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.

//Le fait qu’on ait choisi des doubles comme composantes est évident : on cherche à utiliser la classe Vecteur pour modéliser des systèmes physiques, ou nos grandeurs peuvent prendre toutes valeurs réelles imaginables. //

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).

//On a considéré le sens de donner une base avec notre vecteur, car les composantes d’un vecteur sont par définition, les éléments obtenu en décomposant un vecteur selon une certaine base. Mais en sachant que le complément mathématique inclut une matrice de passage S pour passer de la base dans le repère du référentiel d’inertie a la base dans le repère situe sur la toupie, on a décidé que ça sera inutile en pratique, et que ça alourdirait beaucoup le code.//

Méthodes :

* **//Vérifier :** comme pour la classe matrice nous avons trouvé qu’il était important de créer une méthode vérifier permettant de corriger l’erreur créé par la manipulation de double et de remplacer une des composante du vecteur en mémoire par zéro si celui-ci est en dessous d’une certaine précision placé dans le fichier constant.cc. De plus cette fonction sera utilisé dans la surcharge de l’operateur +=,\*= et ^=.//
* //Nous avons créé une méthode interne, constante **meme\_dimension** prenant un objet Vecteur en référence constante (pour permettre d’éviter des copies inutiles). Cette méthode retourne un booléen (True si les deux vecteurs ont la même dimension autrement retourne False). De plus, cette méthode est placée en private car l’utilisateur n’en aura nul besoin, celle-ci sert simplement à simplifier le code d’autres méthodes de la classe Vecteur.//
* **//Augmente** : méthode placée en public, cette méthode prend en argument une valeur (un double) et augmente la dimension du vecteur par 1, en affectant la valeur de la dernière composante avec la valeur passe en argument. Cette méthode n’a pas beaucoup de sens mathématique, l’augmentation de la dimension d’un vecteur ne voulant rien dire dans un sens d’algèbre linéaire. Par contre, cette méthode pourrait être utile dans un système physique, par exemple quand on étudie la dynamique d’un système en 2 dimensions (dans une première étude simple, et qu’on passe plus tard à une étude en 3 dimensions plus générale), qui pourrait aussi advenir dans l’étude de la toupie (par exemple en considérant la toupie avec ou sans nutation).//
* **//Set\_coord**: méthode placée en public, cette méthode prend en argument un indice (un size\_t) et une valeur (un double) et affecte à la composante du vecteur a l’indice donnée la valeur passe en argument. Cette méthode nous trouble conceptuellement : on n’arrive pas à trouver une occasion ou l’utilisation possible de la méthode contrebalancerai le cassement de l’encapsulation. On l’a gardé dans le cas où on trouve une utilisation possible plus tard en travaillant avec les toupies, mais dans le cas générale, si l’utilisateur de la classe Vecteur pense devoir change une certaine composante d’une instance, il devrait soit créé un nouveau Vecteur, car le vecteur est différent mathématiquement, ou il s’est trompe dans l’initialisation du vecteur.//
* **//Get\_coord :** méthode placée en public, cette méthode prend en argument un size\_t et retourne à l’utilisateur la composante de dimension correspondante au nombre rentré en argument. Cette méthode casse un peu l’encapsulation de la classe, mais elle est nécessaire pour la multiplication matrice-vecteur, et pourrait aussi être utile pour connaitre une certaine composante a un certain moment sans tous les autre détails (mais il faut aussi reconnaitre que cela peut se faire en projetant le vecteur sur l’axe voulu, i.e. en prenant le produit scalaire du vecteur avec le vecteur de base de la composante considère). //
* **//Constructeurs :** Nous avons implémentés quatre différents constructeurs :

1. Un constructeur prenant un initializer\_list de double de composantes et qui crée un Vecteur avec cette liste.
2. Un constructeur prenant un vector de double de composantes et qui cree un Vecteur avec cette liste.
3. Un autre prenant trois doubles en argument et créant un vecteur à trois dimensions avec ceux-ci.
4. Un dernier prenant un unsigned int en argument et prenant 3 par défaut, il permet de créer un vecteur nul de dimension rentrée en argument.//

* **//Affiche :** méthode constante qui permet d’afficher un vecteur. Elle sera notamment appelée lors de la surcharge de l’opérateur<<.//
* **//Surcharge des opérateurs :**

1. **Operator== :** surcharge interne constante, qui prends en argument un vecteur regarde si deux vecteurs sont exactement identiques (dimensions et toutes les composantes). Pour cela il utilise la fonction meme\_dimesion puis il comparera chaque double du vecteur un à un.
2. **Operator!=**: surcharge interne constante, prenant en argument un vecteur et regarde si deux vecteurs sont différents pour cela il utilise l’opérateur==(au préalable surchargé).
3. **Operator-**: prends aucun argument permet simplement de retourner l’oppose du vecteur. Pour cela il vient inverser chaque double du Vecteur.
4. **Operator~ :** retourne le vecteur unitaire dans la direction du Vecteur. Il utilise la méthode norme et l’opérateur\*(voir plus tard). Il multiplie donc le Vecteur par 1/norme(), il ne prend rien en argument. Si la norme est nul, le vecteur nul n’ayant aucune direction, une erreur est lance.
5. **Operator+= :** prenant en argument un Vecteur en référence constante, en ajoute a l’instance courante du Vecteur celui rentré en argument, pour cela on passe sur chaque composante des deux vecteurs en additionant aux composantes du vecteur de l’instance courante celle du vecteur en argument.
6. **Operator-=**: prends un Vecteur argument par référence constante et soustrait au vecteur de l’instance courante celui rentré en argument. Pour cela on utilise le même procédé qu’avec l’opérateur+= mais au lieu d’additionner on soustrait.
7. **Operator\*=**:prends un scalaire par argument et multiplie chaque composante du Vecteur en mémoire par ce scalaire.
8. **Operator^= :** prends en argument un Vecteur par référence constante et affecte le produit vectorielle de l’instance courante avec le vecteur passe en argument dans l’instance courante.
9. **Operator\* (produit scalaire) :** prends en argument un autre vecteur et retourne le produit scalaire de l’instance courante avec ce vecteur. Le resultat etant un scalaire (un double), et non un vecteur, il est bien sur illogique d’implementer une methode \*= correspondante.
10. **Norme :**  retourne la norme euclidienne (un double) de l’instance courante en utilisant le fait que la norme euclidienne d’un vecteur est egale la racine carre du produit scalaire avec lui-même.
11. **Norme :** retourne la norme euclidienne au carre, qui pourrait etre utile dans la pratique pour des longs calculs.
12. **Operator+ :** retourne la somme de 2 vecteurs (un vecteur) en utilisant l’operateur +=
13. **Operator- :** retourne la difference de 2 vecteurs (un vecteur) en utilisant l’operateur -=
14. **Operator\* :** retourne le produit d’un scalaire et un vecteur (un vecteur) en utilisant l’operateur \*= (defini 2 fois pour preserver la commutativite, i.e. 2\*v=v\*2).
15. **Operator^ :** retourne le produit vectorielle de 2 vecteurs 3D (un vecteur 3D) en utilisant l’operateur ^=//

**//Vecteur 3D :**

Nous voulions créer une sous classe Vecteur 3D car cela semblait être une bonne idée compte tenu du fait que l’on va travailler sur des référentiels à trois dimensions pour les toupie . De plus cela nous aurait permis une meilleur encapsulation de la classe Vecteur car les constructeurs par défaut (pour une dimension de 3) et celui prenant trois double (x,y,z) aurait été propre à la classe Vecteur 3D mais aussi, le produit vectoriel serait une méthode propre à Vecteur 3D car cela n’est pas pensable mathématiquement et logiquement pour autre que 3 dimensions.

Cependant cela nous apporterait une problème conséquent étant causé par la méthode augmente de la classe Vecteur. En effet, augmenter un Vecteur de 2 dimensions en un Vecteur 3D est tout à fait pensable et logiquement possible, mais le nouveau vecteur (qui appartiendrai a la super-classe Vecteur) ne serai pas un Vecteur 3D, meme si il contient 3 dimensions. La methode augmente nous empeche donc de pouvoir cree une classe Vecteur 3D.//

**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.

//Enfin, choisir des doubles comme coefficient d’une matrice est évident par rapport au fait que l’on travaille sur une système réel et continu.//

Méthodes :

* **//Vérifier :** Nous avons réaliser l’importance d’une méthode nous permettant de vérifier si des coefficients d’une matrice était en dessous d’un seuil de précision (égal à 1e-15, defini dans un fichier constant.cc) pour si cela est le cas, les initialiser à 0. En effet étant donné le fait que l’on travaille sur des doubles il est possible de ne pas obtenir 0 (a cause de la representation des nombre binaires fini utilise par un ordinateur) à un calcul ou l’on devrait l’obtenir. Cette méthode sera seulement en privée car inutile à l’utilisateur.
* **Cofacteur :** Cette méthode privé nous permet de calculer le cofacteur d’une matrice et donc de nous simplifier le code de la méthode inv() permettant de calculer la matrice inverse. De plus après réflexion nous avons décidé de retourner directement la matrice des cofacteurs sans utiliser de boucle conditionnelles ou d’appels à des vecteurs car cela est plus optimiser même si pas très esthétique.
* **Constructeurs :**

1. Un constructeur prenant trois doubles et les initialisant comme les diagonales de la matrice (les autres coefficient de la matrice seront égal à 0). De plus par défaut, ces diagonales sont initialisées à 1, nous permettant de créer la matrice identité si rien n’est rentré.
2. Un autre constructeur prenant tout simplement les 9 doubles et les initialisent comme les coefficients d’une matrice.

* **Affiche :** méthode publique permet d’afficher une matrice ayant les coefficients aligné en utilisant un manipulateur ( setw(7) ) .
* **Transp :** une autre méthode publique permettant de retourner la transposé d’une matrice.
* **Det :** méthode publique qui permet de retourner le déterminant d’une matrice, il sera calculé de façon simple à partir des coefficient d’une matrice sans boucle conditionnelle car cela est plus optimisé et enfin on vérifie si ce déterminant n’est pas en dessous d’une précision étant donné le fait que l’on travaille avec des doubles.
* **inv() :** retourne l’inverse d’une matrice simplement grâce à la méthode cofacteur et det.
* **Surcharge d’opérateurs :**

Nous avons décidé de surcharger les opérateurs ==, !=,+=,-=,<<,+,- comme ayant les mêmes caractéristique que pour la surcharge operateurs pour la classe Vecteur mais cette fois-ci avec des matrice.

Les opérateurs différents seront :

1. **Operator\*= :** (en argument une matrice) produit matriciel entre deux matrices.
2. **Operator\*= :** (en argument un vecteur) produit matriciel entre un vecteur et une matrice.
3. **Operarot\*= :** (en argument un scalaire) multiplication de la matrice par un scalaire.

Enfin, les quatre différents operateurs \* reprennent juste les trois operateurs \*= mais cette fois-ci placé en externe, il prendront donc une matrice en plus en argument.(Par ailleurs l’opérateur en plus par rapport aux operator\*= n’est en fait qu’un operator permettant de faire matrice\*scalaire au lieu de simplement scalaire\*matrice).

**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 classe 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 Newark. L’utilisation du polymorphisme semble donc évidente dans ce cas.

**Méthodes :**

* Intègre : sert à faire évoluer une toupie d’un pas de temps(tous deux rentré en arguments), pour cela on fait « évoluer » les vecteurs position et dérivée de la position à partir de la formule spécifique à l’intégrateur en question.
* Copie : sert à faire une copie du pointeur pointant sur l’intégrateur en question (sera utilisé pour la copie d’un système).

**Classe Support à Dessin :**

Cette méthode permettra à l’utilisateur de choisir si 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.