Trigger Ratio Library

Wendy Gervais
Tristan Debeaune

IMAC S3

Contents

1 Introduction

1.1 Éléments fonctionnels

ÉLÉMENT	DEMANDÉ	CODÉ	GTESTS
Rapport LaTeX	Oui	Oui	
Classe Ratio, getter, setter	Oui	Oui	
Constructeurs	Oui	Oui	RatioConstructors
Destructeur	Oui	Oui	
Opérateurs de base (+, -, x ...)	Oui	Oui	RatioOperators
Opérateurs mathématiques (cos ...)	Oui	Oui	RatioMaths
Conversion Flottant en Rationnel	Oui	Oui	
Conversion Flottant Négatif en Rationnel	Oui	Oui	
Réduction fractions	Oui	Oui	
Algorithme d'Euclide	Non	Oui	
Opérateurs d'assignation (+=, *=...)	Non	Oui	
Fonction variadique (Opérateur moyenne)	Non	Oui	

1.2 Liens

- Dépôt Git : `https://gitlab.com/rubykiara1712/trigger-ratio`
- Documentation Doxygen : `https://etudiant.u-pem.fr/~wendy.gervais/ratio/`

2 Construction des rationnels

2.1 Classe Ratio

Notre classe de rationnels est la classe Ratio, qui contient deux attributs : m_num (numérateur) et m_den (dénominateur). Ceux-ci sont de même type entier (int, long int...), type indiqué par l'utilisateur à la création du Ratio, car la classe est écrite en template.

2.2 Constructeurs

Le type T se substitue aux types (unsigned) int, long int, long long int.

NOM	ARG.1	ARG.2	RES	Test RatioConstructors
Default			0/1	.default
Paramétré	T a		a/1	.p1
Paramétré	T a	T b	a/b	.p2
Copie	Ratio			.copy

Nous avons fait le choix d'inclure toutes les propriétés élémentaires des fractions irréductibles et de la positivité du dénominateur d'un rationnel dans le constructeur paramétré de Ratio : les constructeurs du numérateur et dénominateur sont appelés sur ceux-ci, divisés par leur pgcd. Ainsi, dès sa construction ou après toute opération (qui appellera toujours ce constructeur), une fraction est et restera sous forme irréductible. Cela évite que les numérateurs et dénominateurs ne prennent des valeurs trop grandes qu'il ne serait alors plus possible de représenter avec des entiers en C++.

```
Ratio(const U num, const V den):
    m_num(num/hcd(num, den)),
    m_den(den/hcd(num, den))
```

où hcd(num, den) désigne le pgcd de num et den. Nous avons codé celle-ci dans la librairie en utilisant l'algorithme d'Euclide récursif :

```
fonction hcd(a,b) :
    si b = 0 :
        retourner |a|
    sinon :
        retourner |hcd(b, a modulo b))|
```

De plus, le dénominateur d'un Ratio est toujours positif. Si le constructeur est appelé avec un dénominateur négatif, on considère que l'on prend l'opposé du numérateur, et donc du dénominateur, qui devient positif.

```
Si den<0 :
    m_num = - num
    m_den = - den
```

2.3 Représentation de 0

Nous distinguons l'infiniment petit positif (0^+) et négatif (0^-). 0^+ est représenté par le Ratio 0/1 et 0^- est le seul Ratio à avoir un dénominateur négatif puisqu'il est représenté par 0/-1 (car 0 ne peut pas porter le signe -).

Il n'est pas possible de construire un Ratio ayant 0 pour numérateur et ayant un autre dénominateur que 1 ou -1 : le constructeur change automatiquement le dénominateur en un 1 portant le signe du dénominateur entré ou calculé, et ce grâce à la fonction std::signbit. Celle-ci renvoie 0 si l'entier est négatif et 1 s'il est positif, nous effectuons l'opération suivante pour récupérer -1 ou +1 où "den inférieur à 0" est renvoyé par std::signbit :

```
Si num==0 :
    m_den = (den<=0) * 2 - 1
```

2.4 Représentation de l'infini

L'infini est représenté par le Ratio 1/0 pour $+\infty$ (ou -1/0 pour $-\infty$). Tout comme pour le 0 avec son dénominateur, le numérateur ne peut être que 1 ou -1. L'infini reste égal à 1/0 ou -1/0 pour les calculs ; ainsi, toute multiplication ou addition conservera l'infini, toute division donnera 0 en mettant 0 au numérateur. Les comparaisons fonctionnent aussi.

En revanche, ces Ratios seront affichés en tant que +inf ou -inf respectivement par l'opérateur std::cout (et non 1/0 ou -1/0).

Par extension de la propriété de calculs, il est possible d'inverser l'infini, cela donnera 0/1 pour $+\infty$ et 0/-1 pour $-\infty$: on garde ainsi la propriété du signe.

2.5 Exceptions

Comme en mathématiques, il n'est pas possible de construire le Ratio 0/0, cela arrêtera le programme.

Le programme lèvera aussi une exception si l'utilisateur chercher à construire un Ratio à partir d'autre choses que des entiers (par exemple, des flottants : il faudra utiliser la fonction de conversion pour cela).

Des exceptions sont implémentées dans les opérateurs mathématiques si l'utilisateur cherche à les utiliser hors de leur domaine de définition (voir en section 3.4).

2.6 Conversion des flottants en rationnels

`convertPosFloatToRatio()`

C'est la fonction de base donnée dans le sujet qui permet la conversion pour un nombre flottant positif. La première condition d'arrêt nécessite un égalité à 0 et donc une précision parfaite ; dans la pratique cela n'est pas possible, donc nous avons utilisé un seuil de 10^{-10}.

`convertFloatToRatio()`

Ceci est la fonction finale qui appelle la fonction première. et se présente comme ceci, avec f un flottant :

```
fonction convertFloatToRatio(f):
    sign = (f<=0) * 2 - 1
    retourner convertPosFloatToRatio(sign*f)*f
```

"f inférieur à 0" est toujours renvoyé par std::signbit. sign vaut 1 si f est positif et -1 sinon, ainsi sign*f = abs(f), on peut convertir et remultiplier ensuite par sign pour conserver le signe.

3 Détail des opérateurs

3.1 Opérateurs

Ce sont des méthodes déclarées à l'intérieur de la classe Ratio (fichier Ratio.hpp).

NOM	SYMBOLE	OBJ.	ARG.2	Test RatioOperators
Addition	+	Ratio	Ratio	.adds
Addition	+	Ratio	Int	.Sadds
Soustraction	-	Ratio	Ratio	.subs
Soustraction	-	Ratio	Int	.Ssubs
Moins unaire	-	Ratio		.minus
Produit	*	Ratio	Ratio	.times
Produit	*	Ratio	Int	.Stimes
Division	/	Ratio	Ratio	.div
Division	/	Ratio	Int	.Sdiv

Nous avons formalisé l'opérateur de division entre Ratios comme la multiplication, c'est-à-dire que a/b divisé par c/d construit directement le Ratio :

$$\frac{a*d}{b*c}$$

Afin de traduire la possibilité de "multiplier en haut" ou "multiplier en bas" (ie, diviser "en haut"), nous avons aussi surchargé l'opérateur * et l'opérateur / avec un un argument entier. Il sera ainsi possible de "multiplier un Ratio par a", où a serait un int et non pas un Ratio. On peut aussi directement ajouter ou retrancher un entier à un Ratio.

On ajoute à cette liste l'opérateur std::cout (double chevrons) pour afficher les rationnels sous la forme "a/b". Celui-ci prend en charge l'affichage de l'infini.

3.2 Opérateurs d'assignation

Ce sont des méthodes déclarées à l'intérieur de la classe Ratio.

NOM	SYMBOLE	OBJ.	ARG.2
Addition	+=	Ratio	Ratio
Addition	+=	Ratio	Int
Soustraction	-=	Ratio	Ratio
Soustraction	-=	Ratio	Int
Produit	*=	Ratio	Ratio
Produit	*=	Ratio	Int

Comme pour les opérateurs simples, nous avons choisi de surcharger l'addition, la soustraction et le produit en argument Ratio ET en argument entier, dans l'optique d'une écriture rapide dans des boucles ou des tests ; il sera ainsi plus aisé d'ajouter 5 sans l'exprimer en tant que 5/1 à un Ratio R en utilisant "R+=5" (plutôt que R = R+Ratio R2(5) par exemple).

3.3 Opérateurs de comparaison

Ce sont des méthodes déclarées à l'intérieur de la classe Ratio, sauf max et min, qui sont déclarées à l'extérieur de celle-ci, dans le fichier operatorsMath.hpp.

NOM	SYMBOLE	ARG.1	ARG.2	GTEST RatioOperators
Égalité	==	Ratio	Ratio	
Différent	!=	Ratio	Ratio	
Supérieur	>=	Ratio	Ratio	
Inférieur	<=	Ratio	Ratio	
Supérieur s.	>	Ratio	Ratio	
Inférieur s.	<	Ratio	Ratio	
Maximum	max()	Ratio	Ratio	.max
Minimum	min()	Ratio	Ratio	.min

Pour éviter des erreurs d'arrondi (par exemple, essayer de comparer 0,5555 avec 0,5556) nous avons choisi de comparer les numérateurs des deux rationnels, une fois ceux-ci mis "au même dénominateur". Par exemple, pour comparer a=0,55=$\frac{5}{9}$ et b=0,56=$\frac{9}{16}$, on "multiplie en haut et en bas par le dénominateur de l'autre" ; on comparera en réalité :

$$\frac{5}{9} = \frac{5*16}{9*16} = \frac{80}{144} \text{ et } \frac{9}{16} = \frac{9*9}{16*9} = \frac{81}{144}$$

c'est-à-dire qu'on compare les entiers 80 et 81, ce qui est plus précis.

3.4 Opérateurs mathématiques

Ce sont des fonctions mathématiques déclarées à l'extérieur de la classe, dans le fichier operatorsMath.hpp.

NOM	SYMBOLE	RES	Test RatioMaths	EXCEPTION
Racine carrée	sqrt()	Flottant	.sqrt	< 0
Valeur absolue	abs()	Ratio	.abs	
Exponentiel	exp()	Flottant	.exp	
Logarithme	log()	Flottant	.log	≤ 0
Cosinus	cos()	Flottant	.cos	$\pm\infty$
Arccosinus	acos()	Flottant		$\notin [-1;1]$
Sinus	sin()	Flottant	.sin	$\pm\infty$
Arcsinus	asin()	Flottant		$\notin [-1;1]$
Puissance	pow()	Flottant		
Partie entière (sup)	ceil()	Entier		
Partie entière (inf)	floor()	Entier		

La plupart des opérateurs renvoie un flottant (float ou double en template, par défaut, double).

Au début, nous souhaitions que les opérateurs mathématiques renvoient tous des rationnels, cependant la plupart des résultats devenaient trop grands ou imprécis (exp(20) donnait au-délà de 70milliards que nous n'arrivons pas à exprimer en rationnel), nous laissons donc le choix à l'utilisateur de récupérer le résultat flottant et éventuellement chercher à le convertir en Ratio plus tard, avec la fonction de conversion.

Pour réaliser les calculs en flottant (on utilise les fonctions de math.h expf, cosf...), nous utilisons une fonction convertRatioToFloat, qui renvoie le flottant associé au Ratio.

3.4.1 Opérateur moyenne

Nous avons choisi de réaliser une fonction calculant la moyenne d'un ensemble de Ratio. Celle-ci est variadique. Elle est donc récursive. La moyenne d'un ensemble de n Ratio notés x_1, ... x_n est donnée par :

$$\frac{1}{n} \sum_{i=1}^{n} x_i$$

Que nous avons réécrit et codé récursivement. La fonction average(first, args...), en notant first = x_1, args = $(x_2, x_3 ... x_n)$ et N = sizeof...(args) = $n-1$ retourne :

```
(first + N*average(args))/(N+1)
```

On retrouve bien la formule de la moyenne :

$$\frac{x_1+(n-1)*\frac{1}{n-1}*\sum_{i=2}^{n} x_i}{n} = \frac{x_1+\sum_{i=2}^{n} x_i}{n} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

4 Conclusion

Nous sommes globalement satisfaits du fonctionnement de notre classe de Ratio et ses opérateurs, qui prennent en charge beaucoup de cas d'utilisation, notamment l'∞ et 0 et la plupart des opérateurs flottants, même si on se heurte

rapidement à la limite de représentation des entiers (avec exp qui donne vite plus de 70 milliards, ...).