Projet de programmation : Ratio

Antoine Leblond
Mathurin Rambaud

Table des matières

1 Tableau bilan

Éléments réalisés :

— Classe template
— Classe constexpr
— Polymorphisme
— Utilisation de la STL
— Asserts pour détecter et prévenir les erreurs de programmation
— Exceptions pour traiter les erreurs
— Les différents constructeurs
— Exprimer le nombre 0
— Exprimer l'infini
— Fractions irreductibles
— Dénominateur toujours positif
— Opérations sur les rationnels (avec les nombres à virgule flottante)
— Fonction racine carrée, sinus, cosinus, exponentiel, puissance
— Fonction inverse
— Convertir un réel en rationnel
— Valeur absolue et partie entière
— Opérateurs de comparaison
— Surcharge de l'opérateur «
— Fonctions variadic
— Projet généré avec CMake
— Documentation Doxygen
— Tests unitaires
— Démonstration de la librairie

Éléments non réalisés :

— Fonctions lambda
— Espaces de nommage

2 Partie mathématique

2.1 Formalisation de l'opérateur de division /

Effectuer une division sur un nombre rationnel par un nombre rationnel revient à le multiplier par l'inverse. C'est pour cela que nous utilisons l'opérateur de multiplication * ainsi que la fonction reverse() dans l'opérateur de division /. D'autre part notre professeur nous a indiqué que la multiplication prenait moins de temps à être effectuée que la division, génial ! On obtient alors le résultat suivant :

$$\frac{\frac{a}{b}}{\frac{c}{d}} = \frac{a}{b} \times (\frac{c}{d})^{-1} = \frac{a}{b} \times \frac{d}{c} = \frac{a \times d}{b \times c}$$

2.2 Conversion d'un réel en rationnel

Pour cet algorithme de conversion on nous donnait déjà un pseudo-code mais néanmoins celui-ci n'était pas capable de gérer les nombres négatifs. Pour se faire, il suffit d'appliquer ce pseudo-code sur la valeur absolue du nombre réel sans oublier de rajouter le signe avant chaque appel de la fonction. Pour ce faire nous avons codé une fonction qui prend en paramètre une valeur et qui permet de renvoyer -1 si la valeur est négative ou 1 sinon. Par ailleurs nous avons rajouter des conditions en fonction du type que nous expliquerons un peu plus tard dans le rapport.

On obtient alors l'algorithme suivant :

Algorithm 1: Conversion d'un réel en rationnel

Function `convert_real_to_ratio`

Input: $x \in \mathbb{R}^+$: un nombre réel à convertir en rationnel
nb_iter $\in \mathbb{N}$: le nombre d'appels récursifs restant

if $type(x) == Rational$ **then** return x

if $type(x) == int$ **then** return $\frac{x}{1}$

if $x == 0$ **then** return $\frac{0}{1}$

if $nb\_iter == 0$ **then** return $\frac{0}{1}$

// Appel récursif si $x < 1$

if $x < 1$ **then** return $(\text{convert\_real\_to\_ratio}(\frac{1}{x}, \text{nb\_iter}))^{-1}$

// Appel récursif si $x \geq 1$

if $x \geq 1$ **then** $q = \lfloor x \rfloor$ $return \frac{get\_sign(x) \times q}{1} + convert\_real\_to\_ratio(get\_sign(x) \times (x - q), nb\_iter - 1)$

2.3 Représentation des très grands et très petits nombres

Notre classe a été créée pour supporter des entiers au numérateur ainsi qu'au dénominateur. Il faut savoir qu'un entier est par définition codé sur 32 bits, cela signifie qu'il ne pourra pas dépasser une certaine valeur à savoir 2^{32} pour les entiers non signés et entre -2^{31} et 2^{31} pour les entiers signés. De ce simple fait si une valeur excède ces nombre elle ne pourra être représentée. Une solution a cela serait au lieu d'utiliser des int on pourrait par exemple utiliser des long int ou encore des long long int tout dépend de notre utilisation mais le faire d'avoir rendu notre classe template permet de moduler à ce niveau là.

Après une recherche google "how to represent very large numbers in C++" il semblerait aussi qu'il existe des librairies avec des types capables de stocker des nombres encore plus grands.

Une autre possibilité à laquelle nous avons pensé est de représenter les nombres dans des tableaux où chaque case du tableau représente une puissance de 10 du nombre. Par exemple un tableau de taille 3 aurait sa première case qui représenterait les unités, puis sa deuxième les dizaines et

enfin sa dernière les centaines. On pourrait ainsi utiliser les indices du tableau pour faire les operations entre très grands nombre. On obtiendrait la suite suivante pour une addition :

n : représente les indices du tableau
a : tableau contenant le premier nombre
b : tableau contenant le deuxième nombre qu'on souhaite ajouter au premier

$$a_0 = modulo10(a_0 + b_0)$$
$$a_{n+1} = modulo10(a_{n+1} + b_{n+1} + quotient(a_n + b_n))$$

Cela fait il faudrait ensuite adapter tous les opérateurs pour que ces derniers parcourent l'entièreté des tableaux, on imagine que cette solution peut être très coûteuse mais nous pensons qu'elle n'est pas si compliquée à coder.

3 Partie programmation

3.1 Fonctionnalités réalisées

Grâce à la STL on a pu représenter l'infini notamment grâce à la classe $std :: numeric\_limits$. Ainsi dans nos constructeurs lorsque le dénominateur vaut 0 il nous suffit de renvoyer -inf ou inf selon le signe du numérateur.

Pour ce qui est des constructeurs nous avons en dernier ajouté un constructeur qui prend un réel en paramètre, puisqu'on venait de créer la fonction qui permettait de passer d'un réel à un rationnel on trouvait ça intéressant de faire un constructeur à partir d'un réel.

Nous voulions que les opérateurs puissent être utilisés non pas qu'avec des rationnels mais aussi avec des nombres réels. Ainsi il fallait utiliser la fonction de conversion d'un réel en rationnel dans les opérateurs.

Cependant si on souhaite effectuer une opération entre deux rationnels il est inutile d'utiliser la fonction de conversion, c'est donc pour cela que nous avons rajouter des conditions dans celle-ci qui permet en fonction du type de la valeur qu'on lui donne en paramètre d'effectuer la conversion ou non.

3.2 Problèmes rencontrés

Le réel challenge de ce projet a été de coder la fonction qui permet de convertir un réel en rationnel.

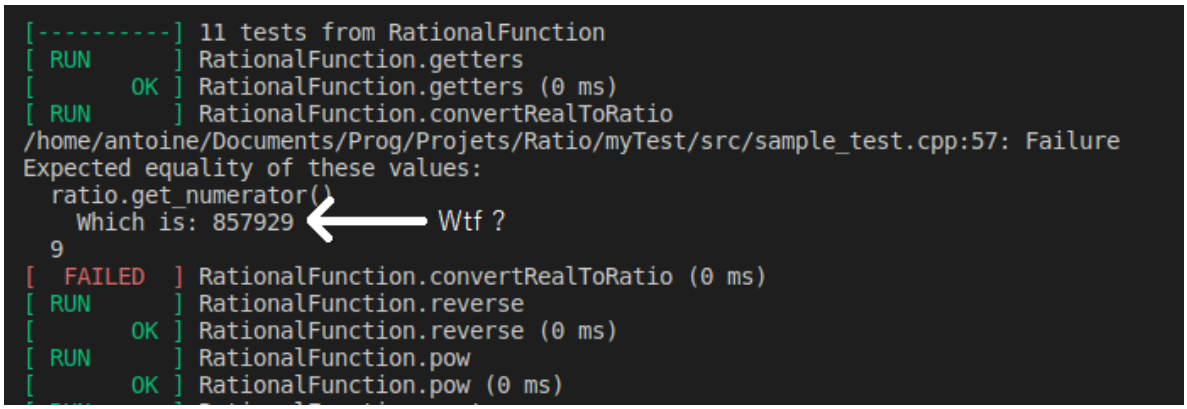
Au commencement de ce projet nous nous sommes cantonnés à faire nos calculs uniquement entre les rationnels. Jusqu'ici tout allait bien mais si notre librairie se limitait à ça elle perdrait un peu de son utilité, alors nous avons décidé de coder cette fonction et de l'utiliser un maximum dans le code (comme dans les constructeurs ou les opérateurs vus précédemment).

À première vue lorsque l'on a observé le pseudo-code qui nous était donné, on pensait que ça allait être une tâche assez facile mais qu'elle fût notre stupéfaction lorsque au numérateur et au dénominateur nous obtenions des nombres astronomiques.
En réalité dans notre fonction nous isolons la partie après la virgule cepen-

dant il peut arriver que celle-ci soit très petite (parfois avec des nombres comme 5.0000001 notre partie après la virgule vallait 0.0000001).

Ainsi lorsque nous faisions l'inverse et qu'on se retrouvait avec 0.0000001 au dénominateur on obtenait des nombres gigantesques qui faisaient littéralement tout foirer...

Il a fallu alors trouver une solution et celle-ci a été de dire que si la partie après la virgule est trop petite (nous avons mis une précision à 10^{-3} par défaut mais c'est modulable) on considère que celle-ci est nulle, ainsi cela règle notre problème et on obtient un résultat satisfaisant.



Enfin dernière chose à noter mais qui n'est pas des moindres, comme vous avec les TPs nous avons voulu s'essayer au LaTeX afin de rédiger notre rapport et bon sang... ça n'a pas été chose facile !