# COSC342: Lab 02

Matrices and Vectors in C++

March 14th 2016

## 1 Introduction

In this lab you will practice using a simple Matrix library in C++. Online documentation for this library is available at `http://www.cs.otago.ac.nz/cosc342/docs/matrices/index.html`. The library provides two classes, `Matrix` and `Vector`, and `Vector` is a subclass of `Matrix`.

Each class has a header file, `Class.h`, and an implementation, `Class.cpp`. There is also a sample file, `matrixTest.cpp` illustrating some basic operations, and a `Makefile` which builds the project.

You can get the source code from
`/home/cshome/coursework/342/pickup/labs/lab02-Matrices`
It is easiest to work from the command line for this lab. Start by making a directory to store your COSC342 work, if you have not already. Open a terminal, and enter the following commands

```
cd ~/Documents
mkdir cosc342
cd cosc342
```

If you have already made a directory to store your COSC342 work, change to that directory. Next we'll get a copy of the lab files

```
cp -r /home/cshome/coursework/342/pickup/labs/lab02-Matrices .
```

The `-r` option copies files recursively, and the dot, `.`, at the end means copy to here.

## 2 Building Projects with CMake

There are a number of different toolchains for C++ on the lab machines. The two main options are to compile code with Makefiles from the command line or to use XCode. To support these we'll be using a tool called CMake, which creates projects for different platforms.

In your `lab02-Matrices` directory there is a file called CMakeLists.txt which describes the project that needs to be made. Lets take a look inside that:

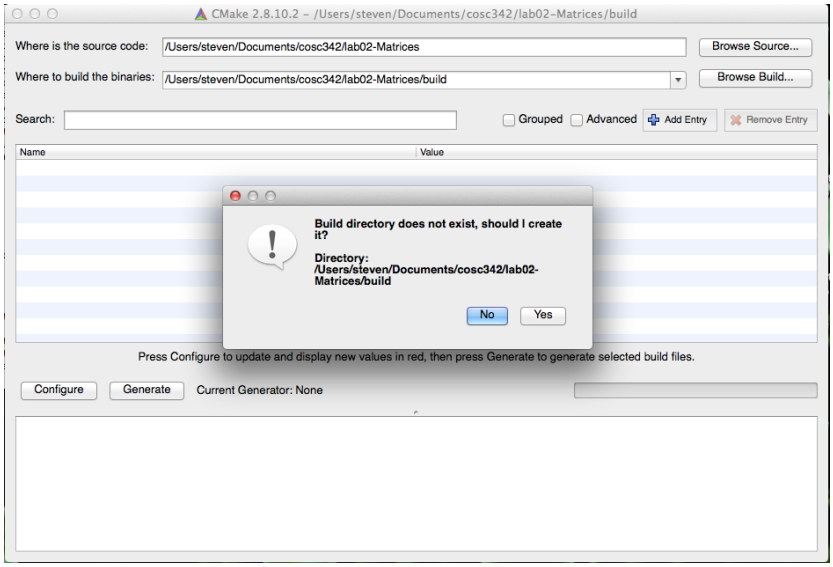
```
project(Matrices)
add_executable(Matrices
  matrixMain.cpp
  Matrix.h
  Matrix.cpp
  Vector.h
  Vector.cpp
)
```

This is a very simple CMake file, which declares that we're making a project called `Matrices`. This is also the name of the executable program we'll be building, and that executable depends on the listed `.cpp` and `.h` files.

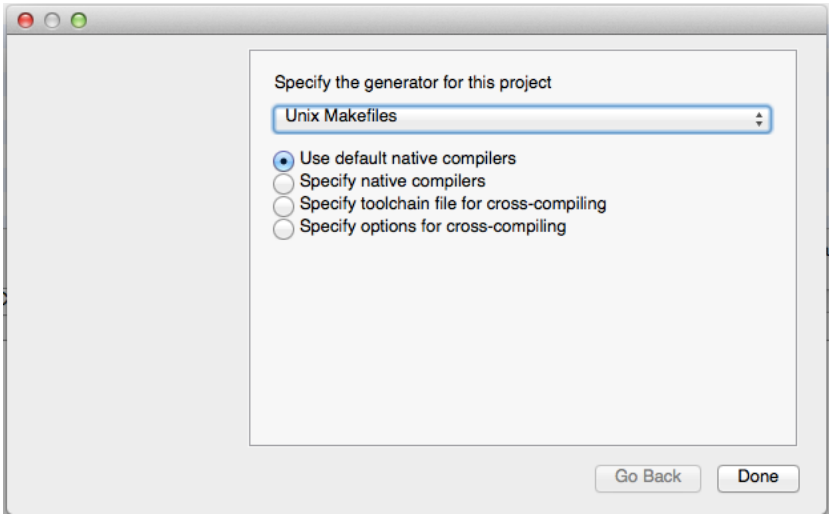
To set up the project, run the program called `CMake`, which is in the Applications directory. If you can't find it press `command-space` and type "CMake" to search for it. CMake asks you two simple questions – "Where is the source code" and "Where to build the binaries". The source code is where you copied it to, so browse to that directory, something like
`/home/cshome/a/astudent/Documents/cosc342/lab02-Matrices`

Where to build the program is up to you, but the conventional thing to do is to add `/build` to the source code location, so something like
`/home/cshome/a/astudent/Documents/cosc342/lab02-Matrices/build`

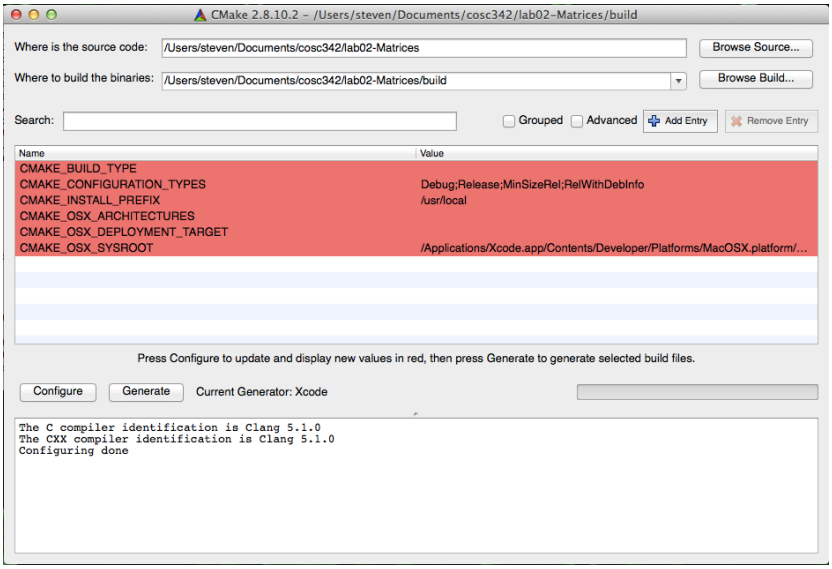
Press the *Configure* button. You will probably get a prompt to create the build directory, click *Yes*.




You'll next be prompted to choose a toolchain to build. You can use whatever is installed on the machine you are using, but for the labs we'll assume you're using either *Unix Makefiles* or *XCode*. Choose your preferred environment from the drop-down, leave the other selection as *Use default native compilers*, and press *Done*.




CMake looks for the relevant tools, and checks that they are available, then updates its settings. You'll see a number of new settings added, which are highlighted in red. You can edit these if you need to, but you don't. So don't.




Once the program is configured, press *Generate* to create the Makefiles or XCode project in the `build` directory.

## 2.1 Building with Makefiles

If you are using Unix Makefiles, you'll need to go back to the terminal. Assuming you're in the `cosc342` directory you made earlier, you'd go to the build directory

and compile the program with

```
cd lab02-Matrices/build
make
```

You can then run the program with

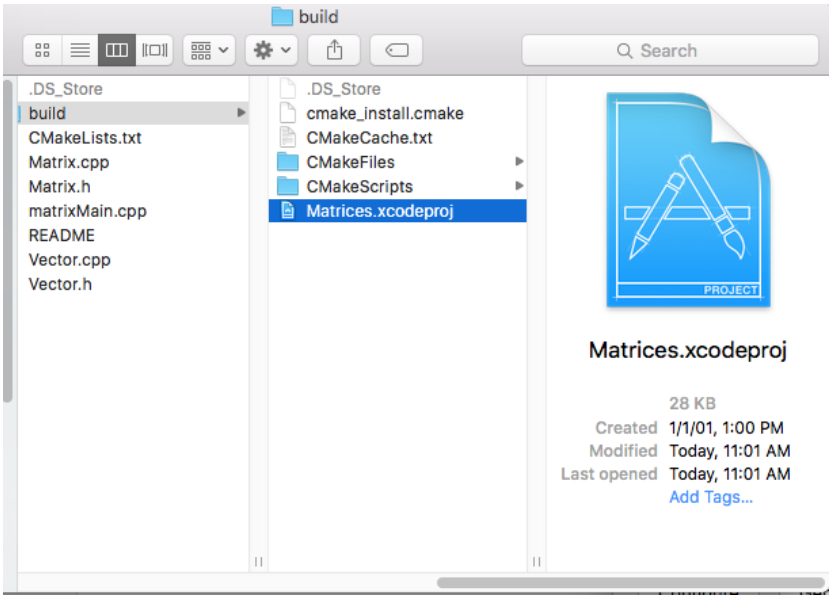
```
./Matrices
```

And should see the following output:

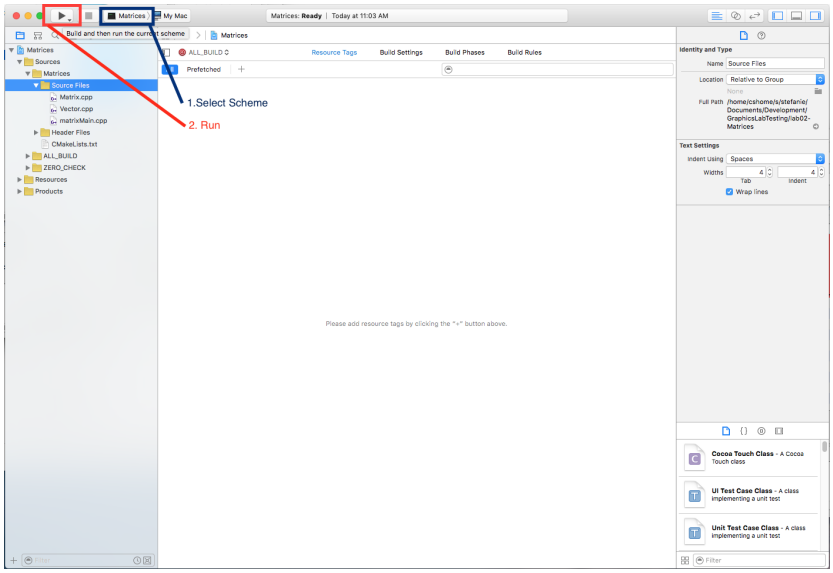
```
The Matrix A is:
1       0       0
0       0.5     -1
2       2       0

The Vector v is:
1
2
3

u = A*v is:
1
-2
6
```

## 2.2 Building with XCode

Open the XCode project file from your build directory.



Select the Matrices scheme and press the run button.




# 3 Working with the Library

This may be your first time working with C++, but the sample file should give you enough information to get started with the `Matrix` and `Vector` classes. We'll go through some basics, do some exercises, then you can go back to look at the classes and their implementations in more detail.

`Matrix` and `Vector` objects are most commonly made with constructors that take the size as parameters:

```
Matrix A(3,2); // A 3x2 matrix
Vector v(3);   // A 3-vector
```

If no arguments are given then a $1 \times 1$ `Matrix` or `Vector` is made:

```
Matrix A; // A 1x1 matrix
Vector v; // A 1-vector
```

You can access the elements of a `Matrix` or `Vector` with a function-style syntax. Following C/C++ convention, indices start from 0 rather than 1.

```
Matrix A(3,3); // A 3x3 matrix
A(0,0) = 1;    // Set top left element to 1
A(1,2) = 2;    // Set 3rd element of 2nd row to 2
A(2,2) = 3;    // Set bottom right element to 3
```

C++ uses *streams* for output, and the standard output stream, `std::cout`, is defined in the `iostream` header. The `<<` operator is used to send information to the stream, so "Hello World" in C++ looks like this:

```
#include <iostream>

int main () {
  std::cout << "Hello, World" << std::endl;
  return 0;
}
```

The special constant `std::endl` ends a line and flushes the stream. You can also use `<<` with `Matrix` and `Vector` objects:

```
std::cout << A << std::endl;
```

Vectors are a single column, so it is often more convenient to output their transpose:

```
std::cout << v.transpose() << std::endl;
```

C++ supports *operator overloading*, which we'll discuss in detail later. For now all you need to know is that this means you can write fairly natural arithmetic expressions using `Matrix` and `Vector` objects:

```
Matrix A(3,3);
Vector v(3), u(3);
u = A*v;
```

# 4 Exercises

These exercises are a subset of last week's tutorial, so you should be able to check your answers against them. For each exercise you should write code to evaluate the required expressions, and display the results in the console with the `std::cout` stream.

## 4.1 Vector Arithmetic

Let $\mathbf{u} = (1, -3, 2)$ and $\mathbf{v} = (3, 2, 0)$. Evaluate the following:

1. $2\mathbf{u}$
2. $\mathbf{u} + \mathbf{v}$
3. $\mathbf{u} - \mathbf{v}$
4. $2\mathbf{u} - 3\mathbf{v}$

## 4.2 Dot and Cross Products

`Vector` objects have methods `dot` and `cross` which take another `Vector` as a parameter and return the appropriate product.

Evaluate the following dot products:

1. $(1, 2, -2) \cdot (1, 2, -2)$
2. $(1, 2, -2) \cdot (-2, 2, 1)$
3. $(1, 2, -2) \cdot (2, 3, 1)$

Evaluate the following cross products:

1. $(1, 2, -2) \times (-2, 2, 1)$
2. $(1, 2, -2) \times (-2, -4, 4)$
3. $(1, 2, -2) \cdot ((1, 2, -2) \times (-2, 2, 1))$

### 4.3 Matrix Arithmetic

Evaluate the following:

1. $2\begin{bmatrix} 1 & -2 \\ 3 & 4 \end{bmatrix} - \begin{bmatrix} 3 & 2 \\ 4 & 1 \end{bmatrix}$
2. $\begin{bmatrix} 2 & 1 \\ 2 & 2 \end{bmatrix}\begin{bmatrix} 1 & -2 \\ -4 & 1 \end{bmatrix}$
3. $\begin{bmatrix} 1 & -2 \\ -4 & 1 \end{bmatrix}\begin{bmatrix} 2 & 1 \\ 2 & 2 \end{bmatrix}$

### 4.4 Matrices and Vectors

These following exercises make use of the matrix $A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 2 \end{bmatrix}$, and the vectors $\mathbf{u} = (1, 2, -2)$ and $\mathbf{v} = (-2, 2, 1)$. Compute the following:

1. $A\mathbf{u}$
2. $\mathbf{u}\mathbf{u}^\mathsf{T}$
3. $\mathbf{u}^\mathsf{T}\mathbf{v}$

## 5 Inside the Library

This may be your first time working with C++, but many of the concepts should be familiar to you from your C and Java experience. For the most part you should just be able to follow along for now, but here are some notes that might help you make sense of things. I'd recommend skimming over these at first, having a go at writing some code, and then coming back for a closer look. There are also plenty of comments in the code which should help you get started, as well as the online documentation.

Classes in C++ are defined using a similar syntax to Java (technically Java borrowed a lot of syntax from C++), and each class typically has two files associated with it – a header file (e.g.: `Matrix.h`) and an implementation file (e.g.: `Matrix.cpp`). If you open `Matrix.h` in a text editor, you will see the definitions (or signatures) of many methods, but no implementation details. These details are all in `Matrix.cpp`, and the header file just gives the interface.

Some of the syntax may be unfamiliar to you, in particular *operator overloading*. A good example is the multiplication method for Matrices:

```
friend Matrix operator*(const Matrix& lhs,
                        const Matrix& rhs);
```

The short version of what this does, is allow us to write code like

```
Matrix A(3,3), B(3,4), C(3,4);

// Code to fill A and B with values

C = A*B;
```

This code creates a $3 \times 3$ matrix called `A`, a $3 \times 4$ matrix called `B`, then multiplies them together, storing the result in `C`.

In more detail:

- `friend` Tells us that even though this function is not exactly a method of the `Matrix` class it can access the internal elements of `Matrix` objects as if it was.
- `Matrix` Tells us that the result of multiplying two `Matrix` objects together is another `Matrix` object.
- `operator*` Tells us that this function provides an implementation for the multiplication operator. Basically, you can use `A*B` as a shorthand for `Matrix::operator*(A,B)`.
- Finally the parameters are `Matrix` objects. These are passed by reference (`&`) to avoid copying complex objects, and the `const` keyword provides a guarantee that if you call `A*B`, `A` and `B` will remain unchanged.
- The parameter names, `lhs` and `rhs`, are supposed to remind you that they are on the *l*eft *h*and *s*ide and *r*ight *h*and *s*ide of the `*` operator respectively.

Operator overloading is a very powerful tool for providing concise and intuitive interfaces to mathematical objects like Matrices and Vectors. It also lets us provide a few other useful things, such as overloading the assignment operator, which is also used in the line `C = A*B;`, above. Another useful pair of overloads are the stream operators, `<<` and `>>`, which are used in C++ to provide a powerful syntax for input and output through *streams*. The classic `Hello World` program in C++ is

```
#include <iostream>

int main () {
  std::cout << "Hello World" << std::endl;
  return 0;
}
```

The `iostream` library provides input and output streams, one of which is `std::cout` – this is an output stream which goes to the console (unless redirected by the operating system). The `std::` part just says that the stream `cout` exists in the standard namespace, which prevents cluttering the global namespace with lots of stuff. In this context the `<<` operator provides the ability to send information to the stream. In this case, we are sending a string, and this gets printed in the console.

Overloading `<<` means that we can just send `Matrix` operators to these streams:

```
std::cout << "Matrix A is:" << std::endl;
std::cout << A << std::endl;
```

One final way that operator overloading is used is to access elements of `Matrix` objects. This is done with the function call operator, `operator()`. There are two versions of this, and here are their signatures:

```
double& operator()(size_t row, size_t col);
const double& operator()(size_t row, size_t col) const;
```

These let us write code like

```
Matrix A(3,3);
double x = A(1,2);
```

On the first line, the brackets tell us that the parameters to send to the constructor are 3 and 3. On the second line, however, the brackets are calls to the function call operator, which returns the corresponding element of the matrix (indexed from 0, of course, since this is C++). So the second line gets the value from the second row and third column of `A`, and stores it in `x`.

Note that the function call operators return a *reference*, this means that you can also assign values to `Matrix` elements like this:

```
A(2,1) = 4.5;
```

This is also why there are two versions of the function call operator. C++ makes a lot of use of the idea of `const` correctness, which is using the keyword `const` to tell the compiler when things should not be altered. If `A` is `const`, then we want to allow read access to its elements, but don't want to be able to write to them. The second version of the function call operator tells us that if we call `operator()` on a `const Matrix` then we get back a `const double&`, and so can't assign a new value to it.

Have a look at the source code for the `Matrix` and `Vector` classes. Notice that the output of the operators on `Vector` and `Matrix` objects might be a

`Vector`, a `Matrix` or a `double` (a scalar). This helps to catch errors at compile time, but other errors (such as multiplying `Matrix` objects whose interior dimensions don't agree) are caught at run time with `asserts`.

Have a look through the code and see how it works. Ask us if you have any questions.

# 6 More Advanced Matrix Libraries

This `Matrix` code is designed to be easy to use and reasonably easy to understand. More advanced C++ programming methods allow for very efficient implementations, but they are much harder to understand. One such library is Eigen, which can be found at `http://eigen.tuxfamily.org`.

Eigen uses a feature C++ called *templates*, which are a bit like Java generics (again, Java stole this from C++) but more powerful. Matrices in Eigen are declared by giving their size and type as template parameters:

```
// A 3x4 matrix of doubles
Eigen::Matrix M<3, 4, double>
```

The use of templates allows the compiler to do a lot of reasoning about the size and type of matrices. It means that the compiler can determine automatically that the result of multiplying a $3 \times 4$ matrix by a $4 \times 2$ matrix is a $3 \times 2$ matrix. If you try assigning this to an incorrectly sized matrix, then you get an error at compile time rather than runtime.

Templates also allow for a crazy C++ technique called *template metaprogramming*. It turns out that the template structure of C++ is Turing complete, so you can do all sorts of things with it. This lets the compiler realise when it is dealing with small matrices and unroll loops etc. to make faster code. Other advanced features of Eigen like lazy evaluation, avoidance of temporary objects, and vectorisation make the code very fast, while keeping a fairly simple interface.

For our purposes, however, code that you can understand is better than code which is very fast.