ATL-2201 - Algèbre linéaire et vision 3D
TP3 : Applications linéaires/matrices en dimension arbitraire

Romain Negrel*

Objectifs

Programmer une classe `Matrice` de taille arbitraire pour :

1. représenter des applications linéaires d'un espace euclidien de dimension m dans un espace euclidien de dimension n, quels que soient les entiers naturels m et n ; et
2. trouver l'image d'un vecteur par une telle application linéaire.

Liste des compétences à acquérir pendant le TP :

- déclarer un tableau ;
- allouer un tableau ;
- accéder aux éléments d'un tableau à deux dimensions ;
- écrire deux boucles 'for' imbriquées pour affecter une valeur à tous les éléments d'un tableau à deux dimensions ;
- écrire un programme pour multiplier à droite une matrice par un vecteur ;
- écrire un programme pour obtenir une matrice correspondant aux applications linéaires remarquables suivantes :
  - identité ;
  - homothétie ;
  - symétrie centrale ;
  - réflexion 2D/3D ;
  - rotation 2D/3D.
- vérifier graphiquement, en les appliquant à des vecteurs, que les méthodes associées aux applications linéaires ci-dessus sont conformes à leur définition.

Ce programme sera réutilisé tout au long de l'unité.

*d'après un sujet de Jean Cousty et Benjamin Perret

Sujet

1. **Travail préliminaire**

   1.1. Terminez le TP2.

   1.2. Créez une copie de votre répertoire TP2, et renommez cette copie TP3. Sous BlueJ, ouvrez le projet intitulé TP3 correspondant à la copie du TP2 que vous venez de créer.

Important

Après avoir répondu à une question, cliquez sur le bouton "`Exécuter les tests`" et vérifiez que l'indicateur de la question passe au **vert**. Si BlueJ indique une croix **rouge** pour la question que vous venez de résoudre, cela veut dire que votre solution n'est pas correcte ! Si l'indicateur passe au vert, cela veut dire que votre solution est peut-être correcte !

Important !

Ajoutez des commentaires dans votre code !

2. **Création de la classe `Matrice`**

   Cette classe devra permettre d'instancier une matrice de taille $m \times n$ arbitraire, c'est-à-dire, une matrice correspondant à une application linéaire d'un espace euclidien de dimension m dans un espace euclidien de dimension n où les dimensions m et n sont choisies au moment de la création de la matrice. En termes de programmation, cela signifie que l'on passera au constructeur de la classe (dont la programmation sera abordée à la question 2.4) deux paramètres indiquant les nombres m et n de lignes et de colonnes de la matrice ; cette matrice comprendra donc $m \times n$ coefficients et chaque coefficient sera indexé par un couple d'entiers dont le premier élément est compris entre 0 et $m-1$ et le second est compris entre 0 et $n-1$.

   2.1. Créez dans votre projet une nouvelle classe `Matrice` (bouton 'New Class' sous BlueJ). Inspectez le code de la classe `Matrice` créé par BlueJ. Que contient-il ? Cela correspond-il à une matrice de taille arbitraire ? Supprimez les attributs et méthodes qui ne sont pas nécessaires. En utilisant uniquement les types de données vus en cours jusqu'à présent, est-il possible de déclarer les attributs correspondant à une matrice en dimension arbitraire ? Pourquoi ?

   2.2. Afin de pouvoir définir les attributs d'une classe pour représenter une matrice en dimension arbitraire, il est possible d'utiliser un tableau à deux dimensions. Après avoir lu cette ressource sur les tableaux à deux dimensions, déclarez dans la classe `Matrice` un attribut pour représenter une matrice de taille arbitraire. Nous rappelons que les coefficients des matrices sont des nombres réels et que le type `double` est utilisé pour représenter ces nombres réels.

   2.3. Déclarez dans la classe `Matrice` deux attributs correspondant respectivement au nombre de lignes et au nombre de colonnes de l'instance courante de la classe `Matrice`.

   2.4. Ajoutez à la classe `Matrice` un constructeur à deux paramètres de type `int` spécifiant les nombres de lignes et de colonnes de la matrice et créant une instance de la matrice nulle. La signature de ce constructeur est donc :

```
public Matrice(final int pm, final int pn)
```

2.5. Ajoutez à la classe `Matrice` un accesseur à deux paramètres de type `int` spécifiant le couple d'indices (i, j) du coefficient à retourner. La signature de cet accesseur est donc :

```
public double getCoefficient(final int pi, final int pj)
```

2.6. Ajoutez à la classe `Matrice` deux accesseurs sans paramètre retournant respectivement le nombre de lignes et le nombre de colonnes de la matrice. Les signatures de ces accesseurs sont donc :

```
public int getNbColonnes()
public int getNbLignes()
```

2.7. Ajoutez à la classe `Matrice` un modificateur à trois paramètres permettant d'affecter une valeur à l'un des coefficients de l'instance courante de la classe `Matrice`. Les deux premiers paramètres, de type `int`, spécifient le couple d'indices du coefficient à modifier et le troisième, de type `double`, spécifie la valeur à affecter à ce coefficient. La signature de ce modificateur est donc :

```
public void setCoefficient(final int pi, final int pj, final double
    pCoefficient)
```

2.8. Afin de tester votre classe, créez une matrice de taille 3×3 et une matrice de taille 3×4. Invoquez les méthodes `getNbColonnes` et `getNbLignes` sur ces deux matrices et vérifiez qu'elles retournent des valeurs conformes aux définitions. Modifiez ensuite la première matrice afin qu'elle représente l'application linéaire identité de l'espace euclidien de dimension 3. Finalement, vérifiez grâce à la méthode `getCoefficient` que les modifications effectuées sont conformes à ce qui est souhaité. Que se passe-t-il lorsque vous invoquez `getCoefficient(3,4)` sur les deux matrices que vous avez créées ? Expliquez.

3. **Image d'un vecteur par une application linéaire / multiplication d'une matrice par un vecteur**

Afin de calculer l'image d'un vecteur par une application linéaire, il est suffisant de savoir multiplier à droite une matrice par un vecteur. L'objectif de cet exercice est d'implémenter cette opération. Pour cela, on s'appuiera sur la décomposition de la matrice en vecteurs-lignes et sur des calculs de produits scalaires.

3.1. Ajoutez à la classe `Matrice` une méthode `getVecteurLigne` à un paramètre, de type `int`, retournant un `Vecteur` ayant pour coordonnées les coefficients de l'instance courante de la classe `Matrice` dont le premier indice est égal au paramètre de la méthode. En d'autres termes, le vecteur retourné correspond à la ligne indicée par l'entier passé en paramètre.

3.2. Après avoir consulté les éléments d'aide présentés ci-après, ajoutez à la classe `Matrice` une méthode `multiplicationVectorielle` à un paramètre, de type `Vecteur`, retournant un `Vecteur` qui est le produit (à droite) de l'instance courante la classe `Matrice` par le vecteur passé en paramètre de la méthode. En d'autres termes, cette méthode retourne l'image du vecteur paramètre par l'application linéaire associée à l'instance courante de la classe `Matrice`.

Aide

Soit une matrice $\mathbf{M}$ de taille $m \times n$, avec m lignes et n colonnes, représentant une application linéaire d'un espace euclidien de dimension n vers un espace euclidien de dimension m. Soit $\mathbf{u}$ un vecteur de l'espace euclidien de dimension n.
Pour calculer le produit de $\mathbf{M}$ par $\mathbf{u}$ (c'est-à-dire $\mathbf{M} \cdot \mathbf{u}$), il suffit de multiplier chaque ligne de la matrice $\mathbf{M}$ par le vecteur $\mathbf{u}$. Cette opération s'effectue de la manière suivante :

1. Considérez chaque ligne de $\mathbf{M}$ comme un vecteur de dimension n. Notons $\mathbf{m}_i$ le vecteur ligne correspondant à la i-ème ligne de $\mathbf{M}$.
2. Pour chaque i-ème ligne, calculez le produit scalaire de $\mathbf{m}_i$ et de $\mathbf{u}$.
3. La i-ème coordonnée du vecteur résultat (qui appartient à l'espace euclidien de dimension m) sera donc donnée par le produit scalaire $\mathbf{m}_i \cdot \mathbf{u}$.

En résumé, le produit de la matrice $\mathbf{M}$ par le vecteur $\mathbf{u}$ se calcule en prenant chaque vecteur ligne de $\mathbf{M}$, en calculant son produit scalaire avec $\mathbf{u}$, et en plaçant chaque résultat obtenu dans la coordonnée correspondante du vecteur final $\mathbf{v}$.

3.3. Testez votre méthode avec la matrice

$$\mathbf{M} = \begin{bmatrix} 0 & -1 & -2 \\ 1 & 0 & 1 \end{bmatrix} \text{ et le vecteur } \mathbf{u} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}.$$

Visualisez à l'aide de la classe `Plan` le vecteur $\mathbf{u}$ et le résultat de la multiplication.

4. **Applications remarquables**

Dans cette partie, vous allez créer des matrices spécifiques appelées "applications remarquables". Ces matrices représentent des transformations linéaires importantes, qui sont souvent utilisées en géométrie et en physique pour manipuler des objets, tels que des vecteurs ou des points, dans l'espace. Les applications remarquables incluent des transformations telles que les rotations, les réflexions, les homothéties, etc.

Aide

Pour construire la matrice de chaque transformation, il est important de comprendre que toute fonction linéaire d'un espace vectoriel peut être décrite sous la forme d'une matrice. Pour cela, nous procédons en deux étapes :

1. Appliquer la transformation à chaque vecteur de la base canonique de l'espace de départ.
2. Construire la matrice en plaçant chaque image dans les colonnes de la matrice. Autrement dit, chaque colonne de la matrice est le résultat de la transformation appliquée au vecteur de la base canonique correspondant.

Par exemple, considérons une transformation linéaire f agissant sur un espace de dimension n avec la base canonique $\{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n\}$. Chaque colonne de la matrice $\mathbf{M}_f$ est alors donnée par :

$$\mathbf{M}_f = \begin{bmatrix} f(\mathbf{e}_1) & f(\mathbf{e}_2) & \ldots & f(\mathbf{e}_n) \end{bmatrix}.$$

Cela signifie que si $f(\mathbf{e}_1) = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$, alors la première colonne de $\mathbf{M}_f$ sera :

$$\mathbf{M}_f = \begin{bmatrix} a_1 & * & \ldots & * \\ a_2 & * & \ldots & * \\ \vdots & \vdots & \ddots & \vdots \\ a_n & * & \ldots & * \end{bmatrix},$$

où les éléments $*$ seront remplis par les résultats des transformations $f(\mathbf{e}_2), \ldots, f(\mathbf{e}_n)$ dans les colonnes suivantes.
En résumé, en connaissant l'image de chaque vecteur de la base canonique, il devient possible de déterminer comment la transformation linéaire agit sur tout autre vecteur de l'espace par simple multiplication matricielle. Ce processus permet de modéliser et d'appliquer les propriétés de transformations géométriques de manière pratique et systématique.

Pour chaque application, il est attendu que vous ajoutiez une méthode dans votre classe `Matrice` qui configure la matrice de manière à ce qu'elle représente la transformation souhaitée.

4.1. **Application identité** Ajoutez à la classe `Matrice` une nouvelle procédure `setIdentite` sans paramètre qui modifie l'instance courante de la classe `Matrice` afin que celle-ci corresponde à l'application identité. On supposera que cette méthode est invoquée uniquement sur des matrices carrées (ayant le même nombre de lignes et de colonnes).

4.2. En utilisant la représentation graphique des vecteurs obtenue grâce à la classe `Plan`, vérifiez, pour la dimension 2, que la méthode `setIdentite`, programmée à la question (a), produit une matrice qui ne modifie pas les vecteurs en dimension 2 par multiplication à droite.

4.3. **Application homothétie** Ajoutez à la classe `Matrice` une nouvelle procédure `setHomothetie` à un paramètre `k`, de type `double`, qui modifie l'instance courante de la classe `Matrice` afin que celle-ci corresponde à l'homothétie (de centre O) de facteur `k`. On supposera que cette méthode est invoquée uniquement sur des matrices carrées (ayant le même nombre de lignes et de colonnes).

4.4. En utilisant la représentation graphique des vecteurs obtenue grâce à la classe `Plan`, vérifiez, pour la dimension 2, que la méthode `setHomothetie` conduit bien à une matrice qui correspond à l'application linéaire homothétie. Testez également cette méthode sur des vecteurs en dimension 3.

4.5. **Application symétrie centrale** Ajoutez à la classe `Matrice` une nouvelle procédure `setSymetrieCentrale` sans paramètre qui modifie l'instance courante de la classe `Matrice` afin que celle-ci corresponde à une symétrie centrale (de centre $\mathcal{O}$). On supposera que cette méthode est invoquée uniquement sur des matrices carrées (ayant le même nombre de lignes et de colonnes).

4.6. En utilisant la représentation graphique des vecteurs obtenue grâce à la classe `Plan`, vérifiez, pour la dimension 2, que la méthode `setSymetrieCentrale` conduit bien à une matrice qui correspond à l'application linéaire symétrie centrale. Testez également cette méthode sur des vecteurs en dimension 3.

4.7. **Application réflexion 2D axe $\mathcal{O}_x$** Ajoutez à la classe `Matrice` une nouvelle procédure `setReflexionOx` sans paramètre qui modifie l'instance courante de la classe `Matrice` afin que celle-ci corresponde à la réflexion dont l'axe est défini par le vecteur $(1; 0)$. On supposera que cette méthode est invoquée uniquement sur des matrices carrées à 2 lignes et 2 colonnes.

4.8. En utilisant la représentation graphique des vecteurs obtenue grâce à la classe `Plan`, vérifiez que la méthode `setReflexionOx` conduit bien à une matrice qui correspond à l'application linéaire de réflexion par rapport à l'axe $\mathcal{O}_x$.

4.9. **Application réflexion 3D plan** $(\mathcal{O}_x, \mathcal{O}_y)$ Ajoutez à la classe `Matrice` une nouvelle procédure `setReflexionOxOy` sans paramètre qui modifie l'instance courante de la classe `Matrice` afin que celle-ci corresponde à la réflexion par rapport au plan défini par les vecteurs $(1; 0; 0)$ et $(0; 1; 0)$. On supposera que cette méthode est invoquée uniquement sur des matrices carrées à 3 lignes et 3 colonnes.

4.10. En utilisant la représentation graphique des vecteurs obtenue grâce à la classe `Plan`, vérifiez que la méthode `setReflexionOxOy` conduit bien à une matrice qui correspond à l'application linéaire de réflexion par rapport au plan défini par les vecteurs $(1; 0; 0)$ et $(0; 1; 0)$.

4.11. **Application Rotation 2D** Ajoutez à la classe `Matrice` une nouvelle procédure `setRotation2d` à un paramètre `alpha`, de type `double`, qui modifie l'instance courante de la classe `Matrice` afin que celle-ci corresponde à une rotation de `alpha` radians (de centre $\mathcal{O}$). On supposera que cette méthode est invoquée uniquement sur des matrices carrées à 2 lignes et 2 colonnes.

4.12. En utilisant la représentation graphique des vecteurs obtenue grâce à la classe `Plan`, vérifiez que la méthode `setRotation2d` conduit bien à une matrice qui correspond à l'application linéaire de rotation.

4.13. **Application Rotation 3D axe $\mathcal{O}_x$** Ajoutez à la classe `Matrice` une nouvelle procédure `setRotation3dOx` à un paramètre `alpha`, de type `double`, qui modifie l'instance courante de la classe `Matrice` afin que celle-ci corresponde à la rotation d'angle `alpha` autour de l'axe défini par le vecteur $(1, 0, 0)$. On supposera que cette méthode est invoquée uniquement sur des matrices carrées à 3 lignes et 3 colonnes.

4.14. En utilisant la représentation graphique des vecteurs obtenue grâce à la classe `Plan`, vérifiez que la méthode `setRotation3dOx` conduit bien à une matrice qui correspond à l'application linéaire de rotation autour de l'axe O_x.

5. Tracé géométrique en 2D

Dans cette section, vous allez créer une classe `Visualisation` dédiée aux tests de visualisation de transformations géométriques en 2D. Cette classe ne doit pas être instanciée, donc le constructeur sera privé, et elle contiendra des méthodes statiques pour visualiser des tracés géométriques tels que des droites et des cercles.

5.1. Créez une classe `Visualisation` qui ne peut pas être instanciée. Pour cela, définissez un constructeur privé. Cette classe contiendra uniquement des méthodes statiques pour effectuer les tests de visualisation.

5.2. Ajoutez à la classe `Visualisation` une méthode publique statique `tracerDroite` avec un paramètre de type `Vecteur`. La signature de cette méthode est donc :

```
public static void tracerDroite(final Vecteur pDirection)
```

Cette méthode doit créer un `Plan` et tracer dans ce plan une série de points alignés pour donner l'impression d'une droite dans la direction spécifiée par le vecteur `pDirection`. Bien que la droite soit théoriquement infinie, il suffit de tracer un nombre limité de points afin qu'elle apparaisse correctement dans la fenêtre de visualisation, définie sur l'intervalle $[-6, 6]$ pour les deux axes.

Aide

Utilisez les applications remarquables pour ajuster la distance entre les points sur la droite, en particulier en appliquant des homothéties successives au vecteur `pDirection`. Calculez l'homothétie maximale en fonction de la diagonale de la fenêtre de visualisation, car cela représente l'échelle maximale nécessaire pour que la droite apparaisse entièrement dans la fenêtre. Utilisez une boucle avec un paramètre `vAlpha` variant entre -1 et 1 pour tracer des points avec une homothétie proportionnelle à `vAlpha`. Cela permet de contrôler facilement le nombre de points tracés.

5.3. Ajoutez à la classe `Visualisation` une méthode publique statique `tracerCercle` avec un paramètre de type `double` pour le rayon du cercle. La signature de cette méthode est :

```
public static void tracerCercle(final double pRayon)
```

Cette méthode doit créer un `Plan` et tracer dans ce plan un cercle de centre O et de rayon `pRayon`.

Aide

Pour dessiner un cercle, placez d'abord un point initial sur le cercle (par exemple, à $(pRayon, 0)$ sur l'axe des x). Ensuite, appliquez des rotations successives à ce point en utilisant une application linéaire de rotation, pour générer les points suivants. Vous pouvez diviser le cercle en un grand nombre de segments en choisissant un petit angle de rotation (comme $2\pi/1000$ pour 1000 segments). En répétant cette rotation, vous obtiendrez une série de points qui, ensemble, forment le cercle.

La classe `Visualisation` est ainsi conçue pour faciliter l'observation des transformations géométriques en 2D. Elle centralise les méthodes de tracé sous forme de procédures statiques, simplifiant ainsi l'implémentation et le test des visualisations géométriques.