1 Introduction

MatrixLib is a free, open-source library for Java developers who want to easily make linear algebraic computations. I, Darrion, wanted to start creating this project while I was taking MATH 242 (Elementary Linear Algebra) at Bard College. This new branch of math piqued my interest, and with its extreme usage in the field of programming, I knew creating MatrixLib would be a great way to learn even more about linear algebra and mathematical computation. MatrixLib is built with Maven, a tool that manages building projects. Using Maven makes MatrixLib all the easier to implement due to Maven's dependency management. Installation instructions can be found on the page of the repository (https://github.com/Darrionat/MatrixLib). In addition, MatrixLib is independent software; therefore, it has no dependencies besides Java itself. MatrixLib is designed around ease of use, and continues to be a tool that I use for various computations.

2 The Quantity

MatrixLib makes use of a custom number system. This system is centered around the abstract class, *Quantity*. *Quantity* is an immutable data structure that represents a mathematical quantity or set of numbers. It also implements multiple interfaces, including one named *Operable*. The *Operable* interface contains the following abstract methods: *add(Quantity)*, *subtract(Quantity)*, *multiply(Quantity)*, *divide(Quantity)*, *pow(int)*, *negate()*, *zero()*. Since *Quantity* is an abstract class that is inherited, it actually does not implement the methods from *Operable* - besides *pow(int)* which will be mentioned later. All of these methods return a *Quantity* except *zero()* which returns a *boolean*. General assumptions based upon the method signatures are most likely accurate, but for further detail, see the JavaDocs.

Currently, the *Quantity* superclass is inherited in both the *Rational* and *Complex* data structures. This polymorphic approach allows for many benefits such as compatibility, using interface methods instead of class methods, etc. Creating a numerical data structure serves a few key problems: speed of computation, memory usage, and ease of use. All three will be addressed in the following.

Rational

The *Rational* is the backbone of the numerical system within MatrixLib. Rational numbers are the foundation for other numerical systems in MatrixLib. Within MatrixLib, the *Rational* class is designed for speed and ease of use. Since the *Rational* will serve as the foundation for any other numerical systems within MatrixLib, it needed to promise speed. While using the operational methods, this class directly accesses private class fields instead

of "getters/setters" in order to access information promptly. This class is based upon *BigInteger*, a utility provided by Java which allows more integers to be represented than primitive types. One of its constructors, for example, is *Rational(BigInteger numerator, BigInteger denominator)*.

The *Rational* class allows for simple operations, such as addition, subtraction, multiplication, and division. In addition to these operations, it also is able to perform exponentiation. There are a few more, but they are more specific and can be seen within the library. As you may have already noticed, not all real numbers can be represented. This includes real numbers such as $\sqrt{2}$. The lack of implementation for representation of all real numbers is currently due to the lack of necessity. However, since MatrixLib is open-sourced, so it is quite simple for one to contribute a data structure that represents real numbers.

Complex

The *Complex* class is another data type that represents a *Quantity*, although it represents imaginary values as well. The *Complex* value can be represented as $a + bi$ where a and b are both *Rational*s. The beautiful part about the representation of complex numbers is that they can also represent a real value. In a case where $b = 0$, the method, *Complex#toRational()* can always be used - of course, if the complex value is non-real, an exception will be thrown. Also, the *Complex* structure uses the *pow(int)* method from *Quantity* - this will be explained soon. The *Complex* data structure is as simple as creating two different rationals and then constructing a new instance. This data structure also allows for the same sort of operations as *Rational*, but with its own object.

Computations

Having more than one type of *Quantity* arises the issue of compatibility. Therefore, each data structure's operations take into account the type of *Quantity* being processed and how to behave from that. For example, let x be a *Rational* instance and y be a *Complex* instance. When *x.add(y)* is executed, the method will return a *Quantity* of *Complex* typing.

As previously mentioned, the only operation that *Quantity* overrides from *Operable* is *pow(int)*. This is because MatrixLib contains a custom algorithm that handles exponentiation if a data type does already define one (as *Rational* does). Each *Quantity* has a *HashMap¡Integer, Quantity¿* that represents its powers. When the *pow(int)* method is called, it puts the powers of 0 and 1 into the map. This is because their values are already known, one and the quantity itself, respectively. It then converts the given *int* into a binary representation of that integer. For example, $55 = 32+16+(0)8+4+2+1 \implies 110111$. There are then two loops that run. The first loops through the binary string, starting from the second to the last character, and moves towards the beginning of the string. For each character, it will calculate that power of two, regardless of the bit value at that location. The second then loops through the entire binary starting at the beginning. The result initially is equal to 1.

Each iteration then multiplies this result by the calculated value of that position if the bit value is 1. After this loop, the result is finished calculating, and returned.

This may seem like a pretty big "info dump," so let's work through an example. Say we want to calculate 5^{55}. As we know from before, 55 as a binary string is 110111. Therefore, $5^{55} = 5^{32} \cdot 5^{16} \cdot 5^4 \cdot 5^2 \cdot 5^1$. Since we already know that 5^1 is 5, we can then just square each power to obtain the next. As one can see, this process allows for an extremely quick calculation.

Rationals, however, do not use the default method for exponentiation. They instead utilize a method from *BigInteger*. Moreover, this algorithm works especially well with *Complex* values. Let $a+bi$ and $c+di$ be two different complex values where a, b, c, d are rationals. Here it is shown that regardless of the values, we already know the product of the two complex values.

$$(a + bi)(c + di) = ac + adi + bci + bdi^2$$

$$\implies ac + bdi^2 + adi + bci$$

$$\implies (ac - bd) + (ad + bc)i$$

Since a, b, c, and d are all rationals, we can quickly compute the product of two complex values. Thus, a minimal amount of computations are performed, and when they are performed, the computations are efficient. This is exactly why the exponentiation algorithm in *Quantity* can be performed efficiently for complex numbers.

3 Matrices

Matrices were the main focus when designing MatrixLib. However, these matrices could have contained any sort of data structure, which is why the numerical system was created. The library supports any sort of $m \times n$ matrix where m and n are positive integers. This starts with the abstract class *OperableMatrix*. The *OperableMatrix* represents a two-dimensional array of quantities. It is able to perform basic operations such as changing values within the matrix, row/column swaps, scaling a row/column, taking row sums, etc. Since the *OperableMatrix* is based upon a two-dimensional array, getting/setting row values is faster than getting/setting. This is due to the array being in the form of *Quantity[rows][row_values]*. This is purely a design choice and could easily be reversed if wanted.

The MatrixBuilder

MatrixLib contains a built-in way to parse matrices from strings. This creates a convenience when loading or saving matrices from files or other use cases. Utilizing the *OperableMatrix#toString()* method, a matrix will be returned as a string. The *MatrixBuilder* can

be utilized for parsing matrices from strings. Here, the *MatrixBuilder#parseMatrix(String)* builds a matrix from a given input. Of course, if the input is formatted incorrectly, exceptions will be thrown. Here is an example of the string formatting,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \iff [\{1, 2\}; \{3, 4\}].$$

It can be seen that the left and right brackets are almost useless information within the string format of the matrix. Since it is only two characters, they are kept as a convenience and could be used to separate multiple matrices. Within the project's test folder, there are multiple methods of saving and loading matrices implemented for testing, including compression handling and saving to a CSV file.

Matrix Operations

The *Matrix* class inherits the *OperableMatrix* superclass. Currently, *Matrix* is the only subclass of *OperableMatrix*; however, it allows for a more modularized approach. The *Matrix* data structure is designed for adding more operations to the matrix and allowing it to interact with other matrices. These operations include methods such as row echelon form, reduced row echelon form, matrix multiplication, scaling, and performing entry-wise sums between matrices.

Other Types of Matrices

There are other, more specialized, types of matrices within MatrixLib as well. The *SquareMatrix* class is a subclass of *Matrix*. Unlike its parent, it must have equal dimensions for its column(s) and row(s). In addition, this class inherits all of the methods from before. The *SquareMatrix* is able to calculate the determinant of the matrix and give a boolean value of its triangularity - the state of being triangular. The *SquareMatrix#det()* method creates a clone of the matrix instance, puts the clone into REF, and then takes the product of all pivots.

The second type of specialized matrix is the *IdentityMatrix*. The *IdentityMatrix* is a subclass of *SquareMatrix*. This type of matrix will only have ones across its diagonal and all other values are zero. This class is **final**, therefore it cannot have subclasses. This is because the *IdentityMatrix* is intended to be immutable. The identity matrix can be utilized in computations such as obtaining the characteristic polynomial of a square matrix.

4 Algebra

As we have seen before, MatrixLib contains its own numerical system and support for different types of matrices. In addition to this, MatrixLib also allows for the representation of numerical values through expressions. These expressions contain mathematical operations

and values that are supported by the MatrixLib numerical system, so rationals and complex values. An expression could be as simple as $2 + 2$.

Operations

To understand how to properly create an expression, one must know the operations of the expression first. The *Operation* enum currently has five different values: *ADD*, *SUBTRACT*, *MULTIPLY*, *DIVIDE*, and *POW*. The operators of these values are $+, -, *, /$, and $\hat{}$ respectively. These five operators represent the same methods that we saw earlier in the *Operable* interface. Each *Operation* value has its own string to represent its operator. Two quantities can be evaluated through this enum as well. For example, *Operator.MULTIPLY.getResult(a,b)* where a and b are both quantities. This would be the same as performing *a.multiply(b)*.

Creating an Expression

These expressions can be created with the *Expression* class and instantiated with the static *Expression#buildExpression(String)* method. There is no public constructor for the *Expression*, but instead, it is required that this builder method. When a string, is fed, it must have proper syntax and only valid operators. In essence, the expression must be formatted in such a way that there are never two adjacent operators, there may not be more right parentheses than left parentheses, and an operator may never start or end an expression.

The Interpreter

An expression alone is to no use, for we do not know its value. This is where the *Interpreter* is a vital piece to the *Expression*. All methods within the *Interpreter* class are static and package-visible or private. So an expression can be only be evaluated as a *Quantity* by using the *Expression#evaluate()* method.

When the interpreter runs, it utilizes a recursive process while abiding by the order of operations. If there are no parentheses, the expression is read and the typical order of operations takes place. If there are parentheses, the Interpreter will first look for pairs of parentheses, finding the deepest set of parentheses. The sub-expression found within this set of parentheses is then evaluated, and the original sub-expression and parentheses are replaced by this value. If parentheses are used to multiply, the interpreter will insert the multiplication operator within the expression's string. The resulting expression is then evaluated recursively until there are no more sets of parentheses. Thus, the expression can be directly evaluated just as the sub-expressions had been prior.

Variables

Currently, MatrixLib has no support for variables. The current reason is that the addition of algebraic variables could propose a great complexity and the task has not yet been taken

on. In addition to this, there would be a great number of algebraic manipulations that would have to be considered as well. In the future, this task may be taken on; however, MatrixLib currently only supports direct computations.