Polymorphism — implementing boundary conditions using virtual functions

Jan van Dijk

February 19, 2004

Contents

Abstract

In this text we shall discuss how C++ *classes*, *class derivation* and *polymorphism* can be used for the implementation of code which handles the boundary conditions of discretised Φ-equations. The code which will be presented here is based on the code which has been used in the practical work which is part of the numerical plasma course.

1 The Φ Equation

In the numerical plasma physics course we have seen many examples of convection-diffusion equations. These are second-order partial differential

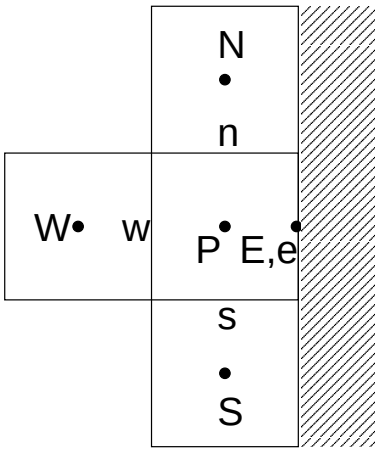

Figure 1: Boundary grid point layout. Note that the distance between the last interior point and the boundary is half the cell size. The boundary point lies on the eastern control volume face.

equations which, in a generalised form, can be written as

$$\frac{\partial \alpha \Phi}{\partial t} + \vec{\nabla} \cdot \rho \mathbf{V} \beta \Phi - \vec{\nabla} \cdot D \nabla \Phi = S_\Phi. \tag{1}$$

The discretisation of such equations has been discussed extensively, the result is that in each interior point an equation of the following form arises:

$$a_P \Phi_P = \sum_{nb} a_{nb} \Phi_{nb} + b. \tag{2}$$

In these equation P denotes a *nodal grid point*, while a_P and a_{nb} are the discretisation coefficients. The subscript nb denotes the direct neighbours of P. The exact form of the coefficients a_{nb} and a_P does not matter here, but we mention the property $a_P = \sum_{nb} a_{nb} \Phi_{nb}$. Finally, b is the source term, integrated over the control volume.

2 Formulation of the Boundary Conditions

In order to solve the equation, we must provide *boundary conditions*. We may provide either the *value* of the variable Φ at the boundaries (Dirichlet conditions), the derivative (Neumann conditions) or use a mixed form (Cauchy conditions).

We will consider the grid point layout as depicted in figure 1. Suppose that we wish to impose a Dirichlet boundary condition at the eastern grid

boundary, E in the figure. How do we incorporate the known value in the discretised equation for the point P next to the boundary? This turns out to be trivial. We start with the discretised transport equation for the point P and isolate the term involving Φ_E,

$$a_P \Phi_P = \sum_{nb\prime} a_{nb\prime} \Phi_{nb\prime} + a_E \Phi_E + b. \tag{3}$$

Here the prime indicates that the sum runs over all neighbours except the point E. Now the value Φ_E is known, $\Phi_E = \Phi_b$, say. Substitution yields

$$a_P \Phi_P = \sum_{nb\prime} a_{nb\prime} \Phi_{nb\prime} + a_E \Phi_b + b. \tag{4}$$

In the computer code, we may implement this as the following sequence of steps (in this order!):

$$\begin{aligned} \Phi_E &\rightarrow \Phi_b; \\ b &\rightarrow b + a_E \Phi_b; \\ a_E &\rightarrow 0. \end{aligned} \tag{5}$$

The incorporation of Neumann conditions is not too different. Let us assume that the derivative at the eastern boundary is given by

$$\left. \frac{\partial \Phi}{\partial dx} \right|_E = F_0 + F_1 \Phi_E, \tag{6}$$

where F_1 is a negative number. Suppose that the distance between the nodal point and the eastern boundary point is given by δ. Then we may approximate the derivative as $(\Phi_E - \Phi_P)/\delta$ to obtain

$$\frac{\Phi_E - \Phi_P}{\delta} \approx F_0 + F_1 \Phi_E. \tag{7}$$

We may rewrite this equation to obtain the value at the eastern boundary point,

$$\Phi_E \approx \frac{\delta F_0 + \Phi_P}{1 - \delta F_1}. \tag{8}$$

Now we may proceed as in the case of Dirichlet boundary conditions. We substitute the value for Φ_E in equation 3 to obtain

$$a_P \Phi_P = \sum_{nb\prime} a_{nb\prime} \Phi_{nb\prime} + a_E \frac{\delta F_0 + \Phi_P}{1 - \delta F_1} + b. \tag{9}$$

Re-ordering terms yields:

$$\left(a_P - a_E \frac{1}{1-\delta F_1}\right) \Phi_P = \sum_{nb\prime} a_{nb\prime} \Phi_{nb\prime} + a_E \frac{\delta F_0}{1-\delta F_1} + b. \tag{10}$$

In this case, this algorithm can be implemented using the following steps (in this order):

$$\begin{aligned} \Phi_E &\rightarrow \frac{\delta F_0 + \Phi_P}{1-\delta F_1}; \\ b &\rightarrow b + a_E \frac{\delta F_0}{1-\delta F_1}; \\ a_P &\rightarrow a_P - a_E \frac{1}{1-\delta F_1}; \\ a_E &\rightarrow 0. \end{aligned} \tag{11}$$

In the next sections we will gradually develop a small C++ class hierarchy for tackling the problem of representing the boundary conditions. We will start with the traditional, procedural approach to show the advantages of using the class approach.

3 Hello, Boundary Condition

Let's start by implementing the boundary conditions as functions. If we look at the algorithm in the equations 5 and 11 we may come up with the implementations in figure 2. Note that we pass *pointers* to those variables we wish to modify. If we pass the variables themselves, we will modify local *copies* of the variables[1].

In a number of steps we will transform this code into a small set of equivalent C++ classes. In this process we have an opportunity to demonstrate the concepts of *class derivation* and *polymorphism*.

4 Using Classes

One disadvantage of the functions in figure 2 is the number of parameters that must be passed each time the functions are called. Unlike a class, a function cannot be told to remember a context in which it is called: a function cannot have data members.

[1]Experienced C++ people would prefer the less-familiar concept of references, but that does not change the idea presented here.

```
void  apply_dirichlet (double bval,
                       double* phiB,
                       double* aB,
                       double* b)
{
        *phiB  = bval;
        b      += (*aB)*bval;
        *aB    = 0;
}

void apply_neumann(double F0,
                   double F1,
                   double delta,
                   double phiP,
                   double* phiB,
                   double* aP,
                   double* aB,
                   double* b)
{
        *phiB  = (delta*F0 + phiP)/(1-delta*F1);
        b      += (*aB)*delta*F0/(1-delta*F1);
        *aP    -= (*aB)/(1-delta*F1);
        *aB    = 0;
}
```

Figure 2: The implementation of the Dirichlet and Neumann boundary conditions (equations 5 and 11) using good-old functions. Note that those variables which are modified by the functions are passed as pointer, otherwise one would merely modify local *copies* of these data.

```
class BoundCondDirichlet
{
  public:
      BoundCondDirichlet(double bval,
                         double* aB,
                         double* b)
      {
            m_bval = bval;
            m_aB   = aB;
            m_b    = b ;
      }
      double Apply(double PhiP)
      {
            m_b      += (*m_aB)*m_bval;
            *m_aB    = 0;
            return m_bval;
      }
  protected:
      double  m_bval;
      double* m_aB;
      double* m_b;
};
```

Figure 3: The implementation of the Dirichlet boundary condition (equation 5) using a class.

```
class BoundCondNeumann
{
  public:
      BoundCondNeumann(double F0,
                       double F1,
                       double delta,
                       double* aP,
                       double* aB,
                       double* b);
      double Apply(double phiP);
  protected:
      double m_F0;
      // ...
};
```

Figure 4: The implementation of the Neumann boundary condition (equation 11) using a class. The implementations and most data declarations have been omitted here, but should be clear by comparison with the Dirichlet class in figure 3 and the Neumann function in figure 2.

For this reason we re-implement the boundary conditions as classes; the results can be seen in figures 3 and 4. The concepts presented there should be clear by now, we see data members, function members *Apply* and a *constructor*.

The advantage is that we can create the object once, providing all arguments. After creation, we can repeatedly call the member *Apply*, without having to provide the parameter list. For reasons we will make clear later, we will use pointers instead of 'normal' automatic variables.

```
// one time:
BoundCondDirichlet* dirbc = new BoundCondDirichlet( /* parameters */ );
// ...
// many times:
phiB = dirbc->Apply(phiP);
```

The same applies to the Neumann class.

5 Base Classes and Derivation

A disadvantage of the class implementations so far is that the relation between the Neumann and Dirichlet classes is not clear, and that both classes

```
class BoundCond
{
  public:
      BoundCond( double delta,
                 double* aP,
                 double* aB,
                 double* b);
  protected:
      double  m_delta,
      double* m_aP,
      double* m_aB,
      double* m_b;
};
```

Figure 5: An appropriate base class for boundary condition implementations. In such class data and functions can be made available which can be used by *derived classes*.

contain a lot of common code. A solution is to create a common *base class*. In C++, classes can be derived from other classes. In this process these *derived classes* inherit the properties of the *base class*.

One of the advantages of using base classes for common data and function members is the reduction in code size. Base class functionality is programmed once, then used by all derived classes. But what should be put in a base class? This is not exact science: the programmer has a choice and must take an appropriate design decision for each individual case.

We propose the base class declaration which is shown in figure 5. All data are relevant to the Neumann boundary condition, not all are used for the Dirichlet case. This decision is based on the observation that these are the data which are *sufficiently general*. As said, such decision are subjective and tend to be debated *ad infinitum*.

With the introduction of this base class, the derived classes for the Dirichlet and Neumann boundary conditions become much simpler. The new versions are shown in figure 6. The members *Apply* are as before, both classes use the data members which have been declared in the base class. This is possible because these data have been declared as *protected*: this means that only derived classes have such access rights.

```
class BoundCondDirichlet : public BoundCond
{
  public:
      BoundCondDirichlet(double bval,
                         double delta,
                         double* aP,
                         double* aB,
                         double* b)
       : BoundCond(delta, aP, aB, b)
      {
            m_bval=bval;
      }
      double Apply(double phiP);
  protected:
      double  m_bval;
};

class BoundCondNeumann : public BoundCond
{
  public:
      BoundCondNeumann( double F0,
                        double F1,
                        double* aP,
                        double* aB,
                        double* b)
       : BoundCond(delta, aP, aB, b)
      {
            m_F0=F0;
            m_F1=F1;
      }
      double Apply(double phiP);
  protected:
      double  m_F0;
      double  m_F1;
};
```

Figure 6: Here the boundary conditions have been derived from a common base class *BoundCond*, see figure 5. The constructors first call the base class constructors, then initialise the extra data members declared in the derived classes. The members *Apply* are exactly as before.

```
class BoundCond
{
  public:
      BoundCond( double delta,
                 double* aP,
                 double* aB,
                 double* b);
      virtual double Apply(double phiP)=0;
  protected:
      double  m_delta,
      double* m_aP,
      double* m_aB,
      double* m_b;
};
```

Figure 7: In a base class we may add pure virtual members. This way an interface can be made available on a basic level where the implementation is not yet available.

6 Base Classes and Polymorphism

In the section in which boundary condition classes were introduced, we have seen the typical use of the boundary condition objects:

```
BoundCondDirichlet* dirbc = new BoundCondDirichlet( /* parameters */ );
// ...
// many times:
phiB = dirbc->Apply(phiP);
```

In this code we can see that in fact we are dealing with a Dirichlet condition. Interestingly enough, at the location where *Apply* is called, we are not really interested in the type of the boundary condition. We wish to organise the code such that

- At the location where the boundary condition pointer is created, we select the boundary condition type;
- Wherever these pointers are used we don't need to know the exact type. We call a function *Apply* that takes Φ_P and returns Φ_b, without knowing how the calculation took place (Dirichlet or Neumann).

C++ provides a mechanism to achieve just that. Most people consider this the most difficult feature of object-oriented programming in C++, but things are really not that bad. The idea is that wherever we are using objects,

we only use the interface of the base class. In this interface, one or more members may be declared which are not yet available, but are required to be made available by derived classes. This syntax is as in figure 7.

The only things which has been changed is the addition of the *Apply* declaration. Don't try to understand this syntax, this just happens to be the way it is in C++. The word ***virtual*** means: if you call me, I will defer that call to the implementation in a derived class; the *=0* means: we do not provide an implementation ourselves.

Indeed implementations are present in the derived classes. In the Dirichlet and Neumann classes we simply add the word ***virtual*** in front of the *Apply* members[2].

How does this affect the usage of the classes? Well, we create a Neumann or Dirichlet object as before, but this time we assign the result to a *base class pointer*. This assignment happens to be valid C++, but also makes sense, since everything the base class has to offer is also available in the more capable derived class.

```
BoundCond* bc = new BoundCondDirichlet( /* parameters */ );
// ...
phiB = bc->Apply(phiP);
```

In the last line *bc* is a pointer to the base class. Still, this will silently call the Dirichlet boundary modification code.

This has huge advantages. *Most code can be written in terms of the base class properties, without detailed knowledge of the available derived classes.* Perhaps like the following snippet:

```
string  name = AskUser();
BoundCond* bc = CreateBoundCond( name, /* parameters */ );
// ...
phiB = bc->Apply(phiP);
```

In this example, we do not have a clue what boundary condition has been created exactly. But we can still use it. The only location where the details of the available boundary conditions matter is in the implementation of the function *CreateBoundCond*, which is listed in figure 8 We promised to get back to the point that we are only using pointers to base classes. Note that nowhere we create an object of the base class type, *BoundCond*. In fact, C++ forbids to create objects which do not implement one or more members. What version of *Apply* should be called? None has been provided in the base class. If you attempt to compile the code like

[2]For clarity, technically this is not necessary.

```
BoundCond* CreateBoundCond(string name, /* other parameters */ )
{
  if (name=="Dirichlet")
      return new BoundCondDirichlet( /* parameters */ );
  if (name=="Neumann")
      return new BoundCondNeumann( /* parameters */ );
  else
  {
    error ("Unknown boundary condition type " + name);
  }
}
```

Figure 8: The implementation of a *factory function*. This creates an object which is derived from *BoundCond*, depending on the name provided as argument.

```
BoundCond bc = BoundCondDirichlet( /* parameters */ );
```

you will simply get a compiler error. We cannot create objects of *abstract* types. As shown before, the version which uses only pointers compiles just fine.

7 Summary & Conclusions

The whole mechanism of introducing such *virtual* functions and accessing abstract properties via base class declarations is called *polymorphism*. It is one of the central concepts in object-oriented design.

Obviously, we have clear separation between interfaces and realisation. In a way, the ***virtual*** declaration in the base class is a contract: a promise to all users of that class that every derived class will provide an implementation of a member *Apply*, taking the indicated arguments and returning a value of the appropriate type.

All parts of the program that need boundary conditions can be implemented on the basis of this promise. No knowledge is needed about whatever Dirichlet and Neumann conditions may be available. Only where the boundary conditions are *created* such details are relevant.