

1.00 Lecture 21

October 28, 2005

Matrices

Matrices

- Matrix is 2-D array of m rows by n columns

a_{00}	a_{01}	a_{02}	$a_{03} \dots$	a_{0n}
a_{10}	a_{11}	a_{12}	$a_{13} \dots$	a_{1n}
a_{20}	a_{21}	a_{22}	$a_{23} \dots$	a_{2n}
\dots	\dots	\dots	$\dots \dots$	\dots
a_{m0}	a_{m1}	a_{m2}	$a_{m3} \dots$	a_{mn}

- In math notation, we use index 1, ... m and 1, ... n.
- In Java, we usually use index 0, ... m-1 and 0, ...n-1

Matrices and Linear Systems

Matrices often used to represent a set of linear equations:

$$\begin{aligned} a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + \dots + a_{0,n-1}x_{n-1} &= b_0 \\ a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + \dots + a_{1,n-1}x_{n-1} &= b_1 \\ \dots & \\ a_{m-1,0}x_0 + a_{m-1,1}x_1 + a_{m-1,2}x_2 + \dots + a_{m-1,n-1}x_{n-1} &= b_{m-1} \end{aligned}$$

- n unknowns x are related by m equations
- Coefficients a are known, as are right hand side b

Matrix representation

$$\begin{array}{c} \left| \begin{array}{ccccc} a_{00} & a_{01} & a_{02} & a_{03} \dots & a_{0,n-1} \\ a_{10} & a_{11} & a_{12} & a_{13} \dots & a_{1,n-1} \\ a_{20} & a_{21} & a_{22} & a_{23} \dots & a_{2,n-1} \\ \dots & \dots & \dots & \dots \dots & \dots \\ a_{m-1,0} & a_{m-1,1} & a_{m-1,2} & a_{m-1,3} \dots & a_{m-1,n-1} \end{array} \right| \begin{array}{c} \left| \begin{array}{c} x_0 \\ x_1 \\ x_2 \\ \dots \\ x_{n-1} \end{array} \right| = \begin{array}{c} \left| \begin{array}{c} b_0 \\ b_1 \\ b_2 \\ \dots \\ b_{m-1} \end{array} \right| \end{array} \end{array}$$

(m rows x n cols)

(n x 1) = (m x 1)

$$Ax=b$$

Matrices, p.2

- If $n=m$, we will try to solve for unique set of x . **Obstacles:**
 - If any row (equation) or column (variables) is linear combination of others, matrix is degenerate or not of full rank. No solution.
 - If rows or columns are nearly linear combinations, roundoff errors can make them linearly dependent during computations. We'll fail to find a solution, even though one may exist.
 - Roundoff errors can accumulate rapidly. While you may get a solution, when you substitute it into your equation system, you'll find it's not a solution.
- Linear systems tend to be close to singular (degenerate). Beware!
 - We'll do solutions for linear systems next lecture

Matrices, p.3

- In this lecture we cover basic matrix representation and manipulation
 - Used most often to prepare matrices for use in solving linear systems
- Java has 2-D arrays, declared as, for example

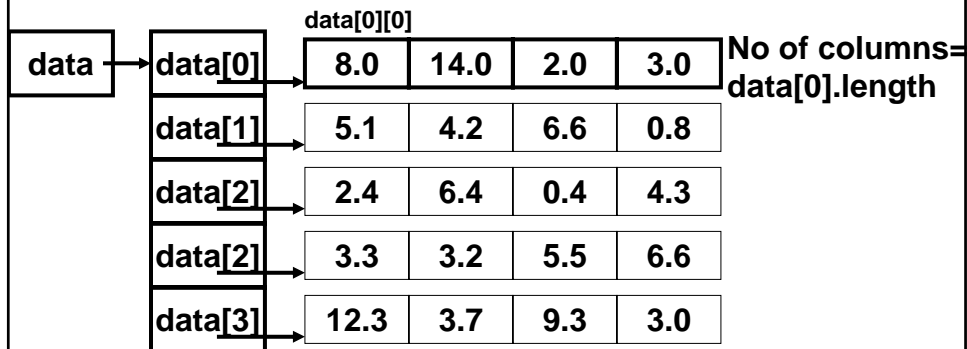
```
double[ ][ ] squareMatrix= new double[5][5];
```

 - But there are no built-in methods for them

Matrices, p.4

- So, it's helpful to create a **Matrix class**:
 - Create methods to add, subtract, multiply, form identity matrix, etc.
- **Sparse matrices are handled differently**:
 - Almost all large matrices are extremely sparse (99%+ of entries are zeros)
 - Store (i, j, value) in a list or 1-D array

2-D Arrays



No. of rows =
data.length

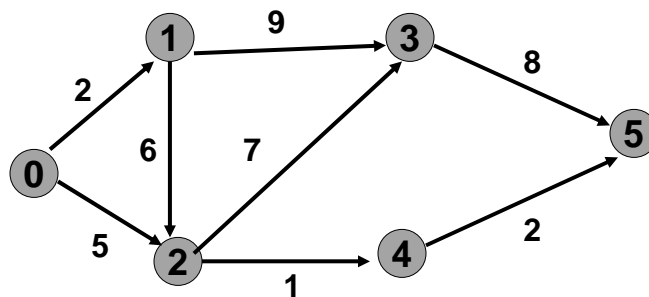
A 2-D array is:

a reference to a 1-D array of references to 1-D arrays of data.
This is how we'll store the matrix data in class Matrix

Matrix class, p.1

```
public class Matrix {  
    private double[][] data;    // Reference to array  
  
    // Constructor for Matrix  
    public Matrix(int m, int n) {  
        data = new double[m][n];  
    }  
}
```

Example of Sparse Matrix



Arc Cost Matrix

0	2	5	0	0	0
0	0	6	9	0	0
0	0	0	7	1	0
0	0	0	0	0	8
0	0	0	0	0	2
0	0	0	0	0	0

36 entries
Only 8 not zero

Matrix class, p.2

```
// Method to set identity Matrix
public void setIdentity() {
    int i, j;
    int nRows = data.length;
    int nCols = data[0].length;
    for(i=0; i<nRows; i++)
        for(j=0; j<nCols; j++)
            if(i == j)
                data[i][j] = 1.0;
            else
                data[i][j] = 0.0;
}
```

Matrix class, p.3

```
// Return no. rows in Matrix
public int getNumRows() {
    return data.length;
}
// Return no. columns in Matrix
public int getNumCols() {
    return data[0].length;
}
//Get element i,j
public double getElement(int i, int j) {
    return data[i][j];
}
// Set element (i,j)
public void setElement(int i, int j, double val) {
    data[i][j] = val;
}
```

Matrix class, p.4

```
public Matrix addMatrices(Matrix b) {
    Matrix result = null;
    int nrows = data.length;
    int ncols = data[0].length;
    if (nrows==b.data.length && ncols==b.data[0].length)
    {
        result = new Matrix(nrows, 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;
}
```

Matrix class, p.5

```
public Matrix mul tMatrices(Matrix b) {
    Matrix result = null;
    int nrows = data.length;
    int p = data[0].length;
    if (p == b.data.length) {
        result = new Matrix(nrows, b.data[0].length);
        for(int i=0; i<nrows; i++)
            for(int j=0; j<result.data[0].length; j++){
                double t = 0.0;
                for(int k=0; k<p; k++) {
                    t += data[i][k] * b.data[k][j];
                }
                result.data[i][j]= t;
            }
    }
    return result;
}
```

Matrix class, p.6

```
public void print() {
    for(int i=0; i< data.length; i++) {
        for(int j=0; j< data[0].length; j++)
            System.out.print(data[i][j] + " ");
        System.out.println();
    }
    System.out.println();
} // end of print() method
} // end of Matrix class
```

Adding a new Matrix method

Suppose we wanted to add a method that would multiply a matrix by a scalar quantity.

For example, suppose we want to multiply every matrix element by 3.0.

Different ways we could do this. Three (of many) options are:

- **A** - create an instance method of the Matrix class that takes a double and multiplies every element of the instance by that value.
- **B** - create a method that takes a reference to a Matrix and a double, and returns a reference to an entirely new matrix object.
- **C** - create an instance method that takes a double and returns a new matrix that is a scaled version of the instance

Option A

```
public void scalarMult(double scalar)
{
    for(int i=0; i<data.length; i++)
        for(int j=0; j<data[0].length; j++)
            data[i][j] *= scalar;
}
```

Option B

```
public static Matrix scalarMult(Matrix a, double scalar)
{
    Matrix result = new Matrix(a.getNumRows(),
                               a.getNumCols());
    for(int i=0; i<a.getNumRows(); i++)
        for(int j=0; j<a.getNumCols(); j++)
            result.setElement(i, j, a.getElement(i, j) *
                               scalar);
    return result;
}
```

Option C

```
public Matrix scalarMult(double scalar)
{
    Matrix result = new Matrix(getNumRows(),
                               getNumCols());
    for(int i=0; i<data.length; i++)
        for(int j=0; j<data[0].length; j++)
            result.data[i][j] = data[i][j]*scalar;
    return result;
}
```

Diagonal Matrix Class

- Only diagonal elements are nonzero:

$$\begin{array}{cccccc} a_{00} & 0 & 0 & 0 & \dots & 0 \\ 0 & a_{11} & 0 & 0 & \dots & 0 \\ 0 & 0 & a_{22} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & a_{m-1,m-1} \end{array}$$

- We can store this as a 1-D matrix of m elements
- We can implement the same matrix methods as our main Matrix class

DiagonalMatrix class, p.1

```
public class DiagonalMatrix { // Does not extend Matrix
    private double[] data; // Replaces data representation

    public DiagonalMatrix(int nr) { // Constructor
        data = new double[nr]; }
    public int getNumRows() {
        return data.length; }
    public int getNumCols() {
        return data.length; }
    public double getElement(int i, int j) {
        if(i == j)
            return data[i];
        else
            return 0.0; }
    public void setElement(int i, int j, double val) {
        if(i == j)
            data[i] = val; }
}
```

DiagonalMatrix class, p.2

```
public Matrix multMatrices(Matrix b)    {
    Matrix result = null;                // Return regular Matrix
    int nrows = data.length;            // Multiply Diag by reg Matrix
    int p = data.length;
    if(p == b.getNumRows()) {
        result = new Matrix(nrows, b.getNumCols());
        for(int i=0; i<nrows; i++)
            for(int j=0; j<result.getNumCols(); j++)
                result.setElement(i, j, b.getElement(i, j)*data[i] );}
    return result;    }
```

DiagonalMatrix class, p.3

```
public void print() {
    for(int i=0; i< data.length; i++) {
        for(int j=0; j< data.length; j++)
            if (i==j)
                System.out.print(data[i] + " ");
            else
                System.out.print("0.0 ");
        System.out.println();    }
    System.out.println();    }
```

MatrixMain, p.1

```
public class MatrixMain {
    public static void main(String[] args) {
        Matrix mat1 = new Matrix(3,3); // Regular Matrices
        Matrix mat2 = new Matrix(3,3);
        Matrix res; // Result
        mat1.setIdentity(); mat2.setIdentity();
        res = mat1.addMatrices(mat2); // Add regular
        System.out.println("mat1:"); mat1.print();
        System.out.println("mat2:"); mat2.print();
        System.out.println("mat1 + mat2: "); res.print();

        Matrix mat3= new Matrix(4, 2); // Regular Matrices
        Matrix mat4= new Matrix(2, 3);
        Matrix res2; // Set values below
        for (int i=0; i < mat3.getNumRows(); i++)
            for (int j= 0; j < mat3.getNumCols(); j++)
                mat3.setElement(i, j, i*j);
        for (int i=0; i < mat4.getNumRows(); i++)
            for (int j= 0; j < mat4.getNumCols(); j++)
                mat4.setElement(i, j, 2*(i+j));
        res2= mat3.mul tMatrices(mat4); // Mul ti ply regul ar
```

MatrixMain, p.2

```
        System.out.println("mat3:"); // Output regul ar
        mat3.print();
        System.out.println("mat4:");
        mat4.print();
        System.out.println("mat3 * mat4: " );
        res2.print();
        // Diagonal matrix
        Diagonal Matrix mat5= new Diagonal Matrix(4);
        for (int i= 0; i < mat5.getNumRows(); i++)
            mat5.setElement(i, i, i+1);
        Matrix res3;
        res3= mat5.mul tMatrices(res2); // Mul t by regul ar
        System.out.println("mat5:");
        mat5.print(); // Output diagonal
        System.out.println("res2:");
        res2.print(); // Output regul ar
        System.out.println("mat5 * res2: " );
        res3.print();
    }
}
```

Matrix Program Design Issues

- **Option 1: The approach used in these notes, based on multiple matrix classes (Matrix, DiagonalMatrix, etc.):**
 - **Disadvantage: We must write methods for each class to take arguments of the others, if we want maximum flexibility. In our brief example:**
 - DiagonalMatrix has method to multiply by Matrix
 - Matrix does not have method to multiply by DiagonalMatrix
 - We would have to write these (and other methods) to have a complete class
 - Also we would need methods to multiply vectors by matrices, matrices by vectors, etc.
 - **Advantages of this approach:**
 - **Efficiency: We access the private data of each class directly, without accessor methods in our class methods (e.g., data.length instead of getNumRows())**
 - **Pattern: Typical of how C++ would implement this, which is the principal language and style used for numerical methods**

Matrix Design Issues, p.2

- **Option 2: Inheritance could be used here:**
 - DiagonalMatrix could extend Matrix
 - Matrix would have to use accessor methods in its class methods; it couldn't access its private data directly
 - Every Matrix method would have to be overridden in DiagonalMatrix or it would not work properly
 - **Advantage:**
 - Fewer methods must be written. E.g., Matrix needs just one multMatrix method, since DiagonalMatrix is a Matrix
 - **Disadvantages:**
 - With each subclass having a different internal representation of the matrix data (1-D arrays, 2-D arrays, Vector) from the superclass, the superclass private data would be hidden but present in each subclass.
 - **Efficiency: Much more method call overhead, and unused superclass data in subclasses**
 - Dr. J Harward (Java): “Elegant, a little sly”
 - Dr. G Kocur (C++): “Awkward, a misuse of inheritance”

Matrix Design Issues, p.3

- **Option 3: A Matrix interface could be defined:**
 - Each class would implement these methods in the best way for its private data (like option 1)
 - Fewer methods would be required than in the approach in these notes (option 1), since all arguments would be Matrix types
 - Method overhead would be large (like option 2), since class methods could not access an argument's private data directly (they wouldn't know whether they were operating on a Matrix, DiagonalMatrix, etc.)
 - Each class could access its own private data directly, though
 - There wouldn't be spurious superclass private data in subclasses (like option 2)
 - Probably a cleaner implementation than option 2, but it doesn't seem totally natural for Matrix to be an interface

What Do You Think?

- Jot down questions you still have, and discuss them with your neighbors
- Jot down your preference, and why
- We'll take a brief poll at the end