CR - Projet Ratio

Romain SERRES - Mathilde STOCCHI

26 décembre 2022

Table des matières

1 Bilan des fonctionnalités de la librairie

ÉLÉMENTS DEMANDÉS	STATUT
Class Rational	Codé :)
Zéro	Codé :)
Infini	Codé :)
Constructeurs (default, copy, value, input)	Codé :)
setters and getters .n() et .d()	Codé :)
invert(), toFloat()	Codé :)
Opérateurs	Codé :)
Opérateurs de comparaison	Codé :)
Fonction d'affichage <<	Codé :)
cos(), sin(), tan(), pow(), sqrt()	Codé :)
Algorithme de conversion	Codé :)
Main() - "une démo incroyable dont vous êtes le héros"	Codé :)
Tests unitaires	Codé :)
Cmakelist	Codé :)
Usage d'outils de la STL	Codé :)
Polymorphisme	Codé :)
Doc avec doxygen	Codé :)
Asserts pour prévenir les erreurs	Codé :)
Exception - UnitTest ASSERT_THROW	Codé :)
Un code super lisible et d'une qualité ++	Codé :)
Gestion des grands nombres	Vite fait :/

ÉLÉMENTS OPTIONNELS	STATUT
abs(), floor(), sign()	Codé :)
Class Rational en template	Codé :)
Toute la classe en constexpr	Codé :)
Fonctions variadics min() max()	Codé :)
Espage de nommage (internal et rational	Codé :)
Fonctions lambda	Pas codé :(

FIGURE 1 – tâches réalisées

2 Opération sur les rationnels

2.1 Formalisation de certains opérateurs

Tout nos opératereurs ont été surchargés de façon à pouvoir réaliser des opérations entre le numérateur et le dénominateur de deux nombres rationnels, sans passer par une écriture décimale.

Exemple de l'opérateur de division : $\dfrac{\frac{a}{b}}{\frac{c}{d}} = \dfrac{a \times d}{b \times c}$

Concrètement, il s'agit ici de multiplier notre rationnel $\frac{a}{b}$ par l'inverse de $\frac{c}{d}$

Vous trouverez ci-dessous le pseudo-code de l'opérateur * :

Algorithme 1 : operator * overloading

1 operator *
Input : $v \in A$: value of type A to be converted into Rational
Output : rational number
2 conversion of v into Rational : $v = \frac{a}{b}$
3 this.n = $\big($this.n $\times$ a $\big)$
4 this.d = $\big($this.d $\times$ b $\big)$
5 **return** $\frac{this.n}{this.d}$

2.2 Surcharge des opérateurs -> templates

Au départ, nous avons surchargé les opérateurs de façon à pouvoir uniquement faire des opérations entre deux rationnels. Après avoir codé la suite $u_{n+1} = 4u_n - 1$, nous nous sommes rendus compte qu'il fallait qu'on trouve une solution plus efficace pour réaliser des opérations entre un rationnel et d'autres types de variables.

Au début, nous avons surchargé plusieurs fois un même opérateur comme l'opérateur + (Rational : :operator+(Rational r), Rational : :operator+(int i), Rational : :operator+(float f)) par exemple. Néanmoins, cette solution n'était pas idéale et nous déçidâmes de passer nos opérateurs en template pour rendre notre code plus propre.

Pour cela, nous réalisâmes un constructeur qui prend en entrée un type A (integrals, nombre à virgule flottant, rationnel) et qui retourne un rationnel (voir **algorithme 4**).

De cette façon, une formule comme :

Rational(4,1)*Rational(a,b)- toRational(3.5,10)

devient :

4*Rational(a,b)-3.5

Impressionnant n'est-ce pas ? ;)

3 Conversion d'un réel en rationnel

3.1 Gestion des nombres réels négatifs

Nous avons modifié l'algorithme de conversion d'un réel en rationnel proposé dans l'énoncé pour qu'il gère également les nombres réels négatifs. La tâche n'était pas aisée néanmoins nous parvînmes à un résultat convenable.

Pour ce faire, nous codâmes une fonction *sign*() (disponible dans internal.hpp), qui renvoie le signe (1 ou -1) de la valeur passée en argument. Nous n'étions alors pas peu fiers de cette dernière.

Dans l'algorithme de la fonction *toRational(A value, int nbIter)*, la valeur passée en argument (v) est convertie en valeur absolue (x). Les comparaisons $x > 1$ et $x < 1$ sont donc réalisées avec une valeur positive, puis chaque valeur est ensuite multipliée par le signe de v ($sign(v)$) à chaque itération (cf **algorithme 2**). Stupéfiant !

3.2 Utilisation des templates dans l'algorithme de conversion

Nous avons également amélioré l'algorithme de conversion afin qu'il puisse prendre en paramètre une variable de type Rational, Integral (int, long int...) et Floating-Point (float, double...). Ces types peuvent alors être convertis en rationnel, ce qui facilite grandement certains calculs.

Attention : ne pas mettre n'importe quoi dans l'algorithme de conversion comme un Bateau bateau() par exemple, cela n'aurait pas de sens ahah !

Attention(bis) : certains types comme bool et char sont considérés comme des integrals. Ceci faisant, on peut créer un Rational('a','b'), n et d seront alors les valeurs ASCII de a et b. C'est à l'utilisateur de ne pas rentrer n'importe quoi !

Algorithme 2 : convert any type into rational

1 function toRational
  Input : $v \in A$: value of type A to be converted into Rational
  nbiter $\in \mathbb{N}$: the number of remaining recursive calls
  Output : rational number
2 **if** v is a rational **then** return v
3 **if** v is an integral **then** return $\frac{v}{1}$
4 **if** v is a floating point **then**
5 x = $|v|$
6 **if** $x = 0$ or $nbiter = 0$ **then**
7 return $\frac{0}{1}$
8 **if** $x < 1$ **then**
9 return $\left(toRational\left((sign(v)\times \frac{1}{x}\right),\text{nbiter}\right)\right)^{-1}$
10 q = $\lfloor x \rfloor$
11 diff = x - q
12 **if** diff < 1e-10 **then** diff = 0
13 return $\left(sign(v)\times \frac{q}{1} + toRationnal\left(sign(v)\times\text{diff},\text{nbiter - 1}\right)\right)$
14 **else** throw "bad argument"

4 Représentation des grands nombres

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 ca ?

Généralement, les grands nombres se représentent mal en informatique. En effet, tous les types ont des bornes et ne peuvent stocker qu'un intervalle défini de valeurs. De plus, les nombres à virgule flottante, n'autorisent qu'un nombre fini de décimal.

Prenons l'exemple des int. La valeur maximale d'un int codé sur 32 bit est 2 147 483 647. Pour simplifier et pour illustrer nos propos, supposons que la valeur maximale vaille : 2 000 000 000.

Prenons le réel 2 000 000.01. Représenter ce nombre sous forme rationnelle reviendrait à écrire $\frac{2\ 000\ 000\ 001}{100}$. Le numérateur dépassant la limite numérique, ce nombre ne peut pas être représenté sous forme rationnelle.

Ainsi, pour un nombre réel déjà grand, le numérateur ou le dénominateur sera encore plus grand en écriture rationnelle et ceci est assez problématique :/

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

Pour palier au problème de la représentation des grands nombres en rationnels, plusieurs pistes ont été évoquées. Dans cette partie, nous allons tâcher de retranscrire notre processus de réflexion et pourquoi nous avons finalement décidé d'implémenter ou non ces solutions :

- **Passer la classe en template** : Le fait de passer la classe en template nous permet déjà d'avoir possiblement des long (long) int ou des unsigned long (long) int au numérateur et au dénominateur et donc d'avoir des valeurs plus grandes. (cf **algorithme 3**)

```
Rational<int>(2 147 483 647,1) = 2147483647/1
Rational<int>(3 147 483 647,1) = -1147483649/1
Rational<long int>(3 147 483 647,1) = 3147483647/1
```

FIGURE 2 – Les templates c'est trop cool

Comme vous pouvez le constater, Rational(3 147 483 647,1) devrait en théorie renvoyer $\frac{3\ 147\ 483\ 647}{1}$. Or ce rationnel est de type int, donc la limite est très largement dépassée au numérateur et renvoie n'importe quoi. Définir ce rationnel de type long int permet de palier à ce problème. La classe reste néanmoins sujette à l'overflow (débordement(fait de déborder)). Nous n'avons pas trouvé de moyen pour tester si la limite était atteinte, et si oui changer le type T.

- **Utiliser des librairies supplémentaires** : nous pouvons utiliser de grands nombres en utilisant la bibliothèque boost et le type de données "grand entier". Différents types de données comme int128_t, int256_t, int1024_t etc peuvent être utilisés. Grâce à cette bibliothèque, nous pouvons facilement obtenir une très grande précision. Pourquoi ne pas l'avoir implémenté nous direz-vous ?
Selon nous, cette solution revient finalement à repousser le problème, puisqu'il y aura toujours débordement à partir d'une certaine valeur.

- **Stocker des grands int dans un vector<int>** : L'utilisation de vector<int> est une idée que nous avons envisagée pour la représentater des grands numérateurs ou dénominateurs. La classe Rational aurait toujours les attributs numérateur et le dénominateur, mais ils seraient de type vector<int>. Dans un système codé sur 32 bits, peut contenir 2^{30} valeur de int, chaque int pouvant être un grand nombre. Il serait ainsi beaucoup plus facile de définir et stocker des nombres beaucoup plus grands. Ceci étant dit, implémenter cette solution signifierait revoir tous nos opérateurs. Ou alors, il aurait fallu coder une autre classe, permettant de créer des grands nombres. Cette classe contiendrait entre autre un attribut de type vector<int> (chaque int étant un nombre plus ou moins grand, (ou un seul chiffre ?)) ainsi que des méthodes permettant de faire les opérations usuelles. Cette solution semblerait être la plus pertinente, mais au vu de la complexité d'un tel programme et surtout par manque de temps, nous avons préféré passer notre chemin.

- **Int + nombre rationnel :** représenter un grand nombre comme un int + un nombre rationnel permet d'avoir un numérateur moins grand.
Exemple :

$$2\ 000\ 000.01 = \frac{200\ 000\ 001}{100} = 2\ 000\ 000 + \frac{1}{100}$$

Dans ce cas, la classe Rational aurait un attribut partie entière $intPart$ qu'il faudrait considérer à chaque opérations. C'est possible à réaliser mais il aurait alors fallu recommencer tout le projet :(
De plus, cette solution ne permet pas de "soulager" le dénominateur, donc dans le cas d'un nombre rationnel avec un dénominateur très grand, cette solution reste inutile.

- **Arrondir** : Une solution que nous avons envisagée serait d'arrondir les nombres trop grands, notamment les nombres avec une trop grande partie entière.
Exemple :

$$2\ 000\ 000.01 \approx 2\ 000\ 000 = \frac{200\ 000\ 000}{1}$$

Ici, on ne dépasse plus la limite ! Ainsi, au moment de la conversion d'un nombre décimal en rationnel par exemple, il faudrait arrondir le nombre de décimal. Cependant, à partir de quand la partie entière est considérée comme trop grande pour arrondir la partie décimale ? Combien de décimales enlever ?
Le principe d'utiliser la classe Rational est justement de ne pas avoir à arrondir les nombres rationnels et garder une grande précision au moment des calculs. Ainsi, il ne nous a pas semblé pertinent d'implémenter cette solution.

5 Pseudo codes des constructeurs

En plus d'un default et d'un copy constructor, nous avons codé 2 autres constructeurs :

1) Constructeur à partir d'un numérateur n et un dénominateur d, tous les deux de type T :
Avant toutes choses, le constructeur vérifie avec un static assert que le type T est bien un integral et pas n'importe quoi. Il renvoie ensuite $\frac{n}{d}$, fraction irreductible.

Dans le cas où d négatif, les attributs n et d sont multipliés par le signe de d grâce à notre super fonction sign. Comme ça, d est toujours positif :D

Que se passe-t-il quand d = 0 ? Deux cas de figures se présentent :
- Soit le numérateur est également nul auquel cas il nous a semblé plus pertinent de déclencher une exception (ça ne veut rien dire de faire des calculs avec $(0/0)$? ? ? ?)
- Soit le numérateur est un entier positif ou négatif et le rationnel est alors respectivement égal à $\frac{1}{0}$ ou -$\frac{1}{0}$ (équivalent de $+\infty$ ou $-\infty$). Finalement, nous n'interdisons pas un dénominateur égal à 0, et tous les calculs peuvent se réaliser avec ce dénominateur nul. Prenons deux exemples :

$$\frac{14}{5} \times \frac{1}{0} = \frac{14}{0} = \frac{1}{0}$$
$$\frac{14}{5} - \frac{1}{0} = \frac{14 \times 0 + 5 \times -1}{0 \times 5} = - \frac{5}{0} = - \frac{1}{0}$$

Il est intéressant de noter que lorsque l'on défini un Rational avec un dénominateur nul, par exemple Rational(n,0) on a pgcd(n,0) = 20 donc dans notre algorithme n = 20/20 = 1. Ainsi, un rationnel avec un d = 0 sera toujours égal à $\frac{1}{0}$. Nous avons ainsi décidé de modifier notre surcharge de l'opérateur « de std : :ostream pour qu'il affiche +inf et -inf pour respectivement 1/0 et -1/0.

Algorithme 3 : constructor of type T Rational from 2 values (numerator and denominator

```
1  constructor
   Input  : n ∈ T : numerator of type T
            d ∈ T : denominator of type T
   Output : rational number
2  this.n = n
3  this.d = d
4  Precondition : type T = integral
5  //case 0/0
6  if this.n = 0 and this.d = 0 then  throw "error : bad argument"
7
8  gcd = pgcd(n/d)
9  if gcd ≠ 1 then
10     this.n = this.n / gcd
11     this.d = this.d / gcd
12 //case this.d<0
13 this.n=this.n ×sign(d)
14 this.d=this.d ×sign(d)
```

2) Constructeur qui prend un type A et le converti en rationnel :

Tout d'abord, *value* est convertie en rationnel, grâce à notre super fonction *toRational*(). Ensuite, la ligne **if** $|toFloat(r) - value| > 0.0001$) permet de voir si la conversion s'est bien passée, puisque comme nous allons le voir dans la partie suivante, tout ne se passe pas toujours comme prévu... (suspens)

Algorithme 4 : constructor of type T Rational from a type A value

```
1  constructor
   Input  : value ∈ A : value of type A to be converted into Rational
   Output : rational number
2  Precondition : type T = integral
3  new Rational<T> r = toRational(value)
4  //check if the conversion failed
5  if |toFloat(r) − value| > 0.0001) throw "Conversion failed"
6  this.n = r.n
7  this.d = r.d
```

6 Problèmes rencontrés

Tout semble bien beau d'apparence, mais quelle ne fut pas notre surprise au moment de tester notre algorithme sur le float 150,2.
En utilisant l'algorithme de conversion toRational à la main, seulement 3 itérations suffisent pour trouver que 150,2 = $\frac{751}{5}$

Reprenons ces 3 étapes (cf. **algorithme 2**) :

La valeur de v est positive ici, donc durant tout le programme, v = $|x| = x$

$x = 150.2$

Itération 1 : $x > 1$
q = $\lfloor 150.2 \rfloor = 150$
x - q = 0.2
x devient 0.2

Itération 2 : $x < 1$
$x = 5$
q = 5
x - q = 0

Itération 3 : $x = 0$
$x = 0$

On obtient à la fin $\frac{150}{1} + \frac{1}{5} = \frac{751}{5}$

Pourtant en utilisant notre algorithme toRational, ce n'est pas ce qu'on obtient. Lors de la création de Rational<long int>(150.2), le constructeur prend en charge le float 150.2, pour le convertir en rationnel grâce à la fonction toRational. Seulement voilà, la valeur renvoyée est $\frac{8930972583259534213}{5659785159111122185}$, ça fait mal aux yeux.

Après plusieurs jours d'investigations (et quelques larmes), nous nous sommes rendus compte que l'algorithme ne fonctionnait plus à partir de la 3ème itération :

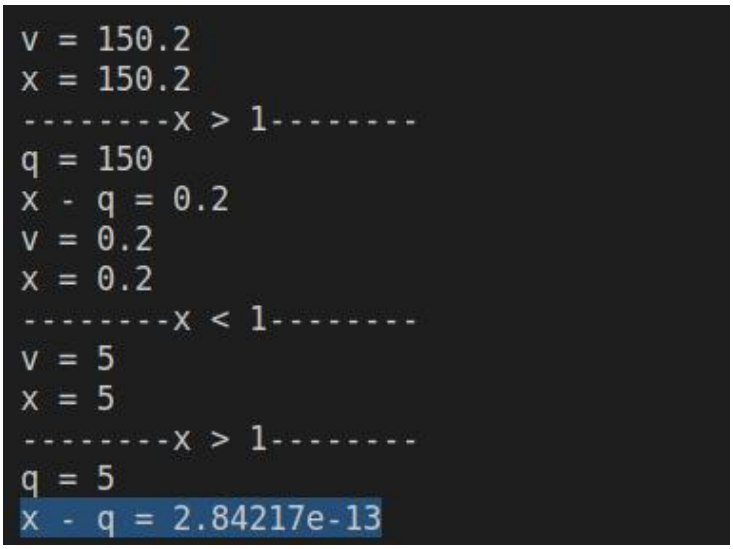

FIGURE 3 – Que se passe-t-il ? ? ?

Lors de cette itération, x - q vaut 2.84217e-13, au lieu de 0. En effet, x et q sont des nombres à virgules flottantes, qui utilisent un nombre fini de bits. Ils ne peuvent donc qu'approximer les nombres réels. Par conséquent, la différence entre ces deux nombres est égale à un nombre très proche de 0 (mais pas vraiment 0). La suite des calculs est donc complètement eronnée, les erreurs s'accumulent et on obtient des valeurs titanesques au numérateur et au dénominateur. N'est-ce pas le comble pour un algorithme visant à la base à prévenir ce genre d'erreurs ?

Pour palier à ce problème, nous avons rajouté une ligne à notre fonction *toRational*(), qui teste si la différence entre x - q est inférieure à 1e-10. Si cette condition est vérifiée, alors le terme x - q devient nul.