Projet Le Meur Ian et Le Fur Yanis :

**Class Vecteur :**

Convention :

Nous avons pris la convention mathématique des vecteurs d’où la premiere composante d’un Vecteur serait numéroté par 1.

Attributs :

Nous avons décidé de mettre comme seul attribut un vector de double nommé vecteur. On a choisi un vector (au lieu d’un array), parce qu’on nous a demandé d’écrire une classe Vecteur représentant des vecteurs multidimensionnel, et qu’un array nécessiterait une dimension fixe, qui ne conformerait pas avec la méthode augmente et qui, en général, n’est pas aussi bien adapte à la création d’une classe vecteur du a l’obligation de donner une dimension au moment de la déclaration d’un array.

On a discuté si nous devions ajouter d’autres attributs à notre classe, mais au final, tous les autres attributs qu’on a pu imaginer pouvait être retourner comme valeur de retour d’une méthode de la classe (dimension, norme euclidienne, direction).

**Classe Matrice :**

Convention :

Nous avons pris la même convention mathématique que la classe Vecteur sur le fait que le coefficient sur la première ligne et première colonne serait numéroter par 1,1. De plus, nous avons pris la convention de l’algèbre lineaire sur les matrices, donc le premier coefficient représente l’indice de la ligne et le deuxième celui de la colonne.

Attributs :

Après avoir pensé à faire une matrice à partir d’un tableau d’objet Vecteur, nous somme repassés sur un tableau de tableau de double pour la matrice. En effet, construire une matrice comme étant un tableau de vecteur permet de simplifier la surcharge des opérateurs du produit matriciel et par cela rendre le code bien plus simplifié.

Cependant, une matrice comme étant un tableau de vecteur est algébriquement réducteur. De plus cela ne compliquerait d’autant plus le code notamment sur les méthodes d’affichages.

De plus, nous avons décidé de représenter une matrice comme un array d’array de dimension 3 chacun car il était demandé de créer des matrices 3x3 et aucune méthode augmenter ou autre permettant de changer la structure d’une matrice était requise, la construction d’une matrice par un array d’array paraissait donc évidente.

**Fichier Constant :**

But :

Le principal but de ce fichier est de permettre d’éviter la redondance de création de constante dans les autres fichier type matrice, vecteur… En effet, celui-ci permet de créer des constantes globales pour tous les fichier et donc ne plus avoir à recréer plusieurs fois les mêmes constantes mais juste à les appeler dans les fichier ou celles-ci seront demandées.

Nous avons donc créé les constantes suivantes :

1. **precision** : un double qui sera utiliser pour gérer la marge d’erreur des doubles
2. **g** : un double de la constante gravitationnelle
3. **grav** : qui est le vecteur ayant comme dernière composante la constante gravitationnelle.

**Classe Toupie :**

**Attributs :**

Nous avons implémenté les attributs de base qu’aurait une Toupie dans un problème de mécanique c’est-à-dire une masse, un Vecteur position et P-point étant le Vecteur qui a comme composantes les dérivées des composantes de P , plus, des composantes de son tenseur d’inertie qui sont I1 et I3, I2 étant égal à I1 il nous ait donc inutile d’implémenter I2.

Enfin nous avons choisi de faire appel à des objets Vecteurs pour les attributs P et P\_point car c’est comme cela que l’on représenterait ces attributs en physique.

**Méthodes :**  
 Nous avons malheureusement due implémenter des getter et setter qui seront utiliser pour la class intégrateur plus tard. Ceux-ci ne donnent qu’accès aux Vecteurs P et P\_point qui seront utilisé dans la méthode intègre. Nous avons longuement réfléchi au fait de faire des accesseurs ou utiliser des friend seulement ces derniers étant absolument déconseillé nous avons donc décidé d’utiliser des accesseurs.

A propos des méthodes affiche et eq\_mouv : celles-ci sont passé en virtual car une toupie étant un objet général, nous allons donc créer plusieurs types de toupie en utilisant l’héritage c’est-à-dire que les types spécifique de toupie hériteront de la classe toupie car elles sont justement des toupies par contre nous voudrions créer des équations de mouvement et un affichage propre à chaque type de toupie, nous utilisons donc le polymorphisme pour répondre à ce problème.

**//A REVOIR**

Les méthodes clone et copie sont toutes les deux liées. La première servira à faire une copie du pointeur(unique\_ptr) pointant sur la toupie pour ensuite différencier les types de retours dans la méthode copie et permettre donc la covariance qui n’est habituellement impossible avec les unique\_ptr. C’est deux méthodes combinées permettent en fait de faire une copie polymorphique du pointeur qui pointe vers la toupie et qui sera utilisé pour les systemes.

**//METHODE DESSINE A CODER**

**Sous-classe ConeSimple :**

Ce type de Toupie (qui hérite donc de la classe Toupie) représente une toupie de la forme d’un cône. En utilisant le polymorphisme, on crée donc aussi les méthodes eq\_mouv et affiche spécifique à la toupie ConeSimple. De plus, nous avons implémenter une fonction externe à la classe Conesimple qui permet de calculer la masse d’un cône à partir de sa masse volumique, son rayon et sa hauteur. Cela permettra de simplifier l’utilisateur lorsqu’il désirera initialiser une Toupie de type cônique.

**Surcharge Operateur :**

Nous avons décidé de surcharger comme dans les autres classes l’opérateur << qui permet d’afficher les caractéristiques des toupies en utilisant la méthodes affiche de façon simplifiée pour l’utilisateur.

**Classe Intégrateur :**

**Attributs :**

La superclasse Intégrateur n’a aucun attribut car celle-ci ne représente pas réellement un objet concret mais plus une méthode de développement des objets.

En effet, l’intégrateur sert uniquement à faire « évoluer » d’un pas de temps l’objet rentré en argument (ici en particulier des Toupies).

**Particularité :**

La classe intégrateur n’est en fait qu’une classe abstraite car il n’existe pas d’intégrateur « universel » mais plusieurs façon d’intégrer et donc plusieurs intégrateurs différents comme l’intégrateur d’Euler-Cromer ou de Newmark. L’utilisation du polymorphisme semble donc évidente dans ce cas.

**Classe Support à Dessin :**

Cette classe permettra à l’utilisateur de choisir s’il veut afficher la toupie comme une simulation graphique, une suite de coordonées, ou même peut-être autrement. De plus, cette méthode n’est que forcement abstraite car il n’existe pas de support « général ».

**Classe Dessinable :**

Après réflexion nous avons pensé que la classe dessinable était en fait une super classe permettantde représenter tout objet physique dans le Support à Dessin désiré. C’est donc pour cela que la classe Toupie (qui est d'ailleurs pour l'instant la seule classe concernée) hérite de la classe Dessinable car c'est un objet possible d'être représenté graphiquement ou par des données.

**Attributs :**

Il est logique que la classe dessinable n’est comme attribut qu’un pointeur sur un support à dessin et non pas un support à dessin tout court car la classe dessinable ne possède pas un support à dessin mais s’adapte à celui-ci. En effet, il n’existe pas autant de support à dessin que de dessinable mais plusieurs dessinable qui ont le même support à dessin.

**Méthode :**

N’a qu’une seule méthode virtuelle pure qui est la méthode dessine(**A CODER SUR LES TOUPIES**)

**Classe Système :**

On a decide qu’un système contiendra une collection heterogene de toupies (dans notre cas un vector de unique\_ptr sur des toupies). La collection heterogene nous permet d’utiliser le polymorphismes avec les methodes virtuelles des toupies comme l’equation du mouvement et l’affichage.

Un système contient aussi un pointeur sur un integrateur qui lui permet de faire evoluer de facon uniforme les differentes toupie de notre système.