UNIVERSITÉ DE MONTRÉAL

Tâche 2 - Documentation

Par
Cédric Guévremont (20266532)
Rima Boujenane (20235550)

TRAVAIL PRÉSENTÉ À BENOIT BEAUDRY
DANS LE CADRE DU COURS IFT 3913
QUALITÉ DU LOGICIEL ET MÉTRIQUES

10 OCTOBRE 2025

Documentation des cas de test - `DistanceCalcEuclidean`

Contexte général. Les tests suivants ont été conçus pour vérifier le comportement de la classe `DistanceCalcEuclidean`, responsable du calcul de distances et de points intermédiaires entre coordonnées géographiques (latitude, longitude).

Test 1 - `testCalcDistSamePoint`

Intention : Vérifier que la distance euclidienne entre deux points identiques est nulle.

Motivation des données : Cas limite fondamental (A = B) qui doit retourner 0 et détecte des erreurs élémentaires (offset, permutations de coordonnées, mutations arithmétiques).

Oracle : Par définition de la distance euclidienne 2D :

$$d(A, A) = \sqrt{(lat_A - lat_A)^2 + (lon_A - lon_A)^2} = 0$$

La valeur attendue est donc 0.0 avec une tolérance de 10^{-12}.

Test 2 - `testCalcDistNaN`

Intention : Confirmer la gestion des entrées invalides (`NaN`) : la distance doit être `NaN`.

Motivation des données : Vérifier la robustesse du calcul et la propagation correcte des erreurs flottantes. Évite qu'un résultat numérique "valide" soit produit par accident.

Oracle : Selon la norme IEEE-754, toute opération impliquant un `NaN` retourne `NaN`. Ainsi, `calcDist(NaN, 0, 0, 0)` doit satisfaire `Double.isNaN(...)`.

Test 3 - `returnsFirstPointWhenFZero`

Intention : Vérifier qu'à $f = 0$, le point intermédiaire correspond exactement au premier point.

Motivation des données : Cas limite gauche de l'intervalle d'interpolation $[0, 1]$, sensible aux inversions de coefficients $(1 - f)$ et f.

Oracle : L'interpolation linéaire 2D est donnée par :

$$P = (1 - f) \cdot A + f \cdot B$$

Pour $f = 0$, on obtient $P = A$. Les valeurs attendues sont donc lat = 10.0, lon = 20.0 (tolérance 10^{-9}).

Test 4 - `returnsSecondPointWhenFOne`

Intention : Vérifier qu'à $f = 1$, le point intermédiaire correspond exactement au second point.

Motivation des données : Cas limite droite de l'intervalle, complémentaire du test précédent.

Oracle :

$$P = (1 - f) \cdot A + f \cdot B \quad \Rightarrow \quad P = B \text{ lorsque } f = 1$$

Les valeurs attendues sont lat = 30.0, lon = 40.0 (tolérance 10^{-9}).

Test 5 - `midpointIsCorrect`

Intention : Valider que le milieu ($f = 0.5$) donne la moyenne arithmétique des coordonnées.

Motivation des données : Le milieu est un repère simple, sensible aux inversions de signe et erreurs d'échelle.

Oracle :

$$P = 0.5 \cdot A + 0.5 \cdot B$$

Entre $(0, 0)$ et $(10, 10)$, on attend $(5, 5)$ avec une tolérance de 10^{-9}.

Test 6 - `symmetricPointsGiveSameMidpoint`

Intention : Vérifier la symétrie de l'interpolation à $f = 0.5$: échanger les points A et B ne change pas le résultat.

Motivation des données : Propriété fondamentale de l'interpolation linéaire ; sensible à toute asymétrie d'implémentation.

Oracle :

$$0.5 \cdot A + 0.5 \cdot B = 0.5 \cdot B + 0.5 \cdot A$$

Les deux midpoints doivent être identiques (écart maximal 10^{-9}).

Test 7 - `latLonWithinRange`

Intention : S'assurer que pour $0 \leq f \leq 1$, les coordonnées P_{lat} et P_{lon} du point interpolé se situent toujours dans l'intervalle défini par les deux points d'origine.

Motivation des données : Vérifie l'invariant de convexité : l'interpolation linéaire doit produire un point compris entre A et B, sans dépassement ("overshoot").

Oracle : Étant donné :

$$P = (1 - f) \cdot A + f \cdot B \quad \text{avec} \quad f \in [0, 1],$$

chaque coordonnée de P est une combinaison convexe de celles de A et B. Ainsi :

$$\min(A_{lat}, B_{lat}) \leq P_{lat} \leq \max(A_{lat}, B_{lat}) \quad \text{et} \quad \min(A_{lon}, B_{lon}) \leq P_{lon} \leq \max(A_{lon}, B_{lon})$$

(tolérance 10^{-9}).

Analyse des résultats de l'analyse de mutation

L'exécution de PIT sur la classe `DistanceCalcEuclidean` a permis de comparer le comportement du code avant et après l'ajout des nouveaux tests unitaires. Les scores globaux montrent une amélioration significative du *line coverage* et du *mutation coverage*, ce qui indique que les tests couvrent désormais un plus grand nombre de chemins d'exécution et détectent davantage de fautes injectées automatiquement.

Mesure	**Sans tests**	**Avec les nouveaux tests**	**Différence**
Line Coverage	62% (21/34)	79% (27/34)	+17%
Mutation Coverage	48% (27/56)	68% (38/56)	+20%
Test Strength	59% (27/46)	70% (38/54)	+11%

Table 1: Résumé des scores PIT

Mutants nouvellement détectés

Ligne	**Mutation**	**Statut avant**
34	1. replaced double return with 0.0d for com/graphhopper/util/DistanceCalcEuclidean::calcDist	NO_COVERAGE
64	1. Replaced double subtraction with addition	SURVIVED Covering tests
65	1. Replaced double subtraction with addition	SURVIVED Covering tests
66	1. replaced double return with 0.0d for com/graphhopper/util/DistanceCalcEuclidean::calcNormalizedDist	SURVIVED Covering tests
98	1. Replaced double subtraction with addition	NO_COVERAGE
99	1. Replaced double multiplication with division	NO_COVERAGE
99	2. Replaced double addition with subtraction	NO_COVERAGE
100	1. Replaced double multiplication with division	NO_COVERAGE
100	2. Replaced double addition with subtraction	NO_COVERAGE
101	1. replaced return value with null for com/graphhopper/util/DistanceCalcEuclidean::intermediatePoint	NO_COVERAGE

Table 2: Liste des 11 mutants nouvellement détectés grâce aux nouveaux tests

Mutants détectés

Sur les 56 mutants générés par PIT, 11 nouveaux mutants ont été détectés grâce aux nouveaux tests. La majorité de ces mutants concernent des remplacements d'opérateurs arithmétiques dans les méthodes `calcDist()` et `calcNormalizedDist()` (par exemple, inversion d'addition/soustraction ou modification du retour). Ces résultats montrent que les nouveaux cas de test ciblant les calculs intermédiaires et les bornes de l'interpolation ont permis de renforcer la robustesse du code testé.

Mutants survivants

Trois mutants sont restés survivants, tous situés dans la méthode `calcNormalizedDist()`, reproduite ci-dessous :

```
@Override
public double calcNormalizedDist(double fromY, double fromX, double toY, double toX) {
    double dX = fromX - toX;    // ligne 64
    double dY = fromY - toY;    // ligne 65
    return dX * dX + dY * dY;   // ligne 66
}
```

Ces mutants correspondent à des remplacements de la soustraction par une addition (`fromX + toX`, `fromY + toY`) ou à une modification de l'expression arithmétique finale. Ils ont survécu car les tests actuels n'appellent pas directement la méthode `calcNormalizedDist()` : elle est uniquement sollicitée de manière indirecte via `calcDist()`. Ainsi, PIT n'a pas pu observer d'effet visible de ces mutations sur les résultats vérifiés par les tests. Autrement dit, la fonction a été exécutée dans le cadre de tests plus globaux, mais aucune assertion ne portait spécifiquement sur la valeur renvoyée par cette méthode.

Mutants non couverts (NO_COVERAGE)

Plusieurs mutants répertoriés dans le rapport présentent le statut `NO_COVERAGE`. Cela signifie que les lignes correspondantes n'ont tout simplement pas été exécutées par les tests existants, il ne s'agit pas d'un échec des assertions, mais d'une absence totale de couverture du code. Ces mutants sont généralement situés dans des portions de code auxiliaires ou rarement utilisées (par exemple, des méthodes de normalisation ou des branches alternatives non atteintes). Leur présence explique la différence restante entre la couverture de lignes (79%) et la couverture de mutation (68%).

Interprétation globale

En résumé :

- Les nouveaux tests ont permis de tuer plusieurs mutants dans les calculs de distance et d'interpolation, améliorant le *mutation coverage* de +20 %.
- Les mutants restés survivants proviennent de la méthode `calcNormalizedDist()`, non testée directement, ce qui empêche PIT de détecter les effets des fautes injectées.
- Les mutants marqués `NO_COVERAGE` sont dus à des lignes non exécutées, et non à un manque de précision des assertions.

Ces résultats montrent que la suite de tests actuelle améliore nettement la robustesse du code, tout en laissant quelques zones non couvertes qui expliquent la survie partielle de certains mutants.

Test avec Java Faker - `testCalcNormalizedEdgeDistance_RandomValues`

Intention : Vérifier la robustesse et la stabilité numérique de la méthode `calcNormalizedEdgeDistance()` lorsque celle-ci est appliquée à des coordonnées aléatoires et réalistes, générées automatiquement à l'aide de la librairie `java-faker`.

Motivation des données : Les valeurs aléatoires simulées par `Faker` reproduisent des coordonnées géographiques plausibles (latitudes entre -90 et 90, longitudes entre -180 et 180). Ce choix permet de tester le comportement de la méthode sur un large domaine d'entrées, sans se limiter à des cas fixes ou idéalisés.

Oracle : L'oracle repose sur plusieurs invariants mathématiques de la distance euclidienne :

- la distance normalisée (somme des carrés) ne peut jamais être négative ;
- elle ne doit produire ni `NaN` ni valeur infinie ;
- elle doit être exactement nulle lorsque les points d'entrée sont identiques.

Ces propriétés permettent de vérifier la cohérence générale du calcul, même sans connaître la valeur exacte attendue pour chaque jeu de données aléatoire.

Commentaires : Ce test met en valeur l'utilisation de la librairie `java-faker` pour la génération automatisée de données de test. Il renforce la fiabilité globale du module en validant des propriétés fondamentales sur un ensemble d'entrées variées, plutôt que sur des cas unitaires prédéterminés.