

Projet : fichier réponses

Raphaël Vock, Lomàn Vezin

22 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 unitaire 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 de classe `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 pro-

totypage de car nous aurions *a priori* deux constructeurs aux prototypes 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 ambiguïté 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 le passant comme 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, selon nous, 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 `^` 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()` (cf. §1.1). Ainsi `u[0]`, `u[1]`, `u[2]` retourne une *copie* 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 valeur ne serait jamais calculée deux fois inutilement.
- ⊖ Leur valeur devrait être calculée systématiquement à chaque mise-à-jour de l'instance. Donc éventuellement 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 *essentiels* d'une particule tandis que le gamma et l'énergie en découlent via des fonctions mathématiques. Conceptuellement, les attributs d'une classe devraient être réservés 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 seront probablement négligeables à l'échelle de l'exécutable final. Moyennant le pour et le contre, et en se focalisant surtout sur la clarté conceptuelle, **nous avons opté pour des méthodes.**

3 Premiers éléments

3.1 Les éléments en général

(P6.1) Dans la hiérarchie des classes représentant les divers éléments de l'accélérateur, nous avons mis comme nœud une classe abstraite nommée `Element` dont les attributs représentent les paramètres purement géométriques (points d'entrée et de sortie, rayon, paramètre de courbure) et relationnels (l'adresse de l'élément suivant, une liste des adresses des `Particle` contenues dans l'instance) des éléments. Nous considérons que tous les éléments sont *a priori* courbes, en sachant qu'en prenant une courbure nulle l'instance dégénère en un élément droit. Nous définissons entre autres une méthode virtuelle pure `add_lorentz_force(Particle &p, double dt)` pour mettre en œuvre les interactions élément-particule ; son comportement est relégué aux sous-classes suivantes :

- `Electric_element` est une sous-classe d'`Element` avec un attribut supplémentaire, un champ de électrique `E` (dont nous discuterons de la représentation au passage suivant). Ainsi nous spécialisons la méthode `add_lorentz_force(Particle &p)` en évaluant le champ vectoriel `E` au point où se trouve `p` et en incrémentant la force résultante de `p` via la relation $\mathbf{F} = q\mathbf{E}$.
- `Magnetic_element`, quant à elle, a comme attribut supplémentaire un champ magnétique `B`. Dans ce cas nous spécialisons `add_lorentz_force(Particle &p)` en évaluant `B` en `p` puis en appliquant la méthode relativiste

`Particle::add_magnetic_force(const Vector3D &B, double dt)`

déclarée auparavant dans `particle.h`.

Ces deux sous-classes peuvent ensuite être spécifiées en des sous-sous-classes vraiment concrètes (e.g. dipôle) sans avoir à dupliquer à chaque fois le code correspondant à l'ajout de force sur les particules affectées.

(P6.2) En ce qui concerne la représentation des champs nous avons profité du paradigme fonctionnel que propose le C++11 : notamment avec `std::function` et le lambda-calcul. En bref : le template `std::function` (fourni par l'en-tête `<functional>`) permet de manipuler des objets de type fonction sans indirection, et les lambdas permettent d'instancier des fonctions génériques, *inline*, et avec la possibilité "d'attraper" des variables dans la portée. Cela nous procure une représentation puissante et relativement légère des champs vectoriels (et en particulier des champs électriques ou magnétiques) :

```
#include <functional>
// déclare une classe pour représenter les champs vectoriels :
typedef std::function<Vector3D(const Vector3D &)> VectorField;

const Vector3D X_0(1.0, 2.0, 0.0);

// initialise un champ vectoriel constant via un lambda :
VectorField X = [&](Vector3D){ return X_0; }
```

Bien entendu, on peut utiliser des lambdas pour instancier des champs vectoriels non constants et aussi compliqués que l'on souhaite.

Muni de cette représentation, l'implémentation de sous-classes spécifiques d'`Element` devient très concise. Il n'y a aucune méthode à redéfinir car il suffit de mettre en attribut les éventuels paramètres spécifiques de l'élément puis d'écrire le calcul du champ électrique (resp. magnétique) dans le corps d'un lambda représentant le champ vectoriel que l'on passe en argument (parmi d'autres) au constructeur de `Electric_element` (resp. `Magnetic_element`).

Illustrons ceci par la définition simplifiée de la classe dipôle (qui, rappelons-le, a

un champ magnétique vertical constant et d'amplitude donnée par un paramètre) :

```
class Dipole : public Magnetic_field {
private:
    double B_0; // amplitude (signée) du champ constant
public:
    Dipole(parametres_generiques, double amplitude) :
    Magnetic_element( // constructeur de la super-classe
        attributs_generiques(parametres_generiques),
        // affectation de B via un lambda :
        B([&](Vector3D){ return Vector3D(0.0, 0.0, amplitude); }
    )
    B_0(amplitude){}
};
```

(On conservera toujours la donnée “brute” des paramètres sous forme d’attributs de sorte à pouvoir les afficher avec les instances lors de l’affichage texte.)

(P6.3) Afin d’alléger par la suite le corps de certaines méthodes, **nous avons implémenté le calcul du centre de courbure** avec la méthode `Element::center(void)`. Celui-ci n’a de sens que si la courbure est non nulle (i.e. l’élément est courbe) donc une exception est lancée dans le cas contraire.