Etudier la qualité numérique d'un code avec Verrou

Bruno Lathuilière
{bruno.lathuiliere}@edf.fr

Mise à jour [1] pour verrou v2.7.0 (à venir)

Contexte et objectifs

Dans ce TP, nous nous intéressons à l'étude de la qualité numérique d'un code réalisant le calcul approché de l'intégrale d'une fonction f sur un intervalle $[a, b]$:

$$I = \int_a^b f(x)\ \mathrm{d}x.$$

Numériquement, ce calcul est réalisé à l'aide de la méthode des rectangles. Si n est le nombre de rectangles, on a :

$$I_n = \sum_{i=1}^{n} f(x_i)\ h,$$

où l'on a noté $h = \frac{b-a}{n}$ la largeur des rectangles d'intégration, et si on partitionne $[a, b]$ en n sous-intervalles de longueur h, on note $x_i = a + (i - \frac{1}{2})\, h$ le point central du i-ème sous-intervalle.

On a mathématiquement les résultats de convergence suivants (dans $\mathbb{R}$) :

$$I_n \xrightarrow[n\to\infty]{} I,$$

avec une vitesse de convergence au premier ordre :

$$\varepsilon_n := |I_n - I| = \mathcal{O}\left(\frac{1}{n}\right).$$

C'est cette dernière propriété qui est utilisée ici pour effectuer une vérification du code : si on considère le cas $f = \cos$, $a = 0$ et $b = 1$, on connaît la valeur exacte $I = 1$. Un tracé de ε_n en fonction de n en échelle logarithmique nous permettra donc de vérifier la vitesse de convergence.

Remarque : du point point de vue de la vérification numérique, il est intéressant de remarquer que ce problème est entièrement décrit par des données (a, b) représentables en format flottant, que le résultat exact (I) est lui-même représentable, mais que le calcul fera intervenir des nombres non nécessairement représentables (h, x_i) voire irrationnels ($\cos(x_i)$).

Sur la base de ce cas-test de vérification, nous allons mener l'étude de la qualité numérique du code de calcul de l'intégrale à l'aide de l'outil Verrou [2, 3]. Bien que le code étudié soit ici petit et facilement manipulable, toutes les techniques présentées ici peuvent passer à l'échelle pour de grands codes industriels (au prix parfois d'un peu d'outillage informatique permettant d'automatiser certaines tâches). Une étude a par exemple été réalisée en suivant la même méthodologie d'analyse pour le code de calcul mécanique code_aster [4].

Prérequis

La réalisation de ce TP nécessite des connaissances (au moins) élémentaires en Python, C++ (avec le compilateur Gnu) et Gnuplot.

Par ailleurs, nous supposons dans ce document que Verrou (version $\geqslant$ 2.7.0) a été correctement installé. Si ce n'est pas le cas, merci de vous reporter aux instructions disponibles sur la page GitHub du projet :

`https://github.com/edf-hpc/verrou/tree/v2.7.0`

La documentation de réference de la dernière version stable `https://edf-hpc.github.io/verrou/vr-manual.html` vous sera utile. Si vous préférez utiliser la version master de verrou, pensez à générer la documentation (cf. procédure d'installation).

Code source fourni

Nous décrivons ici l'organisation générale du code source servant de base à la réalisation de ce TP. L'organisation générale des fichiers est indiquée sur la figure 1a :

`work` : répertoire dans lequel le TP se déroule. Il contient notamment les fichiers suivants (les autres fichiers ne sont pas utiles dans un premier temps, et seront décrits par la suite) :

`integrate.hxx` : le code source C++ réalisant le calcul d'intégrale proprement dit.

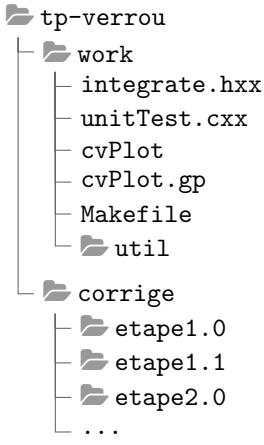
`unitTest.cxx` : le code source C++ réalisant l'étude de convergence permettant de le vérifier. La convergence est testée en réalisant le calcul d'intégrale pour différents nombres de rectangles n variant selon une suite géométrique entre 1 et 100 000. La raison de cette suite géométrique est passée comme argument en ligne de commande du binaire généré (`unitTest`). Pour chaque calcul d'intégrale pour un nombre de rectangles donné, les résultats sont affichés sur 3 colonnes : n, I_n et ε_n. Un exemple d'utilisation du programme est donné sur la figure 1b.

`cvPlot` : un *shell*-script permettant de lancer l'étude de convergence et de générer le graphe correspondant (à l'aide du script gnuplot `cvPlot.gp`). Un exemple de graphe produit est illustré en figure 2.

`Makefile` : permet d'orchestrer la compilation du code étudié, ainsi que l'étude de convergence correspondant à sa vérification.

`util` : contient des scripts utilitaires, qui pourront éventuellement être ré-utilisés dans un contexte autre que celui de ce TP.

`corrige` : contient des sous-répertoires correspondant à certaines étapes clés du processus (correspondant aux sections du présent document). En cas de doute ou de blocage durant la réalisation du TP, il est toujours possible de se référer au corrigé de l'étape en cours afin d'avoir un aperçu de la solution. La structure de chaque sous-répertoire `etapeX.Y` est similaire à celle de `work`. Le `Makefile` fourni permet de réaliser tout ce qui est nécessaire à l'étape correspondante.



(a) Organisation générale du code source de ce TP

Command line

```
$ ./unitTest 10

          1 1.1107207345395915 1.1072073453959153e-01
         10 1.0010288241427083 1.0288241427083289e-03
        100 1.0000102809119049 1.0280911904914092e-05
       1000 1.0000001028083909 1.0280839091159066e-07
      10000 1.0000000010279895 1.0279894713249860e-09
     100000 1.0000000000099656 9.9655839136403301e-12
```

(b) Exemple d'utilisation du programme

Figure 1 – Code source fourni

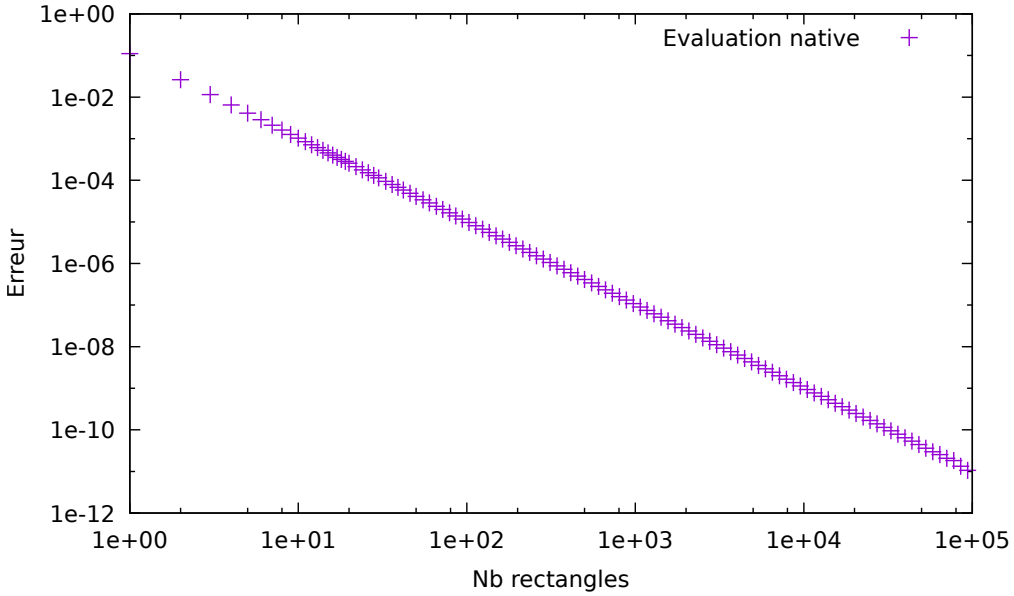


FIGURE 2 – Convergence du calcul d'intégrale

1 Analyse du code en double précision

Le code tel qu'il est fourni dans sa version de base utilise le calcul en double précision. Nous nous proposons dans un premier temps de réaliser l'analyse de ce code, avant de passer dans une deuxième partie à l'analyse (un peu plus délicate) d'une version en simple précision.

Question 1

Familiarisez-vous avec le code fourni.

(a) Étudiez le code source des fonctions `integrate` (dans le fichier `integrate.hxx`) et `testConvergence` (dans le fichier `unitTest.cxx`).

(b) Étudiez le `Makefile` fourni. Compilez le programme `unitTest` et lancez-le manuellement.

(c) Lancez le script `cvPlot` et étudiez la courbe de convergence produite dans `cvPlot.pdf` (vous devriez obtenir une courbe similaire à celle de la figure 2). Est-elle satisfaisante ?

Avant de commencer à utiliser Verrou pour analyser un code, il est préférable de vérifier que les perturbations liées à l'environnement Valgrind lui-même ne sont pas déjà de nature à perturber l'exécution.

Question 2

Vérifiez la reproductibilité des résultats fournis par code, dans tous les cas suivants :

— Mode natif : `./unitTest`
— Valgrind sans outil : `valgrind --tool=none ./unitTest`
— Valgrind/memcheck : `valgrind ./unitTest`
— Verrou sans perturbation : `valgrind --tool=verrou --rounding-mode=nearest ./unitTest`

1.1 Evaluation des erreurs de calcul avec Verrou

On se propose ici d'évaluer la part des erreurs de calcul dans l'erreur globale, en perturbant l'étude de convergence avec les arrondis aléatoires de Verrou.

Question 3

(a) Familiarisez-vous avec Verrou. Par exemple, lancez Verrou sur l'interpréteur Python :

```
valgrind --tool=verrou --rounding-mode=random python3
```

et essayez de réaliser quelques opérations flottantes plusieurs fois.

(b) Expliquez ce qu'il se passe dans chacun des cas suivants (vous devriez obtenir des résultats similaires à ceux présentés ci-dessous) :

(i) `sum((0.1*i for i in range(100)))`

(ii) `sum((0.125*i for i in range(100)))`

(iii) `from math import cos; cos(42.)`

Command line

```
$ valgrind --tool=verrou --rounding-mode=random python3

>>> # cas (i)
>>> sum((0.1*i for i in range(100)))
495.00000000000034
>>> sum((0.1*i for i in range(100)))
494.99999999999983

>>> # cas (ii)
>>> sum((0.125*i for i in range(100)))
618.75
>>> sum((0.125*i for i in range(100)))
618.75


>>> # cas (iii)
>>> from math import cos
>>> cos(42.)
-0.39998531498835127
>>> cos(42.)
-0.39998531498835127
```

Pour plus de détails sur la bibliothèque mathématique, voir partie 4.1 page 10.

Question 4

Utilisation de verrou sur `unitTest` en mode exploratoire.

(a) Faites tourner le `unitTest` en mode natif et stocker le résultat dans un fichier.

(b) Faites tourner le `unitTest` avec verrou en mode random plusieurs fois et stocker les résultats dans des fichiers séparés.

(c) Comparer les résultats avec meld (ou un autre comparateur graphique). La séquence de commande doit ressembler à celle affichée ci-dessous.

(d) Quelles sont les limitations de cette approche ?

Command line

```
$ ./unitTest > out.ref
$ valgrind --tool=verrou --rounding-mode=random ./unitTest > out.random_1
$ valgrind --tool=verrou --rounding-mode=random ./unitTest > out.random_2
$ valgrind --tool=verrou --rounding-mode=random ./unitTest > out.random_3
$ meld out.ref out.random_1
$ meld out.ref out.random_2
$ meld out.ref out.random_3
```

Pour faire face à ces limitations, nous allons calculer pour chaque point de convergence l'estimateur d'erreur flottante calculer par verrou. Le script `util/csv-estimator` fourni avec ce TP permet de les calculer à partir de plusieurs résultats, formattés comme des fichiers textes en colonnes séparées par des espaces. Le script prend en argument une liste de fichiers contenant les résultats à analyser, et affiche sur sa sortie standard les résultats du premier fichier au même format mais en ajoutant des colonnes supplémentaires avec les options `--col-rel` ou `--col-abs` dans la ligne de commande. Ces nouvelles colonnes correspondent à l'estimateur relatif (respectivement absolu) de Bernouilli.

Autrement dit, si le fichier numéro k contient la donnée $x^k_{i,j}$ dans sa i-ème ligne et j-ème colonne, la sortie contiendra à la même place la valeur $x^1_{i,j}$.

En listant des colonnes supplémentaires avec l'option `--col-rel=INDEX` (respectivement `--col-abs=INDEX`) dans la ligne de commande, on ajoute à la sortie une colonne supplémentaire contenant pour chaque ligne i :

$$\hat{s}^{\text{rel}}_i = \frac{\max_k(\left|x^k_{i,j} - x^1_{i,j}\right|)}{\left|x^1_{i,j}\right|}, \qquad \forall i \quad with \quad j = INDEX$$

$$\hat{s}^{\text{abs}}_i = \max_k(\left|x^k_{i,j} - x^1_{i,j}\right|), \qquad \forall i \quad with \quad j = INDEX.$$

Enfin, les estimateurs maximaux (absolu et relatif) sont imprimés sur la sortie d'erreur. S'il dépasse le critère maximum donné par l'utilisateur à l'aide de l'option `--est-max`, le script renvoie une erreur (code de retour : 1).

Dans notre cas, on pourrait par exemple obtenir les sorties suivantes :

Command line

```
$ ./util/csv-estimator --col-abs=2 --est-max=1e-15 cvPlot.dat cvPlot?*.dat

    # (1)                    # (2)                   # (3)                     # (4)
    # input col 1 (first)    # input col 2 (first)   # input col 2 (est abs)   # input col 3 (first)
    1.0000000000000000e+00   1.1107207345395915e+00 7.7969680123367611 4e-17   1.10720734539591581e-01
    1.0000000000000000e+01   1.0010288241427085 5e+00 1.4984583498728077 1e-16   1.02882414270843991e-03
    1.0000000000000000e+02   1.0000102809119060 2e+00 6.9775711326822977 6e-16   1.02809119060243148e-05
    1.0000000000000000e+03   1.0000001028083980 2e+00 4.0782957935980032 6e-15   1.02808397961506870e-07
    1.0000000000000000e+04   1.0000000010282343 9e+00 1.4149110381515172 9e-13   1.02823444203536951e-09
    1.0000000000000000e+05   1.0000000000095785 6e+00 2.2390959112784681 8e-13   9.57850465610476931e-12
    max abs estimator: 2.10476081008437177e-12

$ echo $?
    1
```

dans lesquelles les 4 colonnes représentent :

1. le nombres de rectangles (normalement identiques entre exécutions),
2. les résultats obtenus en natif,
3. **l'erreur flottante estimé par verrou** (c'est cette colonne qui nous intéresse),
4. l'erreur de méthode pour l'exécution native.

Attention : Une erreur classique à éviter est la comparaison de résultats sortis par le code avec trop peu de chiffres significatifs.

Question 5

(a) Modifiez les scripts `cvPlot` et `cvPlot.gp` pour analyser 3 exécutions de l'étude de convergence perturbées avec Verrou, et en déduire une évaluation de l'erreur de calcul. (Si vous n'êtes pas familier avec la syntaxe `gnuplot`, la commande modifiée pour tracer deux courbes est affichée ci-dessous.) On devrait obtenir une courbe similaire à celle présentée sur la figure 3.

cvPlot.gp

```
...
plot "cvPlot.dat"  using 1:3 title "Evaluation native", \
     "cvPlot.stat" using 1:3 title "Estimation verrou"
...
```

A l'issue de cette étape, l'analyse Verrou devrait permettre d'obtenir des résultats similaires à ceux de la figure 3. Ceci permet de confirmer la première impression : les erreurs de calcul sont assez négligeables dans cette gamme d'utilisation du code.

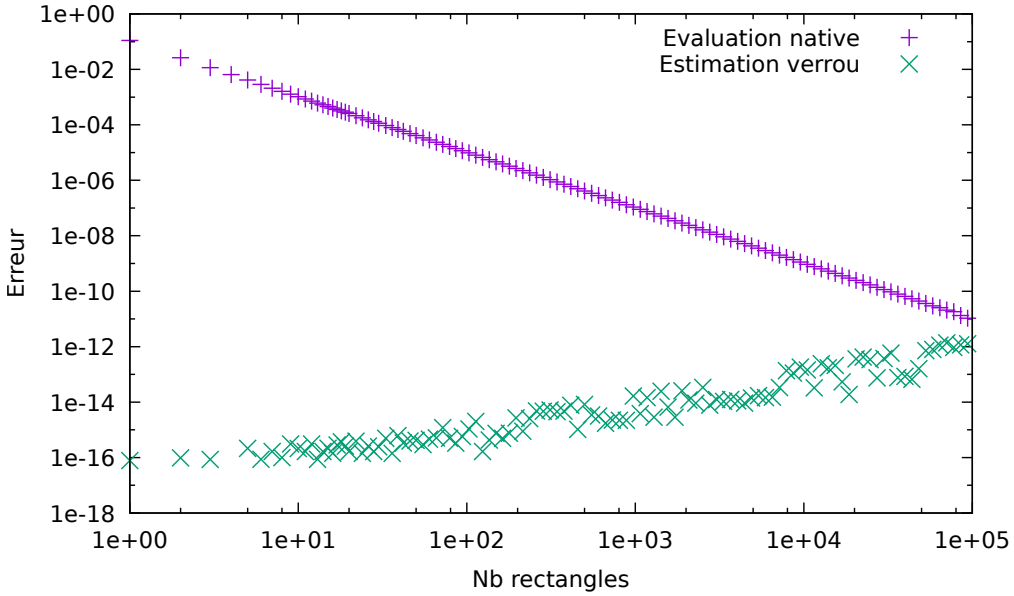


FIGURE 3 – Convergence du calcul d'intégrale et évaluation de l'erreur de calcul avec Verrou

Attention : Ce TP fournit le script ./util/csv-estimator pour être capable de visualiser l'estimateur d'erreur pour un ensemble de points. Ce script dépend du format utilisé par le programme (ici csv). On est parfois amené à écrire des scripts équivalents pour s'adapter à un code industriel. Lorsque qu'on s'intéresse à des variables simples, on peut utiliser directement l'outil `verrou_plot_stat` distribué avec verrou. Ce TP aborde cet outil dans la partie 4.2 (page 11).

2 Analyse du code en simple précision

Nous nous proposons maintenant d'utiliser une arithmétique en simple précision dans notre calcul d'intégration.

Question 6

(a) Modifiez `integrate.hxx` pour basculer en calcul en simple précision.

(b) Relancez l'analyse précédente et interprétez les résultats.

2.1 Déboguage numérique à l'aide du *Delta-Debugging*

Nous nous proposons maintenant de rechercher les instabilités à l'aide d'une approche par bisection, en sélectionnant les portions de code perturbées par Verrou. C'est ce que permet de faire les outils `verrou_dd_sym` et `verrou_dd_line`, livrés avec Verrou. `verrou_dd_line`, (respectivement `verrou_dd_sym`) s'utilise de la manière suivante :

```
verrou_dd_line ./DD_RUN ./DD_CMP
```

Dans la commande ci-dessus, les deux arguments sont des commandes, qui doivent automatiser respectivement le lancement du code avec Verrou, et la comparaison d'un résultat à une référence. Ces deux commandes doivent respecter les prototypes suivants :

`DD_RUN DIR`
: Lance le code à analyser dans Verrou, et range les résultats dans le répertoire `DIR`. Un exemple classique de script de lancement pourrait ressembler à :

ddRun

```
#!/bin/bash

# Récupération des arguments
DIR=$1

# Lancement du code dans Verrou
valgrind --tool=verrou --rounding-mode=random ...

# Rangement des résultats dans le répertoire cible
cp  ...  ${DIR}/...

# Si possible on préfére configurer le code pour écrire
# directement dans le répertoire ${DIR} (pour être réentrant)
```

`DD_CMP REFDIR CURDIR`

Effectue la vérification des résultats contenus dans le répertoire `CURDIR`, potentiellement en les comparant à des résultats "de référence" rangés dans `REFDIR`. Ces résultats "de référence" correspondent à une exécution du code (tel que lancé par `DD_RUN`) non perturbé par Verrou. `DD_CMP` renvoie 0 si et seulement si le résultat contenu dans `CURDIR` est considéré comme valide.

Une trame pour le script de validation pourrait être donnée par :

ddCmp

```
#!/bin/bash

# Récupération des arguments
REFDIR=$1
CURDIR=$2

# Validation des résultats contenus dans ${CURDIR}
# (éventuellement par comparaison à ceux de ${REFDIR})
...
```

ou bien par un script python :

ddCmp.py

```
#!/usr/bin/python3
import sys
import os
def extractValue(rep):
    USER_DEFINED_CODE_TO_PARSE_RESULT

if __name__=="__main__":
    if len(sys.argv)==2:
        #this case is to be compatible with the extract script
        # of verrou_plot_stat and to simplify debug
        print(extractValue(sys.argv[1]))
    if len(sys.argv)==3:
        valueRef=extractValue(sys.argv[1])
        value=extractValue(sys.argv[2])
        relDiff=abs((value-valueRef)/valueRef)

        if relDiff < USER_DEFINED_TOLERANCE:
            sys.exit(0) #OK
        else:
            sys.exit(1) #KO
```

NB : la mise au point de ces deux scripts est l'unique action à réaliser afin d'étudier un code avec les fonctionnalités de *Delta-Debugging* de Verrou. Il est donc important de bien comprendre le comportement attendu de ces scripts, afin de pouvoir les adapter au cas d'intérêt dans un contexte plus complexe que celui de ce TP.

À chaque fois que `verrou_dd_line` cherche à tester une combinaison de fonctions/lignes perturbées, le code est lancé à l'aide de `DD_RUN`, et ses résultats sont analysés à l'aide de `DD_CMP`. Si un échec signifie que la configuration est instable, un succès ne suffit pas pour prouver qu'elle est stable (on pourrait avoir eu de la chance dans les tirages aléatoires). Pour cette raison, une configuration n'est considérée comme stable qu'après un certain nombre d'exécutions validées. Ce nombre est porté à 5 par défaut, mais peut être réglé à l'aide de l'option `--nruns=`.

Durant son exécution, `verrou_dd_line` réalise de nombreux essais, et les résultats correspondants sont rangés dans une arborescence similaire à celle décrite en figure 4, et dont la description détaillée est donnée en annexe A. Dans un premier temps, pour interpréter les résultats du *Delta-Debugging*, il suffit de savoir que les utilitaires `verrou_dd_line` (respectivement `verrou_dd_sym`)créent dans le répertoire `dd.line` (respectivement `dd.sym`) des liens symboliques pour chaque configuration 1-minimal. Pour savoir à quelle ensemble de ligne ou de symboles, correspond à cette configuration il suffit de regarder le fichier d'inclusion et d'exclusion qui s'y trouve.

Pour les codes C++ les noms de symbole n'étant pas facile à lire il peut être nécessaire d'utiliser la commande `c++filt`. Dans la pratique, on utilise rarement cette commande, le résumé founit par les algorithmes de type rddmin, sur la sortie standard en fin d'execution ou dans le fichier `dd.line/rddmin_summary` font ce travail pour vous.

Attention : les fichiers contenus dans les répertoires `dd.line` `dd.sym` servent à contenir les résultats mais aussi de cache pour éviter de refaire plusieurs fois les mêmes essais. Il est donc nécessaire de supprimer ces répertoires avant une exécution du *delta-debugging* afin d'éviter que l'analyse soit perturbée par des scories d'analyses précédentes ou d'utiliser l'option `--cache` pour dire comment ré-utiliser ce cache.

Question 7

Réalisez le *delta-debugging* du code de calcul d'intégrale :

(a) Mettez au point le script `DD_RUN`, permettant d'automatiser le lancement de `unitTest` avec Verrou, et la sauvegarde de ses résultats dans un fichier de votre choix dans le répertoire indiqué en argument.

(b) Dans le langage de votre choix (python ou bash via csv-estimator), mettez au point le script `DD_CMP`, permettant de comparer les résultats de `unitTest`, en cohérence avec le stockage mis en place dans `DD_RUN`. L'analyse de la courbe de convergence pourra permettre de dimensionner de manière adéquate le seuil de tolérance dans la validation des résultats.

(c) Testez vos deux scripts.

(d) Lancez `verrou_dd_line` et analysez les résultats. Vous devriez trouver deux lignes instables.

(e) (optionnel) Étudiez l'impact des deux paramètres-clés de la recherche :
- le seuil de tolérance utilisé par `DD_CMP`,
- le nombre d'exécutions requis pour valider une configuration, fixé par l'option `--nruns=`.

(f) (optionnel) Étudiez l'impact des options de compilation (avec ou sans -g).

(g) Pourquoi est-ce une mauvaise idée d'utiliser `diff` pour le script `DD_CMP` dans ce cas ?

2.2 Détection des branchements instables grâce à `post_verrou_dd`

Le delta-debug permet de détecter l'origine des erreurs dans le code source, mais parfois la correction ne doit pas se faire au niveau de l'origine, mais là où elles sont amplifiées. Un changement du flot d'exécution du code, lorsqu'un branchement dépend d'une valeur perturbée peut constituer une amplification. Nous proposons donc une méthode pour détecter les bracnhements instables.

L'outil `post_verrou_dd` permet de lancer des calculs supplémentaires pour tous les liens symboliques contenus dans un répertoire de résultat de `verrou_dd_line` :
- ddmin ?*
- FullPerturbation
- NoPerturbation
- rddmin-cmp

Les options peuvent permettre, de sélectionner d'autres modes d'arrondis, d'augmenter le nombre d'échantillons, de sélectionner un sous-ensemble d'instructions (par exemple uniquement les additions). Dans ce TP la fonctionnalité qui va nous intéresser est la génération automatique de couverture par Basic Bloc (BB). Un BB est une séquence d'instructions assembleur sans saut (*ie* sans branchement). Les outils de post-traitement de verrou nomment les BB, en fonction de l'ensemble des symboles de debug présent dans le BB. On remarquera que des BB différents peuvent être nommés de manière identique.

Question 8

Détectez les branchements instables :
- Lancez la commande `post_verrou_dd --trace-bin --nruns=10 DD_RUN DD_CMP`.
- Placez vous dans le répertoire `dd.line`
- Comparez les couvertures via les commandes suivantes en jouant sur les paramètres `dd.run`*INDEX* :
  - `meld NoPerturbation-trace/default/dd.run0/cover ddmin0-trace/default/dd.run`*INDEX*`/cover`
  - `meld NoPerturbation-trace/default/dd.run0/cover ddmin1-trace/default/dd.run`*INDEX*`/cover`
- Concluez sur la présence de branchements instables, pour chaque ensemble `ddmin`.

Attention : Pour une compréhension plus fine des approches par couverture, vous pouvez vous référer à la partie 4.4 (page 12).

3 Corrections

3.1 Correction du branchement instable

On se propose ici de corriger une première source d'erreurs de calcul, conduisant au branchement instable détecté dans l'étape précédente : le mécanisme d'itération de la boucle `for` sur les rectangles dans la fonction `integrate`.

Question 9

Correction du test instable.

(a) Corrigez la boucle `for` en introduisant une variable entière pour gérer les itérations.

(b) Vérifiez l'efficacité de votre correction en reprenant l'étude de convergence, et éventuellement en refaisant une détection de branchements instables par couverture de code (ou en vérifiant que le nombre d'opérations est identique entre les différentes exécutions).

3.2 Correction de la sommation

Dans cette dernière étape, on propose de corriger la source d'erreur restante, à savoir l'accumulation d'erreurs d'arrondi dans la sommation par deux approches différentes.

3.2.1 Utilisation de la précision mixte

La première approche (dite *précision mixte*) consiste à utiliser une variable d'accumulation en précision supérieure. Dans notre cas, on utilise un accumulateur en précision double et on conserve toutes les autres opérations en float.

Question 10

Correction de la sommation via l'utilisation de précision mixte.

(a) Implémentez l'accumulateur en double précision.

(b) Vérifiez l'efficacité de votre correction en reprenant l'étude de convergence.

3.2.2 Compensation de la somme

Quand ces problèmes d'accumulation apparaissent sur un code déjà en double précision, augmentez la précision de la variable d'accumulation n'est pas souvent la bonne approche, car l'utilisation de quad (i float128) est très coûteuse. Dans ces cas, on préfère l'utilisation d'algorithmes compensés.

On rappelle ici l'algorithme `FastTwoSum` (alg. 1), qui permet d'effectuer une transformation sans erreur (*Error Free Transformation*, EFT) de l'addition. L'algorithme `FastCompSum` (alg. 2) s'appuie sur cette EFT pour réaliser la compensation d'erreurs sur une somme de n nombres flottants.

NB : La plupart des EFTs et des algorithmes compensés sont prévus pour fonctionner en arrondi au plus près. Il n'est donc pas évident que leur instrumentation en arithmétique stochastique ne pose pas de problème (au même titre que la bibliothèque mathématique par exemple). De récents travaux [5] montrent que certains couples algorithme compensé + EFT sous-jacente continuent à bien se comporter en présence d'arithmétique stochastique ; c'est le cas du choix décrit ici.

Algorithm 1: `FastTwoSum`

Input: (a, b), two floating-point numbers
Result: (c, d), such that $a + b = c + d$

if $|b| > |a|$ **then**
  exchange a and b ;
end
$c \leftarrow a + b$;
$z \leftarrow c - a$;
$d \leftarrow b - z$;

Algorithm 2: `FastCompSum`

Input: $\{p_i, i \in [\![1, n]\!]\}$, n floating-point numbers to sum
Result: $s \simeq \sum_i p_i$

$\pi_1 \leftarrow p_1$;
$\sigma_1 \leftarrow 0$;
for $i = 2 \dots n$ **do**
  $(\pi_i, q_i) \leftarrow$ `FastTwoSum`(π_{i-1}, p_i);
  $\sigma_i \leftarrow \sigma_{i-1} + q_i$;
end
$s \leftarrow \pi_n + q_n$;

Attention : lorsque des options "aggressives" d'optimisation de code sont activées, un compilateur peut facilement conclure qu'une transformation exacte est inutile et la retirer du programme générer. Ceci se produit à partir du niveau `-Ofast` avec `gcc`, mais peut varier d'un compilateur à l'autre, ou même d'une version à l'autre. Pensez toujours à vérifier l'efficacité des EFT introduites dans le code (si besoin en regardant l'assembleur généré). On remarquera que le projet libeft (`https://github.com/ffevotte/libeft`) propose une implémentation pour architecture X86 qui résiste mieux à des options "aggressives" du compilateur.

Question 11

Réalisez la compensation d'erreur dans la sommation des contributions de chaque rectangle :

(a) Revenez en arrière avec un accumulateur en simple précision

(b) Introduire l'algorithme `FastCompSum` dans la fonction `integrate`.

(c) Tester l'efficacité des corrections apportées en relançant l'analyse de convergence ainsi que l'évaluation des erreurs de calcul à l'aide de Verrou.

(d) (optionnel) Tester l'impact des niveaux d'optimisation du compilateur sur la qualité des résultats produits.

4 Quelques points particuliers

4.1 La bibliothèque mathématique

On constate généralement que certains algorithmes de la bibliothèque mathématique (`libm`) sont incompatibles avec les modes d'arrondi autres que *nearest*. Il est donc recommandé en première approche d'éviter de perturber les opérations internes à `libm`, ce qui est fait par défaut. Si la détection automatique de la bibliothèque mathématique échoue [1], on peut être amené à l'exclure manuellement. Dans ce TP, on va déactiver volontairement la détection de la bibliothèque mathématique via l'option `--libm=manual_exclude`, pour montrer la solution de contournement.

Question 12

Observation du problème :

(a) Dans un interpréteur python instrumenté en mode random, en déactivant la détection de la bibliothèque mathématique (`--libm=manual_exclude`), appelez plusieurs fois cos(42).

(b) Lancez `unitTest` (dans sa version en double précision) dans Verrou en mode arrondi aléatoire, en déactivant la détection de la bibliothèque mathématique.

Pour résoudre le problème on peut fournir à Verrou une liste d'exclusion contenant les fonctions à ne pas instrumenter, avec la commande :

```
valgrind --tool=verrou --rounding-mode=random --exclude=LISTE.EX PROG [ARGS]
```

Les fonctions sont listées dans le fichier fourni (`LISTE.EX` dans notre exemple) avec un format en deux colonnes séparées par des blancs :

1. Par exemple, quand la bibliothèque mathématique est linkée statiquement

(i) nom de symbole (correspondant au nom de la fonction, modulo le *mangling* éventuel),

(ii) nom d'objet (*chemin absolu canonique* de la bibliothèque dynamique ou du binaire exécutable).

Le contenu de chacune de ces colonnes peut être remplacée par une astérisque '*', qui permet d'omettre ce critère dans l'identification des fonctions à exclure. Voici un exemple de liste d'exclusion :

libm.ex

```
__cos_avx           /lib/x86_64-linux-gnu/libm-2.23.so
sloww               /lib/x86_64-linux-gnu/libm-2.23.so
do_sin_slow.isra.3  /lib/x86_64-linux-gnu/libm-2.23.so
sloww1              /lib/x86_64-linux-gnu/libm-2.23.so
do_sin.isra.2       /lib/x86_64-linux-gnu/libm-2.23.so
__dubsin            /lib/x86_64-linux-gnu/libm-2.23.so
```

Pour exclure l'ensemble des fonctions de la `libm`, on peut construire une telle liste à la main facilement à l'aide d'une astérisque sur les noms de symboles. Il faudra cependant faire attention à mettre le bon chemin de bibliothèque (retrouvé à l'aide de `ldd` et canonisé avec `readlink -f`). Le script `util/verrou-exclude` fourni avec ce TP permet d'automatiser cette étape. Par exemple, dans le cadre de ce TP, on peut produire une liste d'exclusion pour la `libm` à l'aide de la commande suivante :

Command line

```
$ ./util/verrou-exclude  unitTest  libm.so  | tee libm.ex
    *    /lib/x86_64-linux-gnu/libm-2.23.so
```

Dans le cas, dans le cas où la bibliothèque mathématique est linkée statiquement, cette approche ne fonctionne pas. Dans ce cas il faut générer une liste d'exclusion contenant tous les symboles effectuant des opérations flottantes (option `--gen-exclude`) et sélectionner manuellement tous les symboles issues de la bibliothèque mathématique.

Question 13

(i) Générer manuellement une liste exclusion pour la bibliothèque mathématique.

(ii) Vérifier qu'elle est correcte, en faisant l'analyse de convergence, avec les options `--exclude=LISTE.EX --libm=manual_exclude`

En évitant de perturber les opérations internes à la `libm`, on ignore les erreurs d'arrondis issues de la bibliothèque mathématique. Lorsque la détection automatique de la bibliothèque mathématique fonctionne, il est possible d'utiliser une bibliothèque mathématique fournie avec verrou qui implémente tous les modes d'arrondi de verrou. Pour activer cette fonctionnalité, il suffit d'utiliser l'option `--libm=instrumented`.

Question 14

(i) Faites l'analyse de convergence, avec l'option `--libm=instrumented` (en double precision et en simple précision)

(ii) Vérifiez que l'instrumentation est effective en observant les compteurs.

(iii) Concluez sur l'influence de la bibliothèque mathématique sur cette application.

4.2 Visualisation d'histogramme avec `verrou_plot_stat`

L'outil `verrou_plot_stat` permet de visualiser les distributions de résultats issues des modes stochastiques, et les résultats obtenues avec les modes d'arrondis déterministes. L'outil permet également de calculer automatiquement l'estimateur d'erreur. On notera que l'outil est très adapté lorsque l'on travaille avec des données ponctuelles [2].

`verrou_plot_stat` s'utilise de la manière suivante :

```
verrou_plot_stat --nruns=50 ./PS_RUN ./PS_EXTRACT
```

Dans la commande ci-dessus, les deux arguments sont des commandes, qui doivent automatiser respectivement le lancement du code dans Verrou, et la comparaison d'un résultat à une référence. Ces deux commandes doivent respecter les prototypes suivants :

2. Il est possible de l'utiliser pour des données formatées sous forme de tableau 1D (Hors scope du TP).

`PS_RUN DIR`
Lance le code à analyser dans Verrou, et range les résultats dans le répertoire `DIR`. Il respecte le même format que `DD_RUN`.

`PS_EXTRACT CURDIR`
Affiche sur la sortie standard (première ligne) le résultat contenus dans le répertoire `CURDIR`. Si vous avez pris le squelette python pour le script DD_CMP du delta-debug, DD_CMP est un script valide.

Question 15

(a) Revenir sur une version en float sans correctif.

(b) Ecrire/récupérer les scripts `PS_RUN` et `PS_EXTRACT` (`PS_EXTRACT` doit afficher la dernière valeur de la convergence).

(c) Comparer les distributions random et average ainsi que les résultats déterministes nearest, upward, downward et farthest avec `verrou_plot_stat`.

(d) Que se passe-t-il ? Pourquoi ?

(e) Modifier `PS_EXTRACT` pour récupérer l'avant dernière valeur de l'analyse de convergence. Lancer `verrou_plot_stat` sans nettoyer le cache.

(f) Sur ce cas test, était-il malin de faire un delta-debug en mode random ?

4.3 Précision mixte

Avec verrou, il est possible de convertir les opérations en double précision en opérations en simple précision grâce à l'option `--float`. On notera que cette option est compatible avec tous les modes d'arrondis définis par l'option `--rounding-mode=`.

Question 16

(a) Reproduire la première question de la partie 2 à partir du binaire en double précision, en utilisant conjointement les options `--float` et `--rounding-mode=random`.

Pour la partie recherche de configuration mixte précision avec le delta-debug, il est possible de ne pas utiliser un mode stochastique.

Question 17

(a) Faites une recherche delta-debug sur le code en double précision avec uniquement l'option `--float`. Pensez à adapter le nombre d'échantillons de `verrou_dd_line`.

(b) Comparez cette approche, pour obtenir des configurations mixtes précision satisfaisant un critère de qualité, avec celle illustrée dans la partie 2 du TP.

4.4 Détection des branchements instables par couverture de code

Afin d'identifier les branchements instables dans le code, on peut combiner Verrou avec un outil d'analyse de la couverture de code. Pour les compilateurs `gcc`, l'outil de référence est `gcov`, dont l'utilisation peut être résumée comme suit :

1. Recompiler le code en passant à `gcc` l'option supplémentaire "`-coverage`". Ceci génère un fichier `.gcno` correspondant à la partie statique de l'instrumentation.
2. Lancer l'exécution du code (potentiellement avec Verrou). Ceci génère, en plus des résultats habituels, un fichier `.gcda` contenant les résultats de couverture du code. Plusieurs exécutions du programme viendront accumuler leurs résultats dans le même fichier `.gcda`.
3. Extraire les résultats de couverture sous forme lisible en lançant la commande `gcov` sur tous les fichiers sources. Ceci produit un ensemble de fichiers `.gcov` contenant le code source du programme, annoté avec des indications du nombre de passages dans chaque ligne.

NB : bien que la mise en place de la couverture de code soit relativement simple dans l'exemple du TP, il peut s'agir d'un travail conséquent pour un code industriel. Par ailleurs, la procédure décrite ci-dessus est adaptée à l'utilisation des compilateurs Gnu. Le principe reste le même pour la suite de compilation Intel, mais les commandes précises doivent être adaptées.

Ce mécanisme peut être utilisé pour détecter les branchements instables de la manière suivante :

1. Réaliser un test standard de couverture de code. Stocker les fichiers `.gcov` générés dans un répertoire.
2. Effacer le fichier `.gcda` pour éviter l'accumulation de résultats.
3. Réaliser un deuxième test de couverture de code, dans exactement les mêmes conditions mais en perturbant l'arithmétique avec Verrou. Stocker les fichiers `.gcov` dans un deuxième répertoire.
4. Comparer les deux répertoires pour déterminer quelles lignes ont été exécutées un nombre différent de fois. Un utilitaire graphique comme `meld` sera utile pour réaliser cette comparaison.

Question 18

Réalisez la détection de branchements instables pour le code de calcul d'intégrale :

(a) Familiarisez-vous avec `gcov` en réalisant manuellement une couverture de code (standard).

(b) Réalisez la procédure ci-dessus de génération de deux couvertures de code, avec et sans Verrou. (Si vous trouvez cette étape trop fastidieuse, vous pouvez récupérer directement le script `runGcov` dans le corrigé)

(c) Comparez les résultats de couverture, et concluez sur la présence de branchements instables dans notre code.

4.5 Détection des branchements instables par couverture de basic blocs

Verrou propose également un mécanisme de détection de branchements instables ne reposant pas sur les outils fournis par le compilateur : il s'agit de la couverture par Basic Blocs (un BB est une séquence d'instructions sans branchements). Comme il n'y a pas de correspondances évidentes entre les lignes (ou plus exactement les symboles de debug obtenus avec `-g`), l'interface est un peu plus rustre, mais en contre-partie, on n'a plus besoin de mettre en oeuvre les outils du compilateurs et via l'utilisation de la client-request `VERROU_DUMP_COVER` ou les commandes `IOmatch`, on obtient des informations nouvelles.

Pour obtenir une couverture, on doit d'abord spécifier un fichier `trace.inc` qui possède le même format qu'un fichier d'exclusion pour spécifier quelle partie du code on souhaite analyser. Pour l'instant, on conseille de n'analyser que le binaire sans les bibliothèques [3] :

```
$ echo "*"  `readlink -f unitTest` > trace.inc
```

L'exécution de verrou avec l'option `--trace=trace.inc` permet de générer deux fichiers, `trace_bb_cov.log-PID` qui contient les informations de couverture et `trace_bb_info.log-PID` qui contient les informations de debug contenus dans chaque basic bloc. L'utilitaire `genCovBB` post-traite ces deux fichiers [4] pour obtenir un fichier lisible par l'utilisateur.

```
genCovBB trace_bb_cov.log-*
```

Question 19

Réalisez la détection de branchements instables pour le code de calcul d'intégrale, via les couvertures par basic-bloc :

(a) Créez le fichier `trace.inc` ;

(b) Générez une couverture en mode `nearest` ;

(c) Générez des couvertures en mode `random` ;

(d) Comparez les résultats de couverture, et concluez sur la présence de branchements instables dans notre code ;

(e) (optionnel) Ajoutez des clients requests `VERROU_DUMP_COVER` avant et après l'appel à `integrate` dans la fonction `testConvergence` et re-générez les couvertures. Quelles informations supplémentaires avez vous obtenues par rapport à la méthode par couverture de code ?

(f) (optionnel) Obtenez les mêmes informations sans avoir besoin de recompiler son code via l'utilisation de commande IOMatch.

3. Si le coeur de votre application se trouve dans des biblothèques dynamiques, il faut les inclure. L'idée ici est d'éviter de tracer ce qui se passe dans des bibliothèques externes.

4. On ne passe en paramètre qu'un seul fichier (le fichier de couverture), en cas de déplacement il faut les déplacer conjointement.

Références

[1] François Févotte and Bruno Lathuilière. Etudier la qualité numérique d'un code avec Verrou. Ecole d'été PRECIS (2018).

[2] François Févotte and Bruno Lathuilière. Verrou : Assessing floating-point accuracy without recompiling. HAL: 01383417.

[3] François Févotte and Bruno Lathuilière. Debugging and Optimization of HPC Programs with the Verrou Tool *International Workshop on Software Correctness for HPC Applications (Correctness) (2019)*, DOI: 10.1109/Correctness49594.2019.00006.

[4] François Févotte and Bruno Lathuilière. Studying the numerical quality of an industrial computing code : A case study on code_aster. *10th International Workshop on Numerical Software Verification (NSV)*, pp. 61–80, Heidelberg, Germany, July 2017. DOI: 10.1007/978-3-319-63501-9_5.

[5] Stef Graillat, Fabienne Jézéquel and Romain Picot. Numerical Validation of Compensated Algorithms with Stochastic Arithmetic. *Applied Mathematics and Computation*, vol. 329 (2018), pp. 339–363. DOI: 10.1016/j.amc.2018.02.004.

[6] Takeshi Ogita, Sigfried Rump and Shin'ichi Oishi. Accurate sum and dot product. SIAM Journal of Scientific Computing, vol. 26 (2005). DOI: 10.1137/030601818.

A Description des résultats du *Delta-Debugging*

- `dd.sym` : résultats du *delta-debugging* réalisé au niveau des fonctions (ou "symboles" dans la terminologie des fichiers objets) via la commande `verrou_dd_sym`.
  - `ref` : résultats de l'exécution de référence. Cette exécution n'est pas perturbée par les arrondis aléatoires, et permet aussi de générer la liste complète des fonctions (symboles) rencontrés durant l'exécution du programme.
    - `dd.{out,err}` : contenu de la sortie standard/erreur de `DD_RUN`
    - `checkRef.{out,err}` : contenu de la sortie standard/erreur de `DD_CMP ./ref ./ref`
    - `dd.sym` : liste complète de symboles (union des fichiers `dd.exclude.PID`)
    - `dd.sym.PID` : liste complète de symboles pour chaque execution
    - *autres fichiers* : résultats produits par `DD_RUN` et `DD_CMP`
  - *hash md5* : résultats d'une configuration partiellement perturbée.
    - `dd.sym.exclude` : liste des symboles non perturbés durant cette exécution
    - `dd.sym.include` : liste des symboles perturbés durant cette exécution
    - `dd.runX` : un répertoire par exécution, jusqu'à ce que le résultat soit considéré comme invalide par `DD_CMP` ($\Rightarrow$ configuration instable), ou que le nombre maximal d'exécutions soit atteint ($\Rightarrow$ configuration stable).
      - `dd.run.{out,err}` : sortie standard/erreur de `DD_RUN`
      - `dd.compare.{out,err}` : sortie standard/erreur de `DD_CMP`
      - `dd.return.value` : contient le résultat de `DD_CMP`
      - *autres fichiers* : résultats produits par `DD_RUN` et `DD_CMP`
  - `ddmin*` : lien symbolique vers une configuration 1-minimal. Il s'agit d'une configuration en échec tel que chaque sous ensemble privé d'un unique élément conduit à un succès (pour le nombre d'échantillons passé en paramètre).
  - `ddmin-cmp` : lien vers la configuration complémentaire à l'union des `ddmin*`. Il s'agit d'une configuration en succès.
  - `FullPerturbation` : lien vers la configuration où tous les symboles sont perturbés. Il doit s'agir d'une configuration en échec.
  - `NoPerturbation` : lien vers la configuration où aucun symbole n'est perturbé. Il doit s'agir d'une configuration en succès.
  - `rddmin_summary` : fichier contenant pour chaque lien symbolique `ddmin*` la statistique d'erreur, ainsi qu'une mise en forme avec `c++filt` des fichiers `dd.sym.include`.
- `dd.line` : résultats du *delta-debugging* réalisé au niveau des lignes de code source via la commande `verrou_dd_line`. Ce répertoire adopte la même structure que `dd.sym`.

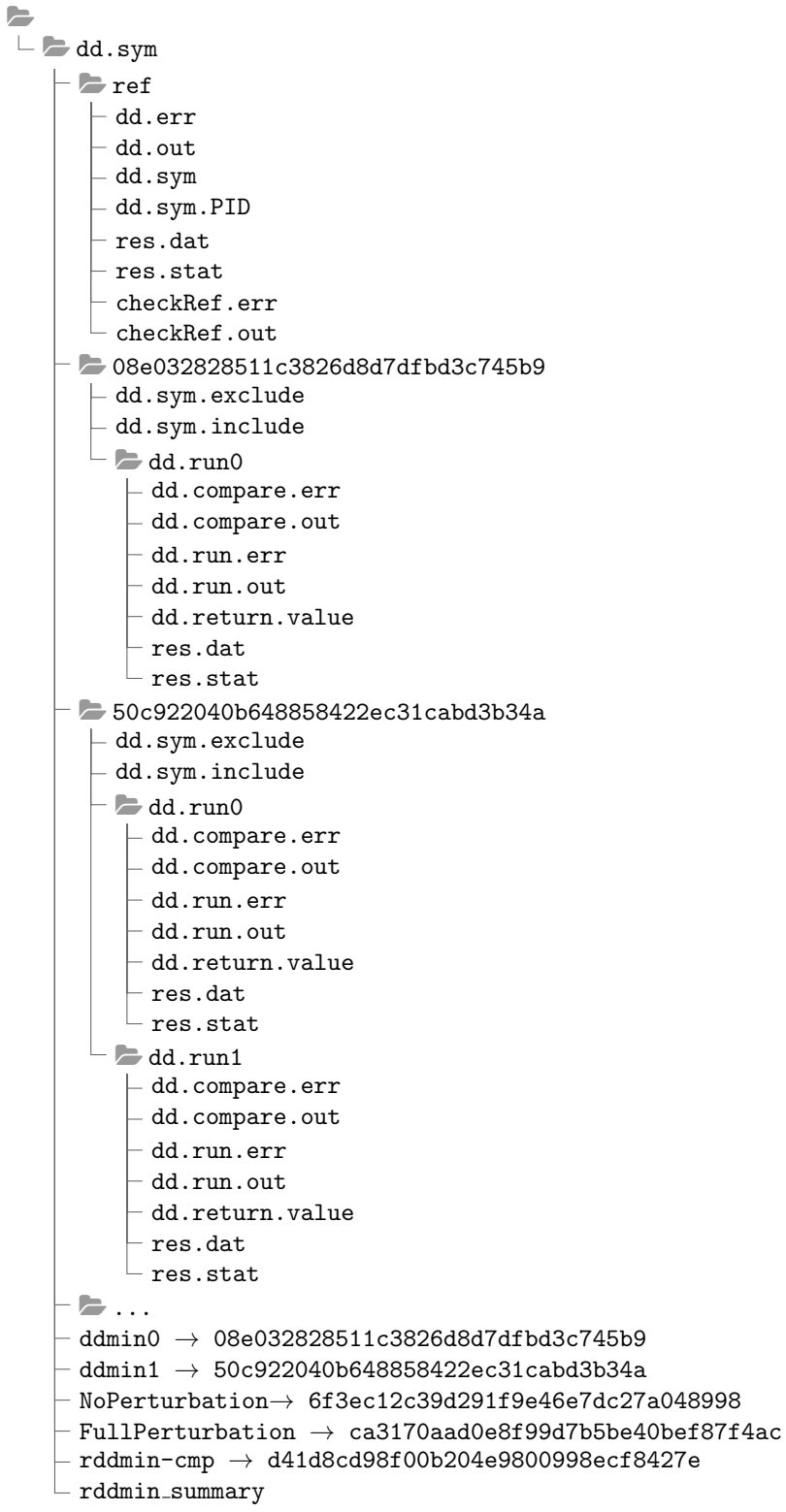


FIGURE 4 – Résultats du *Delta-Debugging*