Réponses aux questions

J. Odersky C. Vázquez

29 mai 2011

Question P1.1

Un vecteur est représenté par la classe 'Vector3D'. Cette classe comprend trois champs privés du type double : x, y et z correspondant aux composantes du vecteur. Ces champs peuvent être accédés respectivement par les méthodes publiques getX(), getY() et getZ(). Les méthodes "opérateurs" sur les vecteurs (par exemple l'addition, la norme, etc...) sont toutes publiques.

Ces méthodes s'appuient entre-elles, par exemple la méthode 'opposée' retourne le vecteur multiplié par moins un et la méthode 'soustraction' additionne l'opposé. C'est à partir de cette idee que nous avons aussi implémenté une norme au carré (=produit scalaire du vecteur avec lui-même) et une norm (=racine de la norme au carré). Ces appels consécutifs diminuent la performance d'une façon minimale mais évitent un duplicage de code considérable.

Un vecteur est complètement invariable. C'est-à-dire qu'une fois un vecteur initialisé, on ne peut plus changer ses composantes (pas de méthodes 'set'). De même, tous les opérations internes du sens mathématique (qui renvoyent un vecteur), renvoyent une nouvelle instance d'un vecteur. En aucun cas l'instance d'un vecteur n'est modifiée ! Ceci facilite énormément le raisonement sur toute variable de type vecteur, c'est-à-dire qu'on peut toujours être sûr que personne n'a modifié le vecteur sur lequel on travaille. De plus, l'invariance d'un vecteur paraît naturelle, comme celle d'un nombre réel.

Quelques vecteurs remarquables sont définis comme variables statiques constantes. Parmi ceux-ci notamment le vecteur nulle (Null) et les vecteurs unitaires i, j, k selon respectivement les axes x, y et z.

Pour contourner la perte de performance due à l'immutabilité des vecteurs, nous avons rajouté un classe 'MutableVector3D'. Cette classe représente des vecteurs mutables lesquelles peuvent changer leurs coordonnées. Ces vecteurs sont presque exclusivement utilisés pour représenter des champs qui varient souvent tel que la position d'une particule.

Question P3.1

Nous n'avons pas rajouté un constructeur de copie. Comme la classe 'Vector3D' est invariable et ne contient pas de pointeurs ou références sur d'autres objets mutables, elle n'a pas d'état et donc l'utilisation du constructeur de copie par défaut suffit.

En revanche la classe 'MutableVector3D' contient un constructeur de copie qui copie les champs d'un vecteur donné.

Question P3.2

Lorsque l'on décide d'implémenter le constructeur par défaut (qui crée un vecteur nul) et le constructeur par coordonnés cartésiennes dans une seule et même méthode, il se peut que lorsque l'on crée une nouvelle instance de Vector3D sans l'initialiser (en laissant donc le soin au constructeur par défaut de le faire), on se retrouve avec un vecteur nul à un endroit où il vaudrait mieux ne pas en avoir. Par conséquent, en séparant ces deux méthodes, le programmeur est amené à penser dès l'instanciation à quelle fin il crée son objet.

Question P3.3

a

En ajoutant un constructeur par coordonnées sphériques, les attributs de la classe ne devraient pas forcémant être changées. Il serait toute à fait envisageable, de garder les coordonnées carthésiennes comme attributs et de convertir les coordonnées sphériques avec le constructeur. L'accès et la modification des coordonnées shpériques pourrait égelement se faire par un méthode convertissant les deux types de coordonnées.

b

La surcharge serait une difficulté majeur pour créer un constructeur par coordonnées sphériques. Etant donné qu'un tel constructeur prendrait comme paramètres deux angles et une longueur, représentés par trois doubles, il serait en conflit avec le constructeur de coordonnées carthésiennes. Il serait alors impossible d'avoir les deux constructeurs dans une classe.

Néanmoins, une solution alternative serait d'implémenter une méthode statique ("factory method") qui prendrait comme paramètres des coordonnées sphériques et qui renverait un vecteur ayant des coordonnées carthésiennes équivalentes (par exemple `Vector3D Vector3D::fromSpherical(double phi, double theta, double r)`).

Question P3.4

La méthode 'affiche()' d'un vecteur a été implémenté sous forme de l'opérateur '<<' de 'std : :ostream'. La méthode 'compare' est équivalent à l'opérateur '==='.

Question P5.1

Les membres d'une particule représentant le facteur gammaγ et l'énergie$E$ peuvent être implémentés soit sous forme d'attributs soit sous forme de méthodes. Il y a des avantages et inconvénients pour chaque forme.

L'avantage d'un attribut est que son accès est très rapide et ne prend (presque) pas de temps de calcul. Par contre, si la valeur d'un attribut est relié logiquement à la valeur d'un autre attribut et que ce dernier est modifié, il faudra manuellement changer le premier. Par exemple, le facteur gamma étant défini par :

$$\gamma = \frac{1}{\sqrt{1 - \left(\frac{v}{c}\right)^2}}$$

si il est défini comme variable, il faudrait le mettre à jour à chaque fois que la vitesse change.

Contrairement à un attribut, une méthode est évaluée à chaque fois qu'on l'appelle. Ceci a l'avantage que si le résultat d'une méthode dépend d'une variable, la variable pourra être modifiée sans autre considération. Ainsi, si le facteur gamma est une méthode, une mise à jour de la vitesse pourra être effectuée directement sans explicitement changer γ.

Dans le cas de notre projet, nous avons décidés d'implémenter l'énergie sous forme de méthode et le facteur gamma sous forme d'attribut. Ceci pour plusieurs raisons :

1. Au cours de la simulation, le facteur gamma est accédé autant de fois que la vitesse, il est donc plus rapide de n'evoluer une racine qu'une seule fois pour un pas de temps.
2. L'énergie est accédée beaucoup plus rarement que la vitesse mais dépend quand même de cette dernière, ainsi nous avons décidé de l'implémenter sous forme de méthode.

Question P6.1

Pour représenter et organiser les éléments, nous avons créé une classe abtsraite 'Element'. Celle-ci contient tous les attributs généraux d'un élément. De plus, elle contient des méthodes virtuelles pures tels que la détermination si une particule a heurté le bord ou est passé au suivant.

Dans la hierarchie en dessous de 'Element' se trouvent deux classes representants les élements droits et courbes. Ceux-ci implémentent tout méthode relatif à la géométrie d'un élément (p.ex. les méthodes mentionés ci-dessus).

Finalement, en dessous de ces classes se trouvent les implémentations concrètes telles que les quadripoles, dipoles, etc...

Nous n'avons pas faits de classes séparés pour des élements générants un champ électrique ou magnétique car nous considérons qu'un élément ne générant pas de champ *possède* un champ nul. (voir question P6.2)

Question P6.2

Les champs magnétiques et électriques sont représentés par les méthodes 'magneticFieldAt' et 'electricFieldAt'. Ces méthodes prennent comme paramètre une position et renvoyent un vecteur représentant le champ à cette position.

Ceci représente concrètement les lignes de champs à un endroit donné.

En effet, en physique on représente les lignes de champ par des lignes continues. Or dans ce projet, cela ne nous intéresse pas de représenter ces lignes. Puisque les seules interactions de ces champs avec les particules invoquent des vecteurs représentant l'intensité et la direction du champ à l'endroit où se trouve la particule par rapport à l'`Element`, on n'a besoin que d'un `Vector3D`.

Question P6.3

Nous représentons le centre de courbure `curvatureCenter` à l'aide d'un attribut du type `Vector3D` que l'on instancie indirectement en fournissant la courbure `k` au constructeur des éléments courbes.

En effet, le centre de courbure d'une courbe est défini d'une manière unique si nous avons :

- une position d'entrée `entry` ;
- une position de sortie `exit` ;
- une courbure `k` (donc un rayon de courbure `curvatureRadius`) avec son signe ! ;
- une convention à l'aide de laquelle nous savons, en fonction du signe du rayon, de quel côté de l'Element la courbe se courbe. [1]

Question P6.4

Afin qu'une particule soit dans un et un seul `Element` à la fois, nous avons décidé de rajouter un attribut privé (un pointeur) à chaque particule, qui soit responsable de lui indiquer dans quel `Element` elle se trouve. Elle ne peut se retrouver dans deux `Element`s. C'est un moyen simple de coller une étiquette 'Où je suis ?' sur une particule, et on pourra facilement vérifier qu'à chaque évolution du système, notre particule ne soit que dans un et un seul `Element` à la fois. Cela modifie la classe `Particule`.

Question P7.1

La classe `Accelerator` un accélerateur. Elle contient une collection de particules et d'élements (en attributs). De plus, nous avons décidé qu'un accélérateur sera entièrement responsable de la gestion de ses particules et elements. Cela implique que lors de l'ajout d'un de ces derniers, il sera copié.

Cette copie entraîne d'eutres conséquences tel que le rajout d'un méthode 'clone' dans les éléments et particules.

Question P7.2

Le constructeur de copie privé : il doit servir à empêcher tout appel au constructeur de copie par défaut, puisqu'il ne fait rien, on ne pourra simplement pas faire de copie d'`Accelerator`, donc pas de passage par valeur.

L'opérateur '=' : de même que pour le constructeur de copie, sa déclaration en privé empêche la copie d'un accélérateur vers un autre.

1. Les éléments courbes auront de plus une courbure k, constante (l'inverse du rayon de courbure), dont le signe indique le sens de courbure par rapport à l'orientation donnée par l'opposé de l'axe vertical (noté $-\vec{e^3}$) .

Ces restrictions sont présents pour éviter des problèmes de responsabilités d'allocation et de délocation de mémoire (car un accélérateur contient beaucoup de pointeurs d'éléments et de particules qui eux-mêmes pointent vers des particules). Mais aussi d'un point de vue logique : comment serait-ce physiquement possible de copier un accélérateur dans un état donné avec tous ses particules ?

Question P8.1

En termes de POO, cela signifie que la méthode `heurte_bord` ('isBeside' dans notre cas) est virtuelle (même virtuelle pure à l'intérieur de la classe `Element`).

Question P8.2

Cela implique donc que la classe `Element` est *abstraite*.

Question P8.3

La méthode 'dessine' est elle aussi virtuelle pure dans la classe dessinable.

Question P9.1

FODO

Question P10.1

Concernant les classes contenant des pointeurs, il s'agit de fixer une manière de gérer les pointeurs, les montrer ou non. Si on choisi de les montrer, alors il faut gérer les `delete` en externe, et cela induit un plus grand risque d'erreurs, plus que si on choisi la deuxième option. Dans ce cas, on peut envisager d'écrire les `new` et les `delete` en interne, ce qui implique qu'on est tout seul à manipuler des pointeurs, ce qui est plus élégant.

Question P10.2

Dans le cadre de la programmation oritentée-objets, on créerait une classe abstraite (p.ex. "Dessinable") contenant la methode dessine() virtuelle pure. Tout objet dessinable hériterait ensuite de cette classe et implémenterait la méthode.

Question P10.3

La reponse à la dernière question n'implique pas de changement de notre code car, depuis le début du projet, nous avons défini la classe "Printable" ("Dessinable" ci-dessus) et tout nos objets dessinables (notamment l'accelerateur, les particules et les elements) heritent de cette dernière et implémentent la méthode virtuelle pure "print()" qui affiche l'objet.

Question P10.4

Voir réponse à question P10.3.

Question P12.1

Comme tous ces propriétés changent pendant l'evolution de la simulation, nous les avons implémentés en tant que méthodes.

Question P14.1

Nous avons créé une classe Color qui implémente les couleurs données en exemple. Elle est en outre dotée de quelques méthodes pratiques (voir dans doxygen).

Aucun objet héritant de la classe Printable n'hérite de cette classe, car nous nous sommes concentrés sur l'interface graphique de GUI dans ce projet.

Question P19.1

Voici l'ensemble de balises (ouvrantes) que nous utilisons (dans l'ordre d'apparition dans le fichier "accelerator.xml") :

- <System>
- <Accelerator>
- <Particle>
- <Position>
- <Mass>
- <Charge>
- <Direction>
- <Energy>
- <FODO>
- <EntryPosition>
- <ExitPosition>
- <SectionRadius>
- <StraightLenght>
- <FocalizingCoefficient>
- <Dipole>
- <Curvature>
- <MagneticField>
- <Quadrupole>
- <StraightElement>

Voici les balises pour les commentaires :

- < ! - -
- - - >

Il n'y a donc pas de balises pour les constantes, car elles font partie d'un namespace commun au projet. Par ailleurs, l'aspect graphique de projet est séparé du reste du projet, ce qui explique qu'il n'y ait pas de balise de type <Camera> ou <Oeil>.