Projet Ratio

par Nina GRIGNOLA et Pauline GOBÉ

PROJET MENÉ DE OCTOBRE À DÉCEMBRE 2022, ESIPE, UNIVERSITÉ GUSTAVE EIFFEL

Professeur encadrant : Vincent Nozick

Date de rendu : 3 janvier

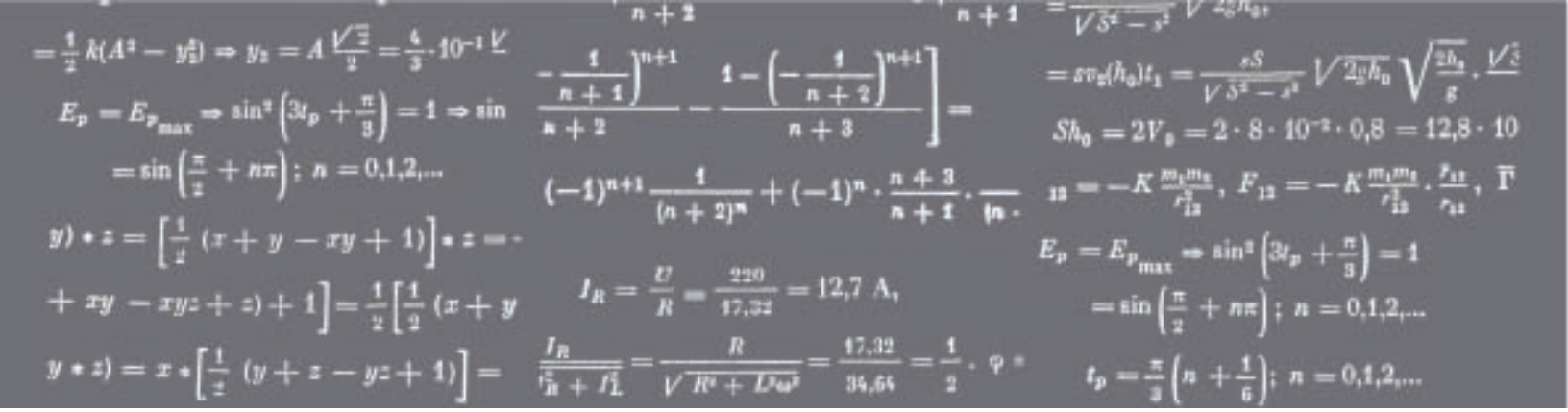
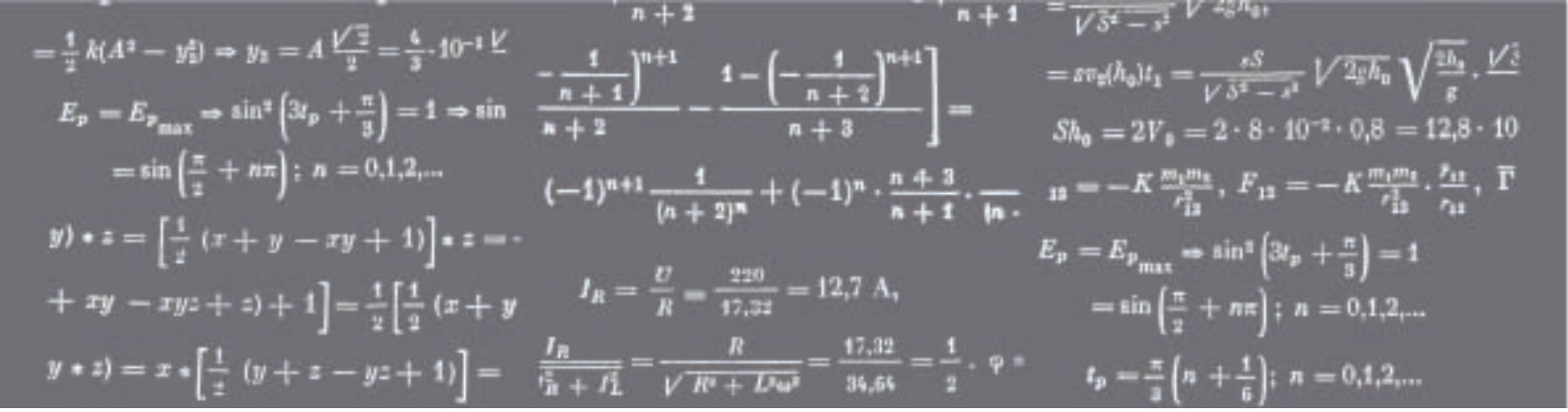


Table des Matières



1. Partie mathématique

L'objectif de la partie mathématique était de comprendre le fonctionnement des rationnels et plus particulièrement leur rapport avec les différentes opérations mathématiques que l'on peut leur appliquer. Les enjeux portaient donc à utiliser les librairies de maths déjà existantes pour les autres types, mais aussi de chercher si d'autres moyens de calcul n'étaient pas plus optimaux, ici, dans le cas des rationnels.

1.1 Comprendre et implémenter les opérateurs

1.1.1 Prise en main : formaliser l'opérateur division

Comme pour les autres opérateurs on applique les règles mathématiques. Ici, diviser un nombre par un ratio c'est le multiplier par son inverse d'où

$$\frac{\frac{a}{b}}{\frac{c}{d}} = \frac{a}{b} * \frac{d}{c} \tag{1.1}$$

Ayant déjà codé l'opérateur $*$ et a^{-1} on peut directement les utiliser pour coder l'opérateur /, ou bien écrire directement le résultat de l'équation ci-dessous :

$$\frac{a}{b} * (\frac{c}{d})^{-1} = \frac{ad}{bc} \tag{1.2}$$

1.1.2 Exploration de nouveaux opérateurs

1. $\sqrt{\frac{a}{b}}$

Pour cette opérateur, mathématiquement nous pouvons écrire que :

$$\sqrt{\frac{a}{b}} = \frac{\sqrt{a}}{\sqrt{b}} \tag{1.3}$$

Nous devons alors faire des tests pour voir si, en informatique, la racine carrée d'un rationnel est plus précis que la fraction entre deux racines carrées. Nous comparerons les résultats en testant avec des résultats connus comme $\sqrt{\frac{1}{4}}$, qui nous donnerait logiquement comme réponse 0.5. Pour cela nous faisons la différence entre la valeur exacte et les valeurs récupérées après calcul.

Après tests sur machine :

Il s'avère que les deux techniques fonctionnement quasiment identiquement. En effet, pour des résultats précis et finis comme l'exemple montré ci-dessus, il n'y a aucune différence entre les deux versions.Dans les autres cas, on observe une première différence entre les deux de l'ordre de 1.10^{-17}. Nous jugeons qu'il est possible de choisir une des deux options arbitrairement sans que cela n'ai d'impact notable sur nos calculs. Pour confirmer cela, nous avons affiché le temps mis par chaque fonction à l'exécution et les temps sont du même ordre de grandeur.

2. $\cos\left(\frac{a}{b}\right)$

Cet opérateur ne peut pas s'écrire dans une autre forme. Ce qu'on peut tester par contre c'est d'essayer de faire une approximation. La seule qui nous est venue en tête était :

$$\frac{\cos a}{\cos b} \tag{1.4}$$

Mathématiquement, écrire ceci c'est une hérésie mais nous allons quand même le tester. On pourrait alors comparer les résultats avec un résultat connu et évaluer notre approximation.

Après tests sur machine :

Nous avons testé les deux formules sur la valeur $\frac{\pi}{3}$ (convertie en nombre rationnel). Nous devrions donc trouver comme résultat : 0.5. On se rend compte en premier lieu, comme attendu, que l'approximation (1.4) est totalement fausse (qui ne tente rien n'a rien), ce qui confirme notre intuition. Nous l'avons donc retiré de notre code. Par ailleurs, pour coder la première formule (7), nous avons transformer le ratio en double avant de lui appliquer le cosinus. Lorsque nous comparons le résultat de la fonction avec le résultat exact nous observons une différence de 1.10^{-8}. Ce résultat n'est pas trop mauvais mais il n'est pas très bon non plus, nous perdons en précision. N'ayant pas trouvé d'autre manière de faire un cosinus d'un nombre rationnel donc nous devrons nous contenter de ce résultat.

3. $\sin(\frac{a}{b})$

Nous rencontrons logiquement le même problème que pour le sinus. Cependant, nous avons une piste de réflexion mathématique pour les angles très petits, car une formule particulière s'applique dans ce cas :

$$\sin\theta \approx \theta \tag{1.5}$$

Nous pourrions essayer d'implémenter cette approximation pour les nombres rationnels très petits. Pour cela, il faudra déterminer à partir de quel moment le nombre rationnel est assez petit pour appliquer cette formule et cela implique de faire un test à chaque fois que l'on veut utiliser le sinus sur un nombre rationnel. De plus, il faudra vérifier si le gain de précision est suffisamment important pour qu'intégrer cette formule dans le programme vaille le coup. Malheureusement par manque de temps, nous n'avons pas eu le temps de l'implémenter.

Après tests sur machine :

Nous avons testé les deux formules sur la valeur $\frac{\pi}{6}$ (convertie en nombre rationnel). Nous devrions donc trouver comme résultat : 0.5. La différence entre l'implémentation du sinus et sa valeur mathématique, a le même ordre de grandeur que pour le cosinus. Ce résultat est parfaitement logique. Nous avons également implémenté la tangente en utilisant le cosinus et le sinus.

4. $(\frac{a}{b})^k$

Ici on sait qu'on a une forme de x^n donc :

$$(\frac{a}{b})^k = \frac{a}{b} * (\frac{a}{b})^{k-1} \tag{1.6}$$

comme ce qu'on a pu faire en cours, on peut s'imaginer un algorithme récursif pour calculer cet opérateur suivant ce pseudo code suivant :

Algorithm 1 Algorithme de puissance de rationnels : pow(int k)

Require: $k \geq 0$, Ratio r
  if k > 0 **then**
    return $r * pow(k-1)$
  else
    return 1
  end if

La deuxième solution est de prendre le dénominateur et le numérateur séparément comme ceci :

$$\frac{a^k}{b^k} \tag{1.7}$$

Pour cela, nous allons utiliser la librairie cmath. Nous pensons que ça sera intéressant de comparer l'algorithme récursif à ce qu'est capable de faire la librairie. Nous comparerons ici aussi comparer les résultats en prenant des nombres connus.

Après tests sur machine :

Nous avons testé d'abord avec une valeur simple : $(\frac{1}{4})^3$. Le résultat à obtenir est donc 1/64. L'algorithme récursif marche très bien ainsi que la librairie cmath pour la formule (1.7). Mathématiquement cela est logique.

En augmentant la puissance, aucune différence apparaît entre les deux fonctions. A partir de la puissance 16, on a $\frac{1}{inf}$ pour les deux fonctions. On peut donc en conclure que les deux algorithmes peuvent être utilisés, choisir l'un ou l'autre n'aura pas d'impact sur le résultat. Pour choisir, nous pouvons donc regarder la complexité de chaque fonction. On se rend compte que les deux fonctions ont une complexité $\mathcal{O}(k)$ donc nous regardons le temps à l'exécution de chaque fonction. On remarque que la fonction utilisant la librairie cmath est 10 fois plus rapide que notre algorithme récursif.

5. $e^{\frac{a}{b}}$

Pour coder l'exponentielle nous pouvons utiliser la librairie <cmath>. Mais il est aussi possible de le coder nous même en utilisant la formule suivante :

$$exp(x) = \sum_{n=0}^{+\infty} \frac{x^n}{n!} \tag{1.8}$$

La fonction factorielle n'est pas présente dans les librairies cmath et math.h, nous devrons donc la coder. Cela se fera aisément de manière récursive. La fonction puissance est déjà implémentée. Ne pouvant pas faire la somme jusqu'à l'infini, nous ferons une boucle while qui s'arrêtera lorsque la différence entre deux termes consecutifs sera inférieur à un seuil fixé.

Après tests sur machine :

Lorsqu'on code la fonction exponentielle dans notre librairie, elle ne donne malheureusement pas le bon résultat. Nous n'arrivons pas à toruver la source de notre erreur mais une autre idée nous est venue. En effet, avec les propriétés de l'exponentielle, nous savons que :

$$e^{ab} = (e^a)^b \tag{1.9}$$

Donc que :

$$e^{\frac{a}{b}} = (e^a)^{b^{-1}} \tag{1.10}$$

Nous pouvons essayer implementer cette nouvelle fonction, pour cela nous utilisons la fonction exponentielle et puissance de la stl. La fonction s'implémente bien comme ceci, les résultats sont cohérents. Les tests unitaires passent.

6. $\ln(\frac{a}{b})$

Le logarithme d'un rationnel est facile à calculer grâce aux propriétés de ce dernier. On sait donc que :

$$\ln(\frac{a}{b}) = \ln(a) - \ln(b) \tag{1.11}$$

Nous pouvons utiliser les fonctions de la librairie cmath pour coder cette fonction.

Après tests sur machine :

Cette façon de faire marche très bien et est validée par les tests unitaires.

7. $E(\frac{a}{b})$

Pour cet opérateur, on calcule le nombre float du ratio et on le caste en int. Ensuite on renvoie le int obtenu sur 1.

$$Ratio\ \frac{a}{b} \quad -> \quad float\ c \quad -> \quad return \quad \frac{c}{1} \tag{1.12}$$

Nous n'avons pas eu de problème à l'implémentation de cette fonction, qui marche correctement.

1.2 Convertir un float en nombre rationnel

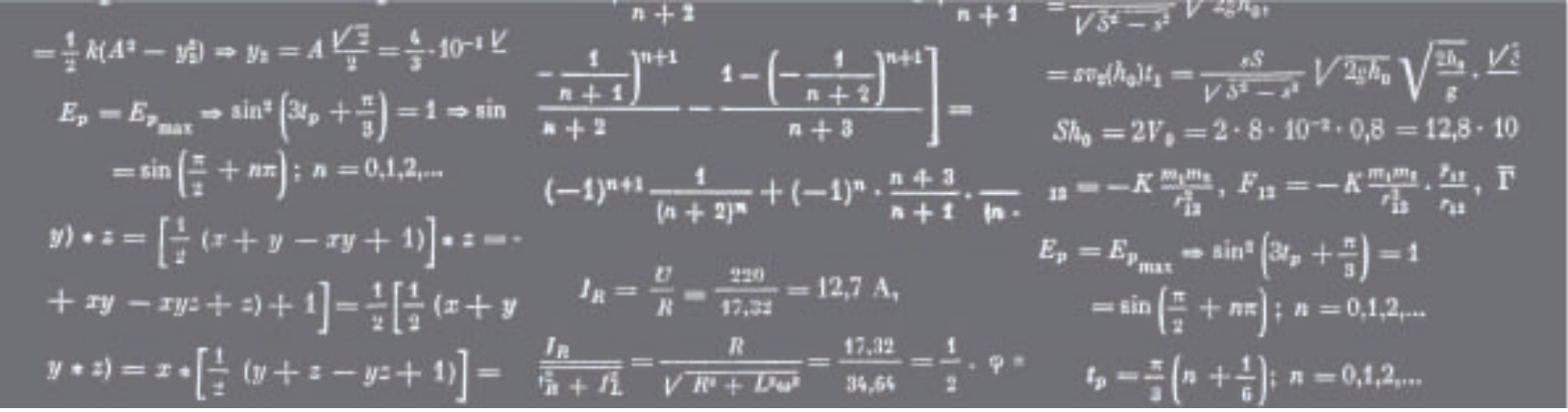
Pour pouvoir créer un nombre rationnel à partir d'un float il a fallu implémenter une fonction prenant en paramètre un float et le convertissant en rationnel. Pour cela, dans le sujet était donné un pseudo-code. Nous avons adapté celui-ci en C++ et ajouté une gestion de nombre négatif. Pour cela, nous avons modifié le peusdo code comme ceci :

Algorithm 2 Conversion d'un nombre réel en rationnel

Require: x : réel à convertir en rationnel
Require: *nb_iter* : le nombre d'appels récursifs
  Create a Ratio r = $\frac{0}{1}$
  if x == 0 || nb$_i ter$ == 0 **then**
    return r;
    if valeur absolue(x) < 1 **then**
      if x < 0 **then**
      r = ConvertFloatRatio($\frac{-1}{-x}$, *nb_iter*)$^{-1}$
      else
      r = ConvertFloatRatio($\frac{1}{x}$, *nb_iter*)$^{-1}$
      end if
      return r;
    else
      q = partie entière(x)
    return $\frac{q}{1}$ + ConvertFloatRatio(x-q, nb$_i ter - 1$);
    end if

Lors de la création de cette fonction, nous nous sommes vite rendu compte que le nombre d'itération de la recursivité pour créer notre rationnel était déterminant dans la précision du rationnel, mais aussi dans l'aspect final du rationnel. En effet, plus le nombre d'itérations était grand, plus le rationnel était précis mais plus il était non présentable.

Pour essayer d'améliorer ce rendu, nous avons poussé la réfléxion autour de l'arrondit et de la troncature du float. En effet, nous avons été confronté au problème de stockage des floats. Après 7 ou 8 chiffres après la virgule, les nombres n'étaient plus significatifs et perturbaient donc juste notre création de rationnel. Sachant que la fonction est utilisé dans pleins d'opérateurs, dans les tests unitaires etc., il était primordial d'avoir une fonction de conversion la plus fiable et propre possible. Après plusieurs essais, la troncature n'amélioraient pas nos résultats voire les empiraient. Le problème d'inexactitude des nombres float n'a malheureusement pas pu être résolu. Ce problème est d'autant plus remarquable et génant pour les très grands ou très petits nombres. A cause de cela, nous ne pouvons pas assuré que les converstions se fassent avec exactitude : nous perdons de l'information à chaque conversion. Notre classe n'est donc pas adaptée lorsqu'il s'agit d'utiliser un trop grand normbre d'opérateurs d'opérteurs, notamment les opérateurs qui font intéragir un ratio avec un autre nombre, surtout ceux à virgule floattante qui nécessitent une conversion. Bien qu'on essaie de palier à l'inexactitude des nombres à virgule floattant (exemple du début de sujet), celle-ci nous rattrape car notre classe Ratio est amenée à se baser sur un nombre à virgule floatante pour constituer des instances.



2. Partie programmation

L'objectif de la partie programmation était de créer une bibliothèque mathématique sur les nombres rationnels accompagnée d'exemple et de tests unitaires pour s'assurer de son bon fonctionnement.

2.1 Tableau bilan

Fonctionnalité	Demandé	Codé	Fonctionne
Constructeurs (défaut, paramétrique, copie)	Oui	Oui	Oui
Opérateurs basiques (+,-,*,/)	Oui	Oui	Oui
Opérateurs avancés (cos,sqrt,exp,ln,abs, partie entière)	Oui	Oui	Oui
Opérateurs de comparaison (<,<=,>,>=,=, !=)	Oui	Oui	Oui
Autres opérateurs (+=,-=,*=,/=)	Non	Oui	Oui
Tests unitaires	Oui	Oui	Pas tous
Compilation cmake	Oui	Oui	Oui
Classe template	Conseillé	Oui	Oui
Classe constexpr	Conseillé	Oui	Oui
Utilisation de Git	Oui	Oui	Oui
Librairie Doxygen	Oui	Oui	Oui
Namespace	Conseillé	Oui	Oui
Exceptions	Non	Oui	Oui
Apparence graphique	Non	Non (pas eu le temps)	Non

2.2 Les tests unitaires

Pour tester la validité de nos opérateurs, nous avons réalisé des tests unitaires sur les différents opérateurs que nous avons créés dans la librairie. Nous les avons repartis en plusieurs fichiers

correspondant aux différents opérateurs. Pour chacun d'eux, nous trouvons au moins un test unitaire avec des valeurs connues et un avec des nombres aléatoires. Pour certains, nous avons aussi fait des tests qui suivent des règles mathématiques.
Exemples :
- Diviser un ratio revient à le multiplier par l'inverse :

$$\frac{\frac{a}{b}}{\frac{c}{d}} = \frac{a}{b} * \frac{d}{c} \tag{2.1}$$

- Multiplier un ratio par son inverse donne 1 :

$$\frac{a}{b} * \frac{b}{a} = 1 \tag{2.2}$$

- La réciprocité de la fonction exponentielle et logarithme :

$$\ln(e^x) = x \qquad e^{lnx} = x \tag{2.3}$$

De plus, il est important de dire qu'étant donné que les tests unitaires utilisaient souvent la fonction convertFloatToRatio, leur niveau de précision changeait en fonction du nombre d'itérations que l'on faisait pour créer le ratio.

Les tests unitaires nous ont aussi fait soulever des problèmes mathématiques. C'est le cas pour la fonction exponentielle par exemple, où les valeurs obtenues peuvent monter très haut empêchant donc nos tests de passer. En regardant de plus près, nous avons remarqué que la différence entre la valeur attendue et notre résultat était très faibles pour des nombres pas trop grands. Dès que la valeur arrive à l'odre de 10^5, on observe des différences plus grandes qui font échouer nos tests unitaires. Malheureusement nous n'avons pas trouvé de solution pour résoudre ce problème...

2.3 Le namespace ratio

Dans l'idée que notre librairie pourrait être téléchargée et utilisée par d'autres personnes, nous avons mis notre classe Template dans un namespace appelé ratio. Cela change l'appel de nos méthodes qui sont désormais sous la forme de ***ratio : :method()***. Le namespace permet d'éviter les conflits avec les fonctions qui ne font pas parties de la classe, mais aussi de créer des nombres types de notre librairie 0, Pi et Infinite. Ceux-ci sont appelés de la manière suivante : ***ratio : :Ratio<int> : :pi()***

2.4 Les exceptions

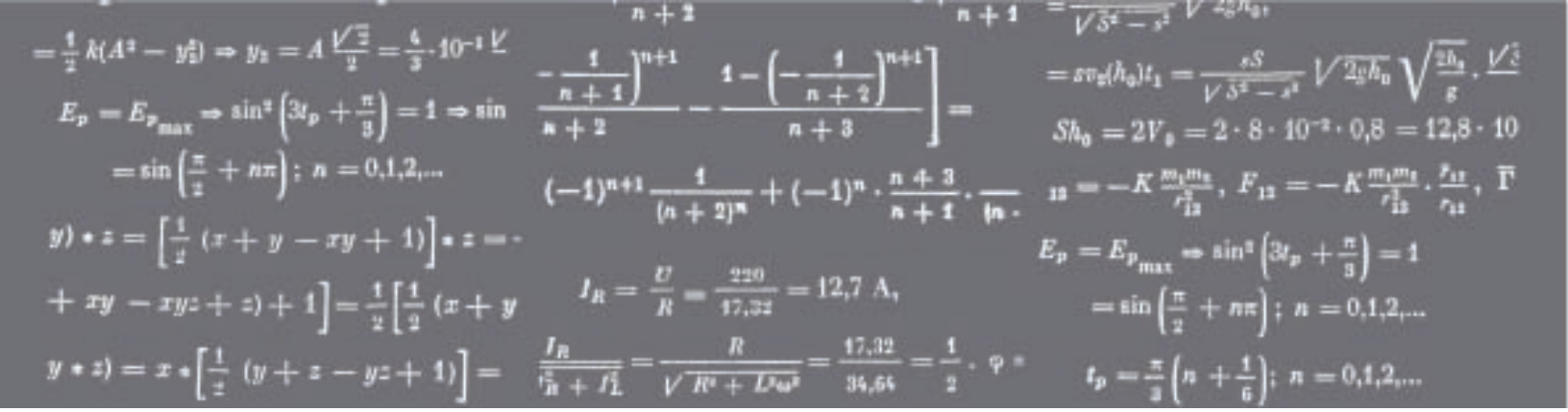
Pour avoir une meilleure gestion de l'utilisation de notre classe Template, nous avons créé des exceptions qui lui sont propres. Nous avons notamment conçu des exceptions qui s'appliquent lors de la construction du rationnel. Elles sont là pour vérifier les valeurs prises en entrée du constructeur. Ainsi, nous vérifions qu'aucun constructeur ne peut être appelé avec des valeurs différentes de deux integrals, d'un float et que son dénominateur soit toujours différent de 0. (Pour que notre rationnel infinite ($\frac{1}{0}$) puisse avoir un 0 au dénominateur, nous avons géré cette attribution directement dans la classe après sa construction.) En plus, nous avons aussi géré dans les exceptions l'interdiction d'avoir un nombre négatif dans la racine carré.

2.4.1 Multiplication par un float

Pour pouvoir multiplier notre ratio par une variable d'un autre type que le type du template T de notre classe, nous avons dû utiliser un autre template U correspondant au type de la valeur par laquelle nous voulions multiplier notre rationnel. Nous avons mis un peu de temps avant de comprendre cette application, mais avons tout de même fini par la maîtriser. En plus de cela, nous avons pu rajouter des exceptions sur le type de U attendu par les différentes fonctions.

2.4.2 Exception dans la classe Template

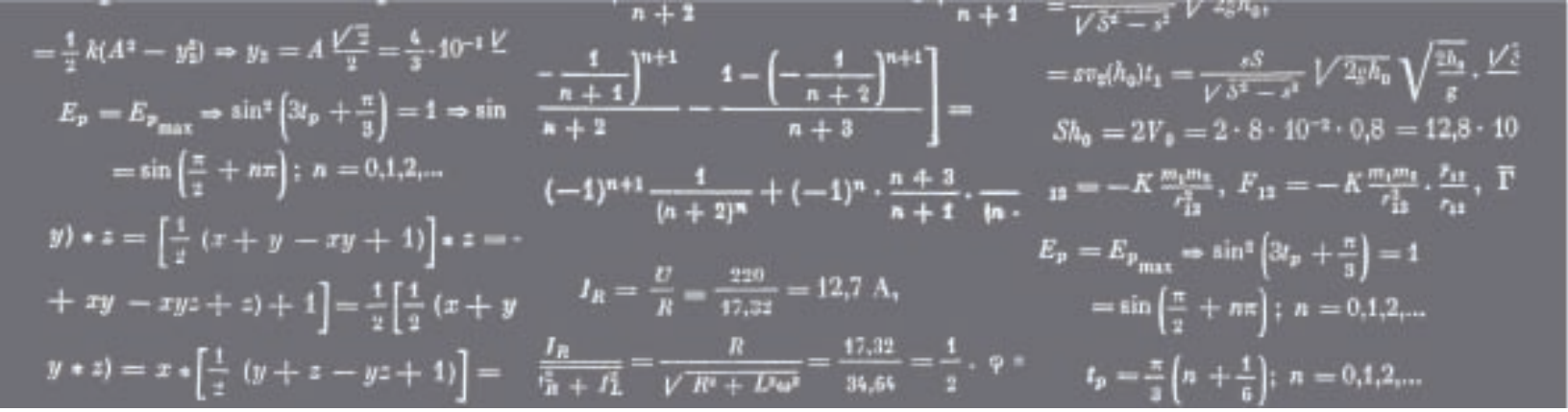
Pour aller plus loin dans notre classe Template, nous voulions interdire la création d'un rationnel à partir d'un string ou d'un char pour les constructeurs classiques. Cependant, nous avons eu quelques soucis. En effet,si nous appelons un Ratio<char> l'exception est bien lancée mais lorsque nous mettons un type char en paramètre mais spécifions que la ratio sera de type int ($Ratio < int > ('a','b')$), une conversion implicite est faite par le programme qui prend leur valeur ASCII. Comme nous disons qu'on veut des int et qu'une conversion est possible, alors le programme la fait. Cependant nous ne savions pas comment éviter cette conversion implicite. Dans la même idée, nous voulions gérer le cas d'appel du constructeur avec un seul paramètre x de type int pour créer un rationnel. Celui-ci nous donnerait : $(\frac{x}{1})$. Nous avons d'abord eu du mal à faire cette exception, mais grâce à cette difficulté nous avons pu résoudre d'autres problèmes liés à l'utilisation de plusieurs templates, ce que nous ne faisions pas mais qui était essentiel pour notre projet ! En passant notre nombre x en templat U (template dédié aux nombres extérieurs à la classe), nos exceptions ont finalement très bien fonctionné.



3. Ressenti du projet, points à améliorer ?

Ce projet nous a vraiment permis de comprendre comment une librairie était créée et fonctionnait. Il nous a fait mettre en application plein d'éléments que nous avions vu tout au long du semestre (classe, cmake, test unitaire, template, exceptions, ...) et par conséquent, nous avons réellement progressé. Notamment au niveau de l'utilisation des templates où nous avons rencontré des difficultés, qu'on a pu comprendre et résoudre, lors de ce projet. Il y a aussi eu une bonne évolution de nos compétences quant à l'utilisation de git.

En amélioration, nous pourrions rajouter une interface graphique par le biais d'une librairie comme OpenGl. Cela permettrait de faire des exemples plus visuels, mais aussi d'aider à l'utilisation de notre librairie. Cela faciliterait la compréhension de la librairie par l'utilisation, mais la rendrait aussi plus attractive. De olus, ayant implémenté les principaux opérateurs,notamment les opérateurs de comparaison, il est tout à fait possible d'aller plus loin et, par exemple, d'implémenter des algorithmes de tri, vus en cours en IMAC1 pour notre classe de rationnels.



Bibliographie

[1] Vincent Nozik, *Programmation Objet*. Cours IMAC2, 2022.

[2] Vincent Nozik, *Mathématiques pour l'informatique*. Cours IMAC2, 2022.

[3] cplusplus reference : `https://en.cppreference.com/w/`

[4] strackoverflow : `https://stackoverflow.com/`

[5] Microsoft, Documentation sur le langage C++ : `https://learn.microsoft.com/fr-fr/cpp/cpp/`