CS319: Scientific Computing

Projects; Vectors and Matrices (DRAFT)

Dr Niall Madden

Week 9: 12 + 14 March, 2025

Slides and examples: https://www.niallmadden.ie/2425-CS319

0. Outline

1. Projects!

Notes for this part are at:
https://www.niallmadden.ie/2425-CS319/2425-CS319-Projects.pdf

2. Review of classes

class

In C++, we define new class with the `class` keyword.
An instance of the class is called an "*object*".
A `class` combines by data and functions (called "methods").

Within a class, code and data may be either

- **Private**: accessible only to another part of that object, or
- **Public**: other parts of the program can access it.

Roughly,

- keep data elements `private`,
- make function elements `public`.

2. Review of classes

The basic syntax for defining a class:

```
class class-name {
private:
    ...     // private functions and variables
public:
    ...     // public functions and variables
};
```

class-name becomes a new object type—one can now declare objects to be of type *class-name*.

This is only a declaration. Therefore,

- functions are not defined, though the prototype is given,
- variables are declared but are not initialised,
- the declaration block is delineated by **{** and **}**, and terminated with a semicolon.
- *scope resolution operator*, `::` , used in function definition.

- A **Constructor** is a public method of a class, that has the same name as the class. It's return type is not specified explicitly. It is executed whenever a new instance of that class is created.
- A **destructor** is a method that is called on an object whenever it goes out of scope. The name of the destructor is the same as the class, but preceded by a tilde.

3. Vectors and Matrices

This is a course in Scientific Computing.

Many advanced and general problems in Scientific Computing are based around **vectors** and **matrices**. So one of our goals is to implement C++ classes for such structures, along with standard operations such as matrix-vector multiplication.

Along the way, we'll learn about

- operator overloading;
- `friend` functions and the `this` pointer;
- static variables.
- and much more

Our first step will be to study some problems and applications so that, before we design any classes or algorithms, we'll know what we will use them for. These problems include:

1. Basic analysis of matrices, for example with applications to image processing, graphs and networks.
2. Solution of linear systems of equations, for example with applications to data fitting;
3. Estimation of (certain) eigenvalues, for example with applications to Network Science.

Of these problems, probably the most ubiquitous is the solution of (large) systems of simultaneous equations.

That is, we want to solve a linear system of 3 equations in 3 unknowns: *find* x_1, x_2, x_3, *such that*

$$\begin{aligned} 3x_1 + 2x_2 + 4x_3 &= 19 \\ x_1 + 2x_2 + 3x_3 &= 14 \\ 5x_1 + 1x_2 + 6x_3 &= 25 \end{aligned}$$

This can be expressed as a **matrix-vector equation:**

More generally, the linear system of N equations in N unknowns:
find $x_1, x_2, \ldots, x_N$*, such that*

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N &= b_2 \\ &\vdots \\ a_{N1}x_1 + a_{N2}x_2 + \cdots + a_{NN}x_N &= b_N. \end{aligned}$$

This, as a **matrix-vector equation** is:

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1N} \\ a_{21} & a_{22} & \ldots & a_{2N} \\ \vdots & & \ddots & \vdots \\ a_{N1} & a_{N2} & \ldots & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix}$$

So, to proceed, we need to be able to represent **vectors** and **matrices** in our codes.

Our first focus will be on defining a class of vectors. This version will be quite **basic** (hence the file names). Will be developed later.

Intuitively, we know it needs the following components:

Due to the level of detail in the matrix and vector classes, the following example is divided into three source files:

1. `VectorBasic.h`, the header file which contains the class definition. Include this header file in another source file with:
   `#include "VectorBasic.h"`
   Note that this is **not** `<VectorBasic.h>`
2. `VectorBasic.cpp`, which includes the code for the methods in the *`Vector`* class;
3. `01TestVectorBasic.cpp`, a test stub.

In whatever compiler you are using, you'll need to create a **project** that contains all the files. (Ask Niall for help if needed).

4. A vector class (Basic version)

See `VectorBasic.h` for more details

```
// File: VectorBasic.h (simple version)
// Author: Niall Madden ¡Niall.Madden@UniversityOfGalway.ie¿
// Date: Week 9 of 2425-CS319
// What: Header file for vector class
// See also: VectorBasic.cpp and 01TestVector.cpp
class Vector {
private:
  double *entries;
  unsigned int N;
public:
  Vector(unsigned int Size=2);
  ~Vector(void);

  unsigned int size(void) {return N;};
  double geti(unsigned int i);
  void seti(unsigned int i, double x);

  void print(void);
  double norm(void); // Compute the 2-norm of a vector
  void zero(void); // Set entries of vector to zero.
};
```

VectorBasic.cpp

```
Vector::Vector(unsigned int Size)
{
  N = Size;
  entries = new double[Size];
}

Vector::~Vector()
{
  delete [] entries;
}

void Vector::seti(unsigned int i, double x)
{
  if (i<N)
    entries[i]=x;
  else
    std::cerr << "Vector::seti(): Index out of bounds."
              << std::endl;
}
```

VectorBasic.cpp continued

```
double Vector::geti(unsigned int i)
{
  if (i<N)
    return(entries[i]);
  else {
    std::cerr << "Vector::geti(): Index out of bounds."
              << std::endl;
    return(0);
  }
}

void Vector::print(void)
{
  for (unsigned int i=0; i<N; i++)
    std::cout << "[" << entries[i] << "]" << std::endl;
}
```

VectorBasic.cpp continued

```
double Vector::norm(void)
{
  double x=0;
  for (unsigned int i=0; i<N; i++)
    x+=entries[i]*entries[i];
  return (sqrt(x));
}

void Vector::zero(void)
{
  for (unsigned int i=0; i<N; i++)
    entries[i]=0;
}
```

Here is a simple implementation of a function that computes $c = \alpha a + \beta b$

See `01TestVectorBasic.cpp` for more details

```
// c = alpha*a + beta*b where a,b are vectors; alpha, beta are scalars
void VecAdd (vector &c, vector &a, vector &b,
       double alpha, double beta)
{
  unsigned int N;
  N = a.size();

  if ( (N != b.size()) )
    std::cerr << "dimension mismatch in VecAdd " << std::endl;
  else
  {
    for (unsigned int i=0; i<N; i++)
      c.seti(i, alpha*a.geti(i)+beta*b.geti(i) );
  }
}
```

In Week 8, was saw how, for example, the + operator was "overloaded" to allow us to "add" (i.e., concatenate) two strings. We want to see how overload operators for classes that we write so that, for example, we can use the + operator to add two vectors.

That is, we want to study **Operator Overloading**. But to get this to work, we need to study **copy constructor**s.

This is a technical area of C++ programming, but is unavoidable.

As we already know, **constructor** is a method associated with a class that is called automatically whenever an object of that class is declared.

But there are time when objects are *implicitly* declared, such as when passed (by value) to a function.

Since this will happen often, we need to write special constustors to handle it.

Easlier we defined a class for vectors:

- It stores a vector of N doubles in a dynamically assigned array called *entries*;
- The constructor takes care of the memory allocation.

```
// From VectorBasic.h (Version W09.1)
class Vector {
private:
  double *entries;
  unsigned int N;
public:
  Vector (unsigned int Size=2);
  ~Vector(void);

  unsigned int size(void) {return N;};
  double geti (unsigned int i);
  void seti (unsigned int i, double x);
  // print(), zero() and norm() not shown
};

// Code for the constructor from VectorBasic.cpp
Vector::Vector (unsigned int Size) {
  N = Size;
  entries = new double[Size];
}
```

Recall: `Vector`

We then wrote some functions that manipulate vectors, such as `AddVec` in `Week09/01TestVectorBasic.cpp`

```
void VecAdd (Vector &c, Vector &a, Vector &b,
                double alpha=1.0, double beta=1.0);
```

Note that the `Vector` arguments are passed by reference...

What would happen if we tried the following, seemingly reasonable piece of code?

```
Vector x(4);
x.zero(); // sets entries of a all to 0
Vector y=x; // should define a new vector, with a copy of x
```

This will cause problems for the following reasons:

The solve this problem, we should define our own **copy constructor**. A **copy constructor** is used to make an exact copy of an existing object. Therefore, it takes a single parameter: the address of the object to copy. For example:

See Vector09.cpp for more details

```
// copy constructor
Vector::Vector (const Vector &old_Vector)
{
  N = old_Vector.N;
  entries = new double[N];
  for (unsigned int i=0; i<N; i++)
    entries[i] = old_Vector.entries[i];
}
```

The **copy constructor** can be called two ways:

(a) *explicitly*, .e.g,

```
Vector V(2);
V.seti(0)=1.0; V.seti(1)=2.0;
Vector W(V); // W is a copy V
```

(b) *implicitly*, when ever an object is passed by value to a function. If we have not defined our own copy constructor, the default one is used, which usually causes trouble.

In this section, we'll study **"Operator overloading"** .

Our main goal is to overload the addition (`+`) and subtraction (`-`) operators for vectors.

In the "basic" version of the Vector class, we wrote a function to add two `Vector`s: `AddVec`.

It is called as `AddVec(c,a,b)`, and adds the contents of vectors a and b, and stores the result in c.

It would be much more natural redefine the standard **addition** and **assignment** operators so that we could just write `c=a+b`. This is called **operator overloading**.

To overload an operator we create an **operator function** – usually as a member of the class. (It is also possible to declare an operator function to be a `friend` of a class – it is not a member but does have access to private members of the class. More about `friend`s later).

The general form of the operator function is:

```
return-type class-name::operator#(arguments)
{
    ⋮     // operations to be performed.
};
```

return-type of a operator is usually the class for which it is defined, but it can be any type.
Note that we have a new key-word: `operator`.
The operator being overloaded is substituted for the `#` symbol

Almost all C++ operators can be overloaded:

`+`	`-`	`*`	`/`	`%`	`^`	`&`	`|`	`~`	`!`
`=`	`<`	`>`	`+=`	`-=`	`*=`	`/=`	`%=`	`^=`	`&=`
`|=`	`<<`	`>>`	`>>=`	`<<=`	`==`	`!=`	`<=`	`>=`	`&&`
`||`	`++`	`--`	`->*`	`,`	`->`	`[]`	`()`	`new`	`delete`

but not `.` `::` `.*` `?`

- Operator precedence cannot be changed: `*` is still evaluated before `+`
- The number of arguments that the operator takes cannot be changed, e.g., the `++` operator will still take a single argument, and the `/` operator will still take two.
- The original meaning of an operator is not changed; its functionality is extended. It follows from this that operator overloading is always relative to a user-defined type (in our examples, a `class`), and not a built-in type such as `int` or `char`.
- Operator overloading is always relative to a user-defined type (in our examples, a `class`).
- The assignment operator, `=`, is automatically overloaded, but in a way that usually fails except for very simple classes.

We are free to have the overloaded operator perform any operation we wish, but it is good practice to relate it to a task based on the traditional meaning of the operator. E.g., if we wanted to use an operator to add two matrices, it makes more sense to use `+` as the operator rather than, say, `*`.

We will concentrate mainly on binary operators, but later we will also look at overloading the unary "minus" operator.

. . . .

For our first example, we'll see how to overload `operator+` to add two objects from our `vector` class.

First we'll add the declaration of the operator to the class definition in the header file, Vector09.h:

```
    vector operator+(vector b);
```

Then to Vector09.cpp, we add the code

See Vector09.cpp for more details

```
// Overload the + operator.
Vector Vector::operator+(Vector b)
{
  Vector c(N); // Make c the size of a
  if (N != b.N)
    std::cerr << "vector::+ : cant add two vectors of "
              << "different size!"  << std::endl;
  else
    for (unsigned int i=0; i<N; i++)
      c.entries[i] = entries[i] + b.entries[i];
  return(c);
}
```

First thing to notice is that, although `+` is a binary operator, it seems to take only one argument. This is because, when we call the operator, `c = a + b` then `a` is passed **implicitly** to the function and `b` is passed **explicitly**.

Therefore, for example, `a.N` is known to the function simply as `N`.

The temporary object `c` is used inside the object to store the result. It is this object that is returned. Neither `a` or `b` are modified.

There are many more aspects of Operator Overloading not covered in these notes. Among the topics omitted are:

- overloading the unary `++` and `--` operators. There are complications because they work in both prefix and postfix form.
- Overloading the ternary operator: `? :`
- **Important:** overloading the `[]` operator.

See **"extras"** section from Week 9 for more examples of classes and overloading (points, dates, complex numbers); Code for these is in the `Week09/extras/` folder on the website.

We now want to see another way of accessing the implicitly passed argument. First, though, we need to learn a little more about pointers, and introduce a new piece of C++ notation.

Recall that if, for example, `x` is a `double` and `y` is a pointer to `double`, we can set `y=&x`. So now `y` stores the memory address of `x`. We then access the contents of that address using `*y`.

Now suppose that we have an object of type `Vector` called *v*, and a *pointer to vector*, *w*. That is, we have defined

```
Vector v;
Vector *w;
```

Then we can set `w=&v`. Now accessing the member *N* using `v.N`, will be the same as accessing it as `(*w).N`.

It is important to realise that `(*w).N` is **not** the same as `*w.N`.

C++ provides a new operator for this situation: `w->N`, which is equivalent to `(*w).N`.

When writing code for functions, and especially overloaded operators, it can be useful to **explicitly** access the implicitly passed object.

That is done using the `this` pointer, which is a pointer to the object itself.

. . . .

As we've just noted, since `this` is a pointer, its members are accessed using either `(*this).N` or `this->N`.

We often use the this pointer when a function must return the address of the argument that was passed to it. This is the case of the assignment operator.

See Vector09.cpp for more details

```
// Overload the = operator.
Vector &Vector::operator=(const Vector &b)
{
  if (this == &b)
    return(*this); // Taking care for self-assignment

  delete [] entries; // In case memory was already allocated

  N = b.N;
  entries = new double[b.N];
  for (unsigned int i=0; i<N; i++)
    entries[i] = b.entries[i];

  return(*this);
}
```

8. Unary Operators

So far we have discussed just the **binary** operator, `+`. By "**binary**", we mean it takes **two** arguments.

But many C++ operators are **unary**: they take only one argument; examples include `++` and `--`.

For our `Vector` class, we want to overload the `-` (minus) operator. Note that this can be used in two ways:

- $c = -a$ (unary).
- $c = a - b$ (binary)

In the first case here, "minus" is an example of a **prefix** operator. (See "Extras" for example of overloading **postfix** operators, like `a++`, which are a little more complicated).

After that we will then define the binary minus operator, by using addition and unary minus.

For the unary "minus" operator, when we write "-a" the object a is passed *implicitly*. This is a little different from previous cases, where the object passed implicitly is to the left of the operator.

See Vector09.cpp for more details

```
// Overload the unary minus (-) operator. As in b=-a;
Vector Vector::operator-(void)
{
  Vector b(N); // Make b the size of a
  for (unsigned int i=0; i<N; i++)
    b.entries[i] = -entries[i];
  return(b);
}
```

And now that we have defined this operator, we can define the **binary** minus operator. Now this time when we write "`a-b`", it is the *left* argument that is implicit.

See `Vector09.cpp` for more details

```
// Overload the binary minus (-) operator. As in c=a-b
// This implementation reuses the unary minus (-) operator
Vector Vector::operator-(Vector b)
{
  Vector c(N); // Make b the size of a
  if (N != b.N)
    std::cerr << "Vector:: operator- : dimension mismatch!"
              << std::endl;
  else
    c = *this + (-b);
  return(c);
}
```

9. `friend` functions

In all the examples that we have seen so far, the only functions that may access private data belonging to an object has been a member function/method of that object.

If we need a function that does not below to the class to be able to access `private` elements, it can be designated a `friend` of the class.

For non-operator functions, there is nothing that complicated about `friend`s. However, care must be taken when overloading operators as `friend`s.

In particular:

- All arguments are passed explicitly to `friend` functions/operators.
- Certain operators, particularly the **insertion/put-to** `<<` and **extraction/get-from** `>>` operators can only be overloaded as friends.

In last week's version of the `Vector` class, we could output its elements using the `print()` method. E.g.:

```
Vector v;
v.zero()
std::cout << "v  has  values  ";
v.print();
```

But it would be much more convinient just to do

```
std::cout << "v  has  values  " << v;
```

But the **insertion** operator was not defined for our class.

We can fix that, by overloading it. However, the `<<` operator belongs to `std::cout`, not to `Vector`. So it cannot access its `entries` member.

Here is how we resolve this...

We add the following line to the definition of the Vector class.

```
  friend std::ostream &operator<<(std::ostream &, Vector &v);
```

And the we define:

```
std::ostream &operator<<(std::ostream &output, Vector &v)
{
   output << "[";
   for (unsigned int i=0; i<v.size()-1; i++)
      output << v.entries[i] << ",";
   output << v.entries[v.size()-1] << "]";

   return(output);
}
```

Now we can display a vector using std::cout directly.

10. Preprocessor Directives

Our next step is to define a `Matrix` class, and overload some of the associated operators. One of those is the multiplication ("times") operator `*` for matrix-vector multiplication.

With those done, we can think about overloading the multiplication operator for *`Matrix`*-*`Vector`* multiplication.

This introduces a few small new complications:

- the return type is different from the class type;
- if we use multiple source files, how do we know where exactly to place the `#include` directives?

So, before we can proceed, we need to take a short detour to consider **preprocessor** directives.

The preprocessor in C++ is a hang-over over from early versions of C. Originally, that language did not have a construct for defining constants and including header files. To get around this, an early version of C introduced the **preprocessor**. This is a program that

- reads and modifies your source code by checking for any lines that being with a hash symbol (`#`);
- carries out any operations required by these lines;
- forms a new source code that is then compiled.

We usually don't get to see this new file, though you can view it by compiling with certain options (with `g++`, this is `-E`).

The preprocessor is *separate* from the compiler, and has its own syntax.

The simplest preprocessor directive is `#define`. This is used for defining global constants, and doing a simple search-and-replace. For example,

```
#define SIZE 10
```

will find every instance of the word (well, token, really) *`SIZE`* and replaces it with `10`.

In general, this use of the `#define` directive to define identifiers to be used like "global variables" is not very good practice. However, it can be very useful as a way of checking if a piece of code has already been compiled.

The most familiar preprocessor is `#include`, e.g.,

```
#include <iostream>
#include "Vector09.h"
```

This tells the preprocessor to take the named file(s) and insert them into the current file.

If the name is contained in angle brackets, as in `iostream`, this means the preprocessor will look in "the usual place" – where the compiler is installed on your system.

If the named file is in quotes, it looks in the current directory/folder, or in the specified location.

Finally, we have **conditional compilation**.

Suppose we want to write a member function for the *Matrix* class that involves the `Vector` class.

So we need to include `Vector09.h` in `Matrix09.h`. But then if our main source file includes both `Matrix09.h` and `Vector09.h` we could end up defining it twice.

To get around this we use *conditional compilation*.

In the files we can have such lines as the following in `Vector09.h`

```
#ifndef _VECTOR_H_INCLUDED
#define _VECTOR_H_INCLUDED
// stuff goes here
#endif
```

11. Solving Linear Systems

We now move towards learning about **matrices**. When implementing the class, we will learn about

- operator overloading;
- `friend` functions and the `this` pointer;
- static variables.
- and much more

One of the most ubiquitous problems in scientific computing is the solution of (large) systems of simultaneous equations. That is, we want to solve a linear system of N equations in N unknowns: *find* $x_1, x_2, \ldots, x_N$, *such that*

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N &= b_1\\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N &= b_2\\
&\vdots\\
a_{N1}x_1 + a_{N2}x_2 + \cdots + a_{NN}x_N &= b_N.
\end{aligned}$$

There are several classic approaches:

1. Gaussian Elimination;
2. Related: LU- and Cholesky factorisation;
3. Stationary Iterative schemes such as **Jacobi's method**, **Gauss-Seidel** and Successive Over Relaxation (SOR);
4. Krylov subspace methods, of which Conjugate Gradients is the best known;
5. Enhancements of the Methods 3 and 4, using preconditioning with, for example, MultiGrid and Incomplete LU-factorisation.

Of the approaches listed above, Jacobi's is by far the simplest to implement, and so is the one we will study first.

See annotated slides.

We now write a class implementation for a matrix, along with the associated functions.

We'll first consider how the matrix is data is stored. The most natural approach might seem to be to construct a two dimensional array. This can be done as follows:

```
double **entries = new double *[N];
for (int i=0; i<N; i++)
   entries[i] = new double N;
```

A simpler, faster approach is to store the N^2 entries of the matrix in a single, one-dimensional, array of length N^2, and then take care how the access is done:

Matrix09.h

```
class Matrix {
private:
  double *entries;
  unsigned int N;
public:
  Matrix (unsigned int Size=2);
  Matrix (const Matrix &m); // Copy constructor
  ~Matrix(void);

  Matrix &operator=(const Matrix &B); // assignment operator

  unsigned int size(void) {return (N);};
  double getij (unsigned int i, unsigned int j);
  void setij (unsigned int i, unsigned int j, double x);

  Vector operator*(Vector u); // Define later!
  void print(void);
};
```

First we'll look at the code for the constructor, to verify that the data is stored just as an 1D array:

from Matrix09.cpp

```
// Basic constructor. See below for copy constructor.
Matrix::Matrix (unsigned int Size)
{
  N = Size;
  entries = new double [N*N];
}
```

12. A matrix class

Next we'll look at the setij() member, to see how indexing works.

from Matrix09.cpp

```
void Matrix::setij (unsigned int i, unsigned int j, double x)
{
  if (i<N && j<N)
    entries[i*N+j]=x;
  else
    std::cerr << "Matrix::setij(): Index out of bounds.\n";
}
```

Other components of the `Matrix` class are similar to the corresponding functions for the `Vector` class, such as the assignment operator, and the copy constructor.

So, we'll just focus on overloading the `operator*` for multiplication of a vector by a matrix: $c = A * b$, where A is an $N \times N$ matrix, and c and b are vectors with N entries.

Since the left operand is a matrix, we'll make this operator a member of the `Matrix` class; its header has the line:

```
Vector operator*(Vector b);
```

The code from Matrix09.cpp is given below.

```
// Overload the operator multiplication (*) for a Matrix-Vector
// product. Matrix is passed implicitly as "this", the Vector is
// passed explicitly. Will return v=(this)*u
Vector Matrix::operator*(Vector u)
{
  Vector v(N); // v = A*u, where A is the implicitly passed Matrix
  if (N != u.size())
    std::cerr << "Error: Matrix::operator* - dimension mismatch"
              << std::endl;
  else
    for (unsigned int i=0; i<N; i++)
    {
      double x=0;
      for (unsigned int j=0; j<N; j++)
        x += entries[i*N+j]*u.geti(j);
      v.seti(i,x);
    }
  return(v);
}
```