# 3 – Write programs to solve problems involving simultaneous linear equations

# 5 – Write programs to use matrix arithmetic to work with graphic objects

## Matrices

A **matrix** is a two-dimensional rectangular array of numbers, denoted by *Ai*,*j*, such as the following:

is a 3×2 matrix (3 rows by 2 columns)

is a 2×2 square matrix

In general, an *m*×*n* matrix is of the form:

a1,1 a1.2 … (for *n* columns)

a2,1 a2,2 …

… … …

(for *m* rows)

*Note:* Indices start at 1 rather than 0 for arrays in programming languages like Java, C#, or JavaScript.

Matrices are often indicated using capital letters such as *A*, *B*, etc., while a particular entry in a matrix can be referenced using lowercase letters with a subscript in the form *ai*,*j* (for example, *a*3,2 refers to the element in row 3, column 2).

Some uses for matrices (we will apply matrices, not just talk about them in theory):

* Solving systems of equations
* Calculating least-squares polynomials
* Graphics
* Calculus
* Graphs (data structures)

## Matrix math

See also **“Matrix math notes MATH282-CST.docx”**.

### Addition

To add two matrices, add corresponding elements of the two operands.

Restrictions: operands must be of the same size/dimensions (but they do not need to be square)

Example:

Note that in this example, we start with a 2×2 matrix added to another 2×2 matrix, so our result is also a 2×2 matrix. (In general, the result will be of the same size/dimensions as the operands.)

Also note that the order of the operands does not matter for addition. *A* + *B* = *B* + *A* because it doesn’t matter which way we add corresponding elements.

### Subtraction

To subtract two matrices, subtract corresponding elements of the two operands.

Restrictions: same as for addition

Example:

Note that the result for subtraction is also the same size as the operands.

Also note that the order of the operands **does** matter because *A* - *B* ≠ *B* - *A*. For instance, if we call the two matrices above *A* and *B*, then *B* - *A* is:

### Scalar multiplication

Scalar multiplication means multiplying a matrix by a scalar (dimensionless single) value. To perform scalar multiplication, we multiply each element by the scalar value. There are no restrictions on the matrix size.

Example:

Example: , so -1\*(*A* - *B*) = (*B* - *A*)

### Matrix multiplication

In matrix multiplication, we identify the row in the left operand and the column in the right operand and use them. Take corresponding elements in that row and that column and multiply them, and then add up those products to get the value in the resultant matrix.

For square matrices, a 2×2 matrix multiplied by a 2×2 matrix results in a 2×2 matrix; we will look at rectangular matrices as we go along.

Matrix multiplication is often easier explained by doing an example:

Example:

If we call the two matrices above *A* and *B*, then we can also try to calculate *B*\**A*:

Example:

Note that *A*\**B* ≠ *B*\**A* so matrix multiplication is **not** commutative (order of operands is important). We will see more examples of this later.

A neat way of doing matrix multiplication is to put the second operand above and to the right of the first operand in a triangle, and then calculate the results where the rows of the first operand and the columns of the second operand intersect. See **“Matrix multiply visual MATH282-CST.docx”** for more details and an example.

Matrix multiplication is learned best by practice, so try the following:

Practice #1: Try

Practice #2: Try

Practice #3: Try

Practice #4: Try

Practice #5: For *G* =  and *H* = , try *G*\**H* and *H*\**G*

See **“Matrix Multiplication Practice.docx”** for more practice.

Results of practice #1:

*Note that we can multiply matrices that are not square.*

Results of practice #2:

*Note that C\*D is not only not equal to D\*C, but that the results are two completely different sizes.*

Results of practice #3:

cannot be done

*Note that we cannot always multiply two matrices.*

There are 3 entries in row 1 of *D* and only 2 entries in column 1 of *E*, so we cannot get the corresponding elements to multiply.

Results of practice #4:

*Note that the result is the same as matrix F because we are multiplying against a lot of zeros and ones.*

For matrices, a square matrix with 1’s from the upper left to the lower right diagonal, and all other elements set to 0, is called the multiplicative identity matrix. When any matrix is multiplied by the identity matrix of the correct size, the result is the original matrix.

This is similar to multiplying by 1 in the real number system, where 1*x* = *x*.

Results of practice #5:

*G*\**H =* \*=

*Note that these matrices are also used in* ***“Matrix multiply visual MATH282-CST.docx”*** *– finding the right answers is left for you as an exercise.*

*H*\**G* = \* cannot be done

*Note that we cannot always multiply two matrices, even if they can be multiplied in the reverse order!*

There are 4 entries in row 1 of *H* and only 2 entries in column 1 of *G*, so we cannot get the corresponding elements to multiply.

From these examples, we can draw the following generalization: For matrices of size *m*1×*n*1 and *m*2×*n*2, the matrices can only be multiplied if *n*1 = *m*2 and the result will be of size *m*1×*n*2.

### Matrix division

Division is more complex still (we must find the inverse matrix and multiply by it), so we will not look at it for now. Just be aware that just like solving 2*x* = 7 involves dividing by 2, or in other words multiplying both sides by the inverse of 2 = ½ (so ½\*2*x* = ½\*7, resulting in 1*x* = 7/2 or *x* = 7/2 since 1 is the multiplicative identity), we can find the inverse of a matrix and multiply it against other matrices.

## Coding a matrix class

We will implement matrices in code using C# (although Java could be used as well).

First, we will create an interface in a class called IMatrix. The interface describes what must be implemented in a derived class, but cannot itself be instantiated or implement methods or have members. So all we will include are the signatures for the matrix methods that we want to implement for addition, subtraction, scalar multiplication, and matrix multiplication:

public interface IMatrix

{

IMatrix Add(IMatrix mRight);

IMatrix Multiply(IMatrix mRight);

IMatrix ScalarMultiply(double dScalar);

IMatrix Subtract(IMatrix mRight);

}

Then, we will create an abstract class called AMatrix, which will implement the IMatrix interface but will still leave the exact details about how the matrix is stored to a further (concrete) class:

public abstract class AMatrix : IMatrix

The abstract class will include attributes for the number of rows and the number of columns in the matrix, as well as a way to get or set those values:

#region Attributes

private int iRows;

private int iCols;

public int Rows

{

get

{

return iRows;

}

set

{

if (value < 1)

{

throw new Exception("Rows must be 1 or greater");

}

iRows = value;

}

}

public int Cols

{

get

{

return iCols;

}

set

{

if (value < 1)

{

throw new Exception("Columns must be 1 or greater");

}

iCols = value;

}

}

#endregion

It will also include some abstract methods to actually get or set elements within the matrix, or create a new matrix with a given number of rows and columns – implementation of these methods will be put off until we implement AMatrix in a further (concrete) class that will provide the details of how the matrix will be stored:

#region Abstract Methods

public abstract double GetElement(int iRow, int iCol);

public abstract void SetElement(int iRow, int iCol, double dValue);

internal abstract AMatrix NewMatrix(int iRows, int iCols); // matrix full of 0's

#endregion

We will also need the math methods that we specified in the interface – these methods should apply regardless of how the matrix is stored, so they will appear in the abstract class. We will leave them unimplemented for now:

#region Math methods

public IMatrix Add(IMatrix mRight)

throw new NotImplementedException();

}

public IMatrix Multiply(IMatrix mRight)

{

throw new NotImplementedException();

}

public IMatrix ScalarMultiply(double dValue)

{

throw new NotImplementedException();

}

public IMatrix Subtract(IMatrix mRight)

{

throw new NotImplementedException();

}

#endregion

And we also need a ToString() method so that we can display the matrices as we go through them. A simple ToString method might look like the following. Note how we use the Rows and Cols (with <= since matrices start at 1!) to iterate through the matrix elements, and we access the matrix elements using GetElement. Also note where we add tabs (after each element) and newlines (after each row).

public override string ToString()

{

StringBuilder s = new StringBuilder();

for (int r = 1; r <= this.Rows; r++)

{

for (int c = 1; c <= this.Cols; c++)

{

s.Append(this.GetElement(r, c) + "\t");

}

s.Append("\n");

}

return s.ToString();

}

Finally we need to decide how to store the matrix values. A number of ways are possible. For instance, for sparse matrices (matrices where most of the entries are zero), we might want to store only the values that are non-zero. Or for some other matrices, we might have some equation that generates matrix entries. But the simplest method of storing the matrix values is to use a two-dimensional array and map those values to the matrix values (remembering that arrays are zero-based, while matrices are one-based).

First, we implement the abstract class:

public class Matrix : AMatrix

Then, we set up an attribute to store the two-dimensional array:

#region Attributes

// set up a 2D array

private double[,] dArray;

#endregion

Next, we create the constructors, which will set the values for our attributes:

#region Constructors

public Matrix(double[,] dArray)

{

// set the row and col properties

// Note how we obtain dimensions of arrays in C#

this.Rows = dArray.GetLength(0); // rows

this.Cols = dArray.GetLength(1); // columns

this.dArray = dArray; // shallow copy - update to deep copy later

}

// copy constructor - using constructor chaining to call the above constructor

public Matrix(Matrix m) : this(m.dArray)

{

}

#endregion

Finally, we implement the abstract methods from AMatrix – note how we use iRow - 1 and iCol - 1 to switch between the array representation (zero-based) and the matrix representation (one-based).

#region Concrete methods

public override double GetElement(int iRow, int iCol)

{

return dArray[iRow - 1, iCol - 1];

}

public override void SetElement(int iRow, int iCol, double dValue)

{

dArray[iRow - 1, iCol - 1] = dValue;

}

internal override AMatrix NewMatrix(int iRows, int iCols)

{

// create a new matrix with all values set to 0.0

return new Matrix(new double[iRows, iCols]);

}

#endregion

At this point, we have enough implemented to be able to create and display a matrix in our Main method:

double[,] dArray = { { 1, 2, 3 }, { 4, 5, 6 } };

IMatrix mTest = new Matrix(dArray);

Console.WriteLine(mTest);

This code results in the following output:

1 2 3

4 5 6

We can also demonstrate that our array copy is only a shallow copy and test the copy constructor as follows:

dArray[0, 0] = 0.0; // won't affect mTest if we do a deep copy of the array

IMatrix mCopy = new Matrix((Matrix)mTest);

Console.WriteLine(mCopy);

This code results in the following output – note how changing the array has changed the matrix (and thus its copy), which shouldn’t happen if we make a deep copy of the array:

0 2 3

4 5 6

We should thus update our constructor to make a deep copy of the array with the following code:

// Set the array property

//this.dArray = dArray; // creates a shallow copy

// Do a deep copy

this.dArray = new double[this.Rows, this.Cols];

for (int i = 0; i < this.Rows; i++)

{

for (int j = 0; j < this.Cols; j++)

{

this.dArray[i, j] = dArray[i, j];

}

}

This change results in the following output:

1 2 3

4 5 6

1 2 3

4 5 6

We could also make our output of the matrices fancier – see **“fancy ToString.txt”** for an example of a ToString method that adds borders around the output matrix.

With the ability to create and display matrices completed, we can now work on implementing our math methods.

### Add method

As you may recall, adding two matrices is only possible if the matrices are the same size, so that is something that we should test in our code. The sum matrix will be the same size as the operand matrices, and its elements will be found by adding the corresponding elements in the operands.

An algorithm for this process (from **“Matrix math notes MATH282-CST.docx”**) is as follows, where *S* denotes the sum matrix and *A* and *B* denote the left and right operands:

For each row *i* in **S**

For each column *j* in **S**

The current entry **s***ij* = **a***ij* + **b***ij*

This leads to the following code – note that for clarity, it uses LeftOp for the invoking matrix and RightOp for the parameter, and it breaks the calculation of and storage of the element in Sum into two lines:

public IMatrix Add(IMatrix mRight)

{

AMatrix LeftOp = this;

AMatrix RightOp = (AMatrix)mRight;

AMatrix Sum = null;

// if operands are the same size

if (LeftOp.Rows == RightOp.Rows && LeftOp.Cols == RightOp.Cols)

{

// create new matrix of the same size as the operands

Sum = NewMatrix(LeftOp.Rows, LeftOp.Cols);

// go through each row and column, adding corresponding elements

for (int r = 1; r <= LeftOp.Rows; r++)

{

for (int c = 1; c <= LeftOp.Cols; c++)

{

double dVal = LeftOp.GetElement(r, c) + RightOp.GetElement(r, c);

Sum.SetElement(r, c, dVal);

}

}

}

else

{

throw new Exception("Operands must be the same size");

}

return Sum;

}

Add the following test code in Main:

IMatrix mSum = mTest.Add(mTest);

Console.WriteLine(mSum);

The results will be:

┌ ┐

│ 2 4 6 │

│ 8 10 12 │

└ ┘

Note that the Add method creates and returns a new matrix, so mTest will not be affected by the addition.

Further tests should be done to ensure that we can add different matrices, and that we cannot add matrices of differing sizes – see **“Basic Matrix Test Code.txt”** for tests for all of the matrix operations.

If you haven’t been able to get your code working at this point, see the **Matrix2019** folder, which has all the code up to this point except for the deep copy of the array in the constructor and the fancy ToString method.

### Scalar multiply method

Creating the code for scalar multiply is left as an exercise for you. Note that there are no restrictions on the matrix size for scalar multiplication.

### Subtraction method

Creating the code for subtraction is left as an exercise for you. Can you find a way to use existing code to make subtraction quicker to implement?

### Multiplication method

Creating the code for matrix multiplication is left as an exercise for you. See the algorithms and discussion of matrix multiplication in **“Matrix Multiplication Algorithm.docx”** and **“Matrix math notes MATH282-CST.docx”**.