Projet de Mathématiques / Programmation

Laurence WANG, Elia ROARD

4 janvier 2023

1 Présentation

Voici notre rapport présentant toutes les fonctionnalités implémentées à notre librairie Ratio.

Tableau bilan des fonctionnalités implémentées

Fonctionnalité	Attendue	Etat
Classe Rationnelle en template	Non	Codé et fonctionnel
Opérateurs de comparaison <=, >=, ==, !=	Oui	Codé et fonctionnel
Opérateurs + , - , * , / entre Ratio	Oui	Codé et fonctionnel
Opérateur - unaire	Oui	Codé et fonctionnel
Opérateur * par un float / int	Oui	Codé et fonctionnel
Opérateurs +=, -=, *=, /=	Non	Codé et fonctionnel
Exprimer 0	Oui	Codé et fonctionnel
Exprimer l'infini	Non	Codé et fonctionnel
Inverse d'un rationnel	Oui	Codé et fonctionnel
Valeur absolue	Oui	Codé et fonctionnel
Partie entière	Oui	Codé et fonctionnel
Racinée carrée	Oui	Codé et fonctionnel
Cos	Oui	Codé et fonctionnel
Sin	Non	Codé et fonctionnel
Puissance	Oui	Codé et fonctionnel
Exponentielle	Oui	Codé et fonctionnel
Ratio sous forme irréductible	Oui	Codé et fonctionnel
Conversion des réels en ratio (négatifs compris)	Oui	Codé et fonctionnel
Conversion des ratio en float	Oui	Codé et fonctionnel
Readme	Oui	Codé et fonctionnel
Cmake	Oui	Codé et fonctionnel
Doxygen	Oui	Codé et fonctionnel
Tests unitaires	Oui	Codé et fonctionnel
Exemples	Oui	Codé et fonctionnel
Surcharge de l'opérateur << (1) pour l'affichage	Oui	Codé et fonctionnel
Fonction Display ratio pour l'affichage	Non	Codé et fonctionnel
Fonction nombre de chiffres après la virgule	Non	Codé et fonctionnel
Utilisation de constexpr	Non	Codé et fonctionnel
Tests de rapidité et de précision	Non	Codé et fonctionnel

2 Partie Mathématiques

2.1 Opération sur les rationnels

Pour tous les opérateurs présentés dans cette partie, à l'exception de "/", nous avons conscience qu'il était également possible (et plus rapide) d'utiliser directement les fonctions de la STL. Mais nous avons pris le parti de recoder ces fonctions nous-même à l'aide des différents algorithmes vus en cours.

.

***Question* - Comment formaliseriez vous l'opérateur de division / ?**

L'opérateur de division dans le cas des ratios peut se formaliser de façon très simple en multipliant le ratio divisé par l'inverse du ratio diviseur. Ainsi dans le programme il suffit juste de multiplier le divisé par le diviseur auquel on échange numérateur et dénominateur.

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

***Question* - Comment formaliseriez vous l'opérateur de puissance ?**

Nous avons implémenté l'opérateur de puissance en utilisant un algorithme récursif. Nous appliquons l'algorithme sur le numérateur et le dénominateur de notre nombre, puis nous le réduisons à sa forme simplifiée.
Afin d'améliorer la complexité de cet algorithme,

nous avons remarqué que pour n pair, $a^n = a^{(n/2)^2}$ Par exemple $a^6 = (a^3)^2$
Et par la même réflexion pour n impair, $a^n = a * a^{((n-1)/2)^2}$ Par exemple $a^7 = a * (a^3)^2$
Ainsi on peut calculer la puissance avec une complexité en log(n)

Algorithm 1: Algorithme de la puissance

```
Data: n
if n == 0 or this == 1 then
    Return : Un
end
if n pair then
    calculPair = pow(n/2)
    Return : calculPair * calculPair
else
    calculImpair = pow(n-1)/2
    Return : this * calculImpair * calculImpair
end
```

Pour les opérateurs suivants, nous ne pouvons pas obtenir la valeur précise du résultat en nombre rationnel. Nous avons donc décidé de renvoyer une approximation de la valeur en convertissant notre nombre rationnel en nombre à virgule flottante puis en reconvertissant ce nombre en nombre rationnel après opération.

***Question* - Comment formaliseriez vous la racine ?**

Pour formaliser la racine, nous avons choisi d'utiliser l'approche de Newton. Celui-ci se repose sur l'itération sur la suite suivante :

$$\begin{cases} u_n = 1 \\ u_{n+1} = \frac{u_n + x/u_n}{2} \end{cases}$$

Pour notre condition d'arrêt, nous estimons que lorsque la valeur absolue de la différence entre Un et Un+1 est inférieure à une Precision Epsilon que nous avons définie, le résultat est convenable.

Algorithm 2: Newton SQRT

Data: n
Data: $PRECISION$
reel $Un \leftarrow 1$;
reel $next\_Un \leftarrow Un$;
while $|Un - next\_Un| > PRECISION$ **do**
  $Un = next\_Un$
  $next\_Un = \frac{Un+n/Un}{2}$
end
Return : Un

***Question* - Comment formaliseriez vous l'exponentielle ?**

Pour formaliser l'exponentielle, nous avons choisi d'utiliser le développement de Taylor avec la méthode Horner

Taylor

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \tag{2}$$

Horner

$$\sum_{i=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!}... = (1 + x(1 + \frac{x}{2}(1 + \frac{x}{3}(1 + \frac{x}{4}(...)))) \tag{3}$$

Algorithm 3: Algorithme de l'exponentielle

réel **Data:** x
entier**Data:** max_iter
reel $x0 \leftarrow |x|$;
ratio $result \leftarrow 1 + \frac{x0}{max\_iter}$;
for *entier* $i \leftarrow max\_iter$ *to 1* **do**
  $result \leftarrow$ result* $\frac{x0}{i}$
  $result \leftarrow result + 1$
end
if $x < 0$ **then**
  Return : $\frac{1}{result}$
end
Return : $result$

Nous avions pensé à directement exploiter le nombre rationnel pour l'exponentielle de cette façon :

Algorithm 4: Algorithme de l'exponentielle n.2

entier **Data:** max_iter
ratio $x0 \leftarrow \frac{|this.num|}{this.denom}$;
ratio $result \leftarrow \frac{1}{1}$+x0
for *entier* $i \leftarrow max\_iter$ *to 1* **do**
  $result \leftarrow result * x0 * \frac{1}{i}$
  $result \leftarrow result + 1$
end
if $x < 0$ **then**
  Return : $\frac{1}{result}$
end
Return : $result$

Mais nous rencontrons un problème avec cette méthode. En effet, bien qu'elle soit plus précise

et plus rapide à l'exécution comparée à la première qui fait appel à une conversion en float puis une reconversion en ratio, cette méthode fait s'enchaîner des opérations entre ratios. Ainsi, nous prenons davantage de risques de faire dépasser la limite de représentation des nombres avec cette implémentation. Pour pouvoir éviter le dépassement, une des solutions serait de réduire le nombre d'itérations ce qui :
- ne nous garantit pas de ne pas dépasser (c'est toujours possible, en fonction des nombres mis en entrée)
- nous fait aussi perdre en précision

C'est pourquoi nous avons plutôt opté pour la méthode avec conversion, qui nous semblait plus robuste.

***Question* - Comment formaliseriez vous le cos ?**

Pour cos, nous avons à nouveau utilisé le développement de Taylor. Il est à noter que cette méthode est principalement efficace lorsque le nombre est entre -PI/2 et PI

Taylor

$$cos(x) = \sum_{n=0}^{\infty} (-1)^n * \frac{x^{(2n)}}{(2n)!} \tag{4}$$

Algorithm 5: Algorithme du cosinus

reel **Data:** x
entier **Data:** max_iter
reel $result \leftarrow 0$;
reel $x0 \leftarrow 1$;
for *entier* $i \leftarrow 1$ *to* max_iter **do**
  $result \leftarrow result + x0$
  x0 $\leftarrow x0 * \frac{(-x*x)}{(2*i-1)*(2*i)}$
end
Return : $result$

Nous avons appliqué la même méthode pour sin.

Taylor

$$sin(x) = \sum_{n=0}^{\infty} (-1)^n * \frac{x^{(2n+1)}}{(2n+1)!} \tag{5}$$

Algorithm 6: Algorithme du sinus

reel **Data:** x
entier **Data:** max_iter
reel $result \leftarrow 0$;
reel $x0 \leftarrow 1$;
for *entier* $i \leftarrow 1$ *to* max_iter **do**
  $result \leftarrow result + x0$
  x0 $\leftarrow x0 * \frac{(-x*x)}{(2*i+1)*(2*i)}$
  result $\leftarrow result * x$
end
Return : $result$

2.2 Conversion d'un réel en rationnel

.

***Question* - D'après vous, à quel type de données s'adressent la puissance 1 de la ligne 8 et la somme de la ligne 12 ?**

Le -1 de la fonction est le -1 de notre classe Ratio. En effet, la fonction renvoie un Ratio, ainsi pour appliquer l'inverse sur son return, il faut utiliser l'inverse de notre classe.

Il en est de même pour la somme de la ligne 12 qui est aussi la somme entre deux ratios et qui donc utilise l'opérateur + surchargé de notre classe.

***Question* - Comment modifier l'algorithme de conversion d'un réel en rationnel pour qu'il gère également les nombres négatifs ?**

Un nombre négatif se code de la même façon que son opposé à la différence qu'il revêt un signe moins à la sortie. Pour adapter cet algorithme aux négatifs il suffit de créer une variable *signe* qui stocke son signe et le réapplique en sortie. Ainsi le réel négatif est traité par la fonction comme un réel positif mais ressort bien comme un réel négatif. Et le programme fonctionne désormais pour les réels négatifs aussi.

Algorithm 7: Fragment de code du convert_float_to_ratio

```
...
if x < 0 then
    signe = -1 ;
    ...
    Return : (...) * signe
end
```

2.3 Analyse

.

***Question* - D'une façon générale, on peut s'apercevoir que les grands nombres (et le très petits nombres) se représentent assez mal avec notre classe de rationnels. Voyez vous une explication à ca ?**

Les entiers sont codés sur 4 octets en C++, soit 32 bits. Avec un bit réservé pour le signe. La représentation des entiers en C++ s'étend donc de

$-2^{31} \leq x \leq 2^{31} - 1$

$2147483648 \leq x \leq 2147483647$

(environ 4 milliards d'entiers)

C'est pourquoi les grands nombres et les petits nombres sont mal représentés.

Dans notre classe, lorsque l'on souhaite convertir un float en entier cela risque de poser davantage de problèmes lorsque l'on veut garder de la précision. En effet un très grand nombre à virgule risque de faire "dépasser" la limite de représentation, étant donné que la fraction sera multipliée par un multiple de 10 pour pouvoir représenter les décimales.

Exemple

$100000000 \rightarrow 100000000,1$

$\frac{100000000}{1} \rightarrow \frac{1000000001}{10}$

Nous serons donc contraints à enlever la précision de sa mantisse.

Utiliser des long int :

Privilégier l'utilisation des long int lorsqu'on initialise notre nombre Ratio peut également nous aider à éviter certains dépassement étant donné qu'ils sont codés sur 8 octets. Surtout si nous savons que nous allons utiliser des fonctions comme exponentielle ou puissance sur le nombre.

***Question* - Lorsque les operations entre rationnels s'enchainent, le numerateur et le dénominateur peuvent prendre des valeurs très grandes, voire dépasser la limite de représentation des entiers en C++. Voyez vous des solutions ?**

Nous avons en effet remarqué lors de nos premiers tests que nombreux de nos outputs étaient très différents du résultat attendu. (Par exemple une différence de signe sur un nombre très éloigné de celui de départ). Nous avons trouvé 2 principales causes à cela :
- L'enchaînement des opérations
- A l'étape return(x-q), il se peut que nous renvoyons un nombre très petit qui explose complètement lorsqu'on veut l'inverser.

Pour palier ces problèmes nous avons réfléchi à plusieurs solutions :

1. Tronquer le nombre pour pouvoir gagner en itérations

→ récupérer le nombre de digits après la virgule. Tronquer le nombre si le nombre de digits est supérieur à un certain seuil.

En faisant nos tests, nous avons décidé de tronquer lorsqu'il y a 7 chiffres après la virgule.

Algorithm 8: Algorithme nombre de digits

reel **Data:** x
entier **Data:** $count\_digits\_limit$
$count \leftarrow 0$;
$x \leftarrow |x|$;
$x \leftarrow x - x\_tronque$;
while $|x| - epsilon$ ***and*** $count < digit\_limit$ **do**
  $x \leftarrow x * 10$;
  count ++ ;
  $x = x - x\_tronque$
end
Return : $count$

Pour nos epsilon et notre digits_limit nous avons choisi d'utiliser les éléments de std : :numeric_limits

Problème :
- Ces opérations augmentent le temps d'exécution du programme.
- Il se peut que nous entrions dans une boucle infinie en tronquant "à l'aveugle". En effet, puisqu'on ne peut pas représenter tous les nombres en float ou en double, nos troncatures font l'objet d'erreurs d'arrondi. A ce moment là, il est possible que nos deux if se fassent "la passe" avec les mêmes valeurs.

Par exemple
3,0016789 → 3,003 (erreur de troncature)
3,003 - 3 = 0,003
1 / 0,003 = 333,333
333,333 - 333 = 0,333
1/0,333 = 3,003003
etc.

2. Utiliser une valeur de précision pour éviter les explosions avec les petits nombres

Dans notre $if(x1)$, on utilise une valeur de précision pour vérifier que x-q ne donne pas un résultat trop petit. En effet, si le résultat est inférieur à cette valeur, cela veut dire que nous somme suffisamment proches du résultat souhaité et que nous pouvons arrêter d'itérer.

Par la suite, nous avons exploité cette méthode pour éviter le problème de boucle infinie expliqué plus haut.

Problème :
En utilisant un epsilon plus petit pour éviter la boucle infinie, nous perdons en précision

.

3. Réduire le nombre d'itérations

Nous avons remarqué rapidement que faire 3 ou 4 itérations nous produisait souvent des résultats satisfaisants, même si parfois imprécis.

(Tests fait avec implémentation de la troncature et de la précision)

4 itérations semblaient convenables mais nous avons cherché un moyen de faire le nombre d'itérations adéquat en fonction de l'input. La précision que nous avions implémenté permettait d'arrêter l'itération mais pas toujours de façon appropriée.

Notre principale idée s'est reposée sur une fonction qui choisirait le bon appel de fonction (celle avec 4 itérations ou celle avec 20) en fonction de sa différence avec l'input de départ.

Algorithm 9: Algorithme closest

Data: n
Data: a
Data: b
if $|num - a| \leq |num - b|$ **then**
  Return : a
end
Return : b

Algorithm 10: Algorithme best_convert_to_float

$resultat1 \leftarrow convert\_to\_float(4)$;
$resultat2 \leftarrow convert\_to\_float(20)$;
if $compare\_closest(x, result1, result2) == result1)$ **then**
  Return : $result1$
end
Return : $result2$

Problème : le temps d'exécution est plus long

Voici quelques exemples de tests que nous avons fait pour comparer les deux méthodes :

Convert				
Runs	100000			
Range	1000000			
Speed			**Precision**	
				real value
Best convert	1,1546		Best convert	0,0023
convert 4 ite	0,1227		convert 4 ite	0,0031

Sqrt				
Runs	100000			
range	[0 ; 1e5]			
Speed			**Precision**	
	Ratio (newton)			real value
Best convert	0,5311		Best convert	0,0057
convert 4 ite	0,0225		convert 4 ite	0,0071

Exp				
Runs	100000			
range	[0 ; 10]			
Speed			**Precision**	
	Ratio (horner)			real value
Best convert	1,0328		Best convert	0,4991
convert 4 ite	0,1081		convert 4 ite	0,5335

Conclusion :

Nous avons décidé de finalement faire tous nos calculs sur 4 itérations et de ne pas utiliser le best convert. En effet, nous estimons que la précision gagnée ne justifie pas le temps d'exécution. De plus, les ratio renvoyés avec 4 itérations ont tendance à être moins grand au numérateur et au dénominateur ce qui est plus simple à manipuler.

D'une manière générale, nous avons remarqué qu'il était possible de gagner en précision sur certains intervalles en augmentant le nombre d'itérations. Mais nous avons préféré garder 4 itérations car il s'agit du nombre d'itération le plus fiable.