

Clase adicional 8

Temas

Métodos de matrices

- La clase Matrix
- Método de suma de matrices
- Método de producto de matrices

Sistemas lineales

Colas y pilas

- Biblioteca java.util
- La clase Stack
- Marco para las colecciones de Java
- Interfaz Queue

Métodos de matrices

En esta sección analizaremos algunos de los métodos de la clase Matrix tratados en el material de clase.

La clase Matrix La clase Matrix es una clase de Java creada para representar y llenar matrices. Se utiliza sobre todo para preparar matrices para su uso en la resolución de sistemas lineales. Consta de:

§ Un array 2D de m filas por n columnas

```
private double[ ][ ] data = new double[m][n];
```

```
int nrows = data.length;
```

```
int ncols = data[0].length;
```

§ Métodos para operaciones con matrices, como la suma, la resta o la multiplicación

```
Public Matrix addMatrices (Matrix b)
```

```
Public Matrix multMatrices (Matrix b)
```

```
Public void print ();
```

Suma de matrices

Dos matrices (a y b) se pueden sumar sólo si tienen el mismo número de filas y columnas. El resultado es una nueva matriz. La base del proceso de suma es:

```
c[i][j] = a[i][j] + b[i][j];
```

Éste es el código

```
Public Matrix addMatrices (Matrix b)
{
    //La matriz resultante
    Matrix result = new Matrix(nrows, ncols);
    //Sólo se suman si tienen el mismo número de filas y de columnas
    if (b.nrows == nrows && b.ncols == ncols)
    {
        for (int i=0; i<nrows; i++)
            for (int j=0; j<ncols; j++)
                result.data[i][j] = data[i][j] + b.data[i][j];
    }
    return result;
}
```

Producto de matrices

Dos matrices se pueden multiplicar sólo si se cumple $a.ncols = b.nrow$. El resultado es una nueva matriz. La base del proceso de multiplicación es:

Por ejemplo:

$$c[0][0] = a[0][0] * b[0][0] + a[0][1] * b[1][0] + a[0][2] * b[2][0] + \dots + a[0][ncols] * b[nrows][0];$$

Éste es el código

```
Public Matrix multMatrices (Matrix b)
{
    //La matriz resultante tiene a.nrows por b.ncols
    Matrix result = new Matrix(nrows, b.ncols);
```

```

//Se multiplica sólo si a.ncols = b.nrow
if (b.nrows == ncols)
{
  for (int i=0; i<nrows; i++)
    for (int j=0; j<b.ncols; j++)
      for (int k=0; k<ncols; k++)
        result.data[i][j]+= data[i][k] * b.data[k][j];
}
return result;
}

```

Sistemas lineales

En muchos problemas de ingeniería, se obtienen varias ecuaciones simultáneas: se trata de un sistema de n ecuaciones con n incógnitas. Las matrices se suelen utilizar para resolver sistemas lineales. La formulación es la siguiente:

$$AX=B$$

donde A es la siguiente matriz,

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

$$X = (x_1, x_2, \dots, x_n)$$

$$B = (b_1, b_2, \dots, b_n)'$$

Por ejemplo: $2X_0 + 3X_1 + 5X_2 = 10$

$$X_0 + 2X_1 + 4X_2 = 5$$

$$4X_0 + 7X_1 + 3X_2 = 15$$

A es una matriz de 3 por 3: $\begin{pmatrix} 2 & 3 & 5 \\ 1 & 2 & 4 \\ 4 & 7 & 3 \end{pmatrix}$

$$1 \ 2 \ 4$$

$$4 \ 7 \ 3$$

X es una matriz de 3 por 1: $\begin{pmatrix} X_0 \\ X_1 \\ X_2 \end{pmatrix}$

$$X_1$$

$$X_2$$

B es una matriz de 3 por 1: $\begin{pmatrix} 10 \\ 5 \\ 15 \end{pmatrix}$

$$5$$

$$15$$

En el material de clase ya se abordó el proceso para derivar programas de Java y poder resolver un sistema lineal. En esta ocasión, le enseñaremos cómo utilizarlo. Básicamente, se divide en tres pasos:

1. Crear la matriz a de la izquierda
2. Crear la matriz b de la derecha
3. Interconectar a y b en el método gaussiano de la clase Gauss

Sigamos ahora los pasos necesarios para resolver el problema anterior:

1. Crear la matriz a de la izquierda

(Para simplificar el código, asumiremos que todos los *arrays* de las matrices son públicos)

```
Matrix a = new Matrix(3,3);
```

```
a.data[0][0] = 2.0;
```

```
a.data[0][1] = 3.0;
```

```
a.data[0][2] = 5.0;
```

```
a.data[1][0] = 1.0;
```

```
a.data[1][1] = 2.0;
```

```
a.data[1][2] = 4.0;
```

```
a.data[2][0] = 4.0;
```

```
a.data[2][1] = 7.0;
```

```
a.data[2][2] = 3.0;
```

2. Crear la matriz b de la derecha

```
Matrix b = new Matrix(3,1);
b.data[0][0] = 10.0;
b.data[1][0] = 5.0;
b.data[2][0] = 15.0;
```

3. Interconectar a y b en el método gaussiano de la clase Gauss

```
Matrix result = new Matrix(3,1);
GaussMain.gaussian (a, b, result);
```

```
//Imprimir la solución
```

```
result.print();
```

Éste es el resultado

X0 = 6.0

X1 = -1.5

X2 = 0.5

Como puede ver, resolver un sistema lineal con la clase Gauss es realmente sencillo. Si está interesado en saber más sobre el modo en que hemos derivado el método gaussiano, consulte el apéndice

Colas y pilas

Una estructura de datos es una forma sistemática de organizar los datos y acceder a ellos; ya hemos visto varias estructuras de datos simples en este curso (por ejemplo, los arrays y los vectores). En esta sección (y en la siguiente clase adicional), le enseñaremos a implementar algunas de las estructuras de datos tradicionales (Queue, Stack, Linked List y Tree) con Java.

Biblioteca java.util

Antes del lanzamiento de la plataforma Java 2, la biblioteca java.util proporcionaba un pequeño conjunto de clases para las estructuras de datos más útiles, como son Vector, Stack y Hashtable:

La clase Vector implementa un array de objetos que crece con el tiempo. Al igual que ocurre con un array, contiene componentes a los que se puede acceder utilizando un índice integer. Sin embargo, el tamaño de un vector puede crecer o reducirse tanto como sea necesario para poder agregar y eliminar elementos una vez creado el vector.

La clase Stack representa una pila de objetos LIFO (el último en entrar es el primero en salir). Proporciona un método de empuje para agregar un objeto en la parte superior de la pila y un método de extracción para sacarlo.

Por su parte, la clase Hashtable almacena objetos como pares de clave y valor a los que se puede tener acceso de forma aleatoria mediante la clave.

Clase Stack

Una pila es un contenedor de objetos que se insertan y se extraen siguiendo el principio LIFO (el último en entrar es el primero en salir). Un ejemplo de este principio podría ser una lata de pelotas de tenis. La última pelota introducida en la lata será la primera que se pueda sacar. La clase `java.util.Stack` contiene los métodos siguientes:

```
void push(elemento Object)
introduce un elemento en la pila. Parámetros: el elemento que se va a agregar
Object pop()
extrae y devuelve el elemento superior de la pila. No llame a este método si la
pila está vacía.
Object peek()
devuelve la parte superior de la fila sin extraerla. No llame a este método si
la pila está vacía.
```

En el siguiente ejemplos, simularemos una lata de pelotas de tenis con dos clases: `TennisBall` y `TennisBallCan`. En primer lugar, definiremos una clase simple `TennisBall` que contenga un solo atributo: `color`.

```
class TennisBall{
    String color;

    TennisBall (String color){
        this.color = color;
    }

    String getColor() {
        return color;
    }
}
```

A continuación, definiremos la clase `TennisBallCan` que utiliza una pila para organizar las pelotas.

```
import java.util.*;
import TennisBall; class TennisBallCan {

    public static void main (String args[]) {

        //Crear 3 objetos de TennisBall

        TennisBall t1 = new TennisBall ("roja") ;
        TennisBall t2 = new TennisBall ("verde");
```

```

TennisBall t3 = new TennisBall ("amarilla")
Stack can = new Stack();

//Introducir las pelotas en la lata

System.out.println ("Pelota " + t1.getColor() + " dentro");
can.push (t1);
System.out.println ("Pelota " + t2.getColor() + " dentro");
can.push (t2) ;
System.out.println ("Pelota " + t3.getColor() + " dentro");
can.push (t3);
System.out.println ("Contenido de la lata: " + can.size());

//Echar un vistazo a la última pelota

System.out.println ("Color de la última pelota: color = " +
((TennisBall)can.peek()).getColor());
System.out.println ("Contenido de la lata tras el análisis: " + can.size());

//Extraer las pelotas

for (int i=0; i<3; i++)
    System.out.println (("Pelota " + (TennisBall)can.pop()).getColor() + " fuera" );
}
}

```

Éste es el resultado

Pelota roja dentro

Pelota verde dentro

Pelota amarilla dentro

Contenido de la lata: 3

Color de la última pelota: amarilla

Contenido de la lata tras el análisis: 3

Pelota amarilla fuera

Pelota verde fuera

Pelota roja fuera

Contenido de la lata tras la extracción: 0

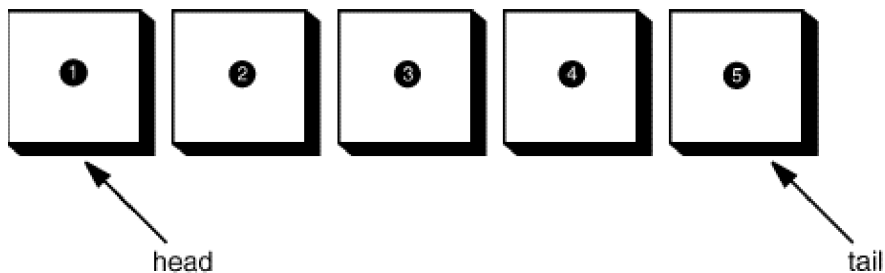
Marco para las colecciones de Java

Presentado con la plataforma Java™ 2, el marco para las colecciones de Java proporciona un conjunto bien diseñado de interfaces y clases para almacenar y manipular grupos de datos como una sola unidad: una colección. El marco ofrece una API adecuada para muchos de los tipos de datos abstractos: mapas, conjuntos, listas, árboles, arrays, tablas *hash* y otras colecciones. Por su diseño orientado a objetos, las clases de Java de este marco de colecciones agrupan tanto las estructuras de datos como los algoritmos asociados con estas abstracciones.

Tal como suele suceder con las bibliotecas de estructuras de datos actuales, la biblioteca de la colección de Java separa las interfaces y las implementaciones. A continuación, estudiaremos dicha separación con una estructura de datos familiar: la cola. La biblioteca de Java no proporciona una clase de cola ni una interfaz, pero aun así es un buen ejemplo para presentar los conceptos básicos.

Interfaz Queue

Una interfaz de cola especifica que se pueden añadir elementos al final de la cola, eliminarlos de la cabecera y saber cuántos elementos existen en la cola. Utilizará la cola cuando necesite recopilar objetos y recuperarlos según la secuencia "el primero en entrar es el primero en salir".



La interfaz Queue debe realizar las operaciones siguientes:

```
interface Queue{  
    void add(Object obj);  
    Object remove();  
    int size();  
}
```

Observe que la interfaz no aporta ninguna información sobre el modo de implementación de la cola. A continuación presentamos un ejemplo para ilustrar cómo se puede implementar la interfaz Queue:

```
class TennisBallQueue implements Queue {  
    public void add(Object obj) { . . . }  
    public Object remove() { . . . }
```

```
public int size() { . . . }  
}
```

NOTA: Si realmente necesita una cola, simplemente puede utilizar la clase `LinkedList` de la biblioteca `java.util`. Entre otros métodos, `LinkedList` proporciona las siguientes operaciones "similares a las de las colas":

`void addLast(Object o)`
Añade un elemento dado al final de la lista.

`Object removeFirst()`
Elimina y devuelve el primer elemento de la lista.

Hablaremos de `LinkedList` la próxima semana.

4) Problemas de la clase adicional

1. Métodos de matrices

Tenemos tres matrices: A, B y C

1 2 3
A = 5 6 7

9 4 5
3 2 4

B = 2 1 6
4 10 20

3
C = 5

6

¿Cuál es el resultado de $(A-B)*C$?

2. Sistemas lineales

Resuelva el siguiente sistema lineal para X, Y y Z.

$$X + Y + Z = 8$$

$$X + Y - Z = 4$$

$$X - Y = 4$$

3. Pila

He modificado el ejemplo TennisBallCan y le he añadido las siguientes acciones en mi programa:

Crear una lata vacía

Mover todas las pelotas de una lata a otra

Las pelotas de la nueva lata deben tener el orden contrario al de la lata original

5) Apéndice: ecuaciones lineales simultáneas

Son muchos los problemas de ingeniería en los que aparecen ecuaciones simultáneas: un sistema de n ecuaciones con n incógnitas. La formulación de la matriz para estos sistemas lineales es la siguiente:

$$AX=B,$$

donde A es la siguiente matriz,

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

$$\text{y } X = (x_1, x_2, \dots, x_n)', B = (b_1, b_2, \dots, b_n)'$$

Los pasos básicos para la resolución de ecuaciones simultáneas es la siguiente.

Triangularización

Es preciso convertir el conjunto de ecuaciones a una forma en que sólo se conserven los elementos triangulares superiores. Este proceso recibe el nombre de triangularización. El conjunto triangularizado de ecuaciones es:

$$a_{11} * x_1 + a_{12} * x_2 + a_{13} * x_3 + \dots + a_{1n} * x_n = b_1$$

$$0 * x_1 + a'_{22} * x_2 + a'_{23} * x_3 + \dots + a'_{2n} * x_n = b'_2$$

$$0 * x_1 + 0 * x_2 + a'_{33} * x_3 + \dots + a'_{3n} * x_n = b'_3$$

.....

$$0 * x_1 + 0 * x_2 + 0 * x_3 + \dots + a'_{nn} * x_n = b'_n$$

El concepto que se esconde detrás de este proceso es muy sencillo. Sabemos que si multiplicamos los dos lados de una ecuación, ésta no cambia. Nuestro objetivo es conseguir que los elementos por debajo de la diagonal sean cero. Así, si comenzamos con la primera columna, necesitaremos convertir a cero todos los elementos excepto el primero. Asimismo, no queremos modificar las ecuaciones. Consideremos primero la fila inferior: si multiplicamos la fila inferior por la fila a_{11} y, a continuación, la dividimos por a_{n1} y le restamos la primera fila, habremos conseguido que el primer elemento sea igual a cero. Hacemos esto para todas las filas y las columnas hasta conseguir la matriz triangular. El código es el siguiente.

```
private static void forward_solve(Matrix q)
{
    int i, j, k, maxr, n;
    double t, pivot;
    n= q.getNumRows();

    for (i=0; i < n; i++) { // Busca la fila con el elemento máx.
        maxr= i; // de la columna, en la diagonal o por debajo
        for (j= i+1; j < n; j++)
            if (Math.abs(q.getElement(j,i)) > Math.abs(q.getElement(maxr,i)))
                maxr= j;
        if (maxr != i) // Si la fila no es la actual, saltar
            for (k=i; k <= n; k++)
                { t= q.getElement(i,k); // t= q(i,k)
                  q.setElement(i,k, q.getElement(maxr, k)); // q(i,k)= q(maxr, k)
                  q.setElement(maxr, k, t); // q(maxr, k)= t
                }
        for (j= i+1; j < n; j++) // Calcula la relación de giro
            { pivot= q.getElement(j,i)/q.getElement(i,i); // q(j,i)/q(i,i)
              for (k= n; k >=i; k--)
                  q.setElement(j, k, q.getElement(j,k)-q.getElement(i,k)*pivot);
                  // q(j,k) -= q(i,k)*pivot; // Actualizar la fila j por debajo de la diag.
            }
    }
}
```

Sustitución inversa

Una vez conseguida la matriz triangular, la solución se obtiene mediante un proceso conocido como sustitución inversa. Comenzamos por la fila situada más abajo. Dado que la matriz es triangular, sólo tendremos un elemento distinto de cero en esta fila. Así, el valor de x_n se puede obtener fácilmente, x_n se puede calcular como b'_n/a'_{nn} y $x_{n-1} = (b'_{n-1} - a'_{n-1,n} * x_n) / a'_{n-1,n-1}$. Repetimos el proceso hasta x_1 . El código para este proceso de sustitución inversa es:

```
private static void back_solve(Matrix q) // Función de sust. inversa
{ // comienza en la fila n-1
    int j, k, n;
    double t; // t- temporal
    n= q.getNumRows();
```

```

for (j=n-1; j >=0; j--) // Comienza en la última fila
{
    t= 0.0;
    for (k= j+1; k < n; k++) // t += q(j,k)* q(k,n)
        t += q.getElement(j,k)* q.getElement(k,n);
    q.setElement(j, n, (q.getElement(j, n) -t)/q.getElement(j,j));
    // q(j, n)= (q(j, n) -t)/q(j,j);
}
}

```

Implementación de giro parcial de la eliminación gaussiana

Esta implementación aún todo lo que hemos estado diciendo hasta ahora:

```

public static void gaussian(Matrix a, Matrix b, Matrix x) {
    int i, j, n;
    n= a.getNumRows(); // Número de incógnitas
    Matrix q= new Matrix(n, n+1);
    for (i=0; i < n; i++) {
        for (j=0; j < n; j++) // Formar la matriz q
            q.setElement(i, j, a.getElement(i, j)); // q(i,j)= a(i,j)
        q.setElement(i, n, b.getElement(i, 0)); // q(i,n)= b(i,0)
    }
    forward_solve(q); // Realizar eliminación gaussiana
    back_solve(q); // Realizar sustitución inversa

    for (i=0; i<n; i++)
        x.setElement(i, 0, q.getElement(i, n)); // x(i,0)= q(i,n)
}

```

Podríamos hacer que este método forme parte de la clase Matrix.