ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, le 13 février 2003

Libaa :
Une librairie C++
d’ « Affine Arithmetic »

Olivier Gay <olivier.gay@epfl.ch>

Assistant responsable : Tuan-Viet NGUYEN
Professeur responsable : Prof. Boi FALTINGS

Table des matières

1. Introduction

L'arithmétique des intervalles est une technique d'analyse numérique où chaque quantité est représentée par un intervalle. Ces intervalles peuvent être ensuite ajoutés, soustraits ou multipliés de telles façons que chaque intervalle ***x*** calculé garantit de contenir la valeur inconnue x. C'est un domaine qui a subi une forte progression depuis son introduction dans les années 1960 par Ramon E. Moore. Un journal (« Interval Computations », maintenant « Reliable Computing ») est même dédié aux recherches sur l'arithmétique des intervalles. C'est un domaine encore en plein essor. Périodiquement des conférences sont données sur le sujet et des meetings lui sont consacrée. Plusieurs applications profitent directement des résultats de l'arithmétique des intervalles comme l'analyse numérique, l'infographie, la modélisation géométrique et l'optimisation globale. De plus bien que les algorithmes diffèrent avec des intervalles, la plupart des algorithmes classiques peuvent être adaptés, certains même améliorés, avec l'utilisation d'intervalles.

Cependant, l'arithmétique des intervalles contient certaines faiblesses. Le principal problème est une conservation excessive. Dans certains cas, après un calcul, un intervalle résultant peut être beaucoup trop large, au point quelques fois d'être inutile. Ce phénomène apparaît lorsque la chaîne de calcul est élevée, ce qui n'est pas rare dans des applications pratiques. Toutes les valeurs dans un calcul d'arithmétique des intervalles varient de manière indépendante. Cette hypothèse n'est en fait souvent pas valide d'un point du vue mathématique.

Pour corriger ce problème qui conduit dans plusieurs cas à une explosion de l'erreur, un nouveau model appelé « Affine Arithmetic » a été proposé par Stolfi et Figueirido. Leur modèle garde l'information de la corrélation entre les variables et produit ainsi des intervalles beaucoup plus minces lors de longues chaînes de calculs. Le modèle est toutefois plus complexe et plus coûteux que celui de l'arithmétique des intervalles. Dans ce projet, nous avons implémenté une librairie C++ d' « Affine Arithmetic » et établissons une comparaison (en terme de temps et de précision) des modèles d'Interval Arithmetic et d'Affine Arithmetic.

2. Interval Arithmetic

L'arithmétique des intervalles est une arithmétique définie sur des intervalles, plutôt que sur des nombres réels. Elle apparu d'abord en 1924 mais eût un développement moderne depuis les travaux de Ramon E. Moore en 1962. Ces dernières années, elle connut un essor considérable avec l'augmentation des capacités de calculs des ordinateurs. Ainsi, maintenant de nombreux constructeurs informatiques proposent l'accès à des méthodes liées à l'IA (pour Interval Arithmetic). En IA, chaque quantité x est représentée par un intervalle $x = [\underline{x}, \overline{x}]$ ce qui signifie que $\underline{x} \leq x \leq \overline{x}$.

L'addition et la soustraction sont définies ainsi :

$$x + y = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$$
$$x - y = [\underline{x} - \underline{y}, \overline{x} - \overline{y}]$$

La multiplication et la division se calcule de façon analogue mais il faut prendre compte le signe des bornes. De même, pour les autres fonctions de base (comme racine carrée, exponentiel, sinus…) on définit des fonctions qui prennent un intervalle pour variable et renvoient un intervalle comme valeur.

En arithmétique des intervalles, les nombres sont donc remplacées par des intervalles, les vecteurs par des vecteurs d'intervalles et les matrices par des matrices d'intervalles. L'intérêt est que l'incertitude sur les données est prise en compte. Les calculs effectués sont donc certifiés et le résultat garanti. Le désavantage est qu'on peut avoir des solutions sur-encadrées.

3. Le problème de conservation

Comme on l'a dit, le modèle IA tend à avoir une conservation excessive : certains intervalles calculés sont beaucoup trop larges ce qui rend l'information inutile. Ceci est du au fait que l'IA considère que toutes les variables varient de manière indépendante. Les corrélations entre les variables ne sont pas préservées et lors de longues chaînes de calcul on a assiste à une explosion de l'erreur. Nous donnerons dans cette section quelques exemples de la décorrélation lié au modèle IA et certains effets directement observables.

Soit l'intervalle x, lors du calcul $x-x$ l'arithmétique des intervalles va ainsi considérer deux variables indépendantes. Si nous prenons $x=[2,5]$, $x-x=[-3,3]$: en arithmétique des intervalles $x-x$ n'est pas nul ! Les opérations perdent ainsi certaines de leurs propriétés : - n'est plus l'opposé de + : $[-2,3]+[5,7]=[3,10]$ alors que $[3,10]-[5,7]=[-4,5]\neq[-2,3]$.

Un autre cas de la non-dépendance des variables : soit I un intervalle, $I*I=\{x*y \mid x\in I, y\in I\}$ alors que $I^2=\{x^2 \mid x\in I\}$: la dépendance entre le premier x et le second est perdue. L'intervalle $[-3,2]*[-3,2]=[-6,9]$ diffère de $[-3,2]^2=[0,9]$.

Voyons un exemple classique où la décorrélation des variables apparaît : Soit $x=[4,6]$ que l'on veut évaluer avec la fonction $x(10-x)$. Le calcul en IA nous donne :

$(10-x)=[10,10]-[4,6]=[4,6]$ et $x(10-x)=[4,6]\times[4,6]=[16,36]$

Or une analyse de la fonction nous donne le résultat exact qui est $[24,25]$. L'IA nous renvoie un encadrement 20 fois plus large que le résultat exact. Cette conservation excessive de l'IA s'aggrave particulièrement lors de calculs plus complexes et la précision relative du résultat décroît alors de manière exponentiel jusqu'à fournir un résultat complètement inutile.

Pour palier à cette faiblesse du modèle, lorsque l'IA calcule des intervalles inutilisables parce que trop larges, une solution est de diviser en sous-parties les intervalles des arguments. On calcule la fonction pour chacune des sous-parties et on réunit les intervalles résultants pour avoir un résultat unique. L'augmentation de la division des intervalles de départ a pour effet de donner des intervalles plus minces. Cette technique n'est toutefois pas des plus efficaces car la précision relative de l'IA est généralement indépendante de la largeur de l'intervalle de départ. Ainsi, si la précision relative d'un calcul est trop mauvaise pour être utile d'un facteur 1000, il faudra certainement diviser le domaine en plus de 1000 sous-intervalles pour obtenir un résultat acceptable.

Plusieurs travaux ont tenté de corriger ces faiblesses de l'arithmétique arithmétique : la représentation de polynômes lors d'un calcul par des polynômes de Bernstein nous donne

ainsi de bien meilleurs résultats en terme de précision. Hansen a proposé une amélioration du modèle avec son « arithmétique des intervalles généralisée ». Plus récemment, Stolfi et Comba ont proposé un nouveau modèle, nommé « Affine Arithmetic » dont le but est d'éliminer le problème de conservation présent dans l'arithmétique des intervalles. Le modèle Affine Arithmetic est présenté aux chapitres suivants.

4. Affine Arithmetic

Le modèle Affine Arithmetic (AA) est un nouveau modèle pour le calcul numérique proposé en 1994 par Stolfi et Comba. Comme l'arithmétique des intervalles, l'arithmétique affine peut être utilisée pour manipuler des données imprécises et évaluer des fonctions sur des intervalles. De même, elle garantit les bornes des intervalles, mais contrairement à l'IA, l'arithmétique affine n'omet pas la corrélation entre les quantités d'entrée et les quantités calculées. Le modèle est donc capable de fournir des intervalles plus minces et des résultats plus précis que l'arithmétique des intervalles et l'AA est résistante à l'explosion de l'erreur observée lors de longs calculs.

Dans le modèle Affine Arithmetic, une quantité incertaine x est représenté par une *forme affine*, qui est un polynôme du premier degré :

$$\hat{x} = x_0 + x_1\varepsilon_1 + \ldots + x_m \cdot \varepsilon_m = x_0 + \sum_{i=1}^{m} x_i \cdot \varepsilon_i$$

Les x_i sont les coefficients réels et les ε_i sont des variables symboliques qui varient entre –1 et 1. x_0 est appelé *valeur centrale* de la forme affine $\hat{x}$. Les coefficients x_i constituent les *déviations partielles* et les ε_i, les *symboles de bruit*.

Chaque symbole de bruit représente une source d'incertitude ou d'erreur qui contribue à l'incertitude totale de la quantité $\hat{x}$. L'incertitude peut venir soit des données (l'erreur originale de la mesure) ou du calcul (arrondis, approximations de fonctions). En effet, les fonctions de bases non-affines doivent être approximés dans l'implémentation du modèle AA.

Pour convertir, une forme affine $\hat{x}$ en un intervalle X , il suffit de calculer :

$$X = [x_0 + \xi, x_0 - \xi], \text{ où } \xi = \sum_{i=1}^{m} |x_i|$$

De même, pour convertir un intervalle $[\underline{x}, \overline{x}]$ en une forme affine $\hat{x}$:

$$\hat{x} = x_0 + x_k \varepsilon_k \text{ , avec } x_0 = \frac{\underline{x} + \overline{x}}{2} \text{ et } x_k = \frac{\underline{x} - \overline{x}}{2}$$

Les opérations comme l'addition, la soustraction et la multiplication par une constante se font ainsi :

$$\hat{x} \pm \hat{y} = (x_0 \pm y_0) + \sum_{i=1}^{m} (x_i \pm y_i)\varepsilon_i$$
$$\hat{x} \pm \zeta = (x_0 \pm \zeta) + \sum_{i=1}^{m} x_i \varepsilon_i$$
$$\alpha \times \hat{x} = \alpha x_0 + \sum_{i=1}^{m} \alpha x_i \varepsilon_i$$

où $\hat{x}$ et $\hat{y}$ sont des formes affines et α, ζ des nombres réels.

Les autres fonctions de base (multiplication, la division, etc.) ne sont pour la plupart pas des fonctions affines. La fonction ne peut ainsi pas être représentée comme une combinaison affine des ε_i. Dans ces cas, la fonction non-affine doit être approximée et l'erreur d'approximation stockée dans une nouvelle variable de bruit ε_k.

Pour illustrer la préservation de la dépendance entre les variables par le modèle AA, reprenons l'exemple d'IA, vu au chapitre précédent. On calculait $x(10-x)$ pour un intervalle $X = [4,6]$ Le résultat était obtenu était $[16,36]$ alors que le résultat exact était $[24,25]$. Si nous effectuons le même calcul avec le modèle AA, nous obtenons :

$$\hat{x} = 5 + \varepsilon_1 \text{ et } \hat{x}(10 - \hat{x}) = (5 + \varepsilon_1)(5 - \varepsilon_1) = 25 + 0\varepsilon_1 + \varepsilon_2$$

(Pour une explication sur la multiplication de deux formes affines, voir le chapitre 5.3.1).

Soit un intervalle résultant de $[24,26]$, beaucoup plus proche du résultat exact que le résultat de l'IA et cela grâce à une annulation des corrélations qui sont négatives entre elles..

Le modèle est évidemment plus complexe et plus coûteux que celui de l'arithmétique des intervalles mais selon Stolfi et Comba, il compense le coût du au problème d'explosion de l'erreur de l'IA.

5. Domaines d'application

Nous voyons ici plusieurs domaines d'applications de l'arithmétique des intervalles. Dans certains cas, le modèle AA produit des résultats meilleurs. L'arithmétique des intervalles est utilisée avec succès dans les domaines suivants :

- Optimisation globale. C'est à dire la recherche du minimum et du maximum global d'une fonction. Les méthodes numériques usuelles trouvent des optima locaux (sauf si des propriétés fortes sont vérifiées par la fonction, ex la convexité). La méthode usuelle est d'utiliser un algorithme de type Branch-and-Bound. Par exemple, l'algorithme Branch and Bound d'Ichida-Fuji ou celui d'Hansen.
- Recherche des zéros d'une fonction. Par exemple, en utilisant l'algorithme de Newton avec des intervalles.
- Infographie : calcul d'intersection des surfaces et en Ray-tracing, l'arithmétique des intervalles augmente la rapidité des algorithmes.

Ce sont les domaines où les travaux d'arithmétique des intervalles sont les plus intenses et leur contribution la plus importante. Dans ces trois domaines, le problème de dépendance des variables est apparu et il semble que les résultats produits par le modèle AA soient encore meilleurs.

Il y a de nombreux autres domaines, où elle est utilisée, notamment : en ingénierie éléctrique (équations de réseau AC), en Chimie (pour la résolution de certaines équations

non-linéaires), en Physique (pour les systèmes dynamiques et le Chaos), en Economie (pour vérifier les effets de l'incertitude de certains paramètres d'entrées), etc.

6. Implémentation

6.1 But du projet

Le but du projet était de développer une librairie C++ d'arithmétique affine pour Linux. On demandait aussi d'effectuer une comparaison des performances de l'arithmétique affine avec l'arithmétique des intervalles. Tout au long de chapitre, nous verrons comment a été implémentée cette librairie. Nous verrons ainsi les choix que nous avons faits. Enfin, nous détaillerons brièvement comment installer la librairie puis comment l'utiliser dans un programme C++.

6.2 Structure de donnée

Nous avons développé cette librairie tout en gardant à l'esprit plusieurs objectifs. La librairie devait non seulement être fonctionnelle mais comme il s'agit d'une librairie mathématique nous avons été sensible à implémenter une solution optimisée en terme de calculs. Plusieurs décisions lors de l'implémentation ont été faites dans ce sens. De plus, comme il s'agit d'une librairie, il fallait fournir une interface agréable et adaptée pour le programmeur qui aurait à l'utiliser.

Notre librairie est constituée deux seulement deux classes : la classe AAF et la classe Interval.

Dans le modèle d'arithmétique affine, nous avons besoin à plusieurs endroits d'avoir à manipuler des intervalles. Par exemple, nous aimons bien créer des formes affines à partir d'intervalles et surtout obtenir un résultat sous forme d'intervalle. Plutôt que de faire appel à une autre librairie pour les intervalles, nous avons préféré développer directement notre classe d'intervalle. En effet, l'utilisation d'intervalles pour l'AA reste assez sommaire et il n'était pas nécessaire d'avoir une classe d'intervalle très évolué.

La classe Interval fournit ainsi des méthodes pour créer des intervalles, modifier et extraire leurs bornes. Il y a aussi plusieurs fonctions utilisées en interne par l'AA, parmi celles-ci :

- `Interval::mid()` : Donne le point milieu de l'intervalle.
- `Interval::radius()` : Calcule le rayon (la moitié de la largeur) d'un intervalle

La classe AAF, pour Affine Arithmetic Form, constitue le type des données de forme affine et évidemment le cœur de notre projet. Au départ, nous avions débuté avec un modèle de données où nous stockerions uniquement les coefficients associés aux symboles de bruit. De cette manière l'indice du coefficient dans le tableau indiquait directement de quel symbole de bruit il s'agissait. Le problème est qu'ainsi, à chaque création de nouvelle variable, la taille du tableau était augmentée pour stocker tous les autres coefficients de bruit qui sont nuls. Dans un calcul de dimension n, où n est grand, l'espace gaspillé pour ces variables nulles devient vite conséquent. De plus, si on imagine une simple multiplication de n variables, on effectue à chaque fois un grand nombre d'opérations inutiles (dans ce cas précis de nombreuses multiplications par 0 superflues).

Le modèle que nous avons finalement choisi élimine ces problèmes et optimise les calculs (au prix d'une implémentation plus complexe des opérations élémentaires à deux variables). Voici la partie private de la classe AAF :

```
private:

  static unsigned last;  // highest noise symbol in use

  double cvalue;       // central value vo
  unsigned length;         // lenght of indexes

  // At creation we don't store null coefficients

  double * coefficients; // values of noise sym
  unsigned * indexes;   // indexes of noise sym
```

On utilise deux tableaux. Un pour stocker les déviations partielles de la forme affine et un pour stocker la valeur de l'indice de la variable de bruit associée. Ainsi chaque fois qu'une variable est crée, on ne stocke pas les coefficients nulles associés au variables de bruits

des précédentes variables. On utilise ici des tableaux propre au langage C plutôt que des vector dans un souci de performances afin d'y accéder plus rapidement. Nous utilisons donc une variable supplémentaire (length) pour indiquer la taille des deux tableaux. Notons encore que la valeur centrale x_0 est stockée séparément des déviations partielles x_i.

A chaque création de variable où après chaque opération non-affine, un nouveau symbole de bruit est crée. La variable static last indique l'indice le plus haut des symboles de bruit de tous les objets AAF. Nous voulons en effet que l'utilisateur n'ait pas à se soucier de réserver un nouveau symbole de bruit lorsqu'il crée une variable depuis un intervalle par exemple.

6.3 Les fonctions affines

Les fonctions affines sont celles qui peuvent être exprimées comme une combinaison affine des variables de bruit. Les opérations affines élémentaires sont l'addition, la soustraction et la multiplication par une constante. Nous ne reviendrons pas sur leur définition pour les formes affines ; elle a été donnée au chapitre 3. Comme toutes les fonctions élémentaires, nous les avons implémentées de façon surchargée pour qu'elles puissent être utiliser naturellement par le développeur.L'originalité du code réside pour les fonctions à deux variables : l'addition et la soustraction. A cause du choix de notre modèle nous pouvions pas, pour l'addition par exemple, additionner membre à membre les éléments des tableaux des deux formes affines. Nous avons toutefois réussi à améliorer (c'est-à-dire réduire sa complexité par rapport à la taille des données) plusieurs fois l'algorithme qui effectue les opérations entres les deux formes affines. Nous avons pour cela pu bénéficier que nos tableaux contiennent toujours les indices des variables de bruit de manière triée.

6.4 Les fonctions non-affines

Par opposition aux fonctions affines, les fonctions non-affines sont donc celles qui ne peuvent pas être exprimées comme une combinaison affine des variables de bruit. Ces fonctions doivent donc être approximées par une fonction affine. Une nouvelle variable de bruit doit alors être introduite et contenir l'erreur due à l'approximation. Pour des raisons d'optimisations des calculs, nous avons eu à effectuer quatre types d'approximation

différents selon les fonctions. Dans les sections suivent nous nous intéresserons beaucoup à l'aspect mathématique lié à l'implémentation.

6.4.1 La multiplication de deux formes affines

Soit x et y deux formes affines, nous désirons calculer $z = f(x, y) = xy$. z n'étant pas une forme affine des symboles de bruits, nous chercherons à calculer $\tilde{z}$ la forme affine qui approxime z.

$$\begin{aligned} z &= xy \\ &= z(\varepsilon_1,\ldots,\varepsilon_m) \\ &= (x_0 + \sum_{i=1}^{m} x_i\varepsilon_i)\cdot(y_0 + \sum_{i=1}^{m} y_i\varepsilon_i) \\ &= x_0y_0 + \sum_{i=1}^{m}(x_0y_i + y_0x_i)\varepsilon_i + (\sum_{i=1}^{m} x_i\varepsilon_i)\cdot(\sum_{i=1}^{m} y_i\varepsilon_i) \end{aligned}$$

La valeur z dépend donc de manière quadratique des symboles de bruit. Nous voyons ici facilement que la meilleure approximation affine de $z(\varepsilon_1,\ldots,\varepsilon_m)$, c'est $A(\varepsilon_1,\ldots,\varepsilon_m) = x_0y_0 + \sum_{i=1}^{m}(x_0y_i + y_0x_i)\varepsilon_i$, à laquelle il faut ajouter la meilleure approximation affine de $Q(\varepsilon_1,\ldots,\varepsilon_m) = (\sum_{i=1}^{m} x_i\varepsilon_i)\cdot(\sum_{i=1}^{m} y_i\varepsilon_i) = \sum_{i=1}^{m}\sum_{j=1}^{m} x_iy_j\varepsilon_i\varepsilon_j$. Or on remarque que la fonction Q est paire, $Q(-\varepsilon_1,\ldots,-\varepsilon_m) = Q(\varepsilon_1,\ldots,\varepsilon_m)$, la meilleure approximation affine est donc une fonction constante!

Plus exactement, soit a et b le minimum et le maximum de la fonction sur U^n (le domaine définition des $\varepsilon_1,\ldots,\varepsilon_m$), la meilleure approximation de la fonction affine Q est la constante $\frac{a+b}{2}$ et l'erreur maximale $\frac{b-a}{2}$.

On a ainsi :

$\tilde{z} = A(\varepsilon_1,\ldots,\varepsilon_m) + \frac{a+b}{2} + \frac{b-a}{2}\varepsilon_k$, où ε_k est un nouveau symbole de bruit

Cependant, le calcul des extrema globaux de Q sur U n'est pas trivial. Il nécessite un calcul trop coûteux et n'est pas acceptable pour une opération élémentaire comme la multiplication.

Heureusement, nous ne sommes pas obligé de calculer exactement les bornes. Nous pouvons utiliser trivialement $\overline{Q} = \pm rad(x) \cdot rad(y)$. La fonction $rad()$ a été implémentée dans notre librairie et représente la déviation totale d'une forme affine, c'est à dire la somme des déviations partielles absolues.

$$rad(x) = \sum_{i=1}^{m} |x_i|$$

L'erreur de cette approximation est au plus de quatre fois l'erreur de la meilleure approximation affine. Malgré cela, l'erreur reste quadratique par rapport aux variables x et y et les termes négativement corrélés peuvent toujours s'annuler. Ainsi quand les arguments deviennent plus petits, la multiplication devient dans le modèle AA asymptotiquement plus précise que dans l'IA.

6.4.2 La fonction racine carrée

Pour les fonctions qui suivent nous ne pouvons malheureusement pas user de stratagème comme avec la multiplication en développant le calcul. La fonction racine carrée est une fonction monotone croissante à une variable. Nous cherchons l'approximation affine qui va minimiser le maximum de l'erreur. Nous désirons donc obtenir une fonction et l'erreur maximale. Evidemment tout deux varient en fonction de l'intervalle sélectionné pour l'approximation. Il faut donc extraire les bornes de la forme affine passée en argument en la convertissant en intervalle. Graphiquement, on peut modéliser cela pour un intervalle $[a,b]$ par le parallélogramme (voir figure 1) entre a et b qui contient entièrement la fonction à approximer et minimise son aire.

Chebyshev a développé sur le sujet toute une théorie de l'approximation. Concernant l'approximation affine de fonction à une variable, un théorème de Chebyshev énonce :

Théorème *Si* f *une fonction bornée, deux fois différentiable définie sur un intervalle* $I = [a,b]$ *dont la deuxième dérivée* f'' *ne change pas de signe à l'intérieure de* I *et si* $f^a(x) = \alpha x + \zeta$ *son approximation affine minimax sur* I *. Alors :*

- Le coefficient α *est* $(f(b) - f(a))/(b-a)$, *la pente de la droite* $r(x)$ *qui interpole les points* $(a, f(a))$ et $(b, f(b))$

- Le maximum absolu de l'erreur apparaît deux fois (avec le même signe) aux bornes a *et* b *de l'intervalle, et une fois (avec le signe opposé) au point* u *de* I *où* $f'(u) = a$

- Le terme indépendant ζ *est tel que* $\alpha u + \zeta = (f(u) + r(u))/2$ *et le maximum de l'erreur absolu est* $\delta = |f(u) - r(u)|/2$

Remarque : une fonction $\tilde{f}$ qui minimise le maximum absolu de l'erreur d'une fonction f sur Ω est appelée, dans la théorie de Chebyshev , approximation minimax de f sur Ω.

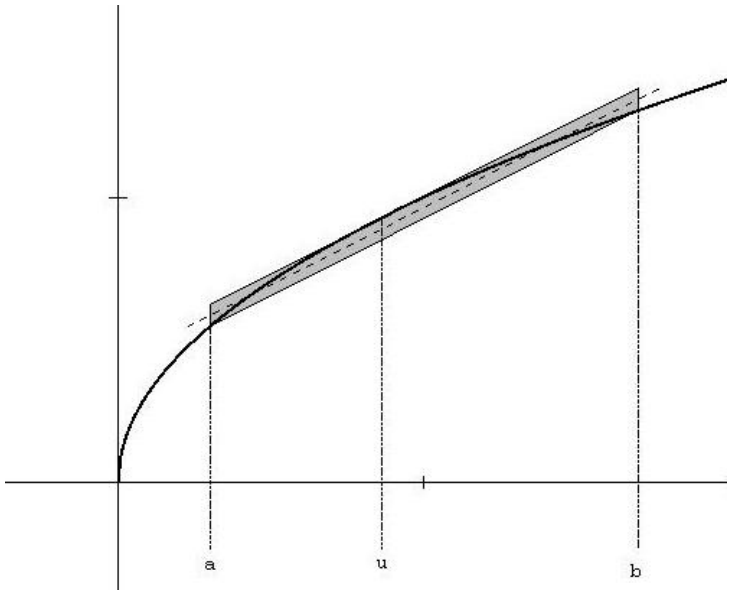

Fig.1 Approximation de Chebyshev

Nous allons utiliser ce théorème pour déterminer la fonction affine qui approxime racine carrée et calculer le maximum de l'erreur. Soit $[a,b]$ l'intervalle de départ.

Nous allons donc chercher une fonction affine $\alpha x + \zeta$. Selon le théorème de Chebyshev on a :

$$\alpha = \frac{\sqrt{b} + \sqrt{a}}{b - a} = \frac{1}{\sqrt{b} + \sqrt{a}}$$

Avec $f'(u) = a = \frac{1}{2\sqrt{u}}$, on obtient $u = \frac{1}{4\alpha^2} = \frac{a + b + 2\sqrt{a}\sqrt{b}}{4}$

De là on déduit que :

$$\zeta = \frac{f(u) + r(u)}{2} - \alpha u = \frac{\sqrt{a} + \sqrt{b}}{8} + \frac{1}{2}\frac{\sqrt{a}\sqrt{b}}{\sqrt{a} + \sqrt{b}}$$

Et l'erreur maximale est :

$$\delta = \frac{f(u) - r(u)}{2} = \frac{1}{8}\frac{(\sqrt{b} - \sqrt{a})^2}{\sqrt{a} + \sqrt{b}}$$

D'où l'approximation de $z = \sqrt{x}$ nous donne :

$\tilde{z} = (\alpha x_0 + \zeta) + \sum_{i=1}^{m} \alpha x_i \varepsilon_i + \delta \varepsilon_k$, avec ε_k un nouveau symbole de bruit

6.4.3 La fonction 1/x

Un des désavantages de l'approximation de Chebyshev, qui a été utilisée dans la section précédente est qu'elle peut être gourmande en temps de calcul pour certaines fonctions. De plus pour certaines fonctions des problèmes d'overshoot/undershoot surgissent quand l'intervalle est plus ou moins large. Par exemple, dans le cas de $1/x$ si on prend un intervalle positif assez large on voit que le parallélogramme produit par Chebyshev et contenant la fonction possède une grande partie de sa surface en dessous de 0, soit dans des y négatifs.

Nous utiliserons ici une méthode d'approximation simplifiée, la méthode « minirange ». On utilise le fait que la fonction est monotone et que sa deuxième dérivée ne change pas de signe. On prend comme pente de la fonction affine qui approxime notre fonction, la pente de la tangente au point le plus bas de notre fonction à approximer (voir figure 2). Pour le calcul le paramètre ζ de la fonction et δ de l'erreur maximale, nous calculons de la même manière que selon Chebyshev. Le parallélogramme obtenu a donc un côté tangent à la courbe à une des bornes de l'intervalle de départ.

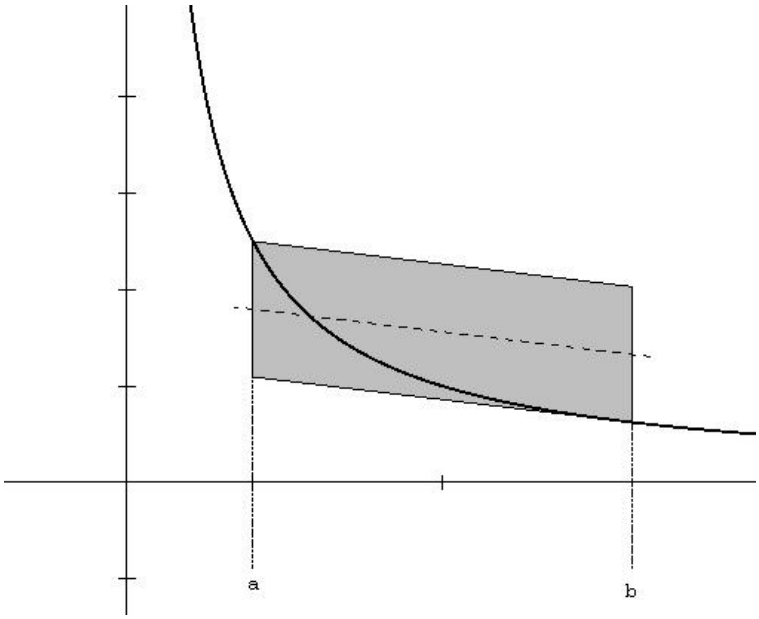

Fig.2 Approximation « minirange »

Voyons comment effectuer cette approximation avec la fonction $1/x$. Tout d'abord, si l'intervalle contient 0, $1/x$ peut prendre des valeurs très grandes et très petites. Regardons le cas où l'intervalle $[a,b]$ est entièrement positif. Nous cherchons donc une fonction affine :

$$\alpha x + \zeta$$

La pente α est donc égale la dérivée de $1/x$ au point b, soit :

$$\alpha = -1/b^2$$

De plus, nous savons que l'autre côté du parallélogramme est parallèle à cette tangente. La fonction affine qui approxime $1/x$ est donc parallèle à ces droites et située à égale distance de celles-ci. Nous pouvons ainsi facilement calculer la valeur du paramètre C. Pour l'erreur maximale, nous savons donc qu'il s'agit de la distance entre notre fonction affine et la tangente que nous avions précédemment calculée. Le cas où l'intervalle est négatif se calcule de manière analogue.

La fonction $1/x$ résout ainsi le problème de la division de deux formes affine x et y, qui est définie comme suit : $x/y = x \cdot 1/y$.

6.4.4 Les fonctions trigonométriques

Pour illustrer le cas des fonctions trigonométrique nous allons voir comment la fonction sinus a été implémentée. Contrairement aux deux fonctions présentées précédemment (racine carrée et $1/x$), la fonction sinus n'est pas monotone, et sa deuxième dérivée change de signe dans un intervalle donnée. Si l'intervalle est supérieur à 2π, il est trivial que la fonction affine qui approxime sinus est la droite $y = 0$ avec une erreur maximale de 1 (le sinus variant entre –1 et 1).

Nous pouvons donc nous concentrer sur les cas où la largeur de l'intervalle de départ est inférieure à 2π. Une des idées et de faire une étude de cas suivant l'intervalle où nous nous trouvons : on regarde le nombre de fois où le graphe change sa croissance et le nombre de points d'inflexion qu'il y a dans l'intervalle. Nous pourrions ensuite appliquer l'algorithme de Chebyshev pour certains intervalles, une version adaptée de l'algorithme de Chebyshev s'il y a un point d'inflexion et pour des autres cas l'algorithme minirange pour éviter l'overshoot et l'undershoot de l'algorithme de Chebyshev en dehors de $[-1,1]$. Malheureusement, avec cette méthode il y a beaucoup de cas à distinguer ce qui entraîne un grand nombre de tests sur l'intervalle. Nous avons jugée cette solution beaucoup trop coûteuse et lui avons préférée la méthode de régression linéaire simple. Avec un faible nombre points, elle est beaucoup plus performante que la méthode précédente.

Nous nous intéressons toujours aux cas où l'intervalle est inférieur à 2π. Le nombre de points pour la régression linéaire simple a été défini dans un DEFINE de notre code et fixé à 8, nous estimons que cela est suffisant. L'intervalle est donc divisé en 8 points.

Le modèle de régression linéaire simple est de la forme suivante :

$$y_i = \alpha + \beta x_i + \varepsilon_i \quad (i = 1,\ldots,n)$$

où $\varepsilon_1,\ldots,\varepsilon_n$ sont des réalisations non corrélées d'une variable aléatoire avec espérance 0 et variance σ^2.

Le principe des moindres carrés est utilisé pour estimer les paramètres du modèle. Nous avons :

$$\hat{\beta} = \frac{y_1(x_1 - \bar{x}) + \ldots + y_n(x_n - \bar{x})}{(x_1 - \bar{x})^2 + \cdots + (x_n - \bar{x})^2}$$
$$\hat{\alpha} = \bar{y} - \hat{\beta}\bar{x}$$

Où $\bar{y} = (y_1 + \ldots + y_n)/n$ et $\bar{x} = (x_1 + \ldots + x_n)/n$

Nous obtenons ainsi les paramètres de la fonction affine qui va approximer sinus dans notre implémentation.

Nous calculons ensuite sur la base de ces paramètres les valeurs ajustées :

$$\hat{y}_i = \hat{\alpha} + \hat{\beta} x_i$$

De là nous pouvons extraire les résidus pour l'erreur :

$$r_i = y_i - \hat{y}_i$$

L'erreur de notre approximation est ainsi calculée comme le maximum absolu des résidus. Une fois que la fonction sinus a été implémentée nous avons pu intégrer simplement les

autres fonctions trigonométriques (cosinus, tangente et cotangente) avec les formules trigonométriques d'usage.

6.5 Quelques notes sur le format floating point

Quand on travaille avec des nombres en floating point, il est essentiel de connaître les subtilités quant à son fonctionnement. Le format a été codifié dans le standard IEEE 754. Des valeurs exceptionnelles font parties du standart et représentent les valeurs non finies:

- +Inf : inifini positif. big*big, num/0.0 avec num > 0.0
- -Inf : inifini négatif. big*(-big), num/0.0 avec num < 0.0
- NaN : Not a Number. 0.0/0.0 , 0*infini, infini-infini

De plus plusieurs exceptions floating point peuvent être interceptées (Overflow, Underflow, Division by zero, Invalid, Inexact).

Il faut savoir que les nombres floating ne sont pas uniformément espacés (voir figure 3). D'abord, un système de floating point ne représente qu'un nombre fini de nombres. On sait qu'un floating point est représenté par une mantisse et un exposant. La mantisse est écrite généralement en base 2 ; certaines représentations décimales sont donc inexactes. Par exemple, 0.1 n'a pas de représentation exacte dans un système de floating point binaire : $10 \times 0.1 \neq 1$.

Fig.3 Non-uniformité de la distribution des floating points

Un autre problème est celui lié à l'arrondi. Comme l'ensemble des floating point est un ensemble discret, les nombres sont arrondis par l'ordinateur avant d'être stockés. Par défaut, l'ordinateur arrondit les nombres au plus proche (TONEAREST) ce qui n'est pas forcément correct suivant les calculs. Il est possible de changer la direction de l'arrondi

vers d'autres modes : par exemple UPWARD ou DOWNWARD. Selon le mode d'arrondi, le résultat peut ainsi changer sur un des digits. Le gros désavantage de modifier la direction de l'arrondi du processeur est que cela est très coûteux : environ une douzaine d'instructions machines. Sur les conseils de l'assistant, pour des questions de performances nous n'avons pas intégré les différents changements de direction de l'arrondi lors des calculs. L'erreur relative d'arrondi est petite (environ 10^{-15} pour des double), cependant nous pensons que cela est quand même important et afin d'avoir un canevas d'utilisation de ces changement de mode en cas de future implémentation, nous avons crée un fichier .h et un fichier .cpp pour les directions d'arrondis. Les changements de directions ont été intégrées dans quelques fonctions de notre classe Interval de telle sorte qu'il n'y ait aucune incidence sur les performances de notre librairie mais afin d'avoir un bon exemple d'utilisation.

6.6 Utilisation et installation de la librairie

Nous voulions que la librairie puisse être installée facilement sur un système. Pour une configuration optimale de l'installation et des paramètres de l'installation par l'utilisateur nous avons décidé d'utiliser les outils autoconf et automake pour créer les fichiers configure et les canevas des fichiers Makefile. De plus pour une compilation portable de la librairie, nous avons utilisé conjointement aux outils précédents le programme libtool. La librairie a été développé à la fois sur Solaris et Linux, et fonctionne sous Linux. Comme nous avons utilisé Emacs pour le développement, les fichiers de configuration pour autoconf, automake et libtool ont été écrits directement à la main (normalement les IDE les génère automatiquement).

Récupération de la librairie :

```
$ wget http://diwww.epfl.ch/~ogay/libaa/libaa-0.9.4.tar.gz
```

Décompression du tarball :

```
$ tar –zxvf libaa-0.9.4.tar.gz
```

Configuration :

```
$ cd libaa-0.9.4/
$ ./configure
```

Compilation :

```
$ make
```

Installation :

```
$ make install
```

Il s'agit de l'installation classique de la librairie. Une version dynamique et une version static de la librairie sont installées. Par défaut, les fichiers de la librairie sont installés dans /usr/local/lib et les fichiers header dans /usr/local/include. Il est possible de préciser un autre chemin d'accès en le précisant avec --prefix à l'appel du fichier ,`./configure`. Pour des configurations particulières,`./configure --help` donne des informations supplémentaires. D'autres informations sont aussi indiquées dans le fichier INSTALL (c'est un fichier générique).

Pour utiliser la librairie dans un programme, il n'y a qu'un fichier header à inclure :

```
#include <aa.h>
```

pour compiler ensuite il suffit de linker le programme avec la librairie dynamique :

```
$ g++ -laa tst.cpp –o tst
```

Ci-dessous le programme example1.cpp du répertoire examples :

```
#include <aa.h>
#include <cstdio>
#include <iostream>


int main()
```

```
{
  Interval u(-2,2);
  Interval v(-1,1);
  AAF x = u;
  AAF r = v;
  AAF s = v;

  AAF temp1 = (10+x+r);
  AAF temp2 = (10-x+s);

  AAF z = (10+x+r)*(10-x+s);

  cout << " x" << endl;
  cout << x;
  cout << " r" << endl;
  cout << r;
  cout << " s" << endl;
  cout << s;;

  cout << " (10+x+r)" << endl;
  cout << temp1;
  cout << " (10-x+s)" << endl;
  cout << temp2;

  cout << " (10+x+r)*(10-x+s)" << endl;
  cout << z;
  cout << z.convert();

  //

  exit(0);

}
```

C’est un programme très simple qui montre comment utiliser la librairie. Les formes affines (AAF) sont généralement crées à partir d’intervalles et on a pas à s’occuper des variables internes. Les fonctions mathématiques utilisées par la librairie sont des fonctions surchargées et ainsi on manipule les formes affines comme on manipulerait n’importe quelle nombre. Dans cet exemple, on crée des formes affines à partir d’intervalles dans le

but d'évaluer une fonction. Ensuite on effectue le calcul et on affiche son résultat sous forme d'intervalle. Tout au long du programme, on affiche également la valeur des variables internes des formes affines pour montrer ce qui se passe.

7. Comparaison avec l'Interval Arithmetic

Dans ce chapitre, nous allons comparer les performances de notre librairie Affine Arithmetic avec celles d'une librairie qui effectue ses calculs avec l'arithmétique des intervalles. Plusieurs tests sont effectués dans ce chapitre qui comparent les temps de calculs des deux modèles et leur précision. Nous avons aussi développé un programme qui affiche à l'écran les courbes avec leurs boxs afin de comparer visuellement les deux modèles sur certaines fonctions données

Pour la comparaison entre les deux modèles, nous avons utilisé la librairie d'arithmétique des intervalles Easyval de Johan Vervloet et Stefan Becuwe. Comme la notre c'est une librairie C++, elle est récente (novembre 2002), sous licence GPL et a été développée par un groupe de recherche en Analyse Numérique de l'Université de Antwerp en Belgique.

Nous avons fait attention à compiler les deux librairies de la même façon et avec les mêmes options de compilation. Notre librairie est compilée avec des options d'optimisation agressives, nous avons donc compilé la librairie Easyval de la même manière. De plus seule la version static de la librairie Easyval est compilée dans le Makefile, nous l'avons donc recompilée pour en avoir une version dynamique.

Il aurait pu être utile aussi d'effectuer une comparaison entre notre implémentation d'AA avec une autre implémentation. Malheureusement, à part la librairie en C, qui est dépassée (1993), des auteurs du modèle AA pour montrer leur théorie, il n'existe à notre connaissance aucune implémentation (publique du moins) du modèle AA !

Les programmes que nous allons voir dans les sections suivantes se trouve dans le répertoire 'examples' du tarball. Ils ne sont pas compilés lors de la compilation de la librairie car ils nécessitent la librairie pour fonctionner. Pour compiler les cinq programmes d'un coup, il faut vérifier que les librairies libaa et libEasyval soient au préalable installées. Ensuite, comme à l'accoutumée : ./configure puis make.

7.1 Affichage via gnuplot

Il y a deux programmes pour afficher des graphiques : example2.cpp et exemple3.cpp. Le premier affiche uniquement la représentation AA, tandis que le deuxième affiche deux graphes avec la représentation AA et la version IA. Il y a deux programmes car le deuxième ajuste les valeurs de l'axe y afin de pouvoir comparer les deux modèles et nous donne ainsi un graphique généralement différent pour le modèle AA.

Les deux programmes génèrent les données pour l'affichage et font appel au programme gnuplot pour l'affichage. Le programme gnuplot a été choisie car il s'agit d'un programme libre et distribué par défaut sur la plupart des UNIX. Gnuplot affichera à l'écran le graphe de la fonction que nous avons choisi d'évaluer sur un intervalle. Selon le nombre de sous divisions que nous avons passés en argument, un certain nombre de boxs sont affichées qui indiquent les intervalles résultants de l'évaluation de la fonction sur les sous divisions.

Le premier programme nous a notamment été utile pour visualiser s'il n'y avait pas des aberrations dues à des bugs dans l'implémentation mais est surtout utile pour illustrer le modèle AA. On voit ici le graphe généré par notre programme et notre librairie pour la fonction $f(x) = (\sin^2(x)\cos(x) - 4)/\sqrt{x}$.

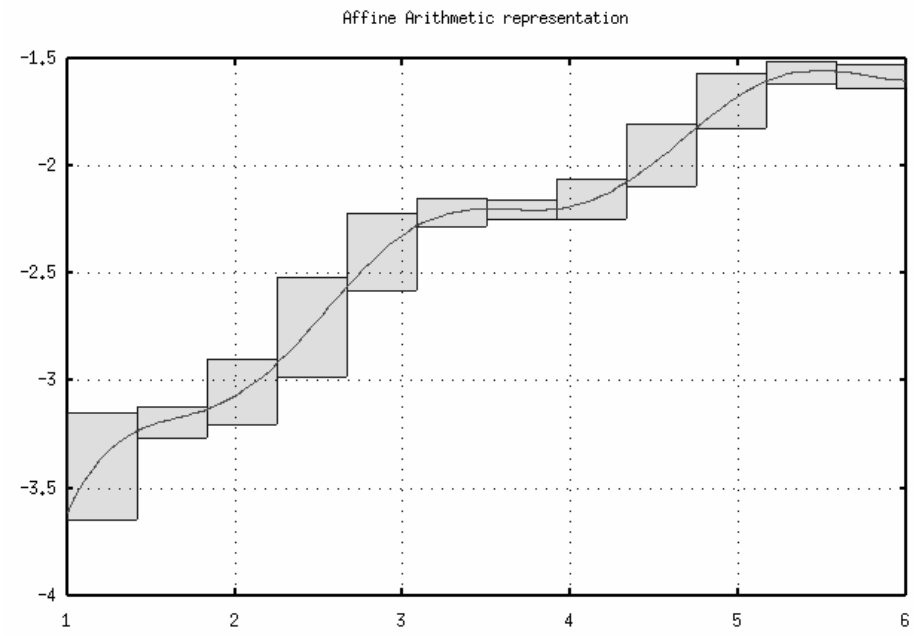

Fig.4 f(x) entre [1,6], 12 intervalles

Les deux programmes étant assez semblables, nous allons surtout nous pencher sur example3.cpp. Comme on l'a dit, pour une fonction donnée (univariée), à partir d'un intervalle de départ et d'un nombre de sous-divisions, il affiche les graphes avec des boxs pour les deux modèles (mode multiplot de gnuplot). L'axe des y est ajusté pour être le même dans les deux modèles. Nous pouvons alors facilement comparer la largeur des intervalles obtenus pour les deux modèles.

Il y a dans le source du programme un template eval_fct() dans lequel on place la fonction à évaluer. Ce template sera utilisé par le programme à la fois pour l'affichage du graphe mais aussi pour les calculs AA et IA.

Une fois le programme compilé pour une fonction donnée, on l'appelle ainsi :

```
./example3 LOWER_BOUND UPPER_BOUND BOXN [function]
```

LOWER_BOUND et UPPER_BOUND désigne l'intervalle de départ et BOXN le nombre de sous divisions. Function est un argument optionnel pour l'affichage d'un nom de fonction avec gnuplot. Le programme va générer trois jeux de données : un pour le pour le graphe, un pour les boxs de l'IA et un pour les boxs de l'AA. Il génère aussi un fichier de commandes pour indiquer à gnuplot comment afficher ces données.

Il suffit ensuite d'appeler ./plot.sh dans un shell pour afficher les graphes via gnuplot. C'est un simple shellscript du répertoire examples qui fait appel à gnuplot. Ainsi qu'il avait été dit dans les premiers chapitres : lors de longues chaîne de calculs, le modèle IA peut fournir des intervalles gigantesques, il peut ainsi arriver que lors de calculs nous tombions sur un intervalle infini. Gnuplot n'est pas capable d'interpréter ces données. Nous contrôlons donc lors du calcul si une valeur non-finie est produite et dans ce cas nous l'indiquons à gnuplot comme une donnée manquante. Si des valeurs non-finies sont rencontrées nous affichons une * à la fin du titre d'un des graphes pour l'indiquer.

Plusieurs statistiques de précision sont aussi affichées sur la console lors de la génération des données par notre programme. Ces statistiques ont fait l'objet d'un autre programme et seront vues au chapitre suivant.

Nous donnons ici un exemple de graphique obtenu avec notre programme qui compare les résultats entre IA et AA. La fonction évaluée est $y = \sqrt{x^2 - x + 1/2} / \sqrt{x^2 + 1/2}$ entre 0 et 5.

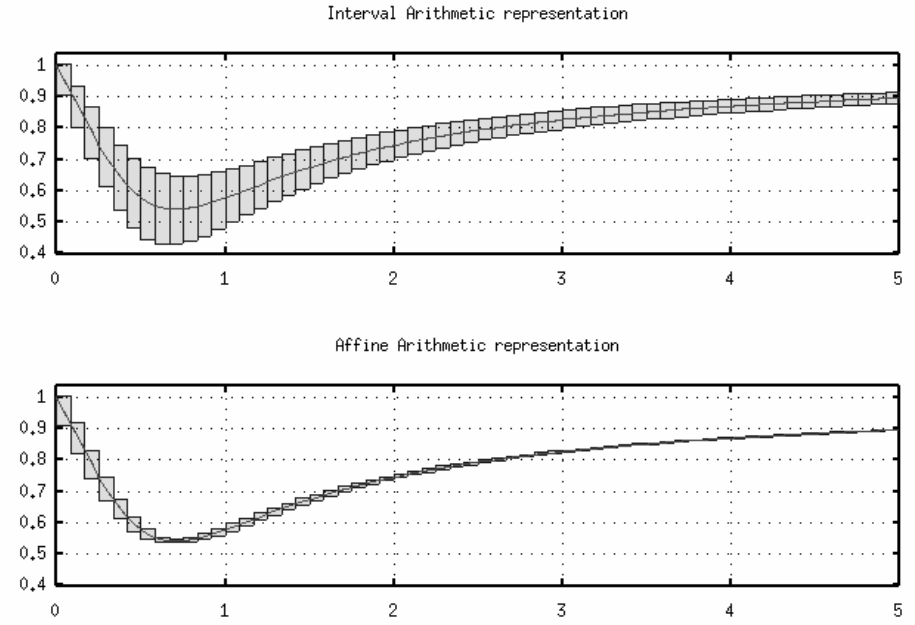

Fig.5 y(x) entre [0,5], 60 intervalles

On voit déjà clairement une différence d'estimation entre les deux modèles favorable à l'AA. Nous allons voir plus bas la même fonction qui s'appelle elle-même, soit $y(y(x))$.

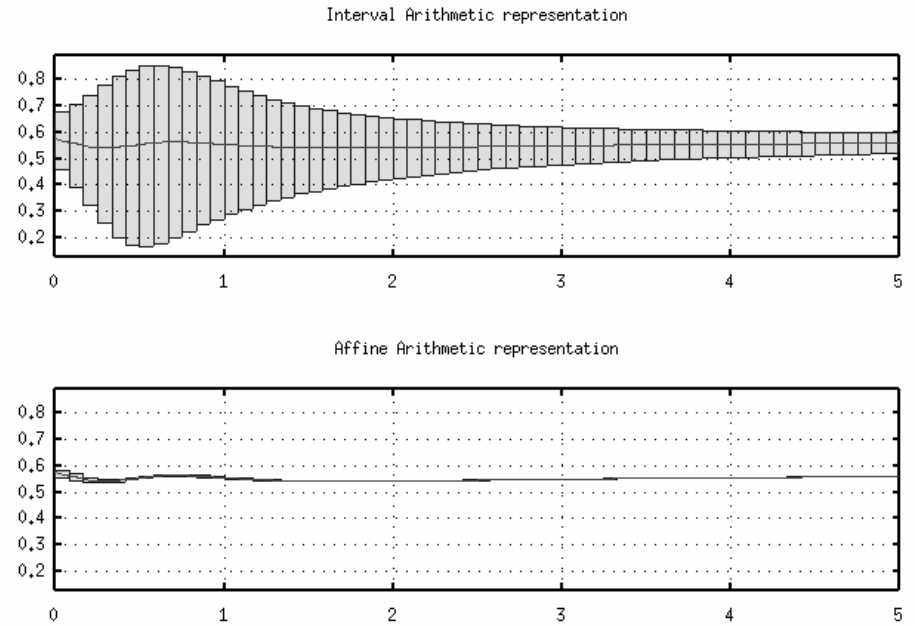

Fig.5 y(y(x)) entre [0,5], 60 intervalles

Si on augmente la chaîne de calcul, la différence devient de plus en plus frappante. Le modèle AA devient plus puissant (en comparaison avec l'IA) à mesure que la complexité de la fonction augmente.

7.2 Précision

Nous avons choisi cinq fonctions sur lesquelles nous allons effectuer nos tests. Les deux premières sont des fonctions à une variable ($y = f(x)$) : ce sont celles représentées dans les figures du chapitre précédents. Les trois autres sont des fonctions quadratiques à deux variables. Pour chacune des fonctions (excepté les deux premières), nous utilisons des intervalles différents ainsi qu'un nombre initial de sous divisions différent. Nous présentons dans cette section des statistiques liées à la précision des deux modèles pour des problèmes similaires. Nous avons pour cela développé un programme (example4.cpp) qui calcule pour nous ces statistiques.

Voici les cinq fonctions :

1. $f_1(x) = \sqrt{x^2 - x + 1/2} / \sqrt{x^2 + 1/2}$, $X = [0,5]$, 60 intervalles
2. $f_2(x) = f_1(f_1(x))$, $X = [0,5]$, 60 intervalles
3. $f_3(x) = x_1^{\,2} + x_2^{\,2} - x_1 x_2$, $X = [0,2] \times [0,1]$, 10 intervalles
4. $f_4(x) = (x_1 - 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$, $X = [-2.5,3.5] \times [-1.5,4.5]$, 100 intervalles
5. $f_4(x) = 1 + (x_1^{\,2} - 2)x_2 + x_1 x_2^{\,2}$, $X = [1,2] \times [-10,10]$, 100 intervalles

Ci-dessous (tableau 1) les statistiques que nous avons extraites. Les champs du tableau ainsi que les résultats sont commentés plus bas.

	IA	AA	AA Width	AA WidthA	IABOX
f_1	[0.427793:1.00692] (0.579128)	[0.536835:1.0044] (0.467565)	80.736%	18.2846%	~750
f_2	[0.161165:0.855952] (0.694786)	[0.535791:0.583995] (0.0482043)	6.938%	1.21414%	~1600
f_3	[-0.03:3.38] (3.41)	[-0.025:3.01] (3.035)	89.0029%	47.8571%	~115
f_4	[93.8816:200.004] (106.122)	[94.2248:198.507] (104.282)	98.2665%	72.6687%	~200
f_5	[0.95:221] (220.05)	[0.966685:221] (220.033)	99.9924%	97.967%	~180

Tab. 1 Précision entre IA et AA

Les colonnes « IA » et « AA » nous donnent l'intervalle obtenu après le calcul par les deux modèles. Le nombre entre parenthèses est la largeur de l'intervalle (la borne supérieure – la borne inférieure). La colonne « AA Width » est une comparaison avec le modèle IA. Soit un intervalle de 100% obtenu pour le modèle IA, quel est la taille de l'intervalle AA en comparaison. Dans la colonne « AA Width A», on effectue le même calcul pour toutes les subdivisions calculées et on donne la moyenne de tous les pourcentages. Ce nombre est donc forcément toujours plus petit que celui obtenu dans la colonne précédente. La dernière colonne sera expliquée plus bas.

Nous aurions bien aimé aussi comparer ces résultats avec les résultats des extrema de la fonction sur l'intervalle de définition, mais nous aurions du alors développer un algorithme Branch and Bound, ce qui dépassait le cadre de ce projet.

Comme prévu, nous obtenons toujours des intervalles plus minces avec notre modèle AA qu'avec celui du modèle IA. On a toutefois des degrés de réussite divers. On voit bien avec la fonction f_2 combien une longue chaîne de calcul donne de meilleurs résultats au modèle AA. Les fonctions ont été choisies au hasard s'en se préoccuper avant des résultats. Les fonctions quadratiques nous donnent ainsi des résultats meilleurs mais moins impressionnants que ceux de fonctions compliquées.

Dans la dernière colonne, nous avons la valeur approximative du nombre de sous divisions que nous aurions du utiliser pour obtenir un résultat semblable que l'AA avec l'IA. Il faut comparer ces chiffres avec le nombre de sous divisions initiales. Cela va de 1.8 x plus (f_5) à 27 x plus (f_2) de sous divisions.

Bien que ces résultats soient meilleurs pour l'AA il faut les mettre en relation avec le temps de calcul effectué pour pouvoir les obtenir. En effet, nous savons que du à la complexité du modèle AA, les calculs sont plus coûteux. Le calcul de la durée d'exécution des deux algorithmes est présenté dans la section suivante.

7.3 Temps d'exécution

Nous avons développé un dernier programme (example5.cpp) qui calcule le temps d'exécution des deux algorithmes pour des paramètres initiaux donnés. Nous avons pour cela utilisé la fonction libc times() qui nous une donne une mesure du temps avec une précision de 10 ms. Cela est suffisant pour nous car nous n'avons qu'à augmenter le nombre de sous divisions du problème pour obtenir des mesures acceptables. Nous savons en effet que pour les deux modèles, la durée d'exécution pour des paramètres initiaux donnés, dépend linéairement du nombre de sous divisions. Ces tests ont été effectués sur un vieux Pentium MMX à 166 MHz, ces chiffres n'ont donc pas de grande valeur du point de vue absolu, ils sont intéressants de manière relative pour comparer les résultats IA et AA.

	IA time	AA time
f_1	0.14 s	0.91 s
f_2	0.27 s	1.83 s
f_3	0.08 s	0.59 s
f_4	0.22 s	1.46 s
f_5	0.11 s	0.75 s

Tab.2 Précision entre IA et AA

Tous ces tests ont été effectués avec une sous division de 5000 intervalles. Nous voyons pour ces cinq fonctions que le modèle AA met entre 6.6 x (f_4) et 7.4 x (f_3) plus de temps que le modèle IA pour arriver à un résultat. Cela n'a rien d'étonnant car le modèle AA est beaucoup plus complexe, notamment à cause du de l'approximation des fonctions non-affines.

Dans la section suivante, ces temps seront mis en relation avec les valeurs de précisions obtenues dans la section précédente.

7.4 Analyse des résultats

Au chapitre précédent, nous avons vu que le modèle AA, pour les fonctions f_1 à f_5 est en moyenne 7x plus lent que le modèle IA. Dans le tableau de la section consacrée aux performances, on peut calculer le ratio des valeurs de la dernière colonne et du nombre initiale de sous divisions, et le comparer avec notre ratio moyen de 7 dans le tableau des temps. Nous constatons alors ceci : malgré que les calculs soient coûteux dans le modèle AA, en terme de précision / temps notre librairie produit de meilleures performances que la librairie Easyval pour les fonctions f_1 , f_2 et f_3.

Les résultats obtenus sont à prendre avec des pincettes. Tout d'abord, ils dépendent énormément de la manière dont les deux modèles ont été implémentés. Si nous avions pris une autre librairie d'IA pour effectuer nos tests nous aurions certainement obtenu des résultats différents. De plus pour le temps, la comparaison relative dépend aussi du contexte (processeur, OS, libc, compilateur…) où les programmes ont été comparé : par

exemple, les deux librairies ne sollicitent pas exactement les mêmes instructions du processeur pour effectuer leur calculs. Mais surtout, et insistons sur ce point là, les résultats dépendent des fonctions que nous avons choisies. Nous avons en effet vu des grandes disparités suivant le choix de la fonction.

En définitive, il semble que le modèle AA soit plus performant lors de longues chaînes de calcul mais moins lors de chaînes moins longues.

8. Conclusion

Nous avons rempli le cahier des charges de ce projet. Nous avons développé une librairie C++ d'Affine Arithmetic et avons effectué une comparaison de performances avec l'Interval Arithmetic. Nous avons ainsi vu que notre libraire Affine Arithmetic est toujours plus précise que l'Interval Arithmetic mais que pour des questions de temps de calcul, elle est plus performante lors de longues chaînes de calcul mais moins performante lors de calculs plus simples. Récemment, des recherches ont été effectuées par l'Université de Pau (France) pour encore améliorer le modèle Affine Arithmetic en proposant de nouvelles formes affines quadratiques (avec des symbole de bruit du premier et deuxième degré). Malgré un modèle de nouveau plus complexe mais encore plus précis, il serait intéressant de vérifier si on obtient une augmentation des performances. Il ne fait nul doute que l'arithmétique des intervalles est un domaine en pleine évolution et que des nouveaux résultats de recherche vont se faire jour dans le futur.

Sur un plan plus personnel, ce projet m'a permis d'apprendre le développement d'un projet et d'apprendre un nouveau langage, le C++. J'ai été aussi été sensibilisé à la théorie mathématique de l'approximation de fonctions, de l'arithmétique des intervalles et de l'optimisation globale. Toutes ces théories m'ont beaucoup intéressées et à mon sens elles regorgent de nouveaux domaines d'exploration passionnants.

9. Bibliographie

1. Self-Validated Numerical Methods and Applications Luiz H. de Figueiredo and Jorge Stolfi

2. Surface intersection using affine arithmetic. L. H. de Figueiredo.

3. Affine arithmetic. Marcus Vinícius A. Andrade, João L. D. Comba, and Jorge Stolfi.

4. Interval Computations: Introduction, Uses, and Resources, R.B. Kearfott

5. Interval and Affine Arithmetic for Surface Location of Power- and Bernstein-form polynomials, Irina Voiculescu, Jakob Berchtold, Adrian Bowyer, Ralph R. Martin, and Qijiang Zhang

6. New Affine Forms in Interval Branch and Bound Algorithms, F. Messine,

7. Functions of intervals, J.C. Burkill

8. Interval Arithmetic and Automatic Error Analysis in Digital Computing, R. E. Moore