Grenoble INP – ENSIMAG, UGA
École Nationale Supérieure d'Informatique et de Mathématiques Appliquées

Projet Génie Logiciel

Documentation de l'extension TAB

Yanis BOUHJOURA, Hugo DABADIE, Garance DUPONT-CIABRINI, Marek ELMAYAN, Julien LALANNE

GL09 Groupe 2

27 janvier 2022

Table des matières

1 Spécifications de l'extension

1.1 Spécifications prévues

1.1.1 Tableaux

- Déclarer un Tableau : *Must Have*

  Voici les trois manières de déclarer un tableau :

```
1 int[] array;
2 array = new int[taille];
3
4 int[] array = new int[taille];
5
6 int[] array = {0 , 1 , 2};
7
```

- Accès à un élément du tableau : *Must Have*

  L'accès à un élément du tableau se fait un appelant son indice entre crochet, c'est-à-dire, comme ceci :

```
1 int element = array[indice];
2
```

- Méthodes natives sur les tableaux : *Must Have*

  Il sera possible d'accéder à la longueur du tableau de cette manière :

```
1 int longueur = array.length;
2
```

1.1.2 Matrices

- Déclarer une matrice : *Must Have*

  Voici les trois manières de déclarer une matrice :

```
1 int[] matrix;
2 matrix = new int[longueur, largeur];
3
4 int[] matrix = new int[longueur, largeur];
5
6 int[] matrix = {{0 , 1 , 2}, {3, 4, 5}};
7
```

- Accès à un élément de la matrice : *Must Have*

L'accès à un élément de la matrice se fait un appelant son indice entre crochet, c'est-à-dire, comme ceci :

```
int element = matrix[ligne, colonne];

```

- Méthodes natives sur les matrices : *Must Have*

Il sera possible d'accéder à la longueur du tableau de cette manière :

```
int longueur = matrix.length;
int largeur = matrix.width;

```

1.1.3 Calcul Matriciel

- Matrice identité : *Nice to Have*

Il sera possible de créer une matrice identité de cette manière :

```
int[] matrix = new int[longueur, largeur];
matrix.setIdentity();

```

- Matrice nulle : *Nice to Have*

Il sera possible de créer une matrice nulle de cette manière :

```
int[] matrix = new int[longueur, largeur];
matrix.setZero();

```

- Sommes sur les matrices : *Must Have*

Il sera possible de faire la somme terme à terme de matrices de cette façon :

```
int[] matrixResult = matrix1.sum(matrix2);

```

- Multiplication par une constante : *Must Have*

Il sera possible de multiplier une matrice par une constante de cette manière :

```
1 int constant = 2;
2 int[] matrixResult = matrix.multConst(constant);
3
```

- Produit matriciel : *Must Have*

Il sera possible d'effectuer un produit matriciel comme ceci :

```
1 int[] matrixResult = matrix1.matrixProd(matrix2);
2
```

L'algorithme utilisé, afin d'avoir les meilleures performances possible, sera celui de Strassen.

- Transposée : *Should Have*

Il sera possible de calculer la transposée d'une matrice de cette façon :

```
1 int[] matrixResult = matrix.transpose();
2
3
```

- Déterminant : *Nice to Have*

Il sera possible de calculer un déterminant comme ceci :

```
1 float determinant = matrix.getDeterminant();
2
```

Pour calculer le déterminant rapidement, nous allons utilisé la décomposition LU, un des algorithmes les plus efficaces pour faire cela. De plus, on pourra envisager de stocker les matrices L et U afin d'optimiser l'inversion.

- Inverse : *Nice to Have*

Il sera possible de calculer une matrice inverse comme ceci :

```
1 float[] inverseMatrix = matrix.invert();
2
```

Tout comme le déterminant, nous calculerons l'inverse à partir de la décomposition LU.

- Valeurs propres : *Nice to Have*

Il sera possible de calculer les valeurs propres d'une matrice comme ceci :

```
1 float[] eigenvalues = matrix.getEigenvalues();
2
```

Cette dernière spécification sera sans doute compliqué à mettre en place. Une piste pour les algorithmes à utilisés sont la méthode de Jacobi ou la méthode de Givens.

1.2 Spécifications réalisées

1.2.1 Tableaux

- Déclarer un Tableau

Il est obligatoire de fournir la taille du tableau pour la création.

```
1 int[taille] arrayInt;
2 float[taille] arrayFloat;
3
```

Il est possible de déclarer plusieurs tableaux de même type en même temps de cette façon :

```
1 int[taille1] arrayInt1, [taille2]arrayInt2;
2 float[taille1] arrayFloat1, [taille2]arrayFloat2;
3
```

Toutes les valeurs d'un tableau déclarées sont initialisées à 0 au départ.

- Affecter des valeurs aux éléments du tableau

Pour affecter des valeurs aux différents indices d'un tableau il est nécessaire d'affecter les valeurs indice par indice. Pour parcourir les tableaux, il est possible d'utiliser une boucle while.

```
int[taille] arrayInt;
int i = 0;
while (i < taille) {
    arrayInt[i] = i;
    i = i + 1;
}

```

- Accès à un élément du tableau

L'accès à un élément du tableau se fait un appelant son indice entre crochet, c'est-à-dire, comme ceci :

```
int elementInt = arrayInt[indice];
float elementFloat = arrayFloat[indice];

```

- Méthodes natives sur les tableaux :

Il sest possible d'accéder à la longueur du tableau de cette manière :

```
int longueur = array.width;

```

1.2.2 Matrices

- Déclarer une matrice

Voici comment déclarer une matrice, il est obligatoire de fournir le nombre de lignes et de colonnes à la déclaration :

```
int[Ligne,Col] matrixInt;
float[Ligne,Col] matrixFloat;

```

Comme pour les tableaux, il est possible de réaliser des déclarations multiples :

```
int[Ligne1,Col1] matrixInt1, [Ligne2,Col2] matrixInt2;
float[Ligne1,Col1] matrixFloat1, [Ligne2,Col2] matrixFloat2;

```

Toutes les valeurs sont initialisées à 0 au début

- Affecter des valeurs aux éléments de la matrice

A l'instar des tableaux, il faut affecter les valeurs de la matrice une à une. Il est possible d'utiliser une double boucle while pour parcourir la matrice et y insérer les valeurs souhaitées.

```
int[Ligne,Col] matrixInt;
int i = 0, j = 0;
while (i < Ligne) {
    while (j < Col) {
        matrixInt[i,j] = i*j;
        j = j + 1;
    }
    i = i + 1
}

```

- Accès à un élément de la matrice

L'accès à un élément de la matrice se fait un appelant son numéro de ligne et de colonne entre crochet, c'est-à-dire, comme ceci :

```
int elementInt = matixInt[Ligne, Col];
float elementFloat = matrixFloat[Lignes, Col];

```

1.2.3 Fonctions sur les matrices

- Méthodes de la classe MatrixSet

SetIdentity transforme la matrice passée en paramètre (nécessairement carrée) en la matrice identité de même dimension.

```
void floatSetIdentity(float[] matrixFloat)
void intSetIdentity(int[] matrixInt)

```

SetZero transforme la matrice passée en paramètre en la matrice nulle de même dimension.

```
void floatSetZero(float[] matrixFloat)
void intSetZero(int[] matrixInt)

```

SetOne transforme la matrice passée en paramètre en la matrice unité (tous ses termes sont égaux à 1) de même dimension.

```
1 void floatSetOne(float[] matrixFloat)
2 void intSetOne(int[] matrixInt)
3
```

Pour utiliser ces méthodes de la classe MatrixSet, il faut créer une instance de cette classe.

```
1 MatrixSet ms = new MatrixSet();
2 ms.intSetIdentity(int[5,5] matrix);
3
```

- Méthodes de la classe MatrixOperations

Sum modifie mat1 en la somme des 2 matrices placées en paramètre (de même dimension)

```
1 void floatSum(float[] matrixFloat1, float[] matrixFloat2)
2 void intSum(int[] matrixInt1, int[] matrixInt2)
3
```

MultConst modifie la matrice en le produit de celle-ci et de la constante.

```
1 void floatMultConst(float[] matrixFloat1, float[] matrixFloat2)
2 void intMultConst(int[] matrixInt1, int[] matrixInt2)
3
```

Transpose stocke la transposée de la matrice input dans la matrice output, les dimensions doivent être vérifiées.

```
1 void floatTranspose(float[] matrixInput, float[] matrixOutput)
2 void intTranspose(int[] matrixInput, int[] matrixOutput)
3
```

Mult modifie matrix1 en produit des 2 matrices placées en paramètre (matrix1 et matrix2).

```
1 void floatMult(float[] matrix1, float[] matrix2)
2 void intTMult(int[] matrix1, int[] matrix2)
3
```

Pour utiliser ces méthodes de la classe MatrixOperations, il faut créer une instance de cette classe.

```
1 MatrixOperations mo = new MatrixOperations();
2 mo.intSum(int[5,5] matrix1, int[5,5] matrix2);
3
```

- Méthodes de la classe MatrixUtils

Equal retourne true si les 2 matrices passées en paramètres sont égales, false sinon.

```
1 boolean intMatrixEqual(int[] mat1, int[] mat2);
2 boolean floatMatrixEqual(float[] mat1, float[] mat2);
3
```

Print affiche le contenu de la matrice passée en paramètre.

```
1 \noindent void intMatrixPrint(int[] matrix);
2 void floatMatrixPrint(float[] matrix);
3
```

Copy copie matrixInput dans matrixOutput

```
1 void floatTranspose(float[] matrixInput, float[] matrixOutput)
2 void intTranspose(int[] matrixInput, int[] matrixOutput)
3
```

intToFloat convertie la matrice matrix en matrice de type flottant et la stocke dans la matrice output.

```
1 void intToFloat(int[] matrix, float[] output);
2
```

Pour utiliser ces méthodes de la classe MatrixUtils il est nécessaire de créer une instance de cette classe :

```
1 MatrixUtils mu = new MatrixUtils();
2 mu.intMatrixPrint(int[5,5] matrix);
3
```

2 Analyse bibliographique

2.1 Analyse des besoins

L'objectif de l'extension était de permettre l'utilisation de tableaux à 1 et 2 dimensions en deca. Il nous semblait donc évident dans un premier temps que la déclaration de ces éléments était clé dans la bonne réalisation du travail convenu sur l'extension. Ensuite, il fallait être capable d'initialiser les valeurs du tableau, de les modifier, ainsi que d'y avoir accès. Les tableaux devaient être de tailles variése. Pour cette partie du travail, aucune aide bibliographique extérieure n'était recquise. Aussi, nous avions dès le départ en tête que l'implémentation des tableaux en deca devait comprendre des tableaux d'entiers ainsi que des tableaux de flottants.

Dans un second temps, il a fallu se pencher sur la bibliothèque matricielle. La plupart des fonctionnalités implémentées sont simples et ne nécessitent qu'un parcours des matrices pour être appliquées, avec parfois des vérifications nécessaires sur les dimensions. Néanmoins, nous avons cherché à réaliser une bibliothèque matricielle des plus complètes. C'est pourquoi nous nous sommes penchés sur la décomposition LU qui permettait d'obtenir le déterminant ainsi que l'inverse d'une matrice de dimension quelconque. De surcroît, la complexité du produit matricielle étant très élevée, nous avons pensé à optimiser l'opération pour les matrices de grande taille avec l'algorithme de Strassen.

2.2 Implémentation des tableaux et des matrices

Dans un premier temps, nous nous sommes inspirés de la définition et de l'utilisation des tableaux en java. Nous visions la possibilité de déclarer des tableaux de 3 façons différentes tel qu'il est possible de faire en java. Cependant, nous avons simplement maintenu la déclaration telle que présentée dans les spécifications. Il en va de même pour les matrices.

2.3 Calcul matriciel

2.3.1 Produit matriciel

l'algorithme de Strassen est un algorithme calculant le produit de deux matrices carrées de taille n, proposé par Volker Strassen en 1969. La complexité de l'algorithme est en $O(n^{2,807})$ *avec pour la première fois un exposant inférieur à celui de la multiplication naïve qui est* $O(n^3)$ *Par contre, il a l'inconvénient de ne pas être stable numériquement.* (Source : Wikipédia)

L'idée de l'algorithme est relativement simple : elle consiste à diviser les matrices en blocs de taille égale et d'effectuer des opérations sur ces blocs pour obtenir 7 matrices intermédiaires qui par de simples combinaisons de sommes entre elles permettent de calculer les blocs composants la matrice produit. La clé de la réussite de cet algorithme réside dans le fait que les matrices intermédiaires sont chacunes obtenues avec une seule multiplication matricielle. Ainsi, si on compare cette méthode à la multiplication naïve, on se rend compte qu'elle nécessite une multiplication matricielle de moins sur les blocs. Les valeurs de complexité proposées dans le paragraphe précédent sont donc justifiées :

la multiplication matricielle naïve utilise $n^{log_2(8)} = n^3$ multiplications tandis que la multiplication par l'algorithme de Strassen utilise $n^{log_2(7)} = n^{2,807}$

Néanmoins, certaines limitations apparaissent : les additions ont un poids négligeable dans le calcul de la complexité car sur un grand nombre d'opérations elles sont moins couteuses que les multiplications. Cependant, lorsque la taille des matrices est petite, il est plus intéressant d'utiliser la multiplication naïve qui contient certes plus de multiplications, mais moins d'opérations arithmétiques au global. Enfin, il est nécessaire de modifier les matrices en entrée lorsqu'elles ne sont pas carrées ou que n n'est pas une puissance de 2 car le découpage en blocs doit être rigoureux pour que la méthode fontionne.

2.3.2 Déomposition LU

- Procédé

  En algèbre linéaire, la décomposition LU est une méthode de décomposition d'une matrice comme produit d'une matrice triangulaire inférieure L (comme lower, inférieure en anglais) par une matrice triangulaire supérieure U (comme upper, supérieure) (Source : Wikipédia)

  L'idée est d'éliminer colonne par colonne les termes en dessous de la diagonale pour obtenir la matrice U. Sur la nième colonne de la matrice de taille n, on élimine les éléments sous la diagonale de cette façon : $L_i = L_i \times l_{i,n}$ avec $l_{i,n} = -\frac{a_{i,n}^{(n-1)}}{a_{n,n}^{(n-1)}}$. Pour les autres colonnes, on multiplie par la gauche A par une matrice triangulaire inférieure de termes $l_{n+k}n$ sur la colonne en question, 1 sur la diagonale et 0 sinon.

  Ainsi, $U = L_{n-1}L_{n-2}...L_1A$ et $L = L_1^{-1}...L_{n-1}^{-1}$
- Déterminant

  $det(A) = det(L) \times det(U)$ Les 2 déterminants sont faciles à calculer car les matrices sont triangulaires.
- Inverse

  De même, il est facile d'inverser des matrices triangulaires. Ainsi, $A^{-1} = U^{-1}L^{-1}$ se calcule aussi sans difficultés.

3 Choix d'algorithmes et conception

3.1 Lexicographie

Afin de pouvoir implémenter les tableaux et les matrices, nous avons dû rajouter les crochets au lexer du compilateur. Nous avons fait comme cela :

- **OBRACKET** = '[' ;
- **CBRACKET** = ']' ;

3.2 Grammaires

3.2.1 Grammaire concrète

decl_var_set
→ type list_decl_var ';'
| matrix_type list_decl_matrix ';'

list_decl_matrix
→ decl_matrix (',' decl_matrix) *

decl_matrix
→ '[' list_expr ']' ident

primary_expr
→ ident
| ident '(' list_expr ')'
| ident '[' list_expr ']'
| '(' expr ')'
| 'readInt' '(' ')'
| 'readFloat' '(' ')'
| 'new' ident '(' ')'
| '(' type ')' '(' expr ')'
| literal

matrix_type
→ matrix_ident

matrix_ident
→ matrix_ident

matrix_ident
→ IDENT

3.2.2 Grammaire abstraite

EXPR
→ BINARY_EXPR
| LVALUE
| READ_EXPR
| STRING_LITERAL
| UNARY_EXPR
| BooleanLiteral↑*boolean*
| Cast[IDENTIFIER EXPR]
| FloatLiteral↑*float*
| InstanceOf[EXPR IDENTIFIER]
| IntLiteral↑*int*
| MethodCall[EXPR IDENTIFIER LIST_EXPR]
| MatrixCall[IDENTIFIER LIST_EXPR]
| New[IDENTIFIER]
| Null
| This↑*boolean*

DECL_MATRIX
→ DeclMatrix[IDENTIFIER LIST_EXPR IDENTIFIER INITIALIZATION]

3.3 Génération de code en assembleur

3.3.1 Schéma de l'allocateur mémoire

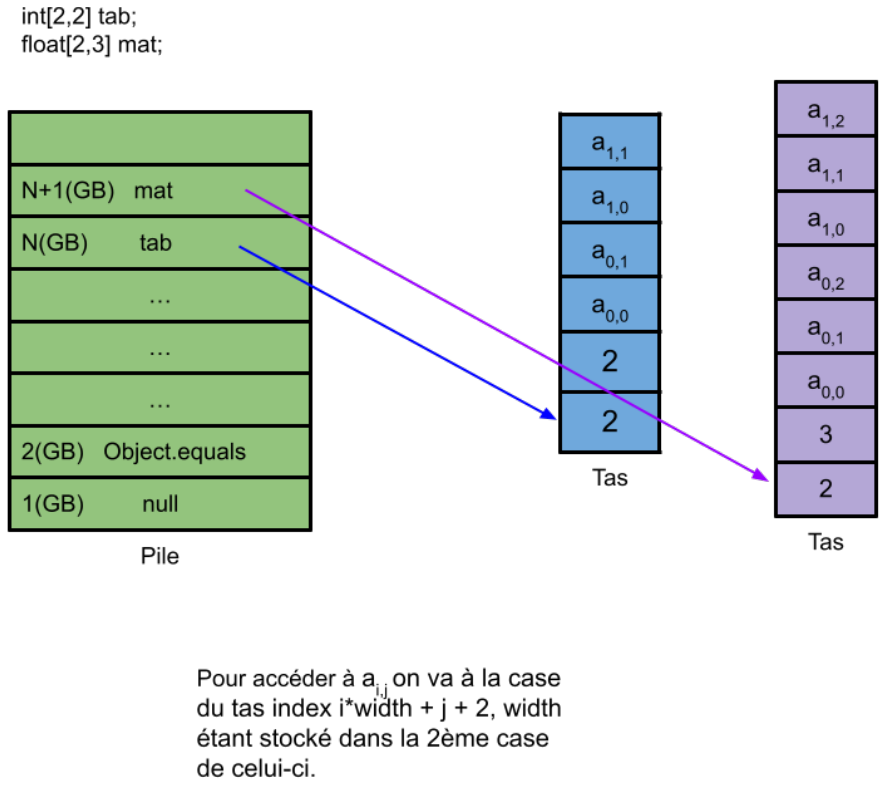

FIGURE 1 – Schéma de l'allocation mémoire.

3.3.2 Initialisation d'un tableau

Afin de pouvoir stocker en mémoire un tableau nous nous sommes inspirés du choix utilisé pour les classes. On stocke les attributs (fields) dans un tas, dans notre cas nous avons décidé d'également utiliser des tas pour stocker nos éléments d'un tableau. Lorsqu'on initialise un tableau, avec *int[2, 2] tab ;* en deca par exemple, on va appeler l'instruction NEW en assembleur et ensuite sauvegarder l'adresse qui pointe vers ce tas dans la case de la pile correspondant à notre variable global tab.
Dans ce cas là, les arguments entrés sont des entiers donc il aurait été facile de les stocker dans les fichiers java, dans *MatrixDefinition.java* par exemple. Cependant dès lors que l'on entre en argument une expression que l'on n'a pas encore évaluée, il nous est impossible de sauvegarder les paramètres de hauteur et de longueur d'un tableau dans les fichiers java.
Ainsi il s'est avéré nécessaire de stocker ces deux paramètres directement dans la structure de donnée initialisée en assembleur. Pour savoir le nombre de case a allouer par

l'instruction NEW, il suffit donc de faire *width*length + 2*.
Pour plus de détails sur les instructions assembleur utilisées il suffit d'aller voir la méthode *codeGenDeclVar()* dans le fichier *DeclMatrix.java*.
Nous avons essayé d'utiliser le moins de registre et d'instruction possible pour optimiser le code cependant concilier les deux n'était pas toujours possible car lorsque l'on utilise moins de registre il est nécessaire de faire plus d'instructions intermédiaires si l'on veut arriver au même résultat.
Ensuite on initialise les différents éléments du tableau à zéro. Pour cela il faut parcourir tous les éléments de notre tas grâce à l'équivalent d'une boucle while mais en assembleur où l'on incrémente l'adresse de la case actuelle après y avoir rangé la valeur 0.

3.3.3 Accès à un élément d'un tableau

La génération de cette partie de code se fait principalement dans le fichier *MatrixCall.java*.
On commence par récupérer l'adresse de la première case du tas avec les éléments de notre tableau. Ensuite on récupère la largeur du tableau située dans la 2ème case, ce qui nous permet de calculer l'index de la case de l'élément voulu. On a donc deux registres actifs : un avec la valeur de l'index, un autre avec l'adresse du tas. Ainsi on va faire une boucle while dans laquelle on décremente notre registre avec l'index jusqu'à obtenir 0 et en parallèle on décale l'adresse de la case de 1 à chaque fois. Finalement on se retrouve avec un registre qui possède l'adresse de la case voulue, on va stocker cette adresse dans un nouveau registre dans les cas où l'on veut attribuer une nouvelle valeur à notre élément du tableau mais on va également stocker la valeur déjà présente dans la case dans le cas où l'on veut seulement récupérer la valeur d'un élément de notre tableau.

3.3.4 Les limitations de notre implémentation

La première fonctionnalité que l'on a pas implémentée est l'initialisation d'un tableau avec ses propres valeurs. Comme cela nous posait des problèmes dès l'étape B, nous n'avons pas non plus implémenté la génération de code. Cependant cette fonctionnalité peut être facilement remplacée en initialisant à 0 par défaut puis faire des accès aux éléments voulus.
La seconde limitation concerne les tableaux déclarés en tant qu'attribut d'une classe qui produisent parfois des erreurs de code assembleur.
Ensuite l'instruction *return* dans une méthode ne va pas fonctionner comme souhaité lorsque que l'on lui donne un tableau en argument. Aucune erreur sera levée à l'exécution cependant les valeurs du tableau censé récupérer la valeur fournie dans le return ne seront pas modifiées.
Enfin nous n'avons pas implémenté une erreur personnalisée lorsque les indices de l'élément voulu dépasse la capacité du tableau. Pour résoudre cela nous pourrions récupérer la hauteur et la largeur de notre tableau stocké dans les deux premieres cases du tas et ensuite faire une comparaison à chaque incrémentation de la boucle while.

3.4 Calcul matriciel

Après avoir implémenté les tableaux ainsi que les matrices, nous avons mis en place une bibliothèque de calcul matriciel. Certaines fonctionnalités de calcul que nous voulions implémenter au départ n'ont pas pu l'être par manque de temps. D'autres ont été ajoutées aux fonctions proposées en amont car jugées utiles.

Dans cette partie, nous allons présenter en détails les différentes fonctionnalités proposées par notre bibliothèque matricielle. Il faut aussi savoir que nous avons décidé d'implémenter pour chaque fonctionnalité 2 fonctions distinctes : une agissant sur les int et une autre sur les float.

3.4.1 Matrices de référence

Pour faciliter la définition des matrices de référence telle que la matrice nulle, la matrice identité ou la matrice unité (dont tous les termes valent 1), nous avons implémenté des fonctions qui changent les valeurs de la matrice en entrée pour la convertir en la matrice désirée.

- Matrice Identité : *SetIdentity*

  Convertit la matrice en entrée en la matrice identité de même dimension. **floatSetIdentity** prend une matrice de flottants en entrée tandis que **intSetIdentity** prend une matrice d'entiers en entrée.

  Une des particularités de la matrice identité est qu'elle doit nécessairement être carrée. Ainsi, si la matrice en entrée ne l'est pas, la fonction print un message d'erreur : *"The matrix must be square"*

```
1 int[n,m] intMatrix;
2 intSetIdentity(intMatrix);
3
4 float[n,m] floatMatrix;
5 floatSetIdentity(floatMatrix);
6
```

- Matrice Nulle : *SetZero*

  Convertit la matrice en entrée en la matrice nulle de même dimension. **floatSetZero** prend une matrice de flottants en entrée tandis que **intSetZero** prend une matrice d'entiers en entrée.

```
1 int[n,m] intMatrix;
2 intSetZero(intMatrix);
3
4 float[n,m] floatMatrix;
5 floatSetZero(floatMatrix);
6
```

- Matrice Unité : *SetOne*

  Convertit la matrice en entrée en la matrice unité de même dimension. **floatSetOne** prend une matrice de flottants en entrée tandis que **intSetOne** prend une matrice d'entiers en entrée.

  Ces fonctions n'avaient pas été proposées à l'origine. Cependant elles paraissaient utiles pour réaliser et tester des calculs matriciels.

```
1 int[n,m] intMatrix;
2 intSetOne(intMatrix);
3
4 float[n,m] floatMatrix;
5 floatSetOne(floatMatrix);
6
```

3.4.2 Opérations sur les matrices

Nous avons ensuite implémenté les opérations de base sur les matrices, à savoir : la somme de 2 matrices, le produit de 2 matrices entre elles, la multiplication d'une matrice par un scalaire et enfin la transposition d'une matrice. Comme pour la section précédente des matrices de référence, chacune des opérations possède 2 versions, une s'appliquant aux matrices d'entiers et une aux matrices de flottants.

- La Somme matricielle : *Sum*

  Récupère 2 matrices de même type en entrée et transforme la matrice de gauche en la somme terme à terme des 2. **intSum** renvoie la somme de 2 matrices d'entiers tandis que **floatSum** renvoie la somme de 2 matrices de flottants.

  Bien entendu, il est nécessaire que les 2 matrices soient de même dimension afin de réaliser la somme terme à terme. On effectue donc une vérification sur la dimension de 2 matrices. Si elles ne coincident pas, on print un message d'erreur : *Incompatible sizes in floatSum* pour l'opération sur les flottants et *Error : Incompatible sizes in intSum* pour l'opération sur les entiers.

```
1 int[n,m] intMatrix1, [n,m] intMatrix2;
2 intSum(intMatrix1, intMatrix2);
3
4 float[n,m] floatMatrix1, [n,m] floatMatrix2
5 floatSum(floatMatrix1, floatMatrix2);
6
```

- Le Produit matriciel : *Mult*

Récupère 3 matrices de même type en entrée et insère le produit des 2 premières matrices dans la 3e. **intMult** calcule le produit de 2 matrices d'entiers tandis que **floatMult** calcule le produit de 2 matrices de flottants.

Il faut ici effectuer 3 vérifications sur la taille des matrices en entrée : D'une part, le nombre de colonnes de la matrice gauche doit coincider avec le nombre de lignes de la matrice droite. D'autre part, la matrice en sortie doit avoir des dimensions qui correspondent à celles des matrices en entrée : le nombre de lignes de la matrice de gauche doit coincider avec le nombre de lignes de la matrice en sortie et le nombre de colonnes de la matrice de droite doit coincider avec le nombre de colonnes de la matrice en sortie. Autrement dit :

$leftMatrix \in \mathcal{M}[n,p]$
$rightMatrix \in \mathcal{M}[p,m]$
$returnMatrix \in \mathcal{M}[n,m]$

Nous avions la volonté d'optimiser ce calcul en utilisant l'algorithme de Strassen pour les matrices de petite taille. Cependant, n'étant absolument pas essentielle, nous n'avons pas investi de temps dans la finalisation de cet algorithme.

```
1 int[n,p] intMatrix1, [p,m] intMatrix2, [n,m] intMatrixMult;
2 intMatrixMult = intMult(intMatrix1, intMatrix2);
3
4 float[n,p] floatMatrix1, [p,m] floatMatrix2, [n,m] floatMatrixMult;
5 floatMatrixMult = floatMult(floatMatrix1, floatMatrix2);
6
```

- La Multipliation par un scalaire : *MultConst*

Récupère 1 matrice et un scalaire de même type que les éléments de la matrice en entrée et multiplie chaque terme de la matrice par ce scalaire. **intMultConst** calcule cette multiplication pour des entiers et **floatMult** la calcule pour des flottants.

```
1 int[n,p] intMatrix;
2 int i = k;
3 intMultConst(intMatrix, i);
4
5 float[n,p] floatMatrix1;
6 float f = l;
7 floatMultConst(floatMatrix1, floatMatrix2);
8
```

- La transposition : *Transpose*

Récupère 2 matrices en entrée et insère la transposée de la première matrice dans la deuxième **intTranspose** calcule la transposée d'une matrice d'entiers alors que **floatTranspose** la calcule pour des flottants.

```
1 int[n,p] intMatrix1, [p,n] intMatrix2;
2 intTranspose(intMatrix1, intMatrix2);
3
4 float[n,p] floatMatrix1, [n,p]floatMatrix2;
5 floatTranspose(floatMatrix1, floatMatrix2);
6
```

3.4.3 Méthodes Utiles

Nous avons finalement implémenté des méthodes utiles à l'utilisation courante des matrices. La fonction print nous paraissait essentielle pour afficher des matrices. La possibilité de convertir une matrice d'entiers en matrice de flottants était surtout importante lorsqu'on avait limité les opérations matricielles aux flottants. Au final, nous avons pris la décision d'étendre les possibilités de nos opérations sur les 2 types, ainsi la fonction **intToFloat** perd un peu de son utilité, c'est pour cela que l'on a pas implémenté une fonction **floatToInt**. Enfin la possibilité de copier une matrice et de vérifier l'égalité entre 2 matrices sont des méthodes qui nous paraissaient utiles.

- Le Print : *Print*

Récupère 1 matrice en entrée et print tous ses termes. **intMatrixPrint** réalise le print d'une matrice d'entiers et **floatMatrixPrint** s'occupe des matrices de flottants.

```
1 int[n,p] intMatrix;
2 intMatrixPrint(intMatrix);
3
4 float[n,p] floatMatrix;
5 floatMatrixPrint(floatMatrix);
6
```

- La vérification de l'égalité : *Equal*

Récupère 2 matrices en entrée et vérifie l'égalité terme à terme. Si la dimension des 2 matrices est différente ou qu'un des termes d'une matrice est différent de son homonyme dans la 2e matrice, alors la fonction renvoie false. Sinon, elle renvoie true. **intMatrixEqual** compare 2 matrices d'entiers quand **floatMatrixEqual** compare 2 matrices de flottants.

```
1 int[n,p] intMatrix1, [n,p] intMatrix2;
2 intMatrixEqual(intMatrix1, intMatrix2);
3
4 float[n,p] floatMatrix1, [n,p]floatMatrix2;
5 floatMatrixEqual(floatMatrix1, floatMatrix2);
```

```
6
```

- La copie : *Copy*

  Récupère 2 matrices en entrée et copie tous les termes de la 1e matrice dans la 2e. Si la dimension des 2 matrices est différente, une erreur est print : *Error : Incompatible sizes in MatrixCopy*. **intMatrixCopy** s'occupe de la copie pour les matrices d'entiers et **floatMatrixCopy** pour les matrices de flottants.

```
1 int[n,p] intMatrix1, [n,p] intMatrix2;
2 intMatrixCopy(intMatrix1, intMatrix2);
3
4 float[n,p] floatMatrix1, [n,p]floatMatrix2;
5 floatMatrixCopy(floatMatrix1, floatMatrix2);
6
```

- La conversion d'entiers en flottants : *intToFloat*

  Récupère 2 matrices en entrée : une matrice d'entiers et une matrice de flottants de même dimension. Si les dimensions ne coincident pas, on affiche : *Error : Incompatible sizes in MatrixCopy*. La fonction insère les termes de la 1e matrice dans la 2e matrice en convertissant chacun d'eux en flottants.

```
1 int[n,p] intMatrix;
2 float[n,p] floatMatrix;
3 intToFloat(intMatrix, floatMatrix);
4
```

4 Méthode de validation

4.1 Types de tests

Pour l'extension, nous n'avons pas réalisé de test unitaire. En effet, par manque de temps, nous avons préféré nous concentrer sur la réalisation de tests systèmes. Ces tests permettent de vérifier plus de fonctionnalités à la fois qu'un test unitaire. L'avantage est donc qu'il y a besoin de réaliser moins de tests systèmes que de tests unitaires pour vérifier autant de fonctionnalités. De plus, écrire un test unitaire est plus long que d'écrire un test système. Toutefois, l'erreur est moins précise et on ne sait pas directement quelle partie du code a provoqué cette dernière.

4.2 Organisation des tests

L'organisation des tests reprend celle faite pour toute le langage Deca complet sans l'extension. Dans chaque répertoire cité dans la documentation de validation tel que

src/test/deca/context/valid ou encore tel que *src/test/deca/codegen/invalid*, nous avons rajouter un dossier *extension* contenant tous les tests que nous avons faits.

4.3 Autres méthodes de validation

Nous avons également utilisé d'autres méthodes de validations. En effet, afin de vérifier que le produit matriciel fonctionnait, par exemple. Nous avons simplement effectué le produit matriciel de deux matrices avec différentes valeures à l'aide d'un logiciel de calcul et comparé les résultats obtenus par les deux méthodes.

De plus, plusieurs personnes ont relu la bibliothèque matricielle afin de vérifier notamment l'exactitude des algorithmes utilisés ou encore la qualité de l'affichage des matrices.

5 Validation de l'implémentation

5.1 Résultats positifs

Les résultats des tests montrent que la majorité des fonctionnalités voulues sur les tableaux et les matrices marchent. Il est possible de déclarer des tableaux et des matrices, que cela soit contenant des entiers ou bien des flottants. On peut également affecter à ces derniers une valeur qui correspond à une expression telle que par exemple :

```
1 int[7, 20] matrix;
2 matrix[1, 2] = 2 * (4 + 1 / 4) % 2
3
```

De plus, nous avons testé la bibliothèque de calcul matriciel et les fonctionnalités d'affectation des matrices identité, nulle ou unité sont opérationnelles. Aussi, il est possible de multiplier une matrice par une constante, d'ajouter deux matrices et de multiplier deux matrices.

La transposition, la copier et l'affichage de matrice sont également possibles et fonctionnelles dans la bibliothèque de calcul matriciel. La conversion d'une matrice d'entier en matrice de flottant est également possible. De même, l'égalité de deux matrices fonctionne.

5.2 Résultats négatifs

Après réalisation des tests, nous nous sommes aperçu que quelques fonctionnalités ne fonctionnaient pas. En effet, il n'est pas possible de renvoyer un tableau ou une matrice. De plus, la déclaration dans un champ d'une classe d'un tableau ou d'une matrice ne fonctionnent pas.

Par conséquent, nous n'avons pas pu implémenter l'algorithme de Strassen pour la multiplication du fait de la récursivité de l'algorithme. De même, la décomposition LU pour le calcul de l'inverse et du déterminant d'une matrice.

6 Annexes

- Algorithme de Strassen
- Décomposition LU
- Méthode de Jacobi et méthode de Givens