POPULAR ARTICLES (/articles)

Quantitative Finance Reading List (/articles/Quantitative-Finance-Reading-List)

How to Identify Algorithmic Trading Strategies (/articles/How-to-Identify-Algorithmic-Trading-Strategies)

Beginner's Guide to Quantitative Trading (/articles/Beginners-Guide-to-Quantitative-Trading)

Understanding How to Become a Quantitative Analyst (/articles/Understanding-How-to-Become-a-Quantitative-Analyst)

Self-Study Plan for Becoming a Quantitative Analyst (/articles/Self-Study-Plan-for-Becoming-a-Quantitative-Analyst)

MATRIX CLASSES IN C++ - THE HEADER FILE

By Michael Halls-Moore on February 18th, 2013

In order to do any serious work in quantitative finance it is necessary to be familiar with linear algebra. It is used extensively in statistical analysis and finite difference methods and thus plays a large role in quant finance. From your undergraduate mathematics days it is known that linear maps can be represented by matrices. Hence any computational version of such methods requires an implementation of a versatile matrix object. This article will be the first in a two part series on how to implement a *matrix class* that can be used for further quantitative finance techniques.

The first stage in the implementation of such a matrix class is to decide on a *specification*. We need to decide which mathematical operations we wish to include and how the interface to such operations should be implemented. Here is a list of factors we need to consider when creating a matrix class:

The **type(s)** which will represent the underlying numerical values

The **STL container(s)** that will be used to actually store the values

The **mathematical operations** that will be available such as matrix addition, matrix multiplication, taking the transpose or elemental access

How the matrix will interact with other objects, such as **vectors** and **scalars**

C++ STL Storage Mechanisms

C++ provides many container classes via the Standard Template Library (STL). The most appropriate choices here are `std::valarray` and `std::vector`. The `std::vector` template class is much more frequently utilised because it is more general. `std::vector` can handle many different types (including pointers and smart pointer objects), whereas `std::valarray` is designed solely for numerical values and thus the compiler can make certain optimisations.

At first glance `std::valarray` would seem like a great choice to provide storage for our matrix values. However, in reality it turns out that compiler support for the numerical optimisations that `std::valarray` is supposed to provide does not really exist. Not only that but the `std::valarray` has a poorer *application programming interface* (API) and isn't as flexible as a `std::vector`. Thus it makes sense to use the `std::vector` template class for our underlying storage mechanism.

Supposing that our matrix has M rows and N columns, we could either create a single `std::vector` of length $N \times M$ or create a "vector of vectors". The latter creates a single vector of length M, which takes a `std::vector<T>` of types as its type. The inner vectors will each be of length N. We will utilise the "vector of vectors" approach. The primary reason to use such a mechanism is that we gain a good API, helpful in accessing elements of such a vector "for free". We do not need to look up the correct element as we would need to do in the single large vector approach. The declaration for this type of storage mechanism is given by:

```
std::vector<std::vector<T> >
```

Where T is our type placeholder. *Note: For nearly all of the quantitative work we carry out, we will use the* `double` *precision type for numerical storage.*

Note the extra space between the last two delimiters: `> >`. *This is to stop the compiler into believing we are trying to access the* `>>` *operator.*

The interface to the matrix class will be such that we could reconfigure the underlying storage mechanism to be more efficient for a particular use case, but the external client code would remain identical and would never need know of any such changes. Thus we could entirely replace our `std::vector` storage with `std::valarray` storage and our client calling code would be none the wiser. This is one of the benefits of *encapsulation*.

Matrix Mathematical Operations

A flexible matrix class should support a wide variety of mathematical operations in order to reduce the need for the client to write excess code. In particular, the basic binary operators should be supported for various matrix interactions. We would like to be able to add, subtract

and multiply matrices, take their transpose, multiply a matrix and vector as well as add, subtract, multiply or divide all elements by a scalar value. We will need to access elements individually by their row and column index.

We will also support an additional operation, which creates a vector of the diagonal elements of the matrix. This last method is useful within *numerical linear algebra*. We could also add the ability to calculate a determinant or an inverse (although there are good performance-based reasons *not* to do this directly). However, at this stage we won't support the latter two operations.

Here is a partial listing for the declaration of the matrix operations we will support:

```
// Matrix mathematical operations
QSMatrix<T> operator+(const QSMatrix<T>& rhs);
QSMatrix<T>& operator+=(const QSMatrix<T>& rhs);
QSMatrix<T> operator-(const QSMatrix<T>& rhs);
QSMatrix<T>& operator-=(const QSMatrix<T>& rhs);
QSMatrix<T> operator*(const QSMatrix<T>& rhs);
QSMatrix<T>& operator*=(const QSMatrix<T>& rhs);
QSMatrix<T> transpose();
```

We are making use of the object-oriented *operator overloading* technique. Essentially we are redefining how the individual mathematical operators (such as +, -, *) will behave when we apply them to other matrices as a *binary operator*. Let's consider the syntax for the addition operator:

```
QSMatrix<T> operator+(const QSMatrix<T>& rhs);
```

The function is returning a `QSMatrix<T>` . Note that we're not returning a reference to this object, we're directly returning the object itself. Thus a new matrix is being allocated when this method is called. The method requires a *const reference* to another matrix, which forms the "right hand side" (rhs) of the binary operation. It is a *const reference* because we don't want the matrix to be modified when the operation is carried out (the const part) and is passed by reference (as opposed to value) as we do not want to generate an expensive copy of this matrix when the function is called. Click the following to read more on passing by reference to const (http://www.quantstart.com/articles/Passing-By-Reference-To-Const-in-C).

This operator will allow us to produce code such as the following:

```
// Create and initialise two square matrices with N = M = 10
// using element values 1.0 and 2.0, respectively
QSMatrix mat1(10, 10, 1.0);
QSMatrix mat2(10, 10, 2.0);

// Create a new matrix and set it equal to the sum
// of the first two matrices
QSMatrix mat3 = mat1 + mat2;
```

The second type of mathematical operator we will study is the *operation assignment* variant. This is similar to the binary operator but will assign the result of the operation to the object calling it. Hence instead of creating a separate matrix C in the equation $C = A + B$, we will assign the result of $A + B$ into A. Here is the declaration syntax for such an operator overload:

```
QSMatrix<T>& operator+=(const QSMatrix<T>& rhs);
```

The major difference between this and the standard binary addition operator is that we are now returning a reference to a matrix object, not the object itself by value. This is because we need to return the original matrix that will hold the final result, not a copy of it. The method is implemented slightly differently, which will be outlined in the article on the source implementation.

The remaining matrix mathematical operators are similar to binary addition. We are going to support addition, subtraction and multiplication. We will leave out division as dividing one matrix by another is ambiguous and ill-defined. The `transpose()` method simply returns a new result matrix which holds the transpose of the original matrix, so we won't dwell too much on it here.

We also wish to support scalar addition, subtraction, multiplication and division. This means we will apply a scalar operation to each element of the matrix. The method declarations are given below:

```
// Matrix/scalar operations
QSMatrix<T> operator+(const T& rhs);
QSMatrix<T> operator-(const T& rhs);
QSMatrix<T> operator*(const T& rhs);
QSMatrix<T> operator/(const T& rhs);
```

The difference between these and those provided for the equivalent matrix operations is that the right hand side element is now a *type*, not a matrix of types. C++ allows us to reuse overloaded operators with the same method *name* but with a differing method *signature*. Thus we can use the same operator for multiple contexts, when such contexts are not ambiguous. We will see how these methods are implemented once we create the source file.

We also wish to support matrix/vector multiplication. The method signature declaration is given below:

```
std::vector<T> operator*(const std::vector<T>& rhs);
```

The method returns a vector of types and takes a const reference vector of types on the right hand side. This exactly mirrors the mathematical operation, which applies a matrix to a vector (right multiplication) and produces a vector as output.

The final aspect of the header file that requires discussion is the access to individual elements via the `operator()`. Usually this operator is used to turn the object into a *functor* (i.e. a function object). However, in this instance we are going to overload it to represent operator access. It is one of the only operators in C++ that can be used with multiple parameters. The parameters in question are the row and column indices (zero-based, i.e. running from 0 to $N-1$).

We have created two separate overloads of this method. The latter is similar to the first except we have added the `const` keyword in two places. The first `const` states that we are returning a constant type which should not be modified. The second `const` states that the method itself will not modify any values. This is necessary if we wish to have *read-only* access to the elements of the matrix. It prevents other `const` methods from throwing an error when obtaining individual element access.

The two methods are given below:

```
// Access the individual elements
T& operator()(const unsigned& row, const unsigned& col);
const T& operator()(const unsigned& row, const unsigned& col) const;
```

Full Declaration

For completeness, the full listing of the matrix header file is given below:

```
#ifndef __QS_MATRIX_H
#define __QS_MATRIX_H

#include <vector>

template <typename T> class QSMatrix {
 private:
  std::vector<std::vector<T> > mat;
  unsigned rows;
  unsigned cols;

 public:
  QSMatrix(unsigned _rows, unsigned _cols, const T& _initial);
  QSMatrix(const QSMatrix<T>& rhs);
  virtual ~QSMatrix();

  // Operator overloading, for "standard" mathematical matrix operations


  QSMatrix<T>& operator=(const QSMatrix<T>& rhs);

  // Matrix mathematical operations
```

```
  QSMatrix<T> operator+(const QSMatrix<T>& rhs);
  QSMatrix<T>& operator+=(const QSMatrix<T>& rhs);
  QSMatrix<T> operator-(const QSMatrix<T>& rhs);
  QSMatrix<T>& operator-=(const QSMatrix<T>& rhs);
  QSMatrix<T> operator*(const QSMatrix<T>& rhs);
  QSMatrix<T>& operator*=(const QSMatrix<T>& rhs);
  QSMatrix<T> transpose();

  // Matrix/scalar operations


  QSMatrix<T> operator+(const T& rhs);
  QSMatrix<T> operator-(const T& rhs);
  QSMatrix<T> operator*(const T& rhs);
  QSMatrix<T> operator/(const T& rhs);

  // Matrix/vector operations


  std::vector<T> operator*(const std::vector<T>& rhs);
  std::vector<T> diag_vec();

  // Access the individual elements


  T& operator()(const unsigned& row, const unsigned& col);
  const T& operator()(const unsigned& row, const unsigned& col) const;

  // Access the row and column sizes


  unsigned get_rows() const;
  unsigned get_cols() const;

};

#include "matrix.cpp"

#endif
```

You may have noticed that the matrix source file has been included before the final preprocessor directive:

```
#include "matrix.cpp"
```

This is actually a necessity when working with template classes. The compiler needs to see both the declaration and the implementation, within the same file (which occurs here as the compiler replaces the include statement with the source file text directly in the preprocessor step), prior to any usage in an external client code. If this is not carried out, the compiler will throw obscure-looking linker errors, which can be tricky to debug! Thus, make sure you always include your source file at the bottom of your declaration file if you make use of template classes.

Make sure to have a look at the second part of this series (http://www.quantstart.com/articles/Matrix-Classes-in-C-The-Source-File) which discusses the source file implementation (http://www.quantstart.com/articles/Matrix-Classes-in-C-The-Source-File).

Michael Halls-Moore

Mike is the founder of QuantStart and has been involved in the quantitative finance industry for the last five years, primarily as a quant developer and later as a quant trader consulting for hedge funds.

Visit Michael's LinkedIn Profile (https://www.linkedin.com/profile/view?id=10276159)

Related Articles

Monte Carlo Simulations In CUDA - Barrier Option Pricing (/articles/Monte-Carlo-Simulations-In-CUDA-Barrier-Option-Pricing)

dev_array: A Useful Array Class for CUDA (/articles/dev_array_A_Useful_Array_Class_for_CUDA)

Installing Nvidia CUDA on Ubuntu 14.04 for Linux GPU Computing (/articles/Installing-Nvidia-CUDA-on-Ubuntu-14-04-for-Linux-GPU-Computing)

Vector Addition "Hello World!" Example with CUDA on Mac OSX (/articles/Vector-Addition-Hello-World-Example-with-CUDA-on-Mac-OSX)

Installing Nvidia CUDA on Mac OSX for GPU-Based Parallel Computing (/articles/Installing-Nvidia-CUDA-on-Mac-OSX-for-GPU-Based-Parallel-Computing)

Calculating the Greeks with Finite Difference and Monte Carlo Methods in C++ (/articles/Calculating-the-Greeks-with-Finite-Difference-and-Monte-Carlo-Methods-in-C)

Jump-Diffusion Models for European Options Pricing in C++ (/articles/Jump-Diffusion-Models-for-European-Options-Pricing-in-C)

Heston Stochastic Volatility Model with Euler Discretisation in C++ (/articles/Heston-Stochastic-Volatility-Model-with-Euler-Discretisation-in-C)

Implied Volatility in C++ using Template Functions and Newton-Raphson (/articles/Implied-Volatility-in-C-using-Template-Functions-and-Newton-Raphson)

Eigen Library for Matrix Algebra in C++ (/articles/Eigen-Library-for-Matrix-Algebra-in-C)

Start the discussion…

Be the first to comment.