Etudier la qualité numérique d'un code avec Verrou

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

Ecole PRECIS – 17 mai 2017

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.

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. 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).

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.cxx` : le code source C++ réalisant le calcul d'intégrale proprement dit, ainsi que 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 (`integrate`). 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.

`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.

```
src
+-- corrige
|   +-- etape1.0
|   +-- etape1.1
|   +-- etape2.0
|   +-- ...
+-- work
    +-- cvPlot
    +-- cvPlot.gp
    +-- integrate.cxx
    +-- Makefile
    +-- ...
```

(a) Organisation générale du code source de ce TP

```
$ ./integrate 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

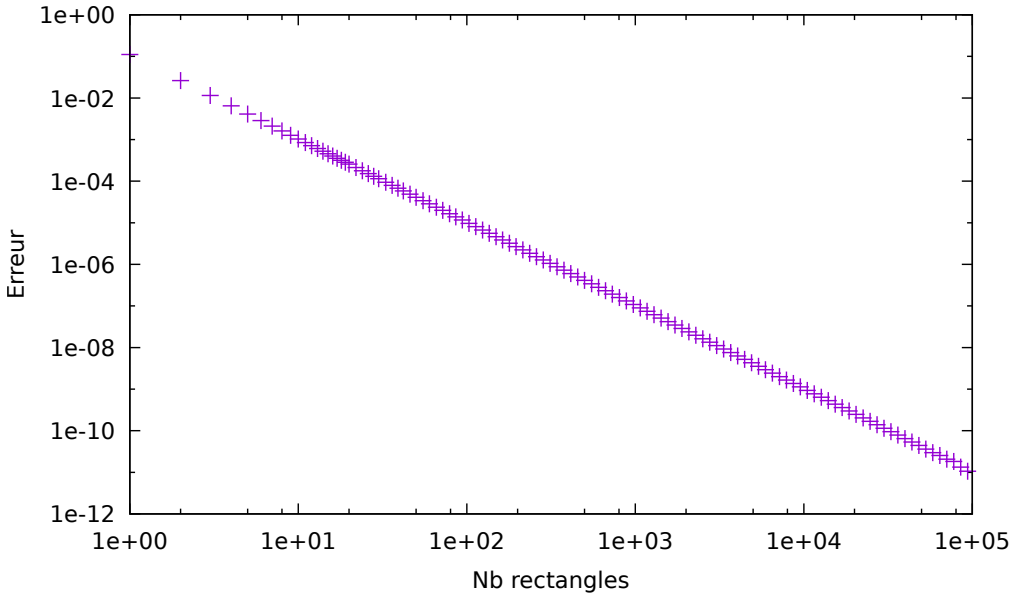

FIGURE 2 – Convergence du calcul d'intégrale

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) Regardez le code source des fonctions `integrate` et `testConvergence` dans le programme `integrate.cxx`.

(b) Lancez manuellement le programme `integrate` (avec un argument suffisamment grand pour ne pas être submergé par les sorties).

(c) Lancez le script `cvPlot` et étudiez la courbe de convergence produite dans `cvPlot.pdf`. 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 : `./integrate 10`
— Valgrind sans outil : `valgrind --tool=none ./integrate 10`
— Valgrind/memcheck : `valgrind ./integrate 10`
— Verrou sans perturbation : `valgrind --tool=verrou --rounding-mode=nearest ./integrate 10`

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 Familiarisez-vous avec Verrou. Par exemple, lancez Verrou sur l'interpréteur Python et essayez de réaliser quelques opérations flottantes :

```
$ valgrind --tool=verrou --rounding-mode=random python
>>> sum([0.1*i for i in xrange(100)])
495.00000000000034
>>> 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=libm.ex
```

Les fonctions sont listées dans le fichier fourni (`libm.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 :

```
__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 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`) :

```
*    /lib/x86_64-linux-gnu/libm-2.23.so
```

Une autre manière de faire consistre à demander à Verrou de dresser la liste complète des fonctions vues durant l'exécution du programme. On peut ensuite la filtrer pour ne garder que les fonctions qu'on souhaite réellement exclure. Cette liste peut être obtenue à l'aide de la commande :

```
valgrind --tool=verrou --demangle=no --gen-exclude=all.ex
```

Question 4 Lancez `integrate` dans Verrou en mode arrondi aléatoire.

(a) Que se passe-t-il ?

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

Le script `compare` fourni dans les sources permet de comparer plusieurs résultats d'exécution de `integrate` : il prend en argument une liste de fichiers contenant des sorties de `integrate`, et affiche sur sa sortie standard des résultats au même format à trois colonnes :

1. nombre de rectangles,
2. moyenne des résultats fournis en entrée,
3. écart-type des résultats fournis en entrée.

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

Question 5 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.

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

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.

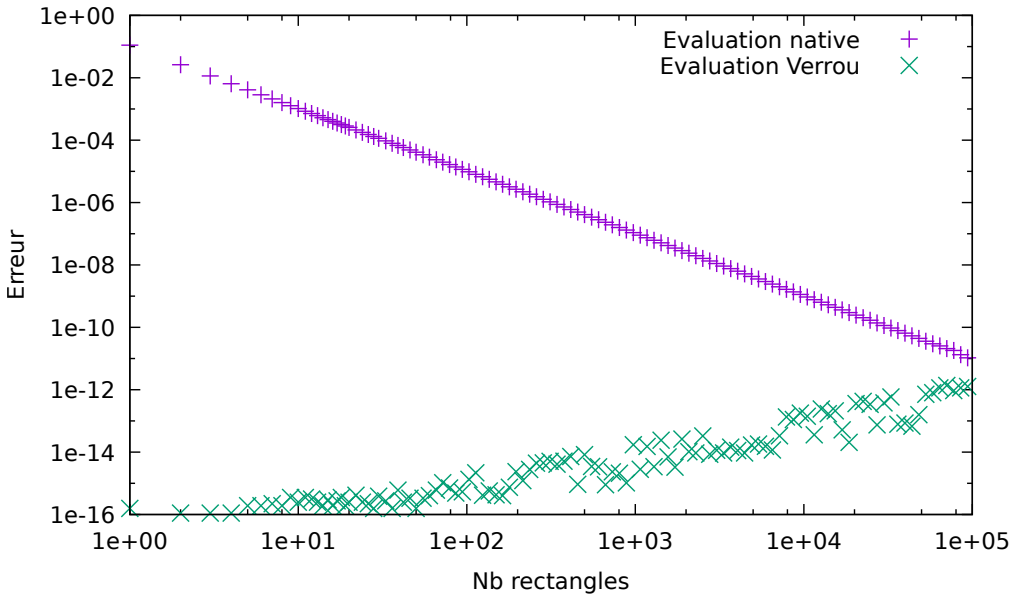

FIGURE 3 – Convergence du calcul d'intégrale et évaluation de l'erreur de calcul avec Verrou

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.cxx` 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*

Afin d'identifier l'origine des erreurs dans le code source, nous nous proposons d'utiliser dans un premier temps la fonctionnalité de recherche des fonctions et lignes instables par bisection. C'est ce que permet de faire l'outil `verrou_dd`, livré avec Verrou. `verrou_dd` s'utilise de la manière suivante :

```
verrou_dd DD_RUN 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`.

`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.

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.

Question 7 Réalisez le *delta-debugging* du code de calcul d'intégrale :

(a) Les scripts `ddRun` et `ddCmp` déjà fournis implémentent les fonctionnalités nécessaires pour utiliser `verrou_dd`. Étudiez leur fonctionnement. L'analyse de la courbe de convergence pourra permettre de dimensionner de manière adéquate le seuil de tolérance dans la validation des résultats effectuée par `ddCmp`.

(b) Lancez `verrou_dd` et analysez les résultats.

2.2 Détection des branchements instables par couverture de code

Outre leur accumulation qui finit par dégrader la qualité des résultats, les erreurs de calculs sont 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, comme `gcov`. L'utilisation de `gcov` 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.

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 8 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.

(b) Le script `runGcov` fourni automatise la procédure ci-dessus de génération des deux couvertures de code. Étudiez son fonctionnement et lancez-le.

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

2.3 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).

Question 10 Identifiez les sources d'erreurs restantes, en reprenant une procédure de *Delta-Debugging*. Il sera éventuellement utile d'adapter le critère d'acceptabilité des résultats pour prendre en compte la stabilité accrue du code suite aux corrections déjà effectuées.

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.

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
$s \leftarrow \pi_n + q_n$;

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.

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.

```
dd.sym
+-- ref
|   +-- dd.err
|   +-- dd.exclude
|   +-- dd.out
|   +-- res.dat
+-- 08e032828511c3826d8d7dfbd3c745b9
|   +-- dd.exclude
|   +-- dd.include
|   +-- dd.run1
|       +-- dd.compare.err
|       +-- dd.compare.out
|       +-- dd.run.err
|       +-- dd.run.out
|       +-- res.dat
+-- 50c922040b648858422ec31cabd3b34a
|   +-- dd.exclude
|   +-- dd.include
|   +-- dd.pass
|   +-- dd.run1
|   |   +-- dd.compare.err
|   |   +-- dd.compare.out
|   |   +-- dd.run.err
|   |   +-- dd.run.out
|   |   +-- res.dat
|   +-- dd.run2
|   |   +-- dd.compare.err
|   |   +-- dd.compare.out
|   |   +-- dd.run.err
|   |   +-- dd.run.out
|   |   +-- res.dat
|   +-- ...
...

dd.sym._Z15testConvergencef
  -> dd.sym/08e032828511c3826d8d7dfbd3c745b9
```

```
dd.line
+-- ref
|   +-- dd.err
|   +-- dd.exclude
|   +-- dd.include
|   +-- dd.out
|   +-- dd.source
|   +-- res.dat
+-- 60f1e1f511628ceacf8e9e166fa2d40a
|   +-- dd.pass
|   +-- dd.source
|   +-- dd.run1
|   |   +-- dd.compare.err
|   |   +-- dd.compare.out
|   |   +-- dd.run.err
|   |   +-- dd.run.out
|   |   +-- res.dat
|   +-- dd.run2
|   |   +-- dd.compare.err
|   |   +-- dd.compare.out
|   |   +-- dd.run.err
|   |   +-- dd.run.out
|   |   +-- res.dat
|   + ...
+-- af7f66d68a24bad1f34daa8d532c0cde
|   +-- dd.source
|   +-- dd.run1
|   |   +-- dd.compare.err
|   |   +-- dd.compare.out
|   |   +-- dd.run.err
|   |   +-- dd.run.out
|   |   +-- res.dat
|   +-- dd.run2
|       +-- dd.compare.err
|       +-- dd.compare.out
|       +-- dd.run.err
|       +-- dd.run.out
|       +-- res.dat
...

dd.line.integrate.cxx:26
  -> dd.line/af7f66d68a24bad1f34daa8d532c0cde

dd.line.integrate.cxx:42
  -> dd.line/50c922040b648858422ec31cabd3b34a
```

FIGURE 4 – Résultats du *Delta-Debugging*