Réponse

## P1.1 :

Les vecteurs 3D ont comme attributs 3 doubles représentants chacune des composantes des coordonnées cartésiennes de l’espace à 3 dimensions. Tout les attributs sont en protected, on a donc créé les trois méthodes get et set en public permettant d’accéder et de modifier les attributs en dehors de la classe ou des sous-classes.

Pour la méthode norme, nous avons décidé de définir deux méthodes : l’une norme\_carre donne la norme au carré en évitant ainsi de faire le calcul de la racine et une norme qui est la racine carré de norme\_carre. Ainsi en fonction des nécessités on peut choisir d’utiliser la racine ou non.

## P3.1 :

Nous n’avons pas ajouté de constructeur de copie car la classe vecteur3D n’est composée que de type de base et sans pointeur. Le constructeur de copie par défaut suffit donc amplement. Il va recopier membre à membre les attributs ce qui nous convient parfaitement. De même pour le destructeur.

## P3.2 :

Si nous avions implémenter un constructeur avec 3 arguments et des valeurs par défaut comme cela : Vecteur3D(double =0, double =0, double =0), cela n’implique aucune erreur, d’avantage ou d’inconvénient au niveau du code. Mais au niveau conceptuel, cela impliquerait qu’il devient possible de construire un vecteur avec seulement un ou deux arguments. Par exemple : Vecteur3D v(3, 2). En effet, il est quelque peu dérangeant de construire un vecteur dans l’espace avec seulement deux coordonnées.

## P3.3 :

A] Dans notre cas (où les attributs sont les coordonnées cartésiennes), il faudrait calculer chaque composante à partir des coordonnées sphériques. Ou redéfinir les attributs comme étant les coordonnées sphériques et pour le constructeur précédent faire les calculs inverses.

B] Le problème viendrait du fait que l’on veut surcharger le constructeur mais en lui passant les même types d’arguments : 3 doubles dans les deux cas. Or ceci est impossible en C++. Le seul moyen, pour différencier deux fonctions ayant les mêmes noms, est les arguments qu’elles prennent. Pour contourner cette difficulté on pourrait utiliser un typedef double Angle. Ce qui en créant avant la fonction deux variable de type Angle permettrait de passer des types différents.

## P3 .4 :

On utilise la surcharge interne de l’opérateur == pour compare et la surcharge externe de l’opérateur << pour affiche.

## P5.1 :

Nous avons préféré implémenter gamma et l’énergie en méthode car ce sont des grandeurs qui dépendent des autres attributs. Ainsi cela libère de l’espace mémoire en évitant de stocker des valeurs inutiles. De plus cela évite a l’utilisateur de ce poser la question de savoir si nous les avons définis comme attributs car il lui suffit d’utiliser les méthode get pour y accéder. Cela est en revanche légèrement plus long à écrire.

## P6.1 :

Nous avons choisi d’implémenter une super classe Element. Celle-ci contient tous les attributs et méthodes communs aux éléments quelconques. Nous avons ensuite définis les classes Droit et Courbe avec leurs spécificités qui héritent de la classe Element. Finalement nous avons programmé la sous-classe Dipole qui hérite de Courbe. De la même manière nous définirons les autres éléments selon leurs caractéristiques.

## P6.2 :

Le champ magnétique des éléments est une méthode qui retourne un vecteur3D. Pour cela les constructeurs demandent l’intensité du champ magnétique sous forme de double qu’il stock dans un attribut protected. La méthode calcule le champ en fonction de l’intensité et (pour les éléments où cela est nécessaire) de la position de la particule concernée.

## P6.3 :

Nous avons représenté le centre de courbure comme une méthode. Car comme pour gamma et energie dans particule, courbure dépend des autres attributs. De plus, comme il est utilisé relativement souvent devoir réécrire la formue peut vite devenir fastidieux.

## P6.4 :

Pour que chaque particule soit dans un seul élément à la fois, nous avons ajouter un attribut à la classe particule : un pointeur sur un élément. De cette façon une particule n’appartient qu’à un seul élément et il suffit simplement de modifier l’élément pointé lorsque la particule change d’élément, c’est à dire lorsque la fonction passe\_au\_suivant retourne « true ».

## P7.1 :

Nous avons représenté l’accélérateur comme un ensemble de particules et d’élément. Pour cela nous avons en attribut un vector de pointeur vers des éléments (pour éviter la surcharge de la mémoire) et de même pour les particules. Ensuite deux méthodes get pour chaque vecteurs l’une qui retourne le ième terme du vecteur et une autre qui retourne le vecteur en entier. De plus une méthode pour chaque vecteur permettant d’ajouter un terme à chacun ou de vider le vecteur. Enfin un constructeur par défaut initialisant les deux vecteurs à vide.

## P7.2 :

Le fait de déclarer ces deux méthodes en privé empêche leur utilisation par l’utilisateur de la classe. Cela permet d’éviter le risque de surcharge de la mémoire puisqu’à terme, l’accélérateur contiendra un nombre important d’objet et si l’on cherchait à le copier cela prendrait certainement du temps mais surtout beaucoup de mémoire pour rien. Ainsi cela évite les éventuelles erreurs d’étourderie.

## P8.1 :

La méthode  heurte\_bord est une méthode abstraite (ou virtuelle pure).

## P8.2 :

Element est une classe abstraite. On ne peut donc pas créer concrètement un simple élément.

## P8.3 :

Dessine est une méthode abstraite.

## P9.1 :

La classe Fodo est une collection hétérogène d’éléments droits (Quadrupôle et Section droite). Le constructeur permet de construire une maille Fodo de base, c’est à dire un quadrupôle focalisant, suivi d’une section droite, d’un quadrupôle défocalisant et pour finir d’une nouvelle section droite. Il est possible de modifier la taille de l’ensemble des mailles Fodo lors de la construction de la première de telle sorte que la suite des quatre éléments soit multiplié (par un entier positif non nul).

## P10.1 :

Les classes contenant des pointeurs sont propices aux erreurs d’allocation de mémoire. Pour éviter un « segmentation fault » lors de la compilation, il faut toujours vérifier trois aspects : le constructeur de copie (profond ou non), le destructeur et l’opérateur égal.

## P10.2 :

La bonne façon d’implémenter cette méthode dessine est d’utiliser le polymorphisme. C’est à dire que les méthodes dessine des classes Element, Droit et Courbe seront virtuelles pures et celles des autres classes héritant d’Element (Quadrupole, Section\_droite, Fodo et Dipole) dessineront chaque élément en tenant compte de leurs propres caractéristiques.

## P10.3 :

Pour que le polymorphisme fonctionne correctement, tous les prototypes des méthodes « affiche » sont précédés du mot clé « virtual » (réellement utile seulement devant le prototype des méthodes virtuelles pures). Et les éventuels arguments (ce n’est pas le cas ici) doivent être passés par référence.

## P10.4 :

On doit donc ajouter une méthode dessine à l’accélérateur qui appelle les méthodes dessine de chaque Element.

## P12.1 :

La particule de référence, lamda et le vecteur de pointeur sur des particules sont les seuls attributs. Le nombre de particule est une méthode qui retourne simplement la taille du vecteur (en fait c’est le nombre de macro particules). L’énergie moyenne, les émittances (verticale et horizontale) et les coefficients d’ellipse (verticaux et horizontaux) peuvent être calculés à partir du vecteur de particules, ce sont donc des méthodes.

## P14.1 :

Les couleurs sont une classe qui est un attribut de la classe Element3D dont héritent tous les éléments représentables en 3D.

## P15.1 :

La complexité des deux algorithmes est en O(n2) puisque nous avons implémenté une deux boucles for imbriquées.