Projet : fichier réponses

Raphaël Vock, Lomàn Vezin

16 mars 2019

1 La classe `Vector3D`

1.1 Conception

(P1.1) Les points et vecteurs de l'espace euclidien sont représentés au moyen de la classe `Vector3D` dont les attributs `x`, `y`, `z` (de type `double`) représentent les coordonnées cartésiennes d'une instance. Celles-ci sont des attributs privés, mais tout le reste est publique. Voici les opérations que nous avons implémentées en premier :

- `getCoords()` est une méthode qui retourne les coordonnées d'une instance via un `std::array<double,3>`.
- `norm()` et `norm2()` retournent la norme euclidienne (resp. norme euclidienne au carré).
- `distance(x,y)` et `distance2(x,y)` sont des méthodes statiques qui retournent la distance euclidienne (resp. distance euclidienne au carré) entre les deux arguments.
- `unitary()` est une méthode qui retourne le vecteur directeur de même sens et de même direction (et lance une exception si l'instance est le vecteur nul ; voir point suivant).
- `is_zero()` est une méthode qui retourne un bool indiquant si l'instance est plus petite (en norme au carré) qu'une (petite) constante `EPSILON` de type `double`.

1.2 Constructeur, constructeur de copie

(P4.1) L'unique constructeur de la classe prend en argument trois `double` (nuls par défaut) et initialise un `Vector3D` avec les coordonnées cartésiennes précisées. En l'absence d'attributs pointeurs, le constructeur de copie minimal par défaut réalise exactement ce que l'on attend, donc **nous n'avons pas défini de constructeur de copie.**

1.3 Coordonnées sphériques

(P4.2) Nous n'avons pas implémenté de constructeur en coordonnées sphériques. Avant tout, l'ajout de ce dernier serait un peu technique au niveau du prototypage de car nous aurions *a priori* deux constructeurs aux protoypes identiques.

Ce n'est pas un problème incontournable ; on pourrait, par exemple, écrire un seul et unique constructeur dont le quatrième argument est un `bool` (`false` par défaut) qui précise si les arguments donnés doivent être compris comme des coordonnées sphériques ou sinon cartésiennes.

Mais le problème fondamental est le suivant : un repère sphérique est intextricablement lié à la donnée d'un point O considéré comme l'origine. Pour écrire un constructeur sphérique sans ambiguité il serait donc nécessaire :

- ou bien de faire un choix canonique de O qui ne changerait pas du début à la fin
- ou bien de préciser O en passant sa valeur (ou éventuellement son adresse) en argument.

Si l'on choisit la première option, l'unique avantage des sphériques serait perdue car on ne serait plus en mesure de choisir l'origine à notre gré pour simplifier, entre autres, des expressions de force (e.g. en choisissant O l'origine d'un champ de force central). Mais la deuxième option serait coûteuse à implémenter et lourde à utiliser pour des avantages qui finalement n'en vaudraient pas le coup.

1.4 Surcharge des opérateurs

(P4.3)

- Nous avons surchargé les opérateurs d'auto-affectation `+=`, `-=`, `*=` correspondant à l'addition (resp. soustraction) vectorielle et la multiplication par un scalaire. Celles-ci retournent le résultat de l'affectation pour optimiser certains calculs par la suite.
- À partir de ces derniers nous avons surchargé les opérateurs binaires `+`, `-`, `*` correspondant aux opérations vectorielles usuelles.
- Les opérateurs binaires `|` et $\wedge$ retournent respectivement le produit scalaire euclidien et le produit vectoriel.
- L'opérateur de comparaison `==` est défini à partir de la méthode `is_zero()` (cf. §1.1). L'opérateur `!=` retourne la négation logique de `==`.
- L'opérateur d'indexage `[]` est surchargé pour permettre un accès direct (mais protégé) à chaque coordonnée sans devoir passer par `getCoords()`. Ainsi `u[0]`, `u[1]`, `u[1]` retournent des *copies* de `x`, `y`, `z` respectivement. Une exception est bien-sûr lancée si l'indice n'est pas `0`, `1` ou `2`.

2 La classe `Particle`

2.1 Facteur gamma, énergie

(P5.1) L'implémentation du facteur gamma et de l'énergie dans la classe `Particle` peut–être accomplie par un attribut ou par une méthode. Chacun présente des avantages et des inconvénients.

- **Sous forme d'attribut :**

⊕ Leur valeur serait calculée une et une seule fois, donc une même grandeur ne serait jamais calculée deux fois inutilement.

⊖ Leur valeur devrait être calculée systématiquement à chaque mise-à-jour de l'instance. Donc eventuellement même si elle ne servira jamais.

⊖ Rajouter des attributs, c'est augmenter le poids des instances.

⊖ Conceptuellement c'est un non sens ; voir ci-dessous.

— **Sous forme de méthode :**

⊕ La masse et la vitesse sont des grandeurs *essentielles* d'une particule tandis que le gamma et l'énergie en découlent via des fonctions mathématiques. Conceptuellement, les attributs d'une classe devraient devraient être reservées aux variables *libres* et les méthodes aux variables *liées*.

⊕ Stockées sous forme de méthodes, on calcule leur valeur précisément lorsqu'on en a besoin, donc on ne stocke jamais d'information redondante.

⊖ . . . quitte à effectuer inutilement plusieurs fois le même calcul.

Certes, il s'agit de calculs simples avec des `double` donc les arguments de complexité et de poids sont probablement négligeables pour un ordinateur moderne. Moyennant le pour et le contre, et en se focalisant surtout sur la clarté conceptuelle, **nous avons finalement opté pour des méthodes.**