Kotlin Vectors and Matrices

COMP40009 - Computing Practical 1

29th January – 2nd February 2024

Aims

- To explore data classes, operator overloading and infix operators in Kotlin
- To gain experience writing extension methods
- To implement matrix and vector operations, making maximal use of the functional programming features that Kotlin offers
- To use generics and function types to implement vectors and matrices over abstract addition and multiplication operations

Introduction

Linear algebra is fundamental to many areas of mathematics and computer science. Linear algebra involves computing with vectors and matrices over the real numbers, which can be approximated using floating-point data types in computer programs. The first part of this lab involves writing Kotlin classes to represent vectors and matrices over the `Double` data type, and using operator overloading to make it convenient to work with these classes using the regular additional and multiplication operators.

Matrices and vectors can be defined more generally over elements of an *algebraic ring*. The second part of this lab involves writing more general routines for matrix and vector operations that work with user-provided addition and multiplication functions.

`A note on efficiency:` This lab is not about writing highly-efficient matrix multiplication algorithms. Instead, the lab uses matrices and vectors as a vehicle for practicing the use of various Kotlin features, and practicing functional programming in Kotlin in particular.

Getting started

As per the previous exercises, use `git` to clone the repository with the skeleton files for this exercise, via

```
git clone git@gitlab.doc.ic.ac.uk:lab2324_spring/kotlinmatrix_username.git
```

Challenge

You should try to implement this exercise without explicitly using loops. Instead, you should use functional programming features such as map and reduce. (It is fine to use loops in test cases.)

The idea of this challenge is to get you even more familiar with Kotlin's support for functional programming, and to help you to think about the pros and cons of working with functional programming constructs vs. working with loops.

If you are not so confident at functional programming, a good approach could be to get things working using loops, and then to gradually rewrite loopy code to use a more functional style.

Some useful Kotlin functions that may help you in this challenge are:

- `map`
- `zip`
- `reduce`

What to do: Part 1

The source files that you create should be under `src/main/kotlin/algebra/real` (where `real` refers to "real number"). If you wish to create additional test files, you can create them under `src/test/kotlin/algebra/real`.

Implementing and testing a vector class

Write a data class, `Vector`, in a new file, `Vector.kt`. Your class should represent a vector of (floating-point approximations of) real numbers, where you should represent (an approximation of) a real number using `Double`.

- A `Vector` should be constructed from a non-empty list of `Double`s. Providing an empty list should lead to an `IllegalArgumentException`.
- It should not be possible for a client of `Vector` to directly access the property that stores the list of vector elements.
- A vector should provide read access to a `length` property that yields the length of the vector.
- If `v` is a vector, it should be possible to write `v[i]` to get the component of `v` at index `i`, where vectors are indexed starting from 0. If `i` is out-of-bounds, an `IndexOutOfBoundsException` should be thrown.
- Given two vectors `u` and `v` of the same length, it should be possible to write `u + v` to obtain the sum of `u` and `v`. If the lengths of these vectors are different, attempting to add them should lead to an `UnsupportedOperationException` being thrown.
- If `x` is a `Double` (a scalar) and `v` is a vector, it should be possible to write both `x * v` and `v * x` to yield a vector that represents `v` scaled by `x`—i.e., the vector containing every component of `v` multiplied individually by `x`.
- Given two vectors `u` and `v` of the same length, it should be possible to write `u dot v` to obtain the dot product of `u` and `v`. The dot product of `u` and `v` is the sum of each individual element of `u` and `v` multiplied together, pairwise. For example, the dot product of $(1, 2, 3)$ and $(4, 5, 6)$ is $1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 32$. Attempting to compute the dot product of vectors whose lengths do not match should lead to an `UnsupportedOperationException` being thrown.
- A vector should be represented as a string via an opening '`(`' character, followed by the string representation of each `Double` in the vector, with elements separated by a comma and a space, followed by a closing '`)`' character.

The `VectorTests.kt` file contains a number of test cases for the `Vector` class, which you should uncomment.

Make sure your solution passes these tests.

Use the IntelliJ "Run 'VectorTests' with Coverage" option to see to what extent your tests achieve coverage of the `Vector` class.

Add additional tests that improve coverage, ensuring that your tests achieve at least 90% statement coverage of `Vector`. Make sure that, as well as covering extra code, your tests include assertions that carefully check that meaningful properties about `Vector` instances hold. High code coverage is a minimum requirement for a good test suite, but it is easy to fall into the trap of writing tests that exercise lots of code, but that do not check that this code actually behaves correctly. (Such tests are sometimes called "blind tests" because they fail to identify errors in portions of code despite the fact that they exercise the code.)

Implementing and testing a matrix class

Write a data class, `Matrix`, in a new file, `Matrix.kt`. Your class should represent a matrix of (floating-point approximations of) real numbers, where `Double` should be used to approximate a real number.

- A `Matrix` should be constructed from a non-empty list of `Vector`s, where each vector represents a row of the matrix. Providing an empty list should lead to an `IllegalArgumentException`. The rows that are used to construct a matrix should all have the same length; if there is a length mismatch then an `IllegalArgumentException` should be thrown.
- A `Matrix` should provide read access to properties `numRows` and `numColumns` that yield the number of rows and columns of the matrix, respectively.
- It should not be possible for a client of `Matrix` to directly access the property that stores the list of row vectors.
- If `m` is a `Matrix`, it should be possible to write `m[i]` to get row `i` of the matrix as a `Vector`. Equivalently, a `getRow(i)` method should be provided to retrieve row `i`, and analogously a `getColumn(i)` method should be provided to retrieve column `i` of the matrix as a vector. It should be possible to write `m[i, j]` to get the element at column `j` of row `i` from the matrix. In all of these methods for getting rows, columns or elements from a matrix, if an out-of-bounds index is supplied, an `IndexOutOfBoundsException` should be thrown. As with the `Vector` class, indexing should start from 0.
- Given two matrices `mat1` and `mat2` that have the same number of rows and same number of columns, it should be possible to write `mat1 + mat2` to yield their elementwise sum. If the number of rows and/or columns does not match, an `UnsupportedOperationException` should be thrown.
- Given a matrix `mat1` with n columns and a matrix `mat2` with n rows, it should be possible to write `mat1 * mat2` to yield their product—i.e. the result obtained by applying standard matrix multiplication to these matrices. If the number of columns of `mat1` does not match the number of rows of `mat2` an `UnsupportedOperationException` should be thrown. Your implementation of matrix multiplication should make use of the `dot` function defined on vectors.
- Given a matrix `mat` and a floating-point number (a `Double`) `s` (for *scalar*) it should be possible to write `s * mat` or `mat * s` to obtain the matrix containing every element of `mat` multiplied individually by `s`.
- A matrix should be represented as a string as follows:
  - Each row of the matrix should start with "`[ `" and end with "` ]`".
  - In between these square braces, string representations of elements of the row should occur, separated by white-space.
  - White-space should be added between entries such that each column of the matrix is right-justified.

The `MatrixTests.kt` file contains a number of test cases for the `Matrix` class, which you should uncomment.

Make sure your solution passes these tests.

Use the IntelliJ "Run 'VectorTests' with Coverage" option to see to what extent your tests achieve coverage of the `Matrix` class.

Add additional tests that improve coverage, ensuring that your tests achieve at least 90% statement coverage of `Matrix`. Again, make sure that your new tests include assertions that carefully check that meaningful properties about `Matrix` instances hold.

What to do: Part 2

The source files that you create should be under `src/main/kotlin/algebra/generic`. If you wish to create additional test files, you can create them under `src/test/kotlin/algebra/generic`.

The aim of this part of the exercise is to give you practice working with generic classes, and using function objects.

Matrices and vectors are usually defined over real numbers. However, matrices and vectors can be defined much more generally, over an algebraic structure called a *semiring*.

A semiring is a set equipped with two operations:[1]

- an "addition" operation that is *associative* ($(x + y) + z = x + (y + z)$ for all x, y, z) and *commutative* ($x + y = y + x$ for all x, y);
- a "multiplication" operation that is *associative* ($x \cdot (y \cdot z) = (x \cdot y) \cdot z$ for all x, y, z) but not necessarily commutative.

The words "addition" and "multiplication" are written in quotes here to emphasise that they can be any operations that satisfy these requirements—they need not be the familiar numeric addition and multiplication operations, and in fact the set that underlies a semiring need not be a set of numbers.

If $(S, +, \cdot)$ is a semiring, where S is the underlying set and $+$ and $\cdot$ are the addition and multiplication operations, a vector over S is just a fixed-length sequence of elements of S. A vector $(x_1, x_2, \ldots, x_n)$ (where each x_i is drawn from S) can be left-multiplied by a scalar s from S to yield $(s \cdot x_1, s \cdot x_2, \ldots, s \cdot x_n)$, or right-multiplied to yield $(x_1 \cdot s, x_2 \cdot s, \ldots, x_n \cdot s)$; the resulting vectors may be different because $\cdot$ need not be commutative. Two vectors of the same length, $(x_1, x_2, \ldots, x_n)$ and $(y_1, y_2, \ldots, y_n)$ can be added to yield the vector $(x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n)$, where $+$ denotes the semiring addition operation. The dot product of two vectors can be defined in the same way as for vectors of real numbers, except that the addition and multiplication operations of the semiring are used.

Similarly, matrices can be defined over a semiring, and matrix addition, matrix multiplication, and multiplication of a matrix by a scalar on the left or right, can be defined in exactly the same way as for matrices over real numbers, except that the addition and multiplication operations of the semiring are used.

As an example, let $\mathbb{Z}_5$ be the set $\{0, 1, 2, 3, 4\}$, and let $\oplus$ and $\otimes$ denote addition and multiplication modulo 5, respectively, so that e.g. $4 \oplus 4 = 3$ (because $4 + 4 = 8$ and $8 \bmod 5 = 3$), and $2 \otimes 3 = 1$ (because $2 \cdot 3 = 6$ and $6 \bmod 5 = 1$). Then $(\mathbb{Z}_5, \oplus, \otimes)$ is a semiring.[2]

[1] The following is not a complete definition of a semiring, but is enough for the purposes of this lab. Furthermore, some authors define semirings to include identity elements for addition and multiplication, which are not required in this lab exercise

[2] In fact, it is a *ring*, which satisfies a larger set of axioms than the more general semiring structure. For any positive integer N, $\mathbb{Z}_N$, with addition and multiplication defined modulo N, is a ring.

Performing addition and multiplication in $\mathbb{Z}_5$, the vectors $(0, 2, 4)$ and $(1, 4, 1)$, whose elements come from $\mathbb{Z}_5$, can be added to yield $(1, 1, 0)$, and the dot product of these vectors is $0 \otimes 1 + 2 \otimes 4 + 4 \otimes 1 = 0 + 3 + 4 = 2$. Similarly, here is an example of matrix multiplication over matrices of elements from $\mathbb{Z}_5$, using addition and multiplication modulo 5:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 3 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} (1 \otimes 1) \oplus (2 \otimes 3) \oplus (3 \otimes 0) & (1 \otimes 2) \oplus (2 \otimes 4) \oplus (3 \otimes 2) \\ (4 \otimes 1) \oplus (3 \otimes 3) \oplus (2 \otimes 0) & (4 \otimes 2) \oplus (3 \otimes 4) \oplus (2 \otimes 2) \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix}$$

Your task is now to implement generalised versions of your vector and matrix classes that work with arbitrary semirings.

Implementing a generic vector class

In the `algebra/generic` package make a Kotlin file, `Vector.kt` that contains a `Vector` class representing a generic vector.

This should be just like the `Vector` class you made earlier, except:

- The class should be generic with respect to some type, T say
- A `Vector` should be constructed from three arguments:
  - A function of type $(T, T) \to T$ representing addition;
  - A function of type $(T, T) \to T$ representing multiplication;
  - A list of elements of type T representing the contents of the vector (analogous to the list of doubles that was previously used to construct a vector)
- The operations on `Vector` should be modified to work with elements of type T instead of with `Double`s, and to use the addition and multiplication operations that were passed to the vector on construction whenever it is necessary to add or multiply elements of type T.

Your generic `Vector` class should support generalised versions of all the operations provided by your previous `Vector` class. The same requirements about when exceptions should be thrown apply. It is probably a good idea to use a copy of your previous `Vector` class as a starting point for implementing your generic `Vector` class.

Implementing a generic matrix class

In the `algebra/generic` package make a Kotlin file, `Matrix.kt` that contains a `Matrix` class representing a generic vector.

This should be just like the `Matrix` class you made earlier, except:

- The class should be generic with respect to some type, T say
- A `Matrix` should be constructed from three arguments:
  - A function of type $(T, T) \to T$ representing addition;
  - A function of type $(T, T) \to T$ representing multiplication;
  - A list of vectors whose elements have type T representing the rows of the matrix (analogous to the list of doubles that was previously used to construct a vector)
- The operations on `Matrix` should be modified to work with elements of type T instead of with `Double`s, and to use the addition and multiplication operations that were passed to the matrix on construction whenever it is necessary to add or multiply elements of type T.

Your generic `Matrix` class should support generalised versions of all the operations provided by your previous `Matrix` class. The same requirements about when exceptions should be thrown apply. It is probably a good idea to use a copy of your previous `Matrix` class as a starting point for implementing your generic `Matrix` class.

A problem with this design that you do not need to try to solve here is that it is possible to construct a generic matrix from vectors that disagree with each other and/or the matrix under construction with respect to the addition and multiplication functions with which they are equipped.

Creating an algebra factory

A problem with these generic vector and matrix classes is that each and every vector and matrix that is created must have an addition and multiplication argument passed to it. This gets very cumbersome when defining matrices, as these functions must be passed to the overarching matrix constructor, as well as to the constructors of the vectors that comprise each matrix row. There is also the risk, discussed above, that vectors and matrices over the same underlying type could be constructed with inconsistent addition and multiplication functions.

To mitigate this problem, create an `AlgebraFactory` class in a new Kotlin file, `AlgebraFactory.kt`. An `AlgebraFactory` should be generic with respect to a type T, and should be constructed with two functions that represent addition and multiplication on this type.

Further, an `AlgebraFactory` should provide two methods:

- `makeVector`, which takes a list of elements of type T and returns a `Vector<T>` constructed from the addition and multiplication operations of the factory together with the given list of elements;
- `makeMatrix`, which takes a list of lists of elements of type T and returns a `Matrix<T>` constructed from the addition and multiplication operations of the factory together with a list of vectors, one constructed from each of the given lists of elements.

Testing your solution

Under the `test/kotlin/algebra/generic` you will find a number of files containing commented-out test cases for generic vectors and matrices, which make heavy use of the `AlgebraFactory` class.

Un-comment these and debug your solution until they pass.

Enhance this set of tests to achieve at least 90% code coverage of the classes under `main/kotlin/algebra/generic`. You can, in part, adapt tests that you wrote for vectors and matrices over real numbers for this purpose.

Extensions

These parts are optional. If you have the above working you might like to try one or more of these.

Allowing a variable number of arguments in vector and matrix construction

It is a bit cumbersome to always have to pass a list to the constructors of `Vector` and `Matrix`.

Read about "vararg" functions in Kotlin and write secondary constructors for these classes that work with a variable number of arguments and avoid the need to wrap vector elements or matrix rows in a list.

Providing iterators for vectors and matrices

If iterators have not been covered by the time you undertake this extension you may want to do some reading online about the topic.

Provide an `iterator` operator for `Vector` that returns an `Iterator<Double>` object that allows for iteration over the vector's elements using a `for (v in vector)` style loop. Express this iterator concisely using an inner class or, better still, an anonymous object.

Similarly, provide an `iterator` operator for `Matrix` that allows for iteration over the rows of the matrix.

Also try equipping your generic vector and matrix classes with `iterator` operators; this should be no harder than providing iterators for the concrete versions of these classes.

Experimenting with additional semirings

Write a class representing integers modulo N for a given N. Overload addition and multiplication to work appropriately for this class. Write unit tests to confirm that your generic classes for vectors and matrices work for vectors and matrices over integers modulo N.

Read online about the *tropical semiring*: `https://en.wikipedia.org/wiki/Tropical_semiring`. Write a class to represent an element of the tropical semiring, using `Double` to approximate a real number and to capture the value $+\inf$. Write unit tests to confirm that your generic classes for vectors and matrices work for vectors and matrices over the tropical semiring.

Note: your repository containing the skeleton for this lab can be found at** `https://gitlab.doc.ic.ac.uk/lab2324_spring/kotlinmatrix_username`**. As always, you should use LabTS to test and submit your code.

Assessment

In general, the assessment for laboratory exercises uses the following scheme:

```
F - E: Very little to no attempt made.
       Submissions that fail to compile cannot score above an E.

D - C: Implementations of most functions attempted;
       solutions may not be correct, or may not have a good style.

B:     Implementations of all functions attempted, and solutions
       are mostly correct. Code style is generally good.

A:     There are no obvious deficiencies in the solution or
       the student's coding style. In addition, there is
       evidence of productive testing.

A*:    As for an A -- plus the student has done additional work
       beyond the basic spec, e.g. by considering (and clearly
       commenting) interesting variations or extensions to the
       given functions; e.g. based on their own research.
```