346 assignment report

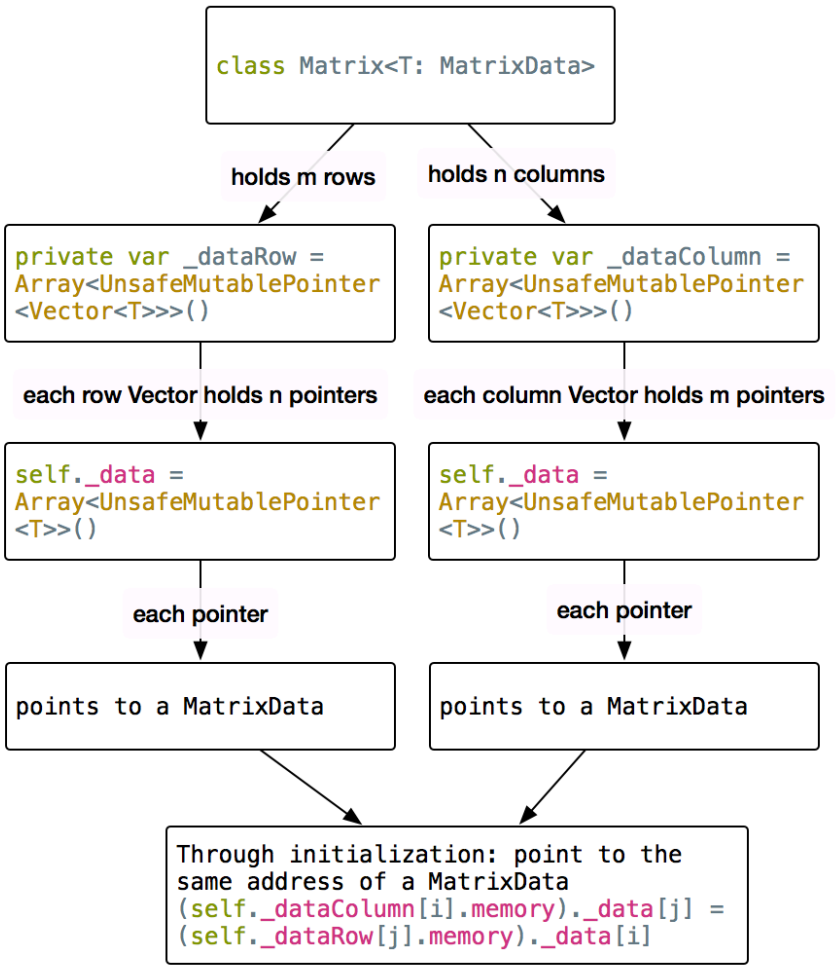
Group member:
Wei Zhao (2767591)
Jialin Yang (7674607)

This is our document for our Matrix&Vector library and the test class. This project is split into two part. Wei Zhao is responsible for the implementation of Matrix&Vector class. And Jialin Yang works on the test class. The following is a description on these two parts.

Part 1: Implementation by Wei Zhao

Generally, I use composition in the implementation of my Matrix and vector library. It is naturally that a matrix is a composition of row Vectors or columns Vectors. More specifically, my Matrix contains the row prospects and column prospects of Vectors at the same time.

The main challenge in this assignment is that: a Vector can be derived from a row or a column of a Matrix, and the later changes on the Vector can be reflected onto the Matrix, and vice versa. To make this happened, the Matrix and Vector should be flexible. So I choose use pointers to accomplish this. The diagram below shows this composition:



Vector Operation:

1. To create a Vector:

var v1 = Vector<Int>(size: 4)

The type of Vector could be Int, Flout, Double, Fraction, Complex.

This will create a Vector with type Int, with size of 4, and its initial value is 0.

$$v1 = [0 \quad 0 \quad 0 \quad 0]$$

```
// Vector initialization
var v1 = Vector<Fraction>(size: 3)
var v2 = Vector<Complex>(size: 5)
var v3 = Vector<Int>(size: 2)
var v4 = Vector<Float>(size: 1)
var v5 = Vector<Double>(size: 4)
```

```
0 0 0
0.0 0.0 0.0 0.0 0.0
0 0
0.0
0.0 0.0 0.0 0.0
```

2. After creating a Vector, the Vector instance can have some basic operation:

1) v1.size return an Int which indicate the size of element the Vector contains.
2) v1[2] = 1, this subscript operation will change v1 into

$$v1 = [0 \quad 0 \quad 1 \quad 0]$$

3) Dot operation, which accept a Vector with same size and same type, and return the dot product of two Vectors.
   var dot: Int
   dot = v1.dot(Vector<Int>(size: 3))
4) var v2 = v1.copy() will return a new Vector instance with the same values as

```
// Vector basic operation
v2.size
v1[1] = Fraction(num: 1, den: 3)
v1
var v6 = v1.copy()
v1.dot(v6)
```

```
5
1/3
 0 1/3 0
 0 1/3 0
1/9
```

3. VectorArithmetic operation

When the return type is a Vector, we choose to always return a new Vector. Because we don't want the operation to interfere with original Vector.

var v1 = Vector<Int>(size: 3), $v1 = [0 \quad 0 \quad 0]$

1) v1 = v1 + 1, then $v1 = [1 \quad 1 \quad 1]$
   Addition of a scalar (single number) value to v1
   Notice:
   The order of operation is important, so 1 + v1 will not be available.
   This actually return a new Vector and assign the new Vector to v1.
   The right size of '+' should be the same type as Vector's data type.
   v1 = v1 + Complex(real: 1, imag: 1), this operation is also not allowed.
   Same rule apply for subtraction, multiplication and division by a scalar value.
   v1 = v1 – 1,
   v1 = v1 * 2,
   v1 = v1 / 2.
2) Vector can also apply '+, -, *' to another size and type Vector

```
// VectorArithmetic operation
v1 = v1 + Fraction(num: 1, den: 1)
v6 = v1 - Fraction(num: 1, den: 2)
v1 = v1 * Fraction(num: 2, den: 1)
v1 = v1 / Fraction(num: 1, den: 3)

var result3 = v1 + v6
var result1 = v1 - v6
var result2 = v1 * v6
```

```
1 4/3 1
1/2 5/6 1/2
2 8/3 2
6 8 6

13/2 53/6 13/2
11/2 43/6 11/2
38/3
```

Matrix Operation

1. To create a Matrix, is similar with create a Vector. The initial value of matrix is zero of associated type.

```
// Matrix initialization
var m1 = Matrix<Int>(rows: 3, columns: 3)
var m2 = Matrix<Fraction>(rows: 4, columns: 4)
var m3 = Matrix<Float>(rows: 2, columns: 2)
var m4 = Matrix<Double>(rows: 2, columns: 4)
var m5 = Matrix<Complex>(rows: 2, columns: 2)
print(m1)
```

```
0 0 0
0 0 0
0 0 0
```

```
print(m4)
```

```
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
```

2. After creating an instance of Matrix, it also supports the basic Matrix operation.

var m6 = Matrix<Int>(rows: 4, columns: 3)

1) m6.columns return the size of columns
2) m6.rows return the size of rows
3) subscript can set and get the Matrix data at specific row and column indexed from 0: m6[2,2] = 7
4) m6.transpose returns the SAME Matrix with columns and rows switches.
   The transpose operation use a Boolean value to indicate whether the Matrix is transposed or not since it has been initialized. Then according to this Boolean value, the column or row prospect data will be used to represent the Matrix.
5) m6.copy() returns a new Matrix. It first creates a Matrix with same size and type, then copy each Matrix data into the new Matrix.

```
var m6 = Matrix<Int>(rows: 4, columns: 3)
print(m6)
```

```
0 0 0
0 0 0
0 0 0
0 0 0
```

```
0 0 0
" 0 0 0\n 0 0 0\n 0 0 0\n 0 0 0\n"
```

```
m6.columns
m6.rows
m6[2,2] = 7
m6[2,2]
print(m6)
```

```
3
4
7
7
" 0 0 0\n 0 0 0\n 0 0 7\n 0 0 0\n"
```

```
0 0 0
0 0 0
0 0 7
0 0 0
```

```
m6.transpose
m6.columns
m6.rows
var m7 = m6.copy()
print(m7)
```

```
0 0 0 0
4
3
0 0 0 0
" 0 0 0 0\n 0 0 0 0\n 0 0 7 0\n"
```

```
0 0 0 0
0 0 0 0
0 0 7 0
```

3. MatrixArithmetic operation

For Matrix operation, they all return a new Matrix according to the rule of operation.

1) When Matrix operate with a scalar (single number) value, same with Vector:
   The order of operation is important
   The type must be the same.

```
var m8 = Matrix<Int>(rows: 2, columns: 3)
print(m8)
```

```
0 0 0
0 0 0
```

```
m8 = m8 + 1
print(m8)
```

```
1 1 1
1 1 1
```

```
m8 = m8 - 4
print(m8)
```

```
-3 -3 -3
-3 -3 -3
```

```
m8 = m8 * 2
print(m8)
```

```
-6 -6 -6
-6 -6 -6
```

```
m8 = m8 / 5
print(m8)
```

```
-1 -1 -1
-1 -1 -1
```

2) When Matrix '+', '-', '*' another Matrix. The size and type of Matrix must be matched with the operation.

```
var m9 = Matrix<Int>(rows: 2, columns: 3)
print(m9)
```

```
0 0 0
0 0 0
```

```
m9 = m9 + m8
print(m9)
```

```
-1 -1 -1
-1 -1 -1
```

```
m8 = m8 - (m9 * -1)
print(m8)
```

```
-2 -2 -2
-2 -2 -2
```

```
var m10 = m8 * (m9.transpose)
print(m10)
```

```
6 6
6 6
```

Matrix and Vector Conversion

A Vector can convert into a single row Matrix.
From a Matrix a Vector can be derived by specified the *i*th column or row of a Matrix.
If the Vector or Matrix is produced by this kind of conversion, then any changes made on Matrix or Vector could reflected back on each other.

1. To get a Vector from Matrix, just specify the index of row or column.

   var v1 = m.column(2)
   var v2 = m.row(0)

```
var m = Matrix<Int>(rows: 3, columns: 4)
m[2, 2] = 7
m[0, 1] = 3
print(m)
```

```
0 3 0 0
0 0 0 0
0 0 7 0
```

```
0 0 0 0
7
3
" 0 3 0 0\n 0 0 0 0\n 0 0 7 0\n"
```

```
var v1 = m.column(2)
var v2 = m.row(0)
v1
v2
m[2, 2] = 6
m[0, 1] = 2
v1
v2
v1[0] = 1
v1
v2[0] = 3
v2
print(m)
```

```
0 0 7
0 3 0 0
0 0 7
0 3 0 0
6
2
0 0 6
0 2 0 0
1
1 0 6
3
3 2 1 0
" 3 2 1 0\n 0 0 0 0\n 0 0 6 0\n"
```

```
3 2 1 0
0 0 0 0
0 0 6 0
```

2. To get a single row Matrix from a Vector, just call:

   var m2 = v1.matrixview
   Also, if the Matrix is single row or single column, then
   var v3 = m3.vectorview will return a Vector which is the only single row or column of the Matrix.

```
v1
var m2 = v1.matrixview
m2[0, 1] = 3
v1
v1[0] = 7
v1
m2
```

```
1 0 6
1 0 6
3
1 3 6
7
7 3 6
7 3 6
```

3. If a Matrix just has one single row or column, then the matrixInstance.vectorview will return a Vector

Design pattern: composition and façade

As previous mentioned, the hard part is the conversion between Vector and Matrix. That's way I choose to use `UnsafeMutablePointer<Vector<T>>`.
To get a Vector from a Matrix, simply return the specified row or column vector data:
return self.dataRow[index].memory or return self.dataColumn[index].memory which is a Vector

Here, I am using façade: The dataRow and dataColumn is not the real matrix data, but rather a return value from _dataRow or _dataColumn depend on whether the Matrix is transposed or not. It can keep me focus on the operation on the columns or rows while regardless of the transpose state of Matrix.

```
private var dataRow: Array<UnsafeMutablePointer<Vector<T>>> {
  get {
    if !self._isTransposed {
      return self._dataRow
    } else {
      return self._dataColumn
    }
  }
}

private var dataColumn: Array<UnsafeMutablePointer<Vector<T>>> {
  get {
    if !self._isTransposed {
      return self._dataColumn
    } else {
      return self._dataRow
    }
  }
}
```

Part two: Test Class by Jialin Yang

Test class API:

The test class applies composition in testing the Matrix and Vector class. To simplify its complexity, we design a template for testing all the matrixes and vectors that share the same protocols. When the objects of the test class are initialized, it will automatically load all the unit test cases that test every simple operation in the protocols. The following is a short description on the details.

Constructor and private test cases

test_matrix_vector<Type>(row, col, seed):
Eg1. var t1 = test_matrix_vector<Int>(row:100,col:100,seed:1000)

Row define the matrix's row, col define the matrix's column. And the vector's size is row*col which contains the same number of elements in matrix

Seed is a parameter for random number generator, which specifies the maximum variance in the test data set.

If row, col and seed are not defined, it will use a random number generator to pick up several random numbers, which ranges from 1 to 5.

During the initialization of the constructor, it will generate some random data pattern. Then, several test cases, related to the matrix and vector, are automatically loaded.

Test cases:

1.createthendestroy():

For example, it created matrix and vector, check the row/col/size matched with the user setting. Then, every element in matrix/vector will be assigned to a new value. And the element retrieved from matrix/vector must match with the expected one. Otherwise, the related error counter will increase by 1 and print some error messages.

2.simplemathMatrix,simplemathVector:

SimplemathMatrix/simplemathVector are math tests, which contains all the +,-,* operation and +,-,*,/ scaler. In this process, it only tests several simple math operation. when the test class is initialized, it will generate some sample data for test. These data are used as function input as well as expected output for comparison. Certainly, there are a few counters that record the possible errors. The matrix data input and expected output comes from initialization of testmatrixandvector class.

3.Testcopy:

Test the copy() interface. It makes sure that its new matrix/vector is not a reference to the original matrix/vector and every element in the new matrix and vector are the same as that in the original one.

4.Testdot:

Test dot operation of the vector. Both vectors must be of the same size. And if the expected output fails to match with its output, it will report an error.

5.Testtranspose:

The matrix's transpose operation will return the original matrix. Therefore, when I try to make sure the transpose operation works, I have to copy the original one, transpose the original matrix and compare the transposed matrix with the copied matrix.

6.testMVconversion:

In this test, it converts one-row matrix and one-column matrix into vector. And it converts a vector into matrix.

7.testVfromM:

This function tests the column vector and row vector from the matrix. If the element value matches, there will be no error output. The returned value is a reference to the elements in the matrix. In other words, if the returned vector changes its value, the element in the matrix will also change.

8.integrated_test

All the test cases above belong to unit tests. In this function, we would include integrated usages of different operations.

9.Gen_report:

Every test case above contains a list of error counters that record related error. In this function, it prints out all the error messages at the end. If all the test cases pass, it will print messages that indicate everything is working well.

Public interface(test manually):

1. testmemoryleakage:

This is an internal function, it is a while loop that never stop. Call the methods in main and open the activity monitor. If the memory used by matrixvector process never rockets up, it means there is no memory leakage.

2. checkborder_MRH(flag:Int):

Check the border of matrix and vector. It contains one flag.
Flag:1, test negative number of row and column when it visits matrix element
Flag:2, test positive number, which is larger than the maximum row and column number of matrix, when it visits matrix element
Flag:3, test positive number, which is larger than the maximum size of vector, when it visit matrix element
Flag:4, test negative number when it visits vector element.

Extension on MatrixData:

During test, it turn out that Complex and Fraction have not != operation and data-type casting, which becomes an obstacle to build an universal test case. Thus, I extend the original protocol and make sure that all the operations could apply to Int\double\float\complex\fraction.

- protocol extension != : this is an effort to make comparison between matrix/vector element easier. Int/Double/float has defined this operation in the library standard. Therefore, we just extend it in Complex and Fraction Class.

- protocol extension cast: Int/Float/Double has an internal data cast function. However, Complex and Fraction do not support data type casting. To generate test data patterns for matrix and vector objects, we make an extension on data casting.

In conclusion, this project is quite helpful in learning swift and cooperation. We work together, discuss bugs and search for solutions. The implementation of matrix&vector class and test class gives us lots of fun.