Rapport : Coder les nombres rationnels

Lou Couard et Audrey Diep

2 janvier 2023

Résumé

Rapport du projet sur les nombres rationnels codés en c++.

1 Tableau récapitulatif

Voici un récapitulatif des fonctionnalités implémentées dans ce projet.

Elements demandés	Etat
Rendre une fraction irréductible	codé et fonctionnel
Moins unaire	codé et fonctionnel
Valeur absolue	codée et fonctionnelle
Partie entière	codée et fonctionnelle
Produit entre rationnel et virgule flottant	codé et fonctionnel
Opérations binaires (+ , - , * , /)	codés et fonctionnels
Opérateurs de comparaison	codés et fonctionnels
Quelques opérateurs connus (exp, ln, pow, sqrt)	codés et fonctionnels
Surcharge opérateur $\ll$	codée et fonctionnelle
Conversion d'un réel en rationnel	codé et fonctionnel
———	———
Exemples d'utilisation	complets
Tests unitaires	complets
Doxygen	fait
Cmake et readme	faits
–Bonus–	———
Coder la classe en template	fait
Coder la classe en constexpr	pas fait :c
Division entre rationnel et virgule flottant	codé et fonctionnel

Table 1 – Tableau récapitulatif.

2 Déroulé du projet

2.1 Mise en place du projet

Au commencement du projet, nous nous sommes attardées sur le `CMake` ainsi que la création du `git`. Nous avons eu quelques problèmes liés notamment au dossier `/build` qui aurait dû figurer dans un `.gitignore` dès le début, en effet nous avions déjà `commit` et `push` une première fois sur l'un de nos ordinateurs et nous nous sommes rendues compte que cela ne fonctionnait pas sur l'autre. Puis, après quelques recherches et grâce à quelques commandes dans le terminal nous avons pu obtenir un `CMake` fonctionnel !

2.2 Création de la librairie

Nous avons débuté le projet en créant une classe template "Rational" avec 2 attributs privés, un numérateur et un dénominateur. Nous avons alors commencé par coder toutes les opérations binaires : + , - , * , / Nous avons ensuite enchainé avec les petites fonctions telles que : la valeur absolue, le moins unaire, ainsi que la partie entière qui nous seront utiles très rapidement. Pour la surcharge de l'opérateur $\ll$, nous avons décidé d'afficher un rationnel comme tel :

Algorithm 1: Affichage d'un rationnel

```
1 Function operator≪
    Input: r1 ∈ ℝ : un nombre rationnel d'entiers
2   cout ≪ r1.numerator ≪ "/" ≪ r1.denominator ≪ endl;
```

Exemple d'utilisation :

```
int main()
    Rationnel r1 = (3,2)
    cout << r1 << endl;
    return 0;

Console : 3/2
```

2.3 Convertir des réels en rationnels

En theorie, les opérations sur les nombres rationnels n'impliquent aucune approximation numérique, et sont par conséquent très précis.
En pratique, le numérateur et le dénominateur sont deux entiers codés sur un nombre fini de bits, ce qui implique des limites sur la précision des opérations effectuées et c'est ce que nous allons explorer lors de ce projet.

3 Problèmes connus

3.1 Comment formaliseriez vous l'opérateur de division / ?

Le temps d'exécution d'une multiplication est beaucoup plus faible que celui d'une division et diviser revient à multiplier par l'inverse, c'est pourquoi nous avons codé l'opérateur diviser comme suit :

Algorithm 2: Surcharge Opérateur /

```
1 Function operator/
    Input: r1, r2 ∈ ℝ : deux rationnels
2   //condition
3   if r2.numerateur == 0 then
4     return inf
5   // si r2.numerateur! = 0
6   else
7     return (r1∗ inverse(r2))
```

3.2 Vous pourrez également explorer les opérations suivantes :

$$\sqrt{\frac{a}{b}}, \quad \cos\frac{a}{b}, \quad (\frac{a}{b})^k, \quad e^{\frac{a}{b}}, \quad etc...$$

ou d'autres opérateurs que vous aimez bien.

Nous avons codé la racine carrée, l'exposant, l'exponetielle et le logarithme en base 2. Pour l'opération exponentielle, nous avons utiliser la formule suivante :

$$e^{\frac{a}{b}} = (e^a)^{\frac{1}{b}}$$

Algorithm 3: Fonction Exponentielle d'un rationnel

```
1 Function exponan()
      Input: r ∈ ℝ : un rationnel
2     return double (pow(exp(r.numerator), 1/r.denominator));
```

Pour l'exposant nous avons utiliser la formule suivante :

$$(\frac{a}{b})^k = \frac{a^k}{b^k}$$

Algorithm 4: Fonction exposant d'un rationnel

```
1 Function power()
      Input: r ∈ ℝ : un nombre rationnel
             n ∈ ℕ : l'exposant
2     Ratio result;
3     result.numerator = (pow(r.numerator, n));
4     result.denominator = (pow(r.denominator, n));
5     return result;
```

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

La puissance -1 de la ligne 8 s'adresse à des rationnels. Il s'agit de faire l'inverse d'un nombre et il est nettement plus aisé d'inverser un rationnel que d'inverser un flottant. La somme de la ligne 12 est une somme entre deux nombres rationnels.

3.4 Comment le modifier pour qu'il gère également les nombres réels négatifs ?

Pour que `convertFloatToRatio` fonctionne aussi pour les réels négatifs il faut rajouter une condition pour x < 0 qui calculerait le `convertFloatToRatio` de sa valeur absolue et qui retournerait son opposé.

Algorithm 5: Conversion d'un réel en rationnel

```
1  Function convertFloatToRatio
       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 < 0
7      if x < 0 then
8          return −(convertFloatToRatio(|x|, nb_iter))
9      // appel récursif si x < 1
10     if x < 1 then
11         return ( convertFloatToRatio(1/x, nb_iter))^−1
12     // appel récursif si x ≥ 1
13     if x ≥ 1 then
14         q = ⌊x⌋ // partie entière
15         return q/1 + convertFloatToRatio(x − q, nb_iter − 1)
```

3.5 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. Voyez vous une explication à ça ?

On sait déjà qu'avec l'inverse les grands nombre et les petits nombres (proches de 0) subiront le même sort. En effet, en inversant un très petit nombre on obtient un très grand nombre et vice versa.

Notre hypothèse est que lorsque l'on manipule des nombres très grands ou très petits nous sommes rapidement ammenés à franchir la limite d'un type et donc de dépasser la capacité du type. En effet en essayant la fonction `convertFloatToRatio` sur un petit ou grand nombre (moins que 10^{-10} ou plus que 10^{10} respectivement), on s'aperçoit que la fonction `intPart(x)` renvoie toujours la même valeur qui est -2147483648, cette valeur correspond au plus petit entier qui peut être représenter. Notre hypothèse est donc bien vérifiée ! Dans la suite, nous apporterons des précisions sur ce phénomène et nous parlerons d'overflow et d'underflow.

3.6 Lorsque les opérations entre rationnels s'enchaînent, le numérateur et le dénominateur peuvent prendre des valeurs très grandes, voire dépasser la limite de représentation des entiers en C++. Voyez vous des solutions ?

Si nous utilisons des entiers codés sur 32 bits :

$$\text{Vmin} = -2^{31} \text{ et } \text{Vmax} = 2^{31}-1.$$

Nous pouvons utiliser des `LONG_INT` voire des `LONG_LONG_INT` mais cela ne fais que repousser le problème. Pendant nos recherches nous avons aussi découvert cette librairie qui semble très intéressante. Comme mentionné sur le site "GMP est une bibliothèque gratuite pour l'arithmétique de précision arbitraire, fonctionnant sur des entiers signés, des nombres rationnels et des nombres à virgule flottante. Il n'y a pas de limite pratique à

la précision, sauf celles impliquées par la mémoire disponible dans la machine sur laquelle GMP s'exécute." Ce qui nous permettrait d'atteindre une excellente précision pour des nombres très petits et très grands mais est-ce qu'une précision aussi fine est vraiment nécessaire pour notre travail ? Nous sommes surement trop gourmandes et c'est pourquoi nous avons décidé de nous pencher vers notre première hypothèse qui était de travailler avec des valeurs approximées et faire une fonction assert qui nous prévient lorsque d'un éventuel dépassement de la limite de représentation des entiers. Nous perdrons en précision mais cela nous permettra tout de même d'obtenir des résultats approximés satisfaisants sans trop de difficulté, on l'espère.

4 Problèmes rencontrés

4.1 La conversion d'un float en ratio

Nous avons remarqué que notre fonction `convertFloatRatio`() fonctionnait bien jusqu'à un certain nombre d'itération et après elle pouvait retourner des "exception en point flottant" c'est à dire des divisions par 0. Nous avons alors observé lors de l'utilisation de `inverse`(), que le nombre de chiffres composant notre variable x augmentait sans cesse, à cause de l'imprécision de représentation des flottants, produisant alors une exception en point flottant après quelques itérations. Dans un premier temps nous avons donc décidé de tronquer nos résultats entre chaque itérations. Nous aurions pu faire un arrondi qui est 2 fois plus précis qu'une troncature, mais nous voulions seulement tester avec une troncature grossière pour observer si notre problème était bien dû à cela. En tronquant nous perdions en précisions mais cela nous permettait de résoudre le problème et d'augmenter le nombre d'itération de la fonction `convertFloatRatio`().

Cependant nous avons remarqué que pour certains cas la fonction retournait encore des résultats surprenants. Notamment des fractions avec une valeur négative pour un float positif. Pour ce qui est des valeurs négatives, ce n'était théoriquement pas possible car `convertFloatRatio`() ne s'occupait pour le moment que de nombres positifs et dans ses étapes aucun calcul ne "faisait passer un nombre en négatif". Cela nous a donc mis la puce à l'oreille et nous avons fait des recherches sur ce qu'il pourrait se produire lorsque nous dépassons les limites de représentation d'un entier et comment les entiers signés sont affectés. Nous en avons déduit que nous dépassions bien les limites de représentation des flottants. C'est là qu'on commence à entrer dans un cercle vicieux car nous avons remarqué ensuite que notre fonction `convertFloatRatio`() n'était pas très robuste. Dans une majorité des cas, elle fonctionne très bien pour un nombre d'itération supérieur à 10 mais ce n'est pas le cas pour tous les nombres. Par exemple, elle fonctionne très bien avec 15 itérations pour 41,001 mais ne fonctionne que jusque 8 itérations pour 41,003.

Résultat :
il faut garder un nombre d'itération réduit pour que la fonction `convertFloatRatio`() fonctionne avec toutes les valeurs, mais cela entraîne une précision moindre. (Jusque là nous ne travaillions qu'avec des rationnels avec des `INT` au numérateur et au dénominateur)

4.2 codage binaire des nombres flottants

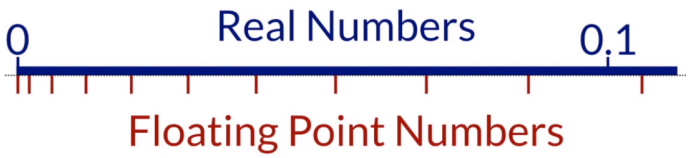
La plupart de nos tests unitaires testent les opérateurs pour des nombres réels entre -1000 et 1000 générés de façon aléatoires.

Algorithm 6: Test unitaire sur l'opérateur +

```
1  Function TESToperator+
2        size ∈ ℝ : intervalle des nombres testés (-size, size)
3        epsilon ∈ ℝ : valeur maximale de la différence, passé ce seuil le calcul est
          considéré comme faux
4     // on fait le test 100 fois
5     while run <100 do
6        // générer aléatoirement des nombres
7        double a = gen() ;
8        double b = gen() ;
9        // convertir les float en rationnels
10       frac1 = (convertFloatRatio(a, maxIter));
11       frac1 = (convertFloatRatio(b, maxIter));
12       // comparer les résultats obtenus avec les différents opérateurs +
13       double actualResult = (convertFloatRatio(a, maxIter));
14       double expectedResult = a + b ;
15       double difference = (abs(actualResult − expectedResult));
16       if (difference ≤ epsilon) then ok
17       run ++ ;
```

Nous avons bien fait en sorte d'utiliser l'opérateur + entre des floats et de le comparer à l'opérateur + que nous avons surcharger pour qu'il fonctionne sur les rationnels.

Nous avons utilisé un intervalle de précision pour vérifier nos résultats. En effet, en travaillant avec des nombres flottants il est difficile voire impossible de trouver une différence égale à 0. Par exemple, 196.6-196.6 $\neq$ 0. En informatique les nombres sont codés en binaire ce qui complexifie la définition des nombres décimaux. Ce qu'il faut retenir c'est qu'en informatique les nombres flottants ne sont que des échantillons des nombres réels, comme nous pouvons le voir sur ce schéma provenant de ce site Certains nombres réels sont mal représentaté en binaire. Par exemple, le nombre 0.1 ne peut pas être exactement défini avec des nombres flottants. Nous sommes obligés de passer par une approximation cela exlique pourquoi 0.1-0.1$\neq$ 0.



Le fait d'utiliser un intervalle de précision nous permet aussi de valider des résultats qui sont théoriquement faux mais sont vraiment proches et dont la différence s'explique avec des arrondis implicites et/ou explicites que nous effectuons lors de nos opérations.

4.3 La troncature

En nous appuyant sur la façon dont les nombres flottants sont codés dans nos machines nous avons fait l'hypothèse qu'à aucun moment nous passions par la première condition d'arrêt de la fonction `convertFloatRatio()`. Prenons l'exemple grossier de la conversion de x=1,5 en nombre rationnel, on remarque que x sera rapidement égal à 0 :

```
convertFloatRatio(1.5)
{
```

```
    x > 1        (x=1.5)
    return (1+convertFloatRatio(0.5, nbIter-1));
    x < 1        (x=2.0)
    return (convertFloatRatio(2.0, nbIter))^(-1);
    x > 1        (x=0.0)
    return (2+convertFloatRatio(0.0, nbIter-1);
    x == 0
    return 0/1;
}

Console : 3/2 (-> 1+(1/2)+0)
```

Or d'après nos recherches lorsque l'on exécute cet algorithme sur notre machine, il ne devrait jamais satisfaire la première condition (x==0) car (5f-5 $\neq$ 0). Nous avons remarqué qu'en effet pour tous nos tests unitaires aucun ne s'arrêtait grâce à la condition x==0. Cependant lorsque nous faisions `convertFloatRatio(1.5)` l'algorithme s'arrêtait bien grâce à la première condition.

En nous débarrassant des approximations des nombres flottants, nous pensions pouvoir satisfaire la condition x == 0 plus souvent et cela nous permettrait d'optimiser notre code. En effet, l'algorithme pourrait, dans certains cas, retourner un résultat assez précis pour satisfaire nos tests unitaires sans faire le maximum d'itération qui lui est proposé. Ce qui s'est avéré exact. Mais nous avons tout de même abandonner l'idée lorsque nous avons remarquer que cette troncature s'adressait à environ 9 tests sur 100. Nous avons aussi pensé aux repercussions que notre troncature pourrait avoir sur de tous petits nombres. En avançant dans notre projet nous avons trouvé l'utilisation de la troncature de moins en moins pertinente surtout en sachant qu'elle était aussi à l'origine d'un problème sur la multiplication entre les rationnels.

Nous avons alors pensé à plutôt changer la condition en remplaçant (x==0) par ($x < 0.0015$ && $x > -0.0015$) mais cela avait encore un énorme impact sur la conversion de nombres d'ordre négatif et très peu d'impact sur le reste de nos calculs. Il est tout de même intéressant d'utiliser la troncature lorsque nous travaillons avec des `INT`.

4.3.1 Les overflows et les underflows

Après avoir fixé des conditions pour notre `floatRatio()` nous pensions ne plus faire d'overflow (ou d'underflow) mais les tests unitaires nous ont prouvé le contraire. Par exemple pour des valeurs telles que :

x = 624,337 et y =-40,0457

Le resultat attendu du test unitaire "+" est 584,292 mais la machine nous retourne-189,281. Le test ne fonctionnait pas, pourtant la conversion de x et y en nombres rationnels était bonne.

$$\text{x}= \frac{1567087}{2510} = 624,337 \quad \text{y}= \frac{-88581}{2212} = -40,0457$$

Nous pensions coder une fonction qui nous prévient lorsque nous dépassons les bornes mais finalement nos tests unitaires suffisaient. Par exemple, grâce à nos tests unitaires nous avons remarqué qu'avec l'`operateur+` nous avions beaucoup plus de chances de dépasser les bornes et on ne parle pas de l'`opérateur`$\times$. En effet, rappelons nous que nous travaillons sur des entiers signés codés sur 32 bits. Lors de l'addition la machine fait :

$$\frac{a}{b} + \frac{c}{d} = \frac{a \times d}{b \times d} + \frac{c \times b}{d \times b}$$

En multipliant le numérateur de x par le dénominateur de y (1567087×-2212) la machine dépasse la limite d'underflow (-2^{31}).
En effet, -3 466 396 444 $\ll$ -2 147 483 648. Afin de rester dans les bornes, la machine a donc lu le dernier bit (le bit de signe) comme étant de signe positif et retourne donc 828 570 852 au numerateur de x et continue à faire les calculs. C'est donc pourquoi elle nous retourne -189,281 au lieu de 584,292.

Exemple sur un entier signé codé sur 4 bits (Vmax = -8 et Vmin = 7). Si on additionne 4 et 6 nous faisons un overflow car $10 \gg 8$:

4=0100 et 6=0110
6+4=1010

comme il s'agit d'entiers signés le dernier bit (celui tout à gauche) est le bit qui porte le signe. La machine ne lira non pas 10 mais -6 (=-8+2).

Dans nos premiers tests nous convertissions les `FLOAT` en fractions rationnelles utilisant des `INT`. Nous avons ensuite décidé d'utiliser des `LONG_INT` et cela nous a permis d'avoir une marge de manoeuvre plus grande pour les tests unitaires (robustesse) et de gagner en précision puisqu'il était possible d'utiliser un nombre d'itération plus grand lors de l'appel de la fonction `convertFloatRatio`(). Grâce à l'utilisation des `LONG_INT`, nos problèmes initiaux étaient résolus ! Nous avons donc décidé de ne plus utiliser la troncature pour garder une précision digne des plus grands ! Cependant, si l'on souhaite utiliser des `INT` plutôt que des `LONG_INT`, il faut alors modifier le nombre d'itérations à 3 maximum et utiliser la troncature, ce qui peut-être un peu fastidieux.

5 Ressentis et conclusion

Partir de l'idée qu'utiliser les rationnels nous permettrait de gagner en précision pour au final s'apercevoir que ça ne fonctionne pas si bien que ça était très intéressant. Notre progression n'était pas linéaire et nous avons passé beaucoup de temps sur des intuitions, théories, recherches qui ne nous menaient pas très loin. Cette façon d'avancer à taton grâce à chacune de nos hypothèses était enrichissante, parfois frustrante mais très enivrante. Plus on avançait dans le projet et plus on s'éloignait du but premier de l'énoncé mais plus on comprenait pourquoi nous ne pourrions pas vraiment atteindre parfaitement ce but.

Durant le projet, nous avons pu observer de nos propres yeux la limite de représentation des entiers ! (Et ça arrive bien plus vite qu'on ne le pense...)

En principe, une classe de rationnel est très pratique pour éviter les approximations. Cependant, lorsqu'on convertit un réel en rationnel en C++, pour avoir une bonne précision, notre numérateur et dénominateur peuvent parfois prendre de grandes valeurs, et en effectuant des opérations entre eux, on dépasse assez rapidement cette représentation... C'est pourquoi, en pratique les nombres rationnels en C++ peuvent être difficilement manipulable.