Documentation extention tab
Compilateur Deca

Projet Génie Logiciel, groupe 15

January 27, 2023

Contents

1 Analyse bibliographique

1.1 Définition de matrice

La définition de matrice proposée à partir de l'extension reprend la même forme que la définition de matrice en java. Il est possible de définir des matrices à une et deux dimensions avec un nombre d'éléments fixés à la création de la matrice là où en java on

peut ne pas spécifier le nombre d'éléments présents dans la matrice à sa création. Les tableaux créés peuvent contenir des éléments de type int, float, boolean, Object et des types de classe définies en amont du programme main.

Exemple:

```
{
int[] tab1D = new int[5]; // Cree un tableau  d entiers  a une dimension de 5 elements
float[][] tab2D = new float[3][2]
// Cree un tableau de flottants a deux dimensions de taille 3x2 (3 lignes et 2 colonne
}
```

1.2 Bibliothèque de calcul matriciel Math

Les idées d'opération à implémenter pour construire la bibliothèque de calcul matriciel ont été dirigées par des ressources déjà présentes: la bibliothèque NumPy du langage Python. Cette bibliothèque est fondamentale pour la définition et la manipulation de matrices pour le langage de programmation Python. Les idées de calcul inspirées de la bibliothèque NumPy sont: la somme, la différence, le produit matriciel, le produit par une constante, la transposition, le calcul de la trace, du déterminant et l'inversion de matrice. Plusieurs de ces calculs nécessitent l'utilisation d'algorithmes pour lesquels des recherches ont également été effectuées.

Source numpy:
`https://numpy.org/doc/1.24/user/whatisnumpy.html`

1.3 Les recherches effectuées sur les algorithmes

Les opérations nécessitant la recherche d'algorithmes pour être implémentés sont le calcul de déterminant et d'inverse de matrice. Des recherches ont également été effectuées afin de trouver un algorithme de calcul de produit matriciel plus efficace que le calcul traditionnel effectué en mathématiques qui se traduit par une complexité en O(n3). La formule utilisée pour calculer le produit matriciel traditionnel est la suivante:

Soient A = (a_ij) de taille (m,n) et B = (b_ij) de taille (n,p). On a alors C = AB = (c_ij) de taille (m,p) où les coefficients c_ij sont calculés à partir de la formule suivante:

$$\forall i, j : c_{ij} = \sum_{k=1}^{n} a_{ik} a_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + ... + a_{in} b_{nj}$$

Une méthode plus efficace pour calculer le produit matriciel est l'algorithme de Strassen qui consiste à décomposer les matrices en 4 matrices par bloc mais se limite aux matrices carrées. La complexité de cet algorithme est $O(n^{2,807})$.

A, B et C sont des matrices carrées de taille n et $A_{1,1}, A_{1,2}, A_{2,1}$ et $A_{2,2}$ les matrices par bloc de A de taille n/2. Cela implique donc que la taille des matrices soit une puissance de deux.

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

Au lieu d'effectuer le produit matriciel classique qui impliquerait de faire 8 produits des matrices par bloc ($C_{1,1} = A_{1,1}.B_{1,1} + A_{1,2}.B_{2,1}$ etc.) On effectue les 7 calculs intermédiaires suivants:

$$M_1 := (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$$
$$M_5 := (A_{1,1} + A_{1,2})B_{2,2}$$
$$M_2 := (A_{2,1} + A_{2,2})B_{1,1}$$
$$M_6 := (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$$
$$M_3 := A_{1,1}(B_{1,2} - B_{2,2}$$
$$M_7 := (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$$
$$M_4 := A_{2,2}(B_{2,1} - B_{1,1}$$

A partir de ces 7 matrices intermédiaires on peut calculer les blocs de la matrice C.

$$C_{1,1} = M_1 + M_4 - M_5 + M_7$$
$$C_{2,1} = M_2 + M_4$$
$$C_{1,2} = M_3 + M_5$$
$$C_{2,2} = M_1 - M_2 + M_3 + M_6$$

Cela permet d'effectuer 7 produits de matrice par bloc au lieu de 8. Le problème avec cet algorithme est qu'il est très spécifique. Il est cependant possible d'augmenter la taille des matrices jusqu'à la prochaine puissance de 2 pour gérer la condition sur la taille des matrices.

Source Strassen:
`https://fr.wikipedia.org/wiki/Algorithme_de_Strassen#Notes_et_r%C3%A9f%C3%A9rences`

Pour calculer le déterminant de manière efficace, des recherches ont été effectuées sur l'algorithme d'élimination de Gauss. Cet algorithme transforme la matrice en matrice triangulaire inférieure ce qui permet de récupérer le déterminant qui est alors le produit des coefficients de la diagonale de la matrice triangulaire. La complexité est en O(n3).

Pour calculer l'inverse d'une matrice, on s'est intéressé à l'algorithme de Gauss Jordan qui consiste à transformer une matrice carrée de déterminant non nul A en identité et effectuer les mêmes opérations sur une matrice identité B, une fois la matrice A transformée en identité, la matrice B est alors devenue l'inverse de la matrice A.

Exemple: Soit A la matrice de taille 3x3 suivante:

$$A = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$$

On génère la matrice augmentée suivante et on effectue des opérations de combinaison linéaire sur la partie de gauche pour la transformer en identité.

$$[A|I] = \left[\begin{array}{ccc|ccc} 2 & -1 & 0 & 1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 1 & 0 \\ 0 & -1 & 2 & 0 & 0 & 1 \end{array}\right]$$

La matrice augmentée obtenue après transformation est le suivant:

$$[I|B] = \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & \frac{3}{4} & \frac{1}{2} & \frac{1}{4} \\ -0 & 1 & 0 & \frac{1}{2} & 1 & \frac{1}{2} \\ 0 & 0 & 1 & \frac{1}{4} & \frac{1}{2} & \frac{3}{4} \end{array}\right]$$

La matrice B récupérée ici est l'inverse de la matrice A.

Sources élimination de Gauss-Jordan:
`https://fr.wikipedia.org/wiki/%C3%89limination_de_Gauss-Jordan`
`https://stackoverflow.com/questions/66192894/precise-determinant-of-integer-nxn-matrix`

2 Conception des tableaux

2.1 Introdution

Les tableaux sont une structure de données très importante quand on essaye de manipuler plusieurs éléments, il en existe plusieurs implémentations possibles qu'on pouvait utiliser pour les représenter. Notre objectif principal lors de la conception de cette extension était de la lier au plus possible à l'architecture existence afin d'exploiter les fonctionnalités développées et écrire le moins de code possible.

Le premier choix qu'on a pris était de se limiter à des tableaux de dimension 1 et 2 car cela répond à tous nos besoins et une généralisation pour une dimension peut apporter une complexité supplémentaire.

Un autre choix qu'on a fait a été de ne pas initialiser les tableaux initialement à la valeur zéro entier ou zéro flottant ou null pour les tableaux de classes, on a senti comme si on faisait un choix pour l'utilisateur.

L'accès à la taille du tableau est quelque chose qui très important pour pouvoir les parcourir et modifier les valeurs et ca aussi important pour faire des vérifications sur les

dimensions des matrices avant de faire des opérations dessous par exemple quand on fait la multiplication entre des matrices, il est très important de faire des vérification sur les dimensions de la matrice et jeter une erreur si l'opération n'est pas possible.

2.2 Enrichissement des types du compilateur

Un nouveau type a été ajouté afin de gérer les tableaux avec le nom table TableType et une définition a été aussi associé à ce type avec le nom TableDefinition. Le nouvelle classe de contient le type des éléments du tableau et un champ avec le dimension pour indiquer si c'est un tableau 1d ou 2d. Par exemple un tableau 1d d'entier int[] a comme dimension égale à 1 et le type de l'élément du tableau sont entiers.

Ajout de nouveaux types associées à ces tableaux pour les tableaux avec les types de bases: int, float, boolean et Object dans EnvironmentType. Pour chaque classe définit on ajoute aussi des types de tableaux associées à la classe, par exemple si on définit la classe A les types A[] et A[][] sont ajoutés aux types d'environnement.

Le code suivant montre l'addition de float[] dans l'environnement de type:

//On crée le symbol associé à float[];

```
Symbol floatTableSymb1D = compiler.createSymbol("float[]");
```

//On crée un nouveau TableType qui prend le nouveau symbol créé et le type(FLOAT) des éléments du tableaux et la dimension(1) comme arguments.

```
TABLEFLOAT1D = new TableType(floatTableSymb1D, FLOAT, 1);
envTypes.put(floatTableSymb1D, new TableDefinition(TABLEFLOAT1D, Location.BUILTIN));
```

D'autres approches qui n'exploitent envtypes peuvent être faites, mais notre objectif est d'intégrer notre travail à l'architecture existence et exploité ce qui a été déjà réalisée pour la vérification de type et la bonne correspondance de type lors de l'attribution de valeur.

2.3 Opérations sur les tableaux

2.3.1 Déclaration des tableaux

La première chose dont on a travaillé dessous dans la partie syntaxique de l'extension était la déclaration des tableaux et afin de faire cela il fallait étendre la grammaire qu'on avait avec des nouvelles règles. Pour la déclaration des tableaux, on a juste modifié la règle de ident.

```
ident -> IDENT
| IDENT OBRACKET CBRACKET
| IDENT OBRACKET CBRACKET OBRACKET CBRACKET
```

avec OBRACKET = "[" ET CBRACKET = "]"

Avec ces nouvelles règles on peut déclarer un tableau si on met un type au début et après des crochets, une paire de crochet pour un tableau 1d et 2 paires de crochets pour une matrice. Un string est généré au niveau antlr en combinant le text de IDENT et les crochets, et on vérifie que ce qui a été donnée en entrée s'agit bien d'un type qui est disponible dans les types de l'environement Ainsi si l'utilisateur donne le code suivant

```
{
    int[] tableaux;
}
```

on validera que ce type "int[]" est valide. Au niveau de la vérification contextuelle et la génération de code, il n'y a rien à faire car les méthodes déjà développées font tout le nécessaire.

2.3.2 Association de l'espace mémoire au tableaux

L'association de l'espace au tableau nécessitait plus de travail que la déclaration des tableaux. La première étape était de créer de nouvelles règles de grammaire afin que nous puissions reconnaître le code écrit par l'utilisateur. Il fallait ajouter une nouvelle dérivation dans expr et dans plus précisement primary_expr, les deux nouvelles règles sont les suivantes.

```
primary_expr ->     .
| NEW ident OBRACKET  expr CBRACKET
| NEW ident OBRACKET expr CBRACKET OBRACKET  expr CBRACKET
```

On utilisées le même token NEW que dans la déclaration de class, heureusement que les grammaires de antlr sont LL(*) et non pas LL(1) sinon ce token ne pouvait être réutilisé de la même manière. Les expr entre les crochets présentent les arguments données lors de cette opération d'initialisation.

Afin d'avoir un nœud dans l'arbre dans java, on ajouté une nouvelle classe qui s'appelle NewTable et qui prend en paramètre un AbstractIdentifier et une listExpr. Le AbstractIdentifier est pour le type qu'on veut définir et la listExpr est utilisée pour pourvoir accéder aux arguments qui ont été données entre les crochets lors de l'entrée dans cette règle.

L'objectif de l'étape de vérification et de s'assurer que les expr donnent en arguments s'agit bien d'entier sinon une erreur est lancée avec le message "Le type de l'initialisation de la matrice n'est pas un int". L'autre étape de la vérification est de valider que le type donnée s'agit bien d'un type de tableau qu'on peut utiliser sinon une erreur est lancé avec le message : "Ce type de tableau n'existe pas"

La figure 1 ci-dessus montre la représentation en mémoire des tableaux qu'on a choisi, pour la un tableau 1d on met le nombre d'éléments à la première case et au dessus on met les éléments du tableau et dans le tableau 2d on met dans les deux premières cases le nombre de lignes et de colonnes et au dessus de cela on met les éléments de la matrice.

(a) Représentation graphique de la mémoire d'un tableau 1d

(b) Représentation graphique de mémoire d'une matrice

Figure 1: Représentation graphique de la mémoire des tableaux

L'exemple suivant de code sera utilisées afin de pouvoir expliquer la génération de code pour l'initialisation de tableaux :

```
{
        int[] tableauInt = new int[5];
        float[] tableauFloat = new float[10][10];
}
```

La génération de code se séparer en trois parties:

1-Vérification que les entiers donnée sont strictement positif, et cela ce fait une simple vérification que les entrées sont inférieur ou égal à 0 sinon une erreur d'exécution est lancée avec le nom

"int_allocation_table_must_be_strictly_positive".

Voici le code assembleur généré lors de l'exécution de l'initialisation pour un tableau 1d:

```
; [NewTable] with type int[]
LOAD #5, R3     ; loading 5 into memory
CMP #1, R3
BLT int_allocation_table_must_be_strictly_positive
```

Voici le code assembleur généré lors de l'exécution de l'initialisation pour un tableau 2d:

```
; [NewTable] with type float[][]
LOAD #10, R3    ; loading 10 into memory
CMP #1, R3
BLT int_allocation_table_must_be_strictly_positive
LOAD #10, R4    ; loading 10 into memory
CMP #1, R4
BLT int_allocation_table_must_be_strictly_positive
```

2-La deuxième étape est allouée d'espace mémoire pour les tableaux. Pour les tableaux 1d on alloue l'argument donné par l'utilisateur plus 1 pour stocker le nombre d'éléments dans le tableau. Pour les matrices, on alloue le produit des arguments plus 2 pour stocker

les dimensions de la matrice. Si jamais l'utilisateur essaye de réserver plus d'espace que disponible dans le tas, une erreur est lancée avec le nom: heap_overflow_error.

Les registres R1 et R0 ont été utilisés temporairement car ne voulait pas modifier les valeurs stockées dans les autres registres ou utiliser le stack car on travaille principalement avec des tableaux 1d et 2d et cela n'était pas nécessaire.

Voici le code assembleur généré lors de l'exécution de l'initialisation pour un tableau 1d:

```
LOAD R3, R1; Getting the number elements of the array from R3
ADD #1, R1; adding 1 to reserve space to store the number of elements
NEW R1, R0; memory space reservation
BOV heap_overflow_error; detects if more space than available has been allocated
```

Voici le code assembleur généré lors de l'exécution de l'initialisation pour un tableau 2d:

```
LOAD R3, R1; retrieving row count from R3
MUL R4, R1; multiply number of rows by number of columns
ADD#2, R1; Addition of 2 to be able to store rows and columns
NEW R1, R0; memory space reservation
BOV heap_overflow_error;  detects if more space than available has been allocated
```

3- La troisième étape est consacré au stockage de la taille du tableau dans la mémoire, pour pouvoir les accéder éventuellement. Pour un tableau 1d on stocke dans un offset de zéro de l'adresse du tableau le nombre d'elments. Pour le tableau 2d, on stocke dans un offset de l'adresse du tableau le nombre de lignes et dans un offset de 1 le nombre de colonnes. Voici le code assembleur généré lors la sauvegarde de ces éléments du tableau 2d donnée dans l'exemple:

```
STORE R3, 0(R0); storing the number of lines
STORE R4, 1(R0); storing the number of columns
```

2.3.3 Attribution des valeurs aux éléments du tableaux et récupération des éléments

Attribution des valeurs aux éléments du tableaux et récupération des éléments L'accès et l'attribution des valeurs aux tableaux exige aussi une extension de la grammaire avec des nouvelles règles. Ce qui été ajouté dans la grammaire sont aussi des expr et des dérivations de primary_expr

```
primary_expr -> . . .
| ident OBRACKET  expr CBRACKET
| ident OBRACKET expr CBRACKET OBRACKET  expr CBRACKET
```

Ident représente l'identificateur du tableau, et les expr représentent les arguments utilisés pour sélectionner les éléments du tableaux.

Une nouvelle classe a été introduit avec le nom TableGetElement qui prend en entrée lors de la construction un identificateur qui est le nom du tableau qu'on utilise et ca prend aussi en entrée une liste d'expression ListExpr similairement à ce la class NewTable auquel on mettra les expressions des arguments du tableaux.

La vérification contextuelle doit assurer que les arguments donnés sont entiers et ça vérifie aussi l'identificateur donné est défini et il s'agit bien un tableau et que la taille de la liste des arguments correspond bien à la dimension du tableau qui ne peut être que 1 et 2.

L'exemple suivant de code sera utilisées afin de pouvoir expliquer l'attribution et la récupération des éléments du tableau:

```
{
        int[] tableauInt = new int[5];
        float[][] tableauFloat = new float[10][10];
        tableauInt[1] = 2;      tableauFloat[3][5] = 2.5;
        println((float)(tableauInt[1]) + tableauFloat[3][5]);
}
```

La génération de code pour cette partie se fait sur quatre étapes: 1- La première est comme celle mentionnée dans la partie de l'initialisation mais cette fois on vérifie que les expr données sont supérieur ou égal à 0 **car les index commencent par 0 au lieu 1**.

2- La deuxième étape met l'adresse du tableau dans un registre, et vérifie l'adresse du tableau n'est pas égal à null, dans le cas ou c'est égal à null un erreur est déclenchée avec le nom deref_null_error. Voici un exemple de code assembleur de cette étape :

```
LOAD 1(GB), R1  ;  loading tableauInt into a register
CMP #null, R1
BEQ deref_null_error    ; Checking if the class identifier is null
```

3-La troisième étape est vérifier que les dimensions du tableau sont respectées i.e. 'vérifier indices données par l'utilisateur sont inférieur à la taille du tableau). Cette étape constitue une vérification qui est ne sera utile la majorité du temps car si on essaye d'accéder à un espace mémoire non réservé la machine virtuelle lancera un message d'erreur, mais dans le cas ou cette espace là est réservé par un autre tableau ou une instance de classes cela pourra avoir un effet désastreux sur le program.

Dans le cas où l'utilisateur donne des indices non valide, une erreur est déclenchée avec le nom: table_dimension_are_not_respected. En assembleur on vérifie si la dimension est inférieur au égal au input car les indices commencent par 0 et donc l'indice limite pour les lignes est le nombre de ligne moins 1.

Voici un exemple du code assembleur pour un tableau 2d dans cette étape:

```
LOAD 0(R1), R0  ; loading size 1d of tableauFloat into a register
CMP R4, R0 ; comparing line input with dimension present in the table
BLE table_dimension_are_not_respected

LOAD 1(R1), R0  ; loading size 2d of tableauFloat into a register
CMP R5, R0 ;  comparing column input with dimension present in the table
BLE table_dimension_are_not_respected
```

4-La quatrième et dernière étape est le calcul de la position de l'élément qu'on veut affecter dans la mémoire. Pour faire cela il fallait introduire dans le paquetage pseudocode une nouvelle classe RegisterIndirectOffset qui nous permet d'accéder à une adresse en

utilisant le registre qui contient l'adresse, un registre qui contient une valeur auquel on utilisera comme décalage par rapport au l'adresse et on peut aussi ajouter un autre offset qui est un Immediate integer. Cette classe implémente la fonctionnalité : registre indirect avec déplacement et index décrit dans le poly à la page 106.

Pour les tableaux 1d, cette opération d'accès au élément est trivial comme on a dans un registre l'indice de l'élément qu'on veut accéder et il faut déplacer par de cette valeur car on stocke le nombre d'éléments dans 0(adresse de la table 1d). Voici un exemple du code assembleur de cette opération:

```
LOAD 1(R1, R3), R0
```

Pour les matrices l'accès n'est pas aussi simple que les tableaux 1d, la formule est simple : pour accéder au élement avec les indice suivant [i][j], on multiplie i par le nombre de colonnes et on ajoute j à cela. Ainsi dans l'assembleur, on récupère le nombre de colonnes de la mémoire, on le multiplie par le registre qui contient l'indice de la ligne, ensuite on ajoute l'indice de la colonne à cette valeur. Enfin, pour accéder à l' élément on se déplace par 2 par rapport à l'adresse car on stocke les dimensions dans les deux premières cases et on se déplace aussi par rapport à la valeur calculée précédemment.

Voici le code assembleur utilisée pour faire cette opération d'accès:

```
LOAD1(R1), R0; We put the number of columns in the register R0
MUL R0, R4; We do the product of the line given by the user by R0
; which contains the number of columns fo the matrix

ADD R5, R4; We add the column given by the user
LOAD 2(R1,R4),R0 ;  We move relative to the address of the table
```

2.3.4 Accès aux caractéristiques du tableaux

La dernière opération qu'on a développée dans les tableaux est l'accès au caratéristique du tableau pour un tableau 1d on peut accéder uniquement au nombre d'éléments dans le tableau et pour les matrices on peut accéder uniquement le nombre de lignes et le nombre de colonnes.

Syntaxiquement, on n'a pas fait de changements car on exploite la sélection qui a été déjà développée pour accéder au champ d'une classe.

Contextuellement, on a modifié la méthode verify de la sélection pour l'ajuster au tableau. On vérifie initialement que l'expression de la selection est un tableau et ensuite on vérifie que l'identificateur du champ concerne les deux noms : Size1D et Size2D sinon erreur est lancée avec le nom "Pour accéder au caractéristique du tableau, il utilise les identificateurs suivants mettre size1D, size2D".

L'autre vérification vérifie que quand l'utilisateur met la valeur size2D, le tableau doit être de dimension 2, sinon l'erreur est déclenché avec le message suivant : "Le tableau est 1D"

Dans la génération de code, on passe par deux étapes:

1-La première est la récupération de l'adresse du tableau dans un registre et la vérification que l'adresse du tableau est non nulle similairement à ce qu'a été fait lors de la récupération des éléments du tableau.

2-La deuxième étape est le déplacement dans le tableau pour accéder au champ désiré.

Exemple, on utilisera la troisième ligne du code suivant pour montrer l'assembleur de cette partie

```
{
    int[] tableauInt = new int[5];
    float[][] tableauFloat = new float[10][10];
    println(tableauInt.size1D + tableauFloat.size2D);
}
```

Récupération de size1D de tableauInt Int

```
LOAD 1(GB), R2; we put the address of tableauInt in a register
CMP #null, R2; Check that the address is not null
BEQ deref_null_error; throw an error if the address is null
LOAD 0(R2), R2; We put the number of elements of the array into a register
```

Récupération de size2D de tableauFloat

```
LOAD 2(GB), R3; we put the address of tableauFloat in a register
CMP #null, R3; Check that the address is not null
BEQ deref_null_error; throw an error if the address is zero
LOAD 1(R3), R3; We put the number of columns in a register
```

2.3.5 Exploitation du polymorphism dans les tableaux

La fonctionnalité la plus avancée qu'on a pu intégrer dans les tableaux était les polymorphismes des classes, même si on n'a pas ajouté de code pour avoir cette fonctionnalité mais comme on fait le maximum pour exploiter les fonctionnalités existantes cela était le résultat du code qu'on a déjà fait.

Comme le compilateur permet de définir un tableau avec un type de classe, et si on veut créer un tableau qui contient plusieurs classes qui sont différentes mais qui dérive de la même classe comme la classe Object qui est le parent de toutes les classes. On peut dans ce cas crée un tableau de la classe parent, et mettre des classes qui dérive de la classe parent et exploitée ce qui est stockée dans les classes

L'exemple suivant illustre cela, on définit une classe A et une classe B extends de A. Dans le main on définit un tableau dont ces éléments sont du même type que la classe A ainsi on peut mettre des instance de la classe de B dans ce tableau. Dans ce cas, on a redéfinit la méthode somme dans la class B et qu'on pourra accéder à partir de la classe A. La sortie de ce programme est égale à 3 et 6, ainsi on peut remarquer que la méthode appelée par le deuxième élément du tableau est celle qui concerne la classe B.

```
class A{
    int x =1; int y=2;
    int somme(){
         return x+y;
    }
}

class B extends A{
    int z=3;
    int somme(){
         return x+y+z;
    }
}


{
    A instanceA = new A();
    B instanceB = new B();
    int iter = 0;
    A[] tableau = new A[2];
    tableau[0] = instanceA;
    tableau[1] = instanceB;
    while(iter < tableau.size1D){
         println("somme ", iter, " = ", tableau[iter].somme());
         iter = iter +1;
    }
}
Output:
somme 0 = 3
somme 1 = 6
```

2.4 Méthode de validation

Dans l'étape de validation des tableaux, on fait un travail similaire à ce qui a été réalisé pour le langage avec objet et sans objet. On crée un nouveau dossier avec le nom extension placée dans le path suivant : src/test/deca/extension. Dans ce dossier on a créé trois dossiers : syntax, context, codegen car pour cette extension on fait des modifications dans ces trois parties. Le test de codegen était les plus important car dans la partie syntaxique et contextuelle on a beaucoup exploité le code réalisée dans le langage avec objet.

Le raisonnement derrière les tests de syntax et context était de passer par toutes les règles qu'il a créées et toutes les fonctions développées. On a testé localement initialement sans utiliser le dossier de test, et quand on avait un produit qui fonctionne et qui lance toutes les erreurs dont on veut on a réécrit les tests dans les dossiers de test et on a ajouté les test automatiques afin d'être sûre que ce test reste valide/invalid quand on fait des changements du code.

Les tests de gencode étaient beaucoup plus importants que les autres tests dans cette partie, car on les a utilisés pour bien tester le code et s'assurer que l'output est généré et correspond à la valeur désirée.

Les tests automatiques de la génération de code prend la sortie quand on exécuté le programme généré et le place dans un fichier dans le dossier résulat, avec le nom du fichier du test concaténé avec le string "_output" et il le compare avec un autre fichier dans le dossier result créé manuellement avec le nom : nom_de_fichier + "_output_expected" auquel on a met le résultat que le fichier deca doit sortir. Par exemple pour le test : test_decl_tableau_class_2d.deca il existe un fichier avec le nom: test_decl_tableau_class_2d_output.res qui contient la sortie lors de l'exécution du program et test_decl_tableau_class_2d_output_expected.res qui contient le résultat qu'on doit avoir quand on exécute le program.

2.5 Validation

Les tests dans le dossier codegen était trés important lors de la partie de debug de l'extension, car quand on trouvé des problémes dans la récupération des éléments on a dû changer certains fonction dans la génération du code et d'une manière accidentel on a affecté une autre partie du code et en utilisant les tests automatiques on a pu détecter ces erreurs et les corrigées rapidement.

3 Conception de la bibliotheque du calcul matriciel

3.1 Introduction

L'objectif de la bibliothèque Math est de pouvoir effectuer les différents calculs usuels sur les matrices définies dans le cadre de l'extension. Cette partie se base sur la conception des tableaux qui a été détaillé dans la partie précédente. A partir de cette conception des tableaux dans le langage deca, il est possible d'effectuer différents calculs en respectant les règles de calculs sur les matrices provenant des mathématiques. Les différentes opérations disponible dans cette bibliothèque sont: la somme, la soustraction, le produit membre par membre, la division membre par membre, le calcul de la trace, du déterminant, de l'inverse d'une matrice, l'aplanissement d'une matrice, la triangulation d'une matrice. Il existe un fichier *.decah associé à presque chacune de ces opérations qu'il faudra appeler par une inclusion pour rendre les méthodes présentes dans ces fichiers utilisables dans un code *.deca . Les fichiers *.decah se trouve dans le répertoire: src/main/resources/include. Pour utiliser ces méthodes dans un fichier *.deca, il est nécessaire de mettre" #include "<nom_du_fichier>.decah" en début de code (avant le main).

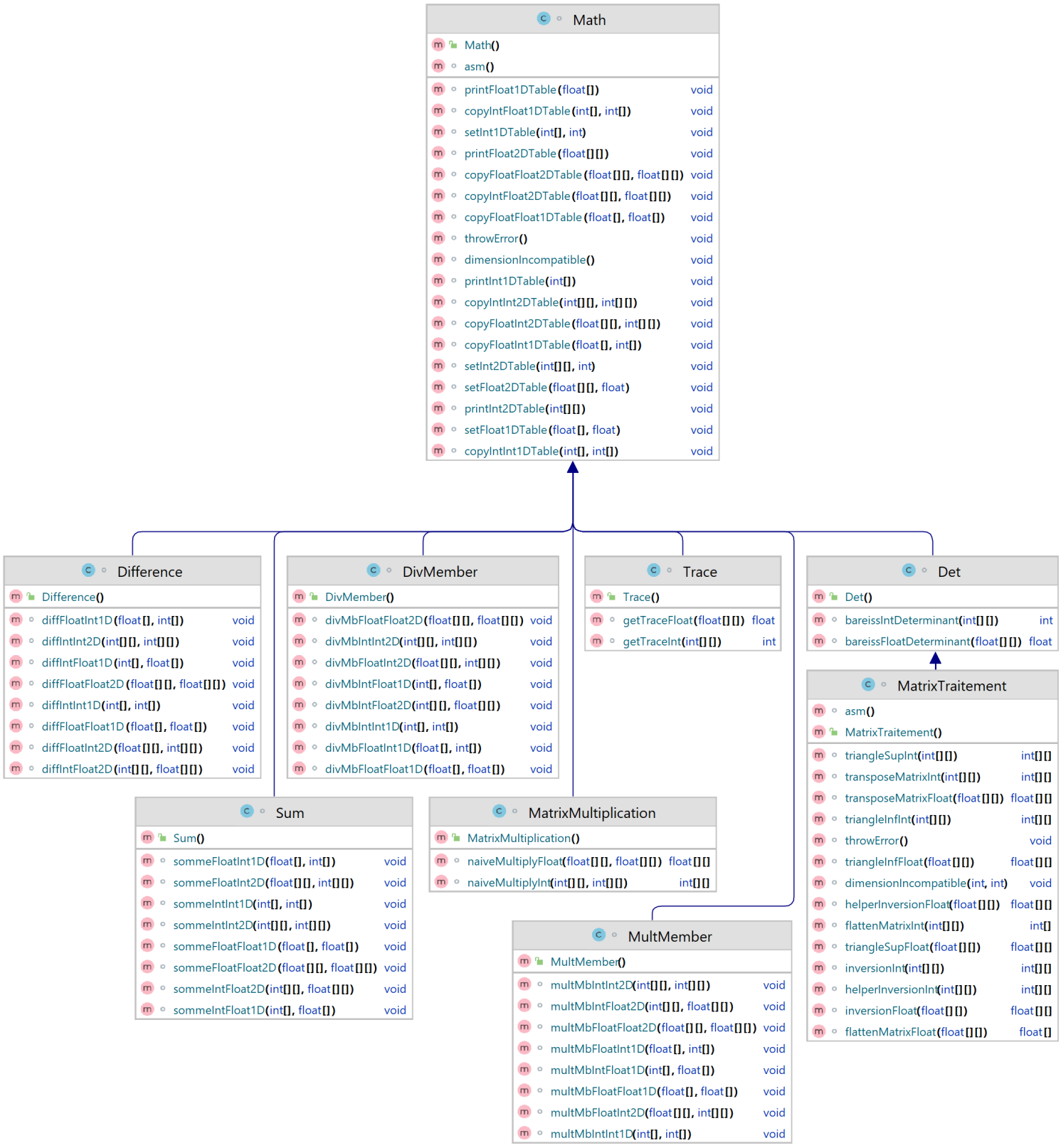

Figure 2: Diagramme représentant l'architecture des fichiers *.decah et les différentes méthodes présentes dans ces fichiers. Les flèches représentent les liens entre ces fichiers (inclusion et héritage)

Explication des méthodes présentes dans les fichiers *.decah . “Math.decah”

Les méthodes générales à l’utilisation des tableaux sont présentes dans ce fichier. Les méthodes présentes sont:

void setInt1DTable(int[] table, int value) Prend en paramètre un tableau d’entiers à une dimension et remplace chacun de ces coefficients par la valeur entière value.

void setFloat1DTable(float[] table, float value) Prend en paramètre un tableau de flottants à une dimension et remplace chacun de ces coefficients par la valeur flottante value.

void setInt2DTable(int[][] table, int value) Prend en paramètre un tableau d’entiers à deux dimensions et remplace chacun de ces coefficients par la valeur entière value.

void setFloat2DTable(float[][] table, float value) Prend en paramètre un tableau de flottants à deux dimensions et remplace chacun de ces coefficients par la valeur flottante value.

void printInt1DTable(int[] table) Affiche le contenu un tableau d’entiers à une dimension void printFloat1DTable(float[] table) Affiche le contenu un tableau de flottants à une dimension

void printInt2DTable(int[][] table) Affiche le contenu un tableau d’entiers à deux dimensions

void printFloat2DTable(float[][] table) Affiche le contenu un tableau de flottants à deux dimensions

void copyIntInt1DTable(int[] dest, int[] src)

Copie le contenu d’un tableau d’entiers à une dimension source src dans un second tableau d’entiers dest. La méthode dimensionIncompatible() est appelée si une des dimensions du tableau source est plus grande que celles du tableau destination.

void copyFloatFloat1DTable(float[] dest, float[] src)

Copie le contenu d’un tableau de flottants à une dimension src dans un second tableau de flottants dest. La méthode dimensionIncompatible() est appelée si une des dimensions du tableau source est plus grande que celles du tableau destination.

void copyIntFloat1DTable(int[] dest, float[] src)

Copie le contenu d’un tableau de flottants à deux dimensions src dans un second tableau d’entiers dest. Convertit tous les membres de src en entiers. La méthode dimensionIncompatible() est appelée si une des dimensions du tableau source est plus grande que celles du tableau destination.

void copyFloatInt1DTable(float[] dest, int[] src)

Copie le contenu d'un tableau d'entiers à une dimensions src dans un second tableau de flottants dest. Convertit tous les membres de src en flottants. La méthode dimensionIncompatible() est appelée si une des dimensions du tableau source est plus grande que celles du tableau destination.

Les mêmes méthodes de copie existent pour les tableaux à deux dimensions. Leurs noms sont modifiés (1D remplacé par 2D) et le type des paramètres pour correspondre à des matrices 2D.

void dimensionIncompatible() Affiche le message "Erreur de dimension" et appelle la méthode throwError().

void throwError() Effectue l'instruction assembleur: asm ("ERROR ");

Les autres fichiers *.decah contiennent tous une inclusion de Math.decah en début de fichier.

"Sum.decah"

Les méthodes de calcul de somme de tableaux sont présentes dans ce fichier. Dans toutes les méthodes suivantes, si la taille des deux matrices opérandes ne sont pas identiques, le message d'erreur "Les deux matrices doivent être de la même taille pour la somme de matrices" est envoyé.

void sommeIntInt1D(int[] table1, int[] table2)

Additionne chaque élément du tableau d'entiers à une dimension table1 à l'élément de mêmes coordonnées du tableau d'entiers à une dimension table2. Le résultat est affecté au coefficient de table1.

void sommeFloatInt1D(float[] table1, int[] table2)

Additionne chaque élément du tableau de flottants à une dimension table1 à l'élément de mêmes coordonnées du tableau d'entiers à une dimension table2. Lors des soustractions, les éléments de table2 sont convertis en flottants. Le résultat est affecté au coefficient de table1.

void sommeIntFloat1D(int[] table1, float[] table2)

Additionne chaque élément du tableau d'entiers à une dimension table1 à l'élément de mêmes coordonnées du tableau de flottants à une dimension table2. Lors des soustractions, les éléments de table2 sont convertis en entiers. Le résultat est affecté au coefficient de table1.

void sommeFloatFloat1D(float[] table1, float[] table2)

Additionne chaque élément du tableau de flottants à une dimension table1 à l'élément de mêmes coordonnées du tableau de flottants à une dimension table2. Le résultat est affecté au coefficient de table1.

Les mêmes méthodes de de calcul de somme existent pour les tableaux à deux dimensions. Leurs noms sont modifiés (1D remplacé par 2D) et le type des paramètres pour correspondre à des matrices 2D.

De manière similaire que la somme, il existe des méthodes permettant d'effectuer la soustraction, la multiplication et la division membre par membre respectivement dans les fichiers "Difference.decah", "Multiply_member_by_member.decah" et "Divider_member_by_member.decal
Les méthodes correspondent à celles présentes dans "Sum.decah" adaptés pour ces opérations.

Il n'est pas possible d'effectuer le produit par une constante k directement. Cependant, effectuer un produit membre par membre avec une matrice dont les coefficients valent tous k revient à faire un produit par une constante.

"Produit_Matriciel_Naif.decah"

Les méthodes permettant d'effectuer des produits matriciels sont présentes dans ce fichier. Dans les méthodes suivantes, effectuer un produit matriciel entre matrices de taille nxp et qxk où p != q envoie le message d'erreur "Taille de matrice incompatible avec le produit matriciel:p != q".

float[][] naiveMultiplyFloat(float[][] a, float[][] b)

Créé une nouvelle matrice de flottants de taille nxk si les tailles des matrices opérandes sont nxp et pxk. Effectue le produit matriciel entre les deux matrices opérandes de flottants et affecte le résultat dans la matrice créée.

int[][] naiveMultiplyInt(int[][] a, int[][] b)

Créé une nouvelle matrice d'entiers de taille nxk si les tailles des matrices opérandes sont nxp et pxk. Effectue le produit matriciel entre les deux matrices opérandes d'entiers et affecte le résultat dans la matrice créée.

"Trace.decah"

Les méthodes permettant d'effectuer le calcul de trace sont présentes dans ce fichier. Le calcul de trace s'effectue uniquement sur des matrices carrées. Par conséquent, le message d'erreur "Impossible de calculer la trace d'une matrice non carrée" est envoyé si la matrice paramètre n'est pas carrée.

int getTraceInt(int[][] table)

Calcule et renvoie la trace du tableau d'entiers table.

float getTraceFloat(float[][] table)

Calcule et renvoie la trace du tableau de flottants table.

"Det.decah"

Les méthodes permettant d'effectuer le calcul de déterminant sont présentes dans ce fichier. Le calcul de déterminant s'effectue uniquement sur des matrices carrées. Pour une matrice de taille nxp avec n différent de p, le message d'erreur"La matrice doit être carrée pour le calcul du déterminant: n! =p " est envoyé.

float bareissFloatDeterminant(float[][] a)

Calcule le déterminant de la matrice de flottants a.

int bareissIntDeterminant(int[][] a)

Calcule le déterminant de la matrice d'entiers a.

"extension_matricielle.decah"

Les méthodes permettant d'inverser, d'aplanir, de trianguler des matrices et la transposée des matrices.

int[][] triangleSupInt(int[][] mat)

Transforme la matrice carrée d'entiers mat en une matrice triangulaire supérieure. Si la matrice en paramètre a pour taille nxp avec n!=p, le message d'erreur suivant est envoyé: "dimension incompatible n!=p".

int[][] triangleInfInt(int[][] mat)

Transforme la matrice carrée d'entiers mat en une matrice triangulaire inférieure. Si la matrice en paramètre a pour taille nxp avec n!=p, le message d'erreur suivant est envoyé: "dimension incompatible n!=p".

float[][] triangleSupFloat(float[][] mat)

Transforme la matrice carrée de flottants mat en une matrice triangulaire supérieure. Si la matrice en paramètre a pour taille nxp avec n!=p, le message d'erreur suivant est envoyé: "dimension incompatible n!=p".

float[][] triangleInfFloat(float[][] mat)

Transforme la matrice carrée de flottants mat en une matrice triangulaire inférieure. Si la matrice en paramètre a pour taille nxp avec n!=p, le message d'erreur suivant est envoyé: "dimension incompatible n!=p".

int[] flattenMatrixInt(int[][] mat)

Transforme une matrice d'entiers à deux dimensions en une matrice d'entiers à une dimension. Pour cela une nouvelle matrice à une dimension est créé (de taille n.p si mat est de taille nxp) et tous les éléments de mat sont copiés dans cette matrice de l'élément de la première ligne, première colonne à l'élément de la dernière ligne, dernière colonne.

float[] flattenMatrixFloat(float[][] mat)

Transforme une matrice de flottants à deux dimensions en une matrice de flottants à une dimension. Pour cela une nouvelle matrice à une dimension est créé (de taille n2 si mat est de taille nxn) et tous les éléments de mat sont copiés dans cette matrice de l'élément de la première ligne, première colonne à l'élément de la dernière ligne, dernière colonne.

int[][] inversionInt(int[][] mat)

Calcule et renvoie l'inverse de la matrice d'entiers mat. Si la matrice n'est pas carrée (de taille nxp avec n != p) ou si le déterminant est nul les message d'erreur suivant est envoyé: "dimension incompatible n!=p".

float[][] inversionFloat(float[][] mat)

Calcule et renvoie l'inverse de la matrice de flottants mat. Si la matrice n'est pas carrée (de taille nxp avec n != p) ou si le déterminant est nul les message d'erreur suivant est envoyé: "dimension incompatible n!=p".

3.2 Algorithme developé

L'algorithme de Strassen a été mentionné et développé dans la partie I.Analyse bibliographique. Celui-ci permet de calculer un produit matriciel plus efficacement que la méthode traditionnelle. Cependant, cet algorithme requiert beaucoup de mémoire pour fonctionner car, lors des étapes intermédiaires, de nouvelles matrices sont à définir pour chacun des sept produits à effectuer. L'algorithme de Strassen implique un calcul récursif sur les blocs des matrices et est donc très demandeur en mémoire que l'on ne peut pas libérer. De plus, l'algorithme se limite uniquement aux matrices carrées dont la taille est une puissance de deux. On a essayé d'implémenter cet algorithme en l'adaptant pour qu'il puisse effectuer des produits matriciels avec de matrices de taille quelconque. Pour cela il faut agrandir la matrice jusqu'à avoir une matrice dont la taille est une puissance de deux en complétant avec des lignes et des colonnes de zéros. On a rapidement remarqué que l'implémentation ne serait pas beaucoup plus efficace que l'algorithme "naïf" pour des matrices de petite taille et demanderait beaucoup plus de mémoire. En effet, de nombreuses opérations annexes étaient à effectuer: construire les blocs des matrices, copier les blocs de la matrice résultat dans celle-ci etc. Le problème principal est le fait qu'on ne puisse pas libérer la mémoire allouée. Pour calculer le déterminant d'une matrice, on a utilisé la méthode d'élimination de d'élimination de Gauss-Jordan permettant la résolution de systèmes d'équations linéaires et donc de trigonaliser des matrices. Cela est particulièrement intéressant ici car le calcul du déterminant d'une matrice triangulaire est le produit des éléments de sa diagonale. Le pseudo code correspondant à la programmation implantée dans le cadre de l'extension est le suivant:

```
function detGauss(A)
    n = A.length
    det = 1
    for i = 1 to n
```

```
        for j = i + 1 to n
            a = A[j][i] / A[i][i] // possibilite de division par zero ici
            for k = i to n
                A[j][k] = A[j][k] - a * A[i][k]
            end for
        end for
        det = det * A[i][i]
    end for
    return det
```

Un problème de cet algorithme est la possibilité d'obtenir une division par zéro à la ligne 6 de ce pseudo-code. Dans le cadre de notre implémentation, on a remarqué que cet algorithme que l'on a implanté ne fonctionne pas pour une matrice avec au moins un zéro sur sa diagonale ce qui représente une limitation de l'extension.

Concernant l'inversion de matrice dans l'extension de Deca, on a utilisé deux méthodes pour inverser une matrice de type int ou float, les signatures sont int[][] inversionInt(ou float) (int[][] mat) et int[][] helperInversionInt(ou float) (int[][] mat) respectivement, la première sert à distinguer si la matrice d'entrée est inversible (déterminant nul ou la matrice non carrée), la seconde méthode est celle qui sert à inverser une matrice éligible et retourner le résultat. L'algorithme qu'on a utilisé dans cette méthode est celui de Gauss Jordan qui consiste tout d'abord créer une nouvelle matrice qui contient deux fois plus de colonnes que la matrice originale (size x 2*size), où la partie gauche est exactement la même matrice que la matrice d'entrée, la partie droite est une matrice identitée, en regardant le code , c'est le rôle de première double boucle while. Ensuite, le but du jeu est de transférer la partie de gauche (de taille size*size) de la nouvelle matrice à la matrice identitée en effectuant des opérations identiques matricielles (combinaisons linéaires de lignes et de colonnes), où la partie droite (de taille size*size) de cette matrice est le résultat, en regardant le code, c'est le rôle de la deuxième et la troisième double boucle while qui joue. Finalement, on ajoute encore une double boucle while pour récupérer la matrice de résultat de la partie droite et la retourner. Dans la méthode "inversionInt", on va simplement appeler la méthode qu'on a présenté ci-dessus si la matrice est éligible pour une inversion et retourner le résultat.

3.3 Validation de la bibliotheque

Pour valider le contenu proposé dans la bibliothèque, des tests ont été écrits après chaque implémentation d'opération matricielle pour s'assurer que celles-ci fonctionnent correctement. Les tests réalisés sont présents dans le répertoire src/test/deca/library. Ce répertoire de tests a été organisé de la même manière que les autres répertoires de tests déjà présents. On trouve donc dans ce répertoire deux dossiers: le dossier "valid" qui contient tous les tests permettant de valider le bon fonctionnement des méthodes implémentées dans la bibliothèque le dossier "invalid" qui comprend des tests provoquant les erreurs que peuvent envoyer les méthodes de la bibliothèque

Les tests présents permettent de couvrir toutes les méthodes et leurs erreurs présentes

dans la bibliothèque. C'est lors des tests que l'on s'est rendu compte que le calcul de déterminant ne fonctionnait pas pour toutes les matrices. Par exemple: Soit A une matrice de taille 4x4:

```
A=[1, 0, 2,-1
   3, 0, 0, 5
   2, 1, 4,-3
   1, 0, 5, 0]
```

Le déterminant de cette matrice vaut 30 alors que notre implémentation renvoie 0. Cela se produit certainement car une division par un nombre très proche de 0 ou nul s'effectue dans certains cas et donc on a décidé de renvoyer 0 comme déterminant pour ne pas que le compilateur n'affiche d'erreur "overflow_error".

Pour valider nos tests, on a utilisé un script qui rend automatique la procédure de testing à partir des tests présentés ci-dessus présents dans les dossiers valid. Ce script compare le résultat donné par ces tests avec le résultat attendu calculé au préalable à partir de calculateurs en ligne. Les résultats de test et les résultats attendus se trouvent dans les dossiers src/test/deca/library/invalid/result et src/test/deca/library/valid/result. Les résultats attendus et les résultats de test sont les mêmes pour tous les tests effectués.

4 Conclusion

La chose la plus difficile dans cette extension était la conception de l'architecture qui a été changée plusieurs fois afin d'avoir quelque chose qui est simple et très proche de de l'architecture réalisée pour le langage avec objet. La bibliothèque n'était pas très difficile à créer mais il fallait créer plusieurs méthodes et cela prenait un temps significatif.