Dépliage de Boucles Versus Précision Numérique

Nasrine Damouche, Xavier Thirioux
University of Toulouse,
IRIT, France
nasrine.damouche@irit.fr
xavier.thirioux@enseeiht.fr

Hanane Benmaghnia, Matthieu Martel
University of Perpignan,
LAMPS Laboratory, France
hanane.benmaghnia@etudiant.univ-perp.fr
matthieu.martel@univ-perp.fr

Résumé

Les calculs en nombres flottants sont intensivement utilisés dans divers domaines, notamment les systèmes embarqués critiques. En général, les résultats de ces calculs sont perturbés par les erreurs d'arrondi. Dans un scenario critique, ces erreurs peuvent être accumulées et propagées, générant ainsi des dommages plus ou moins graves sur le plan humain, matériel, financier, etc. Il est donc souhaitable d'obtenir les résultats les plus précis possibles lorsque nous utilisons l'arithmétique flottante. Pour remédier à ce problème, l'outil Salsa [7] permet d'améliorer la précision des calculs en corrigeant partiellement ces erreurs d'arrondi par une transformation automatique et source à source des programmes. La principale contribution de ce travail consiste à analyser, à étudier si l'optimisation par dépliage de boucles améliore plus la précision numérique des calculs dans le programme initial. À cours terme, on souhaite définir un facteur de dépliage de boucles, c'est à dire, trouver quand est-ce qu'il est pertinent de déplier la boucle dans le programme.

1 Introduction

Les progrès rapides et incessants de l'informatique ne cessent d'envahir notre vie quotidienne, allant des simples montres connectées, des smartphones à l'industrie spatiale en passant par les voitures qui sont de plus en plus sophistiquées. Ces systèmes sont dit critiques, car une petite erreur de calcul peut engendrer de graves conséquences sur la vie humaine, la finance et le matériel. Il est à noter que les ordinateurs utilisent des nombres à virgule flottante [1] qui n'ont qu'un nombre fini de chiffres. Autrement dit, l'arithmétique des ordinateurs basée sur les nombres flottants fait qu'une valeur ne peut être représentée exactement en mémoire, ce qui oblige à l'arrondir. Généralement, ces erreurs d'arrondi sont faibles mais dans un scenario critique, elles peuvent être accumulées et propagées, générant ainsi des dégâts considérables allant de l'explosion de fusées à des mauvais calculs de résultats aux jeux olympiques. Par conséquent, il est souhaitable d'obtenir les résultats les plus précis possibles lorsque nous utilisons l'arithmétique flottante. Notons, que dans l'arithmétique à virgule flottante, le parenthésage a un impact majeur sur la correction des résultats de calculs. Par exemple, il vaut mieux en général commencer par additionner les petits nombres flottants entre eux et puis les grands flottants afin d'éviter tout problème liée à l'absorption ou à l'annulation. À titre d'exemple, les deux expressions $(1.0+100.0^{-20})-100.0^{-20}$ et $1.0+(100.0^{-20}-100.0^{-20})$ sont équivalentes dans l'arithmétique des nombres réels cependant dans l'arithmétique des ordinateurs (arithmétique flottante), ces deux expressions retournent des résultats différents. La première formule renvoie 0.0 alors que la deuxième formule donne 1.0. Par conséquent, la fiabilité des calculs dans des contextes critiques impose de vérifier et de valider la précision des traitements numériques [18].

Plusieurs outils destinés à valider et à vérifier les programmes ont été développés comme Fluctuat [14], Astrée [6], etc. En revanche, une difficulté est que l'arithmétique des ordinateurs n'est pas intuitive, et donc détecter les erreurs ne suffit pas, il faut pouvoir les corriger. Des outils de réécriture automatique des expressions arithmétiques ont été mis en œuvre. On peut cité Sardana [16], basé sur une représentation intermédiaire nommée les APEG, et Herbie [22], basé sur une heuristique permettant de générer des tests aléatoires. À la différence, l'outil Salsa [7] prend en charge des programmes plus complets avec des affectations, conditionnelles, boucles, fonctions, etc. [8, 10], écrits dans un langage impératif. Salsa corrige partiellement les erreurs d'arrondi en transformant automatiquement en source à

source des programmes. La transformation des programmes repose sur une analyse statique par interprétation abstraite [5] qui fournit des intervalles pour les variables présentées dans les codes sources. Cette transformation est définie à l'aide de règles formelles appliquées dans un ordre déterministe afin d'optimiser les programmes en temps polynomial. D'un point de vue théorique, le programme généré après transformation ne possède pas forcément la même sémantique que celui de départ mais les programmes source et transformé sont mathématiquement équivalents pour les entrées (intervalles) considérées. De plus, le programme transformé est plus précis. La correction de notre approche repose sur une preuve mathématique par induction comparant le programme transformé avec celui d'origine [9]. Les résultats obtenus par Salsa sont très prometteurs. Nous avons montré, à travers une suite de programmes provenant de systèmes embarqués et de méthodes d'analyse numérique, que nous améliorons significativement la précision numérique des calculs en minimisant l'erreur par rapport à l'arithmétique exacte des réels.

La principale contribution de cet article consiste à étudier si l'optimisation par dépliage de boucles améliore plus la précision numérique des calculs dans le programme original. Ce dépliage de boucles est défini par le fait d'écrire le corps de la boucle plusieurs fois dans l'optique de réduire le nombre d'itérations. Cependant, il est strictement indispensable de s'assurer qu'il n'y ait pas de dépendances entre les instructions. Cette technique est mise en œuvre dans tous les compilateurs. De plus, cette étude nous permettra de définir par la suite un facteur de dépliage de boucles. Plus précisément, trouver quand est-ce qu'il est pertinent de déplier le corps de la boucle dans le programme.

Cet article est organisé comme suit. La section 2 rappelle brièvement la norme IEEE754. La section 3 donne un bref aperçu sur le fonctionnement de Salsa. La section 4, détaille la principale contribution de cet article. La section 5, décrit les différents résultats expérimentaux obtenus avec Salsa. La section 6 résume nos travaux et ouvre sur quelques perspectives.

2 Arithmétique des nombres flottants

2.1 La norme IEEE754

La norme IEEE754 est le standard permettant de spécifier l'arithmétique à virgule flottante [1, 21]. Les nombres réels ne peuvent être représentés exactement en mémoire sur machine. A cause des erreurs d'arrondi apparaissant lors des calculs, la précision des résultats numériques est généralement peu intuitive. Un nombre x en virgule flottante, en base b, est défini par : $x = s \cdot m \cdot b^{e-p+1}$, avec, $s \in \{0,1\}$ le signe, m la mantisse et e l'exposant. Le standard IEEE754 décrit quatre modes d'arrondi pour un nombre x à virgule flottante : vers $+\infty$ ($\uparrow_{+\infty}(x)$), vers $-\infty$ ($\uparrow_{-\infty}(x)$), vers 0 ($\uparrow_0(x)$) et plus près ($\uparrow_\sim(x)$). Il est à noter que nos techniques de transformation ne dépendent pas d'un mode d'arrondi précis.

La sémantique des opérations élémentaires définie par le standard IEEE754 pour les quatre modes d'arrondi $r \in \{-\infty, +\infty, 0, \sim\}$ cités précédemment pour $\uparrow_r$: $\mathbb{R} \to \mathbb{F}$, est donnée par :

$$x \circledast_r y = \uparrow_r (x * y) \quad , \tag{1}$$

avec, $\circledast_r \in \{+,-,\times,\div\}$ une des quatre opérations élémentaires utilisées pour le calcul des nombres flottants en utilisant le mode d'arrondi r et $* \in \{+,-,\times,\div\}$ l'opération exacte (opérations sur les réels). Clairement, les résultats des calculs à base des nombres flottants ne sont pas exacts et ceci est dû aux erreurs d'arrondi. Par ailleurs, on utilise la fonction $\downarrow_r$: $\mathbb{R} \to \mathbb{R}$ permettant de renvoyer l'erreur d'arrondi du nombre en question. Cette fonction est définie par :

$$\downarrow_r (x) = x - \uparrow_r (x) \quad . \tag{2}$$

2.2 Calcul de bornes d'erreur

Pour calculer les erreurs se glissant durant l'évaluation des expressions arithmétiques, nous définissons des valeurs non standard faites d'une paire $(x,\mu) \in \mathbb{F} \times \mathbb{R} = \mathbb{E}$, où la valeur x représente un nombre flottant et μ l'erreur exacte liée à x. Plus précisément, μ est la différence exacte entre la valeur réelle et flottante de x comme défini par l'équation (2). La sémantique concrète des opérations élémentaires dans $\mathbb{E}$ est détaillée dans [17].

La sémantique abstraite associée à $\mathbb{E}$ utilise une paire d'intervalles $(x^\sharp, \mu^\sharp) \in \mathbb{E}^\sharp$, tel que le premier intervalle $x^\sharp$ contient les nombres flottants du programme, et le deuxième intervalle $\mu^\sharp$ contient les erreurs sur $x^\sharp$ obtenues en soustrayant le nombre flottant de la valeur exacte. Cette valeur abstrait un ensemble de valeurs concrètes $\{(x,\mu) : x \in x^\sharp \text{ et } \mu \in \mu^\sharp\}$. Revenons maintenant à la sémantique des expressions arithmétiques dont l'ensemble des valeurs abstraites est noté par $\mathbb{E}^\sharp$. Un intervalle $x^\sharp$ est approché avec un intervalle défini par l'équation (3) qu'on note $\uparrow^\sharp(x^\sharp)$.

$$\uparrow^\sharp([\underline{x},\overline{x}]) = [\uparrow(\underline{x}), \uparrow(\overline{x})] \ . \tag{3}$$

La fonction d'abstraction $\downarrow^\sharp$, quant à elle, abstrait la fonction concrète $\downarrow$, autrement dit, elle permet de sur-approcher l'ensemble des valeurs exactes d'erreur, $\downarrow(x) = x - \uparrow(x)$ de sorte que chaque erreur associée à l'intervalle $x \in [\underline{x},\overline{x}]$ est incluse dans $\downarrow^\sharp([\underline{x},\overline{x}])$. Pour un mode d'arrondi au plus proche, la fonction d'abstraction est donnée par l'équation (4).

$$\downarrow^\sharp([\underline{x},\overline{x}]) = [-y, y] \quad \text{avec} \quad y = \frac{1}{2}\text{ulp}\Big(\max(|\underline{x}|, |\overline{x}|)\Big) \ . \tag{4}$$

En pratique l'ulp(x), qui est une abréviation de *unit in the last place*, représente la valeur du dernier chiffre significatif d'un nombre à virgule flottante x. Formellement, la somme de deux flottants revient à additionner les erreurs générées par l'opérateur avec l'erreur causée par l'arrondi du résultat (voir équation (5)). Similairement pour la soustraction de deux flottants, on soustrait les erreurs sur les opérateurs et on les ajoute aux erreurs apparues au moment de l'arrondi. Quant à la multiplication de deux nombres à virgule flottante, la nouvelle erreur est obtenue par développement de la formule $(x_1^\sharp + \mu_1^\sharp) \times (x_2^\sharp + \mu_2^\sharp)$ (voir équation (6)).

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = (\uparrow^\sharp(x_1^\sharp + x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow^\sharp(x_1^\sharp + x_2^\sharp)) \ , \tag{5}$$

$$(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2^\sharp) = (\uparrow^\sharp(x_1^\sharp \times x_2^\sharp), x_2^\sharp \times \mu_1^\sharp + x_1^\sharp \times \mu_2^\sharp + \mu_1^\sharp \times \mu_2^\sharp + \downarrow^\sharp(x_1^\sharp \times x_2^\sharp)) \ . \tag{6}$$

Notons qu'il existe d'autres domaines abstraits plus efficaces, à titre d'exemple [4, 14, 15], et aussi des techniques complémentaires comme [2, 3, 11, 23]. De plus, on peut faire référence à des méthodes qui transforment, synthétisent ou réparent les expressions arithmétiques basées sur des entiers ou sur la virgule fixe [13]. On citera également [4, 12, 19, 20, 23] qui s'intéressent à améliorer l'intervalle de valeurs des variables à virgule flottante.

3 L'outil Salsa

Dans cette section, nous rappelons brièvement le fonctionnement de l'outil Salsa. Basé sur les méthodes d'analyse statique par interprétation abstraite, Salsa transforme automatiquement des programmes issus de l'arithmétique des flottants pour améliorer leur précision de calculs numériques. Cette transformation concerne les morceaux de code tels que les affectations, conditionnelles, les boucles, les fonctions, etc. Pour réaliser cette transformation, un ensemble de règles de transformations formelles permettant de réécrire les programmes en des programmes plus précis a été défini et implémenté dans Salsa. À titre d'exemples de nos règles de transformations, prenons la conditionnelle $\text{if}_\Phi\ e$ then $c1$ else $c2$. Un premier cas de règle stipule que si on connaît statiquement l'évaluation de la condition e, en d'autre termes, si la condition est toujours vraie (respectivement est toujours fausse), on transforme uniquement la branche `then` soit $c1$ (respectivement la branche `else` soit $c2$), sinon, on transforme les deux branches de la conditionnelle. Salsa prend en entrée un programme initialement écrit dans un langage impératif (langage C) ainsi que des intervalles sur les valeurs des variables de ce programme, et retourne en sortie un programme numériquement plus précis écrit dans le même langage avec des intervalles sur les valeurs des bornes d'erreurs avant et après la transformation de ce programme. En d'autre termes, Salsa minimise l'erreur de calcul de programmes. Il est à noter que pour chaque programme, il faut spécifier la variable de référence à être optimiser par Salsa. Cette variable correspond à la valeur retournée par le programme. Aussi, il est à rappeler que le programme initial et le programme transformé n'ont pas forcément la même sémantique mais mathématiquement sont équivalents. De plus, Salsa réécrits grâce à son analyseur statique les programmes donnés en entrée sous forme SSA pour *Static Single Assignment* afin que chaque variable soit écrite uniquement une seule fois dans le code source, ce qui évite les

confusions causées par la lecture et l'écriture d'une même variable dans le programme. Les différentes règles de transformation sont utilisées dans un ordre déterministe, c'est-à-dire qu'elles sont appliquées l'une après l'autre. La transformation est répétée jusqu'à ce que le programme final ne change plus. Un programme transformé est plus précis que le programme initial si et seulement si :

1. La variable retournée correspond mathématiquement à la même expression mathématique dans les deux programmes.
2. L'erreur de la valeur abstraite de cette variable de référence est plus petite que l'erreur dans la deuxième valeur abstraite.

Pour plus de détail sur le fonctionnement de Salsa, son architecture ainsi les différentes règles de transformations, nous redirigeons le lecteur vers la référence [7].

4 Dépliage de Boucles

La principale contribution de cet article est d'étudier et d'analyser l'impact de dépliages de boucles sur la précision numérique des calculs. En d'autre termes, l'idée consiste à comparer la précision numérique des programmes initiaux dépliés plusieurs fois avec celle correspondante aux programmes transformés avec Salsa pour le même nombre de dépliages. Lors du dépliage de boucles, on ré-écrit le corps des boucles plusieurs fois tout en tenant compte des différentes dépendances. Notons que le plus qu'il en existe des dépendances dans un programme, le plus qu'on a la possibilité d'en construire de larges expressions que nous parserons de différentes manières afin d'en trouver le meilleur programme en terme de précision lors de la transformation. La figure 1 donne un exemple de dépliage de corps de boucle du programme PID. En pratique, déplier le corps de boucles plusieurs fois revient à créer beaucoup de calculs au sein d'un même programme. Lorsqu'on donne ce programme déplié à Salsa, ce dernier le transforme (réécrit) de façon à trouver un meilleur programme en terme de parenthésage minimisant ainsi les erreurs de calculs.

Dans notre cas d'étude, on compare la précision numérique de chaque programme avant et après transformation et ce pour différent nombre de dépliages. Pour ce faire, on a :

— Déplié le corps de boucle de chaque programme plusieurs fois,
— Transformé le programme déplié avec Salsa,
— Déplié par la suite le programme transformé,
— Calculé l'erreur relative pour les deux programmes,
— Mesuré le temps de calcul pour les deux programmes,
— Calculé le gain en terme de précision numérique,
— Comparé les valeurs obtenues.

```
m0 = [0.0,8.0]
%salsa%
double main() {
invdt = 5.0; kp = 9.4514; ki = 0.69006;
kd = 2.8454; eold = 0.0; dt = 0.2;
c = 5.0; i0 = 0.0; t = 0.0;
  while (t < 200.0) {
    e = c - m0;
    p = kp * e;
    i = i0 + ki * dt * e;
    d = kd * invdt * (e - eold);
    r = p + i + d;
    m = m0 + 0.01 * r;
    eold = e;
    i0 = i ;
    t = t + dt ;
  }
 return m;
}
```

(a)

```
m0 = [0.0,8.0]
%salsa%
double main() {
invdt = 5.0; kp = 9.4514; ki = 0.69006; kd = 2.8454;
eold = 0.0; dt = 0.2; c = 5.0; i0 = 0.0; t = 0.0;
  while (t < 100.0) {
    e = c - m0 ;
    p = kp * e ;
    i = i0 + ki * dt * e ;
    d = kd * invdt * (e - eold);
    r = p + i + d;
    m = m0 + 0.01 * r ;
    eold = e ;
    e1 = c - m ;
    p1 = kp * e1 ;
    i1 = i + ki * dt * e1 ;
    d1 = kd * invdt * (e1 - eold);
    r1 = p1 + i1 + d1;
    m1 = m + 0.01 * r1 ;
    eold = e1 ;
    t = t + 2.0 * dt ;
    m0 = m1 ; i0 = i1 ;
  }
 return m1 ;
}
```

(b)

FIGURE 1 – (a) Le programme PID initial. (b) Le programme PID avec un seul dépliage.

Nbr de dépliages	Erreur avant transformation de programme	Erreur après transformation de programme	Temps de transformation	Gain %
1	[0.18982e-13 , **0.52832e-13**]	[0.23766e-13 , **0.44858e-13**]	0.185s	15,09
2	[0.19081e-13 , **0.53116e-13**]	[0.23912e-13 , **0.45065e-13**]	0.419s	15,15
3	[0.19219e-13 , **0.53438e-13**]	[0.24096e-13 , **0.45310e-13**]	0.844s	15,21
4	[0.19395e-13 , **0.53798e-13**]	[0.24318e-13 , **0.45594e-13**]	1.428s	15,24
5	[0.19610e-13 , **0.54196e-13**]	[0.24578e-13 , **0.45915e-13**]	2.121s	15,27

FIGURE 2 – Mesure d'erreur relative avant et après transformation pour un dépliage allant de 1 à 5, gain de précision et temps de transformation du programme PID.

Il est à noter que le dépliage de corps de boucles de chaque programme est effectué à la main et qu'il est effectué de un à 9. Pour des raisons de simplification, les tableaux de la Section 5 montrent les résultats obtenus pour un dépliage allant jusqu'à 5.

5 Résultats expérimentaux

De nombreux tests ont été menés sur plusieurs exemples provenant des systèmes embarqués et des méthodes d'analyse numérique afin d'évaluer l'efficacité de Salsa pour les entrées (intervalles) considérées. Les résultats obtenus, sur une suite de programmes provenant de l'avionique (PID), de robotique (Odometry) ainsi que les méthodes numériques (Runge-Kutta d'ordre 2), sont concluants. Notons que la taille initiale de programme PID et Runge-Kutta d'ordre 4 est sur une vingtaine de lignes de code alors que le programme d'Odometry est sur une soixantaine de lignes de code. Dans la suite de cette section, on calcule le gain en terme de précision numérique pour chacun de ces programmes ainsi que le temps de transformation nécessaire pour chaque dépliage sur chaque programme considéré.

5.1 Contrôleur PID

Le contrôleur PID est un programme très utilisé dans l'avionique. Il permet de maintenir une mesure physique `m` à une certaine valeur qu'on appelle consigne `c`. Le programme du PID initial ainsi celui déplié une seule fois sont donnés par la Figure 1.

Les différentes mesures effectuées sur le programme PID sont données par la figure 2. La première colonne illustre le nombre de dépliages réalisé. Les colonnes 2 et 3 donnent respectivement la valeur de l'erreur relative du programme avant et après transformation avec Salsa. La colonne 4 illustre le temps de calcul nécessaire pour transformer les programmes. La dernière colonne calcule le gain (en pourcentage) obtenu en terme de précision numérique. Les résultats obtenus montrent qu'en dépliant le corps de la boucle de programme PID, la précision est améliorée en moyenne (géométrique) de **15.23%**. Et si on

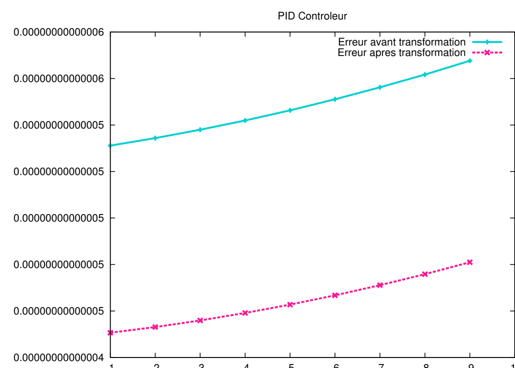

FIGURE 3 – Représentation graphique de l'erreur absolue avant et après transformation du Contrôleur PID pour un dépliage allant de 1 à 9.

Nbr de dépliages	Erreur avant transformation de programme	Erreur après transformation de programme	Temps de transformation	Gain %
1	[$+0.53137e^{-14}$, $\mathbf{+0.53843e^{-14}}$]	[$+0.43109e^{-14}$, $\mathbf{+0.43676e^{-14}}$]	0m26.065s	18,88
2	[$+0.12737e^{-13}$, $\mathbf{+0.13009e^{-13}}$]	[$+0.98486e^{-14}$, $\mathbf{+0.10038e^{-13}}$]	1m53.330s	22,83
3	[$+0.21921e^{-13}$, $\mathbf{+0.22527e^{-13}}$]	[$+0.16526e^{-13}$, $\mathbf{+0.16899e^{-13}}$]	4m53.642s	24,98
4	[$+0.28899e^{-13}$, $\mathbf{+0.29575e^{-13}}$]	[$+0.22501e^{-13}$, $\mathbf{+0.22930e^{-13}}$]	9m34.918s	22,46
5	[$+0.40266e^{-13}$, $\mathbf{+0.41518e^{-13}}$]	[$+0.31046e^{-13}$, $\mathbf{+0.31675e^{-13}}$]	16m49.551s	23,70

FIGURE 4 – Mesure d'erreur relative avant et après transformation, gain de précision et temps de transformation du programme Odometry.

observe le temps de transformation nécessaire pour chaque dépliage, on remarque que le programme PID nécessite 2.121 secondes lorsqu'on le déplie 5 fois. La figure 3 montre que nous améliorons de manière significative la précision numérique en minimisant l'erreur de calcul du programme PID pour un dépliage allant de 1 à 9 fois.

5.2 Odometry

Le deuxième exemple considéré est l'odometry. Ce programme calcule la position instantanée d'un robot à deux roue avec la méthode d'odométrie. Les résultats donné par la Figure 4 obtenus sur le programme Odometry montrent que la précision a été amélioré en moyenne (géométrique) de **23.26%**. Par exemple, l'erreur de calcul pour ce programme passe de $\mathbf{+0.41518e^{-13}}$ à $\mathbf{+0.31675e^{-13}}$ pour un dépliage de 5. Ces résultats illustrent l'efficacité de la transformation du programme déplié. Figure 5 illustre l'amélioration de la précision numérique du programme Odometry pour un dépliage allant de 1 à 6 fois.

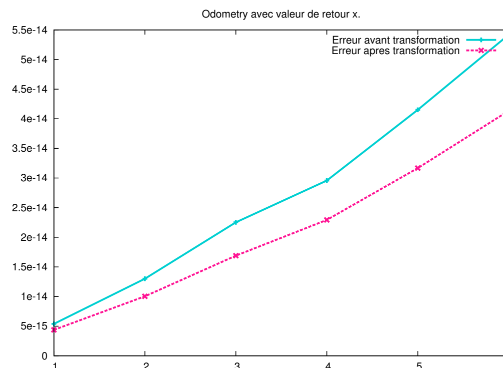

FIGURE 5 – Représentation graphique de l'erreur absolue avant et après transformation du programme Odometry pour un dépliage allant de 1 à 6.

5.3 Runge-Kutta d'ordre 2

La méthode de Runge-Kutta d'ordre 2 (RK2), est une méthode numérique couramment utilisée pour résoudre les équations différentielles ordinaires (EDO).

La figure 6 illustre les résultats obtenus en terme d'erreur relative avant et après chaque transformation pour les différents dépliages ainsi temps de transformation correspondant dans chaque cas pour le programme RK2. Ces résultats montrent que la précision numérique a été améliorée en moyenne (géométrique) de **63.50%**. Si nous observons aussi le temps de transformation nécessaire, nous remarquons que l'erreur de calcul pour ce programme passe de $\mathbf{+0.94748e^{-14}}$ à $\mathbf{+0.27507e^{-14}}$ pour un dépliage de 5. Ces résultats illustrent l'efficacité de la transformation du programme déplié encore une fois. Figure 7 illustre l'amélioration de la précision numérique du programme Runge-Kutta d'ordre 2 pour un dépliage allant de 1 à 9 fois.

Nbr de dépliages	Erreur avant transformation de programme	Erreur après transformation de programme	Temps de transformation	Gain %
1	[+0.53734e-15 , **+0.53734e-15**]	[+0.24091e-15 , **+0.24091e-15**]	0.079s	55,16
2	[+0.14903e-14 , **+0.14903e-14**]	[+0.59159e-15 , **+0.59159e-15**]	0.193s	60,30
3	[+0.30638e-14 , **+0.30638e-14**]	[+0.10899e-14 , **+0.10899e-14**]	0.402s	64,42
4	[+0.55726e-14 , **+0.55726e-14**]	[+0.17867e-14 , **+0.17867e-14**]	0.772s	67,93
5	[+0.94748e-14 , **+0.94748e-14**]	[+0.27507e-14 , **+0.27507e-14**]	1.384s	70,96

FIGURE 6 – Mesure d'erreur relative avant et après transformation, gain de précision et temps de transformation du programme Runge-Kutta d'ordre 2.

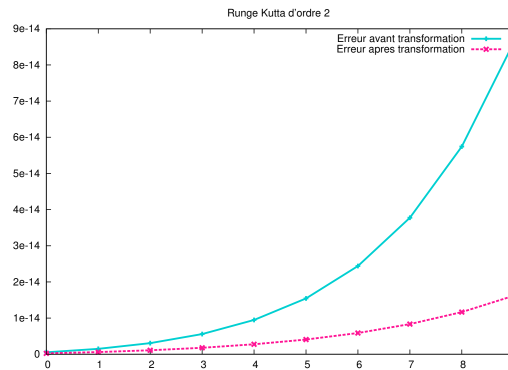

FIGURE 7 – Représentation graphique de l'erreur absolue avant et après transformation du programme Runge-Kutta d'ordre 2 pour un dépliage allant de 1 à 9.

6 Conclusion

L'objectif principal de notre travail est d'étudier l'impact de dépliage de boucles sur la précision numérique des calculs ainsi sur le temps de transformation correspondant. Les résultats obtenus montrent qu'en dépliant le corps de boucles plusieurs fois, la précision numérique des programmes est améliorée. Cette technique est très efficace pour améliorer la précision numérique lorsqu'il existe une forte dépendance entre les instructions de calcul dans le programme.

Une perspective consiste à étendre nos méthodes de transformation automatique de programmes pour améliorer la précision numérique de calculs en dépliant le corps de boucles dans un programme. Autrement dit, nous souhaiterons définir des règles de dépliage de boucles pour transformer les programmes. Un point très ambitieux porte sur les problèmes de reproductibilité des résultats, plus précisément, plusieurs exécutions d'un même programme donne des résultats différents et ce à cause de la variabilité de l'ordre d'exécution des expressions mathématiques.

Références

[1] ANSI/IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. SIAM, 2008.

[2] E-T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *Symposium on Principles of Programming Languages, POPL '13, 2013*, pages 549–560. ACM, 2013.

[3] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *Programming Language Design and Implementation, PLDI '12, 2012*, pages 453–462. ACM, 2012.

[4] J. Bertrane, P. Cousot, R. Cousot, F. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis by abstract interpretation of embedded critical software. *ACM SIGSOFT*, 36(1) :1–8, 2011.

[5] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages, POPL*, pages 238–252, 1977.

[6] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astreé analyzer. In S. Sagiv, editor, *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Proceedings*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.

[7] N. Damouche and M. Martel. Salsa : An automatic tool to improve the numerical accuracy of programs. In B. Dutertre and N. Shankar, editors, *Automated Formal Methods, AFM@NFM 2017, Moffett Field, CA, USA, May 19-20, 2017.*, volume 5 of *Kalpa Publications in Computing*, pages 63–76. EasyChair, 2017.

[8] N. Damouche, M. Martel, and A. Chapoutot. Intra-procedural optimization of the numerical accuracy of programs. In Manuel Núñez and Matthias Güdemann, editors, *Formal Methods for Industrial Critical Systems - 20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings*, volume 9128 of *LNCS*, pages 31–46. Springer, 2015.

[9] N. Damouche, M. Martel, and A. Chapoutot. Improving the numerical accuracy of programs by automatic transformation. *STTT*, 19(4) :427–448, 2017.

[10] N. Damouche, M. Martel, and A. Chapoutot. Numerical accuracy improvement by interprocedural program transformation. In S. Stuijk, editor, *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems, SCOPES*, pages 1–10. ACM, 2017.

[11] E. Darulova and V. Kuncak. Sound compilation of reals. In S. Jagannathan and P. Sewell, editors, *POPL'14*, pages 235–248. ACM, 2014.

[12] J. Feret. Static analysis of digital filters. In David A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004*, volume 2986 of *LNCS*, pages 33–48. Springer, 2004.

[13] X. Gao, S. Bayliss, and G-A. Constantinides. SOAP : structural optimization of arithmetic expressions for high-level synthesis. In *Field-Programmable Technology, FPT*, pages 112–119. IEEE, 2013.

[14] E. Goubault. Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In *Static Analysis Symposium, SAS*, volume 7935 of *LNCS*, pages 1–3. Springer, 2013.

[15] E. Goubault and S. Putot. Static analysis of finite precision computations. In *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *LNCS*. Springer, 2011.

[16] A. Ioualalen and M. Martel. A new abstract domain for the representation of mathematically equivalent expressions. In Antoine Miné and David Schmidt, editors, *Static Analysis - 19th International Symposium, SAS*, volume 7460 of *LNCS*, pages 75–93. Springer, 2012.

[17] M. Martel. Semantics of roundoff error propagation in finite precision calculations. *Higher-Order and Symbolic Computation*, 19(1) :7–30, 2006.

[18] M. Martel. Accurate evaluation of arithmetic expressions (invited talk). *ENTCS*, 287 :3–16, 2012.

[19] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In David A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004.

[20] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.

[21] J-M. Muller, N. Brisebarre, F. De Dinechin, C-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.

[22] J. R. Wilcox P. Panchekha, A. Sanchez-Stern and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *PLDI'15*, pages 1–11. ACM, 2015.

[23] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *FM'15*, volume 9109 of *LNCS*, pages 532–550. Springer, 2015.