CR - Arkulib Project

Leclercq Mattéo
Massa Elise

3 janvier 2023

Table des matières

1 Partie Mathématique

1.1 Opérations sur les rationnels

Question 1 :

A partir de la définition d'un rationnel donnée par le sujet, nous implémentons les opérations basiques sur ces derniers dans notre bibliothèque. On considère comme opération basique l'addition, la soustraction, la multiplication, la division et l'opération inverse $^{-1}$.

Soit les deux nombres rationnels $\frac{a}{b}$ et $\frac{c}{d}$ avec $\begin{cases} a, c \in \mathbb{Z} \\ b, d \in \mathbb{N}^* \end{cases}$

De la même façon, que sont formalisées les opérations +, x et $^{-1}$ dans le sujet, on formalise l'opération / :

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

Question 2 :

On s'intéresse ensuite à quelques opérations mathématiques sur nos rationnels.

- la puissance ;
- la racine carrée (sqrt) ;
- la fonction cosinus ;
- la fonction exponentielle.

On formalise toutes ces opérations par le même raisonnement. On calcule d'abord de manière "brute" le résultat avec une fonction c++ (`std::sqrt()` par exemple) avant de le transformer en rationnel. Nous avons choisi de réceptionner le résultat de la fonction c++ avec un type double, suffisamment précis pour cette utilisation.

Pour le cas de la puissance, nous nous sommes longuement interrogés sur la méthode à adopter. L'une des possibilités était de simplement renvoyer un rationnel ayant reçu l'opération de puissance sur le numérateur et sur le dénominateur. Cette solution nous aurait épargné l'utilisation de la fonction fromFloatingPoint, convertissant un flottant en un rationnel. Nous avons cependant choisi de nous économiser l'utilisation d'un std : :pow en ne l'effectuant qu'une seule fois.

Exemple

```
1     template<typename IntType>
2     constexpr Rational<IntType> Rational<IntType>::sqrt() const {
3         if (isNegative()) throw Exceptions::NegativeSqrtException();
4         // Le résultat est un rationnel, la fonction fromFloatingPoint est appelé dans son constructeur
5         return Rational<IntType>(
6                 std::sqrt(static_cast<double>(getNumerator()) / getDenominator())
7                  // Calcul de la racine du rationel avec std
8         );
9     }
```

Pour le reste, ces opérations mathématiques complexes ne peuvent renvoyer de résultat en rationnel de façon évidente, c'est pourquoi il est plus intéressant en terme d'efficacité du code de calculer via la bibliothèque `std`.

1.2 Conversion d'un réel en rationnel

Question 3 :

On veut modifier l'algorithme 1 de la partie 3.2 du sujet afin que celui-ci puisse également gérer les nombres négatifs. Puisque celui-ci est récursif, il s'agit donc de traiter le cas où le nombre réel x pris en entrée est négatif à l'aide d'une nouvelle condition (`if`) récursive.

En prenant $x \in \mathbb{R}$ et `nb_iter` $\in \mathbb{N}$ comme argument de notre fonction convert_float_to_ratio, on ajoute le pseudo-algorithme suivant à l'algorithme 1 de la partie 3.2 :

```
1 if (x < 0) {
2     return -convert_float_to_ratio(-x, nb_iter);
3 }
```

Ainsi, pour un $x \in \mathbb{R}^-$, on ajoute une récursion en plus puis on déroule les opérations de la même façon que pour $x \in \mathbb{R}^+$. Finalement la complexité de l'algorithme demeure inchangée.

Remarque : la puissance -1 de la ligne 8 et la somme de la ligne 12 de l'algorithme 1 s'adresse à notre objet de type `rationnel` qui est créé à partir du réel d'un type `float` pris en entrée.

1.3 Analyse

Question 4 :

La mauvaise représentation des grands nombres avec notre classe de rationnels est essentiellement due à la limite de précision des nombres en c++.
En effet, le numérateur et le dénominateur sont représentés par des entiers. Or, les entiers ont une taille finie en mémoire. Par exemple pour un entier de type `int`, on nomme cette taille maximale `INT_MAX`.

Alors, en essayant de représenter une fraction x (tel que $x = \frac{num}{denum}$) avec `num` et/ou `denum` très grand, le risque est que `num > INT_MAX` auquel cas le résultat est incorrect.

Remarque : on peut d'ailleurs noter que `num > INT_MAX` est suffisant pour avoir un résultat incorrect.
Le cas `denum > INT_MAX` n'est pas "nécessaire" pour que x soit qualifié de grands nombres dans notre classe de rationnel.

Question 5 :

Nous avons constaté en effectuant des tests unitaires sur notre bibliothèque que l'enchaînement d'opérations entre des rationels entraîne des valeurs très grandes au numérateur et au dénominateur. Or, nous avons vu à la question précédente que les grands nombres sont mal gérés par notre classe de rationnel. C'est pourquoi nous avons réfléchi à plusieurs solutions.

La première solution, ou celle que nous avons gardée et implémentée dans notre classe ERational :

Le numérateur et le dénominateur seront représentés à l'aide de deux valeurs : un flottant ainsi qu'un entier (`short int` dans le programme). L'utilité est de limiter un maximum l'utilisation de grands nombres qui sont difficilement portés en C++.

Ainsi, le flottant permet d'exprimer de stocker la valeur initiale de manière plus précise. Dans notre programme, l'utilisateur a le choix du flottant utilisé (float, double, long double...). L'entier, quant à lui, correspond à l'"exposant" et compte

donc le nombre de décimales passées dans la partie flottante.

Un rationnel a donc la forme suivante : $\frac{k * 10^n}{j * 10^m}$ avec les k, j correspondant au flottant et les n, m correspondant à l'entier.

Exemple : pour écrire le nombre $x = 24075075075075$ comme un nombre rationnel, x sera composée de la paire suivante :

- 2,4075075075075 pour le flottant
- 13 pour l'exposant
- Ainsi, x peut être retrouver en effectuant le calcul 2,4075075075075 * 10^{13}

La deuxième solution, à partir du théorème suivant :

Théorème 1 : soit $x \in \mathbb{R}$
Alors $\forall n \in \mathbb{N}$

- $\alpha_n = \frac{\lfloor 10^n x \rfloor}{10^n}$ est une valeur approchée de x à 10^{-n} près par défaut.
- $\beta_n = \frac{\lfloor 10^n x \rfloor + 1}{10^n}$ est une valeur approchée de x à 10^{-n} près par excès.

$\Rightarrow$ Pour la suite, on fait le choix arbitraire de travailler avec une valeur approchée α_n de notre fraction x à 10^{-n} près par défaut pour les gands nombres.

On cherche donc à déterminer α_n comme un nombre rationnel tel que $\alpha_n = \frac{\nu}{\delta}$

Exemple : on veut écrire le nombre $x = 2{,}4075075075075$ comme un nombre rationnel.
Or, on sait que :

Prop 1 : le développement décimal propre de tout nombre rationnel est périodique.

D'après cette propriété, on peut prendre $x \approx 2{,}4\underline{075}$ et on cherche α_3 un nombre rationnel approché à 10^{-3} près par défaut.
On en déduit : $\begin{cases} \nu = \lfloor 10^3 x \rfloor = 2407 \\ \delta = 10^3 \end{cases}$

D'où $\alpha_3 = \frac{2407}{1000}$

Pour aller plus loin, on peut vérifier et préciser le résultat suivant par la méthode suivante :

$10^3 x$ - $x = 999x$ *(disparition décimale)*
$\Rightarrow$ 2407,5075 -2,4075 $= 999x$
$\Rightarrow$ 2405,1 $= 999x$
$\Rightarrow x = \frac{24051}{9990}$

Après simplification, on obtient $\boxed{x = \frac{8017}{3330}}$

Une vérification à la calculatrice prouve que 8017 / 3330 = 2,4$\underline{075}$.
Finalement, cette méthode est une solution possible pour mettre sous forme de rationnel un grand nombre.

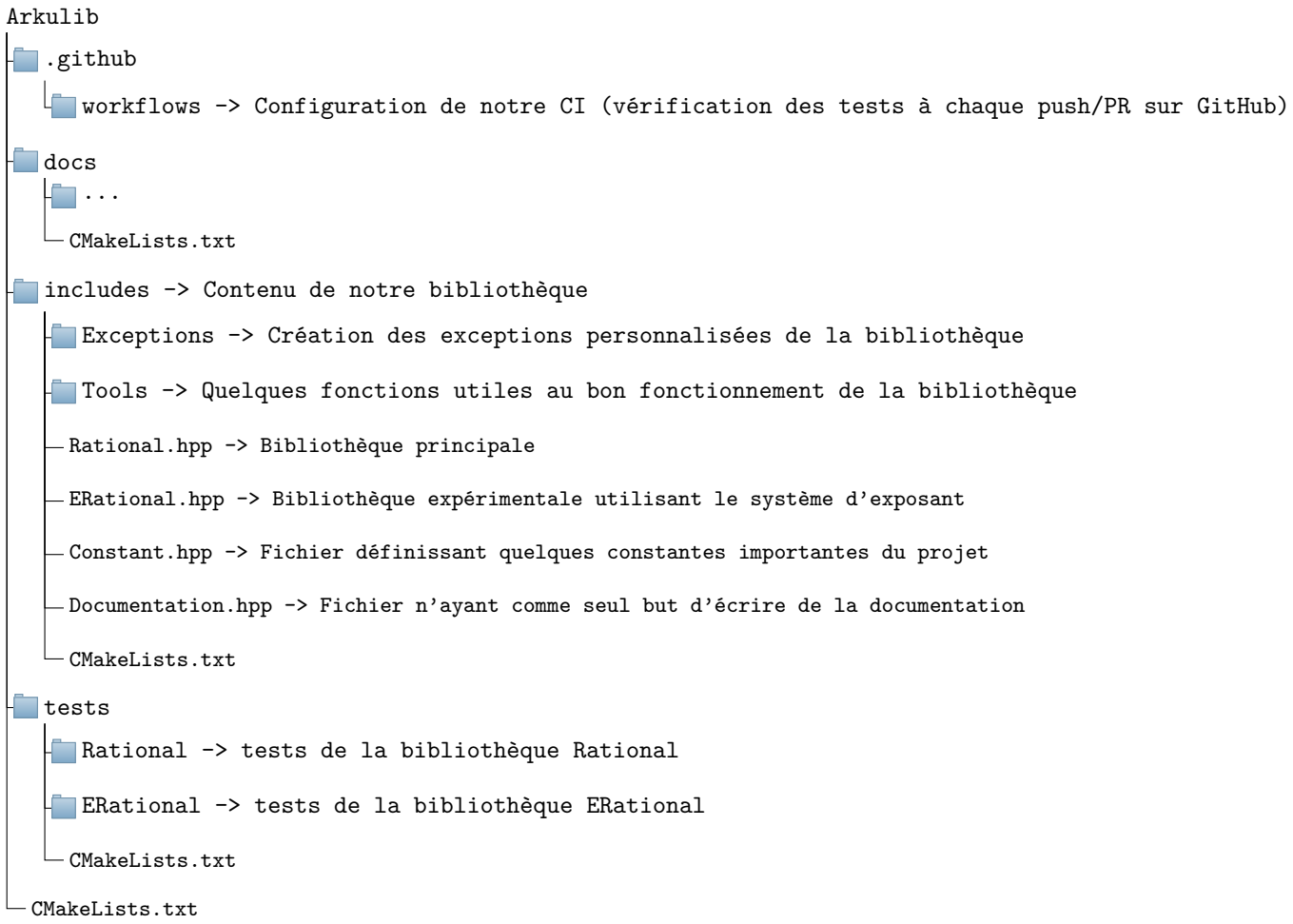
La troisième solution :

Une solution, bien plus pragmatique, est d'installer en dépendance une bibliothèque comme GMP (GNU Multiple Precision Arithmetic Library) qui permet de faire des calculs à virgule flottante et à précision arbitraire. Cette solution, plus lourde et moins expérimentale, aurait pu permettre de ne pas limiter le numérateur et le dénominateur.

2 Partie Programmation

2.1 Architecture du projet

Voici un arbre non-exhaustif de notre projet !



2.2 Notre bibliothèque de rationnel

2.2.1 Construction d'un rationnel

Notre bibliothèque Arkulib gère les nombres rationnels via la classe **Rationnal**, celle-ci est entièrement codée en **template**, et permet de construire notre objet rationnel avec 2 paramètres : **numerator** et **denominator**.
Ceux-ci sont du type template `IntType`, ce type permet de traiter séparément le numérateur et le dénominateur comme des entiers sans se restreindre au type *int*, et pouvoir élargir aux types *long int* ou *long long int*.

Dans toutes notre bibliothèque nous avons utiliser `IntType` pour les nombres entiers. Par le même raisonnement nous avons aussi utilisé d'autres template comme `AnotherIntType` pour pouvoir comparer deux entiers ou `FloatType` pour traiter les flottants.

Constructeur par copie
Nous traitons le cas de la construction d'un rationnel à partir d'un autre déjà existant. Si le rationnel créé est du même type que celui d'origine alors `default` suffit pour déinir le constructeur.
Nous avons également établi un constructeur par copie ayant des types d'entiers (templates) différents. Nous avons donc traité les cas où un rationnel est stocké depuis un entier plus large (long long int -> int). La classe `ERational` a également un constructeur par copie prenant en paramètre un `Rational`.

Construction d'un rationnel à partir d'un non-rationnel
Le dernier constructeur codé permet de créer un rationnel à partir d'un type inconnu (supposément flottant). Sa mécanique peut être un peu surprenante : nous commençons par appeler la fonction fromFloatingPoint qui transforme un flottant en

un rationnel et nous le stockons dans un rationnel de template `long long int`. Cela nous permettra de nous apercevoir de potentielles erreurs d'overflow. Auquel cas une exception (*TooLargeException*) surviendra lors du constructeur par copie vers un rationnel de type `int` par exemple.

Finalement, nos 3 types de constructeurs permettent de couvir un grand champs de possibilités pour la création de rationnel. S'agissant du destructeur, on utilise un *default* classique.

2.2.2 Forme irréductible d'un rationnel

La partie 2.2 du sujet permet de définir la forme irréductible d'un rationnel à partir de laquelle est codée la fonction `simplify()`. Cette fonction prend en argument un `Rational<IntType>` et le renvoie sous sa forme irréductible.

Pour cela, on utilise la fonction `gdc()` de la bibliothèque `std`. Nous avions codé en premier lieu le pgcd avec l'algorithme d'Euclide avant de constater que c'est une perte de temps puisque `std::gcd()` est la méthode la plus optimale pour determiner un pgcd.

La fonction `simplify()` est appelée à de nombreuses reprises dans le code, notamment dans les constructeurs, pour éviter d'effectuer des opérations (successives ou non) avec de grand nombres et limiter ainsi le risque d'erreur.

2.2.3 Opérations basiques sur les rationnels

Les opérations basiques +, -, *, / et l' inverse $^{-1}$ sont toutes codées par le même raisonnement dans notre bibliothèque et sont commentés de façon claire.

Pour la partie ERational, les opérations ont été entièrement retravaillées pour s'adapter à la nouvelle forme du rationnel. Ils suivent cependant le même principe que la classe originelle.

Cas particulier du moins unaire :

```
1 inline friend Rational operator-(const Rational &rational) {
2     return Rational<IntType>(-rational.getNumerator(), rational.getDenominator());
3 }
```

2.2.4 Opérateurs mathématiques plus complexe et conversion d'un flottant en rationnel

Pour les opérateurs mathématiques plus complexes, on fait la méthode expliquée dans la partie 1.1. Le seul cas particulier rencontré est celui de la racine d'un nombre négatif qui doit renvoyer une erreur. Pour cela on utilise une boucle `if ... throw ...`

Notre fonction fromFloatingPoint()

Cette fonction prend 2 arguments `const FloatType floatingRatio` et `size_t iter` et renvoie un `Rational<IntType>`.
Nous avons exploré une piste de recherche pour cette fonction :

- Tentative prédiction du nombre d'itération ;

  L'objectif était de comprendre théoriquement le nombre d'itération nécessaire pour convertir un nombre réel en nombre rationnel via notre algorithme récursif.
  En testant des nombres plus ou moins grand, on remarque que le nombre d'itération ne varie pas de façon conséquente (entre 2 et 8 itérations pour arriver à une précision exacte). Cependant, nous n'avons pas trouvé de méthode efficace pour prédire ce nombre avec nos connaissances en mathématiques et en programmation même après quelques recherches sur Internet.
  C'est pourquoi nous avons fini par prendre la décision de conserver le paramètre `iter` comme un argument choisi de façon arbitraire lors de l'appel de la fonction. Nous avons défini sa valeur par défaut à 10. En effet, nous avons testé le nombre d'itérations pour une quarantaine de valeurs et nous nous sommes rendus que 10 était un bon compromis puisque l'algorithme ne dépassait jamais cette limite pour ces valeurs.
  En guise de synthèse ; nous aurions préféré que ce paramètre ne soit pas statique et s'adapte au flottant donné, mais nous n'avons pas réussi.
  Cela a été toutefois une piste de recherche intéressante à creuser.

2.2.5 Opérateurs de comparaison

Nous avons implémenté les opérateurs de comparaison suivants : ==, !=, <=, <, >=, >.
Ces opérateurs sont tous testés dans `comparaison.cpp` dans le dossier Test. En particulier, l'opérateur == est fondamental dans la construction des tests des autres opérateurs puisqu'il permet de vérifier nos résultats.
Chaque opérateur peut comparer deux rationnels entre eux, un rationnel avec un non-rationnel ou un non-rationnel avec un rationnel à partir d'une approximation via la fonction `toRealNumber()` qui permet d'obtenir un nombre flottant approximatif des rationnels. L'idée est de comparer deux rationnels à partir de la comparaison entre deux réels usuels en C++.

2.2.6 Conversion d'un rationnel

Nous avons ajouté 5 fonctions qui permettent de convertir et de manipuler un rationnel plus facilement en utilisant d'autres types ou d'autres formes.
Notre bibliothèque permet ainsi de pouvoir transformer une variable de type `Rational` en un type `int` ou `string`.
De plus, il est possible de faire une approximation d'un rationnel donné avec la fonction `toApproximation()` :

```
1 template<typename IntType>
2 constexpr Rational<IntType> Rational<IntType>::toApproximation(
3         const unsigned int digitsKept
4 ) const {
5     if (digitsKept > Constant::DEFAULT_MAX_DIGITS_APPROXIMATE)
6         throw Exceptions::DigitsTooLargeException();
7
8     return Rational<IntType>(
9         Tools::roundToWantedPrecision(toRealNumber<double>(), std::pow(10,digitsKept)));
10    }
```

Cette fonction renvoie simplement une approximation d'un rationnel à la précision voulue (le paramètre est le nombre de chiffre après la virgule).

Enfin, comme déjà evoqué dans la partie précédente, la fonction `toRealNUmber()` permet d'obtenir un nombre flottant approximatif à partir d'un rationnel.

2.2.7 Cas particulier des grands nombres

Comme expliqué précédemment, traiter les grands nombres a été la plus grande difficulté rencontrée dans ce projet. C'est pourquoi nous avons orienté notre bibliothèque de la façon suivante : une partie axée pratique qui se veut efficace, fonctionnelle et simple à prendre en main par l'utilisateur, une autre partie axée sur la recherche qui approfondit la théorie derrière les nombres rationnels sous forme d'exposant.
Cette deuxième partie est beaucoup moins développée que la première, mais elle a été développé de sorte à ce qu'elle soit facilement maintenable et extensible.

2.3 Exemples et Tests unitaires

Nous avons mis en œuvre des tests unitaires tout au long du projet pour s'assurer que notre bibliothèque est opérationnelle. Les tests sont réalisés à l'aide de Google Test dans un dossier à part dépendant d'un CMake. Nous avons d'ailleurs fait le choix d'importer Google Test à l'aide d'un FetchContent.

Au niveau de la stratégie abordée pour nos tests, nous avons essayé de couvrir un maximum de cas d'utilisations pour toutes les fonctionnalités qu'implémente notre bibliothèque. Nous avons pris la décision de comparer des valeurs dont nous avons calculé et vérifié les valeurs au préalable. Cette méthode nous a permis d'effectuer de nombreux tests pour une vingtaine de fonctionnalités.
En effet, pour tous les opérateurs et autres méthodes, nous avons implémenté une série de tests de base. Ils sont réalisés dans des fichiers séparés pour garder une organisation irréprochable ; ils ont généralement cette forme : test brut du fonctionnement de l'opération avec deux rationnels, opérations consécutives avec le même opérateur, opération avec 0 ou 1, simplification et opération avec des grands nombres. Lorsque cela s'y prêtait, nous avons rajouté des tests plus spécifiques à cette série de tests de base.

Coder les tests ne nous a pas posé de difficultés particulières puisque la prise en main de ggtest a été bien assimilée en cours. Ces derniers ont été indispensables ; ils nous ont permis de mettre en lumière des défauts de notre code notamment dans la prise en charge des grands nombres.

Pour la partie ERational, nous avons utilisé le système C++ de fonctions d'opérateurs binaires (std : :plus, std : :minus, std : :multiplies) qui nous a fait gagner un temps très précieux.

2.4 Conclusion

La difficulté principale que nous avons rencontré était sans aucun doute de trouver la meilleure manière de coder un rationnel de grande taille. Dans un second temps, la création et la bonne implémentation de la classe ERational n'ont pas été évidentes ; nous sommes passés par beaucoup de versions expérimentales pour arriver au résultat final.

Pour conclure, ce projet a été l'occasion pour nous de mettre en œuvre les notions apprises en cours de façon structurée et réfléchie. Nous avons apprécié travailler ensemble tout au long de ce semestre pour obtenir un résultat aussi concret que celui-ci !