Rapport de projet de programmation objet - mathématiques pour l'informatique

Rémi Poiroux et Loïc Quinquenel

I. Programmation

Bilan

Éléments demandés et codés qui fonctionnent :

- fonction conversion vers rationnel
- fonction inverse (a/b) => (b/a)
- fonction irréductible
- opérations avec un autre ratio +,-,/,* (formules du sujet)
- opérations avec un nombre +,-,/,*
- opération de comparaison entre ratio (>, <, <=, >=, ==, !=)
- surcharge de l'opérateur <<
- moins unaire
- abs (valeur absolue)
- floor (partie entière)

Éléments demandés et codés qui ne fonctionnent pas :

∅

Éléments demandés mais (malheureusement) pas codés :

∅

Éléments non demandés (options) et codés qui fonctionnent :

- Utilisation de templates
- Fonctions en constexpr
- Fonctions sin, cos, tan, exp, log, sqrt, pow (approximation)
- espace de nommage (namespace : rto)
- asserts et exception

Documentation et exemples

La classe est documenté et génère une documentation avec Doxygen (voir readme pour installation)
Il y a aussi un fichier d'exemples dans le répertoire "example" pour comprendre comment utiliser la la classe.

Tests unitaires

Nous utilisons la librairie GoogleTest afin de réaliser les tests unitaires, ces tests nous ont permis tout au long du développement de vérifier si des fonctions déjà codées changeaient de comportement.
Ces tests montrent aussi l'approximation des fonctions sin, cos, tan, log, exp.
Ces tests sont disponibles dans le répertoire "UnitTest".

Pour aller plus loin

La classe est codée en **template** pour que l'utilisateur puisse choisir le type de données du dénominateur et numérateur.
Les fonctions opérateurs utilisent un **template** supplémentaire pour assurer la compatibilité avec tous les types pour représenter un nombre en c++.
Toutes les fonctions de la classe sont **constexpr** pour permettre que les calculs se fassent à la compilation.

II. Mathématiques

Questions du sujet

- **Question - Opérateur de division**

Pour formaliser l'opérateur de division dans le cadre d'une division d'un rationnel 1 par un rationnel 2, nous avons fait le choix de multiplier le rationnel 1 par l'inverse du rationnel 2 (les deux opérateurs étant déjà implémentés).

$$ratio1 \,/\, ratio2 \;=\; ratio1 \;*\; ratio2^{-1}$$

- **Question - D'après vous, à qui s'adresse la puissance −1 de la ligne 8 ou bien la somme de la ligne 12 ?**

La puissance −1 de la ligne 8 :
On a recherché la partie entière de l'inverse de la partie décimale du nombre. On a donc un ratio avec un entier au numérateur et 1 au dénominateur (*et une éventuelle partie décimale).*

On inverse donc le tout pour remplacer la partie décimale du nombre avec un Ratio qui aura un entier au dénominateur et 1 au numérateur (*ou en autre entier si une éventuelle partie décimale est trouvée à l'inverse de la partie décimale)*.

La somme de la ligne 12 additionne un Ratio qui vaut la partie entière du nombre à convertir (avec celle-ci au numérateur et 1 au dénominateur) et un Ratio qui vaut la partie décimale du nombre à convertir. La partie décimale est analysée avec un appel récursif de la fonction dans le **cas x < 1**.

- **Question - conversion d'un float négatif en rationnel**

Nous avons fait le choix de gérer la conversion d'un nombre flottant en rationnel via un constructeur de notre classe Ratio.

Lorsque ce nombre flottant est négatif, le constructeur conserve l'information dans une variable sign (vaut 1 si positif ou nul, -1 si négatif) et appelle la fonction donnée par l'énoncé *convert_float_to_ratio* (renommée en convertRealToRatio) avec en argument la valeur absolue du flottant. La sortie de cette fonction sera ensuite multipliée par sign.

```
Ratio( flottant ) :
    *this = (flottant > 0 ? 1 : -1) * convert_float_to_ratio(valeur_absolue (flottant),nb_iter)
```

- **Question - Grands nombres assez mal représentés**

Les grands nombres comme les plus petits sont assez difficiles à représenter car ils requièrent de très grandes valeurs sur le numérateur et le dénominateur. Ils peuvent même dépasser les limites de représentation du type sur lequel ils sont stockés.
Des solutions sont proposées dans le paragraphe suivant

- **Question - Dépassement des limites de représentation**

Le numérateur et le dénominateur peuvent dépasser la limite de représentation des entiers en C++.
Une première solution serait de remplacer le type (par long long int par exemple) mais plus de mémoire serait nécessaire.
Une deuxième solution serait de remplacer le numérateur et le dénominateur par les multiples d'un nombre N les plus proches et ainsi simplifier la fraction en divisant par N le numérateur et le dénominateur (plus N est petit, plus la précision augmente mais moins les valeurs du numérateur et du dénominateur diminuent).

$$Soit\ N\ tq : \quad numérateur \simeq n * N \quad et \quad dénominateur \simeq m * N, \quad (n,m) \in Z$$
$$On\ obtient : \ Ratio \simeq \frac{n}{m}.$$

Fonctions mathématiques

Nous avons implémenté des fonctions mathématiques que nous avons formalisées des façons suivantes :

Ces fonctions retournent toutes des instances de la classe Ratio.

- Valeur absolue :

$$|ratio| = \frac{|numérateur|}{dénominateur\ positif}$$

- Partie entière :

$$E(ratio) = \frac{numérateur - numérateur[dénominateur]}{dénominateur}$$

Nous avons aussi voulu ajouter la possibilité de calculer le cos, sin, tan, exp, et log directement sur notre classe ratio. Nous avons globalement utilisé les fonctions correspondantes de std. Le principal problème ici est la conversion du résultat vers un ratio, c'est cela qui cause une approximation plus ou moins grande en fonction des cas. Nous ne recommandons pas l'usage de ces fonctions.
Une solution (non codée) serait de faire une approximation sur le résultat avant la conversion afin que le ratio final soit très proche de la valeur attendue

- Sinus :

$$sin\ (ratio) = conversionEnRatio(\ sin(numérateur/dénominateur)\ )$$

- Cosinus :

$$cos\ (ratio) = conversionEnRatio(\ cos(numérateur/dénominateur)\ )$$

Remarque sin et cos :
Nous voulions tout d'abord utiliser les formules d'Euler pour utiliser les propriétés des rationnels plutôt que passer par des calculs sur flottant, nous avons effectué quelques tests avec les nombres duaux et la librairie std::complex, mais nous nous sommes vite aperçu que le problème venait de la conversion d'un nombre en ratio. Même avec un calcul exact l'approximation vient toujours de la conversion finale vers un ratio.

$$\cos(ab) = \frac{(\cos b + i \sin b)^a + (\cos b - i \sin b)^a}{2}$$

- Tangente :

$$tan(ratio) = \frac{sin(ratio)}{cos(ratio)}$$ *(quotient de deux Ratios)*

- Exponentielle :

$$exp(ratio) = \frac{exp(numérateur)}{exp(dénominateur)}$$

Remarque :
Nous aurions pu utiliser la propriété *exp(a/b) = exp(a)^(-b)* , mais cela impliquait en un 2e temps l'utilisation de notre fonction de conversion d'un nombre flottant en Ratio (ce qu'on cherche à éviter pour profiter des propriétés des nombres rationnels).

- Logarithme népérien :

$$ln\,(ratio) = conversionEnRatio(\,ln(numérateur) - ln\,(dénominateur)\,)$$

- Puissance :

Si l'exposant entier x <0 alors :
On inverse le Ratio ratio.

On retourne le ratio : $$\frac{numérateur^{\,|x|}}{dénominateur^{\,|x|}}$$

En cas de **puissance négative**, l'**inversion du Ratio avant le passage à la puissance** permet de conserver des entiers au numérateur et au dénominateur en leur passant en exposant un entier positif au lieu d'un entier négatif.

En cas de **puissance non entière**, le numérateur et le dénominateur étant codés en entiers, le résultat est approximatif.

- Racine carrée : $$\sqrt{ratio} = ratio^{\,0.5}$$

Lien du git : https://github.com/loicqql/IMAC_ratio