Designing Object-Oriented Numerical Libraries

12

Having developed classes that underpin linear algebra operations in Chap. 10 we now demonstrate how to construct object-oriented libraries for scientific computing applications that utilise the functionality of these classes. We use the specific example of developing a library that uses the finite difference method to solve boundary value, second order differential equations.

We begin by developing a library for problems in one spatial dimension that are linear, constant coefficient, second order, boundary value ordinary differential equations. That is, equations of the form

$$A\frac{\mathrm{d}^2u}{\mathrm{d}x^2} + B\frac{\mathrm{d}u}{\mathrm{d}x} + Cu = f(x), \qquad X_0 < x < X_1, \tag{12.1}$$

where A ($\neq 0$), B, C, X_0, X_1 (with $X_0 < X_1$) are given constants, $f(x)$ is a given function, and suitable boundary conditions are given at $x = X_0$ and $x = X_1$. We choose to use the finite difference method to underpin the library as this method for calculating the numerical solution of differential equations is the simplest to explain, and a method that many readers will be familiar with. This allows us to focus on the *implementation* of this method, without a need to explain more technical aspects of the method from a mathematical viewpoint as would be the case with more sophisticated techniques such as the finite element method. Having discussed how to develop a library for this class of equations we conclude this chapter by briefly touching upon how a library for computing the numerical solution of Poisson's equation may be constructed. For ease of explanation, we limit ourselves to a two-dimensional rectangular domain, and apply only Dirichlet boundary conditions, that is, the following partial differential equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y), \qquad X_0 < x < X_1,\ Y_0 < y < Y_1,$$

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_12

where X_0, X_1, Y_0, Y_1 are specified constants with $X_0 < X_1, Y_0 < Y_1$, $f(x, y)$ is a specified function, and u is specified at each point on the boundary. As partial differential equations may be beyond the mathematical scope of some readers, this section is entirely self-contained: the remainder of this chapter may be read independently of the material in Sect. 12.3.

The emphasis of this chapter is to explain the object-oriented structure that may be used when developing a library for solving differential equations. We describe the functionality required from the classes that we use, but give very little detail on the implementation of these classes: implementation of the ideas presented uses C++ techniques introduced in earlier chapters, and is the focus of the exercises at the end of the chapter. The mathematical theory of the finite difference method is not discussed in much detail. Readers unfamiliar with this technique should consult a suitable text such as Iserles [1], Kreyszig [2], or Süli and Mayers [3].

12.1 Developing the Library for Ordinary Differential Equations

When developing software, it is useful to know precisely what type of problems are to be solved using this software. We therefore begin by defining two exemplar model problems that contain all features commonly seen in linear, constant coefficient, boundary value ordinary differential equations. We then explain the mathematical theory behind the finite difference method for these boundary value problems, before concluding this section by explaining how to utilise the theory when developing the library.

12.1.1 Model Problems

We use two example model problems to motivate the development of the library. These model problems have a known solution and can therefore be used to give some verification of the correctness of the output of the library. The first model problem is very simple, whilst the second model problem is more complicated and uses all the features that we will include in our library for ordinary differential equations.

Model Problem 1. The first model problem is the following boundary value problem:

$$\begin{aligned} \frac{\mathrm{d}^2 u}{\mathrm{d}x^2} &= -1, \quad 0 < x < 1, \\ u &= 0, \quad \text{at } x = 0, \\ u &= 0, \quad \text{at } x = 1. \end{aligned}$$

This problem has solution

$$u(x) = \frac{1}{2}x(1-x).$$

This is a very simple problem—we have the minimal number of terms in the differential equation, and only very simple Dirichlet (i.e., non-derivative) boundary conditions.

Model Problem 2. The second model problem is a more complicated differential equation, with one Dirichlet boundary condition, and one Neumann (derivative) boundary condition. This model problem satisfies the following equation and boundary conditions:

$$\begin{aligned} \frac{\mathrm{d}^2u}{\mathrm{d}x^2} + 3\frac{\mathrm{d}u}{\mathrm{d}x} - 4u &= 34\sin\,x, \qquad 0 < x < \pi, \\ \frac{\mathrm{d}u}{\mathrm{d}x} &= -5, \qquad \text{at}\, x = 0, \\ u &= 4, \qquad \text{at}\, x = \pi. \end{aligned}$$

This differential equation has solution

$$u = \frac{4\mathrm{e}^x + \mathrm{e}^{-4x}}{4\mathrm{e}^\pi + \mathrm{e}^{-4\pi}} - 5\sin x - 3\,\cos\,x.$$

12.1.2 Finite Difference Approximation to Derivatives

We now define the notation used for the finite difference approximations to the first and second derivative of a function of one variable. Where we define a derivative at N distinct points, we will denote these points using subscripts starting at 1 and ending at N for consistency with the overloaded parenthesis operators used when writing the classes of vectors and matrices developed in Chap. 10.

Let us suppose that a function u is defined on the interval $X_0 \le x \le X_1$. Suppose further that there is a collection of points x_i, $i = 1, 2, \ldots, N$, that satisfy

$$\begin{aligned} &x_1 = X_0, \\ &x_1 < x_2 < x_3 < \cdots < x_N, \\ &x_N = X_1. \end{aligned}$$

We will refer to the points $x_1, x_2, \ldots, x_N$ as the *finite difference grid*, and the individual points as *nodes*. The nodes x_1 and x_N are referred to as the *boundary nodes* of

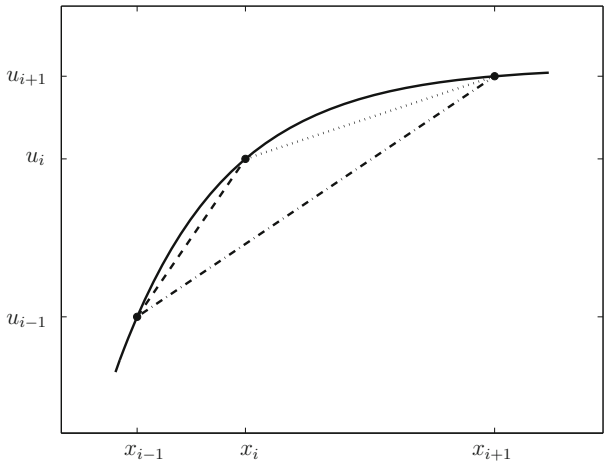

Fig. 12.1 Backward finite difference (*broken line*), forward finite difference (*dotted line*), and central finite difference (*dot-dashed line*) approximations to the first derivative of the function represented by *the solid line* at the point $x = x_i$

Table 12.1 Numerical finite difference approximations to the first derivative at $x = x_i$

Type	Formula	Range
Backward	$(u_i - u_{i-1}) / (x_i - x_{i-1})$	$i = 2, 3, \ldots, N$
Forward	$(u_{i+1} - u_i) / (x_{i+1} - x_i)$	$i = 1, 2, \ldots, N-1$
Central	$(u_{i+1} - u_{i-1}) / (x_{i+1} - x_{i-1})$	$i = 2, 3, \ldots, N-1$

the finite difference grid, whilst all other points are referred to as *interior nodes*. We may evaluate the function u at each node x_i, $i = 1, 2, \ldots, N$, which we denote by u_i:

$$u_i = u(x_i).$$

The first derivative of a function at a given node may be thought of as being the "slope" of the function at that point: i.e. the ratio of the change in u to the change in x. In Fig. 12.1 we motivate three different approximations to the first derivative at $x = x_i$ which are defined in Table 12.1. Note that not all of these approximations are defined at the boundary nodes of the finite difference grid, that is, at $x = x_1$ and $x = x_N$.

A numerical approximation to the second derivative, not defined at the boundary nodes of the finite difference grid, $x = x_1$ and $x = x_N$, is

$$\frac{2}{x_{i+1} - x_{i-1}} \left(\frac{u_{i+1} - u_i}{x_{i+1} - x_i} - \frac{u_i - u_{i-1}}{x_i - x_{i-1}} \right), \qquad i = 2, 3, \ldots, N-1,$$

which may be written

$$\alpha_i u_{i-1} + \beta_i u_i + \gamma_i u_{i+1}, \qquad i = 2, 3, \ldots, N-1, \tag{12.2}$$

where

$$\alpha_i = \frac{2}{(x_{i+1} - x_{i-1})(x_i - x_{i-1})}, \tag{12.3}$$

$$\beta_i = -\frac{2}{(x_{i+1} - x_i)(x_i - x_{i-1})}, \tag{12.4}$$

$$\gamma_i = \frac{2}{(x_{i+1} - x_{i-1})(x_{i+1} - x_i)}. \tag{12.5}$$

This approximation to the second derivative follows from Taylor series expansions: see, for example, Kreyszig [2]. We note that when there is a uniform spacing between the nodes, that is, $x_{i+1} - x_i = h, i = 1, 2, 3, \ldots, N-1$, for some constant h, then the approximation to the second derivative given in Eq. (12.2) may be simplified to the more familiar formula

$$\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}.$$

When developing our classes we will use the approximation given in Eq. (12.2) as it allows more generality.

12.1.3 Application of Finite Difference Methods to Boundary Value Problems

We now explain how the finite difference approximations given in Sect. 12.1.2 may be used to calculate a numerical solution of the model problems given in Sect. 12.1.1. For both problems we use the finite difference grid with N nodes described in Sect. 12.1.2. There are therefore N unknown values of u_i to determine. We will demonstrate how to set up a linear system of size N that allows us to calculate these values.

12.1.3.1 Model Problem 1

Substituting the approximation to second derivative given by Eq. (12.2) into the differential equation at the interior nodes of the finite difference grid yields

$$\alpha_i u_{i-1} + \beta_i u_i + \gamma_i u_{i+1} = -1, \qquad i = 2, 3, \ldots, N-1. \tag{12.6}$$

The boundary conditions imply that

$$u_1 = u_N = 0. \tag{12.7}$$

Equations (12.6) and (12.7) may be combined and written as the linear system $\mathbf{Au} = \mathbf{b}$, where $\mathbf{A}$ is a $N \times N$ matrix, and $\mathbf{u}$ and $\mathbf{b}$ are vectors of length N. The entries of $\mathbf{A}$, $\mathbf{u}$ and $\mathbf{b}$ are then given by

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 & 0 & 0 \\ \alpha_2 & \beta_2 & \gamma_2 & \dots & 0 & 0 & 0 \\ 0 & \alpha_3 & \beta_3 & \dots & 0 & 0 & 0 \\ 0 & 0 & \alpha_4 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \alpha_{N-1} & \beta_{N-1} & \gamma_{N-1} \\ 0 & 0 & 0 & \dots & 0 & 0 & 1 \end{pmatrix},$$

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \vdots \\ u_{N-2} \\ u_{N-1} \\ u_N \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} 0 \\ -1 \\ -1 \\ -1 \\ \vdots \\ -1 \\ -1 \\ 0 \end{pmatrix}.$$

Now that we have written the model problem as a linear system, we may use the methods associated with the vector, matrix and linear system classes to solve this system and calculate the values of u_i, $i = 1, 2, \dots, N$.

12.1.3.2 Model Problem 2

We now write model problem 2 in matrix form. At the interior nodes of the finite difference grid, we use a central approximation to the first derivative, as defined in Table 12.1, and the approximation to the second derivative given by Eq. (12.2). The differential equation may then be approximated by, for $i = 2, 3, \dots, N-1$,

$$\left(\alpha_i - \frac{3}{x_{i+1} - x_{i-1}}\right) u_{i-1} + (\beta_i - 4)u_i + \left(\gamma_i + \frac{3}{x_{i+1} - x_{i-1}}\right) u_{i+1} = 34 \sin x_i. \tag{12.8}$$

The boundary condition at $x = \pi$ may be implemented in the same way as the Dirichlet boundary conditions in model problem 1, that is, we write

$$u_N = 4. \tag{12.9}$$

The Neumann (derivative) boundary condition at $x = 0$ requires a bit more thought. We see from Table 12.1 that the only one of these approximations to the first derivative

that is defined at the node x_1 is the forward approximation. We therefore use this approximation and implement this boundary condition by setting

$$-\frac{1}{x_2 - x_1}u_1 + \frac{1}{x_2 - x_1}u_2 = -5. \tag{12.10}$$

Defining, for $i = 2, 3, \ldots, N-1$, the quantities $\hat{\alpha}_i, \hat{\beta}_i, \hat{\gamma}_i$:

$$\begin{aligned}
\hat{\alpha}_i &= \alpha_i - \frac{3}{x_{i+1} - x_{i-1}}, \\
\hat{\beta}_i &= \beta_i - 4, \\
\hat{\gamma}_i &= \gamma_i + \frac{3}{x_{i+1} - x_{i-1}},
\end{aligned}$$

we may write Eqs. (12.8)–(12.10) as the linear system $\mathbf{Au} = \mathbf{b}$, where the entries of $\mathbf{A}$ and $\mathbf{b}$ are given by

$$\mathbf{A} = \begin{pmatrix}
-1/(x_2 - x_1) & 1/(x_2 - x_1) & 0 & \ldots & 0 & 0 & 0 \\
\hat{\alpha}_2 & \hat{\beta}_2 & \hat{\gamma}_2 & \ldots & 0 & 0 & 0 \\
0 & \hat{\alpha}_3 & \hat{\beta}_3 & \ldots & 0 & 0 & 0 \\
0 & 0 & \hat{\alpha}_4 & \ldots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \ldots & \hat{\alpha}_{N-1} & \hat{\beta}_{N-1} & \hat{\gamma}_{N-1} \\
0 & 0 & 0 & \ldots & 0 & 0 & 1
\end{pmatrix},$$

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \vdots \\ u_{N-2} \\ u_{N-1} \\ u_N \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} -5 \\ 34\sin(x_2) \\ 34\sin(x_3) \\ 34\sin(x_4) \\ \vdots \\ 34\sin(x_{N-2}) \\ 34\sin(x_{N-1}) \\ 4 \end{pmatrix}.$$

As with model problem 1 we may now use the linear system class already written to solve this linear system.

12.1.4 Concluding Remarks on Boundary Value Problems in One Dimension

We have now explained how to write the finite difference approximation to a linear, constant coefficient, second order boundary value problem in matrix notation, thus allowing the classes of vectors, matrices and linear systems developed in Chap. 10 to

be used to calculate the finite difference approximation. In the next section, we will describe an object-oriented structure that allows a very general library for solving such problems to be developed. We should, however, discuss the limitations of this library.

Suppose we want to solve the following equation:

$$\begin{aligned} \frac{\mathrm{d}^2 u}{\mathrm{d}x^2} + u &= 0, \quad 0 < x < 2\pi, \\ u &= 0, \quad \text{at } x = 0, \\ u &= 0, \quad \text{at } x = 2\pi. \end{aligned}$$

This has solution $u = \sin x$, and it may be thought that the library we are writing may be used to solve this problem. However $u = A \sin x$, where A is any constant value, satisfies the differential equation and both boundary conditions: that is, the solution is not unique.

The equation above has a non-unique solution. It is also possible that an equation of the form Eq. (12.1) has no solution. For example, consider the equation

$$\begin{aligned} \frac{\mathrm{d}^2 u}{\mathrm{d}x^2} + u &= 0, \quad 0 < x < 2\pi, \\ u &= 1, \quad \text{at } x = 0, \\ u &= 4, \quad \text{at } x = 2\pi. \end{aligned}$$

It can be shown that this equation, together with these boundary conditions, has no solution.

Proof of existence and uniqueness of solutions to boundary value differential equations is beyond the scope of this book. Nevertheless, the reader should be aware when using this library that some equations have solutions that are not unique, and solutions do not exist for other equations.

12.2 Designing a Library for Solving Boundary Value Problems

To calculate a numerical solution of the boundary value ordinary differential equations discussed above, we may specify the problem by specifying individually: (i) the ordinary differential equation and the interval on which the solution is valid; (ii) the boundary conditions; and (iii) the finite difference grid. Classes will be written for these three entities, called `SecondOrderOde`, `BoundaryConditions` and `FiniteDifferenceGrid`. These will then all be members of a class `BvpOde` that encapsulates a boundary value ordinary differential equation, and contains all the functionality required for the numerical solution of the differential equation. We now discuss the individual classes.

12.2.1 The Class `SecondOrderOde`

To specify the ordinary differential equation, we need to specify the coefficients on the left-hand side of Eq. (12.1), the function on the right-hand side of this equation, and the interval on which the equation is valid. These will all be made members of the class `SecondOrderOde`. To ensure that all of these are specified, we will only allow a user to use a constructor that specifies all of these members. In the exercises at the end of this chapter, we will discuss developing other constructors. A header file for this class is given below.

Listing 12.1 `SecondOrderOde.hpp`

```
#ifndef SECONDORDERODEHEADERDEF
#define SECONDORDERODEHEADERDEF

class SecondOrderOde
{
   // The boundary value class is able to
   // access the coefficients etc. of this equation
   friend class BvpOde;
private:
   // Coefficients on LHS of ODE
   double mCoeffOfUxx;
   double mCoeffOfUx;
   double mCoeffOfU;
   // Function on RHS of ODE
   double (*mpRhsFunc)(double x);

   // Interval for domain
   double mXmin;
   double mXmax;
public:
   SecondOrderOde(double coeffUxx, double coeffUx,
                  double coeffU,
                  double (*righthandSide)(double),
                  double xMinimum, double xMaximum)
   {
      mCoeffOfUxx = coeffUxx;
      mCoeffOfUx = coeffUx;
      mCoeffOfU = coeffU;
      mpRhsFunc = righthandSide;
      mXmin = xMinimum;
      mXmax = xMaximum;
   }
};

#endif
```

12.2.2 The Class `BoundaryConditions`

On the left boundary, we may specify either the value of the function u (a left Dirichlet boundary condition), or the derivative $\mathrm{d}u/\mathrm{d}x$ (a left Neumann boundary condition). It is important to note that there must be *either* a left Dirichlet boundary condition *or* a left Neumann boundary condition: we must have one of these boundary conditions but we cannot have both. Similarly, on the right boundary we must have *either* a right Dirichlet boundary condition *or* a right Neumann boundary condition. In the class `BoundaryConditions`, we will declare class members `mLhsBcIsDirichlet`, `mRhsBcIsDirichlet`, `mLhsBcIsNeumann`, `mRhsBcIsNeumann` that are Boolean variables, thus allowing us to check that we have precisely one boundary condition on the left–hand boundary, and precisely one boundary condition on the right boundary. The default constructor should be overridden to set these variables to the value "`false`" in the absence of any other instruction. Whatever type of boundary conditions are set, values for these are needed at either end of the interval. These class members are called `mLhsBcValue` and `mRhsBcValue`. Finally, we require methods to set these values, and set the appropriate Boolean variable to the value "`true`". The method `SetLhsDirichletBc` takes a double precision floating point variable as input. It sets the member variable `mLhsBcValue` to this input, and sets the Boolean variable `mLhsBcIsDirichlet` to the value `true`. The methods `SetRhsDirichletBc`, `SetLhsNeumannBc` and `SetRhsNeumannBc` perform similar tasks.

The header file `BoundaryConditions.hpp` is shown below.

Listing 12.2 `BoundaryConditions.hpp`

```
#ifndef BOUNDARYCONDITIONSHEADERDEF
#define BOUNDARYCONDITIONSHEADERDEF

class BoundaryConditions
{
public:
   // The boundary value class is able to
   // access the coefficients etc. of this equation
   friend class BvpOde;
private:
   bool mLhsBcIsDirichlet;
   bool mRhsBcIsDirichlet;
   bool mLhsBcIsNeumann;
   bool mRhsBcIsNeumann;
   double mLhsBcValue;
   double mRhsBcValue;
public:
   BoundaryConditions();
   void SetLhsDirichletBc(double lhsValue);
   void SetRhsDirichletBc(double rhsValue);
   void SetLhsNeumannBc(double lhsDerivValue);
   void SetRhsNeumannBc(double rhsDerivValue);
};

#endif
```

12.2.3 The Class `FiniteDifferenceGrid`

The finite difference grid requires access to the interval on which the equation is valid, given in the class `SecondOrderOde`. To create a uniform grid, we also need specification of the number of nodes. To ensure that the number of nodes is specified, we only allow use of a constructor that sets this through a constructor argument. A vector of uniformly spaced nodes can then be generated. We create a class `Node` that stores the coordinate of each node. Header files for the classes `FiniteDifferenceGrid` and `Node` are given below.

Listing 12.3 `Node.hpp`

```
#ifndef NODEHEADERDEF
#define NODEHEADERDEF

class Node
{
public:
   double coordinate;
};

#endif
```

Listing 12.4 `FiniteDifferenceGrid.hpp`

```
#ifndef FINITEDIFFERENCEGRIDHEADERDEF
#define FINITEDIFFERENCEGRIDHEADERDEF
#include <vector>
#include "Node.hpp"

class FiniteDifferenceGrid
{
public:
   // The boundary value class is able to
   // access the nodes
   friend class BvpOde;
private:
   std::vector<Node> mNodes;
public:
   FiniteDifferenceGrid(int numNodes, double xMin,
                        double xMax);
};

#endif
```

12.2.4 The Class `BvpOde`

Now we have described the classes `SecondOrderOde`, `BoundaryConditions` and `FiniteDifferenceGrid` we may develop the class `BvpOde`. We only allow this class to be instantiated through a constructor that specifies: (i) an instance of the class `SecondOrderOde`; (ii) an instance of the class `BoundaryConditions`; and (iii) the number of nodes to be used in the finite difference grid. Once these entities have been specified we then create an instance of the class `FiniteDifference-Grid`, a vector that will contain the solution, a vector that will be on the right–hand side of a linear system, and a matrix associated with the linear system. Methods will then be written to populate both the matrix and the vector associated with the linear system, and to apply the boundary conditions, as discussed in Sect. 12.1.3. Finally, methods will be written to solve the linear system, and to write the solution to file. A header file `BvpOde.hpp` is given below.

Listing 12.5 `BvpOde.hpp`

```
#ifndef BVPODEHEADERDEF
#define BVPODEHEADERDEF

#include <string>
#include "Matrix.hpp"
#include "Vector.hpp"
#include "LinearSystem.hpp"
#include "FiniteDifferenceGrid.hpp"
#include "SecondOrderOde.hpp"
#include "BoundaryConditions.hpp"

class BvpOde
{
private:
    // Only allow instance to be created from a PDE, boundary
    // conditions, and number of nodes in the mesh (the
    // copy constructor is private)
    BvpOde(const BvpOde& otherBvpOde){}

    // Number of nodes in the grid, and a pointer to a grid
    int mNumNodes;
    FiniteDifferenceGrid* mpGrid;

    // Pointer to instance of an ODE
    SecondOrderOde* mpOde;

    // Pointer to an instance of boundary conditions
    BoundaryConditions* mpBconds;

    // Vector for solution to unknowns
    Vector* mpSolVec;

    // Right-hand side vector
    Vector* mpRhsVec;
```

```
35
36      // Matrix for linear system
37      Matrix* mpLhsMat;
38
39      // Linear system that arises
40      LinearSystem* mpLinearSystem;
41
42      // Allow user to specify the output file or
43      // use a default name
44      std::string mFilename;
45
46
47      // Methods for setting up linear system and solving it
48      void PopulateMatrix();
49      void PopulateVector();
50      void ApplyBoundaryConditions();
51
52  public:
53      // Sole constructor
54      BvpOde(SecondOrderOde* pOde, BoundaryConditions* pBcs,
55                                   int numNodes);
56
57      // As memory is dynamically allocated the destructor
58      // is overridden
59      ~BvpOde();
60
61      void SetFilename(const std::string& name)
62      {
63         mFilename = name;
64      }
65      void Solve();
66      void WriteSolutionFile();
67  };
68
69  #endif
```

12.2.5 Using the Class `BvpOde`

When using the classes introduced above, we would like to write code such as that in Listing 12.6 to calculate a numerical solution of the model problems given in Sect. 12.1.1. This will form the basis for the exercises at the end of this chapter.

Listing 12.6 `Driver.cpp` for testing the code in Sect. 12.2 on the model problems discussed in Sect. 12.1.1

```
#include <cmath>
#include <string>
#include "BvpOde.hpp"

double model_prob_1_rhs(double x){return 1.0;}
double model_prob_2_rhs(double x){return 34.0*sin(x);}

int main(int argc, char* argv[])
{
    SecondOrderOde ode_mp1(-1.0, 0.0, 0.0,
                           model_prob_1_rhs,
                           0.0, 1.0);
    BoundaryConditions bc_mp1;
    bc_mp1.SetLhsDirichletBc(0.0);
    bc_mp1.SetRhsDirichletBc(0.0);

    BvpOde bvpode_mp1(&ode_mp1, &bc_mp1, 101);
    bvpode_mp1.SetFilename("model_problem_results1.dat");
    bvpode_mp1.Solve();

    SecondOrderOde ode_mp2(1.0, 3.0, -4.0,
                           model_prob_2_rhs,
                           0.0, M_PI);
    BoundaryConditions bc_mp2;
    bc_mp2.SetLhsNeumannBc(-5.0);
    bc_mp2.SetRhsDirichletBc(4.0);

    BvpOde bvpode_mp2(&ode_mp2, &bc_mp2, 1001);
    bvpode_mp2.SetFilename("model_problem_results2.dat");
    bvpode_mp2.Solve();

    return 0;
}
```

12.3 Extending the Library to Two Dimensions

In this section, we assume that the reader is familiar with partial differentiation: that is, if a differentiable function $u(x, y)$ depends on the variables x and y then *partial derivatives* with respect to both x and y may be calculated. Readers unfamiliar with partial differential equations may wish to skip this section or consult a suitable text on mathematical methods such as Kreyszig [2].

In the previous section, we designed a library for calculating the finite difference solution of linear, constant coefficient, second order, boundary value ordinary differential equations. We will now explain how a library may be developed for the finite

difference solution of Poisson's equation in two spatial dimensions on a rectangular domain, with Dirichlet boundary conditions, that is, equations of the form

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y), \qquad X_0 < x < X_1, Y_0 < y < Y_1,$$

where X_0, X_1, Y_0, Y_1 are specified constants, $f(x, y)$ is a specified function, and boundary conditions for u are given at each point on the boundary of the rectangular domain specified.

12.3.1 Model Problem for Two Dimensions

As with ordinary differential equations earlier in this chapter, we will use a model problem to demonstrate the implementation of the finite difference method. The model problem that we will use is

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -4(1 - x^2 - y^2)\mathrm{e}^{-(x^2 + y^2)}, \quad 0 < x < 1, 0 < y < 2, \tag{12.11}$$

$$u = \mathrm{e}^{-y^2}, \quad x = 0, \ 0 < y < 2, \tag{12.12}$$

$$u = \mathrm{e}^{-(1+y^2)}, \quad x = 1, \ 0 < y < 2, \tag{12.13}$$

$$u = \mathrm{e}^{-x^2}, \quad 0 < x < 1, y = 0, \tag{12.14}$$

$$u = \mathrm{e}^{-(4+x^2)}, \quad 0 < x < 1, \ y = 2. \tag{12.15}$$

This model problem has solution

$$u = \mathrm{e}^{-(x^2+y^2)}.$$

12.3.2 Finite Difference Methods for Boundary Value Problems in Two Dimensions

To define the finite differences that approximate the partial derivatives of a function in two dimensions, we first need to define a finite difference grid. We have already stated that we are assuming that the function u that is to be determined satisfies a partial differential equation defined on the region $X_0 \leq x \leq X_1, Y_0 \leq y \leq Y_1$. We now suppose that there are points x_i, $i = 1, 2, \ldots, M$ and y_j, $j = 1, 2, \ldots, N$ such that

$$
\begin{aligned}
&x_1 = X_0, \\
&x_1 < x_2 < x_3 < \cdots < x_M, \\
&x_M = X_1, \\
&y_1 = Y_0, \\
&y_1 < y_2 < y_3 < \cdots < y_N, \\
&y_N = Y_1.
\end{aligned}
$$

The nodes of the finite difference grid are then the points (x_i, y_j), $i = 1, 2, \ldots, M$, $j = 1, 2, \ldots, N$. The boundary nodes are the nodes where $x = X_0$, $x = X_1$, $y = Y_0$ or $y = Y_1$. All other nodes are interior nodes. An example mesh on the square $0 < x < 1$, $0 < y < 2$ is shown in Fig. 12.2, where the filled circles denote the boundary nodes, and the open circles denote the interior nodes.

Numbering of the nodes for a finite difference grid is slightly more complicated in two dimensions than it was in one dimension. For the finite difference grid in one dimension all nodes could be numbered consecutively, allowing the finite difference approximations to be written down in an intuitive way. To write down finite difference approximations in two dimensions, we will adopt the "compass point" notation shown in Fig. 12.3. The node immediately above node *i* in the computational mesh is denoted by *i*, *N*, where "*N*" corresponds to north. The other nodes that are adjacent to node *i* are the *east*, *south* and *west* nodes, denoted by "*i*, *E*", "*i*, *S*" and "*i*, *W*" respectively.

Provided *i* is an interior node, the four adjacent nodes shown in Fig. 12.3 all exist. Finite differences to the derivatives that appear in Poisson's equation are given below.

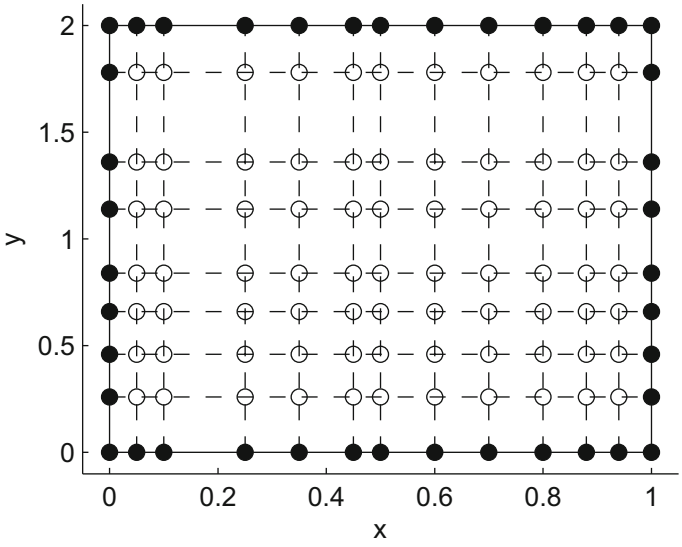

Fig. 12.2 A suitable finite difference grid in two dimensions. Boundary nodes are denoted by a *filled circle*, interior nodes by a *hollow circle*

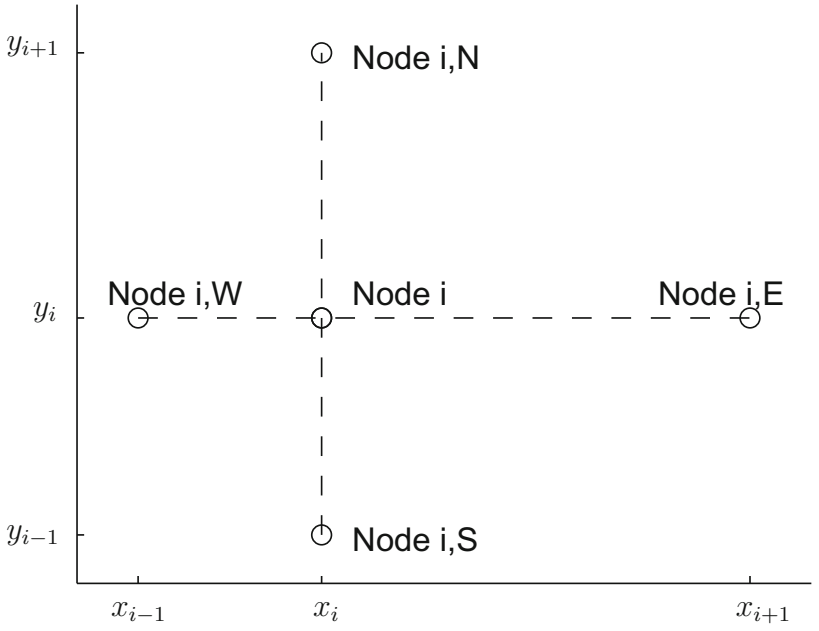

Fig. 12.3 Node i and points used to calculate finite difference approximations in two dimensions

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{2}{x_{i,E} - x_{i,W}} \left(\frac{u_{i,E} - u_i}{x_{i,E} - x_i} - \frac{u_i - u_{i,W}}{x_i - x_{i,W}} \right), \tag{12.16}$$

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{2}{y_{i,N} - y_{i,S}} \left(\frac{u_{i,N} - u_i}{y_{i,N} - y_i} - \frac{u_i - u_{i,S}}{y_i - y_{i,S}} \right). \tag{12.17}$$

We will now explain how these finite difference approximations may be used to set up a linear system to calculate the numerical solution of Poisson's equation.

12.3.3 Setting Up the Linear System for the Model Problem

We will now apply the theory developed in Sect. 12.3.2 to the model problem described in Sect. 12.3.1. Using the finite difference grid described in Sect. 12.3.2, we have M nodes in the x-direction, and N nodes in the y-direction: that is, a total of $M \times N$ nodes. Each of these nodes has an unknown value of u, and so our linear system comprises $M \times N$ equations, with each equation being associated with one node of the mesh.

At interior nodes we may substitute the finite difference approximations given in Eqs. (12.16) and (12.17). Substituting these approximation into Eq. (12.11) and rearranging yields

$$\alpha_i u_i + \alpha_{i,N} u_{i,N} + \alpha_{i,E} u_{i,E} + \alpha_{i,S} u_{i,S} + \alpha_{i,W} u_{i,W} = b_i, \tag{12.18}$$

where

$$\alpha_i = -\frac{2}{(x_{i,E} - x_i)(x_i - x_{i,W})} - \frac{2}{(y_{i,N} - y_i)(y_i - y_{i,S})},$$

$$\alpha_{i,N} = \frac{2}{(y_{i,N} - y_{i,S})(y_{i,N} - y_i)},$$

$$\alpha_{i,E} = \frac{2}{(x_{i,E} - x_{i,W})(x_{i,E} - x_i)},$$

$$\alpha_{i,S} = \frac{2}{(y_{i,N} - y_{i,S})(y_i - y_{i,S})},$$

$$\alpha_{i,W} = \frac{2}{(x_{i,E} - x_{i,W})(x_i - x_{i,W})},$$

$$b_i = -4(1 - x_i^2 - y_i^2)\mathrm{e}^{-(x_i^2+y_i^2)}.$$

The value of u at each boundary node is given by the appropriate equation from Eqs. (12.12)–(12.15). This may be incorporated into the linear system by the equation

$$u_i = b_i, \tag{12.19}$$

where i is a boundary node, and b_i is the value that u takes at that node.

Equations (12.18) and (12.19) fully define the linear system. We may now use the functionality of the classes of vectors, matrices and linear systems developed in Chap. 10 to calculate the value of the finite difference approximation to u at each node.

12.3.4 Developing the Classes Required

We give only minimal guidance on developing the classes required for calculating a numerical solution of Poisson's equation. Designing and implementing these classes is left as an exercise (Exercise 12.4). Our suggestions are given below.

- Creating an instance of the class `FiniteDifferenceGrid` should require the use of a constructor that specifies the number of nodes in the x direction and the number of nodes in the y direction. The grid should consist of a vector of boundary nodes that are all instances of the class `BoundaryNode` (discussed below) and a vector of interior nodes that are all members of the class `InteriorNode` (also discussed below). Each of the nodes in the mesh should have a global numbering that will refer to the row number of the matrix that will correspond to the unknown value of u at that node, u_i.
- An instance of the class `BoundaryNode` will have an integer representing the global numbering, and a double precision floating point variable that represents the value of u at that node from the boundary conditions.

- An instance of the class `InteriorNode` will have an integer representing the global numbering, and the global numbers of the north node, east node, south node and west node: see Fig. 12.3 for a definition of these nodes.

The classes described above, together with a class for encapsulating the partial differential equation that is similar to `SecondOrderOde` in Sect. 12.2, should enable code to be written to calculate the numerical solution of Poisson's equation.

12.4 Tips: Using Well-Written Libraries

In Chap. 10 we developed a linear system class that was based on classes of vectors and matrices. These classes allowed us to perform various linear algebra operations. In this chapter, we utilised these classes to allow us to develop libraries for calculating the numerical solution of boundary value ordinary differential equations.

Although the classes developed in Chap. 10 do have sufficient functionality for the purpose of this chapter, we would recommend that a reader who requires a linear algebra library should consider using one of the many high quality, open-source libraries that are available. (Indeed, in Sect. 1.1.2, we gave the fact that there is a wealth of numerical libraries for scientific computing as one of the reasons for learning C++.) Libraries for linear algebra usually include significantly more functionality than that developed here including, for example: sparse matrices; a wide variety of iterative linear solvers; a wide variety of preconditioners; interfaces with other packages; and support for parallelisation. Indeed, as linear algebra is such a fundamental topic at the core of scientific computing, it is unlikely that any functionality required will not be included in a widely used library. Furthermore, such libraries have the advantage of being well-tested, optimised code and can, as such, be treated as a black box.

One open-source library that is of particular use is the Portable Extensible Toolkit for Scientific Computing (PETSc, pronounced "pet see") which is available for download from https://www.mcs.anl.gov/petsc/. Libraries such as PETSc include an extremely large amount of functionality for systems of both linear and nonlinear equations, with support for parallel implementation on distributed memory architectures through the MPI library.

We conclude this section by reminding the reader of our the remarks in Sect. 1.1.4. We explained in that section that this book focuses on aspects of the C++ programming language that are commonly needed when writing software for scientific computing applications. As such, we haven't touched on the functionality of the language that is rarely required in this field. Should readers wish to develop their C++ skills to use more advanced features we have given a list of suitable references in the Further Reading at the end of this book [5–10].

12.5 Exercises

12.1 Develop the classes described in Sect. 12.2 for second order, constant coefficient, linear boundary value ordinary differential equations. Test these libraries using

the model problems described in Sect. 12.1.1. The code in Listing 12.6 which produces output files that can be readily plotted may be used as a framework. Example solutions for this problem are given in Sect. C.2: these files, together with the header files given in this chapter, may be downloaded from http://www.springer.com/book/9783319731315.

Make sure that the `BvpOde` method `WriteSolutionFile` does not attempt to write a file if `mFilename` is uninitialised or set to an empty string. (This may be achieved by setting `mFilename` to a safe value in the constructor.)

12.2 Extend the library developed in Exercise 12.1 so that the user may specify a nonuniform finite difference grid. Allow this to be done through a method `SetGrid` of the class `FiniteDifferenceGrid` that allows a mesh to be specified as a vector of ordered nodes. Ensure that the boundary nodes have the same value as `mXmin` and `mXmax` in the class `SecondOrderOde`.

12.3 Some programmers may feel that the constructor given in Listing 12.1 is inadequate. They may argue that it would be easy to incorrectly assign one of the coefficients of the equation. One way around this would be to force the user to use a default constructor. Additional class members, such as a Boolean variable `mCoeffOfUxxIsSet` could be deployed. The default constructor would be overridden so that these variables were set to `false` when the constructor was called. A method called `SetCoefficientOfUxx` would then be written, which would have as input the coefficient of $\mathrm{d}^2u/\mathrm{d}x^2$. This method would assign the coefficient correctly and set the Boolean variable `mCoeffOfUxxIsSet` to `true`. Before the methods that calculate the numerical solution are called a check would be carried out to ensure that all required data has been assigned. Design, and implement, classes to specify the differential equation in this way.

12.4 If you understand the theory for finite difference methods for Poisson's equation given in Sect. 12.3.2, develop a library for solving such equations. Test this library using the model problem described in Sect. 12.3.1.

12.5 Exercise 12.1 asks you to develop the classes described in Sect. 12.2 and to test these libraries using the model problems described in Sect. 12.1.1. For this purpose Listing 12.6 gives a program `Driver.cpp`. This way of "testing" is not ideal because it relies on the manual step of checking that the data in the output files matches the expected solution.

Automate the process of testing the classes described in Sect. 12.2 by rewriting the testing functionality within a testing framework such as `CxxTest`. For each model problem you should produce a testing function which runs the problem, reads the output file back into a suitable data structure, and tests that the solution is correct: that is, the solution is close to the analytic form given in Sect. 12.1.1. Think about what the expected error might be for this numerical scheme.

An example solution to this problem is given in Listing C.9 in Sect. C.2.