Rapport Info

Consignes pour faire marcher le programme :
cloner le répertoire libduturfu dans le git
supprimer le build
mkdir build
cd build
cmake ..
make

1.Partie mathématique

1.1. Opérations sur les rationnels

Question : Comment formaliseriez vous l'opérateur division?

On prend deux rationnels R_1 et R_2 et on veut faire $\frac{R1}{R2}$:

$R_1 = \frac{a}{b}$
$R_2 = \frac{c}{d}$

donc $\frac{R1}{R2} = \frac{a*d}{c*b}$

On oublie pas les deux conditions d'arrêts qui sont :
Si le numérateur de R_1 ou le dénominateur de R_2 sont égaux à zéro on retourne zéro.
Si $R_1 = R_2$ on retourne 1.

Sinon on return Rationnels $(\frac{a*d}{c*b})$
Pour les opérateurs + , - , ! ,* On applique la règle donnée dans l'énoncé.

Question : Vous pourrez aussi explorer les opérateurs suivants :
Dans toutes les fonctions on prendra $R_1 = \frac{a}{b}$

$\sqrt{\frac{a}{b}}$:

On a deux conditions d'arrêt :
Si a=0 on renvoie 0.
Si a et b valent 1 on renvoie 1.

Sinon, sachant que $\sqrt{\frac{a}{b}} = \frac{\sqrt{a}}{\sqrt{b}}$.

On renvoie donc Rationnels(sqrt(a)/sqrt(b)).
Assert : Il faut aussi prendre en compte qu'une racine carrée renvoie toujours un nombre positif.

$cos(\frac{a}{b})$, $sin(\frac{a}{b})$, $tan(\frac{a}{b})$:

On a rencontré plusieurs problèmes. On s'est vite rendu compte que faire le cosinus, sinus d'un rationnel n'était pas aussi facile que nous le pensions. Après quelques recherches on a décidé d'utiliser la fonction getRationnels() car c'était la solution la plus simple et la plus compréhensible pour nous. On a donc transformé notre rationnel en float pour pouvoir faire le cosinus et le sinus (donc aussi tangente car $tan = \frac{sin}{cos}$) de ce float puis on a transformé notre résultat en rationnel avec getRationnel().

Malheureusement, le fait de faire des allers-retours entre float et rationnel fait perdre de la précision. En effet, lorsqu'on transforme notre rationnel en float on fait un premier arrondi, puis, lorsqu'on transforme le résultat du cosinus, sinus ou tangente en rationnel on fait un second arrondi. Les résultats obtenus pour cosinus, sinus et tangente sont donc loin d'être précis.

Condition d'arrêt :
tan(0)=sin(0)=0
cos(0)=1
Il faut une exception pour la fonction tangente car il n'est pas possible de faire tan($\frac{\pi}{2}$).
On a également eu un problème avec cette exception car notre tangente arrondissait π à 3 donc l'exception ne fonctionnait pas.

$(\frac{a}{b})^{n}$:

La fonction puissance prend en paramètre un entier n (la puissance)
On sait que $(\frac{a}{b})^{n} = (\frac{a^{n}}{b^{n}})$
On retourne donc pow(a,n),pow(b,n)

Condition d'arrêt :
Si a=0 => R_1=0
Si n=0 => R_1=1

Logarithme :

La fonction logarithme nécessite deux exceptions :

- On ne peut pas faire le logarithme d'un nombre négatif donc (a>0, Sachant que le dénominateur doit toujours être positif)
- On ne peut pas faire le logarithme de 0

On sait que log($\frac{a}{b}$) = log(a) - log(b)
On retourne donc à l'aide de la fonction log : log(a)-log(b)

Problème rencontré :

Exponentielle :

Le problème avec la fonction exponentielle c'est qu'on obtient très vite des numérateurs très grands. On dépasse donc très rapidement la limite, ainsi, le programme renvoie vite zéro.
On a codé deux fonctions exponentielles, une qui utilise la fonction cmath, la plus simple, et une qui utilise la méthode horner. Hors la méthode horner nous renvoie des valeurs très peu précises. Elles fonctionnent mal, on a donc décidé d'utiliser la fonction cmath pour nos tests.

3.2 Conversion d'un réel en rationnel

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

Explication fonction get rationnel :
La fonction prend en paramètre un réel positif (x) et un entier correspondant au nombre d'appels récursifs restant (nb_iter).
On a deux conditions d'arrêts :
Si le réel entré est égal à zéro, alors le rationnel obtenu sera aussi zéro,
Si le nombre d'appels récursif est de zéro, alors le rationnel obtenu sera aussi zéro.

Les différents cas d'appels récursifs :
Pour x= 0.5
On a x<1,
convert_float_to_ratio($\frac{1}{0.5}$, nb_iter) renvoie 2 on met donc la puissance -1 pour obtenir $\frac{1}{2}$ pour plus de précision.

Pour x=2.5
Partie entière de x = 2
on retourne $\frac{2}{1}$ + 2.5 - 2 , ce qui équivaut à faire $\frac{partie\ entière}{1}$ + partie décimale (convertie en rationnel) après il suffit de trouver le plus grand dénominateur commun et mettre la fraction sous sa forme irréductible.

Question : Comment modifier le code pour qu'il gère également les nombres réels négatifs ?
De notre côté, lorsqu'un rentre un réel négatif, l'algorithme marche tout aussi bien.

3.3 Analyse

Question : 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 cela ?

Prenons l'exemple d'un nombre très très grand. Pour le représenter en rationnel on est soit obligés d'avoir un numérateur très grand soit un dénominateur très petit. Hors notre dénominateur doit nécessairement être entier donc il ne peut pas être entre zéro et 1. Donc le numérateur doit être très

grand. Cependant on a vu que lorsque le numérateur ou le dénominateur atteignent des valeurs trop grandes le programme renvoie zéro. Ainsi pour représenter un nombre très très grand notre classe de rationnel nous renverra très certainement zéro car le numérateur aura atteint la limite. Le même principe s'applique pour les nombre très petit, mais cette fois-ci c'est le dénominateur qui atteindra la valeur limite.

Question : 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 ?

En effet, on a remarqué que lorsque le numérateur ou le dénominateur atteignent des valeurs plus grandes, le programme renvoie zéro. On a donc tout d'abord changé les types en long et long long pour atteindre la valeur de non retour le moins vite possible. Cette solution est simple mais malheureusement elle ne fait que repousser le problème.

On a donc décidé de faire une fonction, qui, lorsque le numérateur ou le dénominateur atteint une valeur trop grande au lieu de revenir à zéro elle fait en sorte que le rationnel perte simplement en précision. En effet, si un des deux est trop grand il nous suffit de simplement diviser le numérateur et le dénominateur par un même nombre (ici 2) pour qu'il soit plus petit, jusqu'à ce que le plus grand des deux arrive en dessous de la valeur souhaitée.

Cela permet donc à notre classe Rationnels de ne pas dépasser la limite même en ayant un numérateur et un dénominateur représenté par des ints ou même des shorts.

C'est le rôle de notre fonction `void setNoOverflow(long long nume, long long deno);`

4. Partie Programmation

4.1 Les rationnels

Éléments demandés qui fonctionnent	Valeur absolue : La fonction valeur absolue ne change le signe de la fraction que si le numérateur est négatif (étant donné que nous avons fait en sorte que le dénominateur ne soit pas négatif) . En effet, si a et b sont négatifs on a : $\frac{-a}{-b} = \frac{a}{b}$ Partie entière : Pour nous, la solution la plus simple à adopter était d'utiliser la fonction floor sur notre

	rationnel et de retourner cette valeur. Elle marche aussi bien avec les rationnels négatifs que les rationnels positifs. Fonction pour transformer le rationnel sous forme de fraction irréductible : on a utilisé, comme suggéré dans l'énoncé, gcd. Dans notre constructeur, on initialise une variable gcd qui calcule le pgcd du numérateur et du dénominateur à l'aide de la fonction __algo_gcd puis on a divisé le numérateur et le dénominateur par le pgcd pour avoir une fraction irréductible. Nous n'avons pas fait une fonction à part, on a choisi de la mettre directement dans le constructeur comme ça quand on l'appel tout se fait en même temps, pas besoin d'appeler une fonction spécifique. Les opérateurs : Tout a été fait à partir de l'équation : On prend deux rationnels R_1 et R_2 et on veut faire $\frac{R1}{R2}$: $R_1 = \frac{a}{b}$ $R_2 = \frac{c}{d}$ $R_1 = R_2$ $\Leftrightarrow \frac{a}{b} = \frac{c}{d}$ $\Leftrightarrow a * d = c * b$ Pour les opérateurs <=,<,>,>= il suffit juste de comparer si $a * d$ et $c * d$. Pour les opérateurs == et != il suffit de comparer a avec c et b avec d directement, pas besoin de calcul supplémentaires. fonction d'affichage : Aucun problème, on a appliqué la méthode vue en TP. Le moins unaire :

	Aucun problème rencontré, il nous a suffit de d'inverser la valeur du numérateur de notre rationnel. Produit/Division nombre à virgule avec un rationnel : Utiliser *this permet d'être plus concis et ainsi ne pas avoir à écrire numerator/denominator. Exceptions : - Le logarithme de zéro n'existe pas - On ne peut pas faire le logarithme d'un nombre négatif - La racine carrée d'un nombre négatif n'existe pas (dans les réels) - On ne peut pas avoir un dénominateur égal à zéro (marche dans le cas de la fonction inverse aussi). ASSERT : Présents dans les tests unitaires. Outils de la STL : Dans les tests unitaires nous utilisons les vecteurs de la stl. Espaces de nommage
Éléments demandés et codés qui ne fonctionnent pas	
Éléments demandés mais pas codés	fonctions constexpr
Éléments non demandés et codés qui fonctionnent	Création de notre classe en template Les fonctions loosePrecision(), setNumerator(long long num), setDenominator(long long denum) pour faire en sorte de ne pas dépasser la limite et de ne jamais revenir à zéro (comme expliquée dans la partie mathématiques).. Utilisation de la fonction variadics generate dans les tests unitaires.
Éléments non demandés mais pas codés ou qui	

ne fonctionnent pas, mais pour lesquels on propose des solutions	

4.2 Les exemples

Tous les exemples sont situés dans le main() dans le dossier src de my_examples. ce fichier fournit une batterie d'exemple pour chaque opérateur et pour chaque fonction mathématiques proposées dans la classe rationnels. Chaque fonction est testée soit avec un rationnel positif, un rationnel négatif ou un rationnel nul. Le main() utilise des exemples simples avec des calculs dont nous connaissons tous la réponse pour pouvoir comprendre et vérifier que la fonction marche plus rapidement.

4.3 Les tests unitaires

Les tests unitaires sont situés dans le dossier UnitTest/src le fichier test.cpp. Pour les test nous avons utilisés la libraire gtest. Chaque block de tests servant à vérifier la fonctionnalité d'une partie de notre code.

```
TEST (RationnelsConstructor, floatConstructor) {

   int vectorSize = 500;
   std::vector<float> testers(vectorSize);
   srand(12);
   std::generate(testers.begin(), testers.end(),[](){return ((static_cast <float>
(rand()) / (static_cast <float> (RAND_MAX/1000)))-200);});

   Rationnels<int> ratio;

   for(int i = 0; i<testers.size(); i++){
       ratio = Rationnels<int>(testers[i]);
       ASSERT_NEAR (testers[i], ( ratio.numerator/(float)ratio.denominator),0.001);
   }
}
```

Voici un exemple type de block utilisé pour nos tests, chaque block suivant la même structure. On génère un vecteur de float aléatoire de la taille souhaité ici 500, grâce à la fonction generate. En incluant aussi les nombres négatifs. Puis nous parcourons le vecteur de float en comparant la valeur attendu à la valeur obtenue (ici nous vérifions si notre classe rationnel converti bien les nombres à virgule en rationnel), avec une certaine marge d'erreur, ici à 10^{-3} près.
Ici nous avons choisi d'utiliser un Rationnel de type int car les résultats obtenus était satisfaisant, mais dans d'autre cas nous avons gardé le type long long, pour avoir un résultat plus précis.