Tutoriel sur COJAC - Sniffeur de problèmes numériques et usine de nombres enrichis pour Java

Par Lucy Linder - Frédéric Bapst

Date de publication : 20 janvier 2016

Cet article présente deux aspects de l'outil **COJAC** pour Java : (i) la *détection automatique des pièges que tendent les nombres* ; (ii) une toute nouvelle fonctionnalité capable de *doper les nombres à virgule*.

Pour réagir à ce tutoriel, un espace de dialogue vous est proposé sur le forum : **Commentez**

I - Des nombres ? Des pièges !

Les nombres manipulés par nos ordinateurs ne se comportent pas comme les nombres des mathématiciens. Comme développeur, vous connaissez parfaitement le phénomène du dépassement de capacité sur les nombres entiers (integer overflow), et vous êtes certainement conscients que les nombres à virgule flottante apportent leur lot de limitations et anomalies, comme l'annulation (3.000001f - 3.0f), la souillure (342.0 - 1.0E+43), les résultats spéciaux comme NaN, le soupassement de capacité (underflow), etc. Identifier les différents problèmes est une chose, s'assurer que le code qu'on produit est robuste est bien plus ardu !

Beaucoup a déjà été dit au sujet de ces problèmes numériques. Ils forment une sorte de poison pour les programmeurs, provoquent de graves défectuosités logicielles et ouvrent ainsi des brèches de sécurité. Qui plus est, ils risquent de survenir peut-être plus souvent qu'on pourrait le penser.

Prenons l'addition de deux ints en Java. Parmi toutes les combinaisons de valeurs possibles, quelle est la proportion des données qui vont causer un *dépassement de capacité*, c'est-à-dire qu'effectivement le résultat est complètement faux ? C'est 25 %. Et ce taux atteint 50 % si on considère seulement les nombres positifs.

> La moitié des additions de nombres naturels foirent complètement, honte aux ordinateurs !

Prenons maintenant l'addition de deux doubles. Combien de sommes causeront une souillure, à savoir que le calcul est détourné comme si l'un des opérandes est soudain remplacé par zéro ? Eh bien environ 95 % !

Et à nouveau du côté des entiers, cette fois avec la multiplication, une opération vraiment inhabituelle vous en conviendrez. Si les opérandes sont choisis au hasard, le risque de dépassement de capacité atteint… 99.99999 % !

Iriez-vous encore prétendre que vous êtes l'heureux programmeur qui n'est pas concerné par les problèmes numériques ? Ne souhaitez-vous pas au moins être informé lorsque de tels événements sournois surviennent durant le développement et les tests ?

Java ne signale absolument pas ces anomalies pernicieuses, il n'y a aucune option de compilation pour vous secourir. Aussi avons-nous décidé de réaliser cette fonctionnalité.

II - Exemple de défectuosités révélées par COJAC

Voici un petit exemple. Nous voulons implémenter l'opération « *powerModulo* » sur les nombres naturels : calculer (x^y mod z) (« x puissance y, modulo z »). Nous écrivons deux implémentations différentes, accompagnées de quelques jeux de test pour la validation :

```
1. static int powerModA(int x, int y, int z) {
2.   int res=1;
3.   while(y-- > 0)
4.     res = res*x;
5.   return res % z;
6. }
7.
8. static int powerModB(int x, int y, int z) {
9.   return (int) Math.pow(x,y) % z;
10. }
11.
12. @Test public void tinyTest() {
13.   int a,b;
14.   a = powerModA(2, 4, 5);
15.   b = powerModB(2, 4, 5);
16.   assertEquals(1, a);
17.   assertEquals(1, b);
18.   a = powerModA(497, 1854, 281);
19.   b = powerModB(497, 1854, 281);
20.   assertEquals(157, a);
```

```
21.    assertEquals(157, b);
22. }
```

Ensuite on lance le test :

```
$ java demo.HelloCojac
```

Il se trouve ici que tout semble se dérouler à merveille, alors qu'il est clair que les *deux implémentations sont totalement défectueuses*, et que le succès du second cas de test est purement accidentel.

Maintenant, démarrons exactement le même programme test à travers **COJAC** :

```
$ java -javaagent:cojac.jar demo.HelloCojac

COJAC: Overflow : IMUL
    demo.HelloCojac.powerModA(HelloCojac.java:11)
COJAC: Result is +Infinity: java/lang/Math.pow(DD)D
    demo.HelloCojac.powerModB(HelloCojac.java:16)
COJAC: Overflow : D2I
    demo.HelloCojac.powerModB(HelloCojac.java:16)
```

Comme vous le constatez, on reçoit plusieurs avertissements qui mettent en évidence les problèmes numériques cachés. Sympa, n'est-ce pas ?

COJAC est un outil gratuit, plutôt stable et efficace, avec de nombreuses options pour ajuster ce qui doit être signalé et comment. L'outil s'intègre facilement dans votre IDE. Pas besoin de recompilation. Tout se passe à l'exécution, avec une instrumentation *à la volée*.

Si vous avez déjà perdu un temps précieux à ramper dans le code pour dénicher quelle ligne amène cette stupide valeur NaN qui paralyse toute la suite des traitements, vous apprécierez un peu d'assistance ! Alors, n'hésitez pas à bénéficier de notre outil de diagnostic, nous serions tout fiers s'il pouvait vous rendre service.

L'histoire ne s'arrête pas là : si vous avez envie de découvrir une fonctionnalité encore plus épatante, lisez la suite ! Au début vous aurez de la peine à y croire…

III - Nombres à virgule métamorphosés en superhéros

Or donc vous avez fait connaissance avec l'outil **COJAC** (**C**hecking **O**verflows in **JA**va **C**ode), capable de signaler les dépassements de capacité et plein d'autres anomalies numériques sournoises. Mais il est temps d'étendre les capacités numériques du langage Java de façon bien plus ambitieuse, à l'aide de…**COJAC** (**C**limbing **O**ver **J**ava **A**rithmetic **C**apabilities).

Ce qu'offre Java pour simuler les nombres réels se résume à deux sortes de nombres à virgule flottante : `float` et `double`. Et comme dans n'importe quel autre langage de programmation, ces types de nombres sont terriblement pauvres. D'accord, l'empreinte mémoire est légère et les temps de calcul plutôt courts, mais avoir une petite taille et faire les choses dans la précipitation pourrait être une vue assez limitée du paradis…

Avec **COJAC**, vous écrivez votre code avec des floats et des doubles, comme d'habitude. Mais ensuite, au moment de l'exécution, on devient presque tout-puissant : au lancement du programme, on est *libre de choisir un type de nombre plus riche* pour ses calculs. Ce remplacement de nombres, automatique et à la volée, ouvre de nouvelles perspectives que nous allons esquisser avec le petit exemple suivant.

Supposons que dans votre projet vous avez à calculer un certain polynôme :

$$f(x,y) = \frac{1335}{4}y^6 + x^2(11x^2 * y^2 - y^6 - 121y^4 - 2) + \frac{11}{2}y^8 + \frac{x}{2y}$$

Vous écrivez l'implémentation simple et naïve que voici :

```
1. static double pow(double base, int exp) {
2.     double r=1.0;
3.     while(exp-- > 0) r*=base;
4.     return r;
5.   }
6.
7.   public static double somePolynomial(double x, double y) {
8.     return 1335.0*(pow(y, 6))/4.0
9.          + x*x*(11.0*x*x*y*y -pow(y, 6) -121.0*pow(y, 4) -2.0)
10.          + 11.0*pow(y, 8)/2.0
11.          + x/(2.0*y);
12.   }
13.
14.   public static void main(String[] args) {
15.     double r, x, y;
16.     x=2.0; y=3.0;
17.     r=somePolynomial(x, y);
18.     System.out.println(r);
19.   }
```

Lançons le programme :

```
$ java demo.HelloPolynomial
```

Avec ce premier essai, il se trouve que le résultat est effectivement correct. Maintenant, essayons d'évaluer la fonction en un autre point :

```
1. public static void main(String[] args) {
2.     double r, x, y;
3.     x=77617; y=33096;  // <--------------------
4.     r=somePolynomial(x, y);
5.     System.out.println(r);
6.   }
```

On relance la chose :

```
$ java demo.HelloPolynomial
```

Ce qu'on obtient est alors (-1.18E+21), alors que le vrai résultat vaut environ (-0.827) … Moins de un, contre des millions de milliards… Aïe, honte aux ordinateurs !

Bon, si vous êtes un développeur futé, vous pourriez avoir entendu parler d'un certain « *sniffeur de problèmes numériques* » susceptible de signaler les anomalies qui apparaissent durant les calculs :

```
$ java -javaagent:cojac.jar demo.HelloPolynomial

COJAC: Smearing: DSUB
  demo.HelloPolynomial.somePolynomial(HelloPolynomial.java:13)
COJAC: Cancellation: DADD
  demo.HelloPolynomial.somePolynomial(HelloPolynomial.java:12)
COJAC: Smearing: DADD
  demo.HelloPolynomial.somePolynomial(HelloPolynomial.java:12)
```

Fort bien, recevoir des avertissements est certes de loin préférable à un silence gêné, mais cette information reste assez lacunaire : la souillure ou l'annulation ne sont pas forcément catastrophiques, et on ne sait toujours pas dans quelle mesure le résultat final est entaché d'erreur. C'est là qu'intervient la magie de l'*usine de nombres enrichis* **COJAC**. On va garder le même programme, mais en demandant à **COJAC** de passer à l'*arithmétique d'intervalles* en lieu et place du type primitif `double` :

```
$ java -javaagent:cojac.jar="-Ri" demo.HelloPolynomial
```

```
COJAC: WrapperInterval detects unstability...[-3.54E+22 ; 3.42E+22]
demo.HelloPolynomial.somePolynomial(HelloPolynomial.java:12)
```

Merveilleux, cette fois on est averti que le résultat final a une énorme incertitude puisque l'intervalle correspondant couvre plus de vingt ordres de grandeur, et que le signe même du résultat est incertain ! En passant, notre première invocation qui donnait un résultat correct ne soulève pas de pareille mauvaise nouvelle.

Mais attendez, **COJAC** peut faire mieux qu'expliquer que votre résultat est inutilisable. Il est capable de calculer plus précisément si on lui demande gentiment. Voici comment paramétrer **COJAC** pour utiliser des nombres décimaux à 40 chiffres significatifs au lieu des 16 offerts par le type double.

```
$ java -javaagent:cojac.jar="-Rb 40" demo.HelloPolynomial
```

Et ce qu'on obtient alors est (-0.827396), soit une approximation très raisonnable du résultat mathématique pur.

IV - Des nombres encore plus étonnants

Une fois qu'on a des types plus riches à disposition, on peut faire fructifier le code existant de manière plus intelligente encore. Admettons que le chef a maintenant besoin de la dérivée en (X) de ce polynôme. Pas besoin de se plonger dans l'implémentation de la fonction ni dans un bouquin de maths : on va juste tirer profit de ce qu'on appelle la « *dérivation automatique*"». Ajoutons 2-3 lignes :

```
1.   public static double COJAC_MAGIC_getDerivation(double a) { return 0;  }
2.   public static double COJAC_MAGIC_asDerivationTarget(double a) { return a; }
3.
4.   public static void main(String[] args) {
5.     double r, x, y;
6.     x=2.0; y=3.0;
7.     x=COJAC_MAGIC_asDerivationTarget(x);
8.     r=somePolynomial(x, y);
9.     System.out.println("f (x,y): "+r);
10.     System.out.println("f'(x,y): "+COJAC_MAGIC_getDerivation(r));
11.   }
```

Ensuite, on demande simplement à **COJAC** d'utiliser la dérivation automatique, et on est servi :

```
$ java -javaagent:cojac.jar="-Ra" demo.HelloPolynomial

f (x,y): 238845.58333333334
f'(x,y): -38954.0
```

C'est tout aussi simple si on veut dériver en (Y) plutôt qu'en (X) :

```
1.   public static double COJAC_MAGIC_getDerivation(double a) { return 0;  }
2.   public static double COJAC_MAGIC_asDerivationTarget(double a) { return a; }
3.
4.   public static void main(String[] args) {
5.     double r, x, y;
6.     x=2.0; y=3.0;
7.     y=COJAC_MAGIC_asDerivationTarget(y);   // <--------------------
8.     r=somePolynomial(x, y);
9.     System.out.println("f (x,y): "+r);
10.     System.out.println("f'(x,y): "+COJAC_MAGIC_getDerivation(r));
11.   }
```

Et voilà le travail :

```
$ java -javaagent:cojac.jar="-Ra" demo.HelloPolynomial

f (x,y): 238845.58333333334
```

```
f'(x,y): 8113580.0
```

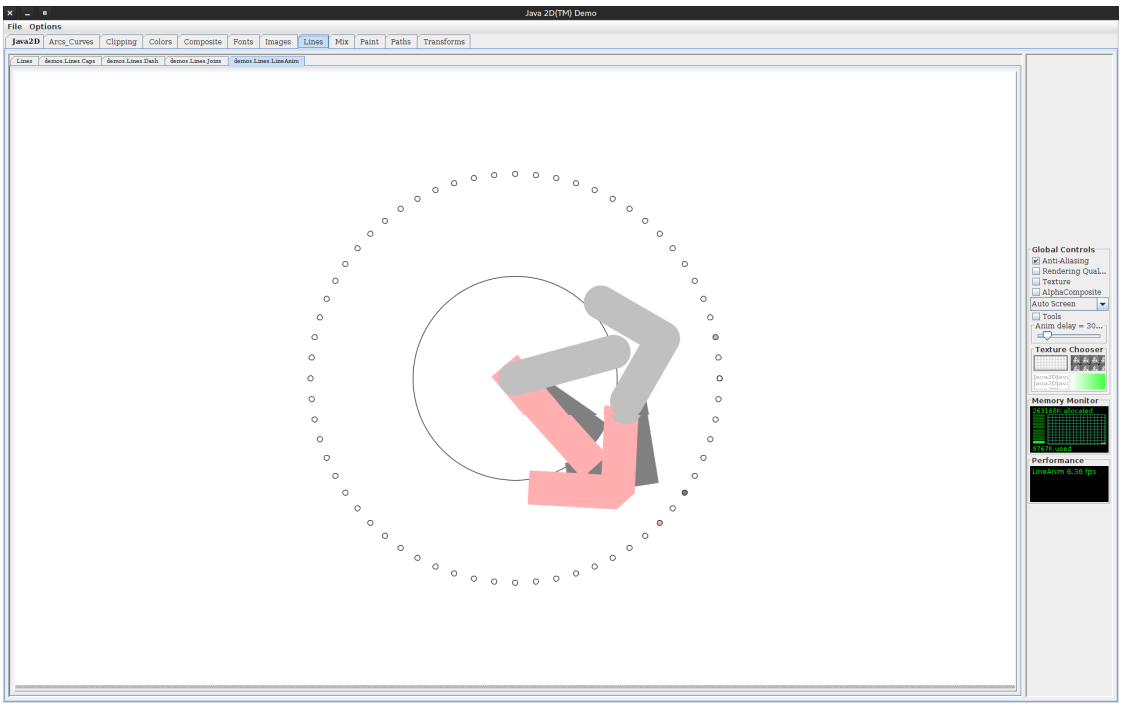
La dérivation automatique est une idée géniale (assez ancienne et un peu méconnue), et nous prétendons qu'aucun autre outil ne permet de l'expérimenter avec ce niveau de simplicité.

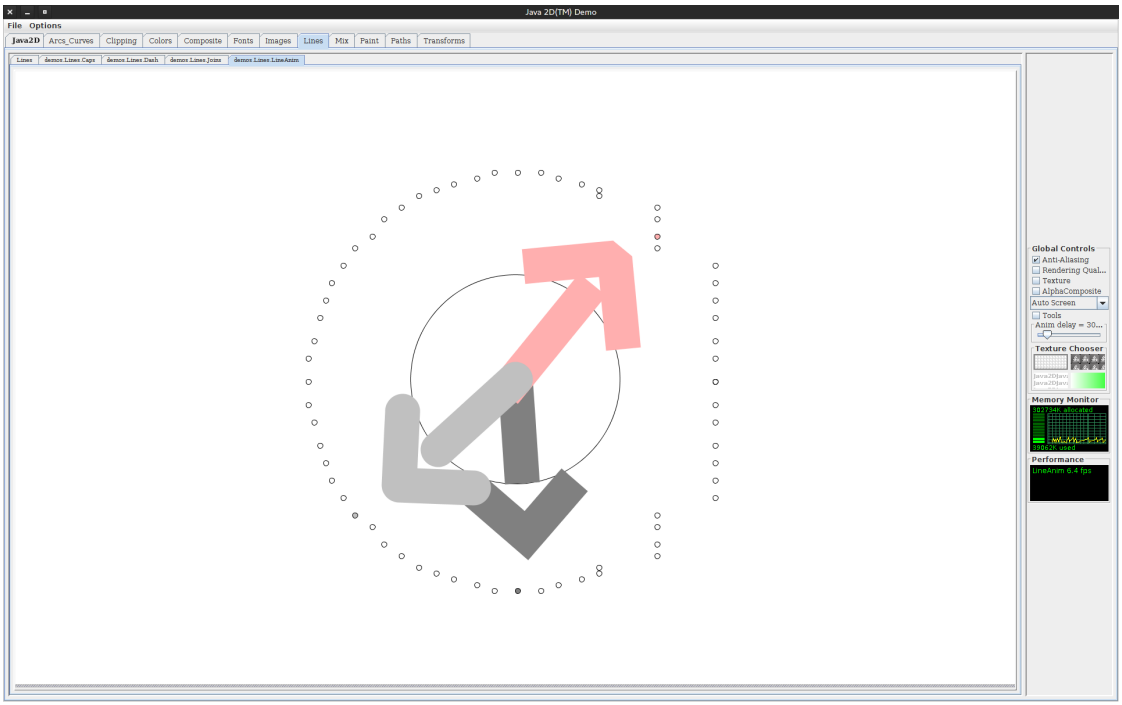
V - Conclusion

Voilà, nos travaux apportent au moins un éclairage original sur les nombres en Java. Honnêtement, nous ignorons si notre outil (qui a certes encore quelques limites) aura un quelconque impact parmi les développeurs… Mais ce qu'il y a de sûr, c'est qu'on trouve chouette de jouer avec ! Juste pour le plaisir, vous pouvez essayer la démo officielle de Java2D, et observer comment elle se comporte lorsqu'on demande de calculer avec seulement deux chiffres significatifs :

```
$ java -javaagent:cojac.jar="-Rb 2" -jar java2Demo.jar
```

Comme on le voit sur les figures ci-dessous, l'horloge subit quelques distorsions par exemple.





Maintenant, à vous de jouer : parcourez la **documentation**, **téléchargez le Jar**, et faites vos premiers pas avec **COJAC**, le meilleur « sniffeur de problèmes numériques » pour Java, doublé de la meilleure « usine de nombres enrichis » ! N'oubliez pas de nous transmettre vos commentaires (via un ticket sur https://github.com/Cojac/Cojac).

VI - Annexe

Pour en savoir plus, vous pouvez consulter la vidéo suivante :

Cliquer sur ce lien pour lancer l'animation