Developing Classes for Linear Algebra Calculations

10

In this chapter, we will apply the ideas introduced earlier in this book to develop a collection of classes that allow us to perform linear algebra calculations. We will describe the design of a class of vectors in the body of this chapter. The exercises at the end of the chapter will focus on developing this class further, developing a companion class of matrices, and developing a linear system class that allows us to solve matrix equations.

10.1 Requirements of the Linear Algebra Classes

As explained above, we will develop a class of vectors called `Vector`, a class of matrices called `Matrix` and a linear system class called `LinearSystem`. The vector and matrix classes will include constructors and destructors that handle memory management. These classes will overload the assignment, addition, subtraction and multiplication operators, allowing us to write code such as "`u = A*v;`" where `u` and `v` are vectors, and `A` is a matrix: these overloaded operators will include checks that the vectors and matrices are of the correct size. The square bracket operator will be overloaded for the vector class to provide a check that the index of the array lies within the correct range, and the round bracket operator will be overloaded to allow the entries of the vector or matrix to be accessed using MATLAB style notation, indexing from 1 rather than from zero.

The remainder of this chapter will focus on the development of a class of vectors. The header file for this class is given in Listing 10.1, and the implementation of the methods is given in Listing 10.2. The two variables that each instance of the class are built upon are a pointer to a double precision floating point variable, `mData`, and the size of the array, `mSize`. We have made both of these private members of the

J. Pitt-Francis and J. Whiteley, *Guide to Scientific Computing in C++*, Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-319-73132-2_10

class. We clearly need to write methods to both access and set values of the array. We shall insist that the size of the array is set through a constructor. As such, we shall not allow the user to change this variable through any method, but will write a public method that allows us to access the size of a given vector.

Listing 10.1 `Vector.hpp`

```
#ifndef VECTORHEADERDEF
#define VECTORHEADERDEF

class Vector
{
private:
   double* mData; // data stored in vector
   int mSize; // size of vector
public:
   Vector(const Vector& otherVector);
   Vector(int size);
   ~Vector();
   int GetSize() const;
   double& operator[](int i); // zero-based indexing
   // read-only zero-based indexing
   double Read(int i) const;
   double& operator()(int i); // one-based indexing
   // assignment
   Vector& operator=(const Vector& otherVector);
   Vector operator+() const; // unary +
   Vector operator-() const; // unary -
   Vector operator+(const Vector& v1) const; // binary +
   Vector operator-(const Vector& v1) const; // binary -
   // scalar multiplication
   Vector operator*(double a) const;
   // p-norm method
   double CalculateNorm(int p=2) const;
   // declare length function as a friend
   friend int length(const Vector& v);
};

// Prototype signature of length() friend function
int length(const Vector& v);

#endif
```

Listing 10.2 `Vector.cpp`

```
#include <cmath>
#include <iostream>
#include <cassert>
#include "Vector.hpp"

// Overridden copy constructor
// Allocates memory for new vector, and copies
```

```
8  // entries of other vector into it
9  Vector::Vector(const Vector& otherVector)
10 {
11   mSize = otherVector.GetSize();
12   mData = new double [mSize];
13   for (int i=0; i<mSize; i++)
14   {
15     mData[i] = otherVector.mData[i];
16   }
17 }
18
19 // Constructor for vector of a given size
20 // Allocates memory, and initialises entries
21 // to zero
22 Vector::Vector(int size)
23 {
24   assert(size > 0);
25   mSize = size;
26   mData = new double [mSize];
27   for (int i=0; i<mSize; i++)
28   {
29     mData[i] = 0.0;
30   }
31 }
32
33 // Overridden destructor to correctly free memory
34 Vector::~Vector() {
35   delete[] mData;
36 }
37
38 // Method to get the size of a vector
39 int Vector::GetSize() const
40 {
41   return mSize;
42 }
43
44 // Overloading square brackets
45 // Note that this uses 'zero-based' indexing,
46 // and a check on the validity of the index
47 double& Vector::operator[](int i)
48 {
49   assert(i > -1);
50   assert(i < mSize);
51   return mData[i];
52 }
53
54 // Read-only variant of []
55 // Note that this uses 'zero-based' indexing,
56 // and a check on the validity of the index
57 double Vector::Read(int i) const
58 {
```

```
59    assert(i > -1);
60    assert(i < mSize);
61    return mData[i];
62  }
63
64  // Overloading round brackets
65  // Note that this uses 'one-based' indexing,
66  // and a check on the validity of the index
67  double& Vector::operator()(int i)
68  {
69    assert(i > 0);
70    assert(i < mSize+1);
71    return mData[i-1];
72  }
73
74  // Overloading the assignment operator
75  Vector& Vector::operator=(const Vector& otherVector)
76  {
77    assert(mSize == otherVector.mSize);
78    for (int i=0; i<mSize; i++)
79    {
80      mData[i] = otherVector.mData[i];
81    }
82    return *this;
83  }
84
85  // Overloading the unary + operator
86  Vector Vector::operator+() const
87  {
88    Vector v(mSize);
89    for (int i=0; i<mSize; i++)
90    {
91      v[i] = mData[i];
92    }
93    return v;
94  }
95
96  // Overloading the unary - operator
97  Vector Vector::operator-() const
98  {
99    Vector v(mSize);
100   for (int i=0; i<mSize; i++)
101   {
102     v[i] = -mData[i];
103   }
104   return v;
105 }
106
107 // Overloading the binary + operator
108 Vector Vector::operator+(const Vector& v1) const
109 {
```

```
110   assert(mSize == v1.mSize);
111   Vector v(mSize);
112   for (int i=0; i<mSize; i++)
113   {
114     v[i] = mData[i] + v1.mData[i];
115   }
116   return v;
117 }
118 
119 // Overloading the binary - operator
120 Vector Vector::operator-(const Vector& v1) const
121 {
122   assert(mSize == v1.mSize);
123   Vector v(mSize);
124   for (int i=0; i<mSize; i++)
125   {
126     v[i] = mData[i] - v1.mData[i];
127   }
128   return v;
129 }
130 
131 // Overloading scalar multiplication
132 Vector Vector::operator*(double a) const
133 {
134   Vector v(mSize);
135   for (int i=0; i<mSize; i++)
136   {
137     v[i] = a*mData[i];
138   }
139   return v;
140 }
141 
142 // Method to calculate norm (with default value p=2)
143 // corresponding to the Euclidean norm
144 double Vector::CalculateNorm(int p) const
145 {
146   double norm_val, sum = 0.0;
147   for (int i=0; i<mSize; i++)
148   {
149     sum += pow(fabs(mData[i]), p);
150   }
151   norm_val = pow(sum, 1.0/((double)(p)));
152   return norm_val;
153 }
154 
155 // MATLAB style friend to get the size of a vector
156 int length(const Vector& v)
157 {
158   return v.mSize;
159 }
```

The files required for the vector class are given above. These files may be downloaded from https://www.springer.com/9783319731315. Subsequent sections of this chapter provide a commentary on why we have chosen to write the methods in the way in which they appear.

10.2 Constructors and Destructors

In the tip given in Sect. 4.3.3, we encouraged the reader to ensure that, when dynamically allocating memory, every `new` statement was matched by a `delete` statement. We explained that if this is not done, then the code may consume large amounts of the available memory. Eventually the computer will run out of memory, preventing the code (and any other application running) from proceeding any further. We have repeated this tip on several occasions. Writing appropriate constructors and destructors for the vector and matrix classes allows us to automatically match a `delete` statement (through the calling of a destructor when the object goes out of scope) with every `new` statement (hidden from the user of the class in a constructor). We now describe appropriate constructors and a destructor for the class of vectors.

10.2.1 The Default Constructor

We want a constructor for the `Vector` class to allocate the memory required to store a given vector when it is called. The default constructor takes no arguments, and therefore this constructor has no way of knowing how many entries the vector requires. As such, it cannot allocate an appropriate size to the vector, and so we ensure that a default constructor is never used by not supplying a default constructor. The automatically generated default constructor will not be available to the user because we are supplying an alternative specialised constructor.

10.2.2 The Copy Constructor

Let us suppose we have an instance of the class `Vector` called `u`. If we were to use the automatically generated copy constructor to create another vector called `v`, then this constructor would *not* perform the tasks that we require of the copy constructor. The member `mSize` would be correctly set. However, the automatically generated copy constructor would not allocate any memory for the new copy of the data, and so it would be impossible for the entries of the vector to be copied correctly. What would actually happen is that the pointer `mData` in the original vector `u` would be assigned to the pointer `mData` in the new vector `v`. As no new memory would be allocated, this would have the effect that `v` would simply become a different name for the original vector `u`: there would only be one vector stored, and changing the entries of `v` would therefore have the unintended effect of changing those of `u`, and vice versa. A further complication of not overriding the default copy constructor

would be that, because two vectors alias their `mData` pointers with the same piece of memory, both vectors would attempt to de-allocate it (by calling `delete` in their destructor, see Sect. 10.2.4) when they went out of scope.

What we actually want to happen when the copy constructor is called is for the member `mSize` of the new vector `v` to be set to the same value as for the original vector `u`. Memory should then be allocated for the new vector so that `v` has the same number of entries as `u`, and the entries of `u` then copied into the correct position in the new vector `v`. We therefore override the automatically generated copy constructor so that it sets the size of `v` to the size of `u`, allocates memory for the vector `v` of the correct size, and then copies the entries of `u` into `v`.

10.2.3 A Specialised Constructor

We have supplied no definition for the default constructor to ensure that it is never used, and have overridden the copy constructor so that if we already have a vector we may create a copy of that vector. We also include a constructor that requires a positive integer input that represents the size of the vector. This constructor sets the member `mSize` to this value, allocates memory for the vector, and initialises all entries to zero.

10.2.4 Destructor

The automatically generated destructor will delete the pointer `mData` and the integer `mSize` when an instance of the class `Vector` goes out of scope, but will not free the memory allocated to this instance of the class: this would be similar to not providing a matching `delete` statement for a `new` statement. We therefore override the automatically generated destructor to free the memory allocated for an instance of the class `Vector` when it goes out of scope.

10.3 Accessing Private Class Members

In Sect. 10.1 we explained that we were going to make both the size of the vector, `mSize`, and the pointer to the entries of the vector, `mData`, private members of the class. This has the advantage that we can only set the size of the vector through the constructor (ensuring that this member is a positive integer, and preventing us from inadvertently changing it while a code is being executed), and allows us to perform a validation that the index of an entry of a vector is correct before attempting to access that entry. In this section, we explain how we have written the methods that allow us to access these private members.

10.3.1 Accessing the Size of a Vector

The size, or length, of a vector is accessed through the public method `GetSize`. This member takes no arguments, and returns the private member `mSize`.

10.3.2 Overloading the Square Bracket Operator

We overload the square bracket operator so that, if `v` is a vector, then `v[i]` returns the entry of `v` with index `i` using zero-based indexing. This method first checks that the index falls within the correct range—that is, a nonnegative integer that is less than `mSize`—and then returns a reference to the value stored in this entry of the vector.

10.3.3 Read-Only Access to Vector Entries

The overloaded square bracket operator can be used for both reading data from the vector and for changing entries of the vector, through a reference. Since we may need to guarantee that some functions which read from a vector do not change it, we also supply a read-only `const` version. This public method `Read` is similar to the square bracket operator. It uses zero-based indexing and first checks that the index falls within the correct range and then returns a copy of the value stored in this entry of the vector.

10.3.4 Overloading the Round Bracket Operator

The round bracket operator is overloaded to allow us to access entries of a vector using one-based indexing. We have chosen the round bracket operator for this purpose as this allows similar notation to that employed by Fortran and MATLAB, both of which use one-based indexing. In common with the overloaded square bracket operator described in Sect. 10.3.2, this method first validates the index before returning the appropriate entry of the vector.

10.4 Operator Overloading for Vector Operations

Readers with experience of programming in MATLAB will appreciate the feature of this system that allows the user to write statements such as "`v = -w;`" and "`a = b + c;`" where `v`, `w`, `a`, `b`, `c` are vectors of a suitable size. We will allow similar looking code to be written for the vectors developed in this chapter through operator overloading: i.e. we will define the assignment operator, and various unary and binary operators. This will be very similar to the operator overloading for complex numbers

in Sect. 6.4. An additional feature required for the class being written here is a check that the vectors are all of the correct size: this will be enforced using `assert` statements.

10.4.1 The Assignment Operator

The overloaded assignment operator first checks that the vector on the left-hand side of the assignment statement is of the same size as the vector on the right-hand side. If this condition is met, the entries of the vector on the right-hand side are copied into the vector on the left-hand side.

10.4.2 Unary Operators

The overloaded unary addition and subtraction operators first declare a vector of the same size as the vector that the unary operator is applied to. The entries of the new vector are then set to the appropriate value before this vector is returned. Note that in the example statement "`v = -w;`" above, it is the assignment operator's responsibility to check that sizes of `v` and `w` match and the unary subtraction need do no error checking.

10.4.3 Binary Operators

The overloaded binary operators first check that the two vectors that are operated on are of the same size. If they are, a new vector of the same size is created. The entries of this new vector are assigned, and this new vector is then returned. In the example statement "`a = b + c;`" above, it is the binary addition operator's responsibility to check that the sizes of the vectors `b` and `c` match, but the assignment operator's responsibility to check that the result can safely be assigned to `a`.

10.5 Functions

A function to calculate the p-norm of a vector is included in our class of vectors. See Sect. A.1.5 for a definition of the p-norm of a vector. This implementation allows the user to call the function with an optional argument p: if this is not specified the default value $p = 2$ (corresponding to the Euclidean norm) will be used.

10.5.1 Members Versus Friends

We note that most functionality in the class is given via member methods and member operators. In order to calculate the 2-norm of a vector or to inspect its size, we must

write "`u.CalculateNorm();`" or "`u.GetSize();`", respectively. This may be considered a clumsy syntax by some users, especially those with experience of MATLAB, and so we provide an alternative `length` function to complement the `GetSize` method. The length function is declared as a *friend* within the class which enables it to read the private `mSize` member. Note that whereas many of the members of the class are declared `const` at the end of the signature—to ensure they do not change the class itself—the length function guarantees that the vector which it is given as an argument will remain constant through making the argument a constant reference variable.

10.6 Tips: Memory Debugging Tools

We stressed in a previous tip (Sect. 4.3.3) that every `new` should be matched with a `delete`. This is especially important when a program allocates memory within a loop. If a long-running program repeatedly allocates memory without de-allocating it, then eventually that program will unnecessarily occupy all the available memory of the computer. This problem—known as a *memory leak*—will eventually cause the program to fail.

There are memory-related problems other than memory leakage. The following code illustrates some common memory errors. The loop in lines 8–11 has an incorrect upper bound and thus the program attempts to write to `x[10]` which does not match the 10 elements allocated to `x` in line 3. The variable `z` is never initialised, which means that the flow of the program at the `if` statement on line 15 is unpredictable. The second `delete` statement—on line 23—is in error since it attempts to de-allocate memory which has already been de–allocated on the previous line. Finally, the memory for `y` which was allocated on line 4 is never deleted.

Listing 10.3 `Broken.cpp`

```
1  int main(int argc, char* argv[])
2  {
3    double* x = new double[10];
4    double* y = new double[10];
5
6    // Error: x[10] is accessed
7    // May cause a run-time error
8    for (int i=0; i<=10; i++)
9    {
10     x[i] = i;
11   }
12
13   // Error: z is not set
14   int z;
15   if (z == 0)
16   {
17     y[0] = x[0];
18   }
19
```

```
20    // Error: x de-allocated twice
21    // May cause a run-time error
22    delete[] x;
23    delete[] x;
24    // Error: y still allocated
25  }
```

The four problems in the program above will not prevent the code from being compiled. The program may also run as expected until the final `delete` statement, but crash at that point. So, in this program, most of the memory errors are undetectable in normal circumstances.

These errors can be detected with a memory debugging tool such as the open source programs *Valgrind* or *Electric Fence*. These tools run an executable file while inspecting all the memory access calls. Some tools (such as Electric Fence) do this by replacing the usual memory libraries with ones which intercept the calls. Others tools (such as Valgrind) run the program inside a virtual machine and externally monitor the memory accesses—a slower process, but one which does not require recompilation of the program.

On running the program given in Listing 10.3 through Valgrind all four memory problems are detected. A summary of the Valgrind output is given below.

```
Invalid write of size 8
  at 0x4006BA: main (Broken.cpp:10)

Conditional jump or move depends on uninitialised value(s)
  at 0x4006D1: main (Broken.cpp:15)

Invalid free() / delete / delete[]
  by 0x4006F8: main (Broken.cpp:23)

80 bytes in 1 blocks are definitely lost...
  by 0x40069A: main (Broken.cpp:4)
```

10.7 Exercises

The exercises in this chapter guide you to build on the `Vector` class with an additional `Matrix` class. These classes are then combined into a `LinearSystem` class (or, in the final exercise, an alternative class derived from it) which has a method for solving systems of the form $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$. Example solutions for these classes are given in Sect. C.1. Figure 10.1 illustrates a typical solution to these exercises with a collaboration diagram for all the classes produced by these exercises. This diagram uses the same UML syntax as Fig. 7.1, as described in Sect. 7.2.

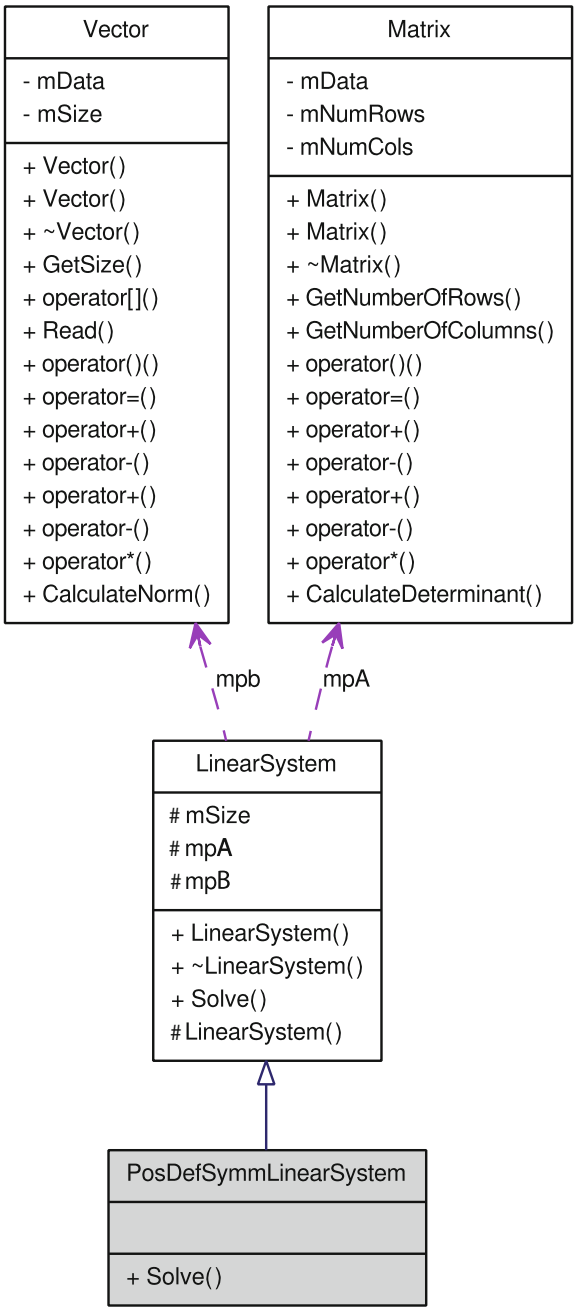

Fig. 10.1 Class collaboration diagram for PosDefSymmLinearSystem

The files `Vector.hpp` and `Vector.cpp` given in Listings 10.1 and 10.2, as well as the example `Matrix` and `LinearSystem` classes given in Sect. C.1, may be downloaded from https://www.springer.com/9783319731315.

10.1 Write a suitable suite of tests to black box test the class of vectors.

10.2 Make any improvements you might deem appropriate to the class of vectors. You might be helped in this task by the following list.

- The assertions for the round bracket operator are almost identical to those of the square bracket operator and those of the `Read` method. Rewrite the `Read` method and one of these operators in such a way that they call the remaining operator (with a suitable offset, as necessary) and all the checks are given in one place.
- There are many assertions in the class as it stands. These mean that it is very easy to write programs which terminate with a run–time error. Can you turn any of the assertions into exceptions or warnings (see Chap. 9)?
- Write an output operator for vectors using the pattern given in Sect. 6.4 for the `operator<<` in the complex number class.

10.3 In this exercise, we will develop a class of matrices called `Matrix` for use with the class of vectors developed in this chapter. The class of matrices should include the features listed below. Your class should have private members `mNumRows` and `mNumCols` that are integers and store the number of rows and columns, and `mData` that is a pointer to a pointer to a double precision floating point variable, which stores the address of the pointer to the first entry of the first row. See Appendix A for details of the linear algebra that underpins these operations. Use a suitable testing strategy when developing this class.

1. An overridden copy constructor that copies the variables `mNumRows` and `mNumCols`, allocates memory for a new matrix, and copies the entries of the original matrix into the new matrix.
2. A constructor that accepts two positive integers—`numRows` and `numCols`—as input, assigns these values to the class members `mNumRows` and `mNumCols`, allocates memory for a matrix of size `mNumRows` by `mNumCols`, and initialises all entries to zero.
3. An overridden destructor that frees the memory that has been allocated to the matrix.
4. Public methods for accessing the number of rows, and the number of columns.
5. An overloaded round bracket operator with one-based indexing for accessing the entries of the matrix so that, provided `i` and `j` are valid indices for the matrix, `A(i,j)` may be used to access `mData[i-1][j-1]`.
6. Overloaded assignment, unary and binary operators to allow addition, subtraction and multiplication of suitably sized matrices, vectors and scalars. You should use `assert` statements to ensure the matrices and vectors are of the correct size.
7. A public method that computes the determinant of a given square matrix.

10.4 In this exercise, we will develop a class called `LinearSystem` that may be used to solve linear systems. Assuming the system is nonsingular, a linear system is defined by the size of the linear system, a square matrix, and vector (representing the right-hand side), with the matrix and vector being of compatible sizes. The data associated with this class may be specified through an integer variable `mSize`, a pointer to a matrix `mpA`, and a pointer to the vector on the right-hand side of the linear system `mpb`. We suggest only allowing the user to set up a linear system through the use of a constructor that requires specification of the matrix and vector: the member `mSize` may then be determined from these two members. If you do not wish to provide a copy constructor, then the automatically generated copy constructor should be overridden and made private to prevent its use. As with the class of vectors, we recommend that use of the automatically generated default constructor is prevented by providing a specialised constructor but no default constructor. A public method `Solve` should be written to solve this linear system by Gaussian elimination with pivoting, as described in Sect. A.2.1.3. This method should return a vector that contains the solution of the linear system.

Test your class using suitable examples. We suggest that you write a set automated tests in a testing framework such as `CxxTest`. An outline model test-suite for testing the linear algebra classes is given in Listing C.5 in Section C.1. When considering what to test think about the following.

- How you might test the various constructors.
- How you will black box test solving problems when the matrix is poorly conditioned.
- How to test that the Gaussian Elimination routine is performing pivoting when small values appear on the diagonal.
- How to test various `Matrix` and `Vector` methods.

10.5 Derive a class called `PosDefSymmLinearSystem` (or similar) from the class `LinearSystem` that may be used for the solution of positive definite symmetric linear systems. Make the method `Solve` a virtual method of the class `LinearSystem`, and override this method in the class `PosDefSymmLinearSystem` so that it uses the conjugate gradient method for solving linear systems described in Sect. A.2.3. If you declared `LinearSystem` member data as `private` in the previous exercises, then this should now be declared `protected`. Your class `PosDefSymmLinearSystem` should perform a check that the matrix used is symmetric: testing that the matrix is positive definite would be rather difficult and so we don't suggest performing a check for this property. Test your class using suitable examples.