IMAC
INGÉNIEUR
ESIPE

# Travaux dirigés C++ n°11

**Informatique**

*—IMAC 2e année—*

---

## Tests unitaires

Les tests unitaires sont des procédures permettant de vérifier le bon fonctionnement d'une partie précise d'un programme. Ils sont souvent pénibles et rébarbatifs à écrire, mais se révèlent néanmoins cruciaux dans le développement de grosses bibliothèques. Ils permettent en effet de vérifier qu'une modification du code (même mineure) n'altère pas le bon fonctionnement global de la bibliothèque.

---

### ▸ Exercice 1. *Installation*

1. *Téléchargez le code disponible sur le site de l'enseignant.*
2. *Les tests unitaires ne font pas partie des bibliothèques standards du* `C++`*, il faut donc choisir un framework dédié à cette tâche. Pour ce TP, nous choisissons la bibliothèque Google Test. À l'aide du fichier* `readme.md` *dans le code source, installez Google Test. Il s'agit de faire une installation sur le système, plutôt qu'une installation locale.*

### ▸ Exercice 2. *Testons les tests*

1. *Commencez par jeter un oeil au code.*
2. *Compilez votre projet à partir du premier* `CmakeList.txt` *associé au projet. Vous obtenez deux exécutables, un pour votre programme et un pour vos tests. Testez les tous les deux.*
3. *L'exécution des tests unitaires génère en sortie une liste de résultats. Il semblerait que le test relatif à l'opérateur + ne passe pas. Debuggez la bibliothèque en conséquence, jusqu'à ce que tous les tests passent.*

▸ **Exercice 3.** ***Vos tests***

*Les tests unitaires sont en fin de compte un programme* `C++` *qui génère des données, applique ces données au code à tester et vérifie que les résultats sont conformes aux attentes. Dans l'exemple du test de l'addition, une possibilité consiste à générer automatiquement deux vecteurs aléatoires, d'en faire la somme et de vérifier que les composantes du vecteur somme sont bien la somme des composantes des vecteurs initiaux.*
*Plus spécifiquement, les tests sont une sorte d'assertion associée à un état. Voici le genre de tests disponibles :*

- `ASSERT_EQ(val1, val2)` *qui correspond à l'assertion* `(val1 == val2)`
- `ASSERT_NE(val1, val2)` *qui correspond à l'assertion* `(val1` $\neq$ `val2)`
- `ASSERT_LT(val1, val2)` *qui correspond à l'assertion* `(val1` $<$ `val2)`
- `ASSERT_LE(val1, val2)` *qui correspond à l'assertion* `(val1` $\leq$ `val2)`
- `ASSERT_GT(val1, val2)` *qui correspond à l'assertion* `(val1` $>$ `val2)`
- `ASSERT_GE(val1, val2)` *qui correspond à l'assertion* `(val1` $\geq$ `val2)`

*Il y a également des assertions spécifiques à certains types :*

- `ASSERT_FLOAT_EQ(expected, actual)`
- `ASSERT_DOUBLE_EQ(expected, actual)`
- `ASSERT_NEAR(expected, actual, absolute_range))`
- `ASSERT_THROW(instruction, exceptionType)`

*Vous pouvez changer le mot clés* `ASSERT` *par* `EXPECT`*, par exemple* `EXPECT_EQ(val1, val2)` *dans le cas où vous souhaitez ne pas arrêter les tests de la fonction test courante. Si le "test raté" est bloquant, pas la peine de continuer :*

```
ASSERT_EQ(my_vector.size(), the_good_size)};
```

*Dans ce cas, pas la peine de continuer, on sait que les tests suivants risquent de ne pas être opérés correctement. Par contre, pour une erreur non bloquante, vous pouvez préférer* `EXPECT`*.*

*En pratique, chaque fonction de la bibliothèque testée doit avoir un ou plusieurs tests. Un test est juste une fonction du genre :*

```
TEST (groupTest, specificTest) {
  EXPECT_EQ (1+1, 2);
}
```

*Ici,* `groupTest` *est un identifiant (sous forme de variable* `C++`*) que vous choisissez et qui définit un sous groupe de vos tests. Par exemple* `VectorDTestConstructor` *sera utilisé pour tous les tests relatifs aux constructeurs de la classe* `VectorD`*. La variable* `specificTest`

*correspond spécifiquement au test réalisé. Par exemple* `defaultConstructor` *si vous testez le constructeur par défaut.*

*À faire :*

1. *En vous inspirant des tests déjà fournis, ajoutez un tests pour la méthode* `dot` *de la classe* `VectorD`*. Ce test fera partie du groupe* `VectorDArithmetic`*. Pour ce faire, vous pouvez par exemple :*
   - *vérifier que le produit scalaire de votre lib fonctionne bien sur des vecteurs que vous connaissez déjà (et donc vous connaissez par ailleurs le résultat de leur produit scalaire).*
   - *générer des vecteurs aléatoires et calculer leur produit scalaire en utilisant une autre méthode que celle employée dans votre lib, puis vérifier que vous trouvez pareil.*
   - *vérifier des propriétés que le produit scalaire doit toujours satisfaire :*
     - *être commutatif*
     - *être défini positif (*$\mathbf{u} \cdot \mathbf{u} \geq 0$*)*
     - *la loi des cosinus :* $\mathbf{u} \cdot \mathbf{v} = \frac{1}{2}(\mathbf{u}^2 + \mathbf{v}^2 - (\mathbf{u} - \mathbf{v})^2)$
2. *Faites également un test pour la méthode* `VectorD operator*(const double value, const VectorD &vec)`*, également dans le groupe* `VectorDArithmetic`*.*
3. *Faites un teste pour le constructeur par recopie, dans le groupe* `VectorDConstructor`*.*
4. *Faites deux tests pour l'*`operator +` *dans le cas de la somme de deux vecteurs de tailles différentes : un qui test le type d'exception levée et l'autre qui teste le message d'erreur. Ces deux tests seront dans le groupe* `VectorDException`*.*