groupproject

Connor Baker
Rae Bouldin

November 28, 2016
Version 0.4a

Contents

1 Summary of Problem Specification

1.1 Abstract

Write a program that reads two 3×3 matrices from file and computes the sum and product of the two matrices. Then, find the transpose, cofactor matrix, and determinant of the two resultant matrices. Then, find the inverse of the first matrix, and multiply it by the the first column of the second matrix. Finally, compute the standard deviation of the diagonal elements of the two inputted matrices. All input and output should be stored in files.

2 Formulae

2.1 Determinant

The determinant of a 3×3 matrix is most readily computed by row reducing to a triangular matrix, and taking the product of the main diagonal. However, failing that, one can calculate the determinant by doing cofactor expansion. Though a horribly inefficient algorithm for larger matrices, it gets the job done. For a 3×3 matrix A, it's determinant, $\det(A)$, can be computed using placeholder values as follows:

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

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

$$\det(A) = a(ei - fh) - b(di - fg) + c(dh - eg) \tag{1}$$

As such, we can calculate the determinant of a 3×3 matrix by expanding across the top row. This yields the approach implemented in `determinantOfMatrix()`, detailed in Section 4.3.

2.2 Transpose

Using the same approach as above, we can easily compute the transpose of a matrix. It involves creating a matrix filled with placeholder values, calculating the transpose matrix by hand, and tracking the positions of the elements in the matrix. For a 3×3 matrix A, let its transpose be A^T. Then:

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

$$A^T = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \tag{2}$$

This yields the approach implemented in `transposeOfMatrix()`, detailed in Section 4.4.

2.3 Matrix Addition

The approach used to create an algorithm for matrix addition is similar to that used above in finding the determinant. We again create matrices full of placeholder values, and track them as we perform the operation. For two 3×3 matrices A and B, the process is as follows:

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \qquad B = \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix}$$

$$A + B = \begin{bmatrix} a+j & b+k & c+l \\ d+m & e+n & f+o \\ g+p & h+q & i+r \end{bmatrix} \tag{3}$$

This yields the approach implemented in `sumOfMatrices()`, detailed in Section 4.5.

2.4 Matrix Multiplication

The approach used to create an algorithm for matrix multiplication is similar to that used above in finding the determinant. We again create matrices full of placeholder values, and track them as we perform the operation. For two 3×3 matrices A and B, the process is as follows:

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \qquad B = \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix}$$

$$AB = \begin{bmatrix} aj+bm+cp & ak+bn+cq & al+bo+cr \\ dj+em+fp & dk+en+fq & dl+eo+fr \\ gj+hm+ip & gk+hn+iq & gl+ho+ir \end{bmatrix} \tag{4}$$

This yields the approach implemented in `productOfMatrices()`, detailed in Section 4.6.

2.5 Cofactor Matrix

The approach used to calculate the cofactor matrix is identical to what was done above. It involves creating a matrix filled with placeholder values, calculating the cofactor matrix by hand, and tracking the positions of the elements

in the matrix. For a 3×3 matrix C, let its cofactor matrix be C'. Then:

$$C = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

$$C' = \begin{bmatrix} ei - fh & fg - di & dh - eg \\ ch - bi & ai - cg & bg - ah \\ bf - ce & cd - af & ae - bd \end{bmatrix} \tag{5}$$

This yields the approach implemented in `cofactorOfMatrix()`, detailed in Section 4.7.

2.6 Inverse of a Matrix

The inverse of a matrix A, denoted A^{-1} can be calculated as follows:

$$A^{-1} = \frac{1}{\det(A)} * A^T \tag{6}$$

Which is very doable using Equations (1) and (2). This approach is used in `inverseOfMatrix()`, detailed in Section 4.8.

3 Main.class

4 Matrix.class

4.1 Matrix()

```
public Matrix() throws IOException
public Matrix(String filename) throws IOException
```

`Matrix()` is the constructor. It is overloaded. The no-arg constructor reads from the default file of `matrix1.txt`. The argumented constructor, `Matrix(String filename)` accepts a file name in the form of a string.

Every `Matrix` object has four matrices (three two-dimensional arrays of type `int`, and one of type `double`, being the matrix read from file by `readMatrixFromFile()`, the transpose, the cofactor, and inverse matrices respectively). They also have an `int` that holds the determinant.

Both the no-arg and argumented `Matrix()` constructor call `readMatrixFromFile()`, the only difference being that the argumented constructor passes in the location of the file to read the matrix from.

4.2 readMatrixFromFile()

```
public void readMatrixFromFile() throws IOException
```

The method `readMatrixFromFile()` reads a two-dimensional array of integers from file. This method does not accept arguments. This method is called from the `Matrix()` constructor described above. The file to read from is passed in via the argumented `Matrix()` constructor.

4.3 determinantOfMatrix()

```
public static void determinantOfMatrix(int a[][])
public static void determinantOfMatrix(int a[][], String
↪  filename) throws IOException
```

Using Equation (1) from Section 2.1, this method calculates the determinant of a matrix. textttdeterminantOfMatrix() is overloaded. Both methods accept a two-dimensional array of `int` as input, with the second of the two accepting a `String` holding the filename and or path to write the output matrix to. This is accomplished by calling the `print()` method.

4.4 transposeOfMatrix()

```
public static void transposeOfMatrix(int a[][])
public static void transposeOfMatrix(int a[][], String filename)
↪ throws IOException
```

Using Equation (2) from Section 2.2, this method calculates the transpose of a matrix. texttttransposeOfMatrix() is overloaded. Both methods accept a two-dimensional array of `int` as input, with the second of the two accepting a `String` holding the filename and or path to write the output matrix to. This is accomplished by calling the `print()` method.

4.5 sumOfMatrices()

```
public static void sumOfMatrices(int a[][], int b[][], String
↪ filename) throws IOException
```

Using Equation (3) from Section 2.3, this method calculates the sum of two matrices. The sum is written to a file passed in through the method signature. This is accomplished by calling the `print()` method.

4.6 productofMatrices()

```
public static void productOfMatrices(int a[][], int b[][],
↪ String filename) throws IOException
```

Using Equation (4) from Section 2.4, this method calculates the product of two matrices. The sum is written to a file passed in through the method signature. This is accomplished by calling the `print()` method.

4.7 cofactorOfMatrix()

```
public static void cofactorOfMatrix(int a[][])
public static void cofactorOfMatrix(int a[][], String filename)
↪ throws IOException
```

Using Equation (5) from Section 2.5, this method calculates the cofactor matrix of a matrix. textttcofactorOfMatrix() is overloaded. Both methods accept a two-dimensional array of `int` as input, with the second of the two accepting a `String` holding the filename and or path to write the output matrix to. This is accomplished by calling the `print()` method.

4.8 inverseOfMatrix()

```
public static void inverseOfMatrix() throws IOException
public static void inverseOfMatrix(String filename) throws
↪ IOException
```

Using Equation (1) from Section 2.1, this method calculates the determinant of a matrix. textttdeterminantOfMatrix() is overloaded. Both methods accept a two-dimensional array of `int` as input, with the second of the two accepting a `String` holding the filename and or path to write the output matrix to. This is accomplished by calling the `print()` method.

4.9 print()

```
public static void print(int a[][], String console) throws
↪ IOException
public static void print(int a[][], String console, String
↪ filename) throws IOException
public static void print(double a[][], String console) throws
↪ IOException
public static void print(double a[][], String console, String
↪ filename) throws IOException
```

The `print()` method is overloaded. It accepts two-dimensional arrays of either type `int` or `double`, and prints either to console, or console and file/a path passed in to the method.

The `print()` method's purpose is limited to printing to console the type of operation that was performed, and then calling and passing the arguments on to either (or both) `printMatrixToConsole()` and `printMatrixToFile()`, described in detail below.

4.10 printMatrixToConsole()

```
public static void printMatrixToConsole(int a[][])
public static void printMatrixToConsole(double a[][])
```

The `printMatrixToConsole()` method consists of a `for` loop and a `System.out.println` holding a row of our 3×3 matrix (the two-dimensional array). It uses the speedup trick described in Section 5.1.

4.11 printMatrixToFile()

```
public static void printMatrixToFile(int a[][], String filename)
 ↪  throws IOException
public static void printMatrixToFile(double a[][], String
 ↪  filename) throws IOException
```

The `printMatrixToFile()` method consists of output streams (a `FileWriter`, `BufferedWriter`, and a `PrintWriter`) `for` loop and a `System.out.println` holding a row of our 3×3 matrix (the two-dimensional array). It uses the speedup trick described in Section 5.1.

4.12 printIntegerToFile()

```
public static void printIntegerToFile(int a, String filename)
 ↪  throws IOException
```

The `printMatrixToFile()` method consists of output streams (a `FileWriter`, `BufferedWriter`, and a `PrintWriter`) and simply prints to file the integer passed in through the method signature.

5 Notes

5.1 A Note About the Methods

By tailoring our methods for the specific order of matrix that we operate on (3×3), we eliminate the need for method calls (specifically `matrix.length`), as well as the need for nested `for` loops. In this case, we should observe a speedup of 900% (as the compiler does not unroll loops, we are effectively doing one ninth of the iterations that we would normally do) in all methods that process arrays.

References

http://download.java.net/java/jdk9/docs/api/