Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Filière informatique

Projet de semestre de printemps

2015

COJAC - Nombres enrichis

Rapport de projet

Auteur :
Sylvain Julmy

Superviseur :
Frédéric Bapst

Fribourg, le 13 mai 2015

Hes·so FRIBOURG FREIBURG
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Table des matières

1. Introduction

Le but de ce projet est d'offrir la possibilité d'utiliser des nombres enrichis en Java. En s'appuyant sur un travail de master réalisé il y a une année, on aimerait offrir la possibilité de vérifier si des programmes sont numériquement stables ou non. Une autre optique de ce projet est d'explorer le calcul de dérivée et d'offrir à n'importe quelle utilisateur la possibilité de calculer la dérivée de n'importe quelle fonction d'un programme Java sans que l'utilisateur n'ai à connaitre la dérivée exacte de la fonction.

1.1 Contexte du projet

Le projet de Master précédemment cité est parvenu à un résultat très intéressant : la possibilité de remplacer, à la volée, des types primitifs présents dans du code Java par des équivalents objets.

Voici un exemple permettant d'illustrer ce principe :

```
double a = 2.0;
double b = 3.0;

a = b + a;
```

→

```
BigDecimalDouble a = new BigDecimalDouble(2.0);
BigDecimalDouble b = new BigDecimalDouble(3.0);

a = BigDecimalDouble.dadd(a,b);
```

Cette percée permet désormais de trouver de nouvelles méthodes pour le diagnostic ou l'amélioration de divers programmes écrits en Java. Pour la suite, ce projet consiste à explorer et à intégrer, dans Cojac, diverses techniques permettant le diagnostic de problèmes de stabilité numérique ainsi que de trouver des utilisations concrètes de cet outil unique qu'est Cojac. Les différents thèmes à explorer sont les suivants :

- Différentiation automatique
- Calcul par intervalle
- Arithmétique stochastique discrète

1.2 Stabilité numérique

Actuellement, quand on parle d'un ordinateur, on voit cela comme étant une puissante machine permettant la résolution de calculs complexes pouvant prendre plusieurs jours. Par contre, on a

tendance à oublier qu'un ordinateur n'obéit pas à des lois mathématiques exactes. L'utilisation d'un nombre limité de bit incombe des limitations aux niveaux de la représentation mémoire des nombres.

Il n'est pas possible, avec un nombre limité de bit, de représenter exactement tous les nombres. Par exemple, le résultat du calcul 10/3 nous donne, en mathématique, $3.\overline{33}$. Tandis que sur machine, on obtient environ 3.33334. En effet, le nombre 3.33 ne peut pas être représenté en base deux, on a donc une légère erreur par rapport au résultat réel du programme.

La stabilité numérique d'un programme permet d'indiquer s'il est numériquement stable lors des opérations arithmétiques qu'il réalise, c'est-à-dire est-ce s'il souffre de problèmes d'arrondi ou non.

Pouvoir affirmer qu'un programme est numériquement stable a un avantage indéniable : il est possible de l'exécuter sur machine sans avoir trop de problèmes du point de vue numérique. Les résultats sont fiables.

1.3 Structure du document

Ce document est divisé en trois parties :

— Analyse

— Conception

— Implémentation et résultats obtenus

L'analyse explore les fondements théoriques des modèles et en ressort une base exploitable pour la suite, la conception permet de spécifier clairement quelles sont les méthodes à définir, les algorithmes à élaborer et les problèmes potentiels. Pour finir, l'implémentation revient sur la réalisation du projet et présente les résultats obtenus avec les trois modèles.

Première partie

Analyse

2. Arithmétique d'intervalles

L'arithmétique par intervalles (ou calcul par intervalles [20]) est une manière de représenter un nombre différemment de ce que nous avons l'habitude de voir. À la place d'utiliser une valeur par un unique nombre à virgule, on utilise un couple de valeurs qui représente l'intervalle dans lequel est compris ce nombre. Par exemple, pour exprimer $\pi = 3.141592654$ on pourrait utiliser l'intervalle $\pi = [3.14; 3.15]$ comme montré à la figure 2.1. L'intérêt de cette technique c'est que l'on est assuré que le résultat se trouve toujours dans l'intervalle, bien que l'on ne connaisse pas sa valeur exacte.

Cette partie va présenter le concept d'arithmétique par intervalle ainsi que deux alternatives majeures à celui-ci, définir quelques concepts théoriques, montrer la redéfinition des opérations usuelles sur des nombres, exposer quelques concepts plus complexes comme les fonctions trigonométriques sur intervalles, parcourir les implémentations déjà existantes en Java et conclure par une présentation des avantages et inconvénients de cette représentation.

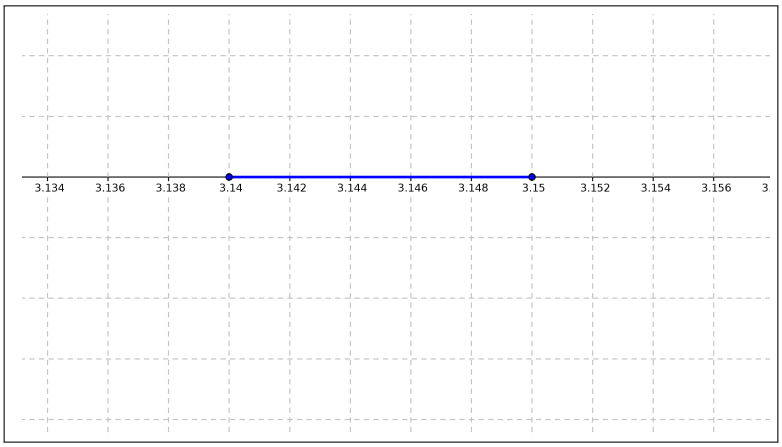

FIGURE 2.1 – On peut voir l'intervalle comme un ensemble de nombres présents sur la droite des nombres.

2.1 Principaux concurrents

L'arithmétique d'intervalles n'est pas le seul modèle de calcul alternatif que l'on puisse trouver, il existe également l'arithmétique multiprécision ainsi que l'arithmétique symbolique (ou formelle), qui possèdent également bon nombre d'atouts mais aussi des inconvénients, tout comme l'arithmétique d'intervalles.

2.1.1 Arithmétique multiprécision

Il s'agit également d'une arithmétique basée sur les nombres flottants, sauf que, contrairement aux nombres représentés sur la machine, on laisse le choix de la précision du nombre. L'avantage le plus flagrant, c'est que le nombre de chiffres représentés est beaucoup plus grand, ce qui conduit à une plus grande précision. Le principal défaut vient du fait que ce n'est plus le processeur qui manipule les nombres et donc que la complexité des opérations est de l'ordre de la longueur n du nombre (par longueur, on entend ici le nombre de chiffres utilisés pour représenter le nombre, par exemple le nombre 9264.54872 est représenté par 9 chiffres) : $O(n)$. L'autre défaut, c'est le même que l'arithmétique standard, celui de la précision. On ne peut pas représenter tous les nombres réels en utilisant une représentation en base deux. Mais cela est bien meilleur qu'en utilisant l'arithmétique flottante standard.

2.1.2 Arithmétique symbolique

L'autre célèbre approche est l'arithmétique symbolique. Ici on cherche à représenter le nombre exact. Pour ce faire on peut utiliser une représentation rationnelle (tout nombre peut être représenté sous la forme d'une fraction (sauf les nombres irrationnels comme π)) ou symbolique (on utilise des symboles ainsi que des règles algébrique pour résoudre des calculs, on obtient donc une représentation symbolique du résultat, par exemple $\frac{\pi*9}{3} = \pi * 3$. Par contre, on ne pourra pas calculer la valeur exacte de $\pi * 3$ avec l'arithmétique symbolique). L'avantage est que l'on obtient une précision sans égale, le problème vient du fait que l'arithmétique symbolique ne peut pas nous donner de valeur réel pour un calcul, on aura des résultats sous formes d'expression (par exemple $\frac{\sqrt{5}}{3}$). Les célèbres logiciels Mathematica ou Maple utilisent cette représentation.

2.2 Quelques bases théoriques

Il est possible de représenter un intervalle de diverses manières :

- en utilisant un segment sur la droite des nombres, où le point de départ et le point d'arrivée sont les bornes de l'intervalle (ou, par abus de langage, comme un vecteur, avec un point d'origine et une longueur).
- en utilisant un couple de nombres entiers, réels, complexes, etc...
- en utilisant une valeur centrale (le centre de l'intervalle) et un rayon.

Dans le cadre de ce projet, on ne verra et n'utilisera que la représentation par un couple de nombres à virgule flottante.

2.2.1 Représentation formelle de l'intervalle

Un intervalle est représenté de la manière suivante : $\mathbf{x} = [\underline{x}; \overline{x}]$ où $\underline{x}$ est la borne inférieure de l'intervalle et $\overline{x}$ la borne supérieure. À partir de cela, on peut définir la relation suivante [17] :

$$\mathbf{x} \in \mathbb{R} \ , \ \underline{x} \leq \mathbf{x} \leq \overline{x}$$

il faut bien se rendre compte que les bornes inférieure et supérieure de l'intervalle sont représentées par des nombres à virgule flottante et que le résultat exact du calcul se trouve entre ces deux bornes.

2.2.2 Propriétés supplémentaires

On note $w(x)$ la **largeur** de l'intervalle qui est définei comme $w(x) = \overline{x} - \underline{x}$. On rencontre parfois $mid(x)$ ou $\breve{x}$ qui représente le milieu de l'intervalle $\mathbf{x}$ et qui est défini comme $mid(\mathbf{x}) = \frac{\underline{x}+\overline{x}}{2}$[17]. Ces quelques définitions seront utilisées dans la suite de ce chapitre.

On peut aussi voir un intervalle comme un ensemble, l'intervalle $\mathbf{x}$ contenant tous les nombres compris entre $\underline{x}$ et $\overline{x}$.

2.3 Opérations de comparaison

Il est également possible de redéfinir les opérations de comparaison pour les intervalles [19] :

$$\text{Certainement positif ou nul} \qquad ([\underline{x};\overline{x}] \geq 0) := (\underline{x} \geq 0) \tag{2.1}$$

$$\text{Certainement strictement positif} \qquad ([\underline{x};\overline{x}] > 0) := (\underline{x} > 0) \tag{2.2}$$

$$\text{Certainement négatif ou nul} \qquad ([\underline{x};\overline{x}] \leq 0) := (\overline{x} \leq 0) \tag{2.3}$$

$$\text{Certainement strictement négatif} \qquad ([\underline{x};\overline{x}] < 0) := (\overline{x} < 0) \tag{2.4}$$

$$\text{Possiblement nul} \qquad (0 \in [\underline{x};\overline{x}]) := (\underline{x} \leq 0 \wedge \overline{x} \geq 0) \tag{2.5}$$

On peut également comparer deux intervalles. Il suffit juste de remplacer la valeur 0 par un intervalle et modifier les opérandes de comparaison :

$$\text{Comparaison entre intervalles : } [\underline{x};\overline{x}] <=> [\underline{y};\overline{y}] \begin{cases} [\underline{x};\overline{x}] & > & [\underline{y};\overline{y}], \text{ si } \underline{x} > \overline{y} \\ [\underline{x};\overline{x}] & < & [\underline{y};\overline{y}], \text{ si } \overline{x} > \underline{y} \\ [\underline{x};\overline{x}] & == & [\underline{y};\overline{y}], \text{ sinon} \end{cases} \tag{2.6}$$

$$\tag{2.7}$$

2.4 Redéfinition des opérations de base

Il est possible d'étendre les opérations arithmétiques usuelles aux intervalles grâce à la formule suivante : si $\mathbf{x}$ et $\mathbf{y}$ sont deux intervalles et $\blacklozenge \in +, -, /, *$ alors (lors de la division, $\mathbf{y}$ ne doit pas contenir 0) [17] [9]

$$\mathbf{x} \blacklozenge \mathbf{y} = \{x \blacklozenge y \ / \ x \in \mathbf{x}, y \in \mathbf{y}\}$$

Voici l'ensemble des opérations redéfinies pour les intervalles [9] :

$$-[\underline{x};\overline{x}] = [-\overline{x};-\underline{x}] \tag{2.8}$$

$$[\underline{x};\overline{x}] + [\underline{y};\overline{y}] = [\underline{x}+\underline{y};\overline{x}+\overline{y}] \tag{2.9}$$

$$[\underline{x};\overline{x}] - [\underline{y};\overline{y}] = [\underline{x}-\overline{y};\overline{x}-\underline{y}] \tag{2.10}$$

$$[\underline{x};\overline{x}] \times [\underline{y};\overline{y}] = [min(\underline{x}\times\underline{y},\overline{x}\times\underline{y},\underline{x}\times\overline{y},\overline{x}\times\overline{y});max(\underline{x}\times\underline{y},\overline{x}\times\underline{y},\underline{x}\times\overline{y},\overline{x}\times\overline{y})] \tag{2.11}$$

$$[\underline{x};\overline{x}] \div [\underline{y};\overline{y}] = [min(\underline{x}\div\underline{y},\overline{x}\div\underline{y},\underline{x}\div\overline{y},\overline{x}\div\overline{y});max(\underline{x}\div\underline{y},\overline{x}\div\underline{y},\underline{x}\div\overline{y},\overline{x}\div\overline{y})]; \text{si } 0 \notin \mathbf{y} \tag{2.12}$$

$$[\underline{x};\overline{x}]^2 = \begin{cases} [\overline{x}^2;\underline{x}^2] & , \text{ si } \forall x \in \mathbf{x}, x \leq 0 \\ [0;max(\underline{x}^2,\overline{x}^2)] & , \text{ si } x \in 0 \\ [\underline{x}^2;\overline{x}^2] & , \text{ si } \forall x \in \mathbf{x}, x \geq 0 \end{cases} \tag{2.13}$$

$$|[\underline{x};\overline{x}]| = \begin{cases} [\underline{x};\overline{x}] & , \text{ si } \forall x \in \mathbf{x}, x \geq 0 \\ [-\overline{x};-\underline{x}] & , \text{ si } \forall x \in \mathbf{x}, x \leq 0 \\ [0;max(-\underline{x},\overline{x})] & , \text{ si } 0 \in \mathbf{x} \end{cases} \tag{2.14}$$

$$[\underline{x};\overline{x}]^n = \begin{cases} [\underline{x}^n;\overline{x}^n] & , \text{ si } \underline{x} \geq 0 \\ [\overline{x}^n;\underline{x}^n] & , \text{ si } \overline{x} < 0 \\ [0;max(\underline{x}^n,\overline{x}^n))] & , \text{ sinon} \end{cases} \tag{2.15}$$

$$\sqrt{[\underline{y};\overline{y}]} = [\sqrt{\underline{x}};\sqrt{\overline{x}}] \ ; \text{ si } \forall x \in \mathbf{x}, x \geq 0 \tag{2.16}$$

$$\ln([\underline{x};\overline{x}]) = [\ln(\underline{x});\ln(\overline{x})] \ ; \text{ si } \forall x \in \mathbf{x}, x > 0 \tag{2.17}$$

$$\exp([\underline{x};\overline{x}]) = [\exp(\underline{x});\exp(\overline{x})] \tag{2.18}$$

Il faut bien comprendre que lorsque le résultat est le fruit de calcul, on a un arrondi forcé vers le bas pour la borne inférieure et un arrondi forcé vers le haut pour la borne supérieure. Ainsi, on est assuré que le résultat est contenu dans l'intervalle.

On notera également que pour chaque opération ci-dessus, la largeur du résultat est toujours plus grande que la largeur des opérandes :

$$\begin{aligned} &\mathbf{x} = [1;4] \\ &\mathbf{y} = [-3;-5] \\ &\mathbf{x} * \mathbf{y} = [1;4] * [-3;-5] = [-20;-3] \\ &w(\mathbf{x}) = 3 \\ &w(\mathbf{y}) = 2 \\ &w([-20;-3]) = 17 \end{aligned}$$

On peut également voir l'exemple à la figure 2.2 que a et b sont plus petits que le résultat de $a + b$. Cela a une grande influence sur les résultats obtenus à l'issue de chaque opération arithmétique, vu que plus l'intervalle est large, moins le résultat est fiable.

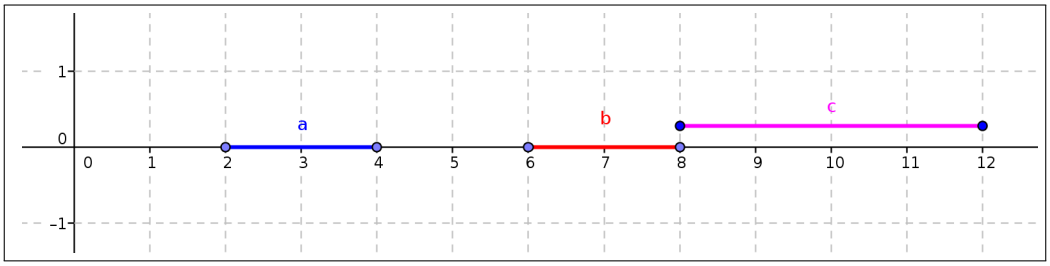

FIGURE 2.2 – On peut voir les intervalles comme des segments sur la droite des nombres (a et b) et c comme le résultat de $a + b = c$

2.5 Mise à la puissance

La fonction de mise à la puissance est définie pour les intervalles (pour autant qu'ils soient supérieurs à 0). Dans le cadre de COJAC, on remplace les types primitifs *double* par un type objet (dans le cas des intervalles : *IntervalDouble*) et lorsque l'on utilise l'opération de puissance (Math.pow) on court-circuite l'appel vers la fonction puissance définie pour l'objet *IntervalDouble*. Cela veut dire qu'il faut être capable de définir les opérations de puissance entre deux intervalles. Comme la fonction de puissance est monotone, il est possible de la définir, mais uniquement si la base de la puissance ne contient pas de nombres négatifs (ce cas s'applique aussi à la racine carrée par ailleurs). Il existe alors trois manières de calculer le résultat de $[\underline{x}; \overline{x}]^{[\underline{y};\overline{y}]}$:

— Calculer la valeur maximale et minimale qui peut être obtenue mais cela implique de traiter les cas spéciaux (ne pas autoriser un intervalle contenant des nombres négatifs) :

$$[\underline{x}; \overline{x}]^{[\underline{y};\overline{y}]} = [min(\underline{x}^{\underline{y}}, \overline{x}^{\underline{y}}, \underline{x}^{\overline{y}}, \overline{x}^{\overline{y}}); max(\underline{x}^{\underline{y}}, \overline{x}^{\underline{y}}, \underline{x}^{\overline{y}}, \overline{x}^{\overline{y}})]$$

— Utiliser la relation entre la puissance et le logarithme/exponentielle (qui sont également définis pour les intervalles) :

$$x^y = e^{\ln x^y} \rightarrow [\underline{x}; \overline{x}]^{[\underline{y};\overline{y}]} = exp([\underline{y}; \overline{y}] * \ln([\underline{x}; \overline{x}]))$$

— Une autre solution serait de "couper" l'intervalle représentant la base jusqu'à 0. Par exemple, l'intervalle $[-5; 3]$ deviendrait $[0; 3]$ et on appliquerait la fonction sur ce nouvel intervalle, cela éviterait de traiter des cas embêtants.

L'ensemble de ces solutions suppose que soit l'intervalle représentant la base soit entièrement positif (pour la deuxième solution) soit que si la base contient des nombres négatifs, alors la puissance doit être positive et entière. La troisième solution ne souffre pas de ces deux problèmes, par contre on perd une partie de l'information contenue par l'intervalle. Pour rappel, la base ne doit pas contenir de nombres négatifs, car si on élève un nombre négatif à une puissance non entière, des racines apparaissent :

$$-3^{\frac{7}{4}} = \sqrt[4]{-3^7}$$

Or les racines paires des nombres négatifs sont complexes, on peut le voir à la figure 2.3 que la fonction n'est pas définie pour des nombres plus petits que 0. Pour chacune des solutions proposées, il faudrait garder une trace du fait que l'intervalle pourrait contenir une partie complexe (via un attribut propre à la classe de l'intervalle par exemple).

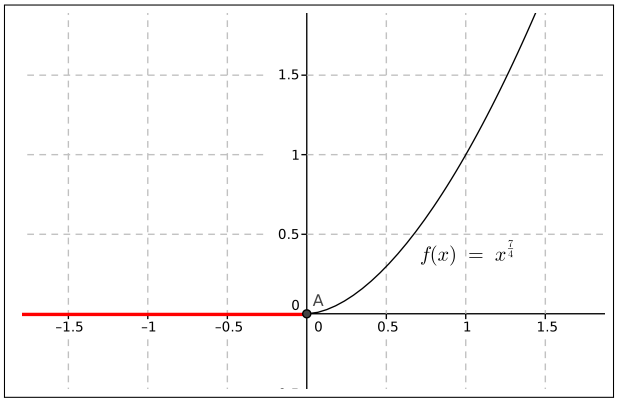

FIGURE 2.3 – Certaines fonctions puissances (ici $f(x) = x^{\frac{7}{4}}$) ne sont pas définies pour des nombres plus petits que 0

2.6 Fonctions trigonométriques

Les fonctions trigonométriques, telles que le sinus ou le cosinus sont également définies pour les intervalles. On s'appuie sur la monotonie par morceaux de ces fonctions (ou monotonie partielle) pour définir les résultats de ces opérations. Typiquement, si on possède un intervalle très large (plus large ou égal à $2*\pi$) le résultat de l'opération sinus sur cet intervalle sera : $\sin(\mathbf{x}) = [-1; 1]$.

Afin de calculer le résultat de l'opération sinus sur un intervalle, deux solutions sont envisageables :

— dans un premier temps, ramener l'intervalle entre -2π et 2π, trouver les minima et maxima globaux de la fonction à évaluer (il y en a 4) et, pour finir, calculer la valeur minimale et la valeur maximale se trouvant entre ces 4 valeurs (la fonction est monotone à l'intérieur de ces 4 valeurs) comme représenté à la figure 2.4.

— utiliser un développement de Taylor de la fonction pour trouver la borne inférieure et la borne supérieure du résultat[5] :

$$\underline{\sin}(x, n) = \sum_{i=1}^{m} (-1)^{i-1} \frac{x^{2i-1}}{(2i-1)!} \tag{2.19}$$

$$\overline{\sin}(x, n) = \sum_{i=1}^{m+1} (-1)^{i-1} \frac{x^{2i-1}}{(2i-1)!} \tag{2.20}$$

$$\underline{\cos}(x, n) = 1 + \sum_{i=1}^{m+1} (-1)^{i} \frac{x^{2i}}{(2i)!} \tag{2.21}$$

$$\overline{\cos}(x, n) = 1 + \sum_{i=1}^{m} (-1)^{i} \frac{x^{2i}}{(2i)!} \tag{2.22}$$

Où $m = 2n$ si $x < 0$ et $m = 2n + 1$ sinon (n représente le paramètre d'approximation, c'est la précision voulue du résultat. Plus on fait d'itérations, plus le résultat est précis).

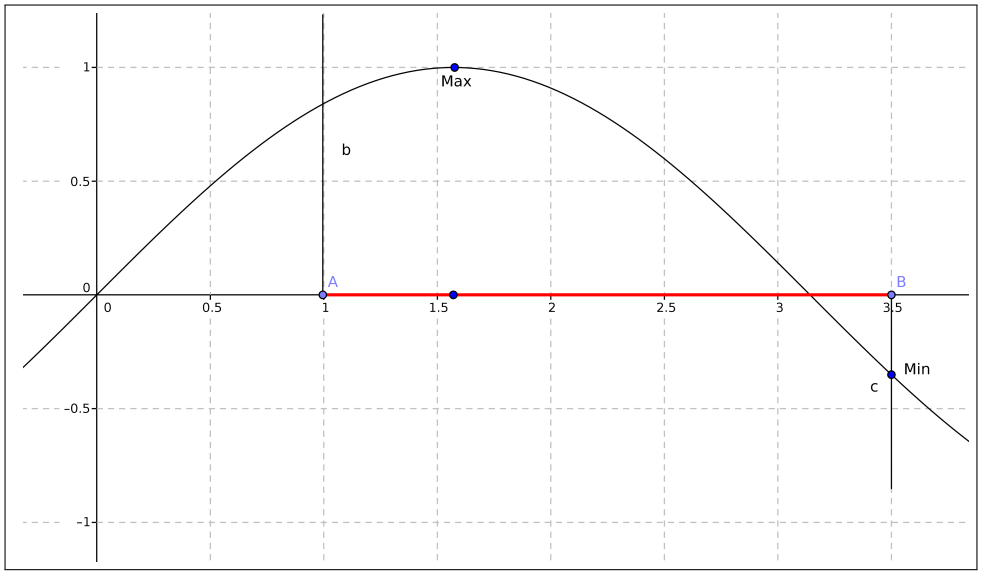

FIGURE 2.4 – Le résultat du sinus sur l'intervalle $[1; 3.5]$ est l'intervalle $[min; max]$, le sinus est monotone de 1 à $\frac{\pi}{2}$ et de $\frac{\pi}{2}$ à 3.5

Les fonctions trigonométriques suivantes :

- acos
- asin
- atan
- sinh
- tanh
- cosh (partiellement)
- coth (partiellement)

sont monotones.

2.7 Propriétés algébriques

Dans le monde de l'arithmétique d'intervalles, les propriétés algébriques des nombres changent. Par exemple, la soustraction n'est plus la réciproque de l'addition :

$$[4; 8] + [3; 9] = [7; 17] \qquad [7; 17] - [3; 9] = [4; 14] \neq [4; 8]$$

Par contre, le résultat inclus l'intervalle $[4; 8]$, cela donne juste un intervalle plus grand. En pratique, il faut faire attention lorsqu'on utilise des intervalles dans les algorithmes : différentes suites d'opérations peuvent être algébriquement correctes mais ne le sont plus avec des intervalles car cela les agrandis inutilement. Par contre, on est assuré d'avoir un intervalle qui contient la réponse.

Si l'on reprend l'équation de la comparaison entre intervalles (2.3), on a une perte de propriété assez désagréable, si on prend les intervalles suivants : $a = [3; 5]$ $b = [4; 6]$ $c = [5.5; 7]$ et bien ont aura $a = b$, $b = c$ mais $a \neq c$. On a une perte de la classe d'équivalence.

2.8 Arithmétique d'intervalles en Java

La librairie standard Java ne fournit, pour le moment, pas de bibliothèque concernant l'arithmétique d'intervalles. Il existe par contre, plusieurs bibliothèques externes, pour Java entre autres, qui fournissent ce mécanisme :

— IA_math [7]

— Java-XSC [2]

— JInterval [6]

Par contre, ces librairies sont soit incomplètes, soit inutilisables pour ce projet (JInterval) soit le développement est complètement abandonné et aucune documentation n'existe. La décision a donc été prise de créer une classe qui fournisse cette représentation d'intervalles. Cela sera néanmoins une implémentation naïve. Dans un premier temps, on utilisera cette solution et si un jour, une alternative viable se présente, on modifiera l'implémentation pour utiliser cette alternative. Java étant mal loti de ce côté, il existe des librairies dans d'autres langages, notamment pour C/C++ : boost [4], qui fournit, entre autres, des objets pour manipuler des intervalles.

2.9 Avantages et inconvénients

Par rapport aux deux autres alternatives présentées en début de chapitre (arithmétique multi-précision et symbolique), la représentation par intervalle d'un nombre utilise des types standards *double*. Cela implique que les opérations usuelles (multiplication, addition,...), qui possèdent une complexité algorithmique en $O(n)$ où n est la longueur du nombre pour les deux autres représentations, sont en $O(1)$. L'autre avantage majeur d'utiliser l'arithmétique d'intervalles, c'est sa robustesse. Quel que soit le calcul effectué, on est assuré que le résultat exact sera toujours compris dans l'intervalle à la fin des calculs [17].

Le défaut majeur et évident de cette représentation est sont pessimisme, en forçant à chaque fois l'arrondi, on détériore volontairement l'intervalle pour s'assurer que le résultat réel est toujours entre les deux bornes. Si l'intervalle obtenu est trop large pour le problème que l'on veut résoudre, alors le résultat est inexploitable. Par exemple, si on cherche à calculer la résistance thermique d'un matériel et que le point de rupture se trouve dans l'intervalle, cela devient risqué de se fier à ce résultat. D'un autre côté, on est assuré d'avoir le résultat exact dans l'intervalle. Il suffit après de déterminer l'erreur relative de l'intervalle ainsi qu'une valeur limite pour déterminer si le calcul est stable ou non.

3. Arithmétique stochastique discrète

L'arithmétique stochastique discrète est une technique utilisée en informatique qui permet d'estimer l'erreur relative d'un nombre ayant subi des arrondis. L'idée principale est de maintenir, en parallèle, trois nombres à virgules flottante qui représente le nombre.Par la suite, lors de chaque opérations arithmétiques, les valeurs maintenues en parallèle vont subir des opérations de calculs et vont être arrondies aléatoirement. C'est de la que vient le mot stochastique, les nombres sont sont aléatoires mais très proche les uns des autres.

Ce chapitre va introduire le concept d'erreur d'arrondi dans un programme informatique, expliquer les différents modes d'arrondi définis sur les nombres flottants selon l'IEEE ainsi que discuter du concept d'arrondi en Java et conclure par les avantages et inconvénients de ce modèle.

Les deux premières parties de ce chapitre sont très largement inspirées de [13] qui lui-même est inspiré de [3].

3.1 Le concept

L'arithmétique stochastique discrète est une approche probabiliste pour l'estimation de l'erreur relative d'un nombre. Comme indiqué plus haut, l'idée est de maintenir, en parallèle d'un nombre, trois (ou plus généralement N) autres nombres qui sont, au départ, égaux au premier nombre. Au fur et à mesure que le nombre va subir des opérations mathématiques, on va répercuter ces opérations sur les nombres mis en parallèle mais le mode d'arrondi utilisé est tiré au sort. Cette technique va donc permettre d'obtenir des nombres aléatoires, mais extrêmement proches. Plus le nombre d'opérations va augmenter, plus l'accumulation des erreurs d'arrondi pourrait se retrouver trop grande.

Après avoir exécuté un certain nombre d'opérations, on obtient N tirages de la variable aléatoire modélisée par [13]

$$R = r + \sum_{i=1}^{Sn} g_i(d) 2^{E_i - p} \varepsilon_i (\alpha_i - h_i)$$

où

- g_i représente des constantes ne dépendant que des données et de l'algorithme ;
- E_i les exposants des résultats intermédiaires ;
- α_i la quantité perdue lors de la troncature ou de l'arrondi ;
- h_i les perturbations aléatoires ;
- ε_i signes des résultats intermédiaires ;

— r le vrai résultat mathématique ;

— n le nombre d'opérations pendant l'exécution.

Une démonstration est faite dans [3].

On peut donc estimer le nombre de chiffres significatifs exacts $C_{\bar{R}}$ avec l'équation suivante :

$$C_{\bar{R}} = \log_{10}(\frac{\sqrt{N}|\bar{R}|}{\sigma\tau_\beta})$$

La validité de la méthode est aussi discutée dans [3].

Par exemple, avec un $C_{\bar{R}}$ de 5, on aura une erreur relative de l'ordre de 10^{-5} et donc la valeur de base possède 5 chiffres significatifs en commun avec ses valeurs parallèles.

3.2 Les arrondis selon l'IEEE

La norme IEEE 754[22] est une norme pour la représentation des nombres à virgule. Elle est employée dans la majorité des systèmes de calculs en informatique et notamment en Java. Cette norme défini les quatre modes d'arrondis suivants :

— Vers moins l'infini : on arrondi vers en haut.

— Vers plus l'infini : on arrondi vers en bas.

— Vers zéro : on arrondi vers en haut si le nombre est négatif et en bas si le nombre est positif.

— Au plus proche : si le nombre est entre deux, on l'arrondit à la valeur la plus proche avec un bit de poids faible de 0. C'est le mode d'arrondi utilisé par Java [15].

3.3 Le cas Java

Java utilise le mode d'arrondis au plus proche. Il n'est pas possible d'influencer le mode d'arrondis utilisé, pour cela il faudrait travailler directement au niveau du CPU et modifier au bon moment les modes d'arrondis de ce dernier. Afin de parer à ce problème, on va utiliser, comme pour les intervalles, la fonction *Math.ulp* pour simuler ces arrondis. Un ulp (*Unit in Last Place* en anglais) est le poids du plus petit bit significatif de la mantisse d'un flottant. Cela représente la distance entre le nombre et son suivant sur la droite des nombres à virgule flottante[10][14]. Un chapitre sur la validité de ce concept est disponible dans [10].

4. Différentiation automatique

Les dérivés mathématiques sont omniprésentes, elles sont utilisées dans un nombre énorme de domaines comme par exemple l'optimisation ou encore la simulation numérique. Il existe principalement deux méthodes permettant de la calculre, la différenciation symbolique et la différenciation numérique. Ces méthodes souffrent pourtant de certains problèmes. La différenciation automatique permet de pallier à divers problèmes rencontrés dans ces deux cas.

La différenciation automatique, est un ensemble de techniques permettant de calculer la dérivée d'une fonction spécifiée dans un programme informatique. Cette technique se base sur le fait que n'importe quel programme exécute une suite d'instructions élémentaires : addition, soustraction, division et multiplication, ainsi que des applications de fonctions "simples" : sinus, cosinus, exponentiel, puissance, ... En utilisant le principe de "dérivé en chaîne", il est possible d'obtenir la valeur de la dérivée automatiquement.

4.1 Avantages

Les deux autres méthodes couramment utilisées pour le calcul de la dérivée sont la différenciation symbolique et la différenciation numérique. Ces méthodes comportent néanmoins quelques problèmes, d'une part le passage vers le monde numérique peut poser problème pour la méthode symbolique et la méthode numérique souffre énormément des problèmes de discrétisation.

La différenciation symbolique utilise des symboles, plutôt que des nombres, et utilise les règles de dérivation connues pour construire la formule générale de la dérivée. Cette méthode souffre du fait que le code produit pour le calcul de la dérivée est souvent inefficace et a de la difficulté à convertir un programme informatique en une seule expression. La différenciation numérique possède des problèmes liés aux erreurs d'arrondis dans le processus de discrétisation et d'annulation. Les deux méthodes ont finalement des lacunes dans le calcul des dérivés d'ordres supérieurs (augmentation du temps de calcul et des erreurs) et, finalement, ces deux modèles sont lents dans le calcul des dérivés partielles. Or, le gradient est souvent la base de nombreux algorithmes connus. La différenciation automatique permet de résoudre ces problèmes[21]. L'autre grand avantage de la différenciation automatique c'est que l'on a pas besoin de connaître la fonction pour en calculer la dérivé.

4.2 Dérivation en chaîne

Le principe de base de la différenciation automatique est le principe de dérivé en chaîne. Pour une fonction $f(x) = g(h(x))$, la dérivé de f par rapport à x donne :

$$\frac{df}{dx} = \frac{dg}{dh}\frac{dh}{dx}$$

on a donc un moyen de calculer la dérivée de chaque bout d'un programme et de les assembler pour obtenir la dérivée complète :

$$\begin{aligned}
&f(x) = e^{\sin(x)} \\
&g(x) = e^x \quad g'(x) = e^x \\
&h(x) = \sin(x) \quad h'(x) = \cos(x) \\
&\frac{dg}{dh} = e^{\sin(x)} \\
&\frac{dh}{dx} = \cos(x) \\
&f'(x) = e^{\sin(x)} * \cos x
\end{aligned}$$

4.3 Accumulation avant

L'accumulation avant consiste à calculer la dérivée de chaque expression au fur et à mesure que le programme s'exécute et utilise la règle de la dérivation en chaîne (avec la dérivée interne) pour calculer le résultat final d'une fonction. Prenons pour l'exemple la fonction suivante :

$$h(x) = \sin(x^2) * \sqrt{e^{(x-1)}}$$

la dérivée de la fonction est :

$$h'(x) = (\cos(x^2) * 2x) * \sqrt{e^{(x-1)}} + (\sin(x^2) * (\frac{1}{2 * \sqrt{e^{(x-1)}}} * e^{(x-1)}))$$

Prenons maintenant une fonction Java qui implémente cette fonction :

Listing 1 – Exemple d'une fonction $h(x)$ implémentée en Java

```java
public static double h(double x)
{
    return Math.sin(x*x) * Math.sqrt(Math.exp(x-1.0));
}
```

Maintenant on va décomposer les opérations de cette fonction et illustrer, pas à pas, le calcul de la dérivée de cette dernière :

TABLE 4.1 – Décomposition d'une fonction Java en sous-opérations et calcul de la dérivée

Opération effectuée par le programme	**Opération ajoutée pour le calcul de la dérivée**
$w_1 = x$	$w_1' = 1$
$w_2 = x^2$	$w_2' = 2x * w_1' = 2x$
$w_3 = \sin(w_2) = \sin(x^2)$	$w_3' = \cos(w_2) * w_2' = \cos(x^2) * 2x$
$w_4 = x - 1$	$w_4' = 1$
$w_5 = e^{w_4} = e^{x-1}$	$w_5' = e^{(w4)} * w4' = e^{(x-1)}$
$w_6 = \sqrt{w_5} = \sqrt{e^{x-1}}$	$w_6' = 1/2 * sqrt(w_5) * w_5' = 1/2 * sqrt(e^{x-1}) * exp(x-1)$
$w_7 = w_3 * w_6 = \sin(x^2) * \sqrt{e^{x-1}}$	$w_7' = w_3' * w_6 + w_3 * w_6' = \cos(x^2) * 2x * \sqrt{e^{x-1}} + \sin(x^2) * \frac{1}{2} * sqrt(e^{x-1}) * e^{(x-1)}$

4.4 Les nombres duaux

Précédemment, il est indiqué qu'on peut calculer la dérivée d'une fonction sans forcément la connaître. On doit pourtant trouver un moyen d'interagir avec cette fonction afin de pouvoir calculer la dérivée en parallèle de l'exécution même de la fonction. C'est à ce moment là qu'interviennent les nombres duaux. Un nombre dual est un nombre x qui n'est pas représenté par une seule valeur mais par un couple de valeurs : $x = \{x_1, x_2\}$. À ce moment-là, on peut voir x comme un objet et, lors de chaque opération, on court-circuite le programme pour passer notre propre implémentation de l'addition. Prenons l'addition pour exemple :

Listing 2 – Exemple d'un nombre dual avec une opération en Java

```
public class Dual
{
    protected int x1;
    protected int x2;

    public Dual(int x1, int x2)
    {
        this.x1 = x1;
        this.x2 = x2;
    }

    public Dual add(Dual a)
    {
        return new Dual(a.x1+this.x1,a.x2+this.x2);
    }
}
```

A chaque fois que l'opération d'addition va être effectuée sur un nombre dual, ce sont les deux valeurs qui vont être modifiée. C'est exactement ce que l'on souhaite avoir pour la différenciation automatique. Grâce à Cojac, on peut remplacer chaque *double* et chaque *float* par un objet et court-circuiter chaque opération sur l'objet afin de calculer la dérivée en parallèle de l'exécution du programme. C'est par ce mécanisme de remplacement que devient possible la différentiation automatique.

Deuxième partie

Conception

5. Cojac : les wrappers

Actuellement, pour le remplacement des types primitifs par des types objets, Cojac a besoin de classes spéciales appelée “wrapper”. Ce sont ces classes qui vont fournir les objets pour opérer le remplacement. Ces classes doivent avoir une spécification bien précise. Pour commencer, on verra cette spécification via les diagrammes de classes généraux de ces deux wrappers (un pour les doubles et un pour les floats). Par la suite, on aura un brève aperçu des méthodes “magiques” de Cojac et on finira par la présentation d’un “bug” qui peut se révéler très ennuyeux et dont la résolution n’est pas abordée dans le cadre de ce projet.

5.1 Diagrammes de classes

Voici les diagrammes de classes représentant les deux wrappers ainsi que les méthodes qu’il faut implémenter pour leur utilisation. Pour l’implémentation des wrappers, le choix a été fait d’utiliser pour toutes les méthodes (sauf les méthodes hérités de *Numbers* et de *Comparable<?>*) des accès statiques.

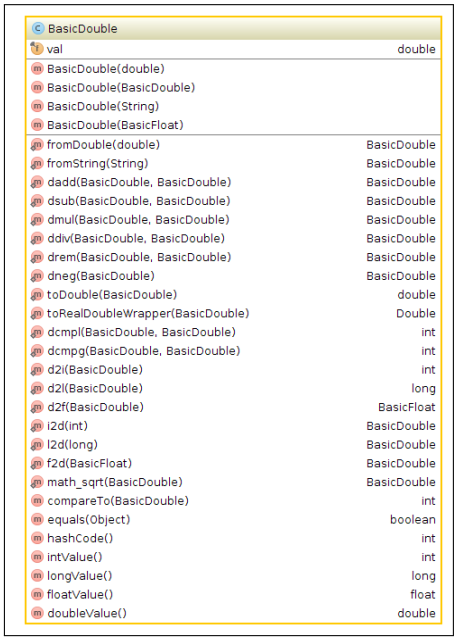

FIGURE 5.1 – Le wrapper pour le type double

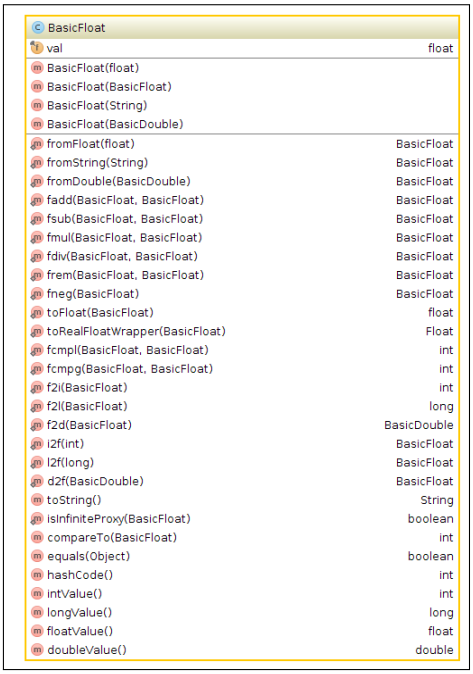

FIGURE 5.2 – Le wrapper pour le type float

5.2 Méthodes magiques

Actuellement, il n'est possible, pour l'utilisateur de Cojac, de communiquer avec l'agent qee via des méthodes "magiques". Ce sont des méthodes dont le nom possède le préfixe "COJAC_MAGIC_DOUBLE_" (respectivement "COJAC_MAGIC_FLOAT_" pour le wrapper des floats) et dont la signature de la méthode prend en paramètre une variable de type double. Lors de l'instrumentation d'une classe par l'agent Cojac, si celle-ci contient une méthode avec cette signature, alors elle va être "remplacée" par la méthode contenue dans la classe du wrapper des doubles qui possède exactement la même signature, à l'exception près que ce n'est pas un double qu'elle prend en paramètre, mais un objet du type du wrapper des doubles. Il ne faut pas oublier que c'est à l'exécution du programme que Cojac va procéder à l'instrumentation des classes chargées par la JVM et que donc, au moment de l'appel de la méthode magique du côté du code utilisateur avec un double, c'est en fait la méthode de la classe du wrapper qui sera appelée avec l'objet représentant le double dans le code utilisateur. La figure 5.3 représente ceci.

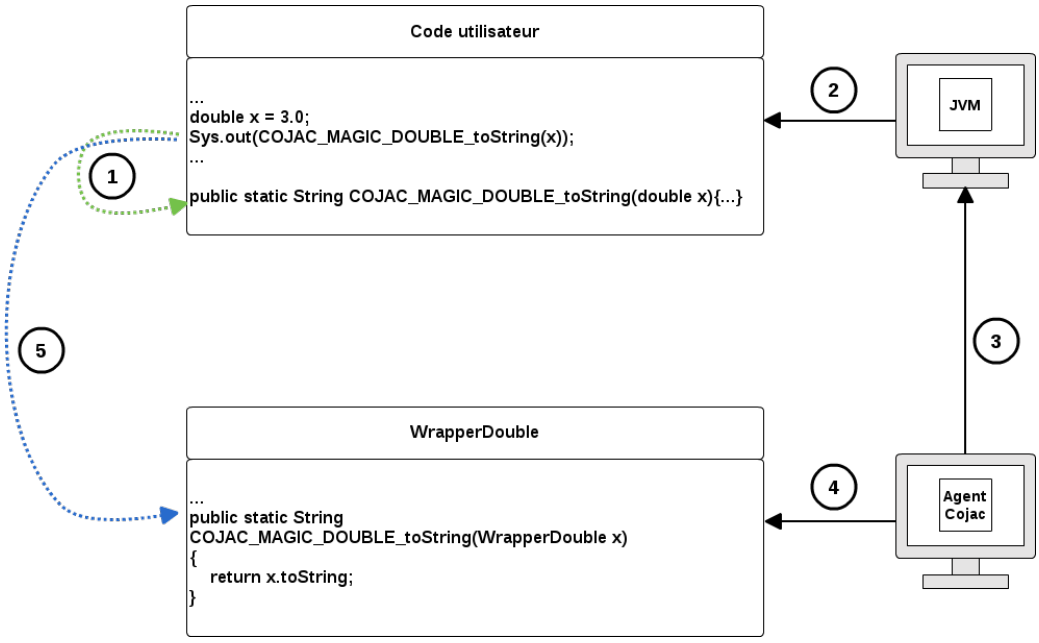

FIGURE 5.3 – Représentation de l'appel à une méthode "magique"

1. Le classe de l'utilisateur est compilée avec une méthode magique à l'intérieur. L'appel à la méthode pointe effectivement vers la méthode contenue à l'intérieur de la classe de l'utilisateur.
2. À l'exécution du programme, la machine virtuelle Java charge la classe qui doit être exécutée.
3. Lors du chargement de la classe, l'agent Cojac va procéder au remplacement de tous les types double en objets de type *WrapperDouble* et détecte aussi la présence de d'une méthode magique.
4. L'agent effectue une modification de la référence de la méthode du code utilisateur afin que l'appel de cette dernière soit dirigée vers la méthode contenue dans la classe du wrapper.
5. Lors de l'appel de la méthode à l'exécution du code, c'est la méthode du wrapper qui est invoquée et non celle du code de l'utilisateur.

6. Arithmétique d'intervalles

Comme il n'y a pas, actuellement, de librairie acceptable pour la réalisation de ce projet, la décision a été prise d'implémenter nous-mêmes, naïvement, une classe *DoubleInterval* représentant ces intervalles. Ce chapitre exposera les différents concepts à utiliser et à définir afin de produire cette classe. Pour commencer, ce chapitre exposera la manipulation des arrondis en Java, la problématique des fonctions trigonométriques et de l'opération modulo et finira par une explication sur les tests conçus pour cette classe. Une partie de conception de test est également présente mais elle se trouve dans le chapitre sur l'"arithmétique stochastique discrète".

6.1 Arrondis

En Java, il n'est pas possible d'avoir un contrôle sur les arrondis effectués sur des types primitifs. Afin de parer à ce problème, on utilise la fonction $Math.ulp()$ disponible dans la librairie standard Java.

Lors de chaque opération qui demande le résultat d'un calcul, la classe *DoubleInterval* va enlever un ulp à la borne inférieure de l'intervalle et ajouter un ulp à la borne supérieure. De cette manière, on encadrera à coup sûr le résultat exact du calcul.

Par contre, parfois on sera bien trop pessimiste. Il est possible, via certaines techniques, de savoir si le nombre à besoin d'un arrondis où non. Dans notre cas, on force l'ajout et le retrait d'un ulp à chaque fois.

6.2 Fonctions trigonométriques

Comme expliqué lors de l'analyse, les fonctions trigonométriques sont un peu plus pénible à implémenter que les autres fonctions : elles ne sont pas entièrement monotones. L'idée est de ramener l'intervalle à un intervalle équivalent mais dont les bornes inférieures et supérieures se trouvent dans l'intervalle $[-2\pi; 2\pi]$. Une fois l'intervalle ramené, on découpe cet intervalle en sous-intervalles monotones comme représentés aux figures 6.1 et 6.2.

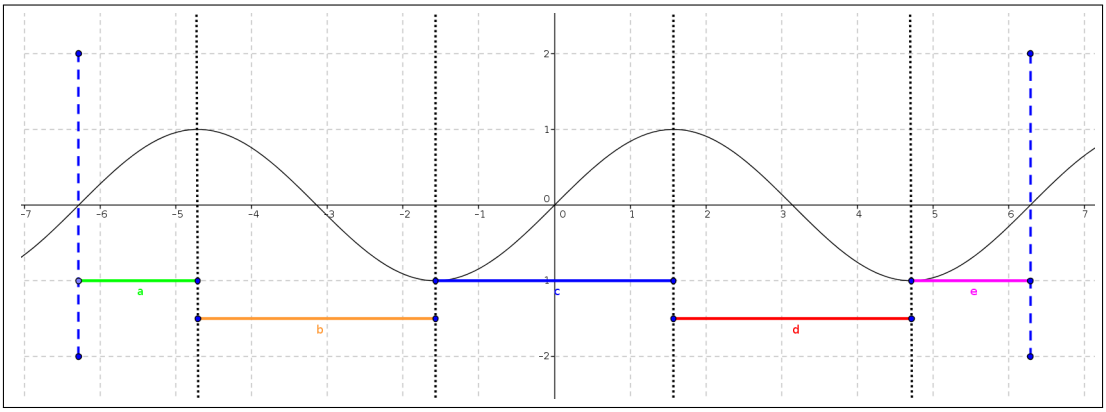

FIGURE 6.1 – Découpage en sous-intervalles monotones de la fonction sinus

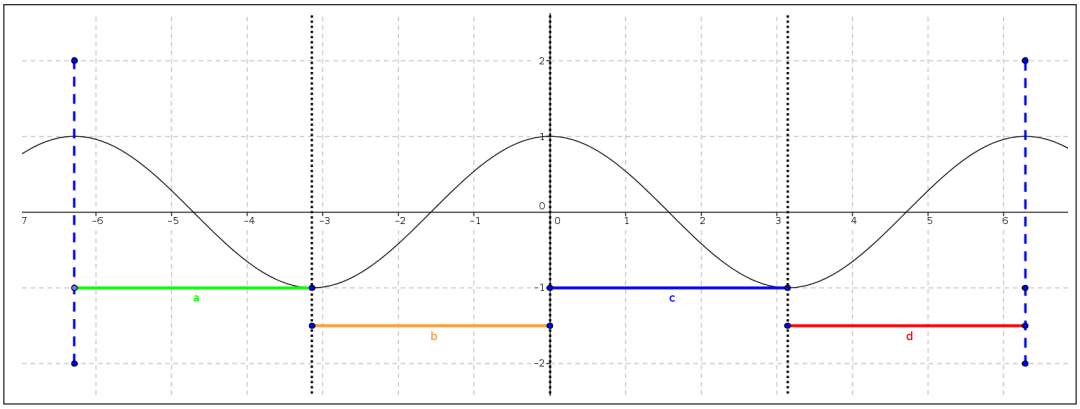

FIGURE 6.2 – Découpage en sous-intervalles monotones de la fonction cosinus

Une fois cela fait, il ne reste qu'à tester les différentes valeurs des bornes inférieures et supérieures de l'intervalle concerné et d'en déterminer l'intervalle résultant. Par exemple, si notre intervalle vaut $[-2; 1]$ et qu'on applique la fonction sinus dessus, on obtiendra $[-1; 0.84]$ comme indiquer à la figure 6.3.

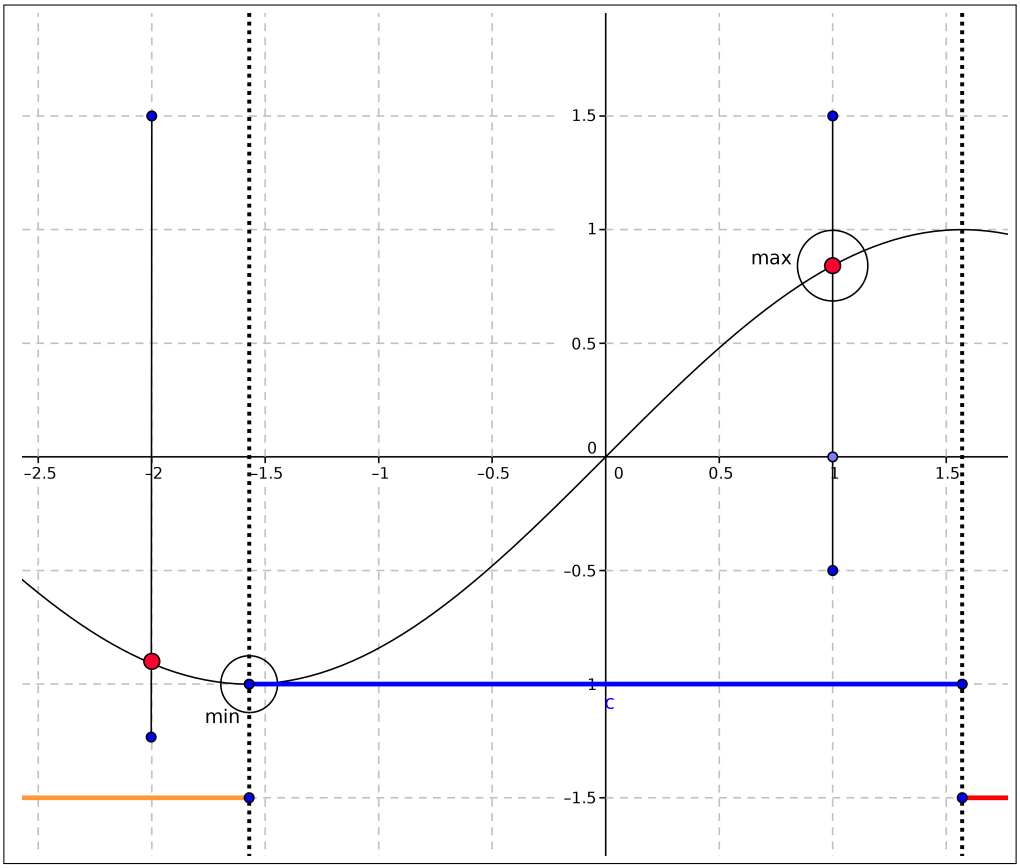

FIGURE 6.3 – L'intervalle $[-2; 1]$ donne comme réponse l'intervalle $[-1; 084]$, il y a un minimum en $x = -\frac{\pi}{2}$ et le maximum se trouve en $x = 1$

La tangente donne une fonction à l'allure légèrement différente, mais l'idée reste la même : séparer l'intervalle en plusieurs sous-intervalles (voir figure 6.4) ramené, cette fois-ci, entre $-\pi$ et π et tester les valeurs des bornes inférieures et supérieures.

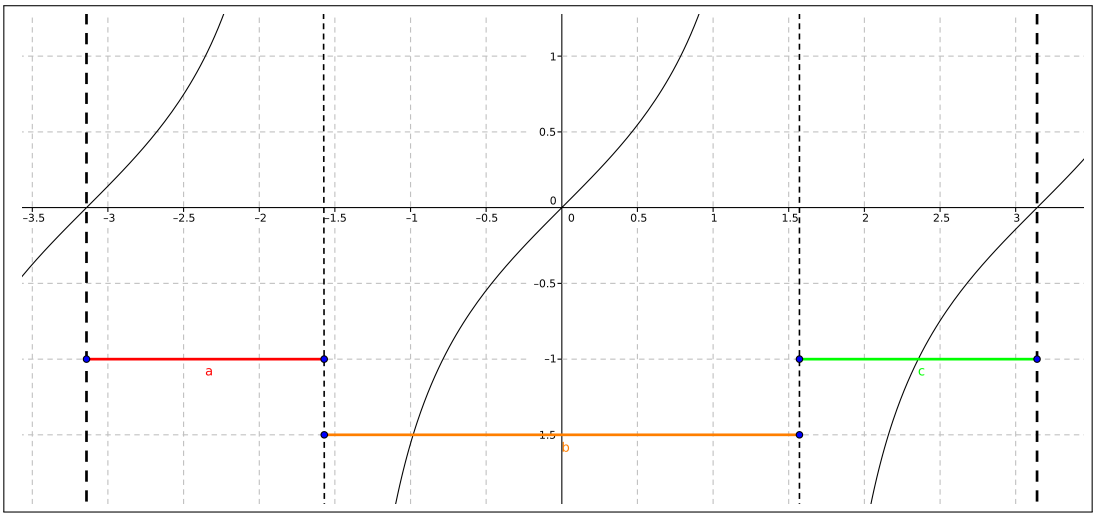

FIGURE 6.4 – Découpage en sous-intervalle monotone de la fonction tangente

Les autres fonctions trigonométriques ne représentent pas de difficultés particulières à l'implémentation, elles sont toutes soit entièrement monotones, soit symétriques (voir figures 6.5 et 6.6)

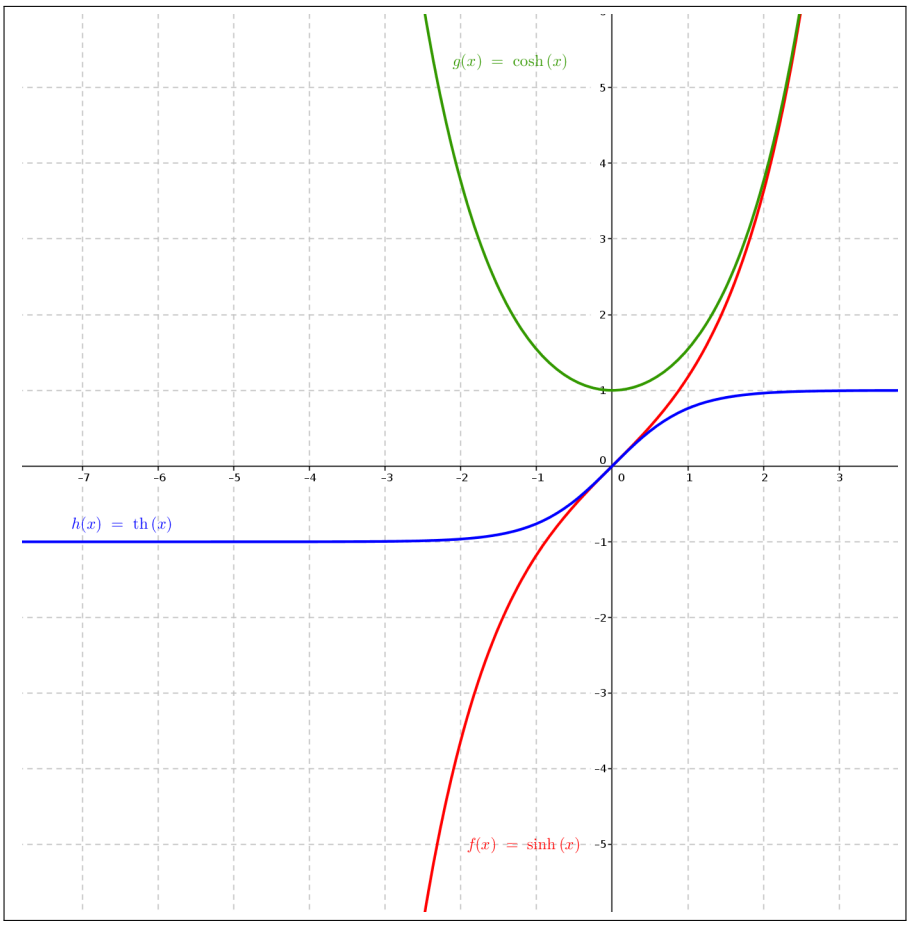

FIGURE 6.5 – Les fonctions hyperboliques

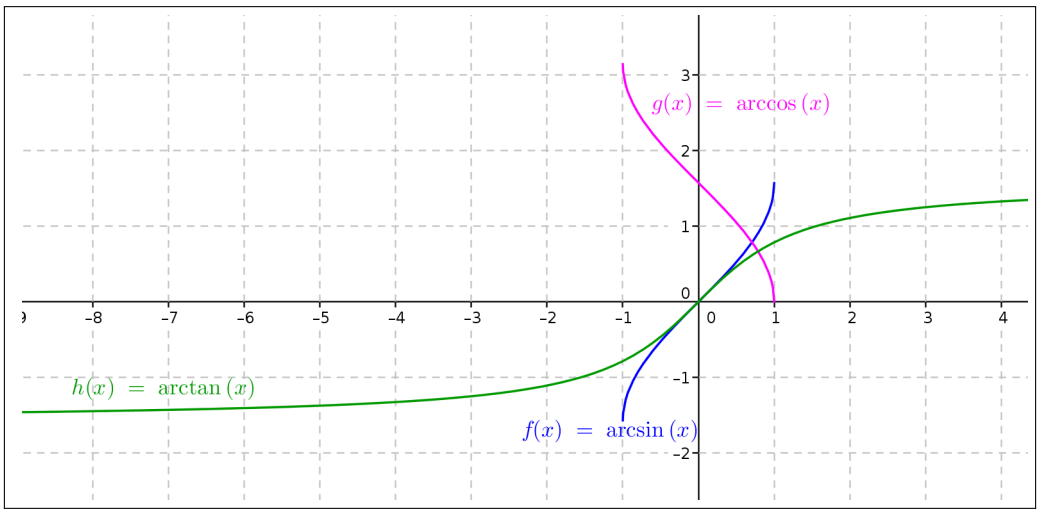

FIGURE 6.6 – Les fonctions trigonométriques inverses

6.3 Modulo

Le modulo présente lui aussi quelques difficultés à implémenter, il faut garder à l'esprit qu'on exécute le modulo sur des nombres à virgule flottante et que donc certains résultats peuvent être assez surprenants. Prenons par exemple le calcul suivant : $[5; 7]\%[3; 4.5]$ la valeur maximale de l'intervalle résultant sera le nombre se trouvant juste avant 4.5 et la valeur minimale sera 0. On obtiendrait donc l'intervalle suivant : $[0; 4.5[$, où la borne supérieure n'est pas comprise dans l'intervalle. L'autre problématique c'est qu'il existe une multitude de cas à prendre en compte et que les risques de créer des "bugs" sont relativement élevés. Afin de pallier ces problèmes et de ne pas prendre trop de temps à la réalisation de la librairie, on a pris la décision de retourner un intervalle pessimiste (il est possible de le réduire encore) mais très simple à implémenter : on retourne l'intervalle $[0; max(|\underline{x}|, |\overline{x}|)$.

Il est possible de consulter une librairie open-source (boost [4] par exemple) pour voir comment est implémentée l'opération modulo en C/C++ et de l'adapter pour Java. L'intervalle retourné n'est pas faux, il est juste très pessimiste et on pourrait faire bien mieux.

6.4 Stabilité du nombre

À chaque fois que les bornes de l'intervalle sont modifiées, on procède à une vérification de la stabilité du nombre (l'erreur relative) par rapport à un seuil. On peut trouver la formule de calcul de l'erreur relative dans l'analyse de cette partie. Le seuil de référence est, quant à lui, précisé par l'utilisateur.

6.5 Tests sur les intervalles

La classe permettant de manipuler des intervalles s'appuie presque exclusivement sur des concepts mathématiques ou bien un grand nombre de cas différents à devoir traiter. C'est pour cela, en partie, qu'il faut être sûr de pouvoir y faire confiance. Une batterie de test, explorant chaque ligne de la classe, est donc fortement conseillée dans ce cas là. Il s'agit d'un ensemble de tests unitaires, testant chaque méthode applicable sur un intervalle. La plupart d'entre eux ne font que de tester des cas simples (pour les opérations de base, ou bien dans le cas où les fonctions sont monotones). Des tests plus poussés ont été créés pour des cas plus complexes, comme le sinus ou le cosinus. L'idée de ces tests c'est de prendre un intervalle tiré au hasard, d'en calculer l'intervalle résultant et de tester chaque valeur présente dans l'intervalle de départ pour savoir si le résultat de la fonction appliquée se trouve bel et bien dans l'intervalle résultant. L'autre partie du test consiste à récupérer, pour chaque intervalle testé, la plus petite et la plus grande valeur rencontrées et de confirmer que ces deux valeurs ne sont ni plus grandes, respectivement plus petites, que les bornes supérieures et inférieures de l'intervalle résultant.

Pour les tests pratiques des wrappers, voir le chapitre de conception sur l'arithmétique stochastique discrète.

7. Arithmétique stochastique discrète

Ce chapitre va présenter la conception de la partie “arithmétique stochastique discrète” au sein de Cojac. On va commencer par voir les deux wrappers pour le type double et le type float, exposer les différents cas de tests pratiques visant à montrer l'utilité d'un tel modèle et conclure par un comparatif entre les modèles d'“arithmétique d'intervalles” et d'“arithmétique stochastique discrète”.

Comme mentionné dans l'analyse, on ne peut pas modifier le comportement de l'arrondi en Java. On utilise donc des ajouts ou des soustractions de ulp sur les nombres pour simuler l'arrondi (c'est bien plus pessimiste, mais le résultat est exploitable).

7.1 Les wrappers

Les deux wrappers de cette partie devront contenir les attributs suivants :

— une valeur de base nommée **value** et notée R, qui représente le nombre de départ

— n valeur maintenue en parallèle notée R_i et étant aléatoirement arrondie lors de chaque opération

Lors de chaque opération qui modifie R (il y a des opérations ou fonctions qui ne modifient pas la valeur même, comme par exemple la valeur absolue), on applique la même opération à R_i mais on tire au hasard un mode d'arrondi, ce qui va conduire à une addition ou soustraction d'un ulp de la valeur.

A chaque fois que la valeur de R est modifiée, on procédera également à une vérification de la stabilité du nombre (l'erreur relative) par rapport à un seuil. On peut trouver la formule de calcul de l'erreur relative dans l'analyse de cette partie. Le seuil de référence est, quant à lui, précisé par l'utilisateur.

7.2 Conception des tests

Pour la création de tests pratiques pour la partie d'arithmétique d'intervalles et d'arithmétique stochastique discrète, j'ai utilisé les trois “problèmes” suivants :

— Le polynôme de Rump

— Le calcul de Pi via une série infinie

7.2.1 Polynôme de Rump

Le polynôme de Rump[1] (voir extrait de code 3) est une fonction mathématique qui est réputée pour être instable sur ordinateur. Le même polynôme, compilé avec le même logiciel donne des résultats différents sur des processeurs différents à cause de leurs implémentations différentes. L'idée du test est de montrer que l'utilisation de l'arithmétique stochastique discrète et de l'arithmétique par intervalles permet de détecter qu'un problème a eu lieu.

Le polynôme de Rump :

$$R(x,y) = \frac{1335}{4}y^6 + (11x^2y^2 - y^6 - 121y^4 - 2)x^2 + \frac{11}{2}y^8 + \frac{x}{2y}$$

Listing 3 – Implémentation du polynôme de Rump en Java

```java
public static double rumpPolynom(double x, double y)
{
    double res = 1335.0;
    res = res * Math.pow(y, 6.0);
    res /= 4.0;
    double tmp = x * x;
    double tmp2 = 11.0 * tmp;
    tmp2 *= y * y;
    tmp2 -= Math.pow(y, 6.0);
    tmp2 -= 121.0 * Math.pow(y, 4.0);
    tmp2 -= 2.0;
    tmp *= tmp2;
    res += tmp;
    tmp = 11.0 * Math.pow(y, 8.0);
    tmp /= 2.0;
    res += tmp;
    res += x / 2.0 * y;
    return res;
}
```

7.2.2 Calcul de PI

π est un nombre qui peut être représenté via la série suivante :

$$\pi = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7}\cdots$$

L'idée du test est de calculer π sur machine avec la fonction suivante et de voir si le calcul dégénère ou non :

Listing 4 – Implémentation du calcul de π en Java

```java
public static double computePi()
{
    double res = 0.0;
    int numerator = 1;
    double tmp = 4.0;
    boolean alt = true;

    while (tmp > 0.1E-5)
    {
        if (alt)
        {
```

```
            res += tmp;
            numerator += 2;
            tmp = 4.0 / numerator;
            alt = false;
        }
        else
        {
            res -= tmp;
            numerator += 2;
            tmp = 4.0 / numerator;
            alt = true;
        }
    }
    return res;
}
```

8. Différentiation automatique

Ce chapitre va présenter la conception des deux classes “DoubleDerivation” et “FloatDerivation” en expliquant les choix faits pour l'interaction avec l'utilisateur dans la spécification des variables à dériver et dans la récupération de l'information. Ensuite, on verra comment faire pour indiquer qu'une variable est dérivable en passant par un exemple d'une fonction à plusieurs variables, puis on discutera de quelques dérivés particulières ou utile dans le cadre de ce projet et pour finir, on passera à la conception des tests pratiques pour ces classes.

8.1 Interaction avec l'utilisateur

Le but de la partie sur la différentiation n'est pas de fournir un moyen permettant la détection d'erreurs numériques dans un programme Java, mais de fournir un outil permettant de dériver n'importe quelle fonction d'un programme écrit en Java. Lors de l'exécution du programme sur la machine virtuelle, l'agent Cojac va instrumenter toutes les variables de type double ou float.

Pour l'utilisation de ce modèle, il faut donc fournir à l'utilisateur les fonctionnalités suivantes :

— la possibilité d'indiquer à l'agent Cojac qu'elle est la variable qu'il faut dériver.

— la possibilité de récupérer la dérivée d'une variable.

Il ne faut pas oublier que Cojac ne va pas dériver une fonction, mais une variable ayant subi diverses opérations, comme dans une fonction. Même les variables qui ne se trouvent pas à l'intérieur de fonctions vont être dérivées automatiquement, il faut donc que l'utilisateur fasse attention lorsqu'il spécifiera les variables à dériver et leur récupération.

8.2 Spécification des variables à dériver

La méthode utilisée pour spécifier à Cojac qu'une variable doit être dérivée est très simple, il suffit d'initialiser sa dérivé interne à 1 (pour indiquer qu'elle doit être dérivée) ou à 0 (pour indiquer que c'est une constante). Prenons la fonction à deux variables suivantes :

$$f(x, y) = x^2 y$$

Si on souhaite obtenir $\frac{df}{dx}$ alors on considère que la variable y est une constante et on la traite tel que :

$$\frac{df}{dx} = 2xy$$

Pour obtenir ce résultat, on a en fait supposé que la dérivée y' de y est nulle et on applique les règles de dérivation standards :

$$
\begin{aligned}
&f(x,y) = x^2 y \\
&g(x) = x^2 \\
&h(y) = y \\
&f(x,y) = g(x) \cdot h(y) \\
&f'(x,y) = g'(x) \cdot h(y) + g(x) \cdot h'(y)
\end{aligned}
$$

Et comme $h'(y)$ vaut 0 :

$$f'(x,y) = g'(x) \cdot h(y) + g(x) \cdot 0 = g'(x) \cdot h(y) = 2xy$$

C'est donc en mettant les dérivés internes de départ à 0 ou à 1 que la variable va être dérivée automatiquement, sans avoir à indiquer autre chose.

8.3 Des dérivés particulières

Comme mentionné dans le chapitre sur l'arithmétique d'intervalles, on manipule des objets qui représentent des doubles et on doit donc fournir toutes les opérations standards sur les doubles via ces objets. On se retrouve donc parfois avec des cas bizarres comme par exemple des dérivés sur une puissance quelconque ou bien sur des modulos.

8.3.1 Puissance

La dérivée usuelle de la puissance est définie comme suit :

$$
\begin{aligned}
&f(x) = a \cdot x^n \\
&f'(x) = a(n-1)x^{(n-1)}x'
\end{aligned}
$$

Malheureusement, cela n'est valide que pour les puissances entières, il faut donc trouver la formule générale pour le calcul de la dérivée des puissances de la forme $f(x) = g(x)^{h(x)}$:

$$
\begin{aligned}
&f(x) = g(x)^{h(x)} \\
&\frac{d}{dx}\ g(x)^{h(x)} = \frac{d}{dx}\ e^{h(x)\cdot\ln(g(x))} \\
&\frac{d}{dx}\ e^{h(x)\cdot\ln(g(x))} = e^{h(x)\cdot\ln(g(x))} \cdot (h'(x) \cdot \ln(g(x)) + h(x) \cdot \frac{1}{g(x)} \cdot g'(x)) \\
&= g(x)^{h(x)} \cdot (\frac{h(x) \cdot g'(x)}{g(x)} + \ln(g(x)) \cdot h'(x) \\
&\text{Avec } g(x) > 0
\end{aligned}
$$

Avec cela, on a la formule générale pour calculer n'importe quelle dérivé de puissances.

8.3.2 Modulo

En mathématiques, le modulo n'est défini que sur les nombres entiers, cela devient donc un peu surprenant d'avoir, dans le cas de Java, l'opérateur modulo défini sur les nombres à virgule. Si on s'intéresse à la fonction modulo sous la forme suivante :

$$f(x) = g(x) \ mod \ h(x)$$

Si on ne s'occupe que de la partie $g(x)$ de la fonction et que l'on considère que $h(x)$ est uniquement une constante, on est capable de calculer la fonction en un point donné car la fonction est définie partout ,sauf tous les h(x) (voir figure 8.1).

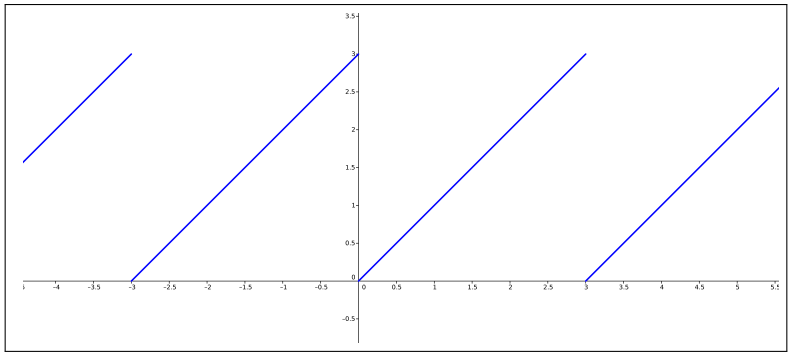

FIGURE 8.1 – Représentation graphique de la fonction $f(x) = x \ mod \ 3$

La fonction est donc dérivable en tout point sauf quand x est un multiple de $-3; 3$. Si x fait partie de la deuxième partie de l'équation $f(x) = g(x) \ mod \ h(x)$ alors on obtient dans ce cas des fonctions relativement étranges (voir figure 8.2 qui n'ont pas de solutions générales pour le calcul de la dérivée.

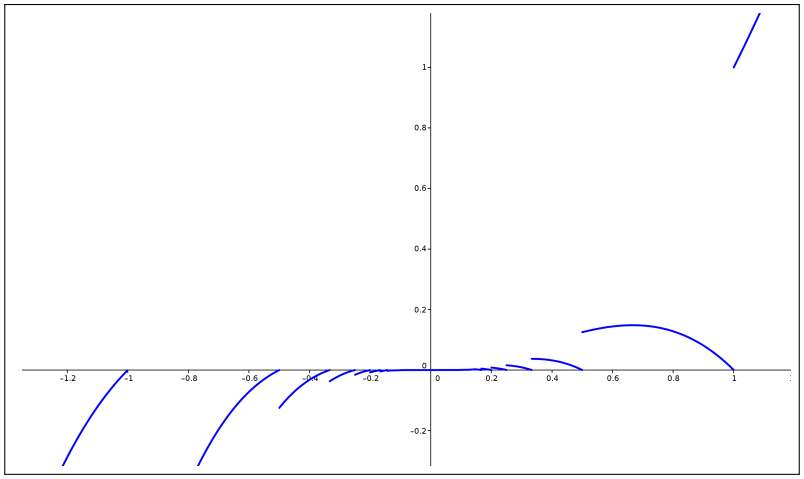

FIGURE 8.2 – Représentation graphique de la fonction $f(x) = x^2 \ mod \ x^3$

Afin de pallier ce genre de désagrément, on a pris la décision de ne pas s'occuper de la dérivée de fonction modulo si le second argument de la fonction est indiqué comme dérivable. La dérivée d'une fonction comme à la figure 8.2 donnera comme résultat **NaN** puisse que le dénominateur n'est pas constant.

8.4 Conception des tests

La majeure partie de l'implémentation se fait dans les wrappers à fournir à Cojac pour le remplacement. Il s'agit juste d'appliquer les règles de dérivation. La majeure partie des tests est de créer diverses fonctions plus ou moins complexes, de calculer, à la main, la dérivée générale de ces fonctions et d'en créer la fonction Java qui calcule la valeur de la dérivée d'une fonction en un point et de comparer les résultats obtenus entre la dérivée calculée automatiquement et la dérivée calculée à la main.

Voici les fonctions, ainsi que leurs dérivées et les divers opérations testées, qui seront utilisées pour les cas de tests pratiques de la différentiation automatique :

TABLE 8.1 – Fonctions qui seront utilisées pour le test de la différentiation automatique

Fonction f	**Dérivée de f**	**Fonctionnalités testées**
$4x^3$	$8x2$	puissance, constante
x^x	$x^x \cdot (\ln(x) + 1)$	exposant variable
$\sin(x) + 4x$	$\cos(x) + 4$	sinus, addition
$\cos(x^2) - 4x + 3$	$-\sin(x^2) \cdot 2x - 4$	cosinus, soustraction
$4x + \tan(x)$	$\tan^2(x) + 1 + 4$	tangente, multiplication
$1/\sqrt{x}$	$-\frac{1}{2x^{\frac{3}{2}}}$	racine, division
$-\log(x)$	$-\frac{1}{x}$	log, négation
$\sinh(x) + \cosh(x) + \tanh(x)$	$\sinh(x) + \cosh(x) + \text{sech}^2(x)$	fonctions hyperboliques
$\text{acos}(x) + \text{asin}(x) + \text{atan}(x)$	$\frac{1}{x^2+1}$	fonctions trigonométriques inverses
e^{x^2}	$e^{x^2} \cdot 2x$	exp
3 mod x	NaN	modulo
x mod 3	1	modulo
$\|x\|$	NaN	valeur absolue

Troisième partie

Implémentation et résultats obtenus

9. Librairie mathématique

La première tâche à réaliser dans le cadre de ce projet, c'est terminer le remplacement des méthodes de la librairie mathématique standard Java (*java.lang.Math*). Une première partie a déjà été réalisée dans un précédent projet[12]. L'idée est que lorsque on utilise la librairie standard, les wrappers *BigDecimalDouble* et *BigDecimalFloat* doivent convertir leurs champs de type BigDecimal en un simple double ou float et on perd toute la précision obtenue jusque là.

Ce chapitre présente la technique utilisée pour spécifier les méthodes à remplacer, par quelle autre ainsi que quelques conseils sur l'implémentation des méthodes de remplacement et une conclusion sur ce qui est fait et sur ce qu'il reste à faire.

9.1 Remplacement des méthodes

La classe qui permet de spécifier le remplacement des méthodes dispose d'un ensemble qui contient toutes les méthodes qu'il faut remplacer. Pour cela, elle a besoin du chemin complet de la classe (java.lang.Math dans ce cas) et de la signature exacte de la méthode. Il est possible de récupérer toutes les signatures de méthodes d'une classe avec la commande `javap -s` sur la classe `java.lang.Math` (voir exemple ci-dessous). Pour terminer, on spécifie la signature de la méthode qui est contenue dans un des deux wrappers ainsi que la manière de l'appeler (méthode de classe, méthode d'objet). La commande `javap -s` fonctionne également sur un fichier *.class* et retourne les signatures de toutes les méthodes présentes dans la classe. La figure 9.1 montre le déroulement de cette action.

Voici un exemple de sortie de la commande `javap -s` :

```
$> javap -s java.lang.Math
...
public static double log(double);
    descriptor: (D)D

  public static double log10(double);
    descriptor: (D)D

  public static double sqrt(double);
    descriptor: (D)D
...
```

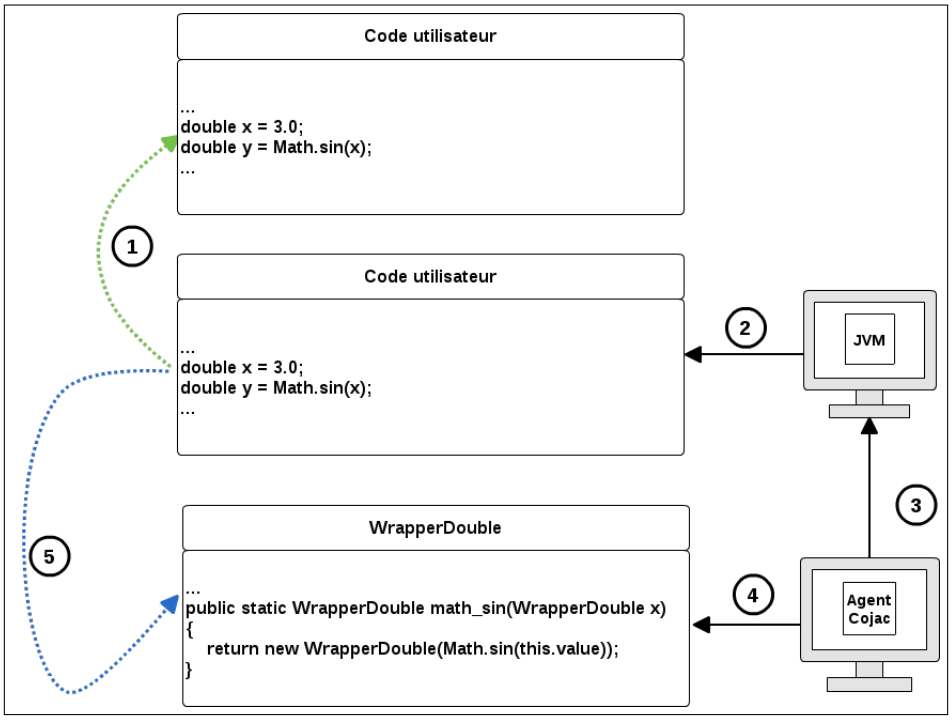

FIGURE 9.1 – Schéma du remplacement de méthode de la librairie standard

1. Le code utilisateur est compilé et l'appel à la méthode *Math.sin* pointe sur la méthode contenue dans la librairie standard
2. La classe est chargée lors de son exécution
3. L'agent Cojac intervient, parse la classe et constate qu'il faut remplacer un appel de méthode en cherchant dans son ensemble de méthodes à remplacer
4. L'agent modifie la méthode à invoquer, qui pointe désormais sur la méthode contenue dans la classe du wrapper

9.2 Quelques conseils et aperçu du futur

Dans la version actuelle de Cojac et du wrapper *BigDecimalDouble*, le remplacement des méthodes de la librairie mathématique standard de Java est effectué, mais le problème vient du fait que la méthode qui remplace l'appel effectue elle-même un appel vers la librairie standard. Il faut savoir que le code des wrappers n'est pas du tout instrumenté par l'agent, donc l'appel se fait bien vers la méthode de la librairie standard Java. En faisant cela, on perd toute la précision obtenue jusque là avec la classe *BigDecimal*.

Une manière de résoudre ce problème serait d'implémenter, nous-mêmes, les méthodes mathématiques nécessaires pour les wrappers en s'appuyant sur la classe *BigDecimal*. C'est-à-dire, il faudrait utiliser les opérations existantes de la classe *BigDecimal* (addition, soustraction, multiplication, division). Ce serait tout sauf trivial dans le cas de certaines fonctions. Par exemple, pour calculer la racine carrée d'un nombre, on peut utiliser la méthode de Héron[23] :

Listing 5 – Implémentation de l'algorithme de Héron avec le type *double*

```
private static double sqrt(double x)
{
    double epsilon = 1e-14;
    double root = 1.0; double lroot = x;
    while (Math.abs(root - lroot) > epsilon) {
        lroot = root;
        root = (root + x / root) / 2.0;
    }
    return root;
}
```

et la même méthode mais cette fois avec la classe BigDecimal :

Listing 6 – Implémentation de l'algorithme de Héron avec le type *BigDecimal*

```
private static BigDecimal sqrtHeron(BigDecimal x)
{
    MathContext context = new MathContext(100);
    BigDecimal epsilon = new BigDecimal(1e-100,context); // precision
    BigDecimal root = new BigDecimal(1.0,context);
    BigDecimal lroot = x.abs(context);

    while(root.subtract(lroot,context).abs(context).compareTo(epsilon) == 1)
    {
        lroot = root.abs(context);
        root = root.add(x.divide(root,context)).divide(new BigDecimal(2.0,context),context);
    }
    return root;
}
```

Il n'est donc pas impossible de re-coder chaque méthode de la librairie standard, cela prend, par contre, du temps. Un autre point important c'est l'utilisation de méthode comme *Math.abs* ou la comparaison. En effet, si on fait bêtement un appel à la méthode de la librairie standard depuis le wrapper *BigDecimalDouble*, on doit fournir un double et donc perdre aussi toute la précision, il est donc préférable d'utiliser les méthodes déjà fournies dans la classe *BigDecimal* (cela coule de source, mais il faut faire attention à ce genre de chose).
Du coté de l'implémentation, la base est présente mais il faut encore implémenter ces fonctions mathématiques nous-mêmes. La méthode de la racine carrée est déjà implémentée pour l'exemple.

10. Arithmétique d'intervalles

Ce chapitre va présenter l'implémentation de la partie d'"arithmétique d'intervalle", il commence par une indication sur les méthodes à implémenter, les interactions définis avec l'utilisateur. On continue par une présentation de la détection d'instabilités et de comparaisons douteuses et on termine par un aperçu de comportement étrange des intervalles implémentés.

10.1 Méthode à implémenter

L'ensemble des méthodes à implémenter est disponible dans les diagrammes 5.1 et 5.2. Dans ce cas (pour le modèle d'arithmétique d'intervalle), il est important aussi de court-circuiter les méthodes mathématiques afin de ne pas perdre l'information contenue par les bornes inférieures et supérieures de l'intervalle du wrapper. Derrière ces méthodes, on fait des appels vers la librairie standard Java, mais ce sont des appels contrôlés, vue que l'on passe la valeur de référence (et non pas une simple conversion vers un double) et l'on applique aussi l'opération sur les deux bornes de l'intervalle.

10.2 Interaction avec l'utilisateur

La force principale de Cojac est que l'agent agit directement au niveau du byte-code et que le développeur n'est pas obligé de mettre des annotations ou encore des interfaces à implémenter. Tout est transparent, il faut néanmoins donner à l'utilisateur un moyen de récupérer ces précieux nombres enrichis, via des méthodes magiques.

Au moment de l'exécution du programme, les variables de type *double* et *float* vont être remplacées par des types objets. Grâce à des méthodes magiques bien précises, on peut donc, depuis le code utilisateur, récupérer toute l'information nécessaire pour voir une "évolution" des différents calculs effectués dans le code : quelle est la valeur de l'intervalle, sa largeur,...

L'autre interaction se fait au moment où Cojac détecte une instabilité numérique, une exception est générée (voir ci-dessous) afin de récupérer le stacktrace de l'appel de méthode (indiquant à l'utilisateur où se trouve l'erreur) et le système affiche aussi la valeur de l'intervalle et l'erreur relative ainsi calculée.

```
Cojac has destected a unstable operation :
ch.eiafr.cojac.models.wrappers.CojacStabilityException
      at ch.eiafr.cojac.models.wrappers.IntervalDouble.checkStability(IntervalDouble.java:567)
      at ch.eiafr.cojac.models.wrappers.IntervalDouble.dadd(IntervalDouble.java:127)
      at Main.rumpPolynom(Main.java:170)
```

```
      at Main.DoubleRump(Main.java:201)
      at Main.runDoubleTest(Main.java:191)
      at Main.main(Main.java:15)
interval value :-1.1805916207174113E21:[-1.534769106932635E22;1.2986507827891526E22]
relative error :2.1818181818181817
```

10.3 Détection d'opérations instables

L'idée principale du wrapper est que lors de chaque opération qui modifie le résultat, on fasse une vérification de la stabilité du résultat via une méthode appelée *checkStability* :

Listing 7 – Méthode qui vérifie la stabilité numérique d'un nombre

```
private void checkStability()
{
    if(this.isUnStable)
    {
        return;
    }
    if (threshold < relativeError())
    {
        CojacStabilityException e = new CojacStabilityException();
        System.err.println("Cojac has destected a unstable operation :");
        e.printStackTrace(System.err);
        this.isUnStable = true;
    }
}
```

On ajoute également un attribut booléen *isUnStable* afin d'éviter d'avoir toute une série de messages d'erreurs sur la sortie d'erreur du programme. Dès qu'un nombre est considéré comme instable, on propage l'erreur dans chaque objet du wrapper qui découle d'une opération réalisée avec le nombre instable.
Pour le calcul de l'erreur relative, on utilise la formule suivante :

$$err_r = \omega(\mathbf{x})/min(|\underline{x}|, |\overline{x}|)$$

où $\omega(\mathbf{x})$ est la largeur de l'intervalle. Cette valeur est comparée avec un seuil (*threshold*) qui est déterminé par l'utilisateur.

10.4 Vérification des comparaisons

La vérification de la stabilité numérique d'un programme ne se limite pas à l'erreur relative de ses nombres. Le wrapper mis en place peut aussi effectuer des vérifications sur la comparaison entre deux nombres. Au chapitre sur l'analyse de l'arithmétique d'intervalles, on peut trouver une section sur la définition des comparaisons entre intervalles. L'idée est de fournir une méthode qui vérifie si la comparaison entre deux nombres est sûre ou non.
Prenons deux intervalles $\mathbf{a} = [-1; 1]$ et $\mathbf{b} = [0; 2]$, au moment de la comparaison, on ne peut pas être sûr que l'intervalle **b** est plus grand que l'intervalle **a** vu qu'ils possèdent un sous-intervalle ($[0; 1]$ en commun. Cojac est donc capable de détecter ce genre de comparaison et d'indiquer à l'utilisateur que la comparaison est douteuse.

10.5 Des comportements étranges

L'utilisation de ulp pour encadrer le résultat à la place de modifier le comportement de certains arrondis mène à des situations parfois si étranges qu'elles doivent être clairement indiquées à un quelconque utilisateur de ce produit.

10.5.1 Un zéro pas forcément nul

Comme on effectue une addition et une soustraction d'un ulp sur les bornes inférieures et supérieures au moment de l'addition, il se peut que le résultat soit complètement modifié alors que l'on ne s'y attend pas :

Listing 8 – Exemple d'une addition erronée

```
DoubleInterval a = new DoubleInterval(-Double.MAX_VALUE,Double.MAX_VALUE);
DoubleInterval b = new DoubleInterval(0.0);
System.out.println(DoubleInterval.add(a,b));
```

On s'attend à obtenir l'intervalle `[-Double.MAX_VALUE;Double.MAX_VALUE]` comme résultat et l'output est :

```
[-Infinity;Infinity]
```

10.5.2 Une étrange soustraction

La soustraction souffre du même problème, si on effectue l'opération $0 - 0$ le résultat n'est pas égal à 0 :

Listing 9 – Un bug avec $0 - 0$

```
DoubleInterval b = new DoubleInterval(0.0);
System.out.println(DoubleInterval.sub(b,b));
```

Avec comme output :

```
[-4.9E-324;4.9E-324]
```

.
On voit ici une des limites de la méthode que nous utilisons, si le processeur manipule deux fois le même nombre, il sait que le résultat de la soustraction vaut 0, quel que soit le mode d'arrondis. Ici, en manipulant des ulp, on fait preuve d'énormément de pessimisme.

11. Arithmétique stochastique discrète

Ce chapitre va présenter les choix d'implémentations effectués pour la partie d'"arithmétique stochastique discrète". On commencera par voir comment est réalisé la gestion de l'ensemble de nombres représentant un double, comment est réalisée la vérification de la stabilité d'un nombre, les interactions possibles avec le wrapper depuis le code utilisateur et on finira par la présentation de la variation aléatoire des arrondis des nombres.

La classe du wrapper *float* a aussi été implémentée, uniquement en utilisant le type float, pour cette partie.

11.1 Gestion des valeurs parallèles

L'arithmétique stochastique discrète précise qu'il peut y avoir N tirages différents pour effectuer l'étude probabiliste de la stabilité numérique d'un programme. Cojac doit donc pouvoir fournir un moyen de gérer jusqu'à N variables en parallèle. On va donc utiliser un tableau de N double, et, à chaque opération, on va effectuer la même opération sur chaque valeur du tableau (voir listing 10) et faire varier, aléatoirement, l'ajout ou le retrait d'un ulp sur chacune des valeurs.

Il faut prendre conscience que le fait de rajouter une boucle allant de 0 à $N-1$ lors de chaque opération va dégrader les performances (qui pour l'instant sont très mauvaises...) encore plus.

Listing 10 – L'opération d'addition est répercutée sur chaque valeur de l'ensemble

```
public static StochasticDouble dadd(StochasticDouble a, StochasticDouble b)
{
    if (a.isNan || b.isNan)
    {
        return new StochasticDouble(Double.NaN);
    }
    StochasticDouble res = new StochasticDouble(a);
    res.value = a.value + b.value;
    for (int i = 0; i < nbrParallelNumber; i++)
    {
        res.stochasticValue[i] = rndRoundDouble(a.stochasticValue[i] + b.stochasticValue[i]);
    }
    res.isUnStable = a.isUnStable || b.isUnStable;
    res.checkStability();
    return res;
}
```

11.2 Détection d'opérations instables

Comme pour le modèle d'arithmétique d'intervalles, ce modèle est chargé de vérifier la stabilité numérique d'un programme. La formule pour calculer l'erreur relative d'un nombre est la suivante (voir chapitre d'analyse sur la partie d'arithmétique stochastique discrète) :

$$10^{-C_{\bar{R}}}$$

où $C_{\bar{R}}$ est obtenu par

$$C_{\bar{R}} = \log_{10}(\frac{\sqrt{N}|\bar{R}|}{\sigma\tau_\beta})$$

Le même mécanisme de propagation de l'information sur une instabilité numérique a été utilisé pour cette partie. Si une opération est instable, alors tous les calculs ainsi que les résultats obtenus qui en découlent sont instables.

11.3 Interaction avec l'utilisateur

L'utilisateur a également la possibilité de communiquer, via des méthodes magiques, avec le wrapper de la classe double (ou de la classe float respectivement). On peut espérer, au minimum, avoir les méthodes suivantes pour savoir ce qui se passe :

— Récupération du String permettant d'obtenir les informations sur l'ensemble de valeur
— Récupération de l'erreur relative d'un nombre

Avec ces deux méthodes, l'utilisateur peut ainsi essayer de trouver, avec les stacktrace générées par Cojac, d'où peut provenir l'erreur (voir exemple 3).

11.4 Gestion de la variation du mode d'arrondis

Comme il n'est pas possible de manipuler le mode d'arrondis en Java, on procède à un ajout ou à un retrait d'un ulp sur la valeur spécifiée. On désire avoir un tirage aléatoire et équivalant sur les 4 modes d'arrondis ainsi proposés. La méthode 11 réalise cette opération.

Listing 11 – L'ajout ou le retrait d'un ulp est tiré aléatoirement

```
private static double rndRoundDouble(double value)
{
    switch (r.nextInt(4))
    {
        case 0: // default rounding mode in Java
            return value;
        case 1: // round to 0
            if (value < 0.0)
            {
                return value + Math.ulp(value);
            }
            else
            {
                return value - Math.ulp(value);
            }
        case 2: // round to negative infinity
            return value - Math.ulp(value);
        case 3: // round to positive infinity
            return value + Math.ulp(value);
        default: return value;
    }
}
```

12. Différentiation automatique

Ce chapitre va présenter l'implémentation de la partie sur la différentiation automatique, on discutera des méthodes à implémenter à l'intérieur du wrapper pour les doubles ainsi que la manière de spécifier les variables à dériver qui est un peu contre-nature.

Le wrapper pour le type float n'est pas très intéressant, dans cette partie on ne s'intéresse qu'au double. La classe du wrapper pour le type float consiste en de la délégation de méthode vers le wrapper de type double.

12.1 Méthode à implémenter

Dans le cadre de ce modèle, il faut également court-circuiter, pour bien, l'ensemble des méthodes de la librairie standard Java. Si une des méthodes de la librairie standard est tout de même invoquée, l'agent Cojac va utiliser une méthode du wrapper permettant de récupérer la valeur en double de l'objet, invoquer la méthode et construire un nouvel objet à partir de ce qui est retourné par la méthode. Résultat : on aura perdu toutes informations sur la dérivé de la variable.

Pour deux méthodes il faut faire légèrement attention à leurs spécifications. Il s'agit de *abs* et de *drem* (modulo). Il faut savoir que ces deux fonctions ne sont pas dérivables entièrement (elles le sont par partie). Par contre, leur implémentation est nécessaire : l'opérateur modulo et la méthode de la valeur absolue sont utilisés dans bon nombre de programmes et cette dernière peut être dérivable (voir analyse de la partie présente).

12.2 Spécification des variables à dériver

La méthode suivante permet de spécifier qu'une variable doit être dérivée :

Listing 12 – Méthode pour spécifier qu'une variable doit être dérivée

```
public static DerivationDouble COJAC_MAGIC_DOUBLE_specifieToDerivate(DerivationDouble a)
{
    return new DerivationDouble(a.value,1.0);
}
```

Dans un premier temps, on avait une méthode qui ne retournait rien et qui modifiait juste la valeur de la dérivé interne de la variable :

Listing 13 – Ancienne méthode pour spécifier qu'une variable doit être dérivée

```
public static void COJAC_MAGIC_DOUBLE_specifieToDerivate(DerivationDouble a)
{
    a.dValue = 1.0;
}
```

Le problème avec cette spécification est que tout objet, issu des classes des wrappers, doit être immuable. Prenons le code suivant :

Listing 14 – Un bug présent avec l'ancienne implémentation

```
x1 =  x2 = 5.0;
y1 =  y2 = 3.0;
COJAC_MAGIC_DOUBLE_specifieToDerivate(x1);
COJAC_MAGIC_DOUBLE_specifieToDerivate(y2);
dg = COJAC_MAGIC_DOUBLE_getDerivation(g(x1, y1)) + COJAC_MAGIC_DOUBLE_getDerivation(g(x2, y2));
System.out.println("g'(" + x1 + "," + y1 + ") = " + dg + " should be " + dg(x2, y2));
```

Au moment de l'instrumentation du code, Cojac découvre qu'il y a une double assignation (ligne 1) et, pour instrumenter cela, Cojac utilise l'opérateur *dup2* qui permet de cloner un objet. Donc, lors de l'instrumentation du code, l'objet est cloné et les variables x1 et x2 possèdent une référence vers le même objet. Ce qui veut dire, qu'avec l'ancienne implémentation, on avait involontairement spécifié qu'il fallait dériver x1, x2, y1 et y2. Ce qui fausse entièrement le résultat !
Avec la nouvelle version voici comment est transformé le code 15 :

Listing 15 – Un bug présent avec l'ancienne implémentation

```
x1 =  x2 = 5.0;
y1 =  y2 = 3.0;
x1 = COJAC_MAGIC_DOUBLE_specifieToDerivate(x1);
y2 = COJAC_MAGIC_DOUBLE_specifieToDerivate(y2);
dg = COJAC_MAGIC_DOUBLE_getDerivation(g(x1, y1)) + COJAC_MAGIC_DOUBLE_getDerivation(g(x2, y2));
System.out.println("g'(" + x1 + "," + y1 + ") = " + dg + " should be " + dg(x2, y2));
```

C'est plus simple d'avoir une méthode qui ne retourne rien, que d'avoir une méthode qui retourne un wrapper, tout ça parce que l'objet n'est pas immuable.

13. Conclusion sur l'implémentation

Dans la version actuelle de Cojac, les wrappers sont définis comme un ensemble de méthodes statiques à implémenter, ces méthodes représentent les opérations de bases, les constructeurs ou encore les différentes méthodes de comparaison. A chaque appel de méthode, il y a une série de vérifications qui est réalisée (voir listing 16) pour s'assurer de la cohérence de chaque objet.

Listing 16 – Méthode de l'addition avec un grand nombre de vérifications

```
public static BigDecimalDouble dadd(BigDecimalDouble a, BigDecimalDouble b)
{
    if (a.isNaN || b.isNaN)
    {
        return new BigDecimalDouble(Double.NaN);
    }
    if (a.isInfinite || b.isInfinite)
    {
        double aVal = a.getDoubleInfiniteValue();
        double bVal = b.getDoubleInfiniteValue();
        return new BigDecimalDouble(aVal + bVal);
    }
    return new BigDecimalDouble(a.val.add(b.val, mathContext));
}
```

Cela entraine une duplication du code qui est assez délirante. Afin de réduire le code produit et d'avoir un meilleur code source, en sachant que les méthodes statiques de chaque classe doivent être clairement implémentées, on pourrait avoir recours à un type énuméré (listing 17) et, dans chaque fonction, faire appel à une autre fonction (listing 18) qui procède, elle, à toutes les vérifications et retourne un résultat cohérent.

Listing 17 – Type énuméré pour chaque opération ou fonction

```
public enum WrapperOperatorOrFunctionUnary
{
    NEG,SQRT // ...
}

public enum WrapperOperatorOrFunctionBinary
{
    ADD,SUB,DIV,MUL,REM,POW // ...
}
```

Il faut bien noté que l'on a besoin de deux types énuméré, un pour les opérateurs unaires et un pour les opérateurs binaires.

Listing 18 – Une seule méthode fait la vérification

```
public static BigDecimalDouble operatorAndFunctionManager
        (WrapperOperatorOrFunction operation, BigDecimalDouble a)
{
    if (a.isNaN)
    {
        return new BigDecimalDouble(Double.NaN);
    }
    if (a.isInfinite)
    {
        double aVal = a.getDoubleInfiniteValue();
        return new BigDecimalDouble(aVal);
    }

    switch (operation)
    {
        // case + operation treatment
    }
}
```

Cette méthode permet aussi d'éviter de faire des erreurs d'inattention, du fait qu'il faut penser, à chaque méthode, à faire la vérification. Cela entraine également moins de copier-coller, ce qui est une bien meilleure manière de programmer.

14. Résultats obtenus

Ce chapitre va présenter les résultats obtenus, à partir des tests pratiques présentés précédemment. On commencera par voir le polynôme de Rump, puis on enchainera avec le calcul de π. Pour finir, on observera les résultats fournis par le modèle de la différentiation automatique.

14.1 Utilisation de Cojac

Cojac agit comme un agent Java qui se lance en spécifiant des options à la machine virtuelle Java. Pour ces tests, des JAR sont créés et lancés ensuite avec Cojac. Pour invoquer un jar on utilise l'option *-jar* de la commande java :

```
$> java -jar myProg.jar
```

Ensuite on spécifie l'agent ainsi que des options :

```
$> java -javaagent:cojac.jar="-I=0.1E-3" -jar myProg.jar
```

ici, on a spécifié que l'on souhaite utilisé l'arithmétique d'intervalles avec comme seuil 10^{-4}.Pour les tests, on utilise les commandes suivantes (fonction BASH) :

```
runCojacInt()
{
    CojacPath='cojac.jar'
    CojacUsePath='COJAC_Use.jar'
    java -javaagent:$CojacPath="-I=0.1E-3" -jar $CojacUsePath
}

runCojacSto()
{
    CojacPath='cojac.jar'
    CojacUsePath='COJAC_Use.jar'
    java -javaagent:$CojacPath="-STO=0.1" -jar $CojacUsePath
}

runCojacAD()
{
    CojacPath='cojac.jar'
    CojacUsePath='COJAC_Use.jar'
    java -javaagent:$CojacPath="-AD" -jar $CojacUsePath
}

runCojacStoN()
{
```

```
    CojacPath='cojac.jar'
    CojacUsePath='COJAC_Use.jar'
    java -javaagent:$CojacPath="-STO=0.1 -STON=$1" -jar $CojacUsePath
}
```

14.2 Le polynôme de Rump

Le polynôme de Rump (voir sous-section 7.2.1) est une fonction relativement connue qui génère des erreurs numériques. Le but du test est de montrer que Cojac parvient à détecter cette erreur numérique, ainsi que de nous fournir la ligne de cette erreur et ce avec les deux modèles. Voici l'implémentation de ce polynôme en Java, le calcul a été étalé afin d'avoir une meilleure vision sur l'erreur. On ajoute également des outputs pour voir ce qui se passe aux endroits critiques.

Listing 19 – Implémentation polynôme de Rump en Java

```
public static double rumpPolynom(double x, double y)
{
    double res = 1335.0;
    res = res * Math.pow(y, 6.0);
    res /= 4.0;
    double tmp = x * x;
    double tmp2 = 11.0 * tmp;
    tmp2 *= y * y;
    tmp2 -= Math.pow(y, 6.0);
    tmp2 -= 121.0 * Math.pow(y, 4.0);
    tmp2 -= 2.0;
    tmp *= tmp2;
    res += tmp;
    tmp = 11.0 * Math.pow(y, 8.0);
    tmp /= 2.0;
    System.out.println("res = " + COJAC_MAGIC_DOUBLE_toStr(res));
    System.out.println("tmp = " + COJAC_MAGIC_DOUBLE_toStr(tmp));
    res += tmp;                                                    // ligne 171
    System.out.println("res = " + COJAC_MAGIC_DOUBLE_toStr(res));
    res += x / 2.0 * y;
    return res;
}
```

14.2.1 Test avec l'arithmétique d'intervalles

A l'exécution, Cojac parvient à détecter une erreur. Grâce à la création d'un objet Exception, on parvient à récupérer la stacktrace de l'exécution du programme. On remarque qu'il y a une erreur à la ligne 171 de notre fonction.

```
--- Test rump Polynom
res = -7.917111340668963E36:[-7.917111340668973E36;-7.917111340668953E36]
tmp = 7.917111340668962E36:[7.917111340668957E36;7.917111340668966E36]
Cojac has destected a unstable operation :
ch.eiafr.cojac.models.wrappers.CojacStabilityException
        at ch.eiafr.cojac.models.wrappers.IntervalDouble.checkStability(IntervalDouble.java:567)
        at ch.eiafr.cojac.models.wrappers.IntervalDouble.dadd(IntervalDouble.java:127)
        at Main.rumpPolynom(Main.java:171)
        at Main.DoubleRump(Main.java:202)
        at Main.runDoubleTest(Main.java:192)
        at Main.main(Main.java:15)
interval value :-1.1805916207174113E21:[-1.534769106932635E22;1.2986507827891526E22]
relative error :2.1818181818181817
```

```
res = -1.1805916207174113E21:[-1.534769106932635E22;1.2986507827891526E22]
Relative error rump : 2.1818181818179663
```

Grâce aux outputs placés avant l'exécution de cette ligne, on remarque que l'on possède deux nombres très proches (res et tmp) et, au moment de l'addition, on a un phénomène d'annulation qui est capté par l'intervalle. On voit bien que la largeur de l'intervalle est immense. L'erreur relative nous indique que le nombre peut être faux à environ 218%.

14.2.2 Test avec l'arithmétique stochastique discrète

A l'exécution, Cojac parvient à détecter une erreur. Grâce à la création d'un objet Exception, on parvient à récupérer la stacktrace de l'exécution du programme. On remarque qu'il y a une erreur à la ligne 171 de notre fonction.

```
--- Test rump Polynom
res = -7.917111340668963E36:[-7.917111340668956E36;-7.917111340668964E36;-7.917111340668959E36]
tmp = 7.917111340668962E36:[7.917111340668961E36;7.91711134066896E36;7.91711134066896E36]
Cojac has destected a unstable operation
Relative error is : 0.9315762275828966
Value : -1.1805916207174113E21:[4.722366482869644E21;-4.722366482869645E21;1.1805916207174116E21]
ch.eiafr.cojac.models.wrappers.CojacStabilityException
    at ch.eiafr.cojac.models.wrappers.StochasticDouble.checkStability(StochasticDouble.java:624)
    at ch.eiafr.cojac.models.wrappers.StochasticDouble.dadd(StochasticDouble.java:100)
    at Main.rumpPolynom(Main.java:171)
    at Main.DoubleRump(Main.java:202)
    at Main.runDoubleTest(Main.java:192)
    at Main.main(Main.java:15)
res = -1.1805916207174113E21 : [4.722366482869644E21;-4.722366482869645E21;1.1805916207174116E21]
Relative error rump : 0.9315762275826812
```

Grâce aux outputs placés avant l'exécution de cette ligne, on remarque que l'on possède deux nombres très proches (res et tmp) et, au moment de l'addition, on a un phénomène d'annulation qui est capté par les nombres stochastiques. Juste après l'exécution de la ligne 171, on peut voir l'output du nombre qui est instable. On voit que les petites différences des nombres dus aux arrondis aléatoires donnent des résultats complètements différents.

14.3 Le calcul de π

Le nombre π peut être calculer via la série 7.2.2, on va donc tenter de déterminer si l'algorithme (voir ci-dessous) utilisé est numériquement stable ou non.

Listing 20 – Implémentation du calcule de π en Java

```java
public static double computePi()
{
    double res = 0.0;
    int numerator = 1;
    double tmp = 4.0;
    boolean alt = true;

    while (tmp > 0.1E-5)
    {
        if (alt)
        {
            res += tmp;
            numerator += 2;
            tmp = 4.0 / numerator;
```

```
            alt = false;
        }
        else
        {
            res -= tmp;
            numerator += 2;
            tmp = 4.0 / numerator;
            alt = true;
        }
    }
    return res;
}
```

Le programme utilisé :

Listing 21 – Programme utilisé pour le test du calcul de π

```
System.out.println("\n--- Test PI");
double pi;
System.out.println(pi = computePiD());
System.out.println(COJAC_MAGIC_DOUBLE_toStr(pi));
System.out.println("Pi relative error : " + COJAC_MAGIC_DOUBLE_relativeError(pi));
```

14.3.1 Test avec l'arithmétique d'intervalles

A l'exécution, Cojac ne trouve pas d'erreur numérique dans l'algorithme. On peut donc dire que l'algorithme est numériquement stable. Voici l'output du programme :

```
--- Test PI
3.141592153589724
3.141592153589724:[3.1415921527015276;3.1415921544779177]
Pi relative error : 5.654426353730959E-10
```

14.3.2 Test avec l'arithmétique stochastique discrète

A l'exécution, Cojac ne trouve pas d'erreur numérique dans l'algorithme. On peut donc dire que l'algorithme est numériquement stable. Voici l'output du programme :

```
--- Test PI
3.141592153589724
3.141592153589724 : [3.141592153367877;3.141592153366625;3.1415921533674536]
Pi relative error : 0.0
```

Dans ce cas, on obtient une erreur relative de 0.0, on voit que les nombres sont extrêmement proches les uns des autres. Le calcule de $C_{\overline{R}}$ donne un nombre de l'ordre de 10^{15} et comme on ne possède que 16 chiffres significatifs, on a un underflow.

14.4 Dérivé de fonction

La différentiation automatique permet de calculer la dérivée de n'importe quelle fonction, au tableau 8.1 il y a toute une série de fonctions qu'il faut implémenter et calculer la dérivée automatiquement pour vérifier qu'elle est correct. Voici un exemple de fonction coder pour la réalisation des tests :

Listing 22 – Exemple d'une fonction Java utilisé pour tester la dérivation de fonction

```
public static void runF1()
{
    System.out.println("Function 1");
    double x = 4.0;

    x = COJAC_MAGIC_DOUBLE_specifieToDerivate(x);
    double res = f1(x);
    double dRes = COJAC_MAGIC_DOUBLE_getDerivation(res);

    System.out.println("f1(x) = " + res);
    System.out.println("f1'(x) = " + COJAC_MAGIC_DOUBLE_getDerivation(res) + " should be " + df1(x));

    if(Math.abs(dRes - df1(x)) < epsilon)
    {
        System.out.println("Test ok");
        nbrTestPassed++;
    }
}
```

Voici l'output du programme de test, la dérivé a été calculée avec succès :

```
Function 1
f1(x) = 256.0
f1'(x) = 192.0 should be 192.0
Test ok
```

Si on lance le programme sans Cojac, le test ne passe pas :

```
Function 1
f1(x) = 256.0
f1'(x) = 256.0 should be 192.0
```

L'ensemble du test est disponible dans les sources du projet, sous le dossier *other*. Lors de l'exécution de tous les programme de test, tous passe.

14.5 Conclusion

Concernant la partie de l'arithmétique d'intervalles et de l'arithmétique stochastique discrète, les résultats sont encourageants mais pas aboutis. On a ici que deux exemples dont un seulement présente une instabilité numérique. Il faudrait trouver d'autres exemples d'algorithme qui sont volontairement instable afin de démontré la viabilité de ces deux modèles.
Le modèle sur la différentiation automatique est un peu plus aboutis que les deux autres. Le calcul de dérivé se passe sans encombre particulière. Par contre, comme pour les deux autres modèles, il n'y a pas d'exemple concret d'application. Les tests utilisés ont l'avantage d'être concluant.

15. Conclusion

Les modèles d'arithmétique d'intervalles et d'arithmétique stochastique discrète sont des modèles reconnus pour la vérification numérique de programmes ou d'algorithme. Mon travail a été de porter ces deux techniques dans le monde Java en passant par le travail d'un étudiant en Master, permettant de remplacer les types primitifs Java par des équivalents objets.

Malheureusement, il n'y a actuellement pas de preuve directe que ce qui a été réalisé dans ce projet est correct ou non. Il y a un manque d'exemple assez contraignant, ce qui limite considérablement la validité actuelle du résultat.

La différentiation automatique permet le calcul automatique de dérivée. Actuellement, dans les autres langages, il y a toujours un travail de mise en place assez désagréable pour permettre l'utilisation de cette technique. En utilisant le travail de M. Monnard, il est possible de s'affranchir complétement de la partie utilisateur du code, et de procéder au calcul des dérivés directement via les wrappers. Il y a certe un travail à effectuer pour pouvoir récupérer la valeur des dérivés, mais cela reste bien minime comparé à d'autres technologies[18].

Le développement de Cojac est en continuel progrès, il y a également d'autres pistes pour améliorer son fonctionnement :

— Implémenter des fonctions mathématiques en utilisant la classe BigDecimal et ainsi s'affranchir complètement de la librairie standard du point de vue mathématiques

— Réaliser une instrumentation partielle du code (basé sur des annotations ?) afin d'améliorer les performances de Cojac ou, sur le même principe, fournir une différentiation automatique de fonction ciblée

J'aimerais remercier sincèrement les personnes suivantes, sans qui la réalisation de ce projet aurait été sûrement impossible :

— Professeur Dr. Frédéric Bapst, pour m'avoir encadré, supporté et être d'une sympathie incroyable à mon égard

— Professeur Dr. Richard Baltensperger, pour les nombreuses réponses à mes questions mathématiques

— Alexis Clément, pour avoir corrigé et relu une partie de ce document

— À mon père, pour son soutien et sa relecture de ce document

— À Steve Nuoffer pour m'avoir soutenu et encouragé dans ce travail

— À toutes les personnes que j'ai oublié de remercier

Fribourg, le 12 mai 2015,

Sylvain Julmy

Contenu du ZIP

Voici l'arborescence du ZIP :

- Coverage : fichiers html générés par l'IDE montrant le résultat du "test coverage" sur la classe *DoubleInterval*.
- Documents : documents en pdf du projet
- Planning : planning du projet
- PV : procès-verbaux du projet
- Ressource : ressource pdf utilisées et garder
- Src : source du projet
- Src-document : source latex des documents
- Tractanda : tractandas du projet

Déclaration d'honneur

Je, soussigné, Sylvain Julmy, déclare sur l'honneur que le travail rendu est le fruit d'un travail personnel. Je certifie ne pas avoir eu recours au plagiat ou à toutes autres formes de fraudes. Toute les sources d'information utilisées et les citations d'auteur ont été clairement mentionnées.

Liste des tableaux

Table des figures

Table des listings

Bibliographie

[1] Thierry Audibert. Les surprenants polynômes de Rump. `http://www.univenligne.fr/Informatique/univInfoPolynomesRump.php`, 2015. [En ligne ; consulté le 27 avril 2015].

[2] Marcilia Campos Bruno Fernandes, Edmo Bezzera. Interval computation with java-xsc. `http://www.cin.ufpe.br/~javaxsc/`, 2005. [En ligne ; consulté le 04 mars 2015].

[3] J.M. Chesneaux. *étude théorique et implémentation en ada de la méthode CESTAC*. 1988.

[4] Boost Community. `http://www.boost.org/`, 2015. [En ligne ; consulté le 28 mars 2015].

[5] Marc Daumas, David Lester, and César Muñoz. Verified Real Number Calculations : A Library for Interval Arithmetic. August 2007.

[6] Sergei I. Zhilin Dmitry Yu. Nadezhin. Jinterval library : Principles, development, and perspectives. `http://interval.louisiana.edu/reliable-computing-journal/volume-19/reliable-computing-19-pp-229-247.pdf`, 2014. [En ligne ; consulté le 03 mars 2015].

[7] Timothy Hickey. Ia-math. `http://interval.sourceforge.net/interval/java/ia_math/README.html`, 2003. [En ligne ; consulté le 03 mars 2015].

[8] R. B. Kearfott. Interval Computations : Introduction, Uses, and Resources. *Euromath Bulletin*, 1(2), April 1994.

[9] Loïc Lamarque. *Modélisation Géométrique et Arithmétique par intervalles*. PhD thesis, Université de Bourgogne, 2006.

[10] Vincent Lefèvre and Paul Zimmermann. Arithmétique flottante. Research Report RR-5105, 2004.

[11] Guillaume Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, École Normale Supérieure de Lyon, 2006.

[12] Romain Monnard. Cojac enriched numbers. Master's thesis, HES-SO, 2014.

[13] Akpémado Montan, Séthy. *On the numerical verification of industrial codes*. Theses, Université Pierre et Marie Curie - Paris VI, October 2013.

[14] Oracle. Java.math. `http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html`, 2014. [En ligne ; consulté le 04 avril 2015].

[15] Oracle. Java language and virtual machine specifications. `http://docs.oracle.com/javase/specs/`, 2015. [En ligne ; consulté le 05 mai 2015].

[16] Doctor Peterson. Les surprenants polynômes de Rump. `http://mathforum.org/library/drmath/view/53775.html`, 2001. [En ligne ; consulté le 05 mai 2015].

[17] Nathalie Revol. Introduction à l'arithmétique par intervalles. Research Report RR-4297, 2001.

[18] uniker9. Jauto-diff. `https://github.com/uniker9/JAutoDiff`, 2014. [En ligne ; consulté le 10 mai 2015].

[19] Wikipédia. Arithmétique d'intervalles. http://fr.wikipedia.org/wiki/Arithm%C3%A9tique_d%27intervalles, 2014. [En ligne ; consulté le 04 mars 2015].

[20] Wikipédia. Interval arithmetic. http://en.wikipedia.org/wiki/Interval_arithmetic, 2014. [En ligne ; consulté le 03 mars 2015].

[21] Wikipédia. Automatic differentiation. http://en.wikipedia.org/wiki/Automatic_differentiation, 2015. [En ligne ; consulté le 21 février 2015].

[22] Wikipédia. IEEE 754. http://fr.wikipedia.org/wiki/IEEE_754, 2015. [En ligne ; consulté le 04 mai 2015].

[23] Wikipédia. Racine carrée. http://fr.wikipedia.org/wiki/Racine_carr%C3%A9e, 2015. [En ligne ; consulté le 12 mai 2015].