UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

COS110 Assignment 2

Matrix Arithmetic

Due date: 16 October 2022, 23:30

1 General instructions

- This assignment should be completed individually.
- Be ready to upload your assignment well before the deadline as no extension will be granted.
- If your code does not compile you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence of certain functions or classes).
- Read the entire assignment thoroughly before you start coding.
- You are not allowed to use any mathematical C/C++ libraries to complete this assignment.
- To ensure that you did not plagiarize, your code will be inspected with the help of dedicated software.
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `http://www.ais.up.ac.za/plagiarism/index.htm`.

2 Overview

For this assignment, you will create a simplistic mathematical "library" to perform various matrix manipulations and solve systems of linear equations. Operator overloading will be used to implement the matrix arithmetic.

3 Matrices

In mathematics, a matrix is a rectangular array of numbers, symbols, or expressions, arranged in rows and columns. Every element in a matrix is described by two indices `i` and `j`, where `i` is the row number of the element, and `j` is the column number of the element. For example, given a matrix A:

$$A = \begin{bmatrix} 1 & 5 & 7 \\ 8 & 7 & 3 \\ 2 & 9 & 1 \end{bmatrix}$$

The A(2,3) element of A is equal to 3, because that is the value located at row 2 and column 3 of A. Matrices can have any number of rows or columns:

$$\begin{bmatrix}7 & 8 & 9\\2 & 5 & 1\end{bmatrix}, \begin{bmatrix}5 & 3\\4 & 4\\9 & 0\end{bmatrix}, \begin{bmatrix}7\\4\\0\\3\\1\end{bmatrix}, \begin{bmatrix}4 & 4\\9 & 0\end{bmatrix}, \begin{bmatrix}4 & 7 & 8 & 9\\6 & 2 & 5 & 1\\5 & 6 & 7 & 8\end{bmatrix}, \begin{bmatrix}7 & 8 & 9 & 5 & 3 & 5\end{bmatrix}$$

Matrices are used extensively in most scientific fields. In every branch of physics, including classical mechanics, optics, electromagnetism, quantum mechanics, and quantum electrodynamics, they are used to study physical phenomena, such as the motion of rigid bodies. In computer graphics, they are used to project a 3-dimensional image onto a 2-dimensional screen. For this assignment, you will implement simple matrix mathematical operations.

4 Matrix arithmetic

This section describes the matrix mathematical operations that you are required to implement for this assignment.

4.1 Addition, subtraction, scalar multiplication and division

The simplest matrix operation is addition and subtraction of matrices. Only matrices with the same dimensions (equal number of rows and columns) can be added or subtracted from one another. Matrix addition is done element-wise: given two matrices A and B of the same dimension, every element A(i,j) is added to the element B(i,j):

$$\begin{bmatrix}2 & 3 & 1\\3 & 6 & 5\end{bmatrix} + \begin{bmatrix}1 & 5 & 7\\2 & 3 & 4\end{bmatrix} = \begin{bmatrix}2+1 & 3+5 & 1+7\\3+2 & 6+3 & 5+4\end{bmatrix} = \begin{bmatrix}3 & 8 & 8\\5 & 9 & 9\end{bmatrix}$$

The same rule applies to matrix subtraction.

Matrices can also be multiplied or divided by a scalar (i.e. non-matrix) value. To multiply a matrix A by a scalar `b`, every element A(i,j) is multiplied by `b`. Similarly, to divide a matrix A by a scalar `b`, every element A(i,j) is divided by `b`:

$$\begin{bmatrix}2 & 3 & 0\\3 & 6 & 4\end{bmatrix} /2 = \begin{bmatrix}2/2 & 3/2 & 0/2\\3/2 & 6/2 & 4/2\end{bmatrix} = \begin{bmatrix}1 & 1.5 & 0\\1.5 & 3 & 2\end{bmatrix}$$

Scalar multiplication:

$$2 * \begin{bmatrix}3 & 5\\1 & 4\end{bmatrix} = \begin{bmatrix}2*3 & 2*5\\2*1 & 2*4\end{bmatrix} = \begin{bmatrix}6 & 10\\2 & 8\end{bmatrix}$$

4.2 Matrix multiplication

Matrix A can be multiplied by a matrix B, provided that the number of columns in A is equal to the number of rows in B. Such restriction is due to the way matrices are multiplied. Before we discuss matrix multiplication further, let us define the *dot product* of two 1-dimensional matrices. Given a `1 x n` matrix A and a `n x 1` matrix B, their dot product is defined as the sum of products of `A(1,j)` and `B(j,1)` for every `0 < j < n`:

$$A = \begin{bmatrix}a_1 & a_2 & \dots & a_n\end{bmatrix}, B = \begin{bmatrix}b_1\\b_2\\\vdots\\b_n\end{bmatrix}$$

$$A.B = a_1 * b_1 + a_2 * b_2 + \cdots + a_n * b_n$$

For example,

$$A = \begin{bmatrix} 2 & 3 & 1 \end{bmatrix}, B = \begin{bmatrix} 0 \\ 4 \\ 5 \end{bmatrix}$$

$$A.B = 2 * 0 + 3 * 4 + 1 * 5 = 0 + 12 + 5 = 17$$

We can say that every `n x m` matrix A consists of `n` 1-dimensional row matrices A(ri), `0 < i < n`, and `m` 1-dimensional column matrices A(cj), `0 < j < m`. Now, to multiply a `n x p` matrix A by a `p x m` matrix B, we say that every `(i,j)` element of the resulting `n x m` matrix C will be the dot product of A(ri) (i-th row of A) and B(cj) (j-th column of B). Let A(rx) be a **row** of A, and B(cx) a **column** of B:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1p} \\ a_{21} & a_{22} & \dots & a_{2p} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{np} \end{bmatrix}, B = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \vdots & & & \vdots \\ b_{p1} & b_{p2} & \dots & b_{pm} \end{bmatrix}$$

$$C = A * B = \begin{bmatrix} A(r1).B(c1) & A(r1).B(c2) & \dots & A(r1).B(cm) \\ A(r2).B(c1) & A(r2).B(c2) & \dots & A(r2).B(cm) \\ \vdots & & & \vdots \\ A(rn).B(c1) & A(rn).B(c2) & \dots & A(rn).B(cm) \end{bmatrix}$$

For example:

$$A = \begin{bmatrix} 2 & 3 & 1 \\ 5 & 1 & 0 \end{bmatrix}, B = \begin{bmatrix} 4 & 2 \\ 8 & 1 \\ 3 & 4 \end{bmatrix}$$

$$A * B = \begin{bmatrix} A(r1).B(c1) & A(r1).B(c2) \\ A(r2).B(c1) & A(r2).B(c2) \end{bmatrix} = \begin{bmatrix} (2*4+3*8+1*3) & (2*2+3*1+1*4) \\ (5*4+1*8+0*3) & (5*2+1*1+0*4) \end{bmatrix} =$$

$$= \begin{bmatrix} (8+24+3) & (4+3+4) \\ (20+8+0) & (10+1+0) \end{bmatrix} = \begin{bmatrix} 35 & 11 \\ 28 & 11 \end{bmatrix}$$

Note that A*B is not equal to B*A:

$$B * A = \begin{bmatrix} B(r1).A(c1) & B(r1).A(c2) & B(r1).A(c3) \\ B(r2).A(c1) & B(r2).A(c2) & B(r2).A(c3) \\ B(r3).A(c1) & B(r3).A(c2) & B(r3).A(c3) \end{bmatrix} =$$

$$= \begin{bmatrix} (4*2+2*5) & (4*3+2*1) & (4*1+2*0) \\ (8*2+1*5) & (8*3+1*1) & (8*1+1*0) \\ (3*2+4*5) & (3*3+4*1) & (3*1+4*0) \end{bmatrix} = \begin{bmatrix} (8+10) & (12+2) & (4+0) \\ (16+5) & (24+1) & (8+0) \\ (6+20) & (9+4) & (3+0) \end{bmatrix} =$$

$$= \begin{bmatrix} 18 & 14 & 4 \\ 21 & 25 & 8 \\ 26 & 13 & 3 \end{bmatrix}$$

For more information on matrix multiplication, refer to `http://www.mathsisfun.com/algebra/matrix-multiplying.html`.

4.3 Powers of a matrix

Given a non-negative integer k, a k-th power of a square (`n x n`) matrix A can be calculated by multiplying A by itself k times. If k is zero, the matrix must be transformed into an **identity** matrix, i.e. a square matrix of all zeros, with ones only on the `{i,i}` diagonal:

$$A = \begin{bmatrix} 2 & 3 \\ 5 & 1 \end{bmatrix}, A^0 = I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

General case for `k = 0`:

$$A = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & & & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nn} \end{bmatrix}, A^0 = I = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & & & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

4.4 Matrix transpose

A *transpose* of a `n x m` matrix A is a `m x n` matrix ~A where every row of A becomes a column of ~A. For example:

$$A = \begin{bmatrix} 2 & 3 & 1 \\ 5 & 1 & 0 \end{bmatrix}, \tilde{A} = \begin{bmatrix} 2 & 5 \\ 3 & 1 \\ 1 & 0 \end{bmatrix}$$

5 Systems of linear equations

Now that you know how matrices work, let us consider how matrices can be used to solve systems of linear equations.

A linear equation is an equation that describes a straight line. Recall that it is very easy to solve a linear equation with a single unknown:

$$\begin{aligned} 2x + 3 &= 6 \\ 2x &= 6 - 3 = 3 \\ \therefore x &= 3/2 = 1.5 \end{aligned}$$

Now consider a linear equation with two unknowns:

$$2x + 3y = 12$$

Clearly, no obvious solution to the above equation exists. However, if we were given a *system* of two linear equations, we could have expressed one unknown in terms of the other one, and find out the values of both unknowns by substitution:

$$\begin{cases} 2x + 3y &= 12 \\ x + 2y &= 4 \end{cases} \Rightarrow \begin{cases} 2x + 3y = 12 \\ x = 4 - 2y \end{cases} \Rightarrow \begin{cases} 2(4 - 2y) + 3y = 12 \\ x = 4 - 2y \end{cases} \Rightarrow$$

$$\begin{cases} 8 - 4y + 3y = 12 \\ x = 4 - 2y \end{cases} \Rightarrow \begin{cases} 8 - y = 12 \\ x = 4 - 2y \end{cases} \Rightarrow \begin{cases} y = 8 - 12 = -4 \\ x = 4 - 2y = 4 - 2 * (-4) = 4 + 8 = 12 \end{cases}$$

Solving a small linear system by hand is easy, but imagine having to deal with a dozen or a hundred of unknowns. For such cases, a more efficient solution is necessary. One way to automatically solve a system of linear equations is to use the *Gaussian elimination* algorithm. The basic idea behind the algorithm is to iteratively reduce the equations in a linear system to a single unknown per equation. To achieve it, the following manipulations are applied to a linear system:

1. Multiplying both sides of an equation by a scalar (`x + 2y = 4` is equivalent to `2x + 4y = 8`)
2. Reordering the equations by interchanging both sides of the i-th and j-th equation in the system
3. Replacing equation i by the sum of equation i and a multiple of both sides of equation j

The third operation is by far the most useful. We will now demonstrate how it can be used to reduce a system of equations to a form in which it can be easily solved.

Consider the following system of equations:

$$\begin{cases} 2x + 3y &= 12 \\ x + 2y &= 4 \end{cases}$$

If we divide the first equation by 2 (giving us `x + 1.5y = 6`), and then subtract it from the second equation, we will arrive at the following:

$$\begin{cases} 2x + 3y = 12 \\ x + 2y - (x + 1.5y) = 4 - 6 \end{cases} \Rightarrow \begin{cases} 2x &+3y = 12 \\ &0.5y = -2 \end{cases}$$

The `x` term has been **eliminated** from the second equation. The system is in *upper triangular form*: the bottom equation has a single unknown, and the top equation has two unknowns. Now we can easily derive the value of `y` from the second equation, and once that is known, we can substitute `y` into the first equation to get the value of `x`. In general, any system in an upper triangular form is trivial to solve: we start with the bottom equation, solve it, substitute the solution to the equation above, etc. till all equations are solved.

How do matrices fit in with linear systems? First of all, note that the linear system above can be re-written in matrix form as follows (check it with matrix multiplication):

$$\begin{cases} 2x + 3y &= 12 \\ x + 2y &= 4 \end{cases} \Rightarrow \begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 12 \\ 4 \end{bmatrix}$$

To make notation easier, we often omit the 1-dimensional matrix of unknowns, and write the above in the form of an *augmented* matrix:

$$\left[\begin{array}{cc|c} 2 & 3 & 12 \\ 1 & 2 & 4 \end{array}\right]$$

Now, the row manipulations as described above can be applied to the augmented matrix. When the augmented matrix is in an upper-triangular form (i.e., all entries below the main matrix diagonal are zero), back-substitution is used to iteratively solve the linear system starting with the bottom equation.

To summarise, solving a system of linear equations with matrices and Gaussian elimination consists of two steps:

1. Elimination: reduce the augmented matrix to the upper-triangular form
2. Back-substitution: iteratively solve the linear system, starting with the bottom row of the upper-triangular matrix

Suppose a linear system is expressed by a `n x n` augmented matrix A:

$$A = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \vdots & & & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & b_n \end{array}\right]$$

The following algorithm can be used to reduce A to upper-triangular form:

```
// declare a multiplier m:
// the number by which a row j is multiplied before being added to row i
m;
for j = 1,2,...,n-1 // iterate through all rows of A, starting from the top
{
    for i = j+1, j+2,...,n // eliminate the leftmost unknown of row j
    {                      // from every row i below
        m = A(i,j)/A(j,j);   // calculate the multiplier
        for k = 1, 2,...,n   // iterate through all elements of row i
        {
            A(i,k) = A(i,k) - m * A(j,k); // subtract row j * m from row i
        }
        b(i) = b(i) - m * b(j); // the matrix is augmented:
        // we have to subtract j-th b * m from i-th b
    }
}
// after algorithm execution, augmented matrix A is in upper-triangular form
```

We have thus reduced A to upper-triangular form:

$$A = \left[\begin{array}{ccccc|c} a_{11} & a_{12} & \dots & a_{1,n-1} & a_{1n} & b_1 \\ 0 & a_{22} & \dots & a_{2,n-1} & a_{2n} & b_2 \\ \vdots & & & & \vdots & \vdots \\ 0 & 0 & \dots & 0 & a_{nn} & b_n \end{array}\right]$$

The following algorithm can be used on an upper-triangular matrix to perform back-substitution and obtain solutions to the linear system:

```
s = b; // declare an n x 1 matrix to store the solution values
// and copy values of b into it
for i = n, n-1,...,1 // iterate through A row by row,
{                    // starting from the bottom row
    for j = i+1, i+2,...,n // substitute solutions from the rows below
    {
        s(i) = s(i) - A(i,j) * s(j);
    }
    s(i) = s(i) / A(i,i); // calculate the value of the unknown in row i
}
```

Upon the execution of this algorithm, `n x 1` matrix `s` will contain the solutions to the linear system - in other words, the values of the unknowns.

6 Your task

This assignment consists of three tasks:

1. Implementing the Vector and Matrix class with the basic input/output functionality
2. Implementing the matrix arithmetic with Vector and Matrix class using operator overloading
3. Implementing Gaussian elimination algorithm to solve systems of linear equations using the Matrix class

6.1 Task 1: Implementing the Matrix

File `arithmetic.h` which is the parent class contains an incomplete header of the Arithmetic class. Both Vector and Matrix class will inherit from the base class Arithmetic.

The Matrix class is two-dimensional dynamic array whiles the Vector class is a one-dimensional dynamic array, and are used to store the data.

6.1.1 Constructors

Matrix and Vector constructors takes two and one **`unsigned int`** parameters respectively, representing the number of rows and columns in the matrix and the size in the vector. Upon receiving the dimension parameters, the constructor must allocate a dynamic array of the given size, and fill it with zeros. Remember to implement a destructor to deallocate the memory when the program exits. In addition to the constructor, implement an appropriate copy constructor, and overload the assignment operator = to enable assigning objects to one another.

6.1.2 Data input and output

To populate a matrix object with data, we will override the method **`void`** `readFile(ifstream & infile)` to read matrix data from either the keyboard or a file stream. Make sure you can use >> to input a 3 x 3 matrix stored in a text file in the following format:

```
2 4 3
5 2 3
7 1 0
```

Override the methods **`void`** `print()` to output a matrix to the screen or file in the following format for a matrix:

```
2       4       3
5       2       3
7       1       0
```

and for vector in the following format:

```
2       4       3       5       2       3       7       1       0
```

Every element should occupy exactly 10 character spaces on the screen. The precision (number of decimal points) should be set to 3. Hint: use `setw()` and `setprecision()` functions of the `iomanip` library. Do not add any additional empty lines: just print the data.

To access individual elements of the Matrix overload the operator `[][]` to take two **`int`** parameters. Moreover, to modify individual elements of the Matrix overload the operator `()` to take two **`int`** parameters such that the following is possible (note that matrix indices start from 0,0):

```
Matrix mat(2,3)          // create a 2 x 3 matrix
mat(0,0) =  5           // set element 0,0 to 5
cout << mat[0][0] << endl; // output element 5 to screen
If row/column is out of range: "Error: invalid row index" or "Error: invalid column
    index"
```

Note: There is no operator[][] in C++. You have to return a helper object (vector) and then overload operator[] for that too, or index the return vector, to have this kind of access. This is one way. You use any other logic that best fit.

To access individual elements of the Vector, overload the operator `[]` to take an **`int`** parameters such that the following is possible:

Overload `[]` to take one **`unsigned int`** input parameter such that the following is possible for a vector:

```
Vector vec(6);          // create a 1 x 6 vector
vec[0] = 5;             // set element 0 to 5
cout << vec[0] << endl; // output 5 to screen
```

You should also implement getter functions to return the number of rows and the number of columns in a matrix and the size of a vector.

6.2 Task 2: Arithmetic operations

All arithmetic operations as described in Section 4 is to be implemented via operator overloading. Error checking has to be performed where appropriate, and if invalid input is given, an exception has to be thrown. For the purpose of this assignment, all exceptions will take the form of text error messages.

Table 1 summarises the operators that you have to overload. Assume that `a` and `b` are objects of the derived class from the `arithmetic` parent class, and `n` is of type **`double`**.

Table 1: List of operators to overload

Operator	Usage	Exception	Example
*	object * n	none	a*2
*	n * object	none	2*a
*=	object *= n	none	a*=2
/	object / n	If n is 0: "Error: division by zero"	a/2
+	object + object	If row/column dimensions do not match: "Error: adding matrices of different dimensionality"	a + b
+=	object += object	If row/column dimensions do not match: "Error: adding matrices of different dimensionality"	a += b
-	object - object	If row/column dimensions do not match: "Error: subtracting matrices of different dimensionality"	a - b
-=	object -= object	If row/column dimensions do not match: "Error: subtracting matrices of different dimensionality"	a -= b
*	object * object	If row/column dimensions are incorrect: "Error: invalid matrix multiplication"	a * b
*=	object *= object Not applicable to a vector	If row/column dimensions are incorrect: "Error: invalid matrix multiplication"	a *= b
^	object to the power of n, object^n	If the matrix is not square (n x n): "Error: non-square matrix provided" If the power is negative: "Error: negative power is not supported"	a^2
^=	object to the power of n, object^=n	If the matrix is not square: "Error: non-square matrix provided" If the power is negative: "Error: negative power is not supported"	a^=2
~	object transpose. For a vector, you need to reverse the elements	none	~a

NOTE: For a vector implementation of operator ^, you should perform element-wise calculation as shown below:

```
Vector vec(3);          // create a 1 x 3 vector with elements {1,2,3}
vec ^ 5;             // this should result in {1, 32, 243}
```

NOTE: In the exception message, replace matrix and matrices with vector and vectors respective for vector operations.

6.3 Task 3: Implementing Gaussian elimination

Gaussian elimination described in Section 5 is to be implemented for this part of the assignment. For the purpose of this assignment, we will not create augmented matrices as seen in Section 5. Instead, a system of linear equations will be represented by two separate Matrix objects: a `n x n` matrix `A` and a corresponding `n x 1` matrix `b`.

Operator overloading will be used to implement gaussian elimination. Firstly, operator `|=` will be used to reduce the combination of `A` and `b` to upper-triangular form as follows:

```
Matrix A(4,4), b(4,1); // declare matrices
A.readFile(infile); // read matrix data from file
b.readFile(infile); // read matrix data from file
// reduce A augmented with b to upper triangular form:
A|=b;  // this operator changes both A and b
```

To perform back-substitution, operator `|` has to be overloaded. Given two matrices `A` and `b` representing a system of linear equations, the `n x 1` matrix of solutions `s` can be obtained as follows:

```
Matrix s = A|b; // given A and b, obtain the solution matrix s
```

The operator should not modify `A` or `b` (hint: work on their copies). The operator first checks if `A` and `b` form an augmented matrix in upper-triangular form. If they don't, gaussian elimination is applied to reduce `A|b` to the desired form. Then, back-substitution algorithm is applied to calculate the `n x 1` solution matrix.

Table 2 summarises the two operators that you have to overload for this task. Assume that `a` and `b` are objects of the `Matrix` class.

Table 2: Operators for linear equation

Operator	Usage	Exception	Example
`|=`	matrix |= matrix	If the left hand side matrix is not `n x n`: "`Error: non-square matrix provided`" If the right hand side matrix is not `n x 1`: "`Error: incorrect augmentation`" If division by zero is encountered during gaussian elimination: "`Error: division by zero`"	`a|=b`
`|`	matrix | matrix	If the left hand side matrix is not `n x n`: "`Error: non-square matrix provided`" If the right hand side matrix is not `n x 1`: "`Error: incorrect augmentation`" If division by zero is encountered during gaussian elimination or back-substitution: "`Error: division by zero`"	`a|b`

6.4 Implementation considerations

- implement both `matrix.h/cpp` and `vector.h/cpp` to inherit from the base class `arithmetic`. It should to include the declarations of the overloaded operators, as well as any additional helper functions your implementation might require.
- No mathematical C/C++ libraries can be used
- Input data used to test your program will be space-separated
- Remember to throw exceptions where necessary
- All exceptions are simple one-line text messages. Do not add breaks, new line characters or extra hyphenation.
- Your classes should throw exceptions - not catch them
- All catching is done from the main function
- Test your code thoroughly. Test under different scenarios, with a diverse set of inputs. If an exception can be thrown, provide incorrect input to trigger it.

6.5 Submission

Once you are satisfied that your program is working, compress all of your code, into a single archive (either .tar.gz, .tar.bz2 or .zip) and submit it for marking to the appropriate upload link before the deadline. No object files or executables should be included in the archive. The archive must contain no folders or subfolders.