M1 Informatique C++
Rapport de Mini-Projet phase 2

François Gaits

7 décembre 2019

1 Réorganisation du code

1.1 La classe Function

Afin de pouvoir implémenter les nouvelles fonctionnalités demandées, j'ai décidé d'introduire une nouvelle classe *Function* pour implémenter le comportement de base d'une fonction.

```
class Function {
protected :
    std::vector<double> _args;
    std::string _baseFuncName;
    unsigned _fullArgsNumber;
    unsigned _nbArgs = 0;

public :
    virtual std::string repr()const;
    virtual bool isComplete() const;
    virtual double eval() const = 0;
    virtual std::unique_ptr<Function> addArgs(std::vector<
       double> newArgs) = 0;
    explicit Function(std::vector<double> args, unsigned
       fullA, std::string name);
    explicit Function(unsigned fullA, std::string name);
};
```

Figure 1 – Déclaration de la classe *Function*

Le vecteur *_args* contient les arguments de la fonction (ou NaN si placeholder), et *_nbArgs* le nombre d'arguments effectivement définis.

La méthode *addArgs* permet de définir de nouveaux arguments dans cette fonction et renvoyant un pointeur d'un **nouvel** objet contenant les arguments précédents, et les nouveaux (en remplaçant les placeholders, puis en augmentant le vecteur). Cette fonction procède à toutes les vérifications avant de créer le nouvel objet, les nouveaux arguments pouvant être nuls pour représenter à nouveau les placeholders. De cette façon utiliser un fonction partielle pour en définir une nouvelle n'écrase pas la définition de la précédente.

Dans mon implémentation, toutes les fonctions héritent de cette classe, et redéfinissent les méthodes nécessaires.

Les fonctions "de base" comme *log* ou *lerp* sont définies au lancement du programme comme des fonctions partielles sans arguments.

1.2 Cas des variables réelles

Pour rendre compte des variables réelles dans cette nouvelle organisation (résultat d'évaluation d'une expression algébrique, stockage de fonctions dont tous les arguments ont été définis) j'introduis une fonction *Identité (Id)*. De cette façon toute valeur fixe est stockée dans la même structure que les fonctions partielles.

De plus ma fonction *evaluateExpression* renvoie maintenant un pointeur de fonction et plus une valeur, ce qui permet de renvoyer (et donc de stocker) une fonction partielle.

2 Choix de conception

2.1 Syntaxe

Les fonctions partielles sont soumises à certaines règles de syntaxe :

Tout argument non exprimé est considéré comme étant un placeholder, il est même possible de ne nommer que la fonction sans parenthèse (*sin* ≡ *sin()* ≡ *sin(_1)*).

Les placeholders n'ont pas de limite de nombre (contrairement au sujet, _158 est théoriquement valide), ils doivent cependant être strictement dans l'ordre (*pow(_2,_1)* est invalide).

Il est impossible d'utiliser une fonction partielle dans une expression complexe (2+*pow(_1,2)* est invalide).

2.2 Cas particulier du polynôme

Pour un polynôme, tant que son degré n'est pas défini, il est considéré comme une fonction à un nombre infini d'arguments (autant que la mémoire le permet en tout cas), il n'est cependant pas possible de définir un degré correspondant à un nombre inférieur d'arguments que ceux déjà définis.

3 Remarques

3.1 Bibliothèque standard

Je connais l'existence des fonctions et structures *std ::placeholder* et *std ::bind* ou encore *std ::function*, mais j'ai décidé de ne pas les utiliser, cela me semblait plus complexe d'apprendre à m'en servir que d'utiliser mon propre système, d'autant plus que cela a été une bonne opportunité d'explorer plus en détails les *unique_ptr* et le mécanisme d'héritage.

3.2 Templates

J'ai conscience que mon code pourrait être grandement amélioré par l'utilisation de Templates pour les fonctions, mais ma connaissance de ces dernières ne me permettait pas de les utiliser dans le temps imparti.

3.3 Complexité des fonctions

Comme soulevé dans le rapport précédent, la fonction *evaluateExpression()* est très complexe et longue, chose que la reconnaissance de fonction partielles n'a fait qu'empirer.

3.4 Réorganisation des Tests

L'utilisation des tests reste la même que lors de la phase précédente, cependant le contenu des fichiers a changé :

appeler ./test.sh <num> exécutera le test correspondant au num :
1– tests basiques
2– utilisation de variables
3– fonctions
4– fonctions partielles simples
5– programme mettant en avant l'interaction entre fonctions partielles
6– fonction générant des erreurs

Appeler ./test.sh testera le programme sur l'ensemble des fichiers tests disponibles.