C++ Design Patterns for Managing Parameters in Scientific Computing

Qinghai Zhang

School of Mathematical Sciences, Zhejiang University,
38 Zheda Road, Hangzhou, Zhejiang Province, 310027 China

Abstract

A scientific computing module usually has many parameters to control its behaviors. Instantiating a pair of get/set functions for each parameter clutters the interface, incurs code-bloating, overburdens maintenance, and invites subtle bugs. To solve these problems, two patterns are proposed for parameter management. The "enumeration index" pattern is a simple solution for parameters of a single type. When the parameters belong to multiple types, it is difficult for a statically-typed language to access/modify these parameters using a single pair of get/set functions. The "parameter manager" pattern achieves this by templates and multiple inheritance. It also handles additional semantics such as constancy, dependency, and default values in an easy-to-use manner: a client only has to specify the set of parameters and their associated properties. Both patterns are illustrated by a common task of solving the diffusion equation in the C++ language.

Key words: Design patterns, Enumeration-Indexing, Parameter-Manager

1 Motivation

Scientific programs crunch numbers, a huge amount of numbers. Sometimes the number of parameters that controls the behaviors of a scientific program might also be overwhelming. How to handle these parameters is the central topic of this paper.

A scientific problem can often be thought of as a set of smaller problems coupled together. The solution of each smaller problem can be captured either by a function or a *class*; the latter differs from the former in that it has the

Email address: qinghai@zju.edu.cn (Qinghai Zhang).

notion of *internal state*s afforded by its data members, or parameters. In the object-oriented programming (OOP) paradigm, one often faces two kinds of by-and-large choices: (i) how to break up the big problem into smaller ones, and, (ii) how to organize the individual modules to solve the big problem. A design pattern [3] contains a particular set of these structural choices, aiming to encapsulate possible changes of the big problem elegantly and efficiently.

The most common change to a scientific computation problem is probably varying its input parameters, the number of which is usually large in most practical applications. As an example, consider the diffusion equation in a rectangular `D`-dimensional domain,

$$\frac{\partial \phi}{\partial t} = \nu \nabla^2 \phi, \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^{\mathtt{D}}$ is the location vector, t the time variable, $\phi = \phi(\mathbf{x}, t)$ a continuous scalar function, and ν the dynamic viscosity. (1) is often solved by integrating the system of ordinary differential equations, obtained by replacing the Laplacian term $\nabla^2 \phi$ with a spatial discretization $L\Phi$, such as the widely-used central differencing $\frac{1}{h^2}(\phi_{i+1} + \phi_{i-1} - 2\phi_i)$. Within each time step, the following linear system is solved repeatedly as an approximation to the solution of (1):

$$(\alpha - \beta L)\Phi = \rho, \tag{2}$$

where the scalars α,β and the right-hand-side vector ρ usually depends on the spatial discetization and the time-integrator. See [5] and [6] for two examples using implicit Runge-Kutta methods. A program implementing this algorithm with an iterative linear solver has many input parameters, some of which are listed in Table 1.

One straightforward way to handle these parameters are the traditional `get/set` approach, which declares these parameters as `private` or `protected` data members of class `DiffusionSolver` and access/modify them using public member functions such as `getTimeOrder()`, `setTimeOrder()`, `getSpaceOrder()`, `setSpaceOrder()`, etc. Although the designers of the JavaBeans specification refer to this as the `get/set` pattern, the author agrees with Eckel [2] that this is just a commonplace naming convention rather than a design pattern. Furthermore, there are a number of problems with this `get/set` approach. Firstly, iterations on these parameters will have to repeat all these functions in boilerplate, e.g., consider printing all the parameters:

```
ostream& DiffusionSolver::print(ostream& os) const
{
  os << "timeOrder = "  << timeOrder  << endl;
  os << "spaceOrder = " << spaceOrder << endl;
  os << "t0 = "         << t0         << endl;
  os << "te = "         << te         << endl;
```

Table 1
Example parameters for solving the diffusion equation (1). `IntVect` and `RealVect` are vectors of integers and real numbers. Their sizes equal the dimensionality of the computational domain.

Parameter	Type	Identifier
temporal order-of-accuracy	`int`	`timeOrder`
spatial order-of-accuracy	`int`	`spaceOrder`
beginning time of simulation	`double`	`t0`
ending time of simulation	`double`	`te`
number of time steps	`int`	`numSteps`
number of ghost cells	`int`	`numGhosts`
converge criteria of iteration	`double`	`convergeRatio`
max number of iterations per time step	`int`	`maxNumCycles`
lower end of the problem domain	`RealVect`	`xLo`
higher end of the problem domain	`RealVect`	`xHi`
number of cells	`IntVect`	`numCells`
grid size	`RealVect`	`dx`
write solution to a HDF5 database?	`bool`	`writeHDF5Sol`
write error to a HDF5 database?	`bool`	`writeHDF5Err`
the directory to output results	`string`	`writeHDF5Dir`

```
    ... // repeat for all other parameters
}
```

This leads to "code bloating": the number of lines grows as that of the parameters. It also reminds us of the dubious "copy-and-paste" code-duplication in a subtle form. Secondly, this leads to "interface bloating": the number of member functions is twice as that of parameters. For classes with a large number of parameters, this might be an unacceptable violation of the fundamental OOP principle that the interface of a class be minimized [4]. Thirdly, hard-coding the names of these parameters make the maintenance of `class DiffusionSolver` tedious and error-prone; if one were to add or remove some parameters, she would have to go through all functions like `print` and add additional pairs of `get/set` functions to the interface of `class DiffusionSolver`.

Lastly, `DiffusionSolver` might have a constructor that takes all the parameters:

```
class DiffusionSolver {
```

```
public:
  DiffusionSolver(int a_timeOrder, int a_spaceOrder,
                  double a_t0, double a_te,
                  int a_numSteps, int a_numGhosts,
                  double a_convergeRatio, int maxNumCycles,
                  ...);
...
}
```

The task of checking the correct order of the sequence is tedious and error-prone. Worse, a bug caused by switching the order of parameters of the same type is very hard to catch since the compiler won't complain at all! In fact, if those two parameters have different orders in the header file and the implementation file, the compiler won't notice either! On top of these, just imagine adding or removing some parameters from the list!!

Often, a programmer frustrated by the `get/set` approach will just declare the parameters as public data members so that client programs can get/set them freely. Worse than the `get/set` approach, public members violate 'encapsulation', the most important principle of OOP. Also, if the value of a parameter has to be within a certain range to guarantee the correct behaviors of the module, it is easy to enforce and *hide* this semantics in a `set` function while it is complicated, if not impossible, to do so for the public member approach.

Ideally, we want to have a single pair of accessing/modifying functions as follows.

```
  /// pseudo signature for accessing function
  ParameterType get(ParameterName) const;
  /// pseudo signature for modifying function
  void set(ParameterName, ParameterType);

  /// pseudo code of looping on parameters
  for (Parameter p=ParameterIterator.begin();
       p!=ParameterIterator.end(); p++)
  {
   ... // do something with p
  }
```

In a dynamically typed language such as `python` and `lisp`, implementing the above pseudo-code is easy; however, in a statically typed language like C++, the specific types of `Parameter`, `ParameterName`, and so on must be resolved at compile time. To comply to this, we either require all parameters have the same type or employ a compile-time branching mechanism based on parameter type. In Section 2 and Section 3, the author proposes a pattern in each of these

two categories. The targeted use cases are scientific-computing modules with a large number of parameters and a small number of parameter types.

2 The Enumeration Index Pattern

The enumeration index pattern exploits the C++ enumeration construct by mapping a list of user-defined names to a set of consecutive and non-negative integers, which can be used as indices to parameter arrays. The following C++ code illustrates this pattern for the integer parameters in Table 1.

```
class IntegerManager {
public:
  enum ParmInt {
    timeOrder = 0, spaceOrder, numSteps,
    numGhosts, maxNumCycles, nParmInt
  };
  typedef std::map<ParmInt, int> MAP;

  static char const*const toStr(const ParmInt& p) {
    switch(p)
      {
      case timeOrder     : return "timeOrder    ";
      case spaceOrder    : return "spaceOrder   ";
      case numSteps      : return "numSteps     ";
      case numGhosts     : return "numGhosts    ";
      case maxNumCycles  : return "maxNumCycles ";
      default            : return "";
      };
  }

  int getInt(const ParmInt& p) const {
    return m_int[p];
  }
  void setInt(const ParmInt& p, int v) {
    m_int[p] = v;
  }
  void setInts(const MAPI& map) {
    for (MAPI::const_iterator i=map.begin(); i!=map.end(); i++)
      setInt(i->first, i->second);
  }

  ostream& print(ostream& os) const {
    for (int i=0; i<nParmInt; i++) {
```

```
      ParmInt pInt = static_cast<ParmInt>(i);
      os << toStr(pInt) << " = " << getInt(pInt);
    }
  }

private:
  int m_int[nParmInt];
};
```

After setting the value of the first enumerate to zero, the values of the rest enumerates are automatically assigned to consecutive integers. Consequently, the last enumerate in the list, `nParmInt`, can be used as the size of the integer array for storing the values of the integer parameters. Although an enumerate list can be mapped to string literals by preprocessing macros, a `toStr` function is shown here for clarity.

Compare to declaring two member functions for each parameter, this enumeration index pattern has a number of appealing advantages. Firstly, the number of get/set functions remains the same so long as the number of parameter *type*s is constant. Printing the parameters reduce to a loop on the enumeration list. Secondly, adding and removing a parameter can be achieved by simply changing the enumeration list and the corresponding `toStr` function; the changes automatically propagates to an iterative function such as `print` due to the enumeration mechanism. Thirdly, the function `setInts` can be used as a constructor to set all the `int` parameters in a much more semantically clear way. For example, the client program might read

```
typedef IntegerManager IM;
IM::MAP map;
map[IM::timeOrder]  = 4;
map[IM::spaceOrder] = 4;
... // other int parameters
IM im;
im.set(map);
```

The association between the keys and values is much easier and clearer than that of the get/set approach: it eliminates the tedious task of checking the correct order of parameters passed to the constructor.

Sometimes the parameters of a class are simple in that (i) they only have a single type (ii) they do not have additional semantics such as dependency on each other. In this case, the enumeration-indexing pattern is an excellent solution. However, as the number of types grows, the interface quickly becomes large since an enumeration list, a `toStr` function, and a pair of get/set functions are needed for each type; this is analogous to the bloated interface in

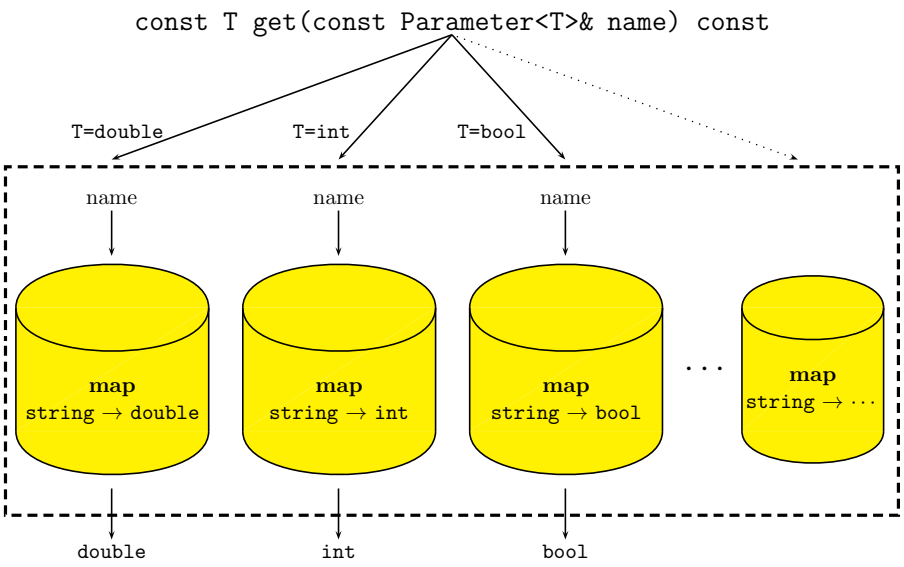

Fig. 1. The main idea of the parameter manager pattern. For each parameter type, `ParameterManagerN` has a corresponding associative container that takes the name of a parameter and returns its value in the particular type. The type of the parameter is resolved at compile time while the name of the parameter is passed at run time to the container of the specific type. The dashed box belongs to the interior of a `ParameterManagerN` object.

the simple get/set approach. In addition, it is difficult to enforce constancy or dependence of parameters. This is where the parameter manager pattern comes to help.

3 The Parameter Manager Pattern

The main idea of the parameter manager is illustrated in Fig. 1, where the `get` function is templated on the type of the parameter. Inside a `ParameterManagerN` object, an associative data structure, e.g. `std::map`, is instantiated for each possible type at compile time. At run time, the container with the specific type of the parameter is queried for the value of the parameter.

3.1 Capture the notion of 'parameter'

The first step to implement the above idea is to capture the notion of parameter in a templated class. This is also conducive to binding more semantics to a parameter such as constancy, dependency, and default values.

```
template<class T>
```

```
class Parameter {
public:
  Parameter(const std::string& key,
            const T& defaultValue = T(),
            bool constant   = false,
            bool dependent  = false) :
    key_(key),
    defaultValue_(defaultValue),
    constant_(constant),
    dependent_(dependent){}

  bool isDependent(void) const { return dependent_; }

  bool isConstant(void) const  { return constant_;  }

  char const*const key(void) const { return key_.c_str(); }

  const T defaultVal(void) const  { return defaultValue_; }

  friend bool operator<(const Parameter<T>& p1,
                        const Parameter<T>& p2) {
    return (p1.key() < p2.key());
  }

private:
  std::string key_;
  T           defaultValue_;
  bool        constant_;
  bool        dependent_;
};
```

Now that `Parameter` has its own abstraction layer, one can easily append more semantics to it.

The next step is to map parameters to their values. For this purpose, `std::map` is an excellent choice and we wrap around it a class `ParameterManager<T>`, with the template parameter class `T` the same as that of `Parameter<T>`. Although the key of `std::map` in the following code is `Parameter<T>` instead of `string` in Fig. 1, they are essentially the same since the compare operator of the `Parameter` class is based on its name as a `string`. Hence, two different parameters must not have the same key.

```
template<class T>
class ParameterManager {
public:
```

```
  typedef std::map<Parameter<T>, T>        MAP;

  virtual ~ParameterManager(){}

  const T get(const Parameter<T>& p) const
  {
    typename MAP::const_iterator mit = map_.find(p);
    if (mit==map_.end()){
      std::cerr << "Warning: Parameter " << p.key()
                << " not set yet! Return default value...\n";
      return p.defaultVal();
    }
    return mit->second;
  }

  void set(const Parameter<T>& p, const T& v)
  {
    if (!p.isConstant())
      map_[p] = v;
    // initialize a constant p, check value
    else if (map_.find(p) == map_.end() && v==p.defaultVal())
      map_[p] = p.defaultVal();
    else
      std::cerr << "Warning: " << p.key() << " = "
                << p.defaultVal() << " is already set!\n ";
  }

  void print(std::ostream& os) const
  {
    for (typename MAP::const_iterator mit = map_.begin();
         mit != map_.end(); mit++)
      os << mit->first.key() << " = "
         << mit->second << std::endl;
  }

protected :
  MAP map_;
};
```

If a parameter is not initialized yet, `ParameterManager::get` will issue a warning and return the default value of the parameter; if a constant parameter is already initialized to its default value, `ParameterManager::set` will also issue a warning for another attempt to set the value. Compare to `getInt/setInt` of `SolverManager` in Section 2, the additional semantics makes the contract more precise and helps to enforce the correct behaviors of the class.

Within one type `T`, we have achieved using a single pair of get/set functions to access/modify the parameters. Also, this provides us with enough ammunition to address the problem of multiple types. Consider the parameters listed in Table 1 in Section 1. Conceptually we are trying to gather a number of `ParameterManager<T>` instances, each corresponding to a certain type, into one entity. The goal of still having a single pair of get/set functions can be achieved by function overloading and multiple inheritance, thanks to `Parameter` being a templated and strongly-typed class. Here, the ambiguity is removed by branching different parameter types to different parameter managers.

```
template<class T1, class T2>
class ParameterManager2 : private ParameterManager<T1>,
                          private ParameterManager<T2>
{
public:
  virtual ~ParameterManager2(){}

  const T1 get(const Parameter<T1>& p) const {
    return ParameterManager<T1>::get(p);
  }
  const T2 get(const Parameter<T2>& p) const {
    return ParameterManager<T2>::get(p);
  }

  void set(const Parameter<T1>& p, const T1& v) {
    ParameterManager<T1>::set(p,v);
  }
  void set(const Parameter<T2>& p, const T2& v) {
    ParameterManager<T2>::set(p,v);
  }

  friend std::ostream& operator<<
  (std::ostream& os, const ParameterManager2<T1, T2>& pmm)
  {
    pmm.ParameterManager<T1>::print(os);
    pmm.ParameterManager<T2>::print(os);
    return os;
  }
};
```

Note that the private inheritance closes all the interfaces of individual pa-

rameter managers and the ambiguity of multiple inheritance is resolved by function overloading. Without this explicit branching, calling a get/set function of `ParameterManager2` will result in a compilation error.

For `N` bigger than 2, `ParameterManagerN` can be defined similarly. The code-bloating caused by the repetition can be alleviated by defining the following preprocessing macros.

```
/// define a non-constant, independent parameter with default value.
#define PM_DEF(TYPE, NAME, VALUE)                                    \
  const Parameter<TYPE> PM_CLASSNAME::NAME(#NAME, VALUE)

/// define a constant parameter
#define PM_DEFC(TYPE, NAME, VALUE)                                   \
  const Parameter<TYPE> PM_CLASSNAME::NAME(#NAME, VALUE, true)

/// define a dependent parameter
#define PM_DEFD(TYPE, NAME, VALUE, C)                                \
  const Parameter<TYPE> PM_CLASSNAME::NAME(#NAME, VALUE, C, true)

/// initialize a parameter to its default value.
#define PM_INIT(P) \
  set(P, P.defaultVal())

#define PMM_GET(N)                                                   \
  const T##N get(const Parameter<T##N>& p) const {                   \
    return ParameterManager<T##N>::get(p); }

#define PMM_SET(N)                                                   \
  void set(const Parameter<T##N>& p, const T##N& v) {                \
    return ParameterManager<T##N::set(p,v);}
```

The stringizing operator `#` encloses the argument in double quotation marks, thus `#NAME` will expand to a string literal at compile time. The token-pasting operator `##` concatenates its argument to tokens immediately before it to form a new token. For example, `T##N` will expand to `T1`, `T2`, and so on depending on the value of the argument `N`. The above macros reduce repetition and clarify the code. Using them, `ParameterManager3` is defined as follows.

```
template<class T1, class T2, class T3>
class ParameterManager3 : private ParameterManager<T1>,
                          private ParameterManager<T2>,
                          private ParameterManager<T3>
{
public:
```

```
  PMM_GET(1)
  PMM_GET(2)
  PMM_GET(3)

  PMM_SET(1)
  PMM_SET(2)
  PMM_SET(3)

  friend std::ostream& operator<<
  (std::ostream& os, const ParameterManager3<T1, T2, T3>& pmm) {
    pmm.ParameterManager<T1>::print(os);
    pmm.ParameterManager<T2>::print(os);
    pmm.ParameterManager<T3>::print(os);
    return os;
  }
};
```

In `ParameterManager3`, the insertion operator '`<<`' is *serial* in that the corresponding function `ParameterManager<T>::print` is called for each type within *one* scope; a `get` or `set` function is *parallel* in that the macros generate *three* functions for it.

Alternatively, when `N` is large, `ParameterManagerN` can be automatically generated by meta-programming, the main enabling mechanisms being a type list, multiple inheritance, and template-based recursion. See, e.g., Chapter 3.13 in the wonderful book by Alexandrescu [1]. However, the approach in [1] can not be applied here without modification because of serial methods such as the insertion operator mentioned in the previous paragraph. In other words, one must clearly differentiate serial methods in `ParameterManagerN` from parallel ones in order to define them correctly. In addition, there are other reasons a simple repetition is preferred here: (1) For the targeted use cases of scientific computing, the number of types is not large; (2) The interface of `ParameterManager` is thin and very static; (3) It is self-contained; (4) It is much easier to understand.

3.3 Which parameters to manage?

By now the reader might have raised a few questions. For examples, a constant parameter can be declared as a static constant data member, so why bother with the parameter manager? Should parameter manager manage *all* data members of a class? If not all, which parameters should be managed? In order to answer these questions, let us examine the relationship between a module

and its parameters.

A module takes some *input* and produces some *output*, it represents a contract between the client who uses the module and the programmer who writes the module. To realize the contract, the two parties have to agree on some conditions: those limit the possible values of the input and the scope of the contract are called *preconditions*; those applies to the output are called *postconditions*. Together they unambiguously define the problem to be solved by the module. In summary, *the behaviors of a module depend on the preconditions and postconditions, which are represented by the parameters of the module.*

A parameter is said to be *fundamental* if it quantifies either a precondition or a postcondition. *The set of fundamental parameters* spans a use-case space, of which each consisting point, represented by a combination of the values of the fundamental parameters, uniquely determines the behaviors of the module. Usually there is more than one way to choose the set of fundamental parameters for a certain module, in this case we require the set of fundamental parameters to have the minimum number of parameters.

The fundamental parameters should be well organized — clearly represented, easily accessed and modified. Indeed, this is the main motivation of the parameter manager pattern. At this point the questions posed in the beginning of this section are easy to answer. Internal data members irrelevant to pre- and post-conditions should not be managed. Whenever a data member, constant or non-constant, belongs to the set of fundamental parameters, it should be managed by `ParameterManager`. As for a data member whose value depends on others, it obviously does not belong to the set of fundamental parameters. Whether it should be managed by `ParameterManager` is a design choice. For example, the grid size `dx` depends on number of the cells, the starting and ending position of the computational domain. However, it should be managed since it will be frequently accessed by other modules. To sum up, the fundamental parameters and frequently used dependent data members should be managed by `ParameterManager`.

To enforce the additional semantics, two pure virtual functions are added to `ParameterManagerN`:

```
template<class T1, class T2, ..., class TN>
class ParameterManagerN : private ParameterManager<T1>,
                          ...,
                          private ParameterManager<TN>
{
public:
  virtual void updateDependentParameters(void) = 0;
  ...
```

```
protected:
  virtual void initialize(void)                = 0;
  ...
};
```

A derived class is responsible for defining the default values and dependency of its parameters. Making the virtual functions pure is a friendly reminder through the compiler if this is forgotten.

3.4 The parameter manager pattern in action

Finally we can apply the parameter manager pattern to the example of solving the diffusion equation.

```
#include "ParameterManagerN.H"

class DiffusionSolver :
  public ParameterManager6<bool,int,double,string,IntVect,RealVect>
{
public:
  /// Independent Parameters : preconditions and postconditions
  static const Parameter<int>            timeOrder;
  static const Parameter<int>            spaceOrder;
  static const Parameter<double>         t0;
  static const Parameter<double>         te;
  static const Parameter<int>            numSteps;
  static const Parameter<int>            numGhosts;
  static const Parameter<double>         convergeRatio;
  static const Parameter<int>            maxNumCycles;
  static const Parameter<RealVect>       xLo; // constant
  static const Parameter<RealVect>       xHi;
  static const Parameter<IntVect>        numCells;
  static const Parameter<bool>           writeHDF5Sol;
  static const Parameter<bool>           writeHDF5Err;
  static const Parameter<string>         writeHDF5Dir;
  /// Dependent Parameters
  static const Parameter<RealVect>       dx;

  DiffusionSolver()
  {
    initialize();
    updateDependentParameters();
  }
```

```
  void updateDependentParameters(void);

private:
  void initialize(void);
};
```

The handlers of the parameters in Table 1 are declared as static constant members. As self-documenting code, the declarations clearly capture the preconditions and postconditions of the diffusion solver module.

Regardless of different types, the value of all declared parameters can be accessed or modified using the single pair of `get/set` functions inherited from `ParameterManager6`, as shown in the following example.

```
   typedef DiffusionSolver DS;
   DiffusionSolver ds;
   ds.set(DS::timeOrder, 4);
   ds.set(DS::convergeRatio, 1.e-10);
   ds.set(DS::writeHDF5Err, false);
   const double t0 = ds.get(DS::t0);
   const int spaceOrder = ds.get(DS::spaceOrder);
   const bool writeHDF5Sol = ds.get(DS::writeHDF5Sol);
   ...
```

Inside the constructor, `initialize()` set the properties of the independent parameters for the first time, then the value of the dependent parameters are updated by `updateDependentParameters()`. The definition file shows more details.

```
#include "DiffusionSolver.H"
#include "ParameterManagerMacros.H"

#define PM_CLASSNAME DiffusionSolver

/// Table of default values
//-----------------------------------------------------------------
PM_DEF (      int,     timeOrder,              4);
PM_DEF (      int,    spaceOrder,              4);
PM_DEF (   double,            t0,            0.0);
PM_DEF (   double,            te,            1.0);
PM_DEF (      int,      numSteps,             10);
PM_DEF (      int,     numGhosts,              2);
PM_DEF (   double, convergeRatio,         1.e-12);
```

```
PM_DEF (      int,  maxNumCycles,             20);
PM_DEFC( RealVect,           xLo, RealVect::Zero);
PM_DEF ( RealVect,           xHi, RealVect::Unit);
PM_DEF (  IntVect,      numCells,  IntVect::Unit);
PM_DEFD( RealVect,            dx, RealVect::Unit, false);
PM_DEF (     bool,  writeHDF5Sol,           true);
PM_DEF (     bool,  writeHDF5Err,           true);
PM_DEF (   string,  writeHDF5Dir,         "hdf5");
///-----------------------------------------------------------

void GridsManager::initialize(void)
{
  PM_INIT(timeOrder);
  PM_INIT(spaceOrder;
  PM_INIT(t0);
  PM_INIT(te);
  PM_INIT(numSteps);
  PM_INIT(numGhosts);
  PM_INIT(convergeRatio);
  PM_INIT(maxNumCycles);
  PM_INIT(xLo);
  PM_INIT(xHi);
  PM_INIT(numCells);
  PM_INIT(writeHDF5Sol);
  PM_INIT(writeHDF5Err);
  PM_INIT(writeHDF5Dir);
}

void GridsManager::updateDependentParameters(void)
{
  const RealVect tmp = (get(xHi)-get(xLo))/get(numCells0);
  set(dx, tmp);
}
```

`RealVect::Zero` and `RealVect::Unit` represent zero and unit vectors with their size as the dimensionality of the computational domain. `dx` is shown as an example of the dependent parameter. Alternatively, one could choose `dx` as an independent parameter and `numCells` as a dependent parameter. So long as the number of independent parameters is minimum, choosing independent parameters is influenced by use cases or even personal preference.

Although each parameter identifier is repeated twice, the macros nicely organize the definition and initialization of the parameters. Consider augmenting Table 1 with two additional columns, one for default values and the other for dependency or constancy. A code-generator can be written to automatically

generate `DiffusionSolver.cpp` (except the function for updating dependent parameters) from the augmented table. We will not elaborate on the code-generator since it can be easily done in a scripting language such as `python` or even in `C++` itself.

4 Concluding Remarks

The author has proposed two patterns for managing parameters in scientific computing. Both patterns use a single pair of get/set functions to access and modify the parameters. The enumeration index pattern is a good choice if all the parameters are of a single type and do not have additional semantics. The parameter manager pattern is a better choice for multiple types and additional parameter semantics.

Using the parameter manager pattern can be automated as much as possible. One only have to inherit from a `ParameterManagerN` class and specify for each parameter its identifier, type, default value, constancy, and dependency. It is then straightforward to convert this specification table to similar code shown in Section 3.4 by a code generator, except that updating dependent parameters cannot be automated.

Scientific computing often uses input files to record the values of the parameters. Using the parameter manager pattern, an input file can be mapped to a parameter manager object by a helper class that parses each key in the input file and then sets the corresponding parameter value for that object.

Acknowledgements

The author would like to thank Jeffrey Johnson for his helpful discussions.

References

[1] A. Alexandrescu. *Modern C++ Design : generic programming and design pattern applied*. Addison-Wesley, 2001. ISBN:0201704315.

[2] B. Eckel. *Thinking in C++: Practical Programming*, volume 2. Prentice Hall, first edition, 2004. ISBN: 0130353132.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : elements of reusable object-oriented software*. Addison-Wesley, 1994. ISBN:0201633612.

[4] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, second edition, 2000. ISBN:0136291554.

[5] Q. Zhang, H. Johansen, and P. Colella. A fourth-order accurate finite-volume method with structured adaptive mesh refinement for solving the advection-diffusion equation. *SIAM J. Sci. Comput.*, 34(2):B179–B201, 2012.

[6] Q. Zhang and P. L.-F. Liu. Handling solid-fluid interfaces for viscous flows: explicit jump approximation vs. ghost cell approaches. *J. Comput. Phys.*, 229(11):4225–4246, 2010. doi:10.1016/j.jcp.2010.02.007.