Etudier la qualité numérique d'un code avec Verrou

François Févotte & Bruno Lathuilière
{francois.fevotte, bruno.lathuiliere}@edf.fr

Ecole Thématique de Simulation Numérique (ETSN) – 23-27 avril 2018

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 [1]. 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 [2].

Prérequis

La réalisation de ce TP nécessite des connaissances (au moins) élémentaires en Python (2), C++ (avec le compilateur Gnu) et Gnuplot.

Par ailleurs, nous supposons dans ce document que Verrou (version $\geqslant$ 1.1.0) 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/v1.1.0`

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.

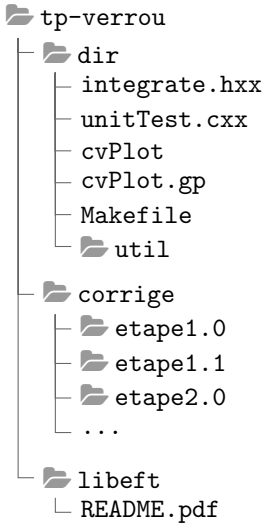
`cvPlot` : un *shell*-script permettant de lancer l'étude de convergence et de générer le graphe correspondant (à l'aide du script gnuplot `cvPlot.gp`). Un exemple de graphe produit est illustré en figure 2.

`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.

`libeft` : contient les sources de la bibliothèque `libEFT`, qui pourra servir à implémenter les algorithmes compensés dans la partie 2.4 du TP. La documentation de l'API fournie par cette bibliothèque est disponible dans le fichier `README.pdf`.



(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

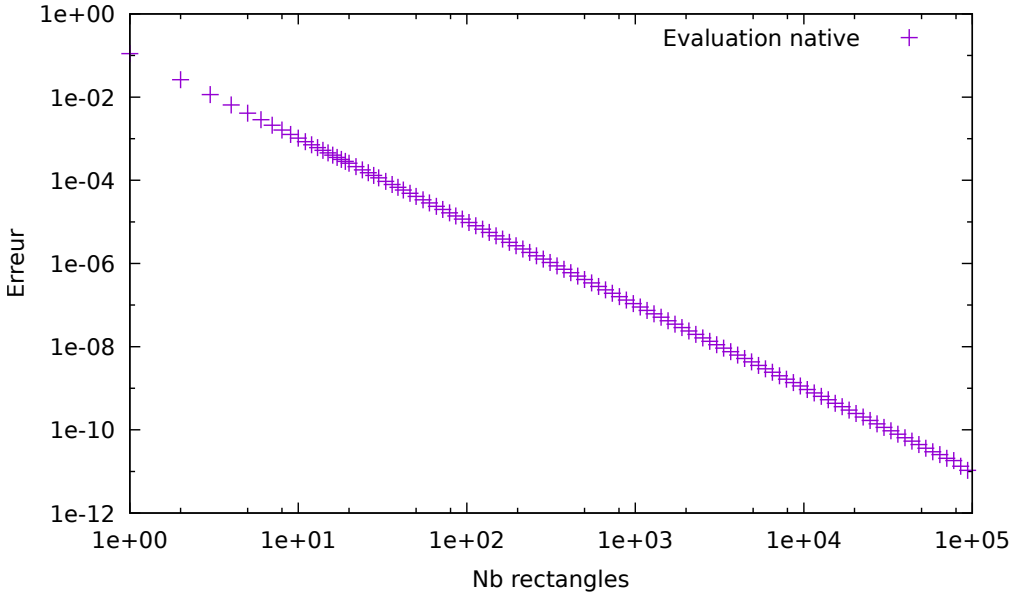


FIGURE 2 – Convergence du calcul d'intégrale

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 python
```

et essayez de réaliser quelques opérations flottantes.

(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 xrange(100)))`

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

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

Command line

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

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

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

>>> # cas (iii)
>>> from math import cos
>>> cos(42.)
-1.0050507702291946
```

On constate généralement que certains algorithmes de la bibliothèque mathématique (`libm`) sont incompatibles avec les modes d'arrondi autres que IEEE/*nearest*. Il est donc recommandé en première approche d'éviter de perturber les modes d'arrondi de la `libm`. On peut à cet effet fournir à Verrou une liste d'exclusion contenant les fonctions à ne pas instrumenter, avec la commande :

```
valgrind --tool=verrou --rounding-mode=random --demangle=no --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 :

(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
```

Question 4

Lancez `unitTest` dans Verrou en mode arrondi aléatoire.

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

(b) Refaites l'expérience en évitant de perturber la bibliothèque mathématique.

Le script `util/avg-stdev` fourni avec ce TP permet de comparer 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 des résultats au même format, mais dans lequel chaque colonne est remplacée par des valeurs moyennées sur tous les fichiers d'entrée.

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 $\mu_{i,j}$ définie par

$$\mu_{i,j} = \frac{1}{N_k} \sum_k x^k_{i,j}, \qquad \forall i, j.$$

En listant des colonnes supplémentaires avec l'option `--col` dans la ligne de commande, on ajoute à la sortie des colonnes supplémentaires contenant l'écart-type relatif des données correspondantes :

$$\sigma^{\text{rel}}_{i,j} = \frac{1}{|\mu_{i,j}|} \sqrt{\frac{1}{N_k} \sum_k (x^k_{i,j} - \mu_{i,j})^2}, \qquad \forall i, j.$$

Enfin, l'écart-type relatif maximal est imprimé sur la sortie d'erreur. S'il dépasse le critère maximum donné par l'utilisateur à l'aide de l'option `--dev-max`, le script renvoie une erreur (code de retour : 1).

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

Command line

```
$ ./util/avg-stdev --col=2 --dev-max=1e-15 cvPlot*.dat

    # (1)                    # (2)                    # (3)                      # (4)
    # input col 1 (avg)      # input col 2 (avg)      # input col 2 (std)        # input col 3 (avg)
    1.0000000000000000e+00   1.1107207345395915 3e+00 7.7969680123367611 4e-17   1.1072073453959153e-01
    1.0000000000000000e+01   1.0010288241427085 5e+00 1.4984583498728077 1e-16   1.0288241427085399 1e-03
    1.0000000000000000e+02   1.0000102809119060 2e+00 6.9775711326822977 6e-16   1.0280911906024314 8e-05
    1.0000000000000000e+03   1.0000001028083980 2e+00 4.0782957935980032 6e-15   1.0280839796150687 0e-07
    1.0000000000000000e+04   1.0000000010282343 9e+00 1.4149110381515172 9e-13   1.0282344420353695 1e-09
    1.0000000000000000e+05   1.0000000000095785 6e+00 2.2390959112784681 8e-13   9.5785046561047693 1e-12

    max dev: 2.2390959112784681 8e-13

$ echo $?
    1
```

dans lesquelles les 4 colonnes représentent :

1. la moyenne des nombres de rectangles (normalement identiques entre exécutions),
2. la moyenne des résultats obtenus,
3. **l'écart-type relatif des résultats obtenus** (c'est cette colonne qui nous intéresse),
4. la moyenne des écarts observés.

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.

(b) Que se passe-t-il selon qu'on intègre les résultats IEEE (non perturbés) ou pas à la statistique ?

(c) (optionnel) Que se passe-t-il avec des statistiques sur 10 échantillons ? Quel est l'impact du mode "proportionnel" de verrou ? Rappel : on active ce mode d'arrondi à l'aide de l'option :

```
--rounding-mode=average
```

```
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.

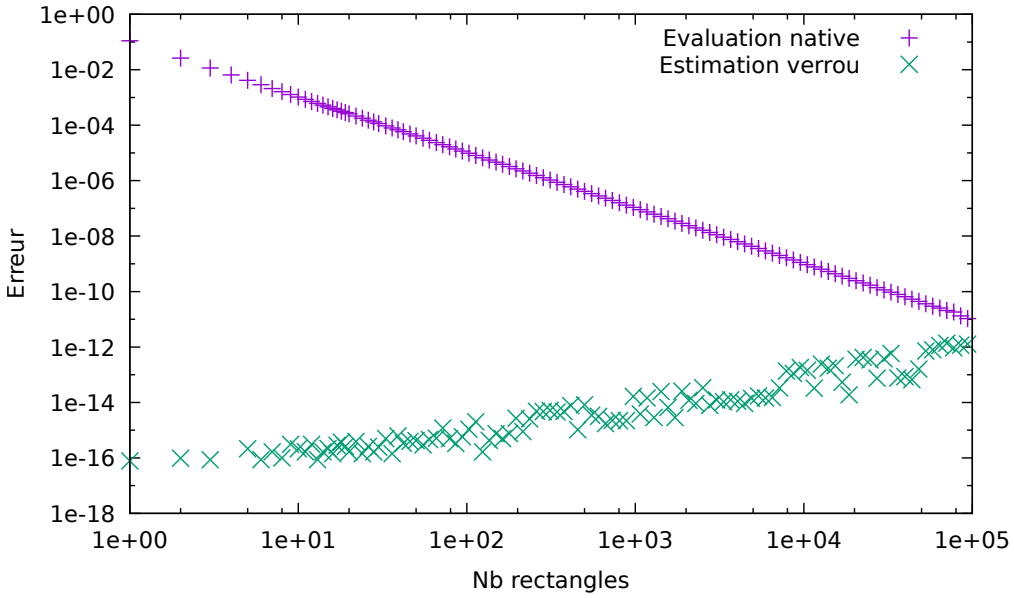


FIGURE 3 – Convergence du calcul d'intégrale et évaluation de l'erreur de calcul avec Verrou

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 Détection des branchements instables par couverture de code

Afin d'identifier l'origine des erreurs dans le code source, nous nous proposons de rechercher dans un premier temps les tests instables. Les erreurs de calculs sont en effet parfois amplifiées par un changement du flot d'exécution du code, lorsqu'un branchement dépend d'une valeur perturbée.

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` les options supplémentaires "`-fprofile-arcs -ftest-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.

La mise en place d'un mécanisme de détection de branchements instables reposant uniquement sur Verrou, sans dépendre d'un outil spécifique au compilateur, constitue l'un des prochains axes de développement de l'outil.

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 7

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.

2.2 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 8

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).

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

Nous nous proposons maintenant de rechercher les instabilités restantes à l'aide d'une approche par bisection, en sélectionnant les portions de code perturbées par Verrou. C'est ce que permet de faire l'outil `verrou_dd`, livré avec Verrou. `verrou_dd` s'utilise de la manière suivante :

```
verrou_dd  ${PWD}/DD_RUN  ${PWD}/DD_CMP
```

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 être identifiées par un *chemin absolu*, et 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}/...
```

`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})
...
```

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` 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 la variable d'environnement `VERROU_DD_NRUNS`.

Durant son exécution, `verrou_dd` 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 `verrou_dd` crée dans le répertoire courant des liens symboliques dont les noms indiquent les fonctions et lignes instables détectées :

`dd.sym.SYMBOLNAME` : indique que le symbole désigné par `SYMBOLNAME` a été détecté comme instable. En C++, le nom de la fonction correspondante n'est pas toujours facile à retrouver ; on peut utiliser l'utilitaire `c++filt` pour faire la conversion ;

`dd.line.FILE:LINE` : indique que la ligne `LINE` de `FILE` a été détectée comme instable.

Le script `util/dd-synthese` fourni avec le TP peut servir à y voir plus clair : en post-traitement d'une recherche par Delta-Debugging, il liste les symboles et lignes instables trouvées ainsi que quelques, informations synthétiques associées :

`NAME` : nom de la fonction (démanglée dans le cas du C++) ou fichier + numéro de la ligne ;

`STATUS` : `FAIL` ou `PASS`, selon que le test perturbant seulement cette fonction a échoué ou réussi. Si `PASS` apparaît ici, c'est probablement que les résultats du Delta-Debugging sont peu fiables ; il faut dans ce cas augmenter la valeur de `VERROU_DD_NRUNS` et relancer le Delta-Debugging ;

`COUNT` : dans le cas d'un *status* `FAIL`, nombre d'exécutions nécessaires pour faire échouer le test ;

`MSG` : si le script `DD_CMP` produit des sorties, la dernière ligne en est reproduite ici.

Dans notre cas, on pourra par exemple trouver :

Command line

```
$ ./util/ddSynthese

    NAME                              STATUS   COUNT   MSG
    ------------------------------------------------------------------------------
    testConvergence(float const&)     FAIL     1       max dev: 5.24523625259403690e-06
    integrate.hxx:23                  FAIL     1       max dev: 3.69550163275684936e-06
```

Attention : les fichiers contenus dans les répertoires `dd.sym` et `dd.run` servent 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.

Question 9

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) 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) Lancez `verrou_dd` et analysez les résultats.

(d) (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 la variable d'environnement `VERROU_DD_NRUNS`.

Question 10 (optionnelle)

Trouvez une manière alternative de mener l'analyse de Delta-Debugging (*i.e.* un autre couple de scripts `DD_RUN` & `DD_CMP`).

2.4 Compensation de la somme

Dans cette dernière étape, on propose de corriger la source d'erreur restante : l'accumulation d'erreurs d'arrondi dans la sommation.

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 [3] 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).

Question 11

Réaliser la compensation d'erreur dans la sommation des contributions de chaque rectangle.

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

(b) 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.

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

3 Pour aller plus loin...

Voici quelques idées de tests complémentaires à réaliser pour aller plus loin :

Question 12

Mesurer l'impact de la compensation sur le temps de calcul global.

Question 13

Réaliser une compensation de l'erreur du produit $h\, f(x_i)$. On obtient ainsi un algorithme compensé similaire à celui donné dans [4] pour le produit scalaire. Cela est-il utile ? coûteux en termes de performances ?

Question 14

Au lieu de ré-implémenter manuellement vos transformations exactes, utilisez les implémentations fournies par `libEFT` (`https://github.com/ffevotte/libeft`). Cette bibliothèque est fournie à côté des sources du TP, ainsi qu'une documentation de l'API qu'elle propose, dans le fichier `libeft/README.pdf`.

L'utilisation de libEFT a-t-elle un impact sur les performances du code ?

Question 15

Augmentez le nombre d'échantillons calculés avec Verrou, et étudiez la distribution des résultats. En particulier, est-elle gaussienne ? Les corrections effectuées aux étapes 2.2 et 2.4 ont-elles un impact uniquement sur l'écart-type de cette distribution, ou aussi sur sa forme ?

Références

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

[2] 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.

[3] 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.

[4] 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 de la passe de *delta-debugging* réalisée au niveau des fonctions (ou "symboles" dans la terminologie des fichiers objets).
  - `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`
    - `dd.exclude` : liste complète de symboles
    - *autres fichiers* : résultats produits par `DD_RUN`
  - *hash md5* : résultats d'une configuration partiellement perturbée.
    - `dd.exclude` : liste des symboles non perturbés durant cette exécution
    - `dd.include` : liste des symboles perturbés durant cette exécution
    - `dd.pass` : ce fichier n'est présent que si la configuration est considérée stable
    - `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`
      - *autres fichiers* : résultats produits par `DD_RUN`
- `dd.sym.SYMBOLNAME` : lien symbolique vers la configuration instable correspondant au cas où la seule fonction perturbée était `SYMBOLNAME`. Les fonctions listées ici sont celles considérées comme instables.
- `dd.line` : résultats de la passe de *delta-debugging* réalisée au niveau des lignes de code source. Ce répertoire adopte globalement la même structure que `dd.sym`, aux différences près listées ci-dessous :
  - `ref` : résultats de l'exécution de référence, servant à lister les lignes de code source à tester : il s'agit des lignes contenues dans les symboles identifiés comme instables lors de la première passe.
    - `dd.{exclude,include}` : liste des symboles non perturbés / perturbés. Les symboles perturbés correspondent à ceux qui ont été identifiés comme instables lors de la première passe.
    - `dd.source` : liste complète des lignes de code source associées à ces symboles.
  - *hash md5* : résultats d'une configuration partiellement perturbée
    - `dd.source` : liste des lignes perturbées dans cette configuration
- `dd.line.FILE:LINE` : lien symbolique vers la configuration instable dans laquelle seule la ligne `FILE:LINE` était perturbée. Les lignes listées ici sont celles considérées comme instables.

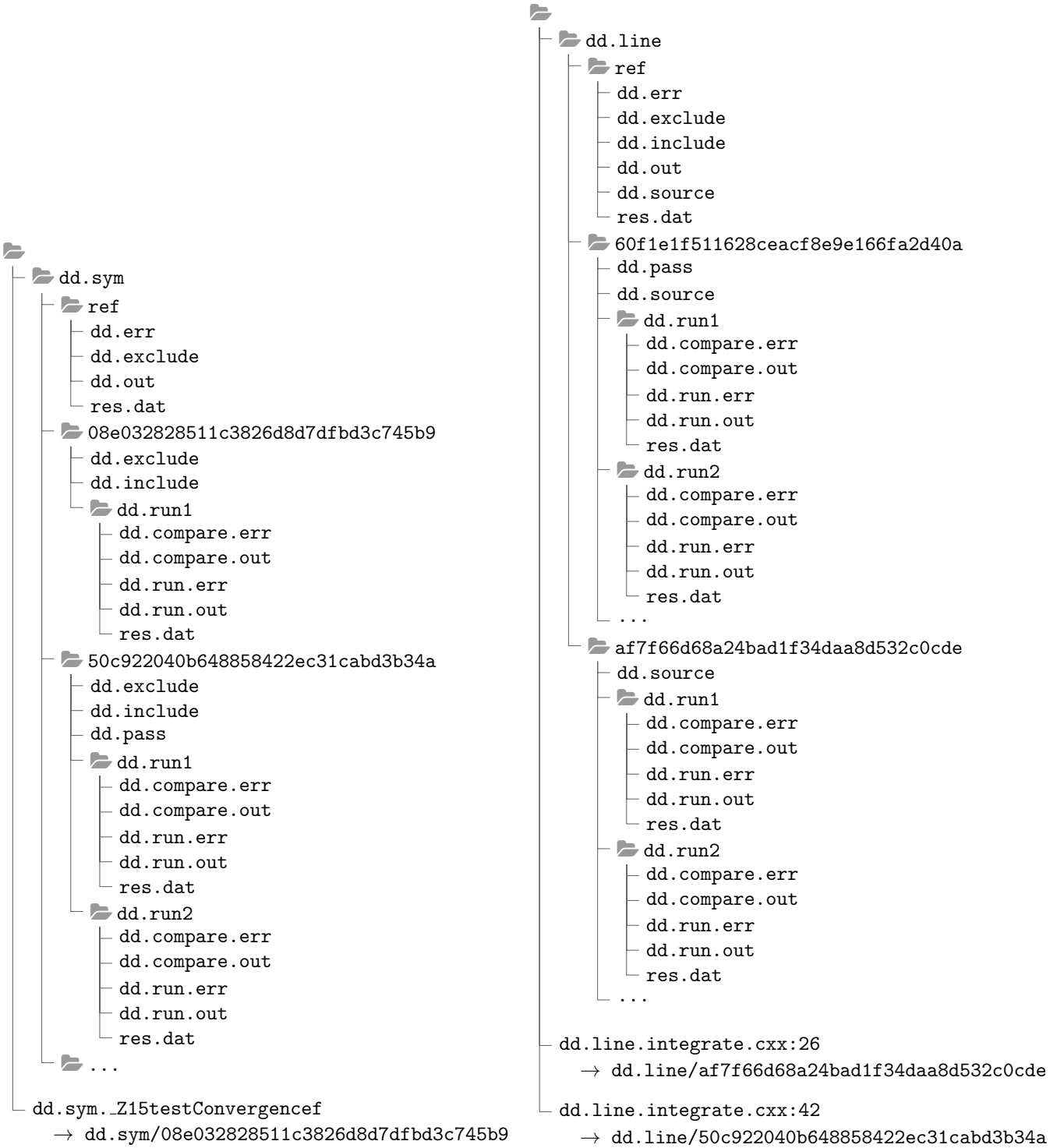


FIGURE 4 – Résultats du *Delta-Debugging*