Projet : fichier réponses

Raphaël Vock, Lomàn Vezin

24 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 deux `Vector3D` passés en argument.
- `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 booléen 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 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 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 $\wedge$ 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()` (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 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 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 seront probablement négligeables à l'échelle de l'executable 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'addresse de l'élément suivant, une liste des addresses des `Particle` contenues dans l'instance) des éléments. Nous considérons les éléments comme *a priori* courbes, en sachant qu'on peut prendre une courbure nulle et 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 intéractions élément-particule ; son comportement est délégué 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 spécalisons 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 spécalisons `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`.

(P6.2) Comment représenter les champs électiques/magnétiques `B` et `E` ?

La solution évidente est par des méthodes virtuelles pures, que l'on spécialise pour chaque sous-classe concrète d'`Element` en fonction de leur nature. Mais supposons que nous ayions envie de représenter les champs dans la sortie graphique, par exemple avec des lignes de champ. On rajouterait alors aux méthodes dessin des sous-classes d'`Element` le code nécessaire au traçage des lignes de champs en faisant référence aux méthodes `B` ou `E`.

Cette approche *ad hoc* et très peu généralisable. Si l'on souhaite en plus visualiser d'autres champs, pas nécessairement liés aux éléments, par exemple le champ magnétique généré par une particule, on serait alors contraint à copier-coller le code lié à la représentation des lignes de champs. Ce qu'on a envie de faire, c'est déléguer la partie liée au dessin des champs de force à une fonction externe qui prend en argument un champ vectoriel et se charge de le dessiner. Mais dans le scénario que nous décrivons, les champs vectoriels sont des méthodes, donc ne peuvent pas être passés en argument !

Force est de constater que les champs vectoriels `B` et `E`, bien qu'ils aient la vocation de se comporter comme des fonctions, apparaissent conceptuellement comme des attributs : ils contiennent *de l'information qui caractérise l'instance*. Par ailleurs ce sont des *objets fonctionnels purs* : en les évaluant en un point on ne modifie pas l'instance. On a donc très envie de les représenter comme des *attributs de type fonctionnel* et non comme des méthodes.

Ceci est possible grâce au paradigme fonctionnel que propose la norme C++11, notamment avec `std::function` et la notation lambda. Pour rappel :

- Le template `std::function` (fourni par l'en-tête `<functional>`) permet de définir des variables fonctionnelles dites "de première classe", c'est-à-dire qu'on peut manier comme des variables quelconques.
- La notation lambda permet d'instancier des fonctions de première classe génériques, *inline*, et avec la possibilité "d'attraper" des variables dans la portée.

Ensemble, ils nous procurent un puissant moyen de représenter des fonctions pures de type `Vector3D` $\mapsto$ `Vector3D` :

```
#include <functional>
// déclare une classe pour représenter les fonctions vectorielles :
typedef std::function<Vector3D(const Vector3D &)> vector_function;
const Vector3D X_0(1.0, 2.0, 0.0);
// initialise une fonction vectorielle constante via un lambda :
vector_function X = [](Vector3D &){ return X_0; }
```

Pour représenter l'*objet* correspondant au champ vectoriel nous voulons en plus qu'il ait une méthode permettant de le dessiner. On va donc encapsuler un `vector_function` dans une sous-classe de la class abstraite `Drawable`. Puisque le C++ donne la possibilité

de surcharger l'opérateur d'appel, on peut conserver son interface "fonctionnelle" malgré l'encapsulation :

```
class VectorField : public Drawable {
   private:
      vector_function F;
   public:
      VectorField(vector_function f) : F(f){}
      // surcharge de l'opérateur d'appel :
      Vector3D operator()(const Vector3D &v) const{ return F(v); }
      // prototype de la méthode pour dessiner les lignes de champ :
      virtual void draw(void) override;
};
```

On peut enfin définir les sous-classes `Electric_field` et `Magnetic_field` en rajoutant un attribut de type `VectorField` correspondant au champ électrique et magnétique respectivement. Munis de cette représentation, l'implémentation des sous-classes spécifiques d'`Element` devient très concise. À l'encontre de la première approche, nous n'avons pas à redéfinir de méthode virtuelle ; il suffit d'ajouter 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 représentant les dipôles (qui, rappelons-le, ont un champ magnétique vertical constant, 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 du champ via un lambda :
         B([](Vector3D &){ return Vector3D(0.0, 0.0, amplitude); }
      ),
      B_0(amplitude){}
};
```

(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.