Bull. Nov. Comp. Center, Comp. Science, 9 (1998), 23–30

Object-oriented approach in approximation of the boundary value problems

Yana L. Gurieva

Several classes of objects for the object-oriented approach in programming the approximation stage of solving the boundary value problems are proposed. They are the classes for a computational cell, for boundary conditions and coefficients of the linear system of equations.

Introduction

The idea of the modern technology of programming is many-sided. One of its components is the the object-oriented (OO) programming (OOP). OOP, as well as the object-oriented methodology in general, has some distinctive features (see, e. g., [1, p. 411]).

To simulate a problem using OOP, one must indentify objects and classes of objects which differ in their data and functions. If the classes share common properties, a base class is introduced from which to derive them. Inheritance is the tool for reusability in OOP. Virtual functions provide a way to tackle a class-specific behavior of the objects. Virtual functions provide polymorphism, i. e. different subclasses perform the same operation differently. A famous programming language which has all these features is C++.

There are several stages in numerical modelling of the problems of mathematical physics. They are: input/preprocessing of all necessary information about the problem, mesh generation, approximation, solving the system of linear equations and output/postprocessing. Each of them is a nontrivial numerical and algorithmic problem as well as their programming and especially their OO implementation. The main problem here is to define suitable numeric objects and the corresponding classes in such a way that they can be used in several applications.

The works connected with OOP can be found for three stages of the solution process. Two of them are input and output. Usually they are connected with some user graphic interface and their OO implementation is based on the classes of graphic objects to present input and output of the boundary value problem on a display screen (see, e. g., [2]).

The third stage is the mesh generation and, in particular, the adaptive mesh refinement. In paper [3], to program the adaptive mesh refinement algorithm for the 3D calculations in gas dynamics, the authors introduced the classes to handle the geometric rectangular mesh regions and a special class which is "... C++ implementation of a Fortran array". The latter is very typical of computational

applications which "... act upon regular arrays of numbers" and perform integration, interpolation, etc., because for such operations Fortran is "an excellent choice of language".

In the present paper the OO approach is applied via using C++ to the approximation stage of the solution process and introducing several classes of objects. Some remarks on reusing the proposed classes will be given.

1. The linear system coefficients

Let the approximation of the 2D boundary value problem for the Helmholtz equation be the box approximation [4] on a rectangular nonuniform mesh and the computational region be a rectangle.

The approximation process can be considered consisting of two parts: approximation of the equation itself and taking into account the boundary conditions via the corresponding approximation. The result of the approximation is the linear system of equations

$$Au = f,$$

where the matrix A is a sparse $(N \times N)$-matrix, i. e. it has only m ($m \ll N$) nonzero entries in each row. Than each row corresponding to the node (i, j) has the following form:

$$p_0 u_0 - \sum_{k=1}^{m-1} p_k u_k = f_0, \qquad (*)$$

where m is 5 or 9 according to the difference pattern used and 0 is the local number in this mesh pattern for the node. The matrix A can be presented as a set of m arrays of the coefficients p_k, each being $(N \times N)$-matrix. It should be mentioned that this representation is very convenient for the approximation process namely (for the system solution, the famous universal storage scheme is the sparse row-wise format and its modifications [5]). Let us assume that there is defined a matrix class, say, `Matrix`, performing all necessary operations to handle any rectangular matrix (we will need here only the constructor of this class; let it be the constructor which has only one argument – the name of the matrix). Than it is natural to introduce the following special class for the coefficients (i. e. for the matrix A):

```
class Coef
{
private:
  void CheckSize(int xdim, int ydim) const
  {
    if( Cols != 0 ) assert(xdim == Cols);
    if( Rows != 0 ) assert(ydim == Rows);
  }
protected:
  int Cols, Rows;
public:
  Matrix<real> p0, p3, p4, p7, p8, p1, p2, p5, p6, f, u;
```

```
public:
  Coef() : p0("p0"), p3("p3"), p4("p4"), p7("p7"), p8("p8"),
           p1("p1"), p2("p2"), p5("p5"), p6("p6"),
           f("f"), u("u"), Cols(0), Rows(0) {}
  int GetCols() const { return Cols; }
  int GetRows() const { return Rows; }
  void create034(int xdim, int ydim);
  void create125678(int xdim, int ydim);
  void create(int idimx, int idimy);
};
```

Here and later we omit the functions dealing with input and output of the class components. We use and will use only the default constructors because our interest is in the class data and class functions which perform some computations. So, the class handles a set of rectangular matrices of the size `Cols`×`Rows`. Three functions `create034`, `create125678` and `create` are necessary to create different subsets of coefficients: if the matrix A is symmetric and $m = 4$ than only three coefficients p_0, p_3, p_4 are sufficient to represent it; otherwise, all nine coefficients including p_1, p_2, p_5, p_6, p_7, p_8 should be created. The latter function from the class just calls the first two functions. The main goal of these functions is to allocate memory for the corresponding coefficients. This class is "dummy" in the sense that the only useful thing in creating it is to have the arrays of coefficients of the same size. However, this class is reusable or can be made reusable by adding several new coefficients because the number of non-zero coefficients of the matrix does not depend on a specific way or a method of approximation. It depends only on the approximation pattern and the incidence of the mesh nodes.

2. Mesh and computational cell

In papers [6, 7], the matrix A was proposed to be built in a cell-by-cell traversing of mesh cells. In doing so, the local (4×4)-matrix was introduced, and all necessary calculations for the approximation were done locally using the mesh cell information. This information consists of the coordinates of the four corner nodes of a rectangular cell, the medium coefficient in the cell and the four numbers of the boundary edges on which lie the cell edges. The medium coefficient and coordinates are necessary to approximate the equation, and numbers of boundary edges – to take into account the boundary conditions if any. We assume that the boundary conditions are given on the boundary edges of the computational region. Let us assume that the mesh is given by the two global arrays of x- and y-mesh coordinates. Then having the mesh indexes (i, j) of the point, its coordinates are available at any place of the program. They are used to calculate the local balance matrix. So, the class corresponding to the mesh cell can be introduced as follows:

```
class Cell
{
public:
  float med;
```

```
  int left, down, right, up;
public:
  Cell() : med(-1), left(-1), down(-1), right(-1), up(-1) {}
};
```

This class is "dummy" in the sense that the matrix of the cells is just a union of five Fortran arrays. The computational domain can be presented as matrix of the computational cells.

3. Boundary conditions

The boundary conditions which are necessary to be taken into account consist of two types of conditions. The first are the Neumann and the Newton conditions. They require integration over the cell. The second type is Dirichlet condition when the given value is put into the right-hand side of the equation (*) for the point on the boundary edge with this boundary condition, and actually there are no additional calculations.

The following classes to perform all types of the boundary conditions are proposed. Let the boundary conditions for the problem under consideration include the constant Dirichlet conditions, the linear Dirichlet conditions, the Dirichlet conditions given by a user function and the Neumann conditions. At first, the enumeration type is introduced. It has all different types (which require different calculations) of the boundary conditions for the considered boundary value problem.

```
enum EFType
{
  BF_CONST, BF_LIN_DIR, BF_NEI, BF_USER, BF_LAST
};
```

Then we introduce the base class BoundCond of the boundary conditions as follows, assuming that there exists a special class, say, `IArray`, to handle the arrays of the integer values:

```
class BoundCond
{
protected:
  IArray edges;
public:
  BoundCond() : edges("edges") {}
  virtual ~BoundCond() {}
  virtual real Func(real &x, real &y) const = 0;
  virtual EFType GetType() const = 0;
  Bool HasRib(int rib_number) const;
};
```

The class contains one array "edges" of the integer values which are the numbers of the boundary edges of the computational region. These edges describe the geometry of the region, i.e., the region boundary, differ in their numbers and each edge

has a certain boundary condition. We use here a pure virtual function **Func**, which will return the values of the coefficients of different boundary conditions to make this class abstract. The virtual function **GetType** is necessary to determine the type of the boundary condition on the given edge. The function **HasRib** is necessary to determine if the edge with the given number has any boundary condition or has not, i.e. if this edge is in the array "edges" (the boundary edges of the computational region can have no boundary condition if they divide the subregions with the different media, i.e. they are the edges of the "inner" boundary). Then we define the derivative classes with their own definition of the functions **func** and **GetType**.

```
class ConstDir : public BoundCond
{
protected:
  real coef;
public:
  ConstDir() : coef(0.0) {}
  virtual real Func(real &x, real &y) const { return coef; }
  virtual EFType GetType() const { return BF_CONST; }
};

class LinDir : public BoundCond
{
protected:
  real a, b, c;
public:
  LinDir() : a(0.0), b(0.0), c(0.0) {}
  virtual real Func(real &x, real &y) const { return a*x+b*y+c; }
  virtual EFType GetType() const { return BF_LIN_DIR; }
};

class Nei : public BoundCond
{
protected:
  real kappa, g;
public:
  Nei() : kappa(0.0), g(0.0) {}
  virtual real Func(real &x, real &y) const
  { x = kappa; y = g; return real(0); }
  virtual EFType GetType() const { return BF_NEI; }
};

class User : public BoundCond
{
protected:
// any data needed
public:
  User() {}
  virtual real Func(real &x, real &y) const
```

```
// call here any extern global function that is needed
  { return real(0); }
  virtual EFType GetType() const { return BF_USER; }
};
```

So, the function **GetType** returns for each class its own value of the type and the function **Func** performs necessary calculations for the class where it is defined and/or returns the values of the coefficients of the boundary condition.

The real boundary conditions (pointers to them) will be stored in a special array (we assume here the existence of the class for array of pointers, say, **ArrayPtr**):

```
class CondArray : public ArrayPtr<BoundCond *>
{
public:
  CondArray(const char *name = "") : ArrayPtr<BoundCond *>(name) {};
  BoundCond* GetRib(int ri) const;
};
```

Function **GetRib** returns the type of the boundary condition on the boundary edge with given number **ri**. Now we can write down a function which performs all necessary calculations for taking into account the boundary condition on the given boundary edge with the number **rib**. Its two ends have the mesh indexes (i_1, j_1) and (i_2, j_2) and coordinates (x_1, y_1) and (x_2, y_2):

```
void bc_on_rib(int rib, int i1, int j1, int i2, int j2,
               real x1, real y1, real x2, real y2, real s)
{
  BoundCond *p = condArray.GetRib(rib);
  switch( p->GetType() )
  {
    case BF_CONST:
    case BF_LIN_DIR:
      PCoef.u(i1,j1)=p->Func(x1,y1);
      PCoef.u(i2,j2)=p->Func(x2,y2);
      PCoef.p0(i1,j1)=-1.;
      PCoef.p0(i2,j2)=-1.;
      break;
    case BF_USER:
// should be the same as BF_CONST, BF_LIN_DIR
      break;
    case BF_NEI:
      real w=area*.5*s;
      real kappa,g;
      p->Func(kappa,g);
      PCoef.f(i1,j1)+=g*w;
      PCoef.f(i2,j2)+=g*w;
      p00+=kappa*w;
      p44+=kappa*w;
```

```
      break;
    default:
      exit(1);
  }
}
```

Here `PCoef` is an object of the introduced type `Coef`.

In the case of the Dirichlet boundary conditions, the given value, calculated via the function `func` (the function is determined by the boundary condition!), is put into the solution array u and the diagonal coefficient p_0 of equation $(*)$ is set to (-1). In the case of the Neumann conditions of the form $æu + \partial u/\partial n = g$, some calculations are done to give additions to the diagonal coefficients of the local matrix and the right-hand sides corresponding to the two points of the boundary edge.

The final function which takes into account the boundary conditions on all the four edges of the mesh cell with its low-left corner with the mesh indexes (i, j) is very simple:

```
void bc_on_edges(int i, int j, const Cell &cell)
{
  int i1=i++;
  int j1=j++;
  bc_on_rib(cell.left,i,j,i,j1,x(i),y(j),x(i),y(j1),cell.med);
  bc_on_rib(cell.down,i,j,i1,j,x(i),y(j),x(i1),y(j),cell.med);
  bc_on_rib(cell.right,i1,j,i1,j1,x(i1),y(j),x(i1),y(j1),cell.med);
  bc_on_rib(cell.up,i,j1,i1,j1,x(i),y(j1),x(i1),y(j1),cell.med);
}
```

Here we assume that the global arrays of the coordinates are defined before this function is called. The `cell` (an object of the `Cell` class) is used to get its four mesh edges and the medium value.

4. Conclusion

One can see from the above constructions that in most cases the computational classes are trivial because they should deal with the regular arrays of numbers. The only exception are the introduced classes of the boundary conditions. They do use the possibility which gives the virtual function. The classes of the boundary conditions do not depend on a specific computational application (e. g., the approximation approach) and depend only on the boundary value problem, or, to be precise, on the types and the forms of the boundary conditions, and so they are reusable.

One of the most important components of the modern technology of programming is the parallel implementation of algorithms for multiprocessor computers. The parallelization has been already done mostly for solving the linear and nonlinear systems of equations arising in numerical applications (see [8] for example). But similar considerations can be made for the approximation process and the corresponding programs. For the approximation of the boundary value problems,

parallelization can be done on different levels. Concerning the OO approach and the introduced classes, parallelization is naturally implemented for the base classes handling arrays and matrices. The next level is the parallelization of calculation of the local balance matrices. Then the cell-by-cell traversing mentioned in Section 3 is replaced by the distributing process, when a computational region is divided into several subregions having approximately the equal number of mesh cells (the number of arithmetic operations for different cells is one and the same except the cells near the boundary). Each subregion is supposed to be processed by its own processor. Another kind of parallelization can be performed, as was mentioned, for the solution of the resulting system of the linear equations. Careful consideration of these important possibilities should be the main subject of the future work.

References

[1] P. DiLascia, *Windows++: writing reusable Windows code in C++*, ISBN 0-201-60891-X, Second printing, 1993.

[2] N.A. Orishich and A.V. Russkov, *Interactive interface for modeliing two-dimensional boundary-value problems*, Proceedings of the Young Scientists Conference, Novosibirsk, 1997, 147–156 (in Russian).

[3] W.Y. Crutchfield and M.L. Welcome, *Object-oriented implementation of adaptive mesh refinement algorithms*, Scientific programming, **2**, 1993, 145–156.

[4] V.P. Il'in, *Balance difference schemes of high order on nonuniform rectangular grids*, Preprint 1031, Novosibirsk, Computing Center, 1994 (in Russian).

[5] S. Pissanetzky, *Sparse Matrix Technology*, Academic press Inc., 1984.

[6] Y.L. Gurieva and V.P. Il'in, *On the finite-volume technology for the mixed boundary-value problems*, Advanced Mathematics: Computations and Applications. Proceedings of the International Conference AMCA-95, Novosibirsk, NCC Publisher, 1995, 650–655.

[7] Y. Gurieva and V. Il'in, *Finite volume approaches for 2-D BVPs: algorithms, data structures, software and experiments*, Univ. of Nijmegen, Dep. of math., Rep. No. 9715, 1997.

[8] I.N. Molchanov, *Introduction to the Algorithms of Parallel Calculations*, Naukova Dumka, Kiev, 1990 (in Russian).