MA117 Project 3: Determinants of Matrices

Administrative Details

- This project is the third of the three assignments required for the assessment in this course. It is to be submitted by **Noon, Monday 4th May 2020**. Details of the method of the submission via the Tabula system have been described in the lecture notes and are also available on the course web page.
- This assignment will count for **40%** of your total grade in the course.
- The automated submission system requires that you closely follow instructions about the format of certain files; failure to do so will result in the severe loss of points in this assessment.
- You may work on the assignment during the lab session, provided you have completed the other tasks that have been set. You can use the work areas at all times when they are not booked for teaching, 7 days per week. If you are working on the assignment on your home system you are advised to make regular back-up copies (for example by transferring the files to the University systems). You should note that **no** allowance will be made for domestic disasters involving your own computer system. You should make sure well ahead of the deadline that you are able to transfer all necessary files to the University system and that it works there as well.
- The Tabula system will be open for the submission of this assignment starting from **23rd April 2020**. You will not be able to test your code for correctness using Tabula but you can resubmit your work several times, until the deadline, if you find a mistake after your submission. A later submission always replaces the older one, but you have to re-submit **all** files.
- Remember that all work you submit should be **your own work**. Do not be tempted to copy work; this assignment is not meant to be a team exercise. There are both human and automated techniques to detect pieces of the code which have been copied from others. If you are stuck then ask for assistance in the lab sessions. TAs will not complete the exercise for you but they will help if you do not understand the problem, are confused by an error message, need advice on how to debug the code, require further explanation of a feature of Java or similar matters.
- If you have more general or administrative problems e-mail me immediately. Always include the course number (MA117) in the subject of your e-mail.

1 Formulation of the Problem

Matrices are one of the most important mathematical concepts to be modelled by computer, being used in many problems from solving simple linear systems to modelling complex partial differential equations.

Whilst a matrix (in our formulation) is simply an element of the vector space $\mathbb{R}^{m\times n}$, it usually possesses some structure which we can exploit to gain computational speed. For example, a matrix-matrix multiplication generally requires of the order of n^3 floating-point operations. If the matrix has some special structure which we can exploit using a clever method, then we might be able to reduce this to n operations. For large values of n, this significantly improves the performance of our code.

In this project, you will write two classes representing matrices of the form:

$$A = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \text{ and } B = \begin{bmatrix} b_{11} & b_{12} & 0 & 0 & 0 \\ b_{21} & b_{22} & b_{23} & 0 & 0 \\ 0 & b_{32} & b_{33} & b_{34} & 0 \\ 0 & 0 & b_{43} & b_{44} & b_{45} \\ 0 & 0 & 0 & b_{54} & b_{55} \end{bmatrix}$$

A is a dense $m \times n$ matrix which, in general, has no special structure and no zero entries. B is a *tri-diagonal* matrix, where all entries are zero apart from along the diagonal and upper and lower diagonals. Note that although B is only a 5×5 matrix, your classes should represent a general $n \times n$ tri-diagonal matrix. Also, the tri-diagonal matrices you need to represent will *always* be square.

In a similar fashion to `Fraction`, you will then write functions to perform various matrix operations:

1. addition and subtraction;
2. scalar and matrix-matrix multiplication;
3. calculating the determinant of the matrix.

Clearly calculating the determinant is the most tricky task here. Probably you will already have seen expansion by minors as a possible method. Whilst this is an excellent method for calculating determinants by hand, **you should not use it for this task**. The reason is that calculating the determinant of a $n \times n$ matrix requires $O(n!)$ operations, since for each $n \times n$ matrix, we must calculate the values of the $n-1$ sub-determinants. This is *extremely* slow.

A much better method is called *LU decomposition.* In this, we write a matrix A as product of two matrices L and U which are lower- and upper- triangular respectively. For example, for a 4×4 matrix, we would find matrices so that

$$\underbrace{\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}}_{A} = \underbrace{\begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}}_{U}$$

Such a factorisation is not guaranteed to exist (and indeed is not unique), but typically it does. In this project, you don't really need to worry about this – your code will be tested with matrices for which the LU decomposition exists. It is up to you to figure out how to calculate the determinant from the LU decomposition!

Throughout the formulation, matrices will be represented by indices running between $1 \leq i, j \leq m, n$. However, in your code, you should stay consistent with Java notation and indices should start at 0 (i.e. $0 \leq i, j \leq m-1, n-1$).

2 Programming Instructions

On the course web page for the project you will find files for the following classes. As with the previous projects, the files have some predefined methods that are either complete or come with predefined names and parameters. You **must** keep all names of `public` objects and methods as they are in the templates. Other methods have to be filled in and it is up to you to design them properly. There are five classes in this project:

- `Matrix`: a general class defining the basic properties and operations on matrices.

- `MatrixException`: a subclass of the `RuntimeException` class which you should use to throw matrix-related exceptions. This class is complete – you do not need to alter it.
- `GeneralMatrix`: a subclass of `Matrix` which describes a general $m \times n$ real matrix.
- `TriMatrix`: another subclass of `Matrix` which describes a $n \times n$ real tri-diagonal matrix.
- `Project3`: a completely separate class which will use `Matrix` and its subclasses to collect some basic statistics involving random matrices.

Please note that unlike other projects, you **may not** assume that the data you receive will be valid. Therefore you will need to check, amongst other things, that matrix multiplications are done using matrices of valid sizes, the user is not trying to access matrix elements which are out of bounds, etc. If something goes wrong, you are expected to throw a `MatrixException`.

The classes you need to work on are briefly described below.

2.1 The `Matrix` class

This is the base class from which you will build your specialised subclasses. `Matrix` is `abstract` – as described in the lectures, this means that some of the methods are not defined, and they need to be implemented in the subclasses. The general idea is that each subclass of `Matrix` can implement its own storage schemes, whilst still maintaining various common methods inherent in all matrices.

In particular, the following functions are **not** `abstract`, and need to be filled in inside `Matrix`:

- the `protected` constructor function;
- `toString`, which should return a `String` representation of the matrix.

Additionally, the following `abstract` methods will be implemented by the subclasses of `Matrix`:

- `getIJ` and `setIJ`: accessor and mutator methods to get/set the ijth entry of the matrix.
- `add`: returns a new `Matrix` containing the sum of the current matrix with another.
- `multiply(double a)`: multiply the matrix by a constant `a` $\in \mathbb{R}$.
- `multiply(Matrix B)`: multiply the matrix by another matrix. Note that this is intended to be a **left** multiplication; i.e. `A.multiply(B)` corresponds to the multiplication AB.
- `random()`: fills current the matrix with random numbers, uniformly distributed between 0 and 1. For a tri-diagonal matrix, this should fill the three diagonals with random numbers.

In subclasses, you should pay attention to what type of matrix needs to be returned from each of the functions. For example, when adding two `GeneralMatrix` objects the result should be a `GeneralMatrix` (which is then typecast to a `Matrix`).

2.2 The `GeneralMatrix` class

`GeneralMatrix` represents a full $m \times n$ matrix and extends `Matrix`.

1. The matrix will be stored in a `private` two-dimensional array.
2. You should implement all of the functions mentioned above using the standard formulae from linear algebra to do so, as well as the usual constructors, accessor and mutator methods.
3. You may choose whatever method you want to calculate the determinant of the matrix. However, it is **strongly** recommended you use the provided `decomp` function, which will perform LU decomposition for you since the algorithm is quite tricky for $n \times n$ matrices.

To call `decomp`, you should pass it a `double` array `d` of length 1. It will return a new `GeneralMatrix` storing both $L = (l_{ij})$ and $U = (u_{ij})$. For instance, when $n = 5$, the matrix returned is

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} & u_{15} \\ l_{21} & u_{22} & u_{23} & u_{24} & u_{25} \\ l_{31} & l_{32} & u_{33} & u_{34} & u_{35} \\ l_{41} & l_{42} & l_{43} & u_{44} & u_{45} \\ l_{51} & l_{52} & l_{53} & l_{54} & u_{55} \end{bmatrix}$$

The reason we can store it in this compact form is that the algorithm insists that $l_{ii} = 1$ for every i, and so this information can be omitted from the array.

On exit, the `double` inside the array you passed in will have a value of 1 or -1. You should multiply the calculated determinant by this value so that it has the correct sign. This constant arises because the decomposition algorithm will flip rows in the matrix to aid with singular matrices, thus changing the sign of the determinant.

As a result, if you explicitly perform the multiplication LU, you probably won't get the original matrix back again, but rather a permutation of it. For example, consider a matrix J which is a slightly altered identity matrix.

$$J = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \xrightarrow{\text{decompose } J} LU = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \neq J$$

In the algorithm, one row was was swapped, so `d[0]` will be -1.

2.3 The `TriMatrix` class

`TriMatrix` represents a tri-diagonal matrix of size $n \times n$ and extends `Matrix`. The constructor therefore only accepts a single parameter.

1. Tri-diagonal matrices are never stored in full two-dimensional arrays because they are *sparse* – that is, most of the entres are zero. Instead, we use three arrays of `double`s: `diag`, `upper` and `lower`. These store the diagonal, upper-diagonal and lower-diagonal elements respectively. In this form, the matrix looks like

$$T = \begin{bmatrix} d_1 & u_1 & & & & \\ l_2 & d_2 & u_2 & & & \\ & l_3 & d_3 & u_3 & & \\ & & l_4 & d_4 & \ddots & \\ & & & \ddots & \ddots & u_{n-1} \\ & & & & l_n & d_n \end{bmatrix}$$

   `diag` should therefore be of length n, whereas `upper` and `lower` should be of length $n - 1$.

2. For this class, you will need to implement your own `decomp` method to perform LU decomposition, which should *not* be copied from `GeneralMatrix`, since the algorithm for a tri-diagonal matrix is very simple to derive. First, we assume that the diagonal elements of the lower-diagonal matrix

L are 1. Then, you should show that the matrix product

$$\underbrace{\begin{bmatrix} 1 & & & & \\ l_2^* & 1 & & & \\ & l_3^* & 1 & & \\ & & \ddots & \ddots & \\ & & & l_n^* & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} d_1^* & u_1^* & & & \\ & d_2^* & u_2^* & & \\ & & \ddots & \ddots & \\ & & & d_{n-1}^* & u_{n-1}^* \\ & & & & d_n^* \end{bmatrix}}_{U}$$

is tri-diagonal. Finally, set the product equal to the matrix T above, and equate co-efficients to find a difference relation for each of the d_i^*, u_i^* and l_i^*. Just like the `decomp` method above, you can then store the matrix in a compact form inside a `TriMatrix`.

2.4 The `Project3` class

The final part of this project is to generate some simple statistics on random matrices. Here, the definition of random is that each co-efficient of the matrix M will have $M_{ij} \sim U(0,1)$ (i.e. a uniformly distributed random number between 0 and 1). $X = \det(M)$ is a random variable: the question is, how is X distributed? In particular, you will estimate the variance $\sigma^2 = \mathrm{var}(X)$ by generating a number of random matrices of various sizes, and then calculate the determinant of each of the samples.

`Project3` contains two functions to aid you in this endeavour. It is not meant to be challenging – indeed, it is probably the easiest part of the assignment!

- `matVariance()`: This function will be passed a `Matrix` object and an integer N_{samp}. It should generate random matrices M_i for $1 \leq i \leq N_{\text{samp}}$ by calling the `random` function on the passed matrix. The variance is estimated by

$$\sigma^2 = \mathbb{E}(X^2) - [\mathbb{E}(X)]^2 = \frac{1}{N_{\text{samp}}} \sum_{i=1}^{N_{\text{samp}}} \det(M_i)^2 - \left(\frac{1}{N_{\text{samp}}} \sum_{i=1}^{N_{\text{samp}}} \det(M_i) \right)^2$$

  You **should not** store each of the random samples, as this will consume a huge amount of memory for large values of N_{samp}.

- `main`: Your main function should not be a tester in this class. Instead, for $2 \leq n \leq 50$, it should create a $n \times n$ `GeneralMatrix` and a `TriMatrix`, and pass these to `matVariance()` to calculate the variance of the distribution for this value of n. For each n, you should generate 15,000 general matrix samples and 150,000 tri-diagonal samples.

  Ensure that you test `matVariance()` intensively **before** running with large numbers of samples. Start with a small number of samples at first to ensure you are not encountering infinite loops, etc. My solution code completes this in around a minute (on my laptop), so this should be your aim. You should print this information out to the terminal. On each line, print out

```
n var1 var2
```

  where `n` is the value of n, `var1` is the variance found for the `GeneralMatrix` and `var2` is the variance found for the `TriMatrix`.

Finally, you should plot two graphs with the data you find; one for the general matrix and one for the tri-diagonal. Along the x-axis plot the matrix size, and along the y-axis, the logarithm of the variance.

To save your output to a file, on `daisy` you can run the command

```
java Project3 > variance.data
```

and then transfer this file to your computer – for more information, see the week 12 and 15 lab notes, where you did something similar. Once you have this on your computer, you can then issue the following commands in Matlab to produce the plots:

```
load 'variance.data'
subplot(211)
semilogy(variance(:,1), variance(:,2), 'r')
subplot(212)
semilogy(variance(:,1), variance(:,3), 'b')
orient landscape
saveas(figure(1), 'VarGraph.pdf')
```

This will create two subplots; the top one containing the general matrix variance, the bottom the tri-diagonal matrix. Finally it saves the plots as a PDF file, which can be opened in Adobe Reader or similar viewers. You should add labels to the plot.

3 A note on efficiency

Your code will not, generally, be tested for efficiency, and will not be tested for very large matrices; at most, you will be given a 100×100 matrix. However, it perfectly possible to calculate the determinant of such a matrix in much less than a second using the methods outlined here. Bear in mind that you will need a certain amount of efficiency in your code in order to complete the `Project3` class.

Whilst it is certainly possible to write all of this code on `daisy`, I heartily encourage you to do your initial testing on your laptop and desktop machines. Not only do they provide a more friendly development environment, but if you inadvertantly run code with an infinite loop, it will not have an impact on other users.

Finally, the `Project3` class can be quite time-consuming, but shouldn't take more than a couple of minutes to run. You should test this on your own machine if possible; if not, then reduce the number of samples generated at first to get an initial indication of the time it will take to run. Remember that if your code is taking minutes to calculate determinants of small matrices, something is very wrong!

4 Submission

You should submit, using the Tabula system, the following four files: `Matrix.java`, `GeneralMatrix.java`, `TriMatrix.java` and `Project3.java`, as well as a PDF containing both of your plots called `VarGraph.pdf`. I will not accept any other format for this plot (Word, Excel, etc). Before you do that you should test that all your methods work properly (use the method `main` you implement in each class).

There will be a large number of tests performed on your classes. This should allow for some partial credit even if you don't manage to finish all tasks by the deadline. Each class will be tested individually so you may want to submit even a partially finished project. In each case, however, be certain that you submit Java files that compile without syntax error. Submissions that do not compile will be marked down. As before you can re-submit solutions as many times as you wish before the deadline; however, ensure that you re-submit **all** files.

Finally, please ensure that you keep back-up copies of your work. Lost data do not present a valid excuse for missing the deadline.