Atelier E2 - Algèbre linéaire et vision 3D

TP4 : Coordonnées homogènes, Transformations géométriques et Visualisation 3D

Romain Negrel*

Objectifs

1. Implémenter et utiliser des méthodes de base pour manipuler des matrices, telles que la multiplication matricielle et les transformations linéaires (rotations, homothéties, etc.).
2. Créer un programme pour visualiser un cube en 3D, effectuer des rotations, homothéties, et translations sur celui-ci.
3. Découvrir et appliquer le concept des coordonnées homogènes pour effectuer des translations et d'autres transformations géométriques en 3D.
4. Réaliser un programme combinant plusieurs transformations pour visualiser des effets complexes comme des rotations composées, des translations, ou des transformations animées.
5. Comprendre les implications géométriques des transformations linéaires et homogènes à travers des exercices pratiques.

Liste des compétences à acquérir pendant le TP :

- écrire un programme pour multiplier deux matrices et composer des applications linéaires ;
- utiliser les matrices de rotation pour effectuer des transformations dans l'espace tridimensionnel ;
- appliquer des transformations homogènes (translations, homothéties) à des vecteurs et interpréter les résultats ;
- développer un programme pour visualiser un cube en 3D avec des transformations statiques et animées ;
- intégrer les coordonnées homogènes pour enrichir les capacités de transformation géométrique.

Rappel de cours

Soient N, M et K trois entiers naturels. Soit f une application linéaire de $\mathbb{R}^N$ dans $\mathbb{R}^M$ et g une application linéaire de $\mathbb{R}^M$ dans $\mathbb{R}^K$.

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

Définition 1.

La composée de f et g, notée $g \circ f$, est une application de $\mathbb{R}^N$ dans $\mathbb{R}^K$ qui associe à tout vecteur $\mathbf{u}$ de $\mathbb{R}^N$ le vecteur $\mathbf{v}$ de $\mathbb{R}^K$ obtenu en appliquant f puis g au vecteur $\mathbf{u}$ (i.e., $\mathbf{v} = g(f(\mathbf{u}))$).

Propriété 1.

La composée $g \circ f$ de f et g est une application linéaire.

D'après la propriété 1, la composée $g \circ f$ est une application linéaire de $\mathbb{R}^N$ dans $\mathbb{R}^K$. Il existe donc une matrice de taille $K \times N$ qui caractérise cette application. Comme indiqué par la propriété 2 ci-dessous, le produit matriciel permet précisément d'obtenir cette matrice caractéristique à partir des matrices caractéristiques de f et g.

Soient $\mathbf{F}$ et $\mathbf{G}$ les deux matrices caractéristiques des applications f et g. Notons que $\mathbf{F}$ et $\mathbf{G}$ sont donc respectivement de taille $M \times N$ et $K \times M$.

Définition 2 (produit matriciel).

Le produit (matriciel) de $\mathbf{G}$ par $\mathbf{F}$, noté $\mathbf{G} \times \mathbf{F}$, est la matrice de taille $K \times N$ telle que, pour tout indice i compris entre 0 et $K-1$ et tout indice j compris entre 0 et $N-1$, le coefficient $[\mathbf{G} \times \mathbf{F}]_{i,j}$ est le produit scalaire du $i^{\text{ème}}$ vecteur ligne de $\mathbf{G}$ avec le $j^{\text{ème}}$ vecteur colonne de $\mathbf{F}$:

$$[\mathbf{G} \times \mathbf{F}]_{i,j} = \mathbf{G}_{i,\cdot} \cdot \mathbf{F}^T_{\cdot,j}$$

où $\mathbf{G}_{i,\cdot}$ désigne le $i^{\text{ème}}$ vecteur ligne de $\mathbf{G}$ et où $\mathbf{F}_{\cdot,j}$ désigne le $j^{\text{ème}}$ vecteur colonne de $\mathbf{F}$.

Propriété 2.

$\mathbf{G} \times \mathbf{F}$ est la matrice caractéristique de l'application $g \circ f$.

En d'autres termes, cela signifie que si l'on veut trouver l'image $\mathbf{v}$ d'un vecteur $\mathbf{u}$ par la composée $g \circ f$ de g et f, on peut procéder de deux manières différentes :

1. on obtient d'abord l'image $\mathbf{w} = f(\mathbf{u})$ de $\mathbf{u}$ par f en multipliant $\mathbf{F}$ par $\mathbf{u}$ (i.e., $\mathbf{w} = \mathbf{F} \cdot \mathbf{u}$), puis on obtient l'image $\mathbf{v} = g(\mathbf{w})$ de $\mathbf{w}$ par g (donc $\mathbf{v} = g(f(\mathbf{u}))$) en multipliant $\mathbf{G}$ par $\mathbf{w}$ (i.e., $\mathbf{v} = \mathbf{G} \cdot \mathbf{w}$) ;

2. on obtient d'abord la matrice $\mathbf{H}$ caractéristique de l'application $g \circ f$ en multipliant $\mathbf{G}$ par $\mathbf{F}$ (i.e., $\mathbf{H} = \mathbf{G} \times \mathbf{F}$), puis on obtient l'image de $\mathbf{u}$ par $g \circ f$ en multipliant $\mathbf{H}$ par $\mathbf{u}$ (i.e., $\mathbf{v} = \mathbf{H} \cdot \mathbf{u}$).

Lorsque l'on souhaite trouver l'image d'un unique vecteur par $g \circ f$, la méthode 1 est moins coûteuse que la méthode 2. On pourra notamment vérifier qu'elle nécessite un moins grand nombre d'opérations. En revanche, la méthode 2 est nettement moins coûteuse que la méthode 1 pour trouver les images de plusieurs (plus que M) vecteurs par $g \circ f$ car le produit de $\mathbf{F}$ par $\mathbf{G}$ ne doit alors être effectué qu'une seule fois. Cette situation sera illustrée dans l'exercice 3 où l'on souhaite trouver l'image des 8 coins d'un cube (i.e., 8 vecteurs de $\mathbb{R}^3$) par une composée d'applications linéaires.

Sujet

1. **Travail préliminaire**

1.1. Terminez le TP2.

1.2. Terminez le TP3.

1.3. Créez une copie du répertoire correspondant au TP3 et renommez cette copie TP4. Sous BlueJ, ouvrez le projet intitulé TP4 correspondant à la copie du TP3 que vous venez de créer.

Important !!

Ajoutez des commentaires dans votre code !

2. **Composition d'applications linéaires / multiplication matricielle**

La composition $g \circ f$ de deux applications linéaires f et g est caractérisée par le produit des matrices associées (pour plus de détails, voir le rappel de cours ci-dessus). L'objectif de cet exercice est d'implémenter cette opération de multiplication matricielle. Pour cela, on s'appuiera sur la décomposition de matrices en vecteurs colonnes et en vecteurs lignes et sur des produits scalaires.

2.1. Ajoutez à la classe `Matrice` une méthode publique `getVecteurColonne` à un paramètre, de type `int`, retournant un Vecteur ayant pour coordonnées les coefficients de l'instance courante de la classe `Matrice` dont le second indice est égal au paramètre de la méthode. En d'autres termes, le vecteur retourné correspond à la colonne indiquée par l'entier passé en paramètre.

2.2. Ajoutez à la classe `Matrice` une méthode publique `produitMatriciel` à un paramètre de type `Matrice`. Cette méthode agit de la manière suivante :

- si le nombre de colonnes de l'instance courante de la classe `Matrice` est égal au nombre de lignes de la Matrice passée en paramètre, alors la méthode retourne une nouvelle Matrice qui est le produit de l'instance courante par le paramètre.
- sinon, la méthode retourne la référence `null` pour indiquer que l'opération ne peut pas être effectuée.

2.3. Testez votre méthode avec la matrice $\mathbf{M_1} = \begin{pmatrix} 2 & 1 & 1 \\ 4 & 2 & 3 \end{pmatrix}$ et la matrice $\mathbf{M_2} = \begin{pmatrix} -1 & 2 \\ 3 & 2 \\ -1 & 2 \end{pmatrix}$.

2.4. **Application Rotation 3D axe $\mathcal{O}_y$**

Ajoutez à la classe `Matrice` une nouvelle procédure `setRotation3dOy` 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 $(0, 1, 0)$. On supposera que cette méthode est invoquée uniquement sur des matrices carrées à 3 lignes et 3 colonnes.

2.5. **Application Rotation 3D axe $\mathcal{O}_z$**

Ajoutez à la classe `Matrice` une nouvelle procédure `setRotation3dOz` 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 $(0, 0, 1)$. On supposera que cette méthode est invoquée uniquement sur des matrices carrées à 3 lignes et 3 colonnes.

2.6. **Composition de rotations** Écrivez, dans la classe `Matrice`, une fonction statique `getRotation` qui prend en paramètres trois angles `pAlphaX`, `pAlphaY`, et `pAlphaZ` (de type `double`) et retourne une matrice correspondant à la composition des rotations d'angles `pAlphaX`, `pAlphaY`, et `pAlphaZ` autour respectivement des trois axes

$\mathcal{O}_x$, $\mathcal{O}_y$, et $\mathcal{O}_z$. On utilisera les méthodes `setRotation3dOx`, `setRotation3dOy` et `setRotation3dOz` pour construire cette matrice de rotation.

3. **Ajout de méthodes pour la visualisation en 3D**

Dans cette partie, vous allez enrichir la classe `Visualisation` avec des méthodes statiques pour visualiser un cube en 3D : la première méthode crée les sommets d'un cube, la seconde méthode dessine le cube en reliant les sommets, et la dernière méthode utilise ces deux fonctions pour créer et afficher directement le cube.

3.1. **Créer les sommets du cube**

Ajoutez la méthode statique `creerSommets` à la classe `Visualisation`. Cette méthode ne prend aucun paramètre et renvoie un tableau de `Vecteur` contenant les sommets d'un cube centré sur l'origine.

```
/**
 * Crée les sommets d'un cube centré sur l'origine et les met dans un
    tableau.
 * @return Tableau de Vecteur
 */
private static Vecteur[] creerSommets() {
    Vecteur p1 = new Vecteur(3); // sommet arrière gauche bas
    p1.setCoordonnee(0, -1);
    p1.setCoordonnee(1, -1);
    p1.setCoordonnee(2, -1);
    Vecteur p2 = new Vecteur(3); // sommet arrière gauche haut
    p2.setCoordonnee(0, -1);
    p2.setCoordonnee(1, 1);
    p2.setCoordonnee(2, -1);
    Vecteur p3 = new Vecteur(3); // sommet arrière droite haut
    p3.setCoordonnee(0, 1);
    p3.setCoordonnee(1, 1);
    p3.setCoordonnee(2, -1);
    Vecteur p4 = new Vecteur(3); // sommet arrière droite bas
    p4.setCoordonnee(0, 1);
    p4.setCoordonnee(1, -1);
    p4.setCoordonnee(2, -1);
    Vecteur p5 = new Vecteur(3); // sommet avant gauche bas
    p5.setCoordonnee(0, -1);
    p5.setCoordonnee(1, -1);
    p5.setCoordonnee(2, 1);
    Vecteur p6 = new Vecteur(3); // sommet avant gauche haut
    p6.setCoordonnee(0, -1);
    p6.setCoordonnee(1, 1);
    p6.setCoordonnee(2, 1);
    Vecteur p7 = new Vecteur(3); // sommet avant droite haut
    p7.setCoordonnee(0, 1);
    p7.setCoordonnee(1, 1);
    p7.setCoordonnee(2, 1);
    Vecteur p8 = new Vecteur(3); // sommet avant droite bas
    p8.setCoordonnee(0, 1);
    p8.setCoordonnee(1, -1);
    p8.setCoordonnee(2, 1);

    return new Vecteur[]{p1, p2, p3, p4, p5, p6, p7, p8};
```

```
}
```

3.2. Dessiner le cube

Ajoutez la méthode statique `dessinerCube` à la classe `Visualisation`. Cette méthode prend en paramètre un objet `Plan` et un tableau de `Vecteur` (les sommets du cube), et dessine les arêtes du cube en reliant les sommets.

```
/**
 * Dessine sur le Plan les arêtes du cube dont les sommets sont dans le
    tableau de Vecteur.
 * @param pPlan~: plan où effectuer le dessin
 * @param pSommets~: sommets du cube
 */
private static void dessinerCube(final Plan pPlan, final Vecteur[]
    pSommets) {
    pPlan.effacer();

    // Face arrière
    pPlan.dessinerSegmentEn2d(pSommets[0], pSommets[1]);
    pPlan.dessinerSegmentEn2d(pSommets[1], pSommets[2]);
    pPlan.dessinerSegmentEn2d(pSommets[2], pSommets[3]);
    pPlan.dessinerSegmentEn2d(pSommets[3], pSommets[0]);

    // Face avant
    pPlan.dessinerSegmentEn2d(pSommets[4], pSommets[5]);
    pPlan.dessinerSegmentEn2d(pSommets[5], pSommets[6]);
    pPlan.dessinerSegmentEn2d(pSommets[6], pSommets[7]);
    pPlan.dessinerSegmentEn2d(pSommets[7], pSommets[4]);

    // Relier la face avant àla face arrière
    pPlan.dessinerSegmentEn2d(pSommets[0], pSommets[4]);
    pPlan.dessinerSegmentEn2d(pSommets[1], pSommets[5]);
    pPlan.dessinerSegmentEn2d(pSommets[2], pSommets[6]);
    pPlan.dessinerSegmentEn2d(pSommets[3], pSommets[7]);
}
```

3.3. Création et affichage automatique du cube

Ajoutez une méthode statique `afficherCubeBase` dans la classe `Visualisation`. Cette méthode doit créer un objet `Plan`, générer les sommets du cube en appelant la fonction `creerSommets`, puis dessiner le cube en appelant la fonction `dessinerCube` avec le plan et les sommets du cube.

3.4. Appel de la fonction et réflexion sur la visualisation obtenue

Appelez la fonction `afficherCubeBase` pour afficher le cube sur le plan. Observez la visualisation obtenue et réfléchissez : voyez-vous clairement un cube ? Pourquoi ou pourquoi pas ? Comment les transformations ou les projections pourraient-elles affecter la perception 3D du cube sur un plan 2D ?

4. Ajout de méthodes génériques pour appliquer des transformations à une liste de vecteurs

Ajoutez les éléments suivants directement dans la classe `Visualisation`. Ce code introduit une interface générique et une méthode permettant d'appliquer des transformations à une liste de vecteurs de manière flexible.

4.1. **Ajoutez l'interface** `FunctionDeTraitementVecteur`

Cette interface définit une méthode abstraite `exécuter` qui sera utilisée pour appliquer une transformation à un vecteur.

```
/**
 * Interface pour le traitement des vecteurs.
 */
public interface FunctionDeTraitementVecteur {
    Vecteur exécuter(final Vecteur pVecteur);
}
```

4.2. **Ajoutez une méthode statique** `appliquer`

Cette méthode générique utilise une implémentation de `FunctionDeTraitementVecteur` pour transformer chaque vecteur dans un tableau et retourne un nouveau tableau contenant les vecteurs transformés.

```
/**
 * Applique une transformation définie par une implémentation de
    FunctionDeTraitementVecteur
 * à chaque vecteur d'un tableau.
 *
 * @param pFTV une implémentation de FunctionDeTraitementVecteur
 * @param pVecteurs le tableau de vecteurs àtransformer
 * @return un nouveau tableau contenant les vecteurs transformés
 */
public static Vecteur[] appliquer(FunctionDeTraitementVecteur pFTV,
    Vecteur[] pVecteurs) {
    Vecteur[] result = new Vecteur[pVecteurs.length];
    for (int i = 0; i < pVecteurs.length; i++) {
        result[i] = pFTV.exécuter(pVecteurs[i]);
    }
    return result;
}
```

Pourquoi ce code est utile ?

Ce code vous permet d'appliquer des transformations génériques sur un tableau de vecteurs, comme des rotations, des translations ou des normalisations. Grâce à cette approche, vous pouvez facilement réutiliser du code existant en utilisant des références de méthode (`::`) au lieu de coder manuellement chaque transformation. Cela rend vos programmes plus modulaires et plus lisibles.

Points clés pour comprendre ce code

- L'interface `FunctionDeTraitementVecteur` permet de généraliser le traitement des vecteurs.
- La méthode `appliquer` utilise cette interface pour exécuter des transformations sur un tableau de vecteurs.
- Vous pouvez utiliser des références de méthode (`::`) pour passer directement des méthodes existantes, qu'elles soient statiques ou d'instance.
- Cette approche réduit la duplication de code et améliore la lisibilité en isolant la logique des transformations.

En suivant ces étapes, vous pourrez facilement transformer des vecteurs de manière flexible et réutilisable dans vos projets.

Exemples d'utilisation

Voici des exemples concrets pour comprendre comment utiliser ce code dans des cas pratiques :

1. Utilisation avec une méthode d'instance

Si vous avez une matrice de rotation `Matrice` contenant une méthode `multiplicationVectorielle`, vous pouvez appliquer cette méthode à tous les sommets d'un cube :

```
// Création d'une matrice de rotation
Matrice vRotMat = Matrice.getRotation(Math.PI / 4, Math.PI / 4, 0);
// Application de la rotation àchaque vecteur des sommets du cube
vSommets = appliquer(vRotMat::multiplicationVectorielle, vSommets);
```

2. Utilisation avec une méthode statique d'une classe

Si vous avez une méthode statique `maMethode` dans une classe `MaClasse`, qui effectue une transformation sur un vecteur, vous pouvez également l'utiliser avec `appliquer` :

```
// Utilisation de la méthode statique comme transformation
vSommets = appliquer(MaClasse::maMethode, vSommets);
```

5. Application de transformations linéaires pour visualiser des objets en 3D

L'objectif de cette question est de produire différentes visualisations d'un cube en 3D, en appliquant des transformations pour modifier l'angle de vue, la taille ou la rotation de l'objet.

5.1. Rotation de base du cube

Ajoutez une méthode statique `afficherCubeRotation` dans la classe `Visualisation`. Cette méthode doit créer un plan, générer les sommets d'un cube, puis appliquer une

matrice de rotation autour de l'axe $\mathcal{O}_x$. Utilisez la méthode `appliquer` de la classe `Matrice` pour transformer le tableau de vecteurs avant de dessiner le cube.

5.2. **Transformation du cube par homothétie**

Ajoutez une méthode statique `afficherCubeHomothetie` dans la classe `Visualisation`. Cette méthode doit créer un plan, générer les sommets d'un cube, appliquer une homothétie pour changer la taille du cube, et afficher le résultat. Utilisez la méthode `appliquer` de la classe `Matrice` pour transformer le tableau de vecteurs.

5.3. **Rotation composée du cube autour des trois axes**

Ajoutez une méthode statique `afficherCubeRotationComposee` dans la classe `Visualisation`. Utilisez la fonction `getRotation` de la classe `Matrice` pour créer une matrice de rotation composée de rotations successives autour des axes $\mathcal{O}_x$, $\mathcal{O}_y$, et $\mathcal{O}_z$ (par exemple, avec $\alpha_x = \pi/4$, $\alpha_y = \pi/4$, et $\alpha_z = \pi/4$). Appliquez cette transformation aux sommets du cube et affichez le résultat.

5.4. **Visualisation animée du cube**

Ajoutez une méthode statique `afficherCubeAnime` dans la classe `Visualisation`. Dans cette méthode, créez un plan, générez les sommets du cube, puis appliquez des transformations successives aux sommets pour obtenir une animation. Décomposez l'animation en étapes où chaque transformation applique une rotation ou une homothétie progressive aux sommets du cube. Utilisez une fonction `Thread.sleep(20)` pour limiter l'affichage à 50 images par seconde.

6. **Coordonnées homogènes**

Dans les questions précédents, nous avons travaillé avec des transformations linéaires dans un espace tridimensionnel en utilisant des matrices de dimension 3×3. Cependant, avec ces matrices, il est impossible d'effectuer certaines transformations géométriques, comme les translations. En effet, la translation n'est pas une transformation linéaire puisqu'elle ne conserve pas l'origine (c'est-à-dire, elle ne satisfait pas la propriété de linéarité selon laquelle l'image du vecteur nul doit être le vecteur nul).

Pour contourner cette limitation, nous allons introduire une représentation en **coordonnées homogènes**, qui ajoute une dimension à l'espace de travail. Cette extension nous permettra d'utiliser des matrices pour exprimer des transformations plus riches, comme les translations, les homothéties et même les projections vers un point de fuite.

Cours : Les coordonnées homogènes

Le passage en **coordonnées homogènes** consiste à ajouter une **dimension supplémentaire** à l'espace de travail. Cette dimension permet de représenter des points et des transformations géométriques d'une manière plus flexible, notamment pour les **translations**. En ajoutant cette coordonnée supplémentaire, souvent fixée à **1** pour plus de simplicité, nous pouvons interpréter l'espace tridimensionnel comme un espace situé à une **distance unitaire** de l'"observateur".
Cette représentation rend possible l'utilisation de **matrices** pour effectuer des transformations géométriques telles que les **homothéties**, les **rotations** et les **translations**.
En pratique, un **vecteur v** de dimension n est étendu en un **vecteur homogène** de dimension $n+1$.
Par exemple, un vecteur en dimension 3, $\mathbf{v} \in \mathbb{R}^3$, devient en coordonnées homogènes $\mathbf{v}_h \in \mathbb{R}^4$:

$$\mathbf{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \Rightarrow \quad \mathbf{v}_h = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Deux vecteurs homogènes $\mathbf{a}_h = (x, y, z, w)$ et $\mathbf{b}_h = (k \cdot x, k \cdot y, k \cdot z, k \cdot w)$ représentent le **même point**, mais vus depuis des **distances respectives** w et $k \cdot w$.
Pour revenir aux **coordonnées classiques**, il suffit de diviser chaque composante d'un vecteur homogène par sa **coordonnée supplémentaire** w. Par exemple, un vecteur homogène $\mathbf{t}_h \in \mathbb{R}^4$ correspond au vecteur classique $\mathbf{t} \in \mathbb{R}^3$:

$$\mathbf{t}_h = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \quad \Rightarrow \quad \mathbf{t} = \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix}$$

Applications en coordonnées homogènes

Les **coordonnées homogènes** facilitent la représentation de certaines **transformations géométriques** à l'aide de **matrices**.

Exemple : Homothétie homogène en 3D
Une **homothétie** de rapport k dans un espace 3D homogène peut être représentée par une **matrice** 4×4 définie comme suit :

$$\mathbf{H}_k = \begin{bmatrix} k & 0 & 0 & 0 \\ 0 & k & 0 & 0 \\ 0 & 0 & k & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Cette matrice, appliquée à un **vecteur homogène**, multiplie chaque **coordonnée spatiale** par k, effectuant ainsi une **homothétie** de rapport k tout en préservant la **coordonnée homogène**.

Exemple : Translation en 3D
En **coordonnées homogènes**, une **translation** en 3D peut également être représentée par une **matrice** 4×4. Supposons un **vecteur de translation** $\mathbf{t} = (t_x, t_y, t_z)$. La **matrice de translation** correspondante est :

$$\mathbf{T_t} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Lorsque cette matrice est appliquée à un **vecteur homogène** $(x, y, z, 1)$, elle effectue une **translation** en ajoutant (t_x, t_y, t_z) aux **coordonnées spatiales** du vecteur.
Ainsi, les **coordonnées homogènes** permettent de manipuler facilement des **transformations géométriques complexes**, incluant **translations** et **homothéties**, par de simples **opérations matricielles**.

Exemple : Translation d'un vecteur 3D avec des matrices homogènes

Considérons un vecteur 3D $\mathbf{v}$ et un vecteur de translation $\mathbf{t}$:

$$\mathbf{v} = \begin{bmatrix} 2 & 3 & -1 \end{bmatrix}^\top \quad \mathbf{t} = \begin{bmatrix} 5 & -2 & 3 \end{bmatrix}^\top.$$

Nous souhaitons appliquer la translation définie par $\mathbf{t}$ à $\mathbf{v}$ en utilisant une matrice homogène.

Étape 1 : Passage en coordonnées homogènes
Pour représenter le vecteur $\mathbf{v}$ en coordonnées homogènes, nous utilisons la méthode `Vecteur.versHomogene`. Cette méthode ajoute une coordonnée supplémentaire fixée à 1 au vecteur $\mathbf{v}$, obtenant ainsi :

$$\mathbf{v}_h = \begin{bmatrix} 2 & 3 & -1 & 1 \end{bmatrix}^\top.$$

Étape 2 : Construction de la matrice de translation
La méthode `Matrice.setTranslationHomogene3d` permet de construire une matrice homogène correspondant à une translation par $\mathbf{t}$. La matrice obtenue est :

$$\mathbf{T}_\mathbf{t} = \begin{bmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Étape 3 : Application de la translation
Nous utilisons la méthode `Matrice.multiplicationVectorielle` pour appliquer la matrice de translation $\mathbf{T}_\mathbf{t}$ au vecteur homogène $\mathbf{v}_h$. Le calcul se fait comme suit :

$$\mathbf{v}'_h = \mathbf{T}_\mathbf{t} \cdot \mathbf{v}_h = \begin{bmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 3 \\ -1 \\ 1 \end{bmatrix}.$$

Étape 4 : Retour en coordonnées classiques
Pour revenir aux coordonnées classiques, nous utilisons la méthode `Vecteur.depuisHomogene`. Cette méthode divise chaque composante par la coordonnée homogène ($w = 1$) et retourne un vecteur de dimension réduite :

$$\mathbf{v}' = \begin{bmatrix} 7/1 \\ 1/1 \\ 2/1 \end{bmatrix} = \begin{bmatrix} 7 \\ 1 \\ 2 \end{bmatrix}.$$

Ainsi, après la translation, le vecteur $\mathbf{v} = \begin{bmatrix} 2 \\ 3 \\ -1 \end{bmatrix}$ devient : $\mathbf{v}' = \begin{bmatrix} 7 \\ 1 \\ 2 \end{bmatrix}$.
Les méthodes utilisées dans cet exemple permettent de manipuler facilement des transformations homogènes pour appliquer des translations ou d'autres transformations géométriques en 3D.

Les classes `Matrice` et `Vecteur` codées précédemment permettent déjà de représenter des vecteurs et matrices en dimension arbitraire. Nous allons maintenant ajouter des méthodes spécifiques pour manipuler des vecteurs en coordonnées homogènes.

6.1. Ajoutez à la classe `Vecteur` une méthode publique statique `versHomogene` qui prend un `Vecteur` en paramètre et retourne une nouvelle instance de `Vecteur` représentant la version homogène du vecteur fourni, avec une distance standard de 1. Cette méthode doit ajouter une coordonnée supplémentaire de valeur 1 à la fin du vecteur.

Par exemple, si le vecteur fourni est (x, y, z), la méthode doit retourner le vecteur homogène $(x, y, z, 1)$.

6.2. Ajoutez à la classe `Vecteur` une méthode publique statique `depuisHomogene` qui prend un `Vecteur` en paramètre et retourne une nouvelle instance de `Vecteur` représentant la version "déhomogénéisée" du vecteur fourni. Cette méthode suppose que le vecteur fourni est en coordonnées homogènes (dernière coordonnée différente de zéro). Elle doit diviser chaque coordonnée par la coordonnée homogène (dernière coordonnée) et retourner un vecteur de dimension réduite, sans la dernière coordonnée.

Par exemple, si le vecteur fourni est (x, y, z, w), la méthode doit retourner le vecteur déhomogénéisé $\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$.

6.3. Ajoutez à la classe `Matrice` une méthode publique `setHomothetieHomogene3d` avec un paramètre `k` de type `double` pour représenter une homothétie de rapport k dans l'espace homogène 3D. Cette méthode doit modifier l'instance courante pour la transformer en matrice 4×4 correspondant à cette homothétie.

6.4. Ajoutez à la classe `Matrice` une méthode publique `setTranslationHomogene3d` avec un paramètre `t` de type `Vecteur` qui modifie l'instance courante pour qu'elle corresponde à une translation de vecteur `t` dans l'espace homogène 3D. On suppose que `t` est un vecteur de dimension 3.

6.5. **Affichage d'un cube avec translation après rotation**

Dans cette question, vous allez compléter une méthode `afficherCubeRotationTranslation` qui effectue les opérations suivantes :

1. Applique une rotation au cube dans l'espace en utilisant des matrices de rotation normales (non homogènes).
2. Passe les sommets du cube en coordonnées homogènes pour appliquer une translation.
3. Effectue une translation du cube à l'aide d'une matrice homogène de translation.
4. Revient en coordonnées classiques pour afficher le cube transformé.

Vous pouvez repartir du code que vous avez obtenu dans la question **Rotation composée du cube autour des trois axes**, car nous n'avons pas encore codé les matrices de rotation en coordonnées homogènes. Le principe est d'effectuer la rotation en coordonnées normales, puis de passer en coordonnées homogènes pour la translation.

6.6. **Observation lors d'une translation sur l'axe $\mathcal{O}_z$ après rotation** Ajoutez une méthode statique `afficherCubeRotationTranslationOz` dans la classe `Visualisation`. Cette méthode doit :

1. Appliquer une rotation composée autour des axes $\mathcal{O}_x$, $\mathcal{O}_y$, et $\mathcal{O}_z$ (par exemple, avec des angles $\alpha_x = \pi/4$, $\alpha_y = \pi/4$, $\alpha_z = \pi/4$) au cube.
2. Appliquer une translation le long de l'axe $\mathcal{O}_z$.
3. Afficher le cube transformé.

6.7. **Question d'observation :** Appelez la méthode `afficherCubeRotationTranslationOz` et observez la visualisation obtenue. Répondez aux questions suivantes :

1. Que remarquez-vous lorsque le cube est d'abord rotationné puis translaté le long de l'axe $\mathcal{O}_z$?
2. Expliquez pourquoi cet effet se produit en vous basant sur le fait que la projection utilisée est orthogonale sur le plan O_xO_y.

6.8. **Comparaison de l'effet d'une homothétie et d'une translation dans deux ordres différents après rotation**

Ajoutez une méthode statique `afficherCubeRotationHomothetieTranslation` dans la classe `Visualisation`. Cette méthode doit :

1. Appliquer une rotation composée autour des axes $\mathcal{O}_x$, $\mathcal{O}_y$, et $\mathcal{O}_z$ au cube.
2. Créer et afficher deux versions du cube transformé :
   - Une version où l'homothétie est appliquée avant la translation.
   - Une version où la translation est appliquée avant l'homothétie.

6.9. **Question d'observation :** Appelez la méthode `afficherCubeRotationHomothetieTranslatio` et observez la visualisation des deux cubes obtenus. Répondez aux questions suivantes :

1. Quelles différences remarquez-vous entre le cube où l'homothétie est appliquée avant la translation et celui où la translation est appliquée avant l'homothétie ?
2. Expliquez pourquoi l'ordre des transformations impacte la position et la taille du cube dans chaque cas.

6.10. **Animation avec translation, homothétie et retour à l'origine**

Ajoutez une méthode statique `afficherCubeAnimationTranslationHomothetieRetour` dans la classe `Visualisation`. Cette méthode doit réaliser une animation en trois étapes distinctes :

1. Appliquer une translation progressive pour déplacer le cube.
2. Appliquer une homothétie progressive pour agrandir le cube.
3. Appliquer une deuxième translation progressive pour ramener le cube à son origine.