École nationale de l'informatique et des mathématiques appliquées de Grenoble

Projet Génie Logiciel

Manuel Extension

Année universitaire 2020-2021

Groupe 8 Equipe 40

Étudiants 2A

Table des matières

1 Spécification de l'extension

Pour l'extension TAB, nous avons choisi d'ajouter à notre langage des tableaux de taille statique et dont les éléments sont tous de même type. Nous avons également choisi d'autoriser la déclaration de tableaux d'objets, respectant la hiérarchie des classes, par exemple ce code sera accepté si Serpent et Elephant sont des classes filles de Animal :

```
{
    Animal[] zoo = { new Serpent(), new Elephant() };
}
```

Concernant la syntaxe des tableaux nous nous sommes inspirés de celle des tableaux en Java. Ainsi les nouveaux changements à prendre en compte ont été les suivants :

- Le type tableau, apparaissant lors de la déclaration de variable, lors d'une méthode renvoyant un tableau ou encore lorsqu'une méthode prend en argument un tableau : int[], float[][], Animal[]...
- La sélection d'un élément d'un tableau : tab[index]
- Initialisation d'un tableau avec new : int[] tab = new int[] ;
- Tableau défini explicitement : {1, 2, 3, 4}

```
{
    int[][][] x = new int[4][3][]; // Non accepte
    int[][] y = new int[2][2];
    x[2][1] = new int[1]; // Non accepte
    y[0] - { 1, 2 }; // Non accepte
}
```

Cependant à la différence de la syntaxe de Java, il n'est pas possible de redéfinir des sous-tableaux. Ainsi ce code est invalide :

```
{
    int[][][] x = new int[4][3][]; // Non accepte
    int[][] y = new int[2][2];
    x[2][1] = new int[1]; // Non accepte
    y[0] - { 1, 2 }; // Non accepte
}
```

Bien que la conversion d'entiers en flottants soit possible pour une variable n'étant pas de type tableau, la conversion d'entiers en flottants au sein même d'un tableau de flottants n'est pas possible. Ainsi ce code est invalide :

```
{
    float[][] x = {{1, 2.0}, {2, 1.0}}; // Non accepte
}
```

Ces choix de conception nous ont permis de réaliser une bibliothèque de calcul matriciel complète qui sera présentée en détail à la partie 3.5 de ce document.

2 Analyse

2.1 Analyse des besoins pour l'extension TAB

La syntaxe choisie par notre extension pour la déclaration de tableau, la création et l'affectation est inspirée de celle de Java. La grammaire a été modifiée de base pour reconnaître tout ce que Java reconnaissait (Sous tableau, déclaration partielle, ...) mais les contraintes de temps sur l'implémentation effective de génération de code assembleur nous a fait choisir des limitations comme stipulées dans le manuel utilisateur.
Le choix concernant la création en mémoire du tableau, lui, s'inspire de ce que *gcc* fait pour la création d'un tableau statique sur la pile. [1] Ainsi, nous créons un seul tableau de la bonne dimension (multiplication des tailles pour chaque dimension) sur le tas (et non la pile comme *gcc*) et nous retrouvons une valeur en calculant son indice. Le choix de créer un tableau des tailles est motivé par le fait qu'en C, les tailles pour chaque dimension ne peuvent être des expressions calculées à l'exécution tandis que cela est possible pour notre extension Deca, il faut ainsi garder les tailles afin de retrouver l'indice dans la mémoire. Les spécifications sont données dans la partie 3.4.

1. http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch12s03.html

3 Choix de conception, architecture et algorithmes

3.1 Lexicographie

Pour prendre en compte la syntaxe des tableaux énoncées ci-dessus, nous avons donc eu besoin uniquement d'ajouter les crochets puisque les accolades étaient déjà finies pour la méthode main par exemple.

Dans notre lexer, les crochets étaient alors définis de la manière suivante :

- LHOOK = '['
- RHOOK = ']'

3.2 Grammaire

3.2.1 Grammaire concrète

primary_expr ⟶
'{' **list_expr** '}'
| **new_object**
| **ident list_hook_expr**

new_object ⟶
'new' **ident** '(' ')'
| 'new' **ident list_hook_expr**

unary_expr ⟶
'(' **select_expr** ')' ('[' **expr**']')+
| **ident** '[' **expr** ']' ('[' **expr** ']')*

type ⟶
ident ('[' ']')*

list_hook_expr ⟶
('[' **expr** ']')*

3.2.2 Grammaire abstraite

EXPR ⟶
| **NEW_OBJECT**
ArrayLiteral[**LIST_EXPR**]

NEW_OBJECT ⟶
New[**IDENTIFIER**]
| **NewArray**[**IDENTIFIER LIST_HOOK_EXPR**]

LIST_HOOK_EXPR ↑*r* ⟶
[(**EXPR**↑*e*)*]

LVALUE ⟶
| **ArraySelecton**[**IDENTIFIER LIST_HOOK_EXPR**]

IDENTIFIER ⟶
| **IdentifierArray** [**IDENTIFIER INT_LITERAL**]

INT_LITERAL ⟶
| **IntLiteral**↑*int*

3.2.3 Grammaire attribuée de décompilation

EXPR ↑*r* ⟶
| **NEW_OBJECT** ↑*r*
ArrayLiteral[**LIST_EXPR** ↑*elements*] { r := '{ '.elements. '}' }

NEW_OBJECT ↑*r* ⟶
New[**IDENTIFIER**↑*class*] { r := 'new '.class. }
|**NewArray**[**IDENTIFIER**↑*type*
LIST_HOOK_EXPR↑*allocation*]
{ r := 'new '.type.allocation}

LIST_HOOK_EXPR ↑*r* ⟶
{ r := ϵ } [(**EXPR**↑*e* { r := r.'['.e.']' })*]

LVALUE↑*r* ⟶
| **ArraySelecton**[**IDENTIFIER**↑*name*
LIST_HOOK_EXPR↑*selection*]
{ r := name.selection}

IDENTIFIER↑*r* ⟶
| **IdentifierArray** [**IDENTIFIER**↑*type* **INT_LITERAL**↑*int*]{ r := type.('[' ']')+ }

INT_LITERAL↑*r* ⟶
| **IntLiteral**↑*int*

3.2.4 Grammaire contextuelle

Changement dans la grammaire contextuelle :
type ↓*env_types* ↑*type*

$\longrightarrow$ Identifier $\uparrow name$
condition $(\_, type) \stackrel{\triangle}{=} env\_types(name)$
$\longrightarrow$ IdentifierArray $\uparrow type$ [
type $\downarrow env\_types \uparrow type\_base$
Entier $\uparrow dim$
]
affectation *type* := ArrayType(type_base, dim)
condition $dim > 0$

On rajoute également une règle de dérivation pour **expr** :
expr$\downarrow env_types \downarrow env_exp \downarrow class \uparrow$type
$\longrightarrow$ NewArray [
type $\downarrow env\_types \uparrow type\_base$
list__expr__array$\downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type\_index \uparrow dim$
]
condition $type \neq \underline{void}$ et $dim > 0$ et $type\_index = int$
affectation type := ArrayType(type_base, dim)

list__expr__array$\downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type \uparrow dim$
$\longrightarrow$ $\{dim := 0\}$
[(**expr**$\downarrow env_types \downarrow env_exp \downarrow class \uparrow$type
$\{dim = dim + 1\}$
condition *type* unique
)*]

lvalue$\downarrow env_types \downarrow env_exp \downarrow class \uparrow$type
$\longrightarrow$ ArraySelection [
expr$\downarrow env_types \downarrow env_exp \downarrow class \uparrow$ArrayType(type_base, dim_ident)
list__expr__array$\downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type\_index \uparrow dim$
]
condition $dim = dim\_ident$

literal $\uparrow$ArrayType(type_index, dim)
$\longrightarrow$ ArrayLiteral [**list__expr__array**$\downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type\_index \uparrow dim$]

3.3 Intégration des nouvelles classes dans la hiérarchie existante

Au vu des terminaux présents dans notre nouvelle grammaire, quatre classes ont été crées s'intégrant dans la hiérarchie proposée par les professeurs. Ainsi on a :

- *ArrayLiteral* dérivant de *AbstractExpr* fournissant l'abstraction d'un littéral de tableau avec la liste des expressions qui le forme comme attribut.
- *NewArray* dérivant de *AbstractExpr* décrivant l'allocation d'un nouveau tableau, il est formé du type de base et de la liste d'expressions désignant les différentes valeurs de taille pour chaque dimension du tableau
- *ArraySelection* dérivant de *AbstractLValue* décrivant la sélection d'un élément du tableau. Il est formé de l'expression décrivant le tableau et d'une liste d'expressions donnant les indices d'accès pour chaque dimension.
- *IdentifierArray* dérivant de *AbstractIdentifier* décrivant un identifiant de type de tableau avec son type de base et son nombre de dimension.

Ces différentes nouvelles classes s'intègrent dans l'architecture précédentes avec les fonctions de vérification de la partie B et les fonctions de génération de code de la partie C. Toutes n'ont pas été utilisées par exemple il n'y a aucune génération de code pour *IdentifierArray*.
La seule difficulté d'intégration de la vérification contextuelle fut de vérifier si un *ArrayLiteral* avait bien tous ses éléments du même type (qu'un élément soit un autre *ArrayLiteral* ou un autre littéral. De même, il fallait que le nombre d'éléments initialisé dans chaque dimension soit cohérent. Par exemple, ceci ne l'est pas :

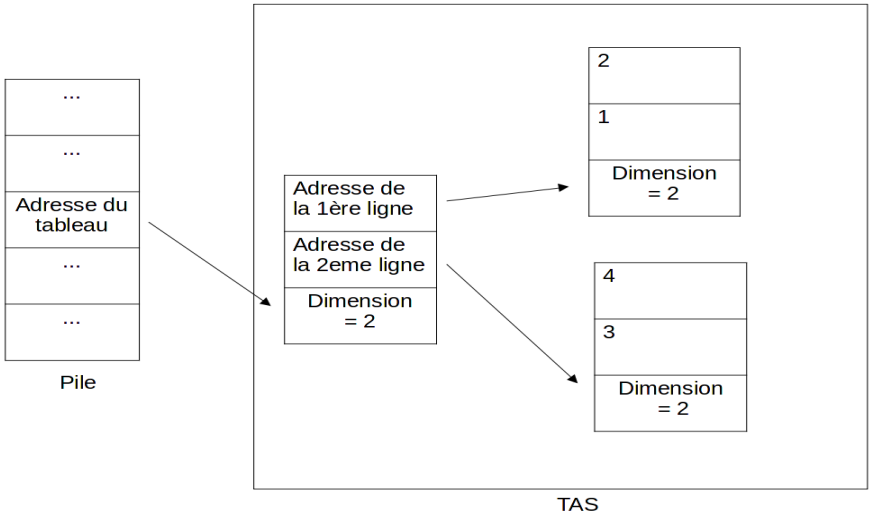
```
1 int[][] t = {{1,2}, {1}};
```

car la deuxième dimension semble être de taille 2 mais le deuxième élément montre un tableau de taille 1.
Il a alors fallu parcourir l'arbre des *ArrayLiteral* en profondeur pour avoir les tailles de chaque dimension et pouvoir ainsi vérifier la cohérence lors de la suite du parcours. Les tailles pour chaque dimension sont alors sauvegardées dans un attribut *dimension* de la classe.

3.4 Génération de code assembleur

Pour l'étape C (génération du code assembleur), il a fallu dans un premier temps effectuer des choix de conception importants, que nous allons donc décrire ici.

Tout d'abord pour le stockage en mémoire nous avions décidé de procéder comme pour les objets à savoir une adresse de tas sur la pile, et une allocation de bloc sur le tas grâce à la commande assembleur *NEW*. De là nous avons déjà du faire un choix. Initialement, nous voulions qu'il y est sur le tas autant de blocs que le tableau ait de dimension. Voici un exemple pour la matrice $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$:

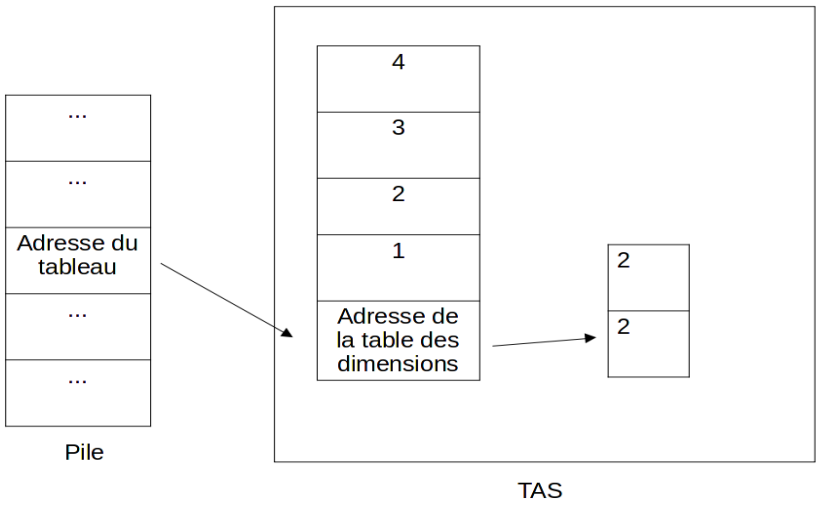


Le choix ci-dessus apportait plusieurs avantages, notamment la capacité à sélectionner des sous-tableaux (par-exemple dans le cas d'une matrice sélectionner une ligne). Néanmoins, ce modèle s'est révélé très difficile à implémenter, et ce en raison d'un choix fait aux étapes précédentes. Le problème est que nous autorisons la déclaration des dimensions avec des expressions autres que IntLiteral. Par exemple, il est possible d'écrire cela :

```
{
    float[] tab;
    int expr = 3;
    tab = new float[expr+expr];
    tab = new float[6];
    tab = new float[2*expr];
    // ...etc
}
```

Ceci implique que nous n'avons pas accès aux dimensions dans le programme java du compilateur, ces dernières étant évaluées à l'exécution. Ainsi, cette solution était possible, mais complexe. Dans l'optique de produire, dans les temps, une solution fiable, nous avons envisagé une seconde conception, moins optimale pour la mémoire mais un peu plus simple. Dans cette

conception, quelques soit les dimensions du tableau, on alloue seulement deux blocs sur le tas : un contenant les éléments et l'autre les tailles de chaque dimensions :



Bien que cette solution demande moins d'adresses sur le tas, elle a l'inconvénient de nécessiter des adresses consécutives pour les éléments, ce qui peut vite devenir conséquent. Cependant, la conception a été grandement facilitée. Avec ce choix, nous avions par exemple l'algorithme de sélection efficace suivant (écrit en pseudocode) :

```
{
    // Notons : float[]...[] tab = new float[t1][t2]...[t(n-1)][tn], voici notre
    algorithme pour l'instruction de selection "tab[k0][k1]...[k(n-1)][kn]"
    int somme = 0;
    int facteur = 1;
    for (i allant de n a 0) {
        -> verifier si ki est out of range;
        somme = somme + facteur * ki;
        facteur = facteur * (taille de la dimension i);
    }
    return somme;
}
```

La variable somme retournée désigne l'offset à effectuer pour accéder à la case mémoire de la valeur. Selon le type d'instruction dans laquelle est impliquée la sélection, il faut retourner l'adresse de l'élément (*LEA*) ou la valeur (*LOAD*) à cette adresse. Cette algorithme est aussi utilisée pour les ArrayLiteral, c'est-à-dire les tableaux initialisés explicitement, comme dans l'exemple ci-dessous. (Plus de détails dans les fichiers ArraySelection.java et ArrayLiteral.java de fr.ensimag.deca.tree)

```
{
    int[] tab;
    int[][] mat;
    tab = {1, 2, 3, 4, 5};
    mat = {{1, 2}, {3, 4}};
```

```
6 }
```

Bien que cet algorithme soit central pour la sélection d'élément d'un tableau, il nous a fallu ajouter un autre outil pour procéder à une sélection. En effet, il est expliqué dans la description de la machine abstraite (polycopié [Machine Abstraite] 2. Modes d'adressages) que l'on dispose du mode d'adressage par registre indirect avec déplacement et index *d(XX, Rm)*. Dans le cas général l'indice de sélection obtenu (somme) est disponible dans un registre (Rn) et l'adresse du tableau sur le tas dans un autre (Rp), il fallait donc faire un accès de la forme *1(Rp, Rn)* (le 1 venant du fait que dans les éléments ne commencent que dans la deuxième adresse mémoire, la première étant l'adresse du tableau des tailles des dimensions, cf figure page 7). Mais cela n'était pas permis par la classe RegisterOffset.java (dans fr.ensimag.ima.pseudocode) qui ne permet qu'un adressage de type registre indirect avec déplacement (*d(XX)*. Nous avons donc ajouté la classe RegisterOffsetRegister.java, nous permettant la sélection.

3.5 Analyse bibliographique pour les bibliothèques

3.5.1 Objectif

Pour l'implémentation de la bibliothèque de calcul, l'objectif est de réussir à implémenter les différentes opérations usuelles sur les tableaux et les matrices, ainsi que des algorithmes classiques que nous détaillerons (principe et complexité notamment).

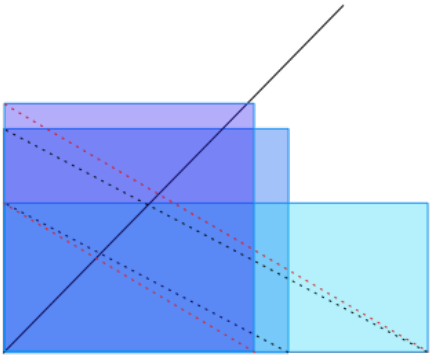
3.5.2 Tableau et racine carrée

Pour les tableaux, cela consiste à implémenter des méthodes d'ajout, de suppression, d'insertion d'un élément. Les méthodes sont alors inspirées des signatures de méthodes tels que les ArrayList et LinkedList en JAVA. Nous incluons aussi dans cette partie notre algorithme de calcul de racine carrée qui a été utilisé régulièrement dans notre bibliothèque.

Commençons par notre algorithme de calcul de racine carrée : la méthode de Héron ([1]). Il a fallu faire un choix pour évaluer la racine carrée. Notre première idée s'est naturellement dirigée vers la méthode de newton (étant donné que résoudre $x = \sqrt{a}$ est équivalent à $x^2 = a$ et que la dérivée de x^2 est $2x$, facilement calculable). Nous avons donc appris que la méthode de Newton appliquée au calcul de racine carrée était en fait connue depuis longtemps (Grèce Antique) et s'appelait la méthode de Héron. Celle-ci possède en fait une interprétation géométrique forte : trouver x tel que $x^2 = a$ revient en fait à chercher un carré de côté x d'aire a. L'idée est donc de commencer avec un rectangle de longueur a et de largeur 1 (aire a), puis de le "rendre de plus en plus carré". Pour ce faire, on prend la moyenne arithmétique de la longueur et de la largeur, notée m et on prend un rectangle de longueur m et de largeur a/m. Ceci donne le pseudo-code suivant :

```
// Pseudo-code
// entree : float a
// sortie : float b, evaluant la racine de a
seuil_de_precision = 10e-04;
b = 1
tant que (b*b - a > seuil_de_precision) faire :
    b = (b + a/b)/2
return b;
```

Figure illustrant différentes étapes de l'algorithme de Héron (plus de détails en source [1]).



La convergence de l'algorithme de Héron (ou Newton) est quadratique, ce qui pour notre bibliothèque était amplement suffisant, son utilisation étant essentiellement pour calculer des normes vectorielles/matricielles.

Présentons maintenant notre algorithme de tri pour les tableaux. Il en existe de multiples (tri bulle, insertion ..). Nous savons que l'un des algorithmes les plus répandus est le tri fusion. Sa complexité est meilleure qu'un tri basique (souvent quadratique), elle est en $O(nlog(n))$, c'est pourquoi nous l'avons choisi. Cet algorithme est construit sur le principe du diviser pour régner. En effet, il est récursif : on casse le tableau en deux, on tri chacune des deux parties, puis on utilise l'algorithme de fusion qui prend en entrée deux listes triées et en renvoie la concaténation triée. Voici le pseudocode associé à cet algorithme (à noter que nous prenons un tri d'entier par ordre croissant mais que cet algorithme s'adapte à tout ensemble d'objet ayant une relation d'ordre totale, entre autre nous aurions pu l'adapter au tri lexicographique si les string était plus intégrés au langage deca) :

```
// Pseudo-code de tri_fusion
// entree : tableau tab
// sortie : tableau tab_trie
int n = length(tab);
si (n == 1) alors :
    return tab;
sinon :
    tab1 = tri_fusion(tab[0:n/2]);
    tab2 = tri_fusion(tab[n/2:n]);
    return fusion(tab1, tab2);
```

```
// Pseudo-code de fusion
// entrees : tableaux tab1 et tab2 tries
// sortie : tableau tab_trie
tab_trie = [];
int i1 = 0;
int n1 = length(tab1);
int i2 = 0;
int n2 = length(tab2);
while (i1 < n1 et i2 < n2) faire :
```

```
    si (tab1[i1] > tab2[i2]) faire
        tab_trie.ajouter(tab2[i2]);
        i2++;
    sinon faire :
        tab_trie.ajouter(tab1[i1]);
        i1++;
    fin_si
fin_while
si (i1 == n1) faire :
    return tab_triee + tab2[i2:];
sinon :
    return tab_triee + tab1[i1:];
```

3.5.3 Matrices

Pour la partie sur les matrices, nous nous sommes inspirés des bibliothèques de calcul courant tels que numpy pour spécifier les différentes méthodes de notre bibliothèque. Nous avons ensuite défini le cahier des charge qui était d'implémenter les méthodes tels que le produit matriciel, l'inverse et le déterminant d'une matrice ainsi que la recherche de valeurs et vecteurs propres.

3.5.4 Produit matriciel

Le produit matriciel AB se fait si le nombre de colonne de la première matrice équivaut au nombre de ligne de la seconde matrice (2).

Si $A = (a_{ij})$ est une matrice de dimension (m, n) et $B = (b_{ij})$ est une matrice de dimension (n,l) alors on peut définir le produit matriciel par :

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

$$Exemple : \begin{pmatrix} 1 & 3 \\ 3 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix} = \begin{pmatrix} 1 \times 1 + 3 \times 3 & 1 \times 2 + 3 \times 1 \\ 3 \times 1 + 1 \times 3 & 3 \times 2 + 1 \times 1 \end{pmatrix} = \begin{pmatrix} 10 & 5 \\ 6 & 7 \end{pmatrix}$$

```
// Pseudo-code produit matriciel
// entree deux matrices A[m][n] et B[n][l]
// sortie C[m][l]
pour tout i de 0 a m-1:
    pour tout j de 0 a l-1  :
        pour tout k de 0 a n  :
           C[i][j] = C[i][j] + A[i][k]* B[k][j]
        fin pour
    fin pour
fin pour
```

3.5.5 Algorithme de reduction de Gauss-Jordan

Pour le calcul de l'inverse et du déterminant, on utilise l'algorithme de réduction de Gauss-Jordan dont le pseudo-code est le suivant (3) :

```
// entr e: matrice A
// sortie: matrice A'    chelonne    r duite
Gauss-Jordan
r = 0  // (r est l'indice de ligne du dernier pivot trouve)
Pour j de 1 a m  //(j d crit tous les indices de colonnes)
     Rechercher max(|A[i,j]|,r+1<=i<=n)// Noter k l'indice de ligne du maximum
                                       // (A[k,j] est le pivot)
        Si A[k,j]!=0 alors // (A[k,j] d signe la valeur de la  ligne k et de la
    colonne j)
           r=r+1 // (r d signe l'indice de la future ligne servant de pivot)
           Diviser la ligne k par A[k,j]       //(On normalise la ligne de pivot
    de fa on que le pivot prenne la valeur 1)
           Si k!=r alors
                 echanger les lignes k et r //  (On place la ligne du pivot en
    position r)
           Fin Si
           Pour i de 1 jusqu a n //(On simplifie les autres lignes)
              Si i!=r alors
                 Soustraire a la ligne i la ligne r multipliee par A[i,j]
                 // (de facon a annuler A[i,j])
              Fin Si
           Fin Pour
        Fin Si
     Fin Pour
  Fin Gauss-Jordan
```

Si la matrice échelonnée réduite en sortie est la matrice identité. En appliquant les même opérations élémentaires d'échange de ligne, de soustraction et de multiplication à la matrice identité, on obtient l'inverse de A.

Exemple : On part d'une matrice de taille 3×3 inversible

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

On génère la matrice augmentée avec la matrice identité $|A|Id| = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$

Le pivot le plus grand se trouve sur à la première ligne et est égal à 1.

On soustrait la première ligne à la troisième L3 = L3 - L1

on obtient :

$$\begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 & 0 & 1 \end{pmatrix}$$

On passe à la deuxième colonne, le pivot le plus grand est 2.
On divise la deuxième ligne par 2 L2 = L2/2 on obtient :

$$\begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 1 & 0 & -1 & 0 & 1 \end{pmatrix}$$

On fait L3 = L3 - L2, on obtient :

$$\begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & -\frac{1}{2} & -1 & -\frac{1}{2} & 1 \end{pmatrix}$$

On choisit $-\frac{1}{2}$ comme pivot on le multiplie par -2, on obtient :

$$\begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 1 & 2 & 1 & -2 \end{pmatrix}$$

On réalise L2 = L2 - $\frac{1}{2}$ L1 et L3 = L3 - L1 on obtient :

$$\begin{pmatrix} 1 & 0 & 0 & -1 & -1 & 2 \\ 0 & 1 & 0 & -1 & 0 & -1 \\ 0 & 0 & 1 & 2 & 1 & -2 \end{pmatrix}$$

La matrice inverse est donc

$$\begin{pmatrix} -1 & -1 & 2 \\ -1 & 0 & 1 \\ 2 & 1 & -2 \end{pmatrix}$$

Pour calculer le déterminant, On applique la formule : (2)

$det(A) = (-1)^p \prod_{k=1}^{n} |A_{ik}|$
Avec $|A_{ik}|$ la valeur numérique du pivot à l'itération k. Et p le nombre de permutation de lignes. Dans l'exemple précédent, le déterminant vaut alors : $(-1)^0 \times 2 \times -\frac{1}{2} = -1$.

L'algorithme de Réduction de Gauss-Jordan est en $O(n^3)$.

3.5.6 Méthode des puissances itérée

Pour rechercher les valeurs et vecteurs propres, La méthodes des puissances itérées est un algorithme de base permettant de déterminer le rayon spectral d'une matrice carré. Il est à noter que la valeur propre recherchée doit être réelle[4]. On a pour cela besoin d'utiliser la norme euclidienne donnée par la formule suivante :

Soit un vecteur de taille n. $v = \begin{pmatrix} v_1 \\ v_2 \\ .. \\ .. \\ v_n \end{pmatrix}$ Alors sa norme euclidienne est égale à

$||v|| = \sqrt{(v_1)^2 + (v_2)^2 + .. + (v_n)^2}$

Avec cette norme, on peut écrire pour la méthode des puissance le pseudo-code suivant on recherche la valeur propre λ :

```
// entree k = 0, matrice A carre de dimension n vecteur unitaire de taille n
// its: le nombre d iteration
// sortie: vecteur propre associe x et le rayon spectral lambda
Pour tout k de 0 a its
    y = Ax
    x = y diviser par sa norme euclidienne
    lambda = transpose(x)Ax
Fin pour
```

On peut ajouter un critère de seuil la k -ieme itération $|\lambda^{k-1} - \lambda^k|$ qui nous donnera la précision de la valeur propre.

3.5.7 Convergence

Soit λ_1 et λ_2 la première et seconde plus grande valeur propre de la matrice en valeur absolue Alors si λ_1 à la même multiplicité algébrique et géométrique, l'algoritme converge géométrique-

ment en O(($\frac{|\lambda_2|}{|\lambda_1|}$), sinon sa convergence est en O($\frac{1}{n}$).

3.5.8 Méthode de la puissance inverse

Cette méthode se base sur la méthode de la puissance itérée. Elle permet alors de trouver les valeurs et vecteurs propres à valeurs réelle. Il faut alors rentrer une estimation de la valeur propre associée. L'algorithme retournera la valeur propre la plus proche de l'estimation propose et le vecteur propre associé. Soit b l'estimation associée à la valeur propre recherchée. Il faut alors rentrer dans l'algorithme la matrice $(A - bI_d)^{-1}$ En adaptant l'algorithme, on a :[4]

```
// entree k = 0, matrice A carre de dimension n vecteur unitaire de  taille n
// its: le nombre d iteration
// sortie: lambda et vecteur
Pour tout k de 0 a its
    y = Ax
    x = y diviser par sa norme euclidienne
    lambda = transpose(x)Ax
Fin pour
```

Le vecteur propre associé est le même et la valeur propre de A est cette fois-ci $\frac{1}{\lambda} + b$

Exemple tirée de [4] :

$$SoitA = \begin{pmatrix} -3 & 0 & 0 \\ 17 & 13 & -7 \\ 16 & 14 & -8 \end{pmatrix} \text{ avec Sp(A) = \{6, -3, -1\}}$$

Alors PuissanceInverse(0) donne -1 comme valeur propre, PuissanceInverse(-2) donne -3 et PuissanceInverse(4) donne 6.

3.5.9 Algorithme QR

[4] Les méthodes présentée ci-dessus permettent de déterminer des valeurs propres et des vecteurs propres. Cependant, ils permettent de déterminer les valeurs une à une. L'agorithme QR permet de les déterminer toutes à la fois en renvoyant une matrice triangulaire supérieurs dont les valeurs propres sont situées sur la diagonale. On détermine alors le spectre de la matrice. L'algorithme se décompose alors de 2 parties.

3.5.10 Factorisation QR

Soit A une matrice de rang n donc les valeurs propres sont réelles et de rang n, celui ci admet une factorisation A = QR ou Q est une matrice orthogonale et R une matrice triangulaire supérieure dont les éléments diagonaux sont strictement positifs. Pour déterminer cette factorisation, on peut soit utiliser la décomposition de Gram-Schmit [6], soit la méthode de Householder. Cette dernière étant plus stable numériquement, on s'intéressera donc à celle-ci

3.5.11 Méthode de Householder [6]

Soit A la matrice carré de dimension n que l'on va essayer de factoriser.

Soit x un vecteur colonne de dimension n. $x = \begin{pmatrix} a_{i1} \\ .. \\ a_{in} \end{pmatrix}$ x prend successivement la valeur du vecteur de la i-ème colonne de A.

soit e_i le vecteur colonne de la base canonique. On a donc $e_1 = \begin{pmatrix} 1 \\ 0 \\ .. \\ 0 \end{pmatrix}$

On pose $\alpha \pm ||x||, \alpha$ doit être du signe opposé au premier coefficient de a_{1i}.
On pose u = x - αe_i, v = $\frac{u}{||u||}$ La Matrice de Householder à l'indice i est donc égal à :
$Q_i = I - 2vv^\top$ Supposons alors que l'on ait calculé Q_1 alors $R_1 = Q_1 A$ Il faut alors répéter le processus mais cette fois avec la matrice R_1. et faire cette étape n fois. Au final, on aura R = $Q_n Q_{n-1}..Q_1 A$ et Q = $Q_1 Q_2..Q_n$. On obtient donc la décomposition souhaitée.

L'algorithme s'écrit alors avec le pseudo code suivant [5] :

$A_0 = A$, its : le nombre d'itération
Pour tout k de 0 à its
$Factorisation\ Q_k R_k = A_k$
$A_{i+1} = R_i Q_i$
$k = k + 1$
Fin pour

3.5.12 Vitesse de convergence

Le coût de cet algorithme est élevée en O(n^3) mais est relativement stable numériquement.

4 Méthode de validation

Comme deca est un sous langage de Java, nous avons d'abord testé nos algorithmes en Java. Cela permet une première vérification de l'algorithmique de nos programmes. Nous avons ensuite testé si les résultats sont justes en les comparant aux valeurs théoriques. Par exemple, pour l'inverse et le déterminant de matrices, il est facile de les calculer à la main et de les comparer aux résultats attendus. Une fois cela fait, il a fallu les intégrer au langage déca. La programmation de la bibliothèque a aussi permis de trouver des dysfonctionnements dans le langage déca avec objet et constitue de bons tests.

5 Résultat de validation de l'extension

5.1 Bibliothèque de calcul

Pour la bibliothèque de calcul, nous avons donc définie une hiérarchie des classes basique.
La classe AbstractTab regroupant les informations sur les dimensions du tableau.
Une classe TabFLoat qui hérite de AbstractTab permettant la déclaration de tableaux de flottant. Cette classe contient des méthodes telles que l'ajout en tête, en queue, et à un indice fixé d'un élément du tableau ainsi que la suppression en tête, en queue et à indice fixé. Il y a aussi une méthode de tri-fusion par ordre croissant du tableau.
Une classe TabInt qui hérite de AbstractTab contenant les même opérations que TabFloat mais avec comme élément des int.
Pour les matrices, nous avons décidé de n'implémenter que des Matrices de flottant. En effet, les calculs tels que la recherche de valeur propre et de vecteur propre se font avec des flottants. Créer des matrices d'entiers dupliquerait le code et n'est pas très utile. En effet,bien que la bibliothèque permette principalement la résolution de systèmes linéaires et donc les coefficients utilisés sont majoritairement des entiers, il sera possible d'initialiser ces matrices avec des tableaux d'entiers. On aura donc :
Une classe Racine permettant le calcul de la racine carré d'un nombre utile pour calculer la norme euclidienne.
Une classe AbstractMatrice qui contient les données de dimension des matrices.
Une classe MatriceFLoat qui hérite de AbstractMAtrice. celle-ci contient les méthodes de produit matriciel, de calcul de déterminant, de recherche de valeur propre par la méthode des puissance itéré et des puissances inverses pour la recherche de valeur et de vecteurs propre, de la méthode de décomposition QR permettant de trouver le spectre d'une matrice. et des opération de usuelles sur les matrices comme la somme de deux matrices ou la multiplication par un scalaire.

5.2 Résultat

Pour le calcul de l'inverse et du déterminant, comme cela implique des divisions par des flottants, nous obtenons dans le cas général une valeur approchée des coefficients cherchés lors de l'application de la méthode du pivot de Gauss. Pour les recherches de valeurs propres, avec une précision de $10^{-}5$ On obtient pour cette matrice tirée de [4] avec nos algorithmes :

$$A = \begin{pmatrix} -3 & 0 & 0 \\ 17 & 13 & -7 \\ 16 & 14 & -8 \end{pmatrix}, \text{ avec Spe(A) = \{-3, -1, 6 \}}$$

L'algorithme de par puissance itérée permet d'obtenir la valeur arrondie de 6 à 5 décimal qui était la précision fixée pour l'arrêt de l'algorithme. De plus on obtient par la méthode des puissances inverses les valeurs de -3 et -1 en rentrant en argument les valeurs de -2 et -0.5. On obtient aussi les vecteurs propres associées :

$$x_0 = \begin{pmatrix} -1.12392.10^{-7} \\ 7.07107.10^{-1} \\ 7.07107.10^{-1} \end{pmatrix} \quad x_{-1} = \begin{pmatrix} 1.68302.10^{-6} \\ -4.47215.10^{-1} \\ -8.94426.10^{-1} \end{pmatrix} \quad x_{-3} = \begin{pmatrix} 4.85071.10^{-1} \\ -7.27607.10^{-1} \\ -4.85072.10^{-1} \end{pmatrix}$$

Pour l'algorithme QR, nous avons été limités pas la capacité de notre compilateur. En effet, les tableaux créés ne sont jamais désalloués. En conséquence, les matrices intermédiaire au calcul de Q saturent le tas assez vite. On ne peut donc pas mettre un nombre d'itération trop grand pour l'algorithme.
Par exemple, pour une matrice 3x3, le tas sature au bout de 24 itérations. Il est alors difficile de tester la vitesse et la précision de convergence de cet algorithme. Nous avons obtenu des résultats satisfaisant pour des matrices à valeurs propres réelles. Par exemple, on a :

Avec la matrice précédente, on obtient après 23 itérations :

$$A' = \begin{pmatrix} 6.00204 & -14.9390 & 27.5771 \\ 1.21941.10^{-3} & -2.97814 & 70.8022 \\ -4.28274.10^{-10} & -3.78864.10^{-10} & -1.00000 \end{pmatrix}$$

On obtient des valeurs approchées de 6, -3 et -1 sur la diagonale, avec une précision minimale de 0.1 pour les valeurs propres.

6 Sources

https://fr.wikipedia.org/wiki/M%C3%A9thode_de_H%C3%A9ron [1]

https://fr.wikipedia.org/wiki/Produit_matriciel [2]

https://fr.wikipedia.org/wiki/%C3%89limination_de_Gauss-Jordan [3]

http://helios.mi.parisdescartes.fr/~gk/MN_S6/cours_MN.pdf [4]

https://fr.wikipedia.org/wiki/M%C3%A9thode_de_Householder [5]

https://fr.wikipedia.org/wiki/D%C3%A9composition_QR [6]