MATRIX

Operacions amb matrius

Daniel Carrillo Cardús
CFGS Desenvolupament Web | Es Liceu
2019-2020

Introducció

En aquest projecte es presentarà la documentació de la pràctica 'Màtrix' amb l'objectiu d'explicar els mètodes emprats en el codi font del programa. També s'hi detallaràn alguns problemes i errors que han hagut aparesquent al llarg de la implementació dels algorismes.

Cal destacar que s'usarà el llenguatge de programació Java per a la implementació de tots els algorismes i el potent editor de textes LaTeX per a totes les matrius i fórmules matemàtiques que apareguin en aquest document.

Explicació de la pràctica

L'objectiu principal d'aquesta pràctica és el d'implementar una sèrie de funcions que siguin capaços de resoldre diferents operacions amb matrius. Aquestes operacions, entre d'altres, són:

1. Suma i resta.
2. Multiplicació i divisió.
3. Traça d'una matriu quadrada.
4. Càlcul de potències.
5. Càlcul de determinants.
6. Càlcul de la transposada d'una matriu.
7. Verificació de ortogonalitat d'una matriu.
8. Resolució de sistemes d'equacions emprant la Regla de Cramer.

A la pràctica s'emprarà un framework de Java, JUnit, que serveix per a la creació de tests, per tant la nostra classe no tindrà un procediment "main".

Funcions de comprovació de matrius

Funció testSameDimension

Aquesta funció comprova que dues matrius (mat1 i mat2) tenen la mateixa dimensió nxn.

```
private static boolean testSameDimension(double mat1[][], double mat2[][]) {

    // comprova les longituds de les files i columnes
    if (mat1[0].length != mat2[0].length || mat1.length != mat2.length) {
        return false;
    }

    return true;
}
```

Funció squareMatrix

Aquesta funció retornarà el valor booleà “true” si una matriu és quadrada o retornarà “false” si no ho és.

```
private static boolean squareMatrix(double mat[][]) {

    // comprova que la matriu tengui el mateix nombre de files que de columnes
    if (mat.length != mat[0].length) {
        return false;
    }
    return true;

}
```

Funció matrixExists

Aquesta funció és de les més importants, ja que comprova que una matriu existeix. És a dir, que no és una matriu buida [1], una matriu que té diferents element per columnes o files [2], etc…

(1) $\begin{pmatrix} \end{pmatrix}$ (2) $\begin{pmatrix} a & b & \\ c & d & e \\ f & g & \end{pmatrix}$

En aquesta funció es fan varies comprovacions. La primera s'assegura de que la longitud d'una matriu és un nombre que existeix.

```
        // si la longitud de la matriu no existeix
        if (Double.isNaN(mat.length)) {
            return false;
        }
```

També es comprova que les columnes o les files de la matriu no siguin 0. Ens trobaríem amb el cas d'una matriu buida *(veure imatge 1 de l'anterior pàgina)*.

```
        // si les columnes o les files són 0
        if (mat.length == 0 || mat[0].length == 0) {
            return false;
        }
```

La darrera comprovació s'assegura que cada fila de la matriu tingui el mateix nombre d'elements que la fila immediatament posterior *(veure imatge 2 de l'anterior pàgina)*.

```
        // comprova que cada fila tingui el mateix nombre d'elements que
l'immediatament
        posterior
        for (int i = 0; i < mat.length - 1; i++) {
            if (mat[i].length != mat[i + 1].length) {
                return false;
            }
        }
```

Si aconsegueix passar aquestes comprovacions, la funció retornarà "true", és a dir, existeix i és compatible amb les operacions que durem a terme en aquesta pràctica.

Funcions principals

Funció trace

Aquesta funció simplement calcula la suma dels elements de la diagonal d'una matriu. Per poder calcular la traça la matriu ha de ser quadrada, és a dir, ha de ser de dimensions nxn.

$$Trace = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = 1 + 5 + 9 = 15$$

El primer que hem de fer en aquesta (i tota) funció és fer una comprovació per saber si podem fer feina amb les matrius que ens passen per paràmetre. Com que per calcular la traça d'una matriu aquesta ha de ser obligatòriament quadrada, comprovarem si la matriu existeix i, a més, és quadrada.

```
// comprovam que la matriu existeix i és quadrada
if (!matrixExists(mat) || !squareMatrix(mat)) {
    return Double.NaN;
}
```

Cal destacar que com que la funció trace retorna un valor double, no podem retornar un valor *null*. Per solucionar aquest problema, sempre que hi hagi qualque error retornarem *Double.NaN*.

Continuant amb la funció, sumarem els elements de la diagonal de la matriu de la següent manera:

```
// sumam els elements de la diagonal
for (int i = 0; i < mat.length; i++) {
    result = result + mat[i][i];
}

return result;
```

Finalment retornarem la nostra variable "result".

Funció add

L'objectiu d'aquesta funció és sumar o restar dues matrius A + B.

$$A = \begin{pmatrix} 3 & 5 & 10 \\ 7 & 3 & 0 \\ 6 & 5 & 9 \end{pmatrix}, \quad B = \begin{pmatrix} 7 & 1 & 3 \\ 5 & 2 & 6 \\ 0 & 4 & 7 \end{pmatrix}$$

$$\begin{pmatrix} 3 & 5 & 10 \\ 7 & 3 & 0 \\ 6 & 5 & 9 \end{pmatrix} + \begin{pmatrix} 7 & 1 & 30 \\ 5 & 2 & 6 \\ 0 & 4 & 7 \end{pmatrix} = \begin{pmatrix} 10 & 6 & 13 \\ 12 & 5 & 6 \\ 6 & 9 & 16 \end{pmatrix}$$

Per dur a terme l'objectiu de la funció, com en totes, el primer que farem és comprovar la compatibilitat de les matrius que ens passen per paràmetre. En aquest cas necessitam que les dues funcions tinguin la mateixa dimensió nxn:

```
// comprovam que les dues matrius existeixen i, a més, que les dues tenen la mateixa
dimensió
    if (!matrixExists(mat1) || !matrixExists(mat2) || !testSameDimension(mat1, mat2))
{
          return null;
    }
```

Seguidament, inicialitzarem la nostra matriu i anirem sumant els elements de la primera matriu amb el complementari de la segona matriu. És a dir;

$$mat1[n, m] + mat2[n, m] = result[n, m]$$

Aquí podem veure l'algorisme emprat per dur a terme la fórmula anterior.

```
double result[][] = new double[mat1.length][mat1.length];

        // sumam element per element
        for (int i = 0; i < result.length; i++) {
            for (int j = 0; j < result.length; j++) {
                result[i][j] = mat1[i][j] + mat2[i][j];
            }
        }

        return result;
```

Funció mult

Per multiplicar amb matrius ens trobem que tenim dues possibilitats: multiplicar una matriu per una altra matriu, o multiplicar una matriu per un nombre real. Amb aquest dilema present la decisió serà la de fer dues funcions diferents, una per a cada possibilitat.

Producte d'una matriu per una matriu

Començarem amb les comprovacions pertinents abans de fer qualque operació. En aquest cas, s'ha de comprovar que:

1. Les dues matrius existeixen.
2. El nombre d'elements de la primera fila de la primera matriu coincideix amb el nombre d'elements de la primera columna de la segona matriu.

```
// comprovació de que les matrius existeixen i de que es pot fer la
multiplicació
        if (!matrixExists(mat1) || !matrixExists(mat2) || mat1[0].length
!=
              mat2.length) {
            return null;
        }
```

Una vegada comprovades, anem a multiplicar les dues matrius.
Per fer això, hem de sumar els productes de cada element de la fila de la matriu A per cada element de la columna de la matriu B. Per a matrius amb dimensions nxm i mxn tenim que:

$$result_{ij} = \sum_{k=0}^{n} a_{ik} b_{kj}$$

```
// multiplicació de dues matrius
        for (int i = 0; i < result.length; i++) {
            for (int j = 0; j < result[0].length; j++) {
                for (int k = 0; k < result.length; k++) {
                    result[i][j] = result[i][j] + mat1[i][k] * mat2[k][j];
                }
            }
        }
```

Producte d'una matriu per un nombre real

En aquest cas, la multiplicació és molt més senzilla que l'anterior. El procediment per multiplicar una matriu per un nombre real és paregut al de la suma.

El primer que comprovarem serà que la matriu existeix i, a més, que el nombre que ens passen com a paràmetre també existeix. Amb això es vol evitar error quan l'usuari et passi per paràmetre un nombre imaginari, infinit, no definits, etc… Com, per exemple, n/0.

```
// comprovam que tant la matriu com n existeixen
        if (!matrixExists(mat) || Double.isNaN(n)) {
            return null;
        }
```

Una vegada estem segurs que la nostra matriu i el nostre nombre n existeixen, simplement anirem multiplicant cada element de la matriu per aquest valor n, per separat, de la següent manera.

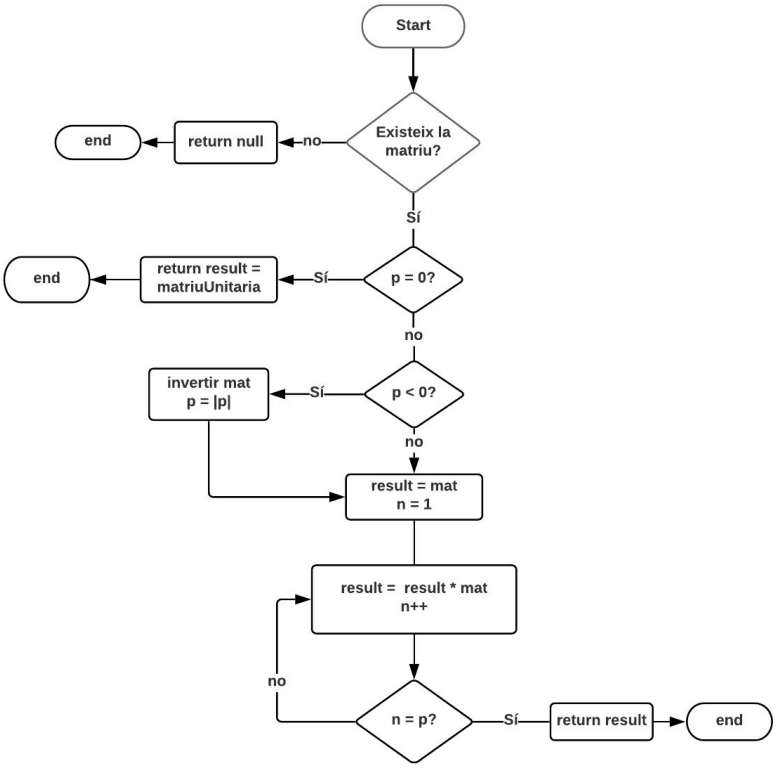
```
double result[][] = new double[mat.length][mat[0].length];

        // multiplicarem cada element de la matriu per n
        for (int i = 0; i < result.length; i++) {
            for (int j = 0; j < result[0].length; j++) {
                result[i][j] = n * mat[i][j];
            }
        }

        return result;
    }
```

Funció power

La potència d'una matriu es pot fer ben igual que amb un nombre real, és a dir, multiplicant aquest nombre per ell mateix tantes vegades com l'exponent expressi.

$$3^4 = 3 * 3 * 3 * 3$$

Aleshores, el diagrama de flux per aquesta funció quedaria:



A l'hora de realitzar aquesta operació tenim certa llibertat en quant a requisits ens referim, ja que en aquest cas només tindrem que comprovar que una matriu, simplement, existeix. És a dir, per fer una potència d'una matriu, aquesta pot ser quadrada o pot no ser-ho.

```
    // comprovació de que la matriu existeix
      if (!matrixExists(mat)) {
          return null;
      }
```

Després, copiarem la matriu "mat" a una altra nova anomenada "result" ja que totes les matrius que ens passen per paràmetre són immutables i no es poden modificar.

```
    // Bucle per copiar la matriu "mat" dins la matriu result amb la que
    farem feina més endavant
      for (int i = 0; i < mat.length; i++) {
          System.arraycopy(mat[i], 0, result[i], 0, mat[0].length);
      }
```

Però, tenim un cas excepcional amb el qual no podem emprar l'algoritme anterior. Aquest cas és el de una matriu elevada a 0. Per a qualsevol nombre real sabem que $n^0 = 1$. Doncs, el mateix ocorre amb una matriu, amb el fet de que l'equivalent a 1 en matriu és la matriu unitària. Per tant;

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}^0 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
      // retorna la matriu unitaria quan una matriu està elevada a 0
      if (p == 0) {
          result = matriuUnitaria(result);
          return result;
      }
```

La funció *matriuUnitaria()* retorna simplement la matriu unitaria amb les dimensions de la matriu que es passi per paràmetre.

```
private static double[][] matriuUnitaria(double[][] mat) {
      double[][] matriuUnitaria = new double[mat.length][mat[0].length];
      for (int i = 0; i < matriuUnitaria.length; i++) {
          for (int j = 0; j < matriuUnitaria[0].length; j++) {
              if (i == j) {
                  matriuUnitaria[i][j] = 1;
              } else {
                  matriuUnitaria[i][j] = 0;
              }
          }
      }
      return matriuUnitaria;
  }
```

Per a exponents negatius tenim un altre procediment. Primer hem d'invertir la matriu i farem l'absolut de p, l'exponent al que elevarem aquesta matriu (ja invertida).

```
// si l'exponent és negatiu invertirem la matriu
if (p < 0){
    mat = invert(mat);
    p = Math.abs(p);
}
```

I més tard, per fer la potència d'una matriu, emprarem la funció mult, en la seva variant de multiplicar una matriu per una altra matriu. En aquest cas, multiplicarem "p" vegades (l'exponent de la potència) la matriu "mat" per la matriu "result", i anirem guardant el valor de la multiplicació dintre "result".

```
for (int i = 1; i < p; i++) {
    result = mult(result, mat);
}
return result;
```

Funció getMinor

Aquesta funció és essencial per, més tard, fer algunes operacions amb ella. el que fa aquesta funció és crear una matriu però eliminant tota la columna i tota la fila que li passem per paràmetre. Es veurà amb un exemple senzill.

Si tenim una matriu A, i volem obtenir el menor de les coordenades [1,2]*, obtenem:

$$A = \begin{pmatrix} 3 & 5 & 8 \\ 7 & 1 & 2 \\ 9 & 0 & 4 \end{pmatrix} \quad , \quad A_{getMinor_{1,2}} = \begin{pmatrix} 7 & 2 \\ 9 & 4 \end{pmatrix}$$

**Cal tenir present que, matemàticament, les coordenades d'una matriu comencen a contar desde 1, però al llenguatge de programació Java, a partir de 0.*

Una vegada explicat el que fa la funció passem al codi. Primer de tot, haurem d'implementar les comprovacions. En aquest cas, estudiarem si la matriu existeix i, a més, si els índexs que ens passen per paràmetre es troben dintre de la nostra matriu.

```
// comprovam que la matriu existeix i que els índexs es troben dintre de
la matriu
        if (!matrixExists(mat)) {
            return null;
        } else if (x < 0 || x > mat.length - 1) {
            return null;
        } else if (y < 0 || y > mat[0].length - 1) {
            return null;
        }
```

Una vegada comprovat això, crearem el nostre array bidimensional on construirem la matriu resultant. Per això hem de tenir en compte que la matriu resultant sempre tindrà una fila i una columna menys que la matriu original. Per això:

```
double result[][] = new double[mat.length - 1][mat[0].length - 1];
```

Ara, construirem la nostra matriu dins l'array creat anteriorment.

Emprarem dos bucles per a recórrer l'array i anirem traslladant els elements que NO compartin fila o columna amb l'element de les coordenades [x, y].

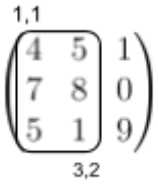
```
        int a = 0, b = 0;       // variables auxiliars

            // copiarem els elements de la matriu excepte aquells que
compartin
        fila o columna amb la coordenada x,y.
        for (int i = 0; i < mat.length; i++) {
              for (int j = 0; j < mat[0].length; j++) {
                  if (x != i && y != j) {
                      result[a][b] = mat[i][j];
                      b++;
                      if (b == result[0].length) {
                        b = 0;
                        a++;
                      }
                  }
              }
          }

          return result;
      }
```

Funció submatrix

Aquesta funció retorna una matriu a partir d'una altra segons els índexs que es passen per paràmetre. És capaç de "crear" un rectangle imaginari damunt la funció i retornar els valors que estiguin dintre d'aquest. amb un exemple es veu clarament.

Donada una matriu A i els índexs de coordenades [1, 1],[3, 2] tenim que:

$$A = \begin{pmatrix} 4 & 5 & 1 \\ 7 & 8 & 0 \\ 5 & 1 & 9 \end{pmatrix}$$



Començarem comprovant que la matriu existeix, i que els índexs de coordenades es troben dintre dels límits de la matriu. També, hem de contemplar l'opció de que l'usuari introdueixi els índexs capgirats, és a dir, quan $x_2 < x_1$ o $y_2 < y_1$. Quan això ocorre simplement intercanviarem els índexs.

```
      // comprovam que la matriu existeix i que els indexs es troben dintre de la
matriu
      if (!matrixExists(mat)) {
          return null;
      } else if (y1 < 0 || y1 > mat.length - 1 || y2 > mat.length - 1) {
          return null;
      } else if (x1 < 0 || x1 > mat[0].length - 1 || x2 > mat[0].length - 1) {
          return null;
      }
      int aux;        // variable auxiliar
      // intercanviar els valors si és necessari
      if (x2 < x1) {
          aux = x1;
          x1 = x2;
          x2 = aux;
      }
      // intercanviar els valors si és necessari
      if (y2 < y1){
          aux = y1;
          y1 = y2;
          y2 = aux;
      }
```

Una vegada fetes les comprovacions pertinents, passarem a la creació de l'array on guardar la matriu resultant. Per això necesitam saber el nombre de files i de columnes que tindrà.
Per averiguar això, recorrerem la matriu amb dos contadors (files i columnes) las espais que queden dintre de les coordenades $[x_1, y_1]$, $[x_2, y_2]$.

```
int files = 0;
int columnes = 0;

// averiguar les files de la nova matriu
for (int i = 0; i < mat.length; i++) {
    if (i >= y1 && i <= y2) {
        files++;
    }
}

// averiguar les columnes de la nova matriu
for (int i = 0; i < mat[0].length; i++) {
    if (i >= x1 && i <= x2) {
        columnes++;
    }
}
```

Una vegada tenim l'array bidimensional creat on guardarem la nostra matriu, és hora de traslladar els elements que es troben dintre del rectangle imaginari creat per les coordenades $[x_1, y_1]$, $[x_2, y_2]$. El mètode emprat per recórrer la matriu varia del de la funció *getMinor*, ja que, en aquest cas, anirem seleccionant els elements inclosos entre x_1 i x_2 per a les files, i el mateix amb y_1 i y_2 per a les columnes.

```
        int a = 0, b = 0;       // variables auxiliars

        // crearem la submatriu
        for (int i = 0; i < mat.length; i++) {
            for (int j = 0; j < mat[0].length; j++) {
                if (i >= y1 && i <= y2 && j >= x1 && j <= x2) {
                    result[a][b] = mat[i][j];
                    b++;
                    if (b >= result[0].length) {
                        b = 0;
                        a++;
                    }
                }
            }
        }

        return result;
    }
```

Funció transpose

El procediment per transposar una matriu és senzill. Només cal intercanviar files per columnes i columnes per files.

$$A = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}, A^t = \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix}$$

Per dur a terme la transposada d'una matriu comprovarem que la matriu existeix.

```
        // comprovacions
        if (!matrixExists(mat)) {
            return null;
        }
```

I, simplement, anirem traslladant els elements de la matriu "mat" de manera inversa a la matriu "result".

```
        // transposam els elements de la matriu
        for (int i = 0; i < result.length; i++) {
            for (int j = 0; j < result[0].length; j++) {
                result[i][j] = mat[j][i];
            }
        }
        return result;
    }
```

Funció determinant

La funció més important, personalment, ja que és emprada a altres funcions molt importants com la funció inversa, funció ortogonal, etc…

Per calcular el determinant d'una matriu d'ordre 1 tenim que el valor del determinant és el valor de l'únic element de la matriu.

$$A = \left(3\right), aleshores \left|A\right| = 3$$

Si ens trobem amb aquest cas, s'implementarà el següent algorisme.

```
// Si la matriu és de 1x1
if (mat.length == 1) {
    result = mat[0][0];
    return result;
}
```

Calculam els determinants d'ordre 2 a través de la següent fórmula:

$$\left|A\right| = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}$$

Per calcular aquests tipus de determinants, s'usarà el següent codi.

```
// Si la matriu és de 2x2
if (mat.length == 2) {
    result = (mat[0][0] * mat[1][1]) - (mat[0][1] * mat[1][0]);
    return result;
}
```

Però la cosa es complica quan tenim un determinant d'ordre 3 o superior, ja que (sense emprar la Regla de Sarrus per a determinants d'ordre 3) hem de trobar la manera de anar convertint aquests determinants en determinants més petits, obtenint els seus menors, fins arribar a molts de determinants d'ordre 2, que ja els sabem calcular.

Ho farem amb una tècnica de programació anomenada recursió. La recursió és basa en cridar a una funció dins aquella pròpia funció.

D'aquesta manera podrem calcular determinants d'ordre 3 o superior. Ja que construirem un algorisme que retorni un determinant d'ordre inferior, i farem aquesta operació tantes vegades com siguin falta fins arribar a un determinant d'ordre 2.

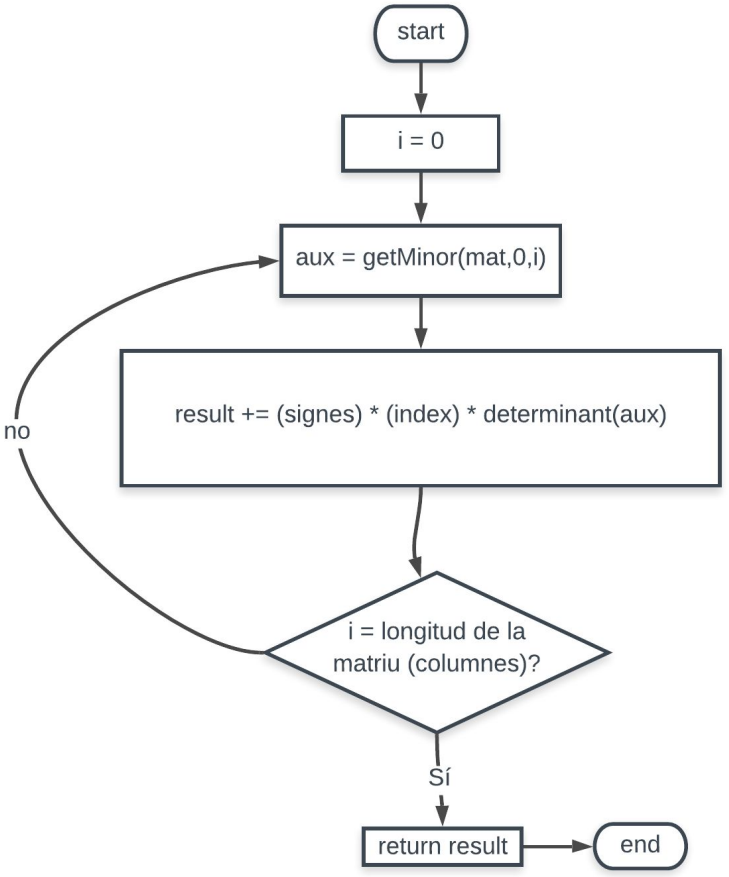
Però, sorgeix un problema que explicarem més endavant. Aquesta és la fórmula matemàtica per calcular un determinant d'ordre 3.

$$det(A) = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} g & f \\ h & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

El problema del qual parlava anteriorment està relacionat amb els signes. Com es pot guardar el signe dels elements parells de la primera fila? La meva proposta és multiplicar tots els índexs (a, b, c) per $(-1)^{i+1}$ on [i, j] son les coordenades de l'element. D'aquesta manera tenim que:

$$(-1)^{1+1}a = 1a = a,\ (-1)^{1+2}b = (-1)b = -b,\ (-1)^{1+3}c = 1c = c, \ldots$$

A continuació es mostrarà el diagrama de flux de l'algorisme emprat.



Per tant, l'algorisme quedaria:

```
        //Si la matriu és de nxn on n>=3
        for (int i = 0; i < mat[0].length; i++) {
            double[][] aux = getMinor(mat, 0, i);
            result = result + (mat[0][i] * Math.pow(-1, i)) *
           determinant(aux);
        }
        return result;
    }
```

Funció invert

Per invertir una matriu hem de dur a terme vàries passes. Primer s'ha d'adjuntar la matriu, seguidament s'ha de transposar i finalment s'ha de dividir per el determinant de la matriu inicial. Aquesta és la fórmula.

$$(A)^{-1} = \frac{(Adj(A))^t}{det(A)}$$

L'únic requisit per invertir una matriu és que sigui quadrada, per tant:

```
        // comprovam que la matriu existeix i que és quadrada
        if (!matrixExists(mat) || !squareMatrix(mat)) {
            return null;
        }
```

També ens hem d'assegurar de que el determinant de la funció sigui diferent de 0.

```
        // no es pot invertir si el determinant de la matriu és 0.
        if (determinant == 0) {
            return null;
        }
```

Després s'ha de calcular la matriu d'adjunts. La matriu d'adjunts és la matriu en la que cada element se substitueix per el determinant del seu adjunt. Amb un exemple es veu més clarament.

$$A = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}, A^{adjunta} = \begin{pmatrix} \begin{vmatrix} e & f \\ h & i \end{vmatrix} & -\begin{vmatrix} d & f \\ g & i \end{vmatrix} & \begin{vmatrix} d & e \\ g & h \end{vmatrix} \\ -\begin{vmatrix} b & c \\ h & i \end{vmatrix} & \begin{vmatrix} a & c \\ g & i \end{vmatrix} & -\begin{vmatrix} a & b \\ g & h \end{vmatrix} \\ \begin{vmatrix} b & c \\ e & f \end{vmatrix} & -\begin{vmatrix} a & c \\ d & f \end{vmatrix} & \begin{vmatrix} a & b \\ d & e \end{vmatrix} \end{pmatrix}$$

Per implementar la fórmula anterior, farem:

```
        // sustituim cada element de la matriu per el determinant del seu adjunt.
        for (int i = 0; i < result.length; i++) {
            for (int j = 0; j < result[0].length; j++) {
                    result[i][j] = (Math.pow(-1, i + j)) * determinant(getMinor(aux, i,
j));
            }
        }
```

Es pot veure a la fórmula que tenim el mateix problema a l'hora dels signes, per tant, s'ha aplicat exactament la mateixa solució que abans.

La següent passa és transposar la matriu. Emprarem la funció transpose creada anteriorment.

```
    result = transpose(result);
```

I, finalment, multiplicarem la matriu adjunta i transposada per 1/determinant(matriu).

```
    result = mult(result, 1 / determinant);

    return result;
```

Funció div

No es pot realitzar la divisió de dues matrius directament, però es pot arribar a aconseguir si s'aplica el següent:

$$\frac{a}{b} = a(b)^{-1}$$

Per aquesta funció s'ha de comprovar que les dues matrius siguin quadrades, i que el determinant de la segona matriu és diferent de 0.

```
    // comprovacions pertinents
                if  (!matrixExists(mat1)  ||  !matrixExists(mat2)  ||
!squareMatrix(mat1)
       || !squareMatrix(mat2)) {
           return null;
       } else if (determinant(mat2) == 0) {
           return null;
       }
```

Seguidament, només hem de multiplicar (amb la funció ja creada *mult*) la primera matriu per la inversa de la segona.

```
double[][] result = mult(mat1, invert(mat2));
```

Funció isOrtho

Una matriu ortogonal és aquella que la seva transposada coincideix amb la seva inversa.

$$(A) \text{ és ortogonal, si i només si, } (A)^t = (A)^{-1}$$

Per poder arribar a saber si una matriu és ortogonal, aquesta ha de ser quadrada, i el seu determinant ha de ser diferent de 0. Per tant:

```
        // comprovacions
         if (!matrixExists(mat) || determinant(mat) == 0 ||
         !squareMatrix(mat)) {
             return false;
         }
```

Una vegada hem emprades les funcions creades anteriorment per a transposar i per a invertir la matriu que hem de averiguar si és ortogonal, simplement hauriem d'igualarles per a saber si són la mateixa, però va sorgir un problema. L'algorisme era correcte, però el mètode *Arrays.deepEquals()* que s'estava emprant per comparar les matrius no era el més adequat, ja que, quan s'enviava la matriu a invertir-la, aquesta retornava amb distincions de 0 i -0, en ocasions molt puntuals.

La solució d'aquest problema podia passar per fer una comprovació abans de retornar la matriu de la funció *invert()*, i recórrer la matriu per comprovar si hi ha algún -0. Però finalment es va decidir per, simplement i dintre de la funció *isOrtho()*, restant element per element les dues matrius, aplicant-hi un "delta" d'error molt petit, i si la resta de dos element és major que aquest delta significa que aquests dos elements són diferents, i, per tant, les dues matrius són diferents.

```
for (int i = 0; i < mat.length; i++) {
            for (int j = 0; j < mat.length; j++) {
                double a = Math.abs(trans[i][j]);
                double b = Math.abs(inv[i][j]);
                if (a - b > 0.0001 || a - b < -0.0001 ) {
                    return false;
                }
            }
        }
        return true;

    }
```

En aquest cas, el nostre delta és molt petit, de 0.0001. Això vol dir que si la resta de dos elements de les matrius és major a 0.0001, aquests dos elements són diferents. Per exemple:

$$(A)^t = \begin{pmatrix} 3.01 & 5 & 9 \\ 6.67 & 5.2 & 8 \\ 4 & 0 & 3 \end{pmatrix}, (A)^{-1} = \begin{pmatrix} 3.01 & 5 & 9 \\ 6.66 & 5.2 & 8 \\ 4 & 0 & 3 \end{pmatrix}$$

Es pot veure que a l'hora de comparar l'element [2, 1] de les dues matrius:

$$6.67 - 6.66 > 0.0001$$

Aleshores, aquestes matrius no serien iguals, per tant, la matriu (A) no seria ortogonal.

Funció cramer

La Regla de Cramer és molt útil per a la resolució de sistemes d'equacions, encara que per a sistemes de moltes incògnites és molt ineficient, i a la pràctica s'empren altre tipus de mètodes.

La Regla de Cramer "transforma" un sistema d'equacions a una matriu de la següent manera.

$$\begin{cases} 2x & +y & +z & = 1 \\ x & +y & -3z & = 4 \\ 3x & +2y & = 4 & \end{cases} = \begin{pmatrix} 2 & 1 & -1 & 1 \\ 1 & 1 & -3 & 4 \\ 3 & 2 & 0 & 4 \end{pmatrix}$$

A aquesta matriu se li diu matriu ampliada, ja que conté, a la darrera columna, la columna de "resultats" del sistema. És important que a la matriu que no conté aquesta darrera columna se li diu simplement "matriu", ho necessitarem saber més endavant.

Aleshores, Cramer diu que hem de substituir la columna de "resultats" de la matriu ampliada a la matriu "petita" segons la incògnita que volguem resoldre. Després s'ha de fer la divisió del determinant de la matriu que s'acaba de substituir entre el determinant de la matriu "petita" inicial. I així per a cada incògnita. Amb un exemple es veu clarament.

$$x = \frac{\begin{vmatrix} 1 & 1 & -1 \\ 4 & 1 & -3 \\ 4 & 2 & 0 \end{vmatrix}}{\begin{vmatrix} 2 & 1 & -1 \\ 1 & 1 & -3 \\ 3 & 2 & 0 \end{vmatrix}} \qquad y = \frac{\begin{vmatrix} 2 & 1 & -1 \\ 1 & 4 & -3 \\ 3 & 4 & 0 \end{vmatrix}}{\begin{vmatrix} 2 & 1 & -1 \\ 1 & 1 & -3 \\ 3 & 2 & 0 \end{vmatrix}} \qquad z = \frac{\begin{vmatrix} 2 & 1 & 1 \\ 1 & 1 & 4 \\ 3 & 2 & 4 \end{vmatrix}}{\begin{vmatrix} 2 & 1 & -1 \\ 1 & 1 & -3 \\ 3 & 2 & 0 \end{vmatrix}}$$

Aleshores, ja podem continuar amb l'algorisme. Després d'haver comprovat que la matriu ampliada existeix, és hora de comprovar que aquesta matriu ampliada només tingui una columna més que la matriu més petita, ja que, en un sistema d'equacions, hi ha tantes incògnites com equacions, no hi pot haver més incògnites que equacions o a l'inversa. Per tant, la matriu ampliada hauria de tenir només una columna més que la matriu "petita". Es pot solucionar això comprovant que la matriu "petita" és quadrada.

```
// comprova que la matriu ampliada tengui només una columna més que files.
        if (!squareMatrix(matriu)) {
            return null;
        }
```

Continuam comprovant que el determinant de la matriu "petita" és diferent de 0, ja que a l'hora de realitzar la divisió, quan dividim entre 0, no està definit.

```
// no es pot continuar si el determinant de la matriu és 0
        if (determinantMatriu == 0) {
            return null;
        }
```

Seguidament, substituirem la darrera columna de la matriu ampliada, la dels "resultats" del sistema, en la nostra matriu "petita", segons la incògnita que estem resolent.

```
      // sustituirem la columna desitjada
        for (int j = 0; j < matriu.length; j++) {
               matriu[j][i] = matriuAmpliada[j][matriuAmpliada[0].length -
1];
        }
```

Realitzarem l'operació:

```
result[i] = determinant(matriu) / determinantMatriu;
```

I re-substituirem la columna que hem modificat. Això es fa perquè, quan acabi de resoldre una incògnita, es pugui resoldre la següent sense haver de fer molts de bucles diferents, sino emprant el mateix, com s'ha fet.

```
// resustituirem la columna modificada per la inicial
            for (int j = 0; j < matriu.length; j++) {
                matriu[j][i] = matriuAmpliada[j][i];
            }
```

Els resultats de x, y, z… s'aniràn guardant dins un array unidimensional en ordre i es retornarà quan s'hagi completat la resolució de totes les incògnites.

Conclusió

Ha quedat claríssim la importància que tenen els arrays bidimensionals, però no només a l'hora de resoldre matrius de diferents dimensions, sinó també de la potència que tenen aquests a l'hora de fer qualsevol cosa, i la potència que poden arribar a tenir. També ha ajudat a assolit coneixements de LaTeX, un editor de text molt potent en quant a la inserció de fórmules matemàtiques en un projecte d'aquestes característiques, i cal recalcar que, en aquesta pràctica, totes les matrius i equacions presents han estat introduïdes amb aquest editor de texte, amb exemples propis, cosa que explica que aquesta documentació no tingui bibliografía.

Aquesta pràctica, definitivament, ha ajudat a la consolidació dels coneixements en quant a arrays bidimensionals es refereix i tot el relacionat amb funcions en programació.