— Rapport de projet —

Programmation objet
Mathématiques pour l'informatique
IMAC 2

Guy Luong **Aude Pora**

I- Programmation

1) Tableaux bilan

Élements demandés et codés qui fonctionnent	Élements non demandés et codés qui fonctionnent	Élements non demandés et codés qui fonctionnent à moitié	Élements non demandés mais pas codés / qui ne fonctionnent pas
Somme de deux rationnels	Opérateurs : - (soustraction de deux rationnels), / (division de deux rationnels)	pow (voir I-3-d)	Opérateurs d'assignation
Produit de deux rationnels	Fonction qui retourne la partie entière.	cos (voir I-3-d)	
Inverse d'un rationnel	Fonction qui arrondit un nombre rationnel.	sqrt (voir I-3-d)	
Fonction pour transformer le rationnel sous forme de fraction irréductible	Variadics (max, min, etc.)	exp (voir I-3-d)	
Moins unaire	Lambdas dans les variadics	Conversion d'un réel en rationnel (voir I-3-d)	
Valeur absolue			
Partie entière			
Produit d'un nombre en virgule flottante avec un rationnel			
Produit d'un rationnel avec un nombre en virgule flottante			
Opérateurs de comparaison ==, !=, >, <, <= et ≥			
Surcharge de l'opérateur <<			

Élements demandés mais pas codés
-

Élements demandés et codés qui ne fonctionnent pas
-

2) Développement

Nous avons opté pour une **méthode de développement définie par les tests unitaires** pour pouvoir visualiser l'avancée et donner une idée claire des fonctions et méthodes à implémenter.

Nous avons donc créé les tests unitaires en amont, ce qui a plusieurs effets :

- **Fixer comment fonctionnent les classes**, au niveau de la syntaxe, des signatures des méthodes ;
- **Connaître les résultats attendus** ce qui permet de chercher des cas problématiques afin de pouvoir les traiter.

Nous avons choisi une **implémentation simple des nombres rationnels**, et non celle proposée résolvant le problème des grands nombres, pour pouvoir rendre l'accès au développement des méthodes et de la compréhension des bases du C++ plus aisé.

Nous avons rendu la classe **immutable**, un des principes recommandés par la programmation objet, en renvoyant un nouveau nombre rationnel à chaque opération. Il existe des opérateurs d'assignation : `-=`, `+=`, `*=` (avec un rationnel et un scalaire), `/=`. Nous ne les avons pas implémentés parce qu'ils brisent l'immutabilité de la classe.

3) Problèmes rencontrés et solutions implémentées (si trouvées)

a) Simplification fraction

Nous avons rencontré des problèmes étranges pour le **test de simplification** de la fraction qui vaut $-\infty$.

L'exemple ci-dessous montre ce qu'il se passe pendant la simplification de la fraction $\frac{-1}{0}$.

```
std::cout << "gcd(-1, 0) = " << std::gcd(m_numerator, m_denominator) << std::endl;
std::cout << "-1 / gcd(-1, 0) = " << -1 / std::gcd(m_numerator, m_denominator) << std::endl;
std::cout << "-1 / 1 = " << -1 / 1 << std::endl;
```

```
gcd(-1, 0) = 1
-1 / gcd(-1, 0) = 4294967295
-1 / 1 = -1
```

Ce problème a été résolu en convertissant le retour de `std::gcd` en entier (`int`).

b) Static assert et tests unitaires

Nous nous sommes rendus compte que nous ne pouvions pas tester les `static_assert` avec les tests unitaires puisque les `static_assert` s'effectuent avant l'exécution des tests : à la compilation.

On souhaitait en utiliser pour **tester des types** (avec `std::is_floating_point()`) pour lever des exceptions.

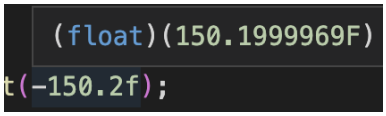
c) Problème lié à std::gcd

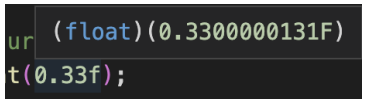
Lorsque Ratio est utilisé avec un flottant au numérateur, il plante automatiquement à la compilation avec un très long paragraphe d'erreur, parce que cette classe utilise `std::gcd` et que cette fonction ne fonctionne pas avec les flottants. Puisque la classe n'est pas censée fonctionner avec des flottants, elle fonctionne comme prévu avec ce crash, bien qu'il serait préférable d'avoir une ligne d'erreur plus propre pour cela.

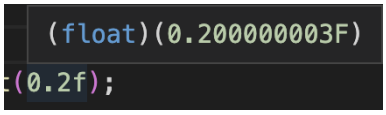
d) Nombres non représentables exactement en float

Certains nombres ne sont pas représentables exactement en flottant.

Exemples :







Cela mène à penser que la conversion en nombre rationnel ne fonctionnera pas, dûe à l'erreur de précision qui explose au fil des répétitions de la fonction `convertFromFloat`. C'est bien un cas qui arrive, notamment avec le nombre `-150.2` ci-dessus, mais pas avec `-150.1`, un nombre proche.

Conditions de test :
La fonction `convertFromFloat` est programmée pour itérer par défaut au maximum 10 fois, nombre atteint dans les deux cas. Le seuil pour passer le test est fixé à 10^{-10}.

Nombre	Nombre d'itérations	Seuil	Différence	Test
`-150.2`	10	10^{-10}	146	Non passé
`-150.1`	10	10^{-10}		Passé
	15	10^{-10}	172	Non passé

On remarque donc que l'erreur numérique explose pour `-150.2` et non pour `-150.1`.
En itérant cependant 15 fois pour `-150.1`, le test ne passe plus. On en déduit que l'erreur explose plus vite pour `-150.2` que pour `150.1`.

> 💡 **Note :** Les deux autres valeurs données en exemple, `0.2` et `0.33` convergent et le nombre d'itération est fixe (inférieur à 10).

`0.2` donne la valeur exacte et `0.33` converge à 10^{-9} près. `0.33` converge en tombant sur un entier exact lors du déroulement de l'algorithme, et la valeur renvoyée est `0.33000000305175503`, légèrement plus précis qu'un `float` représentant `0.33` mais pas aussi précis qu'un `double`.

Autre test avec `0.3333` :

Nombre d'itérations	Seuil	Différence	Test
10	10^{-7}	$\sim 10^{-8}$	Passé
20	10^{-7}	1	Non passé

> 💡 **Note :** L'implémentation de `cos`, `sqrt`, `exp`, etc. dépendent de cette fonction, et pour cette raison, il existe ~~beaucoup~~ de valeurs pour lesquelles le résultat n'est pas celui attendu.

4) Détails de nos structures de données

Pour structurer notre classe Ratio, nous avons tout d'abord créé deux champs pour représenter le numérateur et le dénominateur.

Il est important de noter que le type du numérateur est générique (`template`), mais est forcé d'être un type entier.

Le dénominateur est forcé d'être positif, mais est gardé signé dans le champ, comme le constructeur permet d'utiliser des entiers négatifs pour le dénominateur et corrige le signe pour le mettre sur le numérateur dans ce cas.

$$r = \frac{a}{b},\ avec\, b > 0$$

Des opérations peuvent s'effectuer à la compilation grâce au mot-clé `constexpr`. Elles le sont généralement toutes, sauf celles demandant d'utiliser des `if`, telles que `convertFromFloat` et celles qui dépendent de cette méthode.

II- Mathématiques

1) Opérations sur les rationnels

Comment formaliser l'opérateur de division / ?

Pour formaliser l'opérateur de division il suffit de faire comme suit : $\frac{\frac{a}{b}}{\frac{c}{d}} = \frac{a}{b} * (\frac{c}{d})^{-1} = \frac{ad}{bc}$

Lors d'une division par zéro, nous pouvons soit lancer une exception (`division par zéro`), soit utiliser les limites ($\pm\infty$). Ici, nous avons choisi d'utiliser les limites.

2) Conversion d'un réel en rationnel

Comment modifier l'algorithme ci-après pour qu'il gère également les nombres réels négatifs ?

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)
```

Analyse de la ligne 8 : On appelle la fonction `convert_float_to_ratio` en utilisant l'inverse d'un nombre plus petit que 1. Pendant cet appel, on passera dans le bloc $x > 1$ où on ira extraire la partie entière. La puissance -1 s'applique sur le retour de la fonction : un nombre rationnel.

Analyse de la ligne 12 : La somme utilisée est la somme entre nombre rationnels, pour pouvoir additionner q (la partie entière, sous forme d'un rationnel) au reste, qui est également un rationnel.

Pour que l'algorithme gère également les nombres réels négatifs, nous pouvons l'utiliser avec la valeur absolue de x et on ajoutera le signe à la fin.

Pour l'implémentation, nous avons plutôt pensé à faire comme suit. On réutilise cet algorithme en le renommant `convert_float_to_ratio_aux` et en l'appelant dans une autre fonction qui prend en compte le signe.

Algorithm 2 Conversion d'un réel quelconque en rationnel

```
    function CONVERT_FLOAT_TO_RATIO(r, nb_iter)                ▷ r ∈ IR
2:      p ← valeur_absolue(r)
        if r < 0 then
4:          return -convert_float_to_ratio_aux(p, nb_iter)
        else
6:          return convert_float_to_ratio_aux(p, nb_iter)
```

3) Analyse

> **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. Comment expliquer cela ?**

Il y a deux principaux problèmes avec notre représentation de nombres rationnels :

- les grands nombres entiers se représentent facilement alors que les grands nombres à virgules non car ils font **exploser la limite du numérateur**.
  - L'ensemble des nombres entiers est entièrement représentable.
    $n \mapsto \frac{n}{1}, \forall n \in [-n_{max}, n_{max}]$,
    avec n_{max} le plus grand entier représentable avec un certain type d'entier.
  - Dès qu'on passe chez les nombres décimaux, le nombre le plus grand représentable est inversement proportionnel au dénominateur.
    $\frac{n}{2} \in [\frac{-n_{max}}{2}, \frac{n_{max}}{2}]$, $\frac{n}{3} \in [\frac{-n_{max}}{3}, \frac{n_{max}}{3}]$, etc.
    On remarque vite qu'il est impossible de représenter des grands nombres avec un grand dénominateur, ou une grande partie décimale.

    Exemple :
    $(n_{max} - \frac{1}{2})$ ne peut donc pas être représenté, en supposant n_{max} suffisamment grand (> 2), comme $n_{max} - \frac{1}{2} = \frac{2n_{max}-1}{2} > \frac{n_{max}}{2}$.
    On a donc n_{max} représentable et $n_{max} - \frac{1}{2}$ non représentable alors que $n_{max} > n_{max} - \frac{1}{2}$.

- il y a le même problème que pour les nombres flottants : **la représentation des nombres n'est pas uniformément répartie sur l'ensemble de définition des rationnels.** L'écart entre deux nombres proches de 0 successifs est bien plus petit que celui entre deux nombres très grands.

> **Lorsque les opérations entre rationnels s'enchaînent, le numérateur et le dénominateur peuvent prendre des valeurs très grandes, voire dépasser la limire de représentation des entiers en C++. Quelles sont les solutions possibles ?**

On peut décider de représenter les nombres rationnels sous une autre forme.
Cette forme décompose le nombre rationnel $\frac{a}{b}$ en :

- Une partie entière n : $n = \lfloor \frac{a}{b} \rfloor$
- Un nombre rationnel $\frac{a_0}{b_0}$ tel que $\frac{a_0}{b_0} \in [0, 1[$
- $\frac{a}{b} = n + \frac{a_0}{b_0}$

Cette forme possède :

- l'uniformité des entiers sur $\mathbb{Z}$
- la précision qu'ont les nombres rationnels sur $[0, 1[$, répartie sur l'ensemble de définition.

Avec cette définition, on redéfinit ainsi les opérateurs ci-dessus.

Exemple : addition :

$$(n_0 + \tfrac{a_0}{b_0}) + (n_1 + \tfrac{a_1}{b_1}) = \underbrace{(n_0 + n_1 + \lfloor \frac{a_0 b_1 + a_1 b_0}{b_0 b_1} \rfloor)}_{entier} + \underbrace{(\frac{a_0 b_1 + a_1 b_0}{b_0 b_1} - \lfloor \frac{a_0 b_1 + a_1 b_0}{b_0 b_1} \rfloor)}_{rationnel}$$

Ici, on fait l'addition standard entre nombres rationnels, mais on doit extraire la partie entière issue de $\frac{a_0}{b_0} + \frac{a_1}{b_1}$ comme une retenue dans une addition posée à la main, puis le reste constitue la partie rationnelle.

Les autres opérations seront définies de la même façon.