ÉLABORATION D'UNE CALCULATRICE VIRTUELLE

Travail de maturité

Suivi par M. Cherix Pierre-Alain
Collège Rousseau, Genève
Octobre 2014

Nguyen Phuc-Thien Thomas
Groupe 408 (collège Sismondi)

Table des matières

1. Pré-conception

Note Le document est écrit en suivant les recommandations de 1990 du français. Le traitement de texte est LaTeX. Version originale soumise pour le travail de maturité.

1.1 Introduction

1.1.1 Pourquoi ce sujet

Tout le monde me disait que ça allait être chaud de faire un tel travail de maturité. Alors pourquoi l'ai-je choisi ? Tout d'abord, il faut savoir qu'à la base, je souhaitais étudier un autre sujet, dont j'en avais déjà parlé à certains : le Bitcoin. Il s'agit d'une monnaie électronique innovante, qui ne dépend d'aucune banque. Pour cela, le système fait appel à la cryptographie. Je vous renvoie au site officiel [1] si vous désirez en savoir plus. Ce sujet m'intéressait beaucoup, mais aucun professeur n'était prêt à me suivre pour ça, j'ai donc dû y renoncer.

Ainsi ai-je donc choisi d'élaborer une calculatrice virtuelle. Cette idée m'est venue, parce que je suis un répétiteur en mathématiques, physique, et chimie, et je me suis dit qu'il serait pas mal que je conçoive un outil très pratique pour les collégiens en mathématiques, qui les aide à effectuer leurs devoirs en vérifiant leurs réponses, par exemple. Bien évidemment, le produit qui sera créé d'ici la fin de ce travail de maturité ne couvrira tous les besoins du collégien, le but étant plutôt de poser les bases d'un projet qui aboutira plus tard.

1.1.2 Objectifs de ce travail de maturité

Avant de commencer, regardons ensemble les idées générales sur lesquelles est fondé le développement du programme. Tout d'abord, le but premier est évidemment d'élaborer une calculatrice virtuelle, c'est-à-dire une calculatrice utilisable sur ordinateur. Maintenant, il faut savoir de quelle sorte de calculatrice on parle.

Vous connaissez les calculatrices qu'on trouve partout, ces logiciels ou machines qui peuvent effectuer divers calculs, avec les quatre opérations de base (addition, soustraction, multiplication, division), et quelques autres comme les puissances ou la racine carrée. Les plus avancées d'entre elles permettent de calculer une chaine d'opérations, par exemple $6.67 \times 10^{-11} \frac{5.97 \times 10^{24}}{(6.37 \times 10^6)^2}$. Si vous tapez :

```
(6.67*10^-11)(5.97*10^24)/(6.37*10^6)^2
```

La calculatrice vous répondra 9.81344..., avec plus ou moins de chiffres selon sa précision. C'est très bien, mais c'est un genre de calcul que beaucoup de calculettes savent faire (calculatrice par défaut de `Windows 7`, celle d'`Ubuntu`,...). En cherchant sur le net, vous en trouverez un certain nombre, et pareil dans le monde réel si vous prenez ces calculatrices avec boutons. Par conséquent, il serait intéressant de créer une calculatrice qui propose des fonctions un peu plus exotiques, plutôt que de proposer quelque chose qui existe déjà.

1. `https://bitcoin.org/`

L'une des principales fonctions supplémentaires serait la gestion des polynômes, donc permettre de stocker des polynômes, de les évaluer, de faire des sommes et produits de polynômes... La calculatrice permettrait également de faire des représentations graphiques. Une calculatrice proposant cela est déjà un peu plus difficile à trouver.

On pourrait également gérer les nombres exacts. Par exemple, on ne stockera pas 0.333333, mais $\frac{1}{3}$. On gérera nativement les nombres rationnels, et les nombres rationnels complexes. Le cas des nombres appelés transcendants, comme π ou e, sera laissé de côté, car ils sont très difficiles à implémenter.

Une autre idée m'est venue, celle de gérer des nombres arbitrairement grands. En informatique, il faut savoir que la taille maximale d'un nombre est limitée. Vous avez surement été confronté au message « Overflow error » de votre calculatrice, en essayant de calculer 10^{1000}, par exemple. Cela signifie que le nombre que vous auriez dû obtenir dépasse les capacités de votre machine, pour des raisons de mémoire, de codage, ou arbitraires. Les limites les plus courantes sont 10^{100} sur une calculatrice scientifique d'écolier, et $2^{1024} \approx 1.8 \times 10^{308}$ sur les ordinateurs ou téléphones intelligents (smartphones).

Avec ces fonctions, nous aurons déjà une calculatrice qui se démarque des autres. En plus de toutes ces idées, nous pourrions proposer quelques fonctions supplémentaires, comme le calcul sur des cubiques, ou bien des fonctions cryptographiques qui profitent que les nombres arbitrairement grands soient gérés. Mais en raison de la contrainte du temps, nous en resterons là.

1.2 Préparatifs

1.2.1 Choix du langage de programmation

Pour concevoir un logiciel, la première chose à faire après le choix du sujet, est de choisir le langage dans lequel on va le coder. Un langage de programmation est un ensemble de règles syntaxiques permettant de créer un programme informatique. Il y a plusieurs langages de programmation, comme il y a beaucoup de langues dans ce monde.

J'ai choisi le `C++`, pour plusieurs raisons. Premièrement, il s'agit d'un langage multiplateforme. Le `C++` a été conçu pour pouvoir fonctionner sous n'importe quel système d'exploitation sur ordinateur : avec peu d'efforts, on peut faire fonctionner un même programme sous Linux, Windows ou Mac. En plus, il s'agit d'un langage de programmation très répandu, donc on peut trouver beaucoup de ressources pour l'apprendre, ou pour demander de l'aide. Pour ma part, je l'ai entièrement appris en autodidacte, et c'est d'ailleurs le premier langage que j'ai appris.

Ensuite, il s'agit d'un langage très rapide ! En effet, il est ce qu'on appelle un langage de bas niveau, c'est-à-dire un langage qui est relativement proche du langage machine : cela permet la rapidité d'exécution. Bien sûr, on reste à des années lumières du langage assembleur, mais le fait que ce soit un langage de bas niveau, outre la vitesse d'exécution, permet d'effectuer des choses que d'autres langages comme `Python` ou `Visual Basic` ne peuvent pas faire.

De plus, étant un langage compilé (le code source est une fois pour toutes traduit en langage binaire compréhensible par l'ordinateur), et non interprété (le code est traduit durant son exécution), vous ne dépendez pas ce qui est appelé la « machine virtuelle » qui se charge de traduire les instructions pour l'ordinateur pendant l'exécution. Il arrive souvent que des bogues dans la machine virtuelle `Java` soient découverts, vous avez peut-être fait l'expérience de plugins `Java` automatiquement désactivés à cause de ça. Cela n'arrivera jamais en `C++`.

Il y a également une chose très importante : en plus d'être un langage de bas niveau, le `C++` propose également des fonctionnalités de haut niveau, et tout particulièrement la programmation orientée objet. Pour ceux qui ne sont pas spécialistes en programmation, il sera difficile d'expliquer cet argument. Mais pour les autres, cette fonctionnalité est très importante pour la conception de cette calculatrice. Gérer les nombres arbitrairement grands nécessite de définir de nouveaux types de nombres, ce qui est très difficile sans le concept de programmation orientée objet. Le langage `C` ne propose pas une telle fonction, ou en tout cas pas d'une manière suffisamment poussée pour ce projet.

Bien sûr, il existe aussi des inconvénients à ce langage. Le plus connu d'entre eux est sa réputation d'être un langage très difficile, mais offre en contrepartie des possibilités très avancées, et beaucoup de liberté. Cette liberté est justement un double tranchant, du fait qu'on peut facilement écrire un code qui fait n'importe quoi, sans qu'on ne s'en rende compte. Un autre problème est le fait que les erreurs affichées lors de la compilation, s'il y en a, sont parfois difficiles à comprendre. Mais avec la pratique, on apprend à les déchiffrer rapidement.

1.2.2 Environnement de développement

Pour taper le code, il existe deux principales solutions :

- Utiliser un éditeur de texte pour écrire le code source dans des fichiers, et télécharger séparément les outils nécessaires pour générer le programme à partir de ces fichiers. Pour cela, il faut configurer chaque outil utilisé pour obtenir le résultat qu'on cherche.
- Utiliser ce qu'on appelle un EDI (pour *environnement de développement intégré*) [2], intégrant un éditeur de texte, et plein d'autres outils qui feraient le travail de la solution précédente. On est moins libre dans les options.

Étant une personne qui préfère la légèreté (les EDI sont des programmes relativement lourds), et n'utiliser que ce dont j'ai besoin (je n'utiliserai pas une majorité des fonctions proposées par les EDI), j'ai opté pour la première solution. Les programmes seront écrits sous Linux (`Debian 7 LXDE`), à l'aide de l'excellent éditeur de texte `Gedit` (qui permet par exemple la coloration syntaxique, et intègre un explorateur de fichiers). Le compilateur sera g++, et on utilisera également `Make` et `CMake` pour faciliter la compilation. Le but sera de créer un programme qui fonctionne sur Linux 32 et 64 bits, et Windows 32 bit (la compilation en 64 bits est plus compliquée). Sous Windows, la compilation se fait avec `MinGW` et `MSYS`.

1.2.3 Quelques notions de programmation

Cette section vous présentera succinctement quelques concepts utilisés pour programmer, dont la compréhension est recommandée pour la lecture de la suite. Il est fait en sorte d'être le plus bref possible pour ne pas rebuter les débutants, ou ceux qui ne sauraient pas programmer, mais qui seraient intéressés d'étudier le fonctionnement du logiciel.

Variables

Dans un programme, on utilise très souvent, voire toujours, ce qu'on nomme « variables ». Il s'agit d'un nom associé à une valeur qui peut varier au fil de l'exécution d'un programme. Par exemple, on peut imaginer une variable appelée « distance » qui stocke la distance entre deux objets en mouvement, ou encore « nom » pour stocker votre nom. Ce qui est stocké

2. Souvent appelé IDE d'après le terme équivalent anglais « Integrated development environment ».

est appelé la *valeur* de la variable. La valeur d'une variable est temporairement stockée dans la mémoire de l'ordinateur, jusqu'à ce qu'on en ait plus besoin.

Il existe des variables de plusieurs *types* : on peut stocker des nombres entiers, des nombres à virgule, des lettres, ou encore des mots et des phrases (chaines de caractères). Tous ces types portent un nom, respectivement `int`, `double`, `char`, et `string`.

En `C++`, on doit *déclarer* les variables avant de les utiliser. La déclaration se fait en écrivant sur une ligne le type de la variable, suivi de son nom. Par exemple, la variable « distance » aurait pu être déclarée ainsi en `C++` :

```
1 double distance;
```

On pourra alors stocker des nombres à virgule (type `double`) pour cette variable appelée distance. Le point-virgule est nécessaire à chaque fin d'instruction, en l'occurrence la déclaration d'une variable. Il est également possible de déclarer plusieurs variables en même temps :

```
1 int x, y;
```

Ici, on a déclaré deux variables appelées x et y pouvant stocker des nombres entiers avec signe (type `int`). On peut ensuite leur *affecter* une valeur par la suite :

```
1 int x, y;
2
3 x = 2;
4 y = 3;
```

Ce qui est aussi possible lors de la déclaration :

```
1 int x(2), y(3);
```

À noter qu'il existe le mot-clé `unsigned`, qui permet d'interdire l'usage de nombres négatifs (utile par exemple pour stocker des longueurs qui sont toujours positives). On le met devant le type pour l'utiliser (par exemple, `unsigned int`). Il n'existe que pour les nombres entiers (on ne peut donc pas avoir de « `unsigned double` »).

Fonctions

En programmation, une *fonction* (aussi appelée *procédure*) permet de renvoyer une valeur en fonction de paramètres. Elle doit avoir un nom, des paramètres (types et nom), un type de retour, et les instructions qui permettent de générer la valeur retournée.

Par exemple, une fonction appelée « additionner » qui additionne deux nombres entiers (`int`) peut être codée ainsi :

```
1 int additionner(int x, int y) {
2     int resultat;
3     resultat = x + y;
4     return resultat;
5 }
```

Ainsi, nous avons le nom de la fonction (`additionner`), suivi des arguments entre parenthèses (appelés `x` et `y`, et tous les deux de type `int`), précédé du type de retour (également un `int`), et les instruction entre les accolades `{}` qui permettent de créer le résultat, et de le renvoyer avec `return`. Une fois une telle fonction créée, on peut écrire par exemple :

```
1 int res;
2 res = additionner(8, 13);
```

On aura que `res` sera égal à 21. Ou encore :

```
int x(2), y(3);
int res;
res = additionner(x, y);
```

`res` vaudrait alors 5.

À noter qu'il n'est pas possible de retourner plus d'une valeur en `C++` (ce qui se contourne avec la notion des références qui ne sera pas présentée ici). Il est par contre possible de ne retourner aucune valeur quand on n'en a pas besoin, en spécifiant `void` comme type de retour.

Classes

Les *classes* permettent de créer ses propres types de variables, et sont la base de la programmation orientée objet. Il s'agit formellement de propriétés communes à un ensemble d'*objets* (aussi appelés *instances* de la classe). Ces propriétés se composent d'*attributs*, qui sont des variables en commun de chaque objet de la classe, et les *méthodes*, qui sont des fonctions agissant sur ces attributs.

Par exemple, si on souhaitait stocker des dimensions de rectangles, on pourrait créer une classe appelée « Rectangle » (qui n'existe pas par défaut en `C++`), qui aurait pour attributs une longueur et une largeur (tout rectangle possède ces deux dimensions). On pourrait ensuite ajouter des méthodes permettant par exemple de connaitre (*accesseurs*) ou changer (*mutateurs*) ces dimensions, de calculer l'aire, le périmètre,... En `C++`, on aurait écrit cette classe de la manière suivante :

```
class Rectangle {
    private:
    double longueur, largeur;

    public:
    Rectangle();
    Rectangle(double l, double L);

    double connaitreLongueur();
    void changerLongueur();
    double connaitreLargeur();
    void changerLargeur();

    double calculerAire();
    double calculerPerimetre();
};
```

Les contenus des méthodes (`connaitreLongueur()`, etc.) sont définis ailleurs dans le code, on se contente généralement de juste déclarer ces fonctions dans la classe (ce sont des *prototypes*). `Rectangle()` et `Rectangle(double l, double L)` sont des fonctions particulières, appelées *constructeurs*, qui permettent d'attribuer respectivement une valeur par défaut et une valeur manuelle, comme on l'a fait pour `int x(2), y(3)`.

Ne vous préoccupez pas trop des mots clés `private` et `public`, utilisés en programmation orientée objet dans le but de l'*encapsulation*, qui est un concept un peu long à expliquer de manière simple pour tenir ici.

À l'aide d'une telle classe, on peut donc faire :

```
Rectangle r(3, 4);
double aire;
aire = r.calculerAire();
```

La variable `aire` vaudra donc 12 si on a correctement codé la classe et les méthodes. On fait `r.calculerAire()` pour utiliser la méthode de l'objet de type `Rectangle`.

Patrons de classes

Comprendre ce concept assez compliqué du `C++` n'est pas obligatoire pour la suite, vous pouvez vous en passer si vous estimez les classes déjà assez difficiles.

Un *patron de classe*, également appelé « template », est un modèle générique permettant de générer automatiquement des classes (par le compilateur). Vous comprendrez mieux avec l'exemple concret suivant : reprenons notre classe `Rectangle()`. Nous l'avions définie avec deux attributs de type `double` pour stocker ses dimensions : `double longueur, largeur;`. Mais imaginons que, pour une raison ou une autre, on souhaite gérer en plus des rectangles avec seulement des longueurs entières, donc des attributs de type `unsigned int`. Que faire ? Recopier la classe et toutes les méthodes qui ont été définies, et remplacer les `double` par des `unsigned int` ?

Les patrons de classe permettent d'éviter cela, en créant une seule classe pour toutes ! On peut alors réécrire la classe `Rectangle` ainsi :

```
1  template <class T> class Rectangle {
2      private:
3      T longueur, largeur;
4
5      public:
6      Rectangle();
7      Rectangle(T l, T L);
8
9      double connaitreLongueur();
10     void changerLongueur();
11     double connaitreLargeur();
12     void changerLargeur();
13
14     double calculerAire();
15     double calculerPerimetre();
16 };
```

La seule différence étant que les `double` ont été remplacés par un type quelconque `T` (il est possible de le nommer comme on le souhaite), et on indique qu'il ne s'agit plus d'une classe, mais d'un patron de classe, à l'aide de `template <class T>` au début. Nous avons maintenant un patron de classe, qui s'utilisera ainsi :

```
1 Rectangle<double> r1(2.5, 3.5);
2 double aire1;
3 aire1 = r1.calculerAire();
4
5 Rectangle<unsigned int> r2(3, 4);
6 double aire2;
7 aire2 = r2.calculerAire();
```

On a le nom du patron de classe, suivi par la « version » souhaitée de ce patron entre chevrons, ce qui nous donne `Rectangle<double>` pour stocker des rectangles à dimensions pas forcément entières, et `Rectangle<unsigned int>` pour stocker des rectangles à dimensions entières. Si tout a été bien programmé, `aire1` devrait valoir 8.75, et `aire2` 12.

Le patron de classe `vector` proposée en standard par le `C++` sera beaucoup utilisée.

Boucle for

En programmation, les boucles permettent de répéter une instruction donnée. Il existe plusieurs types de boucles, mais il y en a une qui sera particulièrement utilisée, la boucle `for`. Nous allons nous contenter d'un exemple pour comprendre à quoi elle sert, et comment l'utiliser : supposons que nous voulions additionner les cent premiers entiers naturels entre eux, de 1 à 100.

```
1 unsigned int somme(0);
2
3 for (unsigned int i(1) ; i <= 100 ; i++) {
4     somme += i;
5 }
```

La variable `somme` stocke la somme qu'on cherche. La boucle `for` se compose premièrement d'une condition (entre parenthèses), dans laquelle il y a un compteur (une variable temporaire i qui a été déclarée), la condition proprement dite (`i` doit être inférieur ou égal à 100), et une incrémentation (le fait d'ajouter une unité à une variable, indiqué par `i++`). Et deuxièmement, entre accolades, l'ensemble des instructions à répéter.

L'exécution de la boucle se passe de la manière suivante : on met une valeur initiale au compteur, ici 1. L'ordinateur vérifie ensuite si i est inférieur ou égal à 100, en l'occurrence oui. Donc, on peut exécuter l'ensemble des instructions : ajouter la valeur du compteur `i = 1` à la variable `somme`, et affecter la nouvelle valeur (avec l'opérateur `+=`). Après l'exécution de cette première itération de la boucle, `somme` vaut donc 1. Et après avoir exécuté cette instruction, on ajoute une unité au compteur `i`.

Puis, on revient au point de départ, mais cette fois avec `i = 2`. Ici, le compteur est toujours inférieur ou égal à 100, et donc, on peut répéter l'instruction entre les accolades : `somme` vaudra $1 + 2 = 3$ après la seconde itération. On incrémente à nouveau le compteur, on recommence la boucle, et ainsi de suite... Jusqu'à ce que nous arrivons à l'itération où `i = 101` : cette valeur étant supérieure à 100, la boucle s'arrête, et l'exécution du programme continue. À ce moment, `somme` vaudra l'addition de tous les `i`, qui ont valu de 1 à 100, à savoir 5050.

Bon. Maintenant, nous pouvons vraiment commencer. Dans les chapitres qui suivent, vous aurez un aperçu des étapes et idées qui ont permis de produire le produit final, et un survol des fonctionnalités proposées par mon programme. Tout sera expliqué de sorte à être le plus accessible possible, surtout pour ceux qui seraient intéressés à se mettre à programmer. On restera sur l'essentiel : pas tous les détails seront exposés. De plus, certaines explications seront simplifiées, et ne correspondront pas à 100% au code source, car elles sont très difficiles à expliquer, ou prendraient beaucoup de texte. Par exemple, les implémentations sous forme de patrons de classes seront expliquées comme si elles étaient de simples classes.

2. Nombres et polynômes

2.1 Organisation du code, annexes

Dans un premier temps, j'ai trouvé judicieux de séparer le codage en deux parties : d'un côté, tout ce qui touche aux opérations. De l'autre, tout ce qui concerne l'interface utilisateur-machine.

Dans la première partie, on aura par exemple le codage des nombres, et la définition des opérations, ainsi que d'autres fonctions en rapport avec les nombres comme les arrondis ou d'autres calculs. Dedans sera également codé la gestion des polynômes et l'analyseur syntaxique. L'ensemble des fonctionnalités compose alors une bibliothèque `C++` que j'appelle `Algébra`, comme « algebra » en anglais, mais avec un accent pour indiquer que le développeur est francophone. J'ai eu cette idée, parce qu'il se pourrait qu'à l'avenir, d'autres personnes soient intéressées par les fonctions proposées, et plutôt que mélanger l'interface et les mécanismes internes, il serait bien d'isoler ces derniers pour permettre à d'autres développeurs de concevoir des programmes scientifiques se basant sur ma bibliothèque...

Dans la partie interface utilisateur-machine, il y aura le codage de l'interface graphique, qui permettra à ceux qui utilisent mon programme de taper leur chaine d'opérations, et d'effectuer des calculs. Il se chargera d'afficher les résultats, et c'est aussi lui qui permettra la représentation graphique des polynômes. Tout l'ensemble forme `PolyCalc`. En résumé, la calculatrice `PolyCalc` est basée sur la bibliothèque `Algébra` qui fournit les engrenages qu'on ne voit pas.

Ce dossier est accompagné du manuel d'instructions de `PolyCalc`. Dans le support informatique, il y a la documentation d'`Algébra` destinée aux développeurs, ainsi que les sources et exécutables.

2.2 Gestion des nombres de tailles arbitraires et exacts

Il faut savoir qu'il n'est pas possible de stocker directement des nombres très grands dans des variables. Des limites courantes sont $2^{32} - 1 = 4294967295$ pour des entiers (`unsigned int`), et $2^{1024} \approx 1.8 \times 10^{308}$ (nombre à 309 chiffres) pour des nombres à virgule (`double`). Ainsi, si vous créez un programme contenant :

```
1 unsigned int n1(4000000000), n2;
2 n2 = n1 * 2;
```

`n2` est censé valoir 8000000000, mais en réalité, le programme se comporterait de manière bizarre à la place (plantage, `n2` peut valoir un autre nombre à la place, afficher `inf`...).

Certains pourraient se dire : mais 10^{308}, ou même simplement quatre milliards, sont des nombres très grands. Pour vous dire la vérité, même en physique, on n'a pas besoin de nombres aussi grands. Par exemple, le nombre d'atomes dans l'univers entier s'élève à seulement 10^{80} environ, soit plus de deux cents ordres de grandeur de différence par rapport à 10^{308}. Donc, on est en droit de se dire : « Il y a de la marge ! ».

Mais, il y a deux principales raisons pour lesquelles ces nombres gérés en « natif » ne sont pas toujours suffisants. La première, est que, pour les mathématiciens, il est très facile de dépasser ces limites. Quelques puissances par-ci, par-là, et on est vite arrivé à des grands nombres. Rien que $5^{5^5} = 5^{3125}$ font déjà environ 1.91×10^{2184} ! Ils sont alors bien frustrés des « overflow limit » des calculatrices conventionnelles, ces messages d'erreurs qui apparaissent quand on tape un calcul qui donne un résultat trop élevé. Le second argument, surement le plus important, est même si les nombres peuvent aller jusqu'à 10^{308}, on perd beaucoup de précision : l'ordinateur ne stocke qu'une dizaine de chiffres après la virgule, si on se met en notation scientifique, et tout le reste est perdu. Ceux qui étudient la théorie des nombres ont réellement besoin d'une précision exacte, pour calculer des restes de division, par exemple.

Une gestion des nombres arbitrairement grands et exacts résoudrait ces deux problèmes en stockant *tous* les chiffres d'un nombre de *taille quelconque*. Un argument bonus en faveur d'une telle gestion serait que l'implémentation de ces nombres est très enrichissante, et permettra de manier toute une palette d'outils mathématiques et informatiques.

2.2.1 Implémentation de ces nombres

Nombres naturels

Commençons d'abord par les nombres les plus élémentaires en mathématiques, les entiers naturels. L'idée est simple : prenez par exemple le nombre 40297. À l'école élémentaire, vous avez appris que 7 est le chiffre des unités, 9 le chiffre des dizaines, 2 le chiffre des centaines, 0 le chiffre des milliers, et 4 le chiffre des dizaines de milliers. Il va donc falloir faire le même type de décomposition lorsqu'on fera le codage. Imaginons donc un *tableau* dans lequel on va représenter ce nombre :

4	0	2	9	7

Ceci serait une variable contenant cinq autres, valant 4, 0, 2, 9, et 7. Puisque chaque chiffre est un nombre stocké dans une variable, compris entre 0 et 9, on est très loin des limites imposées par l'ordinateur. Ce qui n'est pas possible, c'est stocker par exemple 10^{309} dans une variable, mais cela ne nous empêche pas de stocker 10^{309} dans 310 variables : la première qui représente un 1, et le reste un 0. 40297 serait stocké dans cinq variables, 2014 dans quatre variables, etc.

Pour manipuler des variables contenant un nombre variable de variables (!), on fait appel à un *tableau dynamique*. Ceci est un peu compliqué à manipuler en C, mais les choses sont simplifiées à l'aide de la classe `vector` en `C++`, qui permet de stocker plusieurs variables (ici des chiffres) dans une seule variable, de type `vector`. La taille de tels tableaux n'étant pas fixe, on peut y mettre autant de chiffres qu'on le souhaite, tant qu'on ne sature la mémoire du système exécutant le programme.

Dans les tableaux, une caractéristique très importante est la numérotation des cases : il reste encore à définir comment les indices seront utilisés pour désigner chaque chiffre. Dans ce tableau stockant le nombre 40297, la case numéro une est-elle le 4, ou bien le 7 ? Il s'agit de choisir intelligemment. Sachant qu'en `C++`, la première case est d'indice zéro et non une, on pourrait admettre que l'indice d'une case corresponde à l'exposant du chiffre qu'il contient : en effet, 40297 peut se décomposer comme suit :

$$40297 = 4 \times 10^4 + 0 \times 10^3 + 2 \times 10^2 + 9 \times 10^1 + 7 \times 10^0$$

Ainsi, il serait logique de faire en sorte que la case 0 soit le chiffre des unités (7), la case numéro une le chiffre des dizaines (9), la case numéro deux le chiffre des centaines (2), et

ainsi de suite.

Exemple d'utilisation de la classe `vector`, ici pour stocker 40297 :

```
std::vector<unsigned int> vect;
vect.push_back(7);
vect.push_back(9);
vect.push_back(2);
vect.push_back(0);
vect.push_back(4);
```

On commence par déclarer l'objet de la classe `vector`, ici appelé « vect », qui stocke un tableau dynamique de variables. Comme `vector` est en réalité un patron de classe, on précise de quel type sont les variables contenues, en l'occurrence `<unsigned int>`. Ne vous préoccupez pas trop du `std::` qui a un rapport avec les espaces de noms. La méthode `push_back` permet d'insérer un nombre dans `vect`, ce qui nous permet d'enregistrer un à un les chiffres du nombre 40297. Bien sûr, on créera des fonctions pour simplifier la création de ce nombre. Les valeurs `vect` sont accessibles à l'aide des crochets `[]`, qui contiennent l'indice (la position du chiffre). Devinez alors quel chiffre renverrait ceci :

```
vect[3];
```

Nous avions dit que l'indice correspondait à la puissance de dix associée au chiffre. 10^3 étant les milliers, `vect[3]` renverrait donc le chiffre 0 de 40297.

Nous avons maintenant une base pour créer une classe contenant des nombres naturels exacts de tailles arbitraires. Je l'ai baptisée « `NNumber` » dans mon code, N pour indiquer qu'il s'agit de nombres de l'ensemble $\mathbb{N}$, et « Number » pour « nombre » en anglais.

En plus de gérer des nombres arbitrairement grands, il peut être intéressant de gérer les cas où on aurait des valeurs « bizarres », par exemple si on fait une division par zéro, ou si on est confronté à une indétermination. Ainsi, en plus du tableau dynamique, on peut ajouter des variables booléennes [1] indiquant si le nombre est infini ou non, et s'il est indéfini ou non. Effectuer $\frac{1}{0}$ donnera un nombre dont la variable indiquant l'infini est vraie, tandis que faire 0^0 produira un nombre avec la variable indiquant que le nombre est indéfini à la valeur vraie. On peut voir cela comme des alternatives aux plantages ou aux messages d'erreurs.

Voici un aperçu simplifié de cette classe `NNumber` :

```
class NNumber {
    private:
    std::vector<unsigned char> _digits; // Tableau dynamique contenant
        dans chaque case un chiffre
    bool _inf, _undef; // Booléennes déterminant ou non le caractére
       infini ou indéfini d'un nombre

    // ... (méthodes privées)

    public:
    // ... (constructeurs et méthodes publiques)
}
```

Notez que le `vector` stocke des variables de type `unsigned char`. Ce type permet de stocker des nombres compris entre 0 et 255, ce qui est suffisant pour stocker des chiffres. Il est utilisé à la place de `int`, car il prend moins de place en mémoire.

1. C'est-à-dire des variables ne pouvant avoir que deux valeurs : vrai (1) et faux (0).

Entiers relatifs

Une fois les nombres naturels gérés, l'implantation des entiers relatifs est très facile. Qu'est-ce qu'un entier relatif ? C'est tout simplement un nombre naturel avec un signe. Donc, en plus du tableau des chiffres, nous ajoutons juste une variable booléenne qui permettra de stocker le signe, 0 indiquant l'absence de signe, et 1 sa présence. On crée donc la classe « `ZNumber` » :

```
1 class ZNumber {
2     private:
3     std::vector<unsigned char> _digits;
4     bool _inf, _undef;
5     bool _sign; // Booléenne déterminant ou non la présence d'un signe
6
7     public:
8     // ...
9 };
```

Nombres rationnels/fractions

Le stockage de ces nombres de manière exacte se code également intuitivement. En effet, un nombre rationnel est simplement composé de deux entiers relatifs : un numérateur, et un dénominateur. On crée donc une nouvelle classe « `QNumber` » composée de deux « `ZNumber` », comme suit :

```
1 class QNumber {
2     private:
3     ZNumber _num, _den; // Numérateur et dénominateur
4
5     // ...
6
7     public:
8     // ...
9 };
```

On pourra se poser la question suivante : le programme simplifiera-t-il lui-même les fractions, ou les laissera telles quelles ? Par exemple, si on tape 4/6, `Algébra` enregistra-t-il ce nombre tel quel ($\frac{4}{6}$), ou bien le simplifiera-t-il en $\frac{2}{3}$? J'ai opté pour la première option, mais en simplifiant automatiquement dès qu'une opération est effectuée. Ainsi, $\frac{4}{6}+1$ donnera $\frac{5}{3}$ et non $\frac{10}{6}$.

Nombres rationnels complexes

Nous désirons également gérer les nombres complexes. Cela n'est pas plus compliqué que la gestion des nombres rationnels : un nombre complexe est composé d'une partie réelle, et d'une partie imaginaire, donc tout simplement de deux nombres réels. Comme nous ne gérons que les nombres complexes rationnels, il suffira donc de définir une classe contenant deux nombres rationnels. Nous appellerons cette classe « `CQNumber` », « C » pour « complex(e) » et Q pour indiquer que nous n'utilisons que des coefficients rationnels :

```
1 class CQNumber {
2     private:
3     QNumber _re, _im; // Parties réelle et imaginaire
4
5     public:
6     // ...
7 };
```

En réalité, l'implémentation dans `Algébra` est différente de celle-ci : j'ai créé à la place de `CQNumber` un patron de classe `C<T>`, où `T` est une classe compatible avec les nombres complexes. Cela permet d'utiliser non seulement des nombres complexes avec comme coefficients des `QNumber`, mais aussi des `double`, par exemple. Ceci a été fait pour des raisons de performances : la représentation graphique de polynômes stockés avec des nombres arbitrairement grands est très lente ! Pour ne pas compliquer les choses, je vais expliquer comme si j'avais implémenté la classe `CQNumber`.

Implémentation des nombres réels/complexes à coefficients réels

Gérer des nombres rationnels complexes est déjà une bonne chose, mais qu'en est-il des nombres réels et des nombres complexes à coefficients réels ? Il faut savoir qu'en informatique, certains nombres réels sont facilement implémentables, comme la racine carrée de deux $\sqrt{2}$ ou même $(1+2i)^{3-4i}$: il suffirait de créer encore une fois une classe contenant deux complexes rationnels, un pour la base, et l'autre pour l'exposant. N'oubliez pas que $\sqrt{2} = 2^{\frac{1}{2}}$.

De même, on pourra gérer des nombres encore plus compliqués, comme $1+\sqrt{2}$, voire $(2+\sqrt[3]{7})^5 + (\frac{17}{23} - \sqrt[5]{9}i)^{3-i} - 6$, si on s'amusait un peu avec les tableaux dynamiques. Une telle gestion sera d'ailleurs possible plus tard, à l'aide du stockage des opérations numériques. Mais, qu'en est-il de la gestion de certains nombres importants, comme π ou e ? Gérer ces nombres est quelque chose de très difficile en informatique, et nous n'allons pas le faire dans le cadre de ce travail.

2.2.2 Implémentation des opérations

En standard, le `C++` propose déjà les opérations de base. Par exemple, il sait faire $2+3$, ou bien $8 \div 2$. Mais, comme nous avons défini nos propres nombres, le `C++` ne peut pas deviner de lui-même comment effectuer des opérations avec nos classes. Ainsi, nous allons (re)définir pour toutes les classes que nous venons de créer les quatre opérations élémentaires (addition, soustraction, multiplication, et division), ainsi que quelques autres que vous découvrirez par vous-même. Commençons par la définition des opérations dans les entiers naturels.

Nombres naturels

Addition Les chiffres des nombres sont stockés individuellement dans des cases d'un tableau dynamique. Pour additionner deux nombres codés ainsi, l'astuce est de s'inspirer à ce que nos petits écoliers font à l'école élémentaire. On va utiliser l'addition en colonnes. Vous savez, poser l'addition, et additionner les unités entre elles, puis les dizaines, les centaines, en retenant si le résultat ne tient pas sur un chiffre. Vous rappelez-vous de cet algorithme ?

Dans le code, l'addition a été implémentée exactement de cette manière. On additionne les chiffres des cases d'indice 0 des deux nombres entre eux, on enregistre l'unité dans la case 0 du résultat, et on retient s'il le faut. On additionne ensuite les cases d'indice 1 en ajoutant si nécessaire la retenue, on enregistre le résultat dans la case 1 du résultat, on retient s'il le faut, et ainsi de suite. On arrête une fois que toutes les cases ont été additionnées. Les cases vides sont considérées comme nulles : le programme effectuera 1536 + 0768, par exemple.

Soustraction Passons maintenant à la soustraction. Ici, pas de surprise, nous allons utiliser la même astuce que pour l'addition, utiliser la soustraction en colonnes. L'implémentation s'est faite en suivant à la lettre cette démarche, à l'instar de l'addition. À noter qu'il faut prendre en compte le fait que dans les nombres naturels, il n'est pas possible de soustraire

par un nombre plus grand. Ainsi, faire $1 - 2$ donnera une valeur indéfinie, à l'aide de la possibilité de stocker des nombres infinis ou indéfinis dont nous avions parlé.

Multiplication Tout comme l'addition et la soustraction, l'implémentation de la multiplication utilise le calcul en colonnes appris à l'école primaire. Le programme fera une multiplication exactement de cette manière. Les termes des additions sont stockés dans des variables temporaires.

Division Ici, un problème se pose. Lorsqu'on souhaite calculer $6 \div 3$ ou $20 \div 5$, on n'aura pas de souci particulier. Mais, comment gérons-nous les cas où la division ne donne pas un nombre entier, par exemple pour $3 \div 2$ ou $11 \div 7$? La classe `NNumber` ne peut stocker que des nombres entiers. Allons-nous stocker des quotients entiers avec restes, ou donner une réponse sous forme de `QNumber` ?

Dans la bibliothèque standard du `C++`, lorsqu'on fait une telle division, la machine retourne simplement la partie entière du nombre. Par exemple, $5 \div 2$ donnera 2 comme réponse. Nous allons faire de même. Pour cela, on utilisera également un algorithme appris à l'école élémentaire, la division euclidienne. On verra que la division entière sera très utile pour le modulo.

Modulo Modulo ? Il est possible que vous n'ayez jamais entendu parler de ce terme, mais vous connaissez certainement son principe : le modulo est un synonyme du *reste d'une division*. Par exemple, si vous faites $7 \div 3$, vous savez qu'il est possible de mettre deux fois le nombre trois dans sept. Par conséquent, le quotient de cette division est 2 (dans le cas d'une division entière). Mais $2 \times 3 = 6$, pas 7 ! Il manque ici 1 pour arriver à 7. On dit alors que $7 \div 3$ a un quotient de 2 et un reste de 1.

On définit alors une opération dérivée de la division, le modulo, qui permet d'obtenir ce reste : 7 modulo 3 égalent 1, qui s'écrit $7 \bmod 3 = 1$, ou aussi $7 \,\%\, 3 = 1$ en `C++`. Voici quelques exemples supplémentaires pour que vous vous assuriez d'avoir compris :

$$\begin{array}{cccc} 5 & \bmod 2 = 1 & 19 & \bmod 10 = 9 \\ 5 & \bmod 10 = 5 & 10 & \bmod 5 = 0 \\ 15 & \bmod 17 = 15 & 64 & \bmod 20 = 4 \end{array}$$

Le modulo est nul si le nombre de gauche est un multiple de l'autre, et est égal au nombre de gauche s'il est le plus petit.

Comment calculer facilement un modulo ? Soient a et b deux nombres, q le quotient entier de la division, et r le reste de la division. Si on exprime a en fonction des autres valeurs, on a : $a = bq + r$, d'où $r = a - bq$. Or, q étant le quotient entier de la division, on peut également dire qu'il s'agit de la partie entière de $\frac{a}{b}$, soit $\lfloor \frac{a}{b} \rfloor$. Sachant que la division entière a justement été implémentée, il suffira donc de calculer le reste avec la formule suivante :

$$r = a - b\lfloor \frac{a}{b} \rfloor$$

Plus grand diviseur commun Le plus grand diviseur commun (PGDC) de deux nombres est le plus grand nombre qui divise ces deux nombres sans laisser de reste. On note $PGDC(a, b)$ pour désigner le plus grand diviseur commun de deux entiers naturels a et b. Il sera utile pour la simplification des nombres rationnels.

Maintenant, comment le calculer ? Une méthode qui peut rapidement venir à l'esprit est la décomposition des nombres en facteurs premiers : on réécrit les deux nombres sous la

forme de facteurs de nombres premiers, c'est-à-dire des nombres ayant exactement deux diviseurs : eux-mêmes et un (1 n'est pas premier). Puis, on pioche les facteurs communs, et on fait leurs produit pour obtenir le PGDC. Par exemple, cherchons le PGDC de 288 et 120. Ces deux nombres se décomposeront comme suit :

$$288 = 2 \times 2 \times 2 \times 2 \times 2 \times 3 \times 3$$

$$120 = 2 \times 2 \times 2 \times 3 \times 5$$

On voit que le nombre $2 \times 2 \times 2 \times 3 = 24$ se retrouve chez 288 et 120 : c'est leur plus grand diviseur commun. Simple, n'est-ce pas ? Cependant, qu'en est-il des nombres avec beaucoup de chiffres ? La décomposition en facteurs premiers ne sera pas aisée. Encore pire s'il s'agit d'implémenter cette fonction, qui risque de couter énormément de ressources lors de la décomposition en nombres premiers.

Il faut donc trouver une autre méthode. Après quelques recherches, j'ai trouvé un algorithme, l'*algorithme d'Euclide*, qui permet de calculer des PGDC sans faire appel aux nombres premiers : il utilise uniquement des modulos (divisions entières et soustractions), ce qui est beaucoup plus léger. Bonne nouvelle, puisque ce sont des opérations que nous avons déjà définies ! Voici comment il fonctionne :

Algorithme d'Euclide Soient a et b deux nombres dont on cherche le PGDC, avec $a > b$. Cet algorithme s'utilise en suivant cette démarche :

- Calculer le reste r de la division de a par b, $r = a \mod b$.
- Si le reste est nul, on a terminé, et b est le $PGDC$ que nous cherchons.
- Sinon, a prend la valeur de b et b la valeur de r, et on recommence.

Exemple Reprenons l'exemple précédent :

- $a = 288$ et $b = 120$.
- On calcule $r = 288 \mod 120 = 48$.
- $r \neq 0$. a prend alors la valeur de b et b la valeur de r.
- On recommence donc avec $a = 120$ et $b = 48$.
- On calcule $r = 120 \mod 48 = 24$.
- $r \neq 0$. a prend alors la valeur de b et b la valeur de r.
- On recommence donc avec $a = 48$ et $b = 24$.
- On calcule $r = 48 \mod 24 = 0$.
- $r = 0$. On a terminé, et le PGDC est $b = 24$.

Démonstration Pour ceux à qui ça intéresse, montrons ici que cet algorithme fonctionne pour des valeurs quelconques. On recherche le PGDC de a et b, $a > b$. En effectuant une division avec quotient q et reste r, on peut établir la relation suivante :

$$a = bq + r$$

Deux cas sont possibles. Si r est nul, alors on a que b divise a, et comme b se divise aussi lui-même, b est le PGDC qu'on cherche.

Sinon, si r n'est pas nul, on peut déjà déterminer grâce à la démarche suivante que tout nombre n divisant b et r divisera également a sans laisser de reste (n'oubliez pas que $\frac{b}{n}$ et $\frac{r}{n}$ sont entiers) :

$$a = bq + r \Leftrightarrow a = \frac{b}{n}nq + \frac{r}{n}n \Leftrightarrow a = n(\frac{b}{n}q + \frac{r}{n})$$

a sera donc aussi divisible par n, et cela donnerait le quotient entier $\frac{b}{n}q + \frac{r}{n}$. On va donc chercher des diviseurs qui divisent b et r. En remplaçant a par b, et b par r, on peut recommencer le premier modulo en calculant le reste r_2 (il y aura aussi un quotient q_2), qui sera de toute manière plus petit que le précédent :

$$b = rq_2 + r_2$$

De nouveau deux cas. Si ce nouveau reste r_2 est nul, alors r divise b et se divise lui-même, et donc divise également a. Donc, r (qui, pour rappel à pris la place de b) est le PGDC qu'on cherche. Sinon, nous devons refaire la procédure, et comme précédemment, si r_3 est nul, r_2 divisera r et lui-même, et donc b et a également, sinon on continue, jusqu'à ce que r_i soit nul, puisqu'on pourra toujours appliquer le même raisonnement et « remonter » jusqu'à a et b.

Plus petit multiple commun Le plus petit multiple commun de deux nombres est le plus petit nombre pouvant être divisé par ces deux nombres donnés. Par exemple, 12 est le PPMC de 6 et de 4, parce que $12 \bmod 6 = 0$ et $12 \bmod 4 = 0$, et qu'il n'y a pas de nombre plus petit respectant cette condition. 24 respecte également cette condition, mais n'est pas le plus petit multiple commun.

Ce multiple commun peut facilement être calculé en effectuant le produit des deux nombres a et b dont on cherche le PPMC, puis en divisant le résultat par le PGDC de ces deux nombres. En effet, le produit donnera obligatoirement un multiple commun, puisqu'il est divisible par a et b. Mais, cela ne donnera pas toujours le plus petit nombre, et donc on divise par le PGDC pour simplifier les facteurs premiers en trop du produit, ce qui donne le PPMC recherché.

Puissance Soit la puissance a^n. Dans les nombres naturels, la puissance s'implémente facilement : il suffit de multiplier la base a avec elle-même n fois, à l'aide d'une boucle. On gèrera les cas particuliers $a^0 = 1$ et $0^0 =$ `undef`.

Entiers relatifs

Concernant les entiers relatifs, donc la classe `ZNumber`, la programmation des opérations consiste à reprendre ceux déjà définis pour `NNumber`, et de gérer en plus les signes.

Addition et soustraction Dans l'addition avec signes, on distingue deux cas :

- Addition entre deux nombres de mêmes signes : on additionne les valeurs absolues, et garde le signe. Par exemple, $3 + 5 = 8$ et $-3 + (-5) = -8$.
- Addition entre deux nombres de signes opposés : on garde le signe du nombre ayant la plus grande valeur absolue, et on soustrait la plus grande valeur absolue par l'autre. Par exemple, $-3 + 5 = 2$ et $3 + (-5) = -2$.

On peut utiliser les opérations déjà définies pour `NNumber` pour effectuer les additions et soustractions sur les valeurs absolues. Ainsi, on a seulement à ajouter la gestion des signes.

Comme la soustraction est équivalente à l'addition par l'opposé, c'est-à-dire $a - b = a + (-b)$, il n'y a pas besoin de définir de fonction dédiée à cette opération : on crée juste une fonction qui additionne par l'opposé. Ce sera la même chose pour `QNumber` et `CQNumber`.

Multiplication et division La multiplication et la division se font de la même manière que les nombres naturels, en appliquant en plus la règle des signes (multiplier ou diviser

deux nombres de mêmes signes donne du positif, tandis que le faire avec deux nombres de signes différents donne du négatif).

Modulo, PPMC et PGDC Le principe du modulo est le même que pour les nombres naturels : on calcule le reste de la division, mais en prenant en compte le signe. Faire ces exemples aide à déterminer quelles réponses on s'attend à obtenir :

$$\begin{aligned} 7 = 3 \times 2 + 1 \Leftrightarrow 7 \mod 2 = 1 \\ 7 = -3 \times (-2) + 1 \Leftrightarrow 7 \mod -2 = 1 \\ -7 = -3 \times 2 - 1 \Leftrightarrow -7 \mod 2 = -1 \\ -7 = 3 \times (-2) - 1 \Leftrightarrow -7 \mod -2 = -1 \end{aligned}$$

Ainsi, le modulo relatif est égal au modulo naturel prenant le signe de l'opérande de gauche. Concernant le PPMC et le PGDC, j'ai décidé de toujours renvoyer les résultats sans le signe. Donc, on utilisera les fonctions définies pour `NNumber`.

Puissance Concernant les puissances, comme un `ZNumber` ne peut pas stocker de fraction, on renvoie un nombre indéfini si l'exposant est négatif. Sinon, on multiplie autant de fois qu'il faut la base, comme pour `NNumber`.

Nombres rationnels

Ici, ce n'est pas vraiment plus compliqué, il s'agira toujours de reprendre les fonctions définies auparavant.

Addition et soustraction Soient deux fractions $\frac{a}{b}$ et $\frac{c}{d}$. Vous savez probablement que la somme de deux fractions se calcule en mettant les deux au même dénominateur, ce qui peut être résumé par l'équation :

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

L'addition a été implémentée en suivant à la lettre cette formule. Quant à la soustraction, on utilise la même astuce que pour `ZNumber` : soustraire, c'est ajouter l'opposé. On simplifie l'expression à la fin si nécessaire, en divisant le numérateur et le dénominateur par leur plus grand diviseur commun.

Multiplication et division Pour les nombres rationnels, la multiplication est plus simple que l'addition ou la soustraction : pas besoin de mettre au même dénominateur, on se contente de multiplier les numérateurs entre eux, et les dénominateurs entre eux. Pour la division, on multiplie par l'inverse, en permutant numérateur et dénominateur.

Puissance entière Pour les puissances avec exposant entier, rien de spécial : on multiplie le nombre avec lui-même autant de fois qu'on le souhaite. On peut désormais gérer les exposants négatifs, en inversant le numérateur et le dénominateur avant de multiplier.

Puissance rationnelle Cela se complique avec les puissances rationnelles, comprenant les racines ; $a^{\frac{m}{n}} = \sqrt[n]{a^m}$. Nous ne pouvons nous contenter que d'approximations. Plus tard, nous pourrons stocker de manière exacte ces nombres avec les expressions, mais nous

pouvons quand même permettre à l'utilisateur d'approximer des nombres rationnels élevés à des puissances rationnelles, en proposant une méthode qui fait ce travail.

Tout d'abord, pour les racines, il existe une formule mathématique que nous allons appliquer à la lettre, qui permet d'effectuer cette approximation par itérations. Il s'agit pour les mathématiciens d'une suite par récurrence. Voici comment on l'utilise :

- On désire calculer la racine n-ième d'un nombre réel positif a, à savoir $\sqrt[n]{a} = a^{\frac{1}{n}}$. n est un nombre naturel non nul.
- Choisir un nombre arbitraire initial $x_i = x_0$.
- Un nombre x_{i+1} plus proche de $\sqrt[n]{a}$ que x_i peut être calculé à l'aide de la formule suivante :
$$x_{i+1} = \frac{1}{n}((n-1)x_i + \frac{a}{{x_i}^{n-1}})$$
- On peut réutiliser autant de fois la formule ci-dessus pour obtenir des nombres de plus en plus proches de $\sqrt[n]{a}$, jusqu'à ce qu'on atteint la précision désirée. Dans ce cas, on remplace x_i par la dernière valeur calculée en utilisant cet formule.

Exemple On souhaite approximer $\sqrt{3}$ en faisant trois itérations de cette formule :

- On a $a = 3$ et $n = 2$.
- On choisit arbitrairement $x_0 = 1$.
- Avec la formule, on substitue les valeurs pour calculer x_1 plus proche de la valeur exacte, et on trouve :
$$x_1 = \frac{1}{n}((n-1)x_0 + \frac{a}{{x_0}^{n-1}}) = \frac{1}{2}((2-1)1 + \frac{3}{1^{2-1}}) = \frac{1}{2}(1 + \frac{3}{1}) = 2$$
- On recommence pour affiner la précision :
$$x_2 = \frac{1}{n}((n-1)x_1 + \frac{a}{{x_1}^{n-1}}) = \frac{1}{2}((2-1)2 + \frac{3}{2^{2-1}}) = \frac{7}{4} = 1.75$$
- On recommence pour affiner encore la précision :
$$x_3 = \frac{1}{n}((n-1)x_2 + \frac{a}{{x_2}^{n-1}}) = \frac{1}{2}((2-1)\frac{7}{4} + \frac{3}{(\frac{7}{4})^{2-1}}) = \frac{97}{56} = 1.732\overline{142857}$$
- On pourrait continuer pour augmenter encore la précision. Après trois itérations, on obtient la valeur approchée $\sqrt{3} \approx \frac{97}{56}$. Et, $(\frac{97}{56})^2 = \frac{9409}{3136} \approx 3.000319$ montre bien que cette valeur est assez proche de la valeur exacte, qui vaut environ 1.732051.

À partir de cette racine approximée, on peut effectuer des puissances rationnelles du type $a^{\frac{m}{n}} = \sqrt[n]{a^m}$ en élevant cette approximation à la puissance m, car $a^{\frac{m}{n}} = (a^{\frac{1}{n}})^m$.

À noter que cet algorithme ne fonctionne pas si a est négatif. On peut toutefois l'utiliser pour des a négatifs si n est impair : on applique la formule pour la valeur absolue de a, et on met le signe après.

Nombres rationnels complexes

Addition et soustraction L'addition et la soustraction consistent à additionner ou soustraire les coefficients entre eux. Donc, rien de franchement compliqué.

Multiplication et division Pour multiplier deux nombres complexes, on fait appel à la double distributivité :

$$(a_1 + b_1 i)(a_2 + b_2 i) = a_1 a_2 + a_1 b_2 i + a_2 b_1 i - b_1 b_2 = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + a_2 b_1)i$$

Pour la division, comme pour les nombres rationnels réels, on multiplie par l'inverse. Soit $z = a + bi$ un nombre complexe. L'inverse d'un nombre complexe s'obtient par :

$$\frac{1}{z} = \frac{\bar{z}}{\bar{z}z} = \frac{a - bi}{(a - bi)(a + bi)} = \frac{a - bi}{a^2 + b^2}$$

$\bar{z}$ est le conjugué de z, et vaut $a - bi$.

Puissance

Rien à expliquer ici : cela consiste toujours à multiplier le nombre complexe avec lui-même autant de fois que le demande son exposant. On inverse avant la multiplication si l'exposant est négatif. On ne gérera que des exposants entiers pour ces nombres, donc $\sqrt{i}$ ne pourra pas être calculé, par exemple.

2.3 Gestion des polynômes

2.3.1 Élaboration des classes

Avant de commencer, nous allons commencer par gérer les noms de variables et les monômes. En effet, un monôme est une expression mathématique composée d'un coefficient, et d'un certain nombre de variables élevées à des puissances naturelles. Par exemple, $-3xy^2$ est un monôme de coefficient -3 et ayant comme variables x (à la puissance 1) et y (à la puissance 2). Lorsqu'on aura terminé la gestion des variables et des monômes, on pourra gérer les polynômes sans trop de peine, car ce ne sont que des sommes de monômes, par exemple $x^2 + 2x - 3$ est un polynôme à trois monômes (trois termes).

Classe des variables

Il s'agit d'une classe qui stocke des noms de variables. Cela permet à l'utilisateur de nommer comme il veut ses variables, par exemple x, y, z pour des coordonnées, t pour le temps, ou encore x_0, x_1,... La conception d'une telle classe est très simple, puisqu'il s'agit juste de stocker une chaine de caractères contenant le nom de la variable, et les méthodes nécessaires pour attribuer ou afficher ce nom. La classe ainsi créée se nomme « `Variable` ».

```
1 class Variable {
2     private:
3     std::string _s; // Stocke le nom de la variable
4
5     public:
6     // ...
7 };
```

Bien que cette classe puisse stocker des chaines de caractères, et donc permet par exemple de stocker des variables dont le nom est un mot, lorsqu'on implémentera l'analyseur syntaxique, on ne détectera que des variables dont le nom ne comporte qu'une lettre. i étant réservé pour l'unité imaginaire, et comme l'analyseur sera sensible à la casse, cela nous donnera pour le moment la possibilité d'utiliser 51 variables différentes dans `PolyCalc`.

Classe des monômes

Comme dit avant, les monômes sont composés d'un coefficient et des variables avec exposants naturels. Pour implémenter cela, on crée une classe ayant comme attributs le coefficient,

les variables, et leurs puissances respectives. Ces deux dernières seront respectivement de type `Variable` et `NNumber`, et sont liées (on stockera autant de variables que de puissances). La classe `pair` existant en natif en `C++` permet de relier ces deux variables. Comme un monôme peut avoir autant de variables que possible, il aura finalement comme attributs son coefficient, et sa partie littérale : un tableau dynamique de paires variable-exposant naturel.

```
1 class Monomial {
2     private:
3     CQNumber _coeff; // Coefficient (partie numérique)
4     std::vector<std::pair<Variable, NNumber> > _litt; // Variables
          avec exposants (partie littérale)
5
6     public:
7     // ...
8 };
```

Les méthodes ajoutées permettront comme d'habitude d'afficher ou de modifier le monôme. On en proposera également d'autres qui déterminent le degré du monôme (somme des puissances), qui détermine si deux monômes sont semblables (on regarde si leur parties littérales sont les mêmes), et on peut aussi permettre la multiplication entre monômes, parce qu'un tel produit donne également un monôme (mais l'addition ou soustraction aurait le plus souvent donné un polynôme, ce qui nous force à sortir de la classe). On créera également des méthodes permettant de dériver et intégrer le monôme, ce qui nous simplifiera considérablement la tâche pour effectuer ces opérations pour les polynômes.

Notez qu'informatiquement, cette classe pour les monômes partage la même caractéristique que celle pour les nombres complexes : on peut imaginer des monômes avec divers types de coefficients : entiers (`ZNumber` ou `int`), rationnels (`double` ou `QNumber`), complexes rationnels (`CQNumber`). Comme avant, j'ai simplifié les explications, mais il est intéressant de savoir que cette classe est également codée en tant que patron de classe.

Classe des polynômes

Comme dit précédemment, un polynôme est une somme de monômes. Comme il peut avoir une longueur arbitraire (on peut avoir autant de monômes que possible), on créera une classe ayant comme attribut un tableau dynamique de monômes. Et c'est tout, concernant les attributs ! D'où l'intérêt d'avoir précédemment créé les classes `Variable` et `Monomial`. Nous appellerons naturellement la classe des polynômes « `Polynomial` ».

```
1 class Polynomial : public Object {
2     private:
3     std::vector<Monomial> _monomials;
4
5     public:
6     // ...
7 };
```

Encore une fois, puisque les coefficients peuvent être de plusieurs types, la vraie implémentation est sous la forme d'un patron de classes. Les polynômes utilisant des nombres de type de base du `C++` sont calculés beaucoup plus rapidement que ceux avec valeurs exactes et arbitrairement grands, moyennant la perte de précision pour les nombres à virgule, et les limites.

Opérations Concernant les méthodes, outre les classiques accesseurs et mutateurs, nous pouvons ici implémenter les opérations de base : addition, soustraction, multiplication. Pour simplifier, on peut d'abord écrire des méthodes permettant ces opérations entre un

polynôme et un monôme : ajouter un monôme à un polynôme revient à ajouter une case dans le tableau dynamique des monômes, mais il faut vérifier que ce polynôme ne contient pas déjà un monôme semblable (auquel cas on additionnera juste le coefficient concerné). Pour la soustraction, on fait pareil, mais en additionnant par le monôme opposé, c'est-à-dire avec le coefficient de signe contraire. Pour la multiplication, on multiplie tous les coefficients par celui du monôme, et on additionne toutes les puissances par celles du monôme ; le nombre de termes est inchangé. Si pour une raison ou une autre, un monôme ou le polynôme devient 0 après une opération, on enlève les termes nuls du tableau dynamique.

Une fois ceci effectué, additionner et soustraire un polynôme à un autre devient aisé, en ajoutant au premier polynôme chaque monôme du deuxième, avec une boucle. Pour la multiplication, la règle de la distributivité est appliquée : on multiplie le premier polynôme par le premier monôme du deuxième, ce qui donne un nouveau polynôme. Puis, on fait la même chose en multipliant par le deuxième monôme pour générer un autre nouveau polynôme, et ainsi de suite. On ajoute ensuite ces nouveaux polynômes entre eux.

Autres méthodes Le degré d'un polynôme se détermine en trouvant le monôme ayant le plus grand degré. La dérivation et l'intégration deviennent également très simples, puisque la dérivée d'une somme est la somme des dérivées des termes, et c'est la même chose pour les primitives. Donc, il suffit d'effectuer cette opération pour tous les monômes du polynôme.

Une fonction indispensable est l'évaluation d'un polynôme, c'est-à-dire remplacer une ou plusieurs variables du polynôme par des nombres. Notre but sera de proposer une fonction prenant en argument un nombre et une variable de type `Variable`, qui substituera le nombre dans cette variable. Elle consiste à multiplier tous les coefficients concernés par sa variable substituée et élevée à sa puissance respective, et à retirer la variable. Il conviendra d'écrire d'abord une fonction d'évaluation pour la classe des monômes avant, pour simplifier celle des polynômes : l'évaluation de polynômes consiste en celle de tous ses monômes. Après, on additionne tous ces monômes évalués entre eux.

Exemple de résultat :

```
P(x, y, z) = x^3y^2z - x^2y^2z + 4xy^2z - 3y^2z - x^3yz + 4xyz + 2x^3z
    + 2x^2z - 2xz + 4z - 5x^3y^2 - 2x^2y^2 - 4xy^2 + 4x^3y + 4x^2y - 3
   xy + x^3 + 3x^2 + x - 3

x = 11
P = 1251y^2z - 1287yz + 2886z - 6941y^2 + 5775y + 1702

y = -7
P = 73194z - 378832

z = 5
P = -12862
```

Calcul des racines des polynômes formels de degré 1 et 2 (survol) Une fonction que devrait proposer `PolyCalc` serait le calcul des racines d'un polynôme. Nous possédons désormais tout l'arsenal nécessaire pour le gestion de polynômes, mais il reste un énorme problème. Si on sait désormais stocker des nombres arbitrairement grands, et n'importe quel polynôme à coefficients complexes rationnels, on est incapable de manipuler les nombres irrationnels. Ainsi, comment va-t-on faire pour stocker les solutions de $x^2 - 2 = 0$? La réponse se trouve dans le chapitre suivant.

3. Objets, expressions, et analyseur syntaxique

3.1 Les expressions

Peut-être que vous vous demandez, où est-ce que je souhaite en venir, lorsque je parle d'expressions. On ne s'en rend pas forcément compte, mais dans les mathématiques, les expressions sont partout. $x + 1$ est une expression, $2 + 3$ est une expression, $xy^2 = 3z$ est une expression,... Il s'agit donc d'un terme très général qui peut se résumer à une suite de symboles mathématiques ayant un sens.

Nous allons réfléchir à l'implémentation d'une classe qui permettrait de stocker des expressions quelconques. Si nous pouvions stocker par exemple $1 + 2^{\frac{1}{2}}$, nous aurions un outil permettant de stocker n'importe quelle solution de polynôme (et aussi quelques nombres transcendants comme i^i).

3.1.1 Première idée, et problème

L'idée qui vient immédiatement à l'esprit, est d'utiliser un tableau dynamique, comme on l'a toujours fait. Stocker un x nécessitera une case, alors que $2x + y$ en nécessitera quatre, par exemple. Visuellement : $\boxed{x}$ et $\boxed{2}\boxed{x}\boxed{+}\boxed{y}$.

Mais, ce ne sera pas aussi simple que précédemment. Un nombre arbitrairement grand peut être stocké à l'aide d'un tableau dynamique de chiffres. Un polynôme peut être enregistré sous la forme d'un tableau dynamique de monômes, étant eux-mêmes composés d'un coefficient et d'un tableau dynamique de variables avec leurs puissances respectives. C'était à chaque fois des tableaux contenant *un seul type* d'éléments. Que des chiffres, que des monômes, ou que des couples variables-entiers. Le stockage d'une expression nécessite un tableau dynamique permettant de stocker *plusieurs types* d'éléments : $2x + y$ impose le stockage de trois types d'éléments : il y a un nombre (2), un opérateur ($+$), et deux variables (x et y).

Or, en `C++`, il n'est pas possible de définir un `vector` avec plusieurs types différents. Que faire alors ? Nous allons suivre cette obligation, mais en rusant. Puisqu'on ne peut définir que des tableaux dynamiques d'un seul type d'élément, on le fera. Et ce type, ce sera un type générique : un objet (mathématique). En réfléchissant orienté objet, on sait qu'un nombre est un objet. Une variable est un objet. Une opération est un objet. Ce sont tous des objets mathématiques, de différents types, certes, mais des objets. Et donc, un tableau dynamique d'objets génériques nous permettra de stocker ces différents types d'objets ! Dans le jargon du programmeur, on dit qu'il s'agit d'une *collection hétérogène*.

On utilisera pour cela le concept d'*héritage*. Il s'agit d'un moyen permettant à des classes d'utiliser les attributs et méthodes d'une autre classe déjà existante. Nous créerons alors une classe abstraite, appelée « `Object` », dont toutes les autres héritent : une variable est un objet, un polynôme est un objet, une opération est un objet. Créer un tableau dynamique d'`Object` permet de gérer des chaines quelconques d'opération.

Dans la réalité, c'est un peu plus compliqué. Dans les collections hétérogènes, on fait appel

à la notion de *polymorphisme*, permettant de créer des fonctions s'adaptant aux différents types d'objets (sinon, parmi les objets, on n'aurait pas pu distinguer un nombre d'une variable, par exemple). Je n'irai pas plus loin dans ce concept, mais je le mentionne pour que vous puissiez faire vos propres recherches si cela vous intéresse. Et informatiquement, les collections hétérogènes ne sont en fait pas de simples `vector` d'objets génériques, mais des `vector` de *pointeurs* vers ces objets ! Sans cette façon de faire, le polymorphisme ne se fait pas, et la collection hétérogène ne fonctionnerait pas. La définition d'un pointeur est simple (il s'agit d'une variable qui stocke la position en mémoire (l'adresse) d'une autre variable), mais leur utilisation est peu évidente, et ne sera pas exposée ici.

3.1.2 La classe Object

Théoriquement, pour une classe aussi générale que celle-ci, on pourrait la laisser vide, et définir spécifiquement les autres. En pratique, il est possible d'ajouter quelques méthodes communes à tous les objets, par exemple une fonction renvoyant le type de l'objet considéré (nombre ? variable ? polynôme ?...), une fonction renvoyant sa taille (longueur d'un nombre, nombre de termes d'un polynôme,...), et une permettant de le mettre sous la forme d'une chaine de caractères pour l'afficher plus facilement. Il n'aura par contre aucun attribut.

Ces méthodes doivent être adaptées suivant le type de l'objet qui hérite de cette classe générique. Dans la classe générale, elles ne seront pas du tout définies, vu qu'elles se comportent différemment suivant le type de la valeur. On indique que la méthode n'est pas définie à l'aide d'un « `= 0` » à la fin du prototype, et d'un « `virtual` », et on dit alors que ce sont des *méthodes virtuelles pures*. En `C++`, si une classe possède au moins une telle méthode, on dit qu'il s'agit d'une *classe abstraite pure*. Une telle classe ne peut pas avoir d'instance : on ne pourra pas déclarer une variable de type `Object`. De plus, toutes les classes héritant d'une classe abstraite pure doit obligatoirement posséder ses propres définitions de toutes ces méthodes virtuelles pures. Sinon, à côté de ces méthodes virtuelles pures, on peut très bien en déclarer d'autres qui n'en sont pas, et qui seraient le comportement général ou par défaut de ce qui hérite de la classe.

```
1 class Object {
2     public:
3     // Tous ls objets ont un type
4     virtual unsigned char type() const = 0;
5     // Tous les objets ont une taille (par défaut 0)
6     virtual unsigned int size() const {return 0;}
7     // Tous les objets peuvent être affichés
8     virtual std::string inString() const = 0;
9 };
```

3.1.3 Classes d'objets

Admettez l'expression suivante :

```
123ix^2y - 987z^0.5 * (3^2 + 4^2)^(1/2)
```

Si on la décompose en caractères individuels, on peut y voir six types d'éléments : les chiffres (`1, 2, 3,...`), les variables (`x, y, z`), les constantes (`i`), les opérations (`+`, `-`, `*`, `/`, et `^`), les virgules (ou points), et les parenthèses.

Les chiffres pourraient simplement être stockés en tant que nombres, à l'aide de la classe `NNumber`, ou même `CQNumber`. Nous avons donc déjà les moyens de gérer ce type d'élément, et on ne compliquera pas les choses ici : on s'arrangera pour ne pas stocker les chiffres

individuellement, mais sous la forme d'un nombre, donc on les regroupera, et stockera directement $123i$ et 987 dans respectivement un `CQNumber` et un `NNumber`.

Les variables sont stockables dans la classe `Variable` déjà existante. Comme mentionné dans la section présentant cette classe, on ne reconnaitra que les variables d'une lettre (minuscules et majuscules différenciées), et la lettre i est réservée pour l'unité imaginaire, et ne pourra donc pas être utilisée en tant que variable.

Les constantes numériques peuvent être stockées de manière similaire aux variables. Les seules différences sont les suivantes : une constante numérique ne peut pas être renommé à la volonté de l'utilisateur, et ne peut pas être substitué (sauf si on souhaite avoir une approximation pour certaines constantes comme π ou e). Il est possible de créer une liste prédéfinie de constantes, à l'aide du mot-clé `enum`, mais en pratique, on ne gérera que la constante i qui est l'unité imaginaire. On le stockera dans dans un `CQNumber` et on ne proposera pas une classe spécifique aux constantes.

Le stockage des symboles des opérations fonctionne sur le même principe des constantes : on crée une liste prédéfinie de symboles à l'aide d'`enum` (plus, moins, fois, divisé, puissance,...). Ce qui donnera lieu à une classe « Operation » :

```
1 class Operation : public Object {
2     private:
3     unsigned int _op; // Stocker l'opération
4
5     public:
6     // Énumére une liste d'opérations
7     enum {NONE, PLUS, MINUS, TIMES, DIV, POW, FAC, DERIVATE, INTEGRATE
          , UNDEF};
8     // ...
9 };
```

3.2 L'analyseur syntaxique (numérique)

Nous allons réfléchir à comment mettre en place un analyseur syntaxique. Pour commencer, il faut peut-être rappeler le principe d'une telle fonction : l'utilisateur entre un texte (une chaine de caractères), et le programme l'analyse, l'interprète, pour en déduire quelles opérations effectuer, quel objet mathématique représente tel ou tel symbole, etc. Puis, il enregistre les informations et les transmet aux autres parties du programme qui fera les opérations demandées par l'utilisateur. Ceci est le rôle de l'analyseur syntaxique.

Donc, cela se résume à trouver des moyens d'interpréter une chaine de caractères, de le transformer en un `vector<Object*>` dont on en a parlé. Nous avons vu précédemment qu'il existait plusieurs types de symboles pouvant être entrés par l'utilisateur, et mettrons en place l'analyseur en gérant au cas par cas chacun de ces types.

3.2.1 Règles générales d'interprétation

Premièrement, il faut réfléchir à la question suivante : qu'attend le programme de l'utilisateur ? Ou, dit autrement, que doit taper l'utilisateur pour entrer une opération ? Un polynôme ? Par exemple, pour faire $1 + 2$, il pourrait aussi bien taper `1 + 2` que `add(1, 2)`. Le but de `PolyCalc` étant d'être le plus accessible possible pour le grand public, nous privilégierons les notations les plus intuitives et simples, donc la première possibilité, dans le cas des opérations. À noter également que les espaces sont ignorées, donc `1 + 2` et `1+2` seront interprétés de la même manière.

Les règles générales d'analyse seront les suivantes :

– Toutes les saisies sont enregistrées dans des `string`.
– On supprime toutes les espaces avant l'analyse. `1 + 2` devient `1+2`.
– Toute suite de chiffres détectée est considérée comme un nombre.
– `i` est interprété comme étant l'unité imaginaire.
– Toutes les autres lettres sont interprétées comme des variables.
– En enregistrant la saisie, les opérations ne sont pas effectuées : entrer `2/3` ne sera pas interprété comme un nombre rationnel, mais une division de deux nombres entiers. `1 + 2` ne sera pas remplacé par `3` après l'analyse, mais restera tel quel. N'oubliez pas que le but ici est d'analyser une saisie, et non d'effectuer les opérations demandées : nous y reviendrons plus tard.
– Les opérations `+ - * / ^ !` seront supportés.

La question des parenthèses

Maintenant, il reste encore à trouver comment gérer les parenthèses. Trois idées me sont venues à l'esprit pour les gérer :

– Créer une classe supplémentaire, spécifique aux parenthèses, ce qui permettrait d'en stocker comme tous les autres objets.
– Utiliser un système de « priorités », qui fait associer chaque objet du tableau dynamique d'objets à un nombre indiquant à quelle profondeur il se trouve.
– Utiliser un système d'imbrications, qui permet de stocker des expressions dans des expressions.

Prenons par exemple la saisie suivante :

```
2*(2 + 3^(4/(5 - 6))) - 1
```

Dans la première solution, nous stockerions cette chaine dans le tableau suivant :

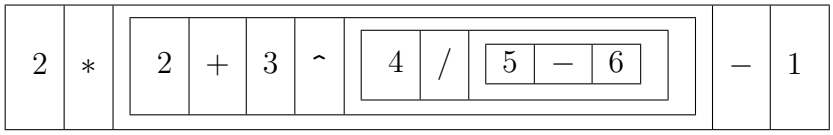
2	∗	(	2	+	3	^	(	4	/	(	5	−	6	)	)	)	−	1

La deuxième stockera les éléments à l'aide de deux tableaux :

2	∗	2	+	3	^	4	/	5	−	6	−	1
0	0	1	1	1	1	2	2	3	3	3	0	0

Les éléments en-dehors de toute parenthèse ont la priorité 0. Celles dans le premier niveau de parenthèses (`2 + 3^(4/(5 - 6))`) ont la priorité 1, celles dans les parenthèses imbriquées dans les précédentes (`4/(5 - 6)`) ont la priorité 2, et les expressions les plus profondes (`5 - 6`) ont la priorité 3.

Et la troisième ainsi :



Tout ce qui est entre parenthèses est considéré comme un objet unique. Des objets peuvent en contenir d'autres, qui peuvent eux-mêmes en contenir d'autres, et ainsi de suite.

J'ai éliminé la première possibilité, car je ne considère pas utile de créer une classe juste pour stocker des parenthèses. De plus, les deux autres dispensent de créer une fonction recherchant les parenthèses correspondantes lorsqu'on effectue une opération. La deuxième est intéressante, puisqu'on peut directement savoir à quelle profondeur se situe un élément, mais j'ai opté pour la troisième solution, car cela facilite certaines choses. Par exemple lorsqu'on a une expression entre parenthèses élevée à une puissance, telle que `4 + (1 + 2)^3`. La deuxième option imposerait de rechercher tous les éléments de priorité 1 se trouvant devant la puissance, alors dans la troisième, `(1 + 2)` étant considérée comme une seule entité, on sait directement ce qui est élevé à la puissance.

3.2.2 Reconnaissance et stockage d'opérations numériques avec parenthèses

C'est là où le concept de collection hétérogène fera une concrète apparition. Nous disposons de tous les ingrédients pour créer une classe « `NumericExpression` » contenant des chaines d'opérations numériques quelconques telles que `2(3 + 4/5)^6`. La classe se composera juste d'un `vector` d'objets comme attribut.

```
1 class NumericExpression : public Object {
2     private:
3     std::vector<Object*> _o; // Stocke les objets
4     // ...
5
6     public:
7     // ...
8 };
```

Comme mentionné avant, il s'agit en réalité d'un tableau dynamique de pointeurs, d'où la présence de l'étoile `*`. La classe `NumericExpression` hérite d'`Object` pour pouvoir utiliser notre méthode de gestion des parenthèses. La suite se consacrera sur l'élaboration des méthodes clés de cette classe.

Validité d'une entrée

La première chose à faire, est de vérifier que la chaine entrée est bien valide. Par exemple, `2 + 3` est valide, alors que `2 + 3 +` ne l'est pas. On créera une méthode appelée « isValid » qui effectuera ce travail. Il faut alors trouver toutes les règles à vérifier permettant de trancher cette validité :

- Les espaces sont ignorées.
- Présence de caractère(s) incorrect(s) : dans une chaine d'opérations numérique, pour tous les caractères, seuls des chiffres, des parenthèses, des points ou virgules (par exemple $2,5 = \frac{5}{2}$), la lettre i pour l'unité imaginaire, et les opérations sont permis. N'importe quel autre caractère (symbole invalide, lettre autre que i,...) rend la saisie invalide.
- Le premier caractère ne peut pas être une opération (sauf - et + pour désigner des nombres avec signe), ni une parenthèse fermante. Par exemple, `*32 + 16`, `^64 + 48`, `)5 - 9` sont invalides. Mais, `-2/3 + 4` et `+37.6` sont valides. On tolère le fait de commencer par une virgule, par exemple `.99^2` correspond à `0.99^2`, et aussi les saisies comme `3.*3 = 3*3`.
- Dans la chaine d'opération, il doit y avoir au moins un nombre, sans quoi elle n'aurait pas de sens. À noter que l'absence de chiffre n'est pas éliminatoire, puisque l'expression `i^i` est valide.
- La chaine ne peut pas se terminer par une opération `- + * / ^`, ni par une parenthèse ouvrante. Exemples : `4^2+`, `5(`, ...
- Une parenthèse ouvrante doit être fermée par exactement une parenthèse fermante (par exemple, `(2 + 3))` est invalide). Cela implique également qu'il doit y avoir autant de parenthèses ouvrantes que fermantes.
- En général, deux opérateurs ne peuvent pas se suivre (par exemple `2**3`, `5+/7`,...), mais certaines combinaisons sont acceptées. Liste de ces combinaisons admissibles :
  - `a--b = a + b`, `a-+b = a - b`
  - `a+-b = a - b`, `a++b = a + b`
  - `a*±b = a(±b)`, `a/±b = ±a/b`

  - Factorielles devant les opérations (mais pas après). On a aussi le droit de mettre plusieurs factorielles qui se suivent : `3!! = (3!)! = 6! = 720`.
- Il ne peut pas y avoir une parenthèse ouvrante directement suivie par une parenthèse fermante (sinon, on aurait une parenthèse vide).
- Il ne peut pas y avoir deux virgules qui se suivent.
- Il ne peut pas y avoir deux virgules dans un nombre, par exemple pour `23.45.6`.

Normalement, ces règles devraient suffire pour avoir un analyseur syntaxique numérique suffisamment performant ; ces règles me semblent être un bon compromis proposant un taux complexité/rigueur acceptable pour une utilisation courante.

Interprétation d'une entrée

Une fois la validation effectuée, on peut interpréter la saisie, et créer la chaine numérique. Si la saisie était invalide, on peut indiquer que la chaine est indéfinie, ou contient juste le nombre 0. L'interprétation consiste à associer chaque caractère ou groupe de caractère à un objet mathématique qu'on ajoute à la chaine. La validation effectuée précédemment dispense d'écrire plusieurs tests durant l'interprétation (pas besoin de vérifier à chaque fois par exemple si un caractère est valide, puisque cela a déjà été fait), et permet donc un code plus lisible.

Ici également, on suivra des règles qui nous aideront pour l'implémentation. D'abord, concentrons-nous aux expressions sans parenthèses :

- On stockera les groupes de chiffres dans des `NNumber`, `ZNumber`, `QNumber`, ou `CQNumber` suivant le cas.
  - Les groupes de chiffres qui ne sont pas dans les cas ci-dessous sont stockés dans des `NNumber`.
  - Si le nombre est précédé d'un signe négatif, et que ce signe se trouve soit au début de l'expression, soit juste après un autre opérateur, soit juste après une parenthèse ouvrante (exemples : `1 + (-2), 2*-3, 3^-4`), on stocke le nombre dans un `ZNumber`, avec le signe qui ne sera pas stocké en tant qu'opération. Sinon, dans le cas `2 - 1`, le 1 est stocké positivement dans un `NNumber`, et est précédé d'un opérateur de soustraction.
  - Dans les cas où le nombre contient un point ou une virgule, on le stocke directement dans un `QNumber`.
  - Seul le nombre i est stocké dans un `CQNumber`. S'il est précédé ou suivi par un nombre, deux nombres distincts seront stockés. Le produit est effectué plus tard.
- Les opérations sont stockées dans la classe `Operation`, à l'exception des cas engendrant un `ZNumber`. Dans le cas des additions et soustractions qui se suivent, par exemple `2+-3`, on stockera comme si on avait écrit `2-3`.

Maintenant, que se passerait-il s'il y avait des parenthèses ? Comme nous pouvons imbriquer des objets dans d'autres, il faudra utiliser un procédé appelé la *récursivité*, c'est-à-dire le fait qu'une fonction s'appelle elle-même. Ceci nous sera utile : avec les règles ci-dessus, nous pouvons écrire une fonction interprétant les expressions sans parenthèses. Dès que des parenthèses apparaissent, la fonction qui effectue l'interprétation s'appellera elle-même pour analyser la sous-chaine numérique, celle contenue dans les parenthèses. Une fois qu'on arrive à la parenthèse fermante, la sous-chaine numérique est créée, et sera stockée en tant qu'élément unique dans la chaine principale : un `NumericExpression` peut en contenir d'autres. Si, dans la sous-chaine, il se trouvait encore d'autres parenthèses, on continuera la récursivité jusqu'à atteindre le niveau le plus profond.

3.2.3 Calcul d'opérations numériques

Maintenant que nous savons vérifier, interpréter, et stocker une chaine d'opérations numériques quelconque, nous pouvons nous intéresser à comment les résoudre. Nous allons créer une fonction effectuant les calculs de la chaine numérique. Dans un premier temps, concentrons-nous sur le calcul d'opérations sans parenthèses. Une fois ceci effectué, quelques adaptations seront faites pour les gérer. Dans ces opérations, il faut s'interroger sur la question de la priorité de chaque opération : laquelle effectuer en premier, deuxième,... ?

J'ai décidé d'effectuer les opérations dans l'ordre suivant, qui est basée sur les conventions (puissances et racines en premier, ensuite produits et quotients, puis sommes et additions ; dans un même niveau, on les effectue de gauche à droite) :

- Les contenus des parenthèses (voir ci-dessous).
- Les factorielles. Par exemple, `2^3! = 2^6 = 64` et `3!^2 = 6^2 = 36`. J'ai choisi d'effectuer les factorielles avant les puissances, parce dans que le cas `3!^2`, on est obligé de faire la factorielle avant. S'il y a plusieurs factorielles, on les effectue de gauche à droite.
- Les puissances (incluant les racines comme `4^(1/2)`), de gauche à droite.
- Les multiplications et divisions, de gauche à droite.
- Les additions et soustractions, toujours de gauche à droite.

Ainsi, on aura par exemple :

```
   2 + 3*4/5 - 6/7^3!
= 2 + 3*4/5 - 6/7^6
= 2 + 3*4/5 - 6/117649
= 2 + 12/5 - 6/117649
= 22/5 - 6/117649
= 2588248/588245
```

Lorsque des parenthèses sont utilisées, on utilisera la récursivité, comme pour l'interprétation. Il ne faut pas oublier que les objets entre parenthèses sont stockés dans une seule entité. Lorsqu'un objet de type `NumericExpression` est détecté, cette fonction qui résout les opérations numériques s'appellera elle-même pour résoudre celle de cet objet. Par exemple, dans `2 + 3(4 - 5) + 6`, l'entité `(4 - 5)` sera détectée, et la fonction de résolution s'appellera elle-même pour effectuer cette opération. Une fois cela fait, la récursivité s'arrête, et l'expression deviendra `2 + 3*9 + 6`, puis on continue en cherchant les factorielles, puissances, produits et quotients,...

Bien sûr, on peut appliquer cette méthode avec des chaines d'opérations aussi longues qu'imaginables [1], avec autant de parenthèses possibles. Si on a une expression avec deux niveaux de parenthèses, par exemple `2 + 3^(4/(5 - 6))`, notre implémentation détectera l'entité `(4/(5 - 6))` avec la récursivité. Cette récursivité détectera ensuite l'entité `(5 - 6)` qui sera effectuée avant toute autre chose. Puis, elle effectuera `(4/(-1))`, et finalement `2 + 3^(-4) = 2 + 1/81 = 163/81`.

Notez que cette façon de stocker des expressions nous permet également de gérer facilement les cas non simplifiables comme `2(i^i)`, car les chaines d'opérations numériques peuvent contenir d'autres chaines d'opérations numériques. Il sera intéressant de permettre des opérations/simplifications telles que `2^(1/2)*4 = 2^(1/2)*2^2 = 2^(1/2 + 2) = 2^(5/2)`, mais on ne les gérera pas pour le moment, car ceci demandera beaucoup de temps, sachant que le but de ce programme est surtout de gérer sans souci les nombres

1. En pratique, il existe un niveau maximal de récursivité, notamment pour éviter les boucles infinies dans le cas où une fonction ne fait que s'appeler elle-même. Mes recherches sur le sujet m'ont permis d'établir qu'en `C++`, la récursivité était limitée par la taille de la pile (*stack* en anglais), et qu'il n'y avait pas de limite codée en dur.

rationnels complexes. `(2^3)^4 = 8^4 = 4096` sera cependant effectué sans problème avec les méthodes ci-dessus.

3.2.4 Calcul des racines d'un polynôme

Maintenant que nous pouvons stocker n'importe quelle expression numérique sans nombres transcendants, il est possible de stocker les racines d'un polynôme. Comme promis dans un chapitre précédent, nous allons nous concentrer sur le calcul des racines des polynômes. Nous n'allons seulement résoudre exactement les polynômes à une seule variable, de degré 1 et 2. Leur résolution est très simple :

Polynôme de degré 1 Il s'agit des polynômes du type $ax+b$, où a et b sont des coefficients rationnels complexes (dans le cas de notre logiciel), et x la variable. L'unique racine du polynôme est donnée par : $x = -\frac{b}{a}$.

Polynôme de degré 2 Il s'agit des polynômes du type $ax^2 + bx + c$, où a, b et c sont des coefficients rationnels complexes, et x la variable. Les deux racines du polynôme sont donnés par la formule de Viète : $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$. Le $\pm$ indique qu'une racine s'obtient en remplaçant ce symbole par une soustraction, et l'autre par un addition. L'implémentation des expressions numériques permet notamment de stocker des racines comme $x = \frac{1+\sqrt{5}}{2}$.

Méthode de Newton

Nous ne proposerons pas la résolution exacte de polynôme avec un degré plus grand. Cependant, nous allons proposer une alternative : la résolution approximée à l'aide de la méthode de Newton. Elle ne fonctionne que pour des polynômes à coefficients réels. Comme pour le calcul de la racine n-ième, il s'agit d'un algorithme utilisant la récurrence. La formule fait appel aux dérivées, et nécessite de connaitre une approximation très grossière de la racine recherchée. On procède ainsi :

- On désire calculer une racine d'un polynôme f difficile, par exemple $f(x) = -4x^5 - 4x^4 - x^3 - 11x^2 - x + 14$.
- On choisit un nombre arbitraire initial $x_i = x_0$, proche de la vraie racine. Avec la représentation graphique (fonction qui sera proposée plus tard), nous avons que la racine vaut un peu moins que 1 (en vrai, elle vaut environ 0.866841).
- Un nombre x_{i+1} plus proche de la racine recherchée du polynôme que x_i peut être calculé à l'aide de la formule suivante :

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

- On peut réutiliser autant de fois la formule ci-dessus pour obtenir des nombres de plus en plus proches de la vraie racine, jusqu'à ce qu'on atteint la précision désirée. Dans ce cas, on remplace x_i par la dernière valeur calculée en utilisant cet formule. Nous nous contenterons de mentionner cette formule, sans la démontrer, ni expliquer son fonctionnement.

Il faut savoir que la valeur initiale doit satisfaire certains critères, sans quoi les itérations ne convergeront pas vers la racine recherchée. Nous n'exposerons pas ces critères ici : l'utilisateur se contentera de modifier manuellement la valeur initiale, jusqu'à ce qu'il arrive à approximer sa racine en lançant autant d'itérations que nécessaire.

3.3 L'analyseur syntaxique (algébrique)

3.3.1 Par rapport à l'analyseur numérique

Comme nous avons déjà créé l'analyseur syntaxique numérique, il ne s'agit pas de recommencer de zéro, mais de réutiliser ce qui a déjà été fait pour implémenter son homologue numérique. Pour stocker des expressions algébriques, on utilisera la même structure : tableau dynamique d'objets. La classe les stockant s'appelle `AlgebraicExpression`.

```
1 class AlgebraicExpression : public Object {
2     private:
3     std::vector<Object*> _o; // Stocke les objets
4     // ...
5
6     public:
7     // ...
8 };
```

3.3.2 Détection et stockage

Validité d'une entrée La validité d'une entrée se vérifie de la même manière que pour les expressions numériques : interdiction d'avoir deux opérateurs qui se suivent à part certains cas particuliers, avoir autant de parenthèses fermantes qu'ouvrantes,... La seule véritable différence réside du fait qu'on accepte également les variables, et donc que les lettres ne sont plus considérées comme invalides. On autorisera plusieurs lettres l'une à côté des autres, quel que soit leur nombre. Et comme nous avions établi que nous ne reconnaitrons que les variables dont le nom n'a qu'une lettre, l'expression « `polynome` » sera par exemple interprétée comme un polynôme de sept variables, tous de degré un, sauf la variable `o` qui est de degré deux (`polynome` = `poolynme` = `po^2lynme`). On ne proposera pas de reconnaissance de mots pour le moment.

Interprétation et stockage d'une expression algébrique Là aussi, il n'y a pas vraiment de différences. Les lettres sont stockées dans des variables, et le reste comme pour les expressions numériques. Nous ne créerons pas de « calculateur » d'opérations algébriques au même titre que les opérations numériques. Une telle fonction aurait été pratique pour effectuer par exemple des simplifications du type `(x^(1/2))^4 = x^2`, mais ceci aurait été trop long à implémenter. Bien que nous pouvons stocker n'importe quelle expression algébrique, nous n'allons nous en servir qu'en tant qu'intermédiaire pour reconnaitre et stocker les polynômes.

3.3.3 Reconnaissance des polynômes

Maintenant que nous disposons d'un outil pour stocker toute expression algébrique, réfléchissons à un moyen de reconnaitre les polynômes à partir d'une chaine de caractères.

Pour cela, il faut tester tous les cas qui empêchent l'expression d'être un polynôme. Si une variable est élevée à une puissance non naturelle, ce n'est pas un polynôme (par exemple `x^(1/2)`). Si l'expression contient un dénominateur où il y a une variable (par exemple `(x + 1)/(x - 1)` ou `1/x`), ce n'est pas un polynôme. Si l'expression contient une élévation à une variable ou une expression contenant une variable (par exemple `2^x` ou `x^(x^2 + 1)`), ce n'est pas un polynôme. Dans tous les autres cas, nous aurons affaire à des

polynômes. Ce test sera le travail d'une méthode de la classe `Polynomial` appelée `isValid`. Bien évidemment, une expression invalide implique un polynôme invalide.

3.3.4 Stockage des polynômes

Une fois la validité de la chaine de caractères établie, réfléchissons à comment nous pourrions enregistrer le polynôme à l'aide d'un constructeur de la classe `Polynomial` prenant un `string` en argument. Nous ne gèrerons que les polynômes à coefficients complexes rationnels, et ne proposerons pas le stockage des polynômes avec des coefficients comme $\sqrt{2}$.

Deux cas de figure peuvent apparaitre : l'utilisateur pourrait entrer soit un polynôme entièrement développé (par exemple x^2+2x+1), soit un polynôme factorisé (comme $(x+1)^2$ ou $(x-1)(x+1)$) ou mixte (comme $(x+1)^2+3$). Il faut savoir qu'il n'est possible que de stocker des polynômes entièrement développés, et donc qu'on développera si nécessaire le polynôme avant de l'enregistrer. J'ai élaboré les étapes suivantes, qui permettent de traiter les deux cas. Le principe consiste à stocker dans un polynôme temporaire le produit de polynôme qu'il y a entre chaque addition ou soustraction, et d'ajouter ce polynôme temporaire au polynôme en cours de construction dès qu'on change de terme :

- On interprète la chaine de caractères avec `AlgebraicExpression` pour le convertir en éléments. Si la chaine n'est pas un polynôme valide, on s'arrête et le polynôme construit est nul. Sinon, on suit les étapes suivantes.
- On crée un polynôme temporaire valant 1. De plus, on déclare une booléenne initialisée à `faux` qui indique si le polynôme temporaire en cours de stockage est précédé d'une soustraction (par exemple `2 - x`). Puis, on commence une boucle qui va parcourir les éléments de l'expression algébrique de gauche à droite.
- Si on détecte un élément de type `AlgebraicExpression`, on utilise la récursivité pour trouver le polynôme contenu dans les parenthèses. Ainsi, si on a des polynômes imbriqués, par exemple dans `(2x^2(x + 1)^3 + x) + x^4`, la récursivité permettra de les gérer. Puis, on a les deux cas suivants :
  - Si l'expression est suivie d'une puissance `^`, qui sera forcément elle-même suivie d'un nombre naturel (car on a auparavant testé qu'il s'agissait d'un polynôme valide), on multiplie cette expression par elle-même autant de fois que l'exposant.
  - Si elle n'est pas suivie par une puissance, on laisse le polynôme contenu dans les parenthèses tel qu'il a été interprété par la récursivité.

  Puis, on multiplie le polynôme temporaire par ce polynôme entre parenthèses.
- Lorsqu'on détecte un nombre qui n'est pas précédé d'une puissance `^`, par exemple dans les cas `3x^2` ou `2(x + 1)`, on multiplie le polynôme temporaire par ce nombre.
- Lorsqu'on détecte une variable, nous devons traiter ces cas :
  - Si la variable est suivi d'une puissance, qui sera forcément lui-même suivi d'un nombre naturel, on multiplie le polynôme temporaire par un monôme contenant seulement cette variable avec cette puissance.
  - Sinon, on multiplie le polynôme temporaire par un monôme contenant juste cette variable à la puissance 1.
- On peut également gérer les divisions par une constante, par exemple dans `3x/2`. Dans ce cas, on vérifie si le nombre est précédé d'une division `/`, et on divise le polynôme temporaire par ce nombre.
- Lorsqu'on détecte une addition ou une soustraction, on ajoute le polynôme temporaire au polynôme en cours de construction. Si la booléenne de la soustraction est vraie, il faut ajouter l'opposé de ce polynôme. Puis, on réinitialise le polynôme temporaire (toujours avec la valeur de 1). Si c'est une soustraction qui est détectée, il faut affecter la booléenne à vrai, sinon il faut la mettre à faux.

4. Tenseurs et outils pour la représentation graphique

4.1 Tenseurs

4.1.1 Qu'est-ce qu'un tenseur ?

Vous savez ce qu'est un nombre, n'est-ce pas ? Vous avez ensuite peut-être appris ce qu'est un vecteur : un ensemble de nombres appelés composantes, qui permet de représenter par exemple des directions. Par exemple, le vecteur $\vec{u} = (0, 1, 2)$ est un vecteur de trois composantes. Après les vecteurs, vous avez les matrices, qui peuvent grossièrement être vues comme des vecteurs en deux dimensions. Et viennent maintenant les tenseurs, qui sont en fait une généralisation des vecteurs et des matrices.

On dit qu'un vecteur est un tenseur d'ordre 1, et une matrice un tenseur d'ordre 2, du fait que les coordonnées d'un vecteur s'écrivent sur une seule dimension (ligne ou colonne). et une matrice sur deux dimensions (un tableau avec lignes et colonnes). Mais on peut continuer ! Un tenseur d'ordre 3 est une « matrice en 3D », donc des coordonnées en lignes, en colonnes, et en profondeur. Maintenant, imaginez les tenseurs d'ordre 4 et plus...

4.1.2 Utilité

Les tenseurs pourront nous servir à stocker des couples préimages-images, qui nous serviront pour la représentation graphique. Comme un tenseur peut être d'une dimension quelconque, on pourra également avoir des couples préimages-images de polynômes à nombre de variables quelconques. Ils ont d'autres applications, notamment en physique, qui ne seront pas traitées ici.

4.1.3 Implémentation de la classe

Comment va-t-on implémenter les tenseurs dans une classe ? L'idée d'imbriquer les `vector` du `C++` pour stocker les éléments d'un tenseur d'ordre dynamique me semble un peu farfelue : par exemple, si on avait un tenseur d'ordre trois, il faudra créer un attribut de type `vector<vector<vector<type de nombre>>>` ! De plus, bonne chance pour faire en sorte que le programme s'adapte à l'ordre du tenseur, et change automatiquement le type, en imbriquant le nombre de `vector` nécessaire... D'ailleurs, il ne me semble pas qu'il soit possible de faire ce genre de choses en `C++`, ou en tout cas pas d'une manière suffisamment simple pour être envisageable.

Que fait-on dans ce cas ? Une autre solution serait de simplifier le problème en n'utilisant que des tableaux dynamiques de dimension 1. Soit un tenseur d'ordre deux de dimensions 4×3 :

$$\begin{bmatrix} c_{0,0} & c_{1,0} & c_{2,0} & c_{3,0} \\ c_{0,1} & c_{1,1} & c_{2,1} & c_{3,1} \\ c_{0,2} & c_{1,2} & c_{2,2} & c_{3,2} \end{bmatrix}$$

Mathématiquement, ceci changera surement la nature de ce tenseur, mais rien n'empêche de le stocker informatiquement dans un attribut de type `vector` de la manière suivante :

$$\begin{bmatrix} c_{0,0} & c_{1,0} & c_{2,0} & c_{3,0} & c_{0,1} & c_{1,1} & c_{2,1} & c_{3,1} & c_{0,2} & c_{1,2} & c_{2,2} & c_{3,2} \end{bmatrix}$$

On fait en sorte qu'`Algébra` sache que ce `vector` de douze composantes contient en fait les coefficients d'un tenseur d'ordre deux, et qu'on change de ligne toutes les quatre cases. Cela permet de reconstituer le tenseur d'ordre deux précédent. On stockera donc dans un autre attribut les dimensions du tenseur, qui nous permettront de reconstituer le tenseur à partir du tableau dynamique contenant les éléments sur une dimension.

Cela s'applique pour n'importe quel ordre, par exemple un tenseur d'ordre trois de dimensions $3 \times 2 \times 2$ pourrait être stocké ainsi, dans l'attribut contenant les composantes :

$$\begin{bmatrix} c_{0,0,0} & c_{1,0,0} & c_{2,0,0} & c_{0,1,0} & c_{1,1,0} & c_{2,1,0} & c_{0,0,1} & c_{1,0,1} & c_{2,0,1} & c_{0,1,1} & c_{1,1,1} & c_{2,1,1} \end{bmatrix}$$

Ici, on fait en sorte que le programme sache qu'il s'agit d'un tenseur d'ordre trois, qu'il faut changer de ligne toutes les trois cases, et de profondeur toutes les $2 \times 3 = 6$ cases. On stockera les dimensions du tenseur dans l'autre tableau dynamique.

En résumé, les attributs d'une classe qu'on appellera simplement « `Tensor` » seront deux tableaux dynamiques : un contenant toutes les composantes du tenseur (ce sera un `vector` de `CQNumber`), et l'autre contenant les dimensions du tenseur (un `vector` d'`unsigned int`). Implémentation (en pratique, il s'agit d'un patron de classe) :

```
1 class Tensor {
2     private:
3     std::vector<CQNumber> _coeffs; // Stocke les composantes
4     std::vector<unsigned int> _dimensions; // Stocke les dimensions du
          tenseur
5
6     public:
7     // ...
8 };
```

Note sur les indices Il faut savoir que la convention veut que le premier indice d'un élément d'un tenseur représente la ligne à laquelle elle se trouve, et le deuxième indice la colonne. Dans mes explications, et le code, c'est le contraire. Au moment de la rédaction de ce chapitre, et de l'écriture de la classe sur les tenseurs, je ne connaissais pas cette convention, et j'ai trouvé plus logique que le premier indice donne la colonne, et le deuxième la ligne, en m'inspirant de x et y qui donnent respectivement la position horizontale et la position verticale. Dans le futur, je réécrirai le code et les explications selon les conventions.

Passage de 1D à nD

Maintenant, il faut trouver un moyen de convertir le `vector` contenant les nombres en un véritable tenseur, et réciproquement. De passer de la 1D du `vector` à la nD du tenseur et inversément. Autrement dit, si on choisit un nombre du tableau dynamique, quelles seraient ses « coordonnées » (quelle colonne, quelle ligne,...) si on mettait ce `vector` sous la forme

d'un tenseur ? Ou l'inverse, si on donne des coordonnées, quelle serait l'indice de ce nombre, lorsqu'on revient en « mode 1D ».

On va réfléchir dans un premier temps à la première possibilité (passage du `vector` au tenseur). Si on prend par exemple un tenseur d'ordre 3 de dimensions $5 \times 4 \times 3$, c'est-à-dire un tenseur de cinq colonnes, quatre lignes, et trois profondeurs (total de 60 cases). Voici une vue développée d'un tel tenseur (les nombres représentent l'indice de chaque case si on représentait ce tenseur en une dimension) :

$$\left[\begin{array}{ccccc|ccccc|ccccc} 0 & 1 & 2 & 3 & 4 & 20 & 21 & 22 & 23 & 24 & 40 & 41 & 42 & 43 & 44 \\ 5 & 6 & 7 & 8 & 9 & 25 & 26 & 27 & 28 & 29 & 45 & 46 & 47 & 48 & 49 \\ 10 & 11 & 12 & 13 & 14 & 30 & 31 & 32 & 33 & 34 & 50 & 51 & 52 & 53 & 54 \\ 15 & 16 & 17 & 18 & 19 & 35 & 36 & 37 & 38 & 39 & 55 & 56 & 57 & 58 & 59 \end{array}\right]$$

Chaque matrice a une profondeur différente (troisième coordonnée différente), et est séparée par une barre verticale. Attention, par exemple les nombres 20, 25, 30, et 35 se trouvent sur la première colonne, et non la sixième (c'est une vue développée).

Le but sera de répondre à par exemple la question suivante : où se trouve la case d'indice 33, quelles sont ses coordonnées ? Pour que nous nous comprenions bien, je vous donne la réponse : cette case se trouve à la quatrième colonne, troisième ligne, deuxième profondeur. Elle a donc informatiquement les coordonnées $(3; 2; 1)$: souvenez-vous, en `C++`, tout commence à zéro, la première colonne est de numéro zéro, la deuxième de numéro une, etc.

Maintenant, comment peut-on programmer un code nous donnant ces coordonnées à partir de l'indice ? Commençons par trouver le calcul de la colonne, c'est-à-dire la première coordonnée. Si vous regardez bien, tous les nombres étant un multiple de 5 se trouvent sur la première colonne (c'est-à-dire la coordonnée 0). S'il ne sont pas divisibles par 5, mais donnent un reste de 1 (par exemple les cases 1, 6, 31, 56), ils se situent à la deuxième colonne (coordonnée 1). En généralisant ce raisonnement, on conclut vite que la première coordonnée n'est que le reste de la division par cinq, soit $n \mod 5$, si n est l'indice de la case considérée. En faisant $33 \mod 5$, on trouve 3, ce qui confirme bien la première coordonnée de cette case.

Peut-on appliquer ceci pour les autres coordonnées ? Pas si vite. Faites $33 \mod 4 = 1 \neq 2$ (modulo 4 car il y a 4 colonnes), et c'est raté. Il faut raisonner d'une autre manière. Comme chaque case ne peut avoir que 0, 1, 2, ou 3 pour la deuxième coordonnée, nous savons déjà qu'il y a obligatoirement un modulo 4 dans le calcul. Mais pourquoi n'est-ce pas aussi simple qu'avant ? Si on prend par exemple le groupe 0 1 2 3 4, tous ces nombres sont dans des colonnes différentes, mais dans la même ligne. En revanche, pour le groupe 5 6 7 8 9, la ligne a été incrémentée, alors que les colonnes respectives sont inchangées. Si on ajoute cinq, on change d'une ligne. En réfléchissant un peu, on en vient à la conclusion suivante : la partie entière de la division de l'indice par cinq donne la ligne, sans tenir compte des dimensions supplémentaires (20 21 22 23 24 seraient sur la ligne numéro 4 au lieu de 0). Or, le modulo 4 règle ce problème. Ainsi, la deuxième coordonnée se calcule avec $\lfloor \frac{n}{5} \rfloor \mod 4$. Démonstration : $\lfloor \frac{33}{5} \rfloor \mod 4 = 6 \mod 4 = 2$: ça fonctionne !

Cette fois, le raisonnement sera similaire pour les dimensions supplémentaires. Comme la troisième coordonnée varie tous les $5 \times 4 = 20$ indices, on déduit facilement qu'elle se calcule par $\lfloor \frac{n}{20} \rfloor \mod 3$. $\lfloor \frac{33}{20} \rfloor \mod 3 = 1 \mod 3 = 1$ donne la réponse attendue. Vérifiez avec d'autres cases, si vous le souhaitez.

On peut généraliser ceci pour n'importe quel tenseur de n'importe quel ordre ! Voici la règle formelle dont on peut tirer du raisonnement, donnant la m-ième coordonnée x_m de la case d'indice n, pour un tenseur d'ordre o, de dimensions $d_1 \times d_2 \times d_3 \times ... \times d_o$:

$$x_m = \lfloor \frac{n}{\prod_{i=1}^{m-1} d_i} \rfloor \mod d_m$$

Passage de nD à 1D

Pour faire le passage inverse, le raisonnement est plus simple. Si on reprend le même exemple : quelle est l'indice du nombre se situant à la position $(3; 2; 1)$ d'un tenseur de dimensions $5 \times 4 \times 3$? On sait que si on avance d'une unité dans la première coordonnée, on se déplace d'un indice. Si on change de ligne, on se déplace de cinq indices. Si on change de profondeur, on se déplace de $5 \times 4 = 20$ indices.

Ainsi, on n'a plus qu'à faire :

$$\begin{aligned} n &= x_1 \times 1 + x_2 d_1 + x_3 d_1 d_2 \\ &= 3 \times 1 + 2 \times 5 + 1 \times 5 \times 4 \\ &= 3 + 10 + 20 = 33 \end{aligned}$$

On trouve bien que l'indice unidimensionnel est 33. Pour généraliser, on pourrait écrire :

$$n = \sum_{i=1}^{o} (x_i \prod_{j=1}^{i-1} d_j)$$

n est l'indice, o est l'ordre du tenseur, donc le nombre de coordonnées, x_i une coordonnée, d_j une dimension du tenseur.

4.1.4 Méthodes

Nous avons maintenant de bonnes bases pour gérer nos tenseurs. Parlons maintenant de quelques fonctions « indispensables » pour une telle classe.

Affichage

Afficher des nombres est simple : il suffit d'afficher tous les chiffres dans l'ordre. On pourrait dire la même chose pour les tenseurs : il suffit d'afficher chaque nombre dans l'ordre. C'est très facile pour les tenseurs d'ordre un (afficher en une ligne les valeurs les unes à côté des autres) et deux (idem, mais en lignes et colonnes) : il suffit de faire une ou deux boucles for. Mais pour les tenseurs d'ordre trois et plus ? L'écran n'a que deux dimensions : par conséquent, on sera obligé d'afficher ces tenseurs sous une forme décomposée en plusieurs matrices (comme on l'a fait avec l'exemple précédent du tenseur d'ordre trois). Par exemple, on affichera un tenseur d'ordre trois de dimensions $4 \times 3 \times 2$ en deux parties, chacune représentant une profondeur du tenseur.

Voilà comment nous allons procéder : nous allons calculer la taille d'une matrice, en faisant le produit de la première dimension avec la deuxième. Donc, pour le tenseur de dimensions $4 \times 3 \times 2$, on l'affichera en deux matrices de $4 \times 3 = 12$ nombres (deux fois quatre colonnes et trois lignes). Si le tenseur avait comme dimensions $4 \times 5 \times 4$, on afficherait le tenseur en quatre matrices de vingt composantes. Idem pour les dimensions supérieures, si on a un tenseur de dimensions $6 \times 5 \times 4 \times 3$, on l'afficherait en douze matrices de trente composantes, en précisant devant chaque matrice où elle se situe dans les dimensions

supérieures. Le nombre de matrices à afficher est le nombre de coordonnées divisé par la taille d'une matrice.

Afficher les matrices se fait en utilisant trois boucles `for` imbriquées, la première permettant d'incrémenter les matrices (matrice 1, matrice 2,...), et les deux autres d'afficher les composantes sous la forme de matrices. Si on appelle l'indice des matrices i_{mat}, et les deux autres i et j, et qu'on décide que i représente les lignes et j les colonnes de chaque matrice, l'indice de la composante à afficher est (d_1 est la première dimension du tenseur)

$$n = i_{mat} \times \text{taille d'une matrice} + d_1 \times i + j$$

Nous savons maintenant afficher le contenu d'une matrice. Pour afficher la position de la matrice, on imbriquera une autre boucle dans la boucle incrémentant les i_{mat}, avant les deux qui affichent la matrice. Celle-ci incrémentera les dimensions ; on initialise l'indice i_{dim} à trois, pour directement commencer par la troisième dimension. À chaque itération de la boucle, on calculera où se trouve la matrice pour la i_{dim}ème dimension donnée (à quelle profondeur, à quelle position dans la quatrième dimension, la cinquième,...). L'astuce utilisée ici est de réutiliser la formule permettant de convertir un indice en coordonnées, en l'adaptant pour les matrices : l'indice de la matrice i_{mat} remplace l'indice d'une composante, et dans le produit du dénominateur, on enlève d_1 et d_2. Le modulo ne change pas.

Exemple de résultat (tenseur d'ordre trois de dimensions $5 \times 2 \times 3$) :

```
(i, j, 0) :

37  271 183 118 419
318 8   415 373 107

(i, j, 1) :

432 63  328 126 496
270 301 282 498 415

(i, j, 2) :

6   267 478 263 220
109 175 182 111 1
```

Par exemple, le nombre 282 a pour coordonnées $(2, 1, 1)$.

Redimensionnement

Une fonction qui pourrait être très pratique serait le redimensionnement d'un tenseur. Par exemple, si, pour une raison ou une autre, je souhaitais redimensionner l'exemple précédant en un tenseur d'ordre 2 de dimensions 4×2, j'obtiendrais le tenseur :

```
37  271 183 118
318 8   415 373
```

Ou au contraire, si je décidais de l'agrandir (voire diminuer des dimensions et en augmenter d'autres), par exemple en redimensionnant encore ce tenseur en un autre de dimensions 3×3 pour obtenir ceci :

```
37  271 183
318 8   415
0   0   0
```

Comme on stocke toutes les composantes dans un vecteur, il semble difficile d'imaginer un moyen de redimensionner le tenseur en lui-même : comment faire pour passer d'un

tenseur de dimensions $2 \times 3 \times 4$ à un tenseur de dimensions $3 \times 5 \times 3$, avec uniquement les méthodes `push_back` et `pop_back` proposés par la bibliothèque standard, qui ne servent qu'à ajouter un élément ou supprimer le dernier du tableau ?

En fait, j'ai simplifié le problème : si on sait retirer la dernière ligne [1] ou ajouter une ligne à une dimension donnée, alors on sait redimensionner, car il suffit d'appeler autant de fois qu'on le souhaite la fonction permettant de retirer ou d'ajouter une ligne pour redimensionner le tenseur.

Suppression d'une ligne Commençons par la suppression de la dernière ligne dans une dimension donnée. Il s'agira d'une fonction appelée `delLine` prenant comme unique argument la dimension à laquelle il faut retirer une ligne. Par exemple, si je reprends ce tenseur :

```
37  271 183 118
318 8   415 373
```

Et que je fais `delLine(1)`, elle retirera la dernière colonne (première dimension) :

```
37  271 183
318 8   415
```

Si je fais `delLine(2)`, elle retirera la dernière ligne (deuxième dimension) :

```
37  271 183 118
```

Enfin, si je fais `delLine(3)`, elle retirera la dernière profondeur (troisième dimension). Comme il n'y a qu'une profondeur, en l'effaçant, on se retrouve avec un tenseur vide ! Le comportement sera le même avec des arguments supérieurs à trois.

```
0
```

Comment programmer un tel comportement ? L'idée est de créer un tableau dynamique temporaire, contenant toutes les dimensions du tenseur actuel, mais avec la dimension en argument décrémentée d'une unité. Par exemple, si les dimensions initiales du tenseur sont 4×2, et qu'on retire une ligne dans la première dimension, les nouvelles dimensions stockées dans ce tableau dynamique temporaire seront 3×2. On créera ensuite un tenseur temporaire, prenant les dimensions de ce tableau dynamique nouvellement créé. Et puis, on n'a plus qu'à remplacer les zéros par les anciennes coordonnées correspondantes du tenseur avant la réduction, à l'aide d'une boucle. En jouant avec les conversions 1D vers nD et inversément, on y parvient assez facilement. Puis, on remplace le tenseur actuel par le tenseur temporaire.

Ajout d'une ligne Passons maintenant à la fonction opposée, `addLine` qui ajoute une ligne à la fin d'une dimension donnée. Encore quelques exemples pour s'assurer que nous sommes sur la même longueur d'onde :

```
37  271 183 118
318 8   415 373
```

Si je fais `addLine(1)`, elle ajoutera une colonne (première dimension) :

1. Enfin, ce ne sont pas tout à fait des lignes, par exemple si vous ajoutez une ligne dans la troisième dimension, vous avez en fait ajouté une matrice supplémentaire. Je n'ai simplement pas trouvé de terme générique pour cette action. Je me suis imaginé que certes, c'est une matrice qu'on a ajoutée pour augmenter la profondeur, mais que si on visualise le tenseur en 3D, et qu'on l'observe de profil, on a l'impression qu'on a ajouté une ligne et non une matrice. Et j'ai généralisé pour les dimensions supérieures...

```
37  271 183 118 0
318 8   415 373 0
```

Si je fais `addLine(2)`, elle ajoutera une ligne (deuxième dimension) :

```
37  271 183 118
318 8   415 373
0   0   0   0
```

Enfin, si je fais `addLine(3)`, elle ajoutera une profondeur (troisième dimension) :

```
(i, j, 0) :

37  271 183 118
318 8   415 373

(i, j, 1) :

0   0   0   0
0   0   0   0
```

Si la dimension est encore plus grande, elle créera les dimensions intermédiaires, et la dernière dimension aura deux positions : exemple si je fais `delLine(6)` :

```
(i, j, 0, 0, 0, 0) :

37  271 183 118
318 8   415 373

(i, j, 0, 0, 0, 1) :

0   0   0   0
0   0   0   0
```

L'implémentation de cette fonction se fait quasiment de la même manière que pour la fonction `delLine`. En effet, on créera un tenseur temporaire rempli de zéros, avec une des dimensions augmentée par rapport au tenseur originel, et on copiera les composantes de ce dernier vers le nouveau, avant de procéder au remplacement de l'originel.

Redimensionnement Le problème qui nous paraissait difficile est maintenant très simple : maintenant que nous savons ajouter ou retirer des lignes, il suffit de créer une méthode `resize` qui prend comme paramètres les nouvelles dimensions, et qui se charge d'appeler autant de fois qu'il le faut les deux méthodes précédentes. Pour cela, il y aura une boucle pour gérer chaque dimension, dans laquelle on calcule la différence de dimensions. Ce qui permet de déterminer combien de fois on devra itérer une autre boucle contenue dans celle-ci, dans laquelle est contenue l'appel à la méthode appropriée.

Amélioration On pourrait encore améliorer ces méthodes en proposant les méthodes homologues qui ajouteraient ou retireraient une ligne particulière plutôt que la dernière, en prenant deux arguments. Je ne me suis pas concentré dessus, vu que le but premier est de permettre le redimensionnement du tenseur.

Somme et soustraction

L'addition et la soustraction sont simples pour les tenseurs : il suffit d'additionner toutes les composantes homologues entre elles. En additionnant la première coordonnée du

premier tenseur avec celle du deuxième, la deuxième coordonnée du premier tenseur avec celle du deuxième, jusqu'à la dernière, on obtient le tenseur somme. Même principe pour la soustraction.

Ceci est valable pour deux tenseurs ayant les mêmes dimensions. Mais, bien que ce soit incorrect en mathématiques, nous allons généraliser cela pour des tenseurs de tailles différentes. Par exemple, l'addition entre ces deux tenseurs :

```
T_1 =
440 42  188 479 125
278 358 82  264 213
128 80  99  20  342

T_2 =
400 433
179 213
287 468
59  467
```

Donnera

```
Somme :
840 475 188 479 125
457 571 82  264 213
415 548 99  20  342
59  467 0   0   0
```

Si une composante n'a pas d'homologue, on considérera ce dernier comme nul. Les composantes n'existant dans aucun des deux tenseurs sont nuls dans le tenseur somme. L'implémentation est simple : tout d'abord, le vecteur somme doit disposer de dimensions englobant les deux tenseurs opérandes. T_1 dispose d'une longueur plus élevée, tandis que T_2 a une hauteur plus grande. Donc, le tenseur somme aura la longueur de T_1 et la hauteur de T_2. Si les tenseurs n'étaient pas du même ordre, le tenseur somme serait du plus grand ordre parmi les deux, et prendrait les dimensions du tenseur ayant le plus grand ordre, pour celles qui n'existent pas dans l'autre tenseur. Par exemple, soient deux tenseurs, le premier de dimensions $2 \times 3 \times 4$ et le deuxième de dimensions $3 \times 2 \times 2 \times 4 \times 6$. Le tenseur somme aurait comme dimensions $3 \times 3 \times 4 \times 4 \times 6$.

Ensuite, on redimensionne les deux tenseurs initiaux aux dimensions du tenseur somme, à l'aide de la fonction `resize` fraichement codée. Il n'y a pas de perte de composantes, puisque le tenseur somme englobe les deux. Puis, on additionne toutes les coordonnées entre elles.

Pour la soustraction, on multiplie toutes les composantes du deuxième tenseur par -1, avant d'appeler l'opérateur d'addition qui permettra d'additionner par l'opposé.

4.2 La représentation graphique

4.2.1 Intervalles

Les intervalles nous seront très utiles par la suite. Elles permettront de borner les représentations graphiques à notre guise, pour afficher uniquement la partie du polynôme qu'on souhaite voir. La création d'une telle classe n'a rien de sorcier : elle contient comme attributs les deux extrémités. En plus de cela, on pourrait également gérer les bornes ouvertes et fermées avec deux autres attributs booléens (cela peut être utile si on projette d'implémenter la résolution d'inéquations, par exemple).

```
class Interval {
    private:
    QNumber _a, _b; // Extrémités
    bool _oa, _ob; // Indiquent si ouvert ou fermé

    public:
    // ...
};
```

Dans l'écriture des méthodes, il faudra veiller à ce que b soit toujours supérieur ou égal à a. Il n'y a pas grand-chose à décrire ici, on écrira seulement des constructeurs et méthodes permettant d'affecter, de modifier et d'afficher l'intervalle.

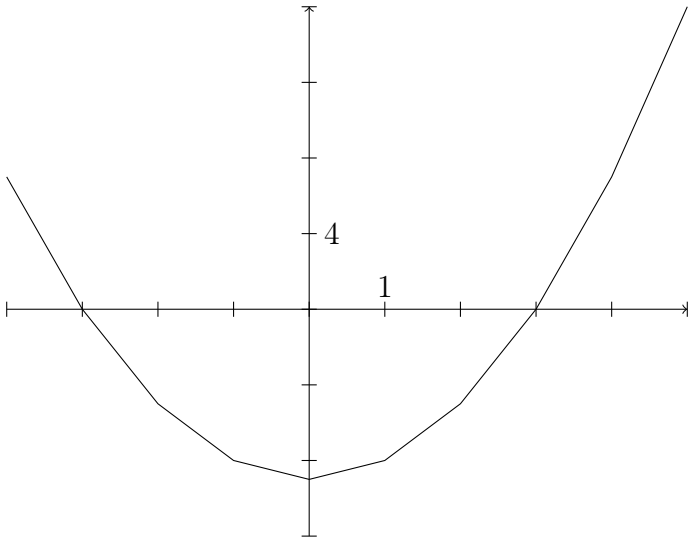
4.2.2 Stockage de données

Avant d'aborder la question informatique, examinons les cas suivants. Soit un polynôme à une variable, disons $x^2 - 9$. Le but sera de proposer un moyen d'évaluer ce polynôme sur un intervalle, par exemple entre -4 et 5, et de stocker les images. Cela nous permettra d'avoir un outil pour la représentation graphique en représentant les points sur un plan, en fonction des préimages et leurs images respectives.

Comme nous ne pouvons pas évaluer tous les nombres réels de cet intervalle, nous nous contenterons d'évaluer en certaines valeurs de x, choisies d'une certaine manière, sachant que plus on choisit de points, plus le graphique sera précis, mais plus le calcul sera lent. Si on décidait d'évaluer dix points de cet intervalle, il serait plus logique de calculer les images pour x égal à $-4, -3, -2, -1, 0, 1, 2, 3, 4, 5$ (précision uniforme sur l'intervalle) que $-4, -3.9, -3.8, -3.7, -3.6, -3, -1, 1, 3, 5$ (très précis sur le début de l'intervalle, mais peu précis sur le reste) : pour la suite, on choisira toujours les points selon cette première manière. Nous aurions alors ce tableau de valeurs :

-4	-3	-2	-1	0	1	2	3	4	5
7	0	-5	-8	-9	-8	-5	0	7	16

Qui donnerait le graphique suivant, si nous avions programmé la représentation graphique :



Pour augmenter la précision, nous pouvons augmenter le nombre de points en diminuant l'écart entre les préimages. Au contraire, si on souhaite un calcul plus rapide, on peut diminuer le nombre de points contre un graphique plus polygonal.

Si nous avions un polynôme à deux variables, comme $x^2 + y$, le principe est le même, nous pouvons choisir d'évaluer ce polynôme sur un intervalle pour x et un autre pour y, par exemple $x \in [-2; 2]$ et $y \in [-3; -1]$, avec un pas de 1 pour les deux variables. On stockera alors les préimages dans deux tableaux, et les images dans un tenseur d'ordre deux :

x	-2	-1	0	1	2

y	-3	-2	-1

1	-2	-3	-2	1
2	-1	-2	-1	2
3	0	-1	0	3

Par exemple, pour $x = -2$ et $y = -1$, $x^2 + y = 3$ (la case tout en bas à gauche).

Si nous voulions enregistrer les préimages/images d'un polynôme de trois variables, nous aurions besoin de trois tableaux dynamiques pour les préimages, et d'un tenseur d'ordre trois pour les images, et ainsi de suite. Ainsi, une classe permettant de stocker les préimages et images sur des intervalles d'un polynôme devra contenir un `vector` de `vector` (pour un nombre variable de tableaux dynamiques de préimages) et un `Tensor` comme attributs. Cette façon de faire permet d'effectuer des opérations très générales, et peut avoir des applications intéressantes, par exemple dans le domaine de l'analyse de données statistiques, qui exige parfois l'étude de beaucoup de variables simultanément, par exemple pour trouver des corrélations entre elles. Cette classe sera appelée « DataPlot » (DonnéesGraphique), ce nom indiquant que son but est de contenir les données « data » (préimages/images) permettant de tracer un graphique « plot ».

```
1 class DataPlot {
2     private:
3     std::vector<std::vector<S> > _pimgs; // Préimages
4     Tensor<T> _imgs; // Images
5
6     public:
7     // ...
8 };
```

Elle ne stocke pas les noms des variables, contrairement à la classe `Polynomial`. Cela n'importe pas, puisqu'on évaluera simultanément toutes les variables du polynôme, donc cette classe ne contiendra que des nombres. À noter également qu'il n'est pas nécessaire que les pas soient de même taille entre chaque élément des intervalles, mais on devra s'arranger pour que chaque élément soit supérieur au précédent, au risque d'aberrations graphiques. En sachant que l'évaluation de polynôme a déjà été écrite, nous avons tous les outils pour implémenter cette classe qui sert d'engrenages pour la représentation graphique de polynômes.

4.2.3 Évaluation

Ici, nous réfléchirons à comment nous y prendre pour remplir les attributs de la classe `DataPlot`, lorsqu'on donne des intervalles pour chaque variable à évaluer, et le polynôme associé. Autrement dit, on essaiera ici de concevoir une méthode « plot [2]» prenant en argument un polynôme à évaluer, des intervalles, et des pas pour chaque intervalle ; ces deux derniers sont stockés dans un `vector` de `pair`. La signature de cette méthode ressemblera alors à [3] :

2. Signifie non seulement « graphique » mais également « tracer », dans le sens calculer les images pour *tracer les points*.

3. Le deuxième argument est appelé « `is` », pour les initiales de « interval » et « step ».

`plot(Polynomial p, std::vector<std::pair<Interval, QNumber> > is).`

Un exemple pourra clarifier le paragraphe précédent. Imaginons que nous voulons évaluer le polynôme $x^3 + xy^2$ sur les intervalles $x \in [-3; 3]$ et $y \in [-1.5; 1]$. Les pas des éléments de chaque intervalle seraient de 1 pour x, et 0.5 pour y. Ainsi, la méthode `plot` prendra comme arguments un paramètre de type `Polynomial` contenant le polynôme $P(x, y) = x^3 + xy^2$, et un `vector` contenant deux couples `pair`, reliant chacun un intervalle ($[-3; 3]$ et $[-1.5; 1]$) et son pas respectif (1 et 0.5).

L'étape suivante de la conception de cette méthode `plot` est de remplir l'attribut des préimages `_pimgs`. En connaissant les intervalles et les pas, vous pouvez établir que les préimages x et y seront les suivants :

```
Préimages x : {-3 ; -2 ; -1 ; 0 ; 1 ; 2 ; 3} (7 éléments)
Préimages y : {-1.5 ; -1 ; -0.5 ; 0 ; 0.5 ; 1} (6 éléments)
```

Ainsi, l'attribut `_pimgs` sera un `vector` contenant deux « sous-`vector` », qui contiennent quant à eux les préimages pour chaque variable. Comme chaque x sera être évalué avec six y différents, on constate qu'il y aura en tout 42 images de $x^3 + xy^2$ évalués. Pour établir ces deux listes informatiquement, on fait deux boucles `for`, une pour récupérer individuellement chaque intervalle et pas, et l'autre qui complètera `_pimgs` en partant de l'extrémité de gauche de l'intervalle, en ajoutant à chaque itération le pas, et en s'arrêtant lorsqu'on atteint l'extrémité de droite.

Une fois que cela est fait, on peut évaluer 42 fois le polynôme, en appelant dans une boucle le nombre de fois nécessaire la méthode `eval` déjà implémentée pour les polynômes, en s'arrangeant pour qu'elle évalue avec comme préimages les éléments de `_pimgs`. Ces évaluations permettront de créer le tenseur d'ordre deux `_imgs` contenant les images du polynôme :

```
-33.75   -12.50   -3.25   0.00    3.25    12.50   33.75
-30.00   -10.00   -2.00   0.00    2.00    10.00   30.00
-27.75   -8.50    -1.25   0.00    1.25    8.50    27.75
-27.00   -8.00    -1.00   0.00    1.00    8.00    27.00
-27.75   -8.50    -1.25   0.00    1.25    8.50    27.75
-30.00   -10.00   -2.00   0.00    2.00    10.00   30.00
```

Par exemple, pour $x = 2$ (sixième élément) et $y = 0.5$ (troisième élément), $P(x, y) = 8.5$.

Comme il n'y a pas de limites quant au nombre d'ordres pour les tenseurs, on peut évaluer les points pour un polynôme avec un nombre quelconque de variables. Cependant, nous n'étudierons que la représentation de polynômes à une variable.

Nombre de points au lieu du pas

À la place de donner la taille d'un pas, on pourrait également appeler cette méthode en donnant comme argument un nombre de points à évaluer, ce qui est plus pratique lorsqu'on souhaite définir la précision du graphique. Dans ce cas, on convertira ce nombre de points en pas Δx, ce qui nous permettra d'utiliser la méthode pour l'évaluation expliquée précédemment.

Essayons de déduire une formule permettant cette conversion, à l'aide de cet exemple : admettons que nous voulons calculer $n = 5$ points sur l'intervalle $[a; b] = [0; 10]$. Une première idée évidente mais trompeuse vient à l'esprit : le pas serait égal à $\frac{10}{n} = 2$. Mais, on voit que les points à évaluer seront les suivants : `{0 ; 2 ; 4 ; 6 ; 8 ; 10}`, soit six points à la place de cinq ! Il faut alors diviser 10 non pas par n, mais $n-1 = 4$, ce qui nous donnerait un pas de 2.5. On aura donc les cinq préimages à évaluer : `{0 ; 2.5 ; 5 ; 7.5 ; 10}`.

Si l'intervalle était $[-4;6]$, le pas serait toujours de 2.5, on commencera simplement par -4 au lieu de 0. La formule de conversion est donc :

$$\Delta x = \frac{b-a}{n-1}$$

Notez que pour $n = 1$, on aura une division par zéro. Il faut donc gérer ce cas. De plus, il faut interdire les nombres de points nuls ou négatifs. Dans le code, si $n < 2$, sa valeur devient 2.

4.2.4 Champ de vision et transformation affine

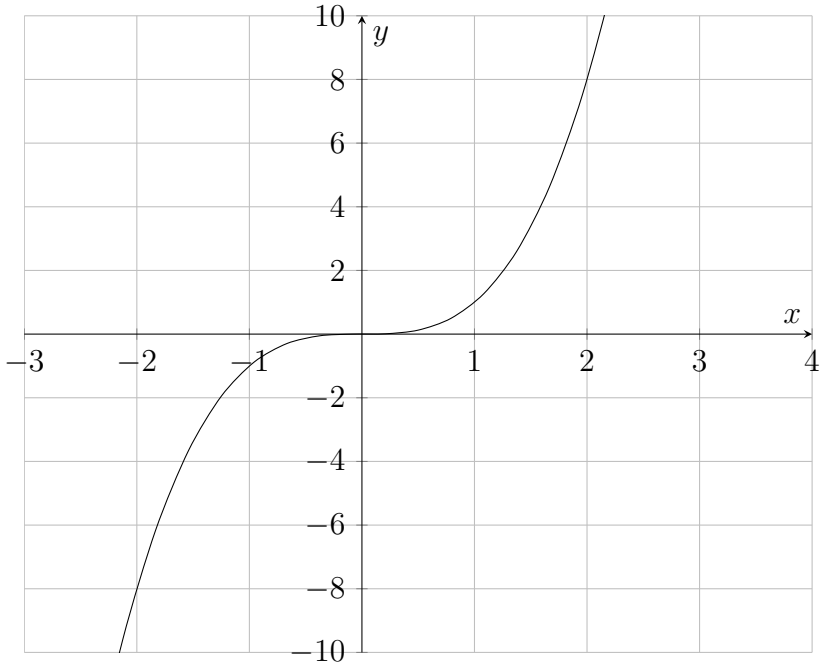
Classe ViewBox

Admettons que nous souhaitons représenter le polynôme $P(x) = x^3$ sur l'intervalle $[-5;5]$. Sachant que $(-5)^3 = -125$ et $5^3 = 125$, et que toutes les images pour $x \in [-5;5]$ sont comprises entre ces deux bornes, on comprend qu'il faut adapter les échelles : si, dans la représentation graphique, on graduait les deux axes avec la même échelle (repère orthonormé), on aurait un graphe 25 fois plus haut que large !

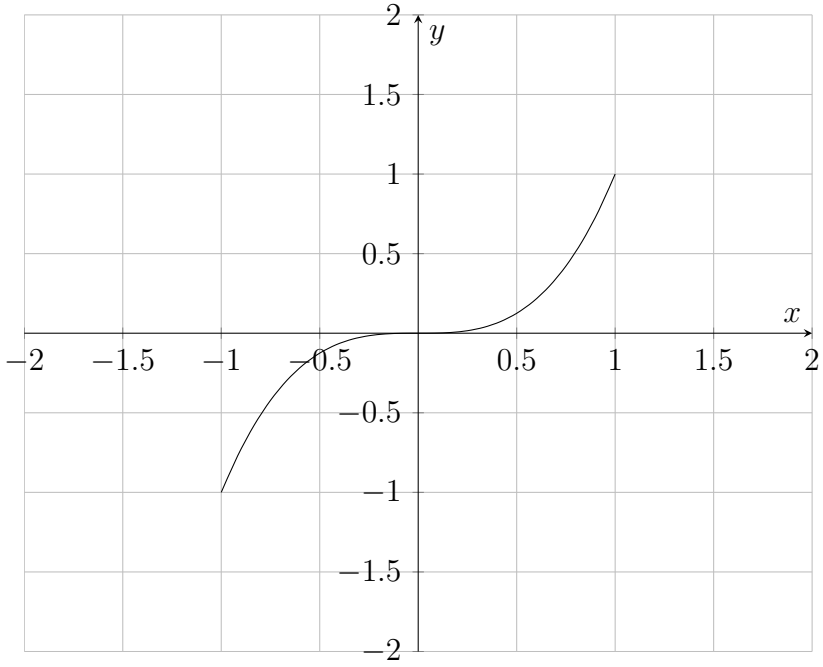
On contournera alors le problème soit en modifiant les échelles pour pouvoir tout voir, ici en « aplatissant » l'axe des ordonnées, soit en restreignant les intervalles des préimages et des images pour ne voir qu'une partie du graphique. Il serait alors utile de créer une classe appelée « ViewBox » (BoiteDeVue) stockant les intervalles couvrant ce qu'on souhaite voir. Comme on gère les polynômes à nombre quelconque de variables, cette classe stockera un tableau dynamique d'intervalles pour pouvoir restreindre toutes les variables.

```
1 class ViewBox {
2     private:
3     std::vector<Interval> _intervals; // Intervalles
4
5     public:
6     // ...
7 };
```

On ajoutera une `ViewBox` comme attribut pour `DataPlot`. Son intérêt est le suivant : supposons qu'on évalue les images de x^3 pour $x \in [-5;5]$, mais qu'on ne voudrait afficher sur un graphique que les points évalués en $x \in [-3;4]$. De plus, on restreint l'intervalle des images à $[-10;10]$. On aura donc un graphique comme celui-ci :



En « coulisses », toutes les valeurs pour $x \in [-5; 5]$ ont bien été calculées, mais on n'affiche que ce que vous voyez sur ce graphique, donc le point $(4; 64)$ est calculé, mais n'est pas visible, car il est trop haut. De la même manière, on pourrait n'évaluer que pour $x \in [-1; 1]$, mais afficher cela sur un graphique avec comme intervalles $x \in [-2; 2]$ pour les images et préimages :



On pourrait donc voir par exemple le point $(1.2; 1.728)$, mais il n'a pas été évalué, et ne figure donc pas sur le graphique. Voilà donc l'utilité de cette classe `ViewBox`.

Transformation affine

On sera amené à représenter sur l'écran le graphique. On pourrait se dire qu'avec tous les outils dont nous disposons, on peut y aller sans autre. Pas tout à fait : pour le moment,

nous ne savons que calculer des points de polynômes, mais pas les afficher. En pratique, on devra affichera les graphiques sur des rectangles de l'écran, par exemple une zone de 640×480 pixels. Pour pouvoir placer les points qu'on a calculés sur ce rectangle, il faut trouver un moyen de convertir les coordonnées « internes » en coordonnées « graphiques » : le point $(20; 8000)$ ne sera pas affiché au pixel de coordonnées $(20; 8000)$, par exemple.

Admettons un rectangle de longueur Δx et de hauteur Δy (ils importent peu), sur lequel on va tracer le graphique. La question est : si on choisit un point du polynôme $(x; y)$, quelles seront les coordonnées en pixels du point $(x'; y')$ qui sera affiché sur le rectangle ? Pour cela, on utilisera une transformation affine : elle permet de convertir les points du polynôme en pixels, à l'aide des deux formules suivantes :

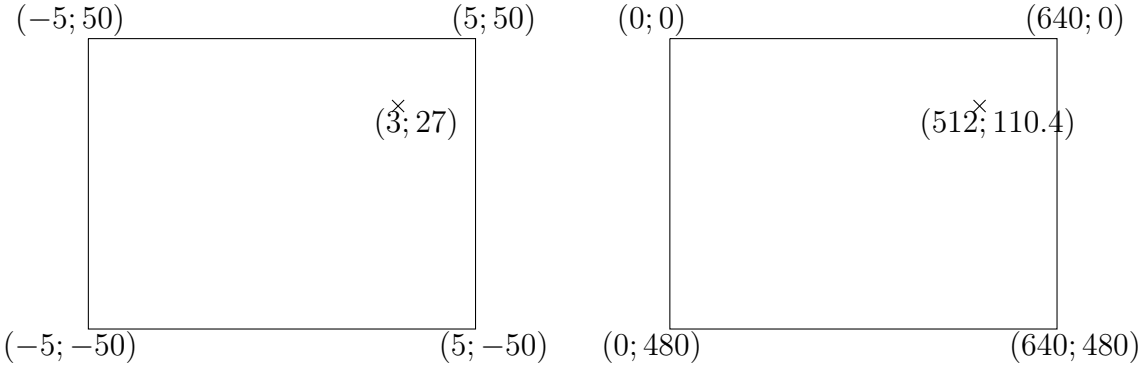
$$x' = \frac{(x - x_a)(x'_b - x'_a)}{x_b - x_a} + x'_a$$
$$y' = \frac{(y - y_a)(y'_b - y'_a)}{y_b - y_a} + y'_a$$

x_a et x_b sont les extrémités de l'intervalle des points x à évaluer, y_a et y_b les extrémités de l'intervalle des images, et les variables avec prime les homologhes en pixels.

Exemple concret : soit le polynôme x^3, dont on ne voit que les points dans les intervalles $x \in [-5; 5]$, et $y \in [-50; 50]$. On désire l'afficher dans un rectangle de 640×480 pixels, avec $x_a = 0$ et $y_b = 0$ (en haut à gauche). On admet aussi que $x_b = 640$ et $y_a = 480$. Par exemple, le point $(3; 27)$: on applique la formule à la lettre :

$$x' = \frac{(3 - (-5))(640 - 0)}{5 - (-5)} + 0 = 512$$
$$y' = \frac{(27 - (-50))(0 - 480)}{50 - (-50)} + 480 = 110.4$$

Ce point sera donc représenté au pixel de coordonnées $(512; 110.4)$:



En répétant le calcul pour tous les points algébriques calculés, on sera capable de représenter tous les points du graphique à l'écran.

5. L'interface graphique, produit final

5.1 Introduction, et choix de la bibliothèque

Un aspect important d'un programme est l'interaction entre la machine est l'utilisateur. En effet, comment utiliseriez-vous votre ordinateur, sans écran ? Il existe deux types d'interfaces permettant l'utilisation d'un ordinateur : le mode console, et le mode fenêtres. La console est une interface très rudimentaire, datant des années 1970 et ayant gardé cette apparence depuis toujours : un texte blanc, gris, ou vert sur un fond noir. Tandis que les fenêtres sont ce qu'on utilise le plus souvent aujourd'hui, sur les ordinateurs. C'est une couche graphique qu'on ajoute au-dessus de la console pour obtenir des programmes plus esthétiques. Pour ajouter cette couche, il faut utiliser une bibliothèque graphique.

Mais quelle bibliothèque utiliser ? Il en existe des dizaines, des centaines, voire plus. Certaines sont plus connues que d'autres. J'ai finalement choisi la bibliothèque `FLTK`[1], car elle est très légère, et présente l'avantage de pouvoir la compiler statiquement, c'est-à-dire l'intégrer directement dans l'exécutable, ce qui évite de devoir livrer des .dll lorsqu'on souhaite distribuer le programme, et les messages d'erreur du type : « Le programme n'a pas pu démarrer car il manque librairie.dll ». Elle ne propose pas des fonctions aussi avancées que beaucoup d'autres bibliothèques, mais elle me suffit, et je préfère utiliser la plupart des fonctions d'une petite bibliothèque, qu'une toute petite partie d'une bibliothèque très riche. Vous trouverez dans la bibliographie le site officiel de `FLTK`.

Cette partie sur l'interface graphique sera beaucoup moins technique, je me contenterai d'exposer des captures d'écran et des descriptions : je n'expliquerai pas comment on crée une zone de saisie ou un bouton. Vous pouvez consulter les documentations et tutoriels disponibles pour cela. Sachez juste pour commencer que les interfaces graphiques sont essentiellement un ensemble de composantes d'interfaces graphiques (aussi appelés widgets), qui interagissent entre elles. Par exemple, un bouton est un widget, qui, lorsqu'on appuie dessus, peut par exemple interagir avec une zone d'affichage (un autre widget) pour afficher un résultat, ou ouvrir une fenêtre... Après la conception d'`Algébra`, il est maintenant temps de commencer celle de `PolyCalc` ! Comme mentionné vers le début, `PolyCalc` est une calculatrice utilisant `Algébra`, et montre donc également des applications intéressantes de cette bibliothèque.

5.2 Présentation des fonctionnalités

Après m'être bien amusé à coder l'interface graphique de `PolyCalc`, permettez-moi de vous la présenter et de vous la décrire. Vous y trouverez plus de détails dans le manuel en annexe. Lorsque vous ouvrez le programme, vous avez ceci :

1. Abréviation de *Fast Light ToolKit* ; le site officiel indique que `FLTK` se prononce « *fulltick* ».

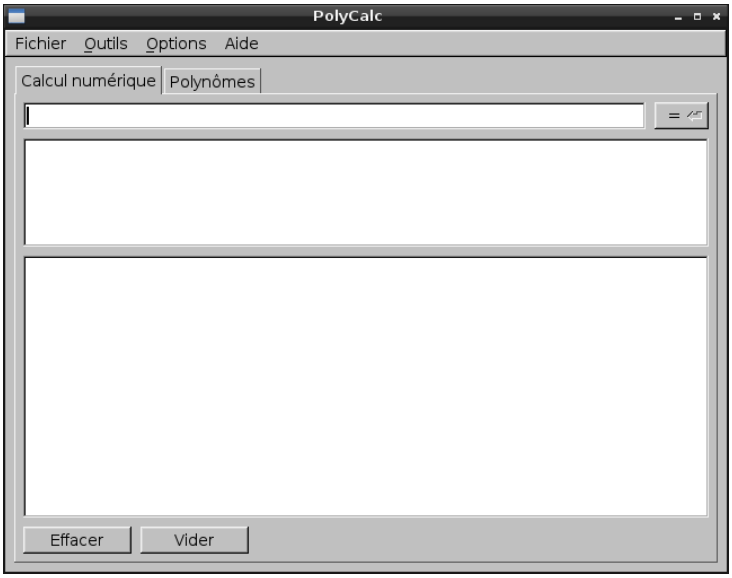


Fig. 5.1 – `PolyCalc` tel qu'il apparait à l'ouverture.

Vous y voyez un menu, des onglets, une zone de saisie (la boite blanche tout en haut), un bouton d'entrée à sa droite, et des boutons « Effacer » et « Vider ». Vous voilà face à la fonctionnalité de calcul numérique de `PolyCalc`. Le principe est le suivant : entrez une chaine d'opération numérique quelconque dans la première case. Appuyez sur le bouton à côté, ou sur la touche `Entrée` du clavier. La calculatrice vous fait le calcul, et donnera sa réponse dans la troisième zone :

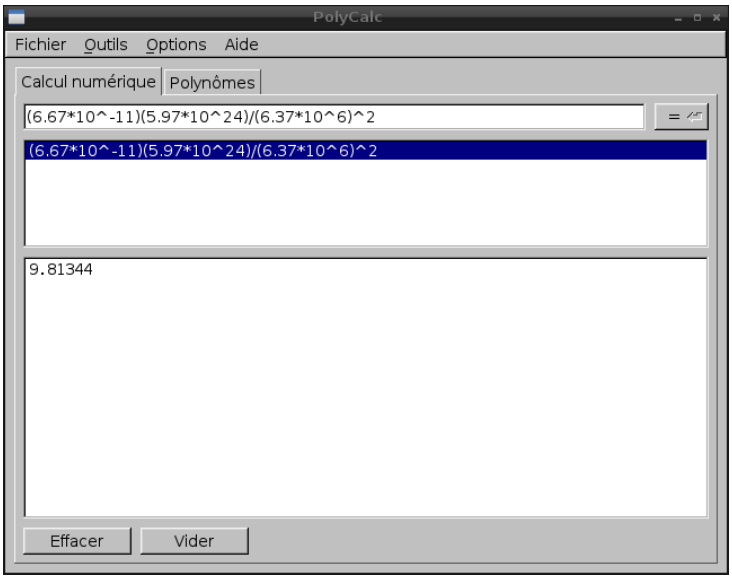


Fig. 5.2 – Calcul de physique tirant parti de la gestion des chaines d'opérations.

Il est possible de changer le nombre de décimales affichées, qui vaut par défaut 6, à 12, par exemple (le calcul donnera alors `9.813440652193`). On peut même mettre en valeur exacte la réponse : `3981990/405769`. On effectue cela en changeant les options dans le menu : `Options > Préférences`.

La deuxième zone permet de stocker un historique des chaines numériques entrées. Vous pouvez effacer celui qui est sélectionné à l'aide du bouton « Effacer », ou même tout vider avec le bouton à sa droite. Si une chaine invalide est entrée, par exemple s'il n'y a pas

autant de parenthèses ouvrantes que fermantes, la saisie n'est pas enregistrée, et une erreur s'affiche dans la zone de réponse.

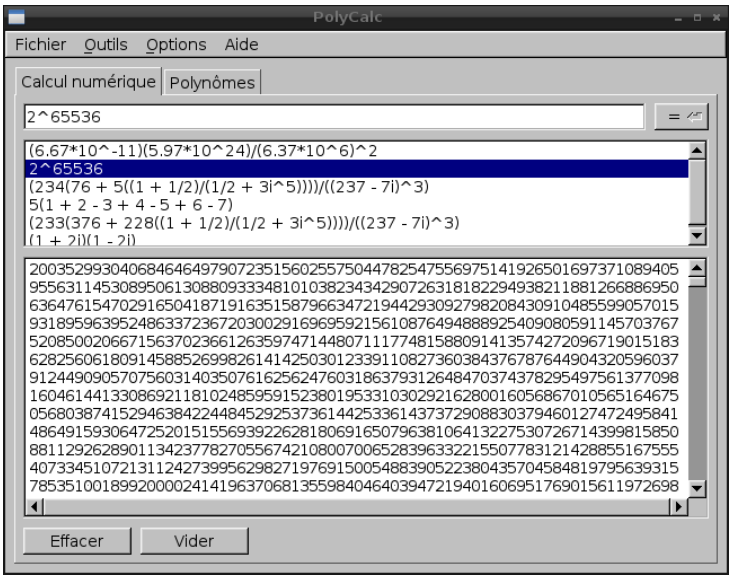


Fig. 5.3 – Plusieurs calculs plus tard... Le calcul sélectionné témoigne du support des nombres arbitrairement grands d'`Algébra`, 2^{65536} aurait fait un « Overflow Limit » dans la plupart des calculatrices.

Passons maintenant à la gestion des polynômes. L'interface est très similaire à celle des opérations numériques. Voici à quoi ça ressemble après avoir entré quelques polynômes :

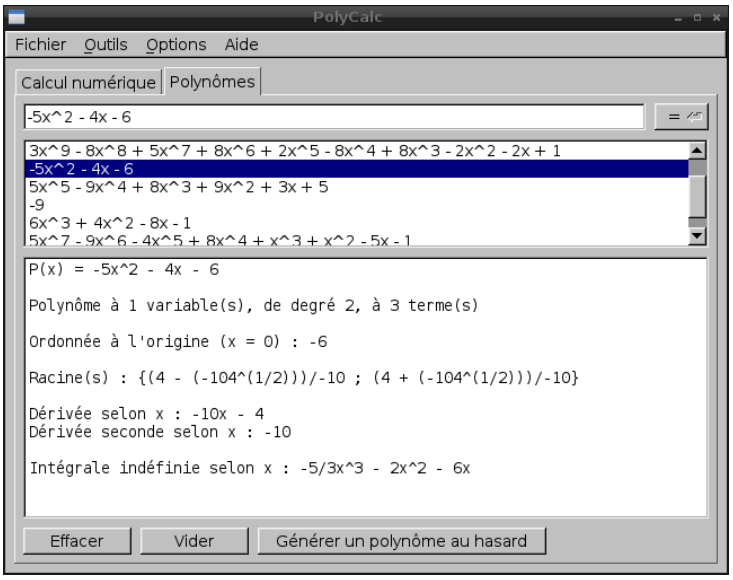


Fig. 5.4 – Interface des polynômes.

Dans la zone de résultat, on y voit quelques informations relatives au polynôme entré. Les informations sont les suivantes :

– Le polynôme entièrement développé. Il est possible d'entrer des polynômes factorisés ou partiellement factorisés : ceux-ci seront automatiquement développés (car on ne sait que stocker des polynômes développés), ce qui peut être utile si on a un exercice

qui demande de développer des polynômes comme $(x^2+4x+7)(x^2-x+1)$: `PolyCalc` vous permet de connaitre facilement la réponse : $x^4+3x^3+4x^2-3x+7$!

- Des informations générales (combien il y a de variables, le degré du polynôme, et combien il y a de termes)
- L'ordonnée à l'origine, donc l'image de zéro.
- Si le polynôme est de degré un ou deux, ses racines sont calculées, à l'aide des formules $ax+b=0 \Leftrightarrow x=-\frac{b}{a}$ et la formule de Viète. La classe `NumericExpression` permet de stocker des racines comme les deux que vous voyez sur l'image (qui sont $\frac{4\pm\sqrt{-104}}{10}$, donc des solutions complexes) !
- La dérivée du polynôme et la dérivée seconde. Cela peut être utile pour étudier la croissance et la concavité/convexité du polynôme.
- Une primitive du polynôme, celle avec un coefficient nul.

Il est également possible d'entrer des polynômes à plusieurs variables :

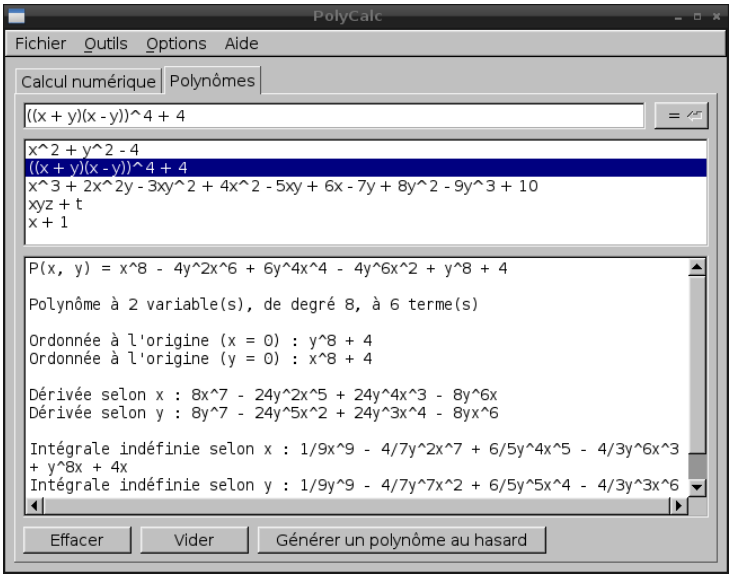


Fig. 5.5 – Les ordonnées à l'origine, dérivées et intégrales se font alors pour chaque variable.

Les polynômes à une variable peuvent être représentés graphiquement en allant dans le menu : `Outils > Graphique 2D`. Cela ouvre une fenêtre, et on obtient par exemple :

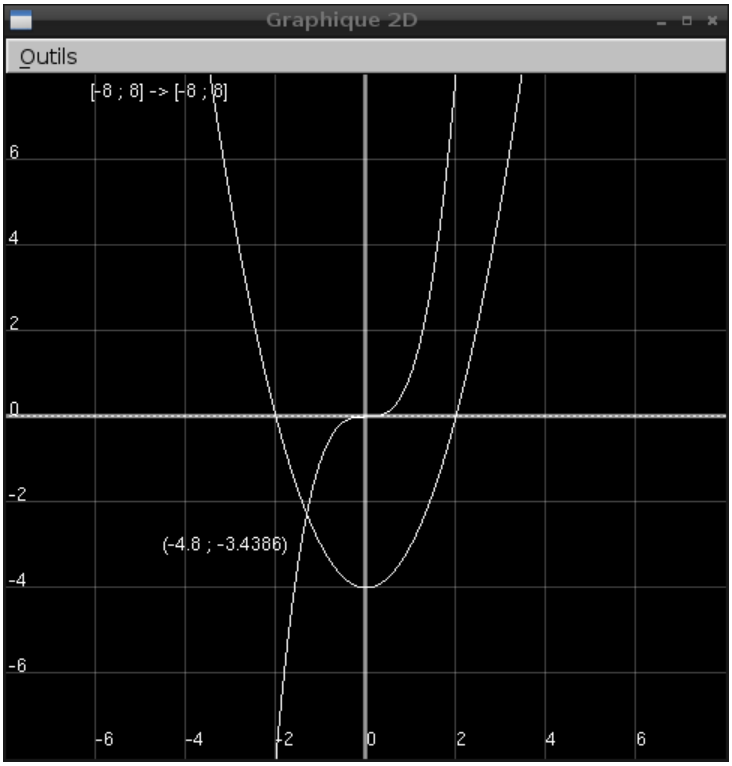


Fig. 5.6 – Représentation graphique de $x^2 - 4$ et x^3. Il est possible de zoomer avec la roulette, ou le clic droit, ou encore de déplacer le graphique.

Dans les outils, on propose de changer l'apparence de chaque graphe (couleur, opacité, épaisseur, nombre de points), et la zone de vision. Le polynôme dont on change l'apparence est celui qui est sélectionné dans la fenêtre principale. On permet également de choisir un repère orthonormé, en prenant en compte les dimensions de la fenêtre du graphique.

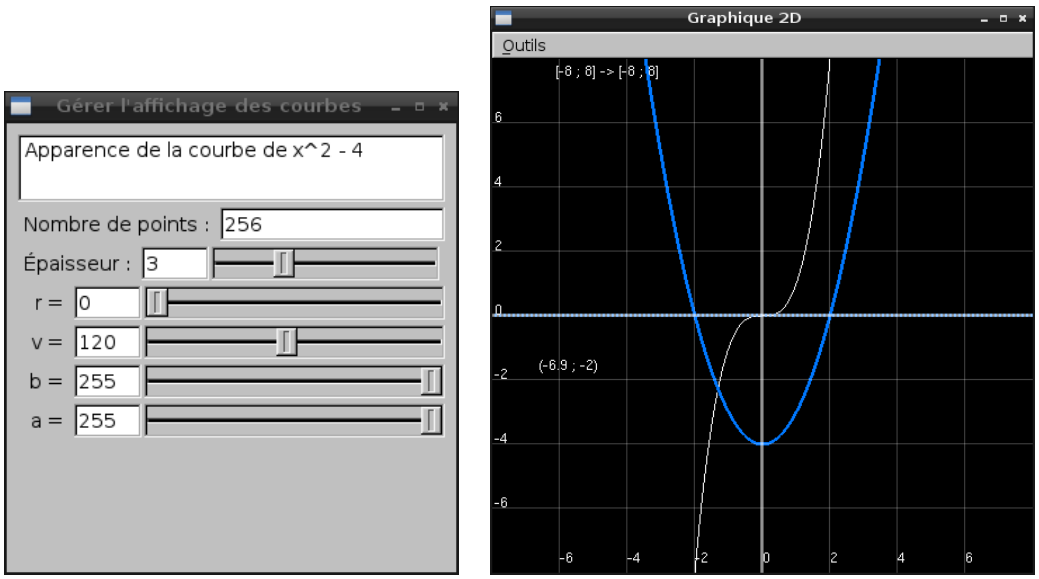


Fig. 5.7 – Changement de l'apparence du graphe et résultat.

Le bouton « Générer un polynôme au hasard » génère un polynôme aléatoire à une variable, de degré compris entre 0 et 8, et de coefficients compris entre -16 et 16. Les couleurs et l'épaisseur du graphe du polynôme sont également aléatoires. Son utilisation

sert uniquement à gribouiller le graphique de polynômes qui en voient de toutes les couleurs. J'ai ajouté cette fonctionnalité un peu inutile en pensant à une amie qui adore les arts, mais hait les mathématiques et l'informatique.

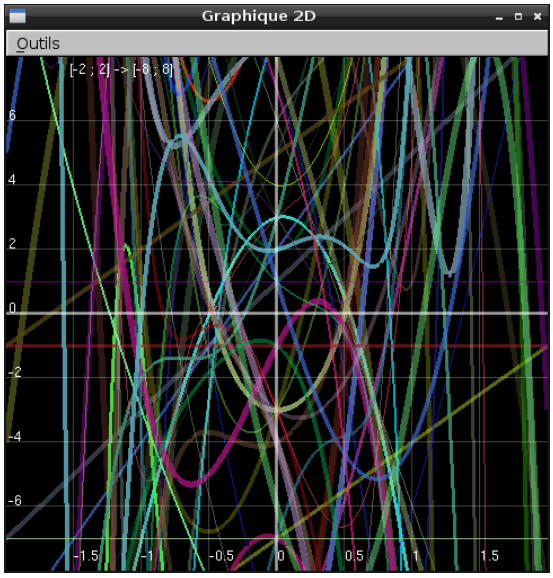


Fig. 5.8 – De l'art abstrait.

Et voilà ! On a fait le tour de ce produit fini... Toutes les fonctionnalités sont basées sur `Algébra`, et `PolyCalc` ne fait que fournir l'interface graphique, et donc un moyen d'utiliser ce qui est proposé par `Algébra`. Il y a encore d'autres fonctionnalités à présenter, comme l'approximation de racines d'un polynôme à l'aide de la méthode de Newton, ou bien la sauvegarde du graphique dans un fichier image. Vous pouvez lire le manuel de `PolyCalc` pour une présentation exhaustive des fonctionnalités.

6. Finitions, conclusion

Maintenant que la conception d'`Algébra` et de `PolyCalc` arrive au bout, il reste quelques dernières retouches à faire. Après, il sera temps de conclure ce travail de maturité.

6.1 Test du logiciel

Lorsqu'on programme, il est pratiquement impossible de ne pas faire d'erreurs dans le code. Cela entraine alors des *bogues*, qui sont des comportements non prévus du programme qu'on est en train de concevoir. Il est donc important de bien tester son programme avant de le proposer au grand public. Cette section vous propose de synthétiser les moyens que j'ai utilisés pour les tests.

6.1.1 Mini-tests

Un moyen de tester ses implémentations, est de les utiliser, en écrivant des petits programmes utilisant les fonctions et classes qu'on implémente, au fur et à mesure de leur écriture.

Ils consistent en l'effectuation d'opérations très simples, comme le fait d'entrer séparément deux nombres, et de les additionner entre eux. Cela permet donc dans ce cas de vérifier si l'addition se fait correctement, et si les deux nombres entrés sont bien reconnus. Beaucoup de fonctions élémentaires ont été testées de cette manière, et cela permet de faciliter le déboguage du programme. Tous ces tests se font sous la console.

Si par exemple, j'entre une chaine numérique contenant des sommes, différences, multiplications, et divisions, et qu'elle n'est pas calculée correctement, je peux regarder si l'addition se fait correctement. Si c'est le cas, je regarde pour la soustraction, puis la multiplication et la division : il se peut qu'une de ces fonctionnalités présente un bogue. Si, en faisant $6 - 2$, j'obtiens 5, j'en déduis que l'implémentation de la soustraction pose un problème. Je la corrige donc, et regarde si le problème initial a disparu. Sinon, je cherche où il pourrait y avoir un autre problème.

6.1.2 Bêta testeurs

Et en plus des tests manuels, un bon moyen de tester son programme est de distribuer le programme à son entourage afin qu'ils jouent avec, et me rapportent des bogues s'il y en a. Je remercie tous ceux qui y ont participé.

6.2 Rédaction de la documentation d'Algébra

Comme mentionné au début de ce dossier, `Algébra` est destiné à être une bibliothèque pouvant être utilisée par d'autres développeurs d'applications scientifiques. Or, comme toute

bibliothèque sérieuse qui se respecte, il faut fournir une documentation décrivant toutes les classes, fonctions, méthodes,... d'`Algébra`, afin de permettre au développeur de l'utiliser.

Il est tout à fait possible d'écrire à la main une telle documentation, et de le distribuer avec `Algébra`. Une autre solution consiste à écrire cette documentation à l'intérieur du code source, dans les commentaires. Cela a l'avantage de faire d'une pierre deux coups : on aura un code bien commenté, organisé, et maintenable, et on pourra générer la documentation à partir d'un outil qui lit ces commentaires. L'outil que j'ai utilisé pour cela s'appelle `Doxygen`. Je vous laisse consulter la bibliographie si vous souhaitez en savoir plus.

La documentation d'`Algébra` automatiquement générée est fournie en annexe. Elle n'est pas destinée à être imprimée, bien que rien ne vous empêche de le faire. Les descriptions ne sont pas détaillées à 100%, et il n'y a pas vraiment de tutoriel à proprement dit de cette bibliothèque ; l'objectif de ce travail de maturité étant avant tout la conception de la calculatrice.

6.3 Améliorations possibles

Tant de choses ont été effectuées jusque-là. Mais, le projet est loin d'être complet, et peut encore être bien amélioré ! Voici quelques propositions :

- Pour le moment, seuls les polynômes sont entièrement gérés. Si un collégien étudie les fonctions trigonométriques ou logarithmes par exemple, `Algébra/PolyCalc` ne sera pas d'une très grande utilité. Il s'agirait donc d'inclure le support de ces fonctions non polynomiales, ce qui pourrait se faire à l'aide des suites et séries pour calculer numériquement ces fonctions, et en adaptant la gestion des expressions algébriques pour permettre le stockage de ces fonctions.
- On pourrait gérer l'affichage de graphiques en 3D pour afficher les expressions à deux variables. Nous avons désormais tous les outils pour cela, mais par manque de temps, cette fonctionnalité n'a pas pu être implémentée.
- Des applications concrètes d'`Algébra`, autres qu'une calculatrice, auraient pu être écrites. Au début, il a été prévu de proposer une fonctionnalité permettant de calculer le troisième point d'intersection d'une droite et d'une courbe cubique (polynôme de degré 3 à deux variables), étant donné deux points d'intersection [1]. En lien avec ceci, il a été imaginé d'implémenter des fonctions de cryptographies basées sur ces courbes (cryptographie elliptique), qui tirerait également parti de la gestion des nombres arbitrairement grands. Nous n'avons pas eu le temps d'implémenter tout cela.
- Seule la base décimale est gérée. Pour intéresser certains théoriciens des nombres, il serait intéressant de proposer la gestion d'une base a quelconque. La gestion des bases binaire et hexadécimale pourrait également servir à des informaticiens.
- Initialement, il a été prévu de supporter les systèmes Android. Je me disais que ceci n'allait pas être très difficile, car je pourrais réutiliser la source d'`Algébra`, et juste écrire la source pour l'interface Android. En pratique, la programmation en `C`/`C++` est vraiment très peu aisée pour Android (qui est basé sur `Java`). D'après mes recherches, il faut utiliser un outil appelé « NDK (Native Development Kit) » proposé par Google. Or, je n'ai trouvé aucune ressource expliquant d'une manière compréhensible comment l'utiliser. Ce qui existe est vraiment mal expliqué et indigeste. Comme je ne suis pas prêt de réécrire tout le code en `Java`, je renonce au portage vers Android. Mais un jour, je reconsidèrerai la question, vu que la technologie devient de plus en plus mobile.
- Des algorithmes d'optimisation pourraient être utilisés pour améliorer le temps de

1. Sachant que d'après le théorème de Bézout, il y a toujours trois intersections dans cette situation.

calcul. Par exemple, il existe l'algorithme de Karatsuba qui permet d'effectuer beaucoup plus rapidement la multiplication que la méthode actuellement utilisée (inspirée de la multiplication en colonnes, également appelée multiplication naïve).
- Il serait bien de faciliter la traduction du programme en écrivant les textes de l'interface non pas directement dans le programme, mais dans des fichiers langue séparés.
- Bien sûr, malgré les tests effectués, de nombreux bogues attendent encore d'être découverts.
- Et il y a tant d'autres améliorations qui pourraient être imaginées...

Pour toutes ces raisons, le projet est encore en version bêta. Une première version « stable » pourrait voir le jour, lorsqu'on s'est assuré qu'il reste très peu de bogues, et après avoir implémenté certaines fonctions de cette liste. PdfTeX est un programme dont chaque nouvelle version révèle une décimale de π, par exemple la version 3.14159265 correspondrait à une version 8. Pour ma part, les numéros de version d'`Algébra` et `PolyCalc` seront 0.9, puis 0.99, 0.999, ..., plutôt que la classique incrémentation 1.0, 2.0, etc. Il n'y aura jamais de version 1.0 de mon logiciel, sauf s'il était possible de tendre le temps vers l'infini... La version en date de la fin de ce travail de maturité est $0.9\beta 3$.

6.4 Conclusion

On est cette fois vraiment arrivé au bout ! Si vous avez été capable de lire ce dossier du début à la fin, félicitations ! J'espère que vous avez eu du plaisir à me lire. Au cours de ce travail, nous avons pu implémenter des fonctions très intéressantes, comme la gestion des nombres arbitrairement grands et exacts, la gestion des polynômes, et leurs représentations graphiques. Le calcul d'opérations numériques a également été un gros morceau de ce travail. D'autres petites fonctionnalités comme la dérivation, l'intégration indéfinie et définie, et les tenseurs ont également pu être écrites.

Ce travail de maturité m'a également donné une excellente occasion d'appliquer les mathématiques qu'on apprend de l'école élémentaire à la fin du collège, sous la forme d'une calculatrice polyvalente. En outre, j'ai également pu approfondir mes connaissances en programmation, et les mettre en pratique : j'ai appris beaucoup de nouvelles choses, allant de notions avancées de la programmation orientée objet (notamment les patrons de classe, l'héritage, les collections hétérogènes), à l'utilisation de certaines techniques comme le calcul d'approximations (approximation des racines et méthode de Newton). Ce travail a donc été une expérience très enrichissante pour moi, et je ne regrette absolument pas le choix de mon sujet.

Il faut rappeler le titre de mon travail de maturité, qui est simplement « Élaboration d'une calculatrice virtuelle », depuis que j'ai soumis le sujet à la direction. Et on peut dire que l'objectif de concevoir cette calculatrice a été atteint, bien qu'elle pourrait encore, et encore être améliorée ! Au début, il a été prévu d'implémenter la plupart des fonctions mentionnées dans la section « Améliorations possibles », mais tout ceci a dû être annulé à cause du temps limité. Je suis évidemment un peu frustré de ne pas avoir pu implémenter des fonctions plus originales comme les cubiques ou la cryptographie, mais j'aurais tout le temps après ce travail de maturité, que je vois plus comme le début d'un long projet, que comme la conception d'un produit entièrement fini en un temps limité.

6.5 Remerciements

Tout d'abord, je remercie bien évidemment M. Cherix Pierre-Alain, mon enseignant d'application des mathématiques de la troisième année au collège Rousseau, et docteur en mathématiques, qui m'a suivi pour ce travail de maturité, et qui m'a fait confiance durant tout le long du travail, ce qui m'a permis d'être autonome et d'aller droit au but. Il a également lu, et relu mon dossier, pendant sa rédaction, ce qui m'a permis de le corriger ; j'espère que ce ne fut pas un travail trop laborieux pour lui.

Je remercie ensuite M. Bach Pierre (mon enseignant de physique en deuxième année, et également docteur en physique) pour m'avoir fait connaitre LaTeX, qui m'a permis de rédiger avec plaisir le présent document, qui m'a donné toutes sortes de conseils pour utiliser ce traitement de texte, qui a pu tester mon programme, et qui m'a donné quelques conseils pour la correction de ce document.

Je remercie également les docteurs Malaspinas Orestis et Latt Jonas, qui avaient supervisé mon extra-muros de deuxième année à l'université de Genève en informatique, pour avoir également testé mon programme et l'avoir commenté. Ils m'avaient également permis de m'introduire dans le monde Linux, et d'améliorer mes capacités à programmer durant ce stage d'extra-muros.

Et je remercie enfin tous les autres qui m'ont aidé, en jouant avec mon logiciel, en me faisant des retours, ou simplement en me souhaitant bonne chance. Et tout particulièrement Mélodie, qui a toujours été présente pour moi.

6.6 Bibligraphie

Si ce dossier avait été un travail de recherche, il aurait fallu citer un bon nombre de sources. Ici, dans le cadre d'une production « artistique », il n'y a pas particulièrement beaucoup de sources à citer. De plus, l'élaboration de ce programme n'utilisant que des mathématiques étudiées avant l'université (sauf peut-être les tenseurs), il est difficile de citer une source particulière, du fait que vous trouverez très facilement n'importe où des informations à ce sujet. Idem pour la programmation, où les informations peuvent être aisément trouvées. Permettez-moi toutefois de mentionner (toutes les pages internet ont été consultées une dernière fois le 25 octobre 2014) :

6.6.1 Programmation

- `OpenClassroom`[2] est un excellent site internet pour autodidactes proposant des centaines de tutoriels de qualité, dont un qui permet d'apprendre pas à pas le `C++`. L'auteur de ce tutoriel, et également de ce site créé en 1999, est Mathieu Nebra.
  - Accueil : `http://fr.openclassrooms.com/`
  - Tutoriel `C++` : `http://fr.openclassrooms.com/informatique/cours/programmez-avec-le-langage-c`
- `Développez.com` est un site qui propose de nombreuses ressources (guides, tutoriels,...) qui peuvent être très utiles. `http://www.developpez.com/`
- Un moteur de recherche est indispensable afin de trouver des solutions à des erreurs de programmation !
- `StackOverflow` est justement un site sur lequel je suis souvent tombé, lorsque je recherchais des solutions à des erreurs. Il s'agit d'un site communautaire où des

2. Anciennement le `Site du Zéro`.

programmeurs posent des questions en cas de problème, et sont répondus par d'autres programmeurs. Il a été créé par Jeff Atwood et Joël Spolsky en 2008. http://stackoverflow.com/
- Le site internet de FLTK. La première version date de 1998, et le principal développeur est Bill Spitzak. http://www.fltk.org/index.php
- Le site internet de Doxygen, qui sert à générer de la documentation en commentant le code. Le principal contributeur est Dimitri van Heesch, et la première version date de 1997, d'après la page Wikipédia anglaise sur Doxygen.
  - Site officiel avec documentation : http://www.doxygen.org/ (redirige vers http://www.stack.nl/~dimitri/doxygen/).
  - Développez.com propose un tutoriel utile pour commencer (l'auteur est Franck Hecht, le tutoriel a été publié le 19 septembre 2007) : http://franckh.developpez.com/tutoriels/outils/doxygen/
- Le site internet de CMake : http://www.cmake.org/
  - Développez.com propose aussi ici un tutoriel utile pour bien commencer (l'auteur est Florian Goujeon, le tutoriel a été publié le 6 janvier 2009) : http://florian-goujeon.developpez.com/cours/cmake/initiation/

6.6.2 Mathématiques

- Cours de mathématiques du niveau gymnasial : des enseignants du Collège de Genève mettent à disposition gratuitement sur le site suivant un manuel de mathématiques : http://disciplines.sismondi.ch/mathematiques/espace-perso-profs/serge-picchione.
- Wikipédia. Oui, je sais, certains vont me dire : « mais c'est pas une source, ça ! ». Mais pour être franc, cette célèbre encyclopédie en ligne m'a beaucoup aidé. Vous pouvez également consulter les références listées en bas de chaque page Wikipédia pour approfondir des sujets, ou avoir des sources plus formelles. Pages en français utiles :
  - Tenseurs : http://fr.wikipedia.org/wiki/Tenseur
  - Algorithme d'Euclide : http://fr.wikipedia.org/wiki/Algorithme_d%27Euclide
  - Algorithme pour l'approximation des racines n-ième : http://fr.wikipedia.org/wiki/Algorithme_de_calcul_de_la_racine_n-i%C3%A8me
  - Méthode de Newton : http://fr.wikipedia.org/wiki/M%C3%A9thode_de_Newton
- WolframAlpha : il s'agit d'un service en ligne permettant de résoudre divers calculs mathématiques. Je m'en suis beaucoup servi pour tester le programme, en vérifiant si les résultats donnés par ma calculatrice étaient corrects. http://www.wolframalpha.com/.
- Le livre *Complex Algebraic Curves* de Frances Kirwan, pour l'étude des cubiques. Année 1992, édité par *Cambridge University Press*, volume 23.

6.7 Récapitulatif des annexes

- Sources d'Algébra et de PolyCalc, et exécutables de PolyCalc fonctionnant sous Windows 32 bits, ainsi que Linux 32 et 64 bits. Les exécutables de ce qui a permis de tester Algébra sont également founies.
- Manuel d'instructions de PolyCalc (imprimé avec ce dossier).
- Documentation d'Algébra (pas destiné à être imprimé pour ce travail de maturité).