C++ for Numerical Programming - lectures 9-10

Martin Robinson

2018

Lecture 9 — A class of vectors

We will design a class of vectors in such a way that:

1. objects of this class behave like a new data type; and
2. code similar in style to Matlab may be written using objects of this class.

We will define operations on and between objects of the class of vectors, and between these objects and data types such as `int` and `float`

We want to be able to write code such as

```
Vector u(3), v(3); // vectors of length 3
Matrix A(3,3); // matrix of size 3 by 3
v = A * u;
u = gmres(A, v)
```

Constructors for vectors

We want the declaration

```
Vector u(3);
```

to create a vector of size 3

The following file should be saved as `Vector.hpp`

```
#ifndef VECTORDEF
#define VECTORDEF
// a simple class of vectors
class Vector
{
public:
  // member variables
    std::vector<double> data; // data stored in vector
    // construct vector of given length
  Vector(int sizeVal);
};
#endif
```

The following file should be saved as `Vector.cpp`

```
#include "Vector.hpp"

// constructor that creates vector of given size with
// double precision entries all initially set to zero
Vector::Vector(int sizeVal):
    data(sizeVal,0)
{}
```

Default constructor (no longer exists)

The default constructor is not appropriate when declaring a vector: we need to know the length of the vector in advance

Because we have given an alternative constructor, and have not overridden the default constructor, the compiler automatically revokes the default constructor

The code

```
Vector a_vector;
```

will now give a compiler error.

Copy constructors

When using the copy constructor generated by the compiler statements such as

```
Vector w(u);
```

Will have the desired effect, `u.data` will be, by default, copied to `w.data`.

If we did wish to override the default the constructor must be added to the list of public members in the file `Vector.hpp`

```
Vector(const Vector& rOther);
```

You could then manually write the copy constructor as

```
Vector::Vector(const Vector& rOther):
    data(rOther.data)
{}
```

and should be included in the file `Vector.cpp`

Note the use of `const Vector& rOther` on the previous slide

The function could have been prototyped by

```
Vector(Vector otherVec)
```

The argument of the function would then not be a reference variable.

A copy of this variable would be made for use in the constructor function. If the vector is big this will slow down the program

Using a reference variable allows us to use the same variable in the function

The qualifier `const` in the prototype ensures that this reference variable cannot be altered inside the function

Destructors

Recall that a destructor function is automatically generated and is called when a variable is destroyed, i.e. goes out of scope

Luckily, we have used a `std::vector<double>` for `data`, which automatically cleans up after itself (e.g. it frees the memory allocated to it)

You should avoid manually allocating and deallocating memory unless absolutly necessary (This is what the STL containers do for you), but if you do, then you would need to override the default destructor to free the memory manually.

First the destructor must be included in the file `Vector.hpp`

```
~Vector();
```

The destructor should then be added to the file `Vector.cpp`

```
Vector::~Vector()
{
    // do stuff here
```

Functions

Recall that functions (methods) may be defined on classes

For example, the method `norm(p)`, the p-norm of `u`

We will assign `p` the default value 2

Add the following line of code into the list of `public` members of the class `Vector`

```
double norm(int p=2) const;
```

together with the function given on the following slide. The `const` keyword after the method informs the compiler that there should be no changes to the class. The norm of a `Vector` may be calculated using statements such as

```
x = u.norm();
y = v.norm(1);
```

```cpp
double Vector::norm(int p) const
{
  double temp, sum, norm_val;
  sum = 0.0;
    for (int i=0; i<data.size(); i++)
  {
    temp = fabs(data[i]); // floating point absolute value
    sum += pow(temp, p);
  }
    norm_val = pow(sum, 1.0/static_cast<double>(p));
  return norm_val;
}
```

External function versus member method

We have written a function that calculates the p-norm of a vector `u` using statements such as

```
dp = u.norm();
```

If you were used to Matlab-style code, this notation might seem clumsy - we could instead write statements like

```
dp = norm(u);
```

The function `norm` is declared as a *friend* in the file `Vector.hpp`

```
friend double norm(Vector vec, int p);
```

Declaring a function as a `friend` of a class allows this function to access the `private` members of the class

The function `norm` is now no longer encapsulated within the class of vectors, and so must be prototyped in `Vector.hpp`:

```
double norm(Vector vec, int p);
```

and is now written

```
double norm(Vector vec, int p)
{
   ...
}
```

Tip: documenting code

In the norm method on a previous slide, it is not obvious what is happening, even though there are only a few lines of code Comments should be added to the code to aid anyone reading the code

For example a description of the function should be given first

```
// Function to calculate the p-norm of a vector
//
// $$
//     \sum_i^n |x_i|^p
// $$
//
// See "An Introduction to Numerical Analysis" by
// Endre Suli and David Mayers, page 60
double Vector::norm(int p) const
{
   ...
}
```

Documenting code is an art rather than a science

A few tips:

Describe what part of the problem the code is solving. Don't describe the code. For example, don't include documentation such as

```
// Loop over values of p going from 0 to n-1
for (p=0; p<n; p++)
```

You code should, as much as possible, be self-documenting, use descriptive variable names, e.g.

```
Dog my_pet_dot;
Lion escaped_lion
escaped_lion.devour(my_pet_dog);
```

Comments should describe everything that isn't clear from your code, e.g. units

```
double transmembrane_potential;  // units - mV
```

We want to write code such as

```
w = u + v;
```

where `u`, `v`, `w` are defined to be objects of the class `Vector`

We have to define within the class what is meant by the operators + and = in this context

This can be achieved by *overloading* the + operator and the = operator for the class of vectors

First we will restrict access to the data in the vector class

Overloading the ( ) operator

We may overload the ( ) operator in order to access elements of an array. The function required to do this is

```
double& Vector::operator()(int i)
{
    return data[i];
}
```

and the following line must be included in the file `Vector.hpp`

```
double& operator()(int i);
```

We can now access the first element of `u` by writing `u(0)`, instead of the clumsy notation `u.data[0]`

Note the appearance of the symbol `&` on the previous slide This indicates that the operator returns a reference

This allows us to use terms such as `u(1)` on the left hand side of expressions such as `u(1) = 2.0`

We can also overload the square brackets operator with

```
double& operator[](int i);
```

Access privileges

Having overloaded the ( ) operator we are now unlikely to access elements of a vector `u` by using the expressions of the form `u.data[2]`

There is now no need for the member `data` to be available outside the class, and so it could be declared as private

As discussed earlier, a good reason for declaring the member `data` as private is that this makes it harder for code to inadvertently alter the elements of a vector

We can also access the length of the vector through a function length that replicates the length function in `Matlab`

The renamed member `mData` is declared as private by writing the file `Vector.hpp` as follows

```
class Vector
{
private:
    std::vector<double> mData;
public:
  Vector(int);
  ...
  friend int length(const Vector& rVec);
  ...
};

int length(const Vector& rVec);
```

Note that the function length is declared as a friend of the class `Vector` and its prototype is also given

The function length is given below

```
int length(const Vector& rVec)
{
    return rVec.mData.size();
}
```

Other functions that are used should also be declared as a friend

Binary operators

To write code such as

```
w = u + v;
```

then the lines

```
Vector& operator=(const Vector& rVec);
friend Vector operator+(const Vector& rVec1, const Vector&
```

should be added within the class description in the file `Vector.hpp`, and then the following two functions should be included in the file `Vector.cpp`

Assignment operator (Option 1)

Implemented using an index loop

```
Vector& Vector::operator=(const Vector& rVec)
{
    for (int i=0; i<v.mData.size(); i++)
  {
    mData[i] = rVec.mData[i];
    }
  return *this;
}
```

We return a reference to the current object, in order that assignments can be "chained" together. i.e.

```
u = v = w;
```

Assignment operator (Option 2)

Implemented using the STL

```
Vector& Vector::operator=(const Vector& rVec)
{
    std::copy(rVec.mData.begin(),rVec.mData.end(),
              mData.begin());
    return *this;
}
```

Addition operator (Option 1)

```
Vector operator+(const Vector& rVec1,
                 const Vector& rVec2)
{
    Vector w(rVec1.mData.size());
    for (int i=0; i<rVec1.mData.size(); i++)
  {
    w.mData[i] = rVec1.mData[i] + rVec2.mData[i];
  }
```

The binary operators - and * can be overloaded in a similar way as to +

When overloading * we first have to define what `u * v` means for a vector, i.e. do we mean the scalar product or the vector product?

We can also also overload * to define multiplication between an array of double precision numbers and a double precision number

For example, if `a` is a double precision floating point variable, and `u` is an array of double precision floating point numbers we can define what is meant by the operator * in the case `a * u`

```
Vector operator*(double a, const Vector& rVec)
{
    Vector w(rVec.mData.size());
    for (int i=0; i<rVec.mData.size(); i++)
  {
    w.mData[i] = a * rVec.mData[i];
  }
  return w;
}
```

after the following line has been included into the file `Vector.hpp`

```
friend Vector operator*(double a, const Vector& rVec);
```

Binary operators without ‘friend’

In overloading operator+ we have used an external friend function rather than a local method because it feels natural to be adding two objects.

```
Vector operator+(const Vector& rVec1,
                 const Vector& rVec2)
{
    Vector w(rVec1.mData.size());
    for (int i=0; i<rVec1.mData.size(); i++)
  {
    w.mData[i] = rVec1.mData[i] + rVec2.mData[i];
  }
  return w;
}
```

However, it is more efficient (in terms of characters typed) to write a binary operator as a member of a class.

```
Vector Vector::operator+(const Vector& rOther)
{
    Vector w(mData.size());
    for (int i=0; i<mData.size(); i++)
  {
    w.mData[i] = mData[i] + rOther.mData[i];
  }
  return w;
}
```

Both operators would be instantiated as `a = b + c`, but in the case of the second style, it would be run as an internal method of `b` (`mData` evaluates to `b`'s `mData`).

Unary operators

The unary operators - and + may also be overloaded in a similar manner to binary operators: add the line

```
friend Vector operator-(const Vector& rVec);
```

to the list of public members of `Vector` in the file `Vector.hpp`, and add the function on the following slide to the file `Vector.cpp`

```
Vector operator-(const Vector& rVec)
{
    Vector w(v.mData.size());
    for (int i=0; i<v.mData.size(); i++)
  {
    w.mData[i] = -v.mData[i];
  }
  return w;
}
```

Overloading the output operator

C++ does not know how to print out a vector, unless you tell it how. (If you print a pointer, then a memory address will be printed.)

Overload the << operator in the hpp file:

```
class Vector
{
private:
  int mSize;
  double* x;
public:
  Vector(int);
  ...
  friend std::ostream&
    operator<<(std::ostream& output, const Vector& rVec);
};
```

The implementation might look like this:

```
// std::cout << "a_vector = " << a_vector << "\n";
// appears as: a_vector = (10, 20)
std::ostream& operator<<
   (std::ostream& output, const Vector& rVec)
{
  output << "(";
  for (int i=0; i<rVec.mSize; i++) {
    output << rVec.mData[i];
        if (i != rVec.mData.size()-1)
      output << ", ";
    else
      output << ")";
  }
  return output; // for multiple << operators.
}
```

or. . . .

```
// std::cout << "a_vector = " << a_vector << "\n";
// appears as: a_vector = (10, 20)
std::ostream& operator<<
   (std::ostream& output, const Vector& rVec)
{
  output << "(";
    std::copy(v.begin(),
              --v.end(),
              std::ostream_iterator<T>(output, ", "));
    output << v.back() << ")";
    return output; // for multiple << operators.
}
```

Tip 1: plugging C++ into Matlab

You may need to interface C++ with Matlab (or with Gnu Octave)

- to get the speed of compiled code in a critical place
- to use an external library written in C++

This is possible with a Matlab executable file (Mex)

In the simplest case the C++ code is a single file containing a function called `mexFunction` with a specific signature

The function `mexFunction` takes points to arrays for output and input

This is compiled with `mex` (a wrapper compiler to g++) and a `.mex` file is produced

The `.mex` file is treated like a `.m` file by Matlab

An example file `myFunc.cpp`

```
#include "mex.h"
#include <iostream>
void
mexFunction(int nlhs, mxArray *plhs[], int nrhs,
    const mxArray *prhs[])
{
  mxArray *v = mxCreateDoubleMatrix(1, 1, mxREAL);
  double *data = mxGetPr(v);
  *data = 3.142;
  std::cout<<"Num args = "<<nrhs<<" \n";
  plhs[0] = v;
}
```

Tip 2: plugging C++ into Python

Python is a useful dynamic (scripting) language for scientific computing.

Many different ways exist to interface C++ to Python

- Boost Python (www.boost.org)
- Swig (www.swig.org)
- Cython (cython.org)
- Using Python API (docs.python.org/2/extending/extending.html) and `distutils`

You can also interleave C and C++ code in Python scripts

- Scipy.Weave (www.scipy.org)

Boost Python

Use the Boost Python library to wrap your C++ functions or classes for Python

An example file `hello.cpp`

```
char const* greet()
{
   return "hello, world";
}

#include <boost/python.hpp>

BOOST_PYTHON_MODULE(hello)
{
    using namespace boost::python;
    def("greet", greet);
}
```