans les itérations suivantes. Or, avec un nombre beaucoup plus grand comme 2101.5562, il n'y a eu aucun problème. Mais, en passant 101.52 en float, il n'y a plus eu cette erreur. Il faudrait voir si en passant nos numérateurs et dénominateurs en long int ou long long int la conversion de 101.52 aurait fonctionné.

---

Nous avons pu aussi remarquer que les **opérations ne marchaient pas dans les deux sens**. C'est-à-dire que nous pouvions, par exemple, ajouter un rationnel avec un entier ou un nombre à virgule flottante, sans qu'il y ait d'erreur. Tandis que l'inverse ne fonctionnait pas. Cela était dû au fait que le compilateur convertissait l'entier ou le nombre à virgule flottante en utilisant les constructeurs que nous avions créés. Pour résoudre cela, nous avons créé des fonctions template de ces opérateurs prenant cette fois-ci en premier paramètre un nombre *template* donc entier ou réel.

Un problème similaire est survenu lors de la surcharge des opérateurs d'incrémentation et de décrémentation ++ et --. Nous nous sommes rendu compte que si on faisait ++ratio et ratio++ cela marchait uniquement avec le premier. C'est à ce moment-là que l'on a trouvé qu'il fallait implémenter séparément ces deux modes d'utilisation de ces opérateurs. Ainsi nous avons fait des opérateurs pour les préfix et pour les postfix.

Une autre difficulté était de répartir les fonctionnalités qui méritaient d'être des méthodes de la classe et d'autres juste des fonctions utilisant la classe. Nous en avions d'abord défini certaines en méthodes, mais nous nous sommes vite rendu compte que cela nous causait des erreurs quand on souhaitait les appeler, ou que cela nous complexifiait l'écriture de leur appel. Puis, nous avons réalisé que les avoir définies en tant que méthode, n'était pas le plus logique dans notre cas. Par exemple, un rationnel ne va pas appeler sa méthode pour se convertir de flottant en rationnel, de même pour certains opérateurs.

Nous avons également eu des difficultés avec les fonctions de trigonométrie qui exigent un code très complexe sans quoi nous sommes obligé de repasser par des float. Ainsi, ces fonctions perdent en précision ce qui est pourtant ce qu'on évite de faire en utilisant les rationnels. Nous n'avons pas trouvé de méthode simple permettant une implémentation précise d'un cosinus (et d'un sinus) de rationnel.