Projet: fichier réponses

Raphaël Vock, Lomàn Vezin

16 mai 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.

— **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 éventuellement effectuer plusieurs fois le même calcul.

A priori ses grandeurs seront accédés plusieurs fois par mise à jour de l'accélérateur. C'est pourquoi nous avons décidé de les implémenter sous forme d'attribut.

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 `apply_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 spécialisons la méthode `apply_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`.

(P6.2) Comment représenter les champs magnétiques ?

Comme expliqué ci-dessus, les champs magnétiques sont des méthodes virtuelles pures qui prennent en argument un point de l'espace, et retournent la valeur du champ vectoriel en se point.

(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.

(P6.4) La relation d'appartenance entre une particule et l'élément dans lequel elle se trouve se manifeste par un attribut `Element* current_element`. Ceci implique directement qu'une particule ne peut appartenir qu'à un et un seul élément.

4 La classe `Accelerator`

4.1 Généralités

(P7.1) Un accélérateur *est* une collection d'éléments donc nous faisons hériter la classe `Accelerator` de `std::vector<std::unique_ptr<Element>>`. Nous passons par des `unique_ptr` pour obtenir un comportement polymorphe des éléments. Par ailleurs, cette héritage est `private` pour rester en accord avec la norme C++.

Énumérons les attributs et méthodes principales de cette classe :

- L'attribut `std::vector<std::unique_ptr<Particle>> particles` stockera les particules contenues dans l'accélérateur. Nous utilisons à nouveau des `unique_ptr` pour pouvoir avoir un comportement polymorphe si le besoin se présente.
- L'attribut `const Vector3D origin` définit un point de repère sur l'accélérateur et servira à faciliter la construction de l'accélérateur.
- L'attribut `double length` décrit la longueur géométrique de l'accélérateur. Celui-ci aidera à définir un système de coordonnées curviligne global sur la longeur de l'accélérateur.
- La méthode `void addParticle(const &Particle)` permet d'insérer une particule dans l'accélérateur *tout en conservant la nature de l'instance s'il s'agit d'une sous-classe.*
- Pour chaque sous-classe concrète `E` de `Element` nous avons une méthode `addE` permettant d'ajouter une instance de cette sous-classe à l'accélérateur. Cette méthode prend en arguments un point de l'espace ainsi que les divers paramètres géométriques et physiques de l'élément en question. L'unique point correspond au point de sortie de l'élément ; le point d'entrée est déduit comme étant le point de sortie du précédent (ou sinon `origin` s'il n'y en a pas).

(P7.2) Nous avons supprimé le constructeur de copie par défaut ainsi que l'opérateur d'affféctation = de la classe `Accelerator` pour garantir l'intégrité des ses données. Cette classe contient en effet une liste de pointeurs intelligents sur des éléments `vector<unique_ptr><Element> >`. Bien que nous puissions effectuer une telle copie en faisant une copie profonde des éléments de la liste, nous considérons que cette solution a peu de sens et nous n'en aurions quand bien même pas l'utilité.

4.2 Révision des éléments

(P8.1) La méthode `has_collided()` de la classe `Element` doit se comporter de façon différente si elle est appelée sur un élément courbe ou sur un élément droit. Dans une première approche il s'agirait donc en terme de programation orientée objet d'une méthode virtuelle. Toutefois afin d'éviter une duplication inutile du code et par soucis de conception, nous avons fais le choix de représenter les éléments droits comme des éléments courbe de courbure nulle, en remarquant que toutes les méthodes appliquées à ces éléments sont les mêmes que celles appliquées aux élements courbes lorsque la courbure tend vers 0.

(P8.2) Si la méthode `has_collided()` était virtuelle, on s'attendrait à ce qu'ell soit virtuelle pure puisqu'on ne saurait la définir dans le cas d'un élément qui ne soit ni courbe si droit, faisant de la classe `Element` une classe abstraite. Ce n'est cependant pas le cas dans notre conception des éléments puisque nous voyons les éléments droits comme des élements de courbure nulle. La classe `Element` n'a pas besoin d'être abstraite.

5 Conception du système

5.1 Objets dessinables

(P9.1) La méthode `draw()` a un comportement différent selon le `Canvas` qu'elle doit dessiner, il s'agit donc d'une méthode virtuelle.

(P9.2) La classe `Accelerator` est, par héritage privé, une collection d'éléments et possède une collection de `Particle` plus tard remplacées par des `Beam`, les faisceaux. On souhaite que la méthode `draw()` s'applique à chacun des éléments et chacune des particules selon les caractéristiques propres à l'instance. Nous avons donc besoin, dans un cadre de programation orientée objet, de *polymorphisme* et plus précisément de résolution dynamique des liens. La classe `Accelerator` doit donc être une collection de pointeurs sur des `Element` et contenir une collection de pointeurs sur des `Beam`.

(P9.3) Nous devons à présent être vigilants vis à vis de la copie puisque notre classe

contient des pointeurs. Une solution serait de redéfinir le constructeur de copie ainsi que l'opérateur d'affectation `=` en implémentant une copie *profonde* et non de surface afin de garantir l'intégrité des données. Toutefois nous ne souhaitons pas copier d'accélerateur, ce qui aurait d'ailleurs peu d'utilité et de sens, nous supprimons simplement ce constructeur et l'opérateur `=` avec le mot réservé `delete`. L'utilisation de pointeurs nous force de plus à faire attention au destructeur de la classe. Puisque nous utilisons des pointeurs intelligents nous n'avons pas besoin de nous préocupper de la gestion de mémoire (`delete`) car celle ci s'effectue automatiquement.

6 La classe `Beam`

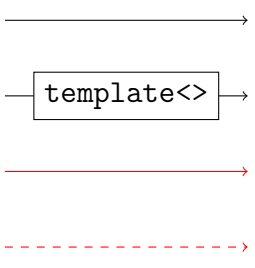
6.1 Conception

(P11.1) Un `Beam`, ou faisceau, est par héritage `private`, une collection de `Particle`. Ses attributs, constants, sont une référence sur la particule modèle du faisceau, le nombre de macroparticules crées `N`, le facteur λ de ces macroparticules ainsi qu'un pointeur sur l'accélérateur dans lequel l'instance évolue. Puisque l'émittance, radiale et verticale, ainsi que les six coefficients des ellipses de phases et l'énergie moyenne du faisceau seront recalculés fréquemment nous avons décidé de les implementer comme méthodes.

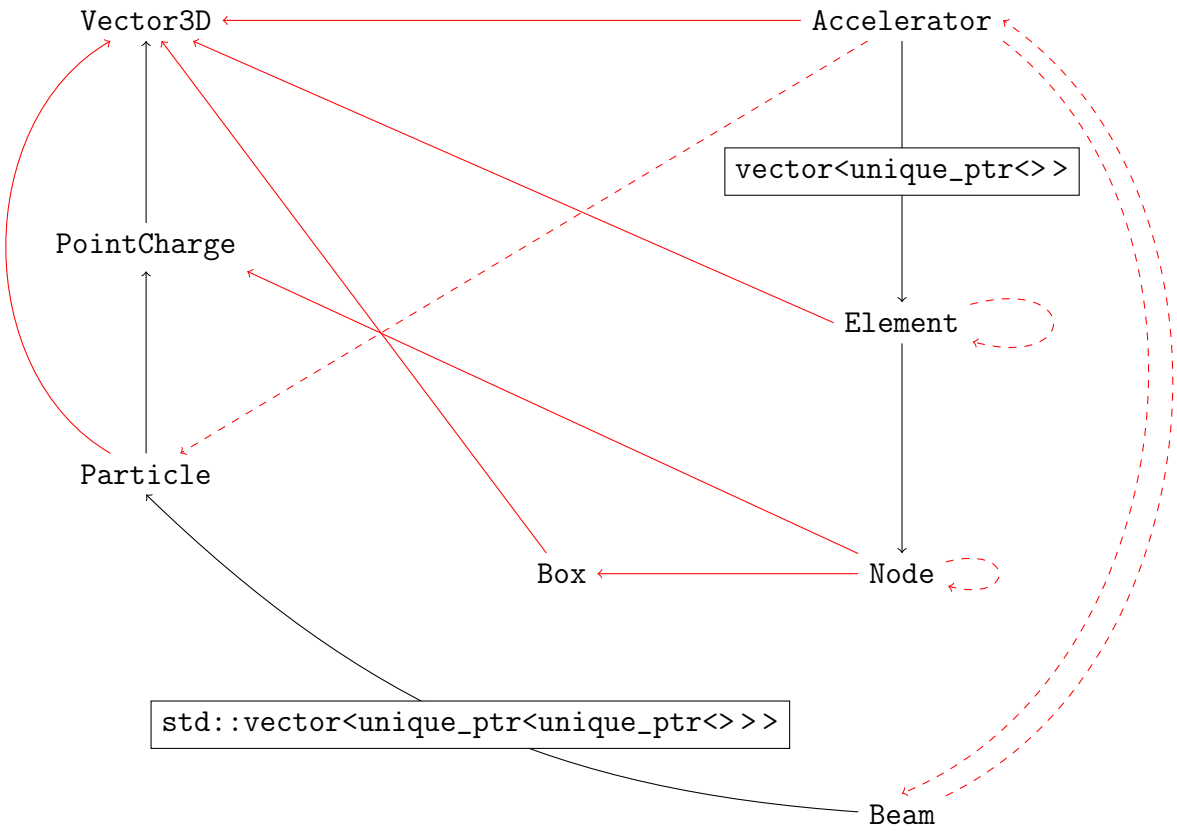
7 Hiérarchie des classes

Ci-dessous deux schémas représentant les hiérarchies relationnelles des classes.

7.1 Légende



7.2 Des classes liées aux objets de la simulation



7.3 Des classes liées à la représentation de la simulation