Molecular Simulation, 2003 Vol. 29 (1), pp. 29–46

Object-oriented Programming Paradigms for Molecular Modeling

AMIT GUPTA, SHAJI CHEMPATH, MARTIN J. SANBORN, LOUIS A. CLARK and RANDALL Q. SNURR*

Department of Chemical Engineering and Center for Catalysis and Surface Science, Northwestern University, Evanston, IL 60208, USA

(Received April 2002; In final form July 2002)

This paper discusses the application of object-oriented programming (OOP) design concepts to the development of molecular simulation code. A number of new languages such as Fortran 90 (F90) have been developed over the last decade that support the OOP design philosophy. We briefly describe the salient features of F90 and some basic object-oriented design principles. As an illustration of the design concepts we implement a general interface in F90 for calculating pairwise interactions that can be extended easily to any number of different forcefield models. The ideas presented here are used in the development of a *mu*ltipurpose *si*mulation *c*ode, named Music. An example of the use of Music for grand canonical Monte Carlo (GCMC) simulations of flexible sorbate molecules in zeolites is given. The example illustrates how OOP allowed existing code for molecular dynamics and GCMC to be easily combined to perform hybrid GCMC simulations with minimal coding effort.

Keywords: Object-oriented programming; Molecular simulation; Fortran 90; Hybrid Monte Carlo

INTRODUCTION

Computer simulation consists of two levels of abstraction. The first level involves taking a physical phenomenon we wish to simulate and, after making suitable assumptions and approximations, describing it as a sequence of mathematical steps. The second level then involves communicating the mathematical model to the computer using a suitable form of programming abstraction called a programming language. Many programming languages have been developed since the early days of assembly language. This development has been spurred by the constant need to enable the programmer to write programs faster with greater program efficiency, understandability, portability and extensibility. Also as new programming paradigms were defined, new languages had to be developed to support these paradigms. Structured programming and object-oriented programming (OOP) are two important paradigms that have shaped the development of most modern languages.

In the 1970s and early 1980s *structured programming* [1] was the dominant programming methodology. Higher-level languages such as Fortran, C and Pascal which supported this form of coding became very popular during that period. The idea behind structured programming was to divide each program into a series of smaller tasks, which in turn could be further simplified until the task could be implemented without further decomposition. This was referred to as the “top-down” design approach, and it worked well since most complex problems are inherently hierarchical. However, each top-down design produced a solution unique to that problem, and hence it was difficult to reuse subroutines in other projects without extensive modifications. Also, this approach did not give adequate consideration to the data being used.

Then in the last one and half decades another approach was developed, where instead of subdividing the problem into tasks the design focus was shifted to the data structure. This came from the realization that a system’s functionality tends to change more than the data on which it acts. Data and ways of structuring it were given primary importance and the code was organized in modules

*Corresponding author. Tel.: +1-847-467-2977. Fax: +1-847-467-1018. E-mail: snurr@northwestern.edu

ISSN 0892-7022 print/ISSN 1029-0435 online © 2003 Taylor & Francis Ltd
DOI: 10.1080/0892702031000065719

around it. By combining the data structure with its intrinsic functionality, these modules were "self-sufficient" entities. These modules could be just "plugged" into new programs, thus improving reusability and reducing coding time. Also, since these modules were designed to emulate real world objects they also made the code easier to understand and maintain. These modules were called objects and the programming approach was referred to as *object-oriented programming*, often abbreviated as *OOP*. That some of the most popular languages such as C++, Java and Fortran 90 (F90) today support this coding methodology is a testament to its advantages.

OOP with F90 has found widespread application in the simulation community over the last few years [2]. The principal advantage of F90 over the other OOP languages is that it is fully backwards compatible with Fortran 77 (F77), allowing programmers to use the vast amount of F77 code that has been developed over the years. Recently, Morieira and Midkiff [3] analyzed the features of F90 with a program for calculating the electrostatic potential in a shielded microstrip structure. They compared the performance of F90 with High Performance Fortran (HPF) and C++. They found that while HPF even on a single node was faster than F90 for some systems, F90 was faster than C++ for every system. Akin [4] illustrates a number of object-oriented features in F90 that are important for engineering computation, and Carr [5] discusses how objects can streamline code development by implementing a module for performing vector operations in F90. These and a host of other papers from the numerical computing community [6–9] are indicative that although this community has its share of die-hard F77 users, F90 has made significant headway in the last few years. However, OOP in F90 has not yet made significant inroads in molecular modeling.

This paper is organized as follows. In the next section we review the basics of OOP. We then describe some of the salient features of F90 and its differences from F77. We then discuss some of the typical challenges encountered while writing molecular modeling programs and discuss how OOP can be used to generate elegant solutions to some of these problems. We then include source code for some objects that are small and may be potentially useful for molecular modeling. Finally, we present the use of our Music code for developing a new type of grand canonical Monte Carlo (GCMC) technique for simulation of adsorption in zeolites.

BASIC TENETS OF OOP

As discussed in the previous section one of the key features of any higher level language is its ability to abstract data. Programming languages that support OOP go further by providing features such as encapsulation, inheritance and polymorphism, which are described below. These features make it easier to reuse and extend code while minimizing the probability of introducing bugs. To understand how these features work it is useful to divide programmers into two groups. One group is concerned with providing fundamental subroutines or libraries which form the building blocks of larger applications. The second group of programmers then are those who use these fundamental libraries in their applications.

Encapsulation refers to hiding information that the creator of the object does not want the user to have access to. The user can only access the properties of any object by function calls provided by the creator. This has a number of advantages. The user sees a much cleaner interface by which the object can be used rather than having to worry about the unnecessary implementation details. The list of functions provided by the creator to access the object is referred to as the *Application Programming Interface* (*API*). The more important benefit is that if the creator of the object module wants to change the data representation of the module it can be done without worrying about breaking the code that uses the object. All that the creator has to ensure is that the functions in the API still work correctly.

OOP tries to give the programmer the power to create objects that correspond closely to objects in the real-world problem domain. Just as objects in the real world are related by an "is-a" type of relationship (e.g. an apple "is-a" fruit), so are objects in OOP. This kind of relationship is implemented in OOP by *inheritance*. Given two objects P and C, if C inherits from P then it automatically gets all the data structures and functionality of P. In addition, the object C can further add to what it inherits, thereby increasing its own functionality. Thus, inheritance is a way of taking a general sort of an object and tailoring to fit specific needs. For instance, every square matrix "is a" matrix, so every function that is possible on a matrix type object is also possible on a square matrix object. In addition, the function eigenvalue is only valid for a square matrix. So, the module `square_matrix` could use inheritance to get all the functionality for the module `matrix` and in addition define its own function for calculating eigenvalues. In OOP terminology module `matrix` would be referred as the *base class* and module `square_matrix` as the *derived class*. Inheritance further emphasizes the data-centric nature of OOP by enabling the user to create hierarchies of data types.

Closely related to inheritance is the concept of polymorphism. Polymorphism enables the programmer to provide a generic interface to a number of related subroutines. Polymorphism comes in two flavors—*static polymorphism* also known as *overloading* and *dynamic* or *run-time polymorphism*. Static polymorphism is used to resolve calls to subroutines with the same names on the basis of parameters passed to the subroutine. For instance, there are a number of ways of specifying parameters to calculate the area of a triangle. Imagine writing two routines for calculating the area of a triangle as follows:

```
  Interface getarea
    Module Procedure getarea2
    Module Procedure getarea3
  End Interface

Real Function getArea3(a, b, c)
  Implicit None
  Real    a, b, c

  Real    s

  s = (a + b + c)/2.0
  getArea3 = Sqrt(s*(s-a)*(s-b)*(s-c))
End Function getArea3

Real Function getArea2(base, height)
  Implicit None
  Real    base, height

  getArea2 = 0.5*base*height
End Function getArea2
```

Then `getArea(3.0, 4.0, 5.0)` calls the first function and `getArea(3.0, 4.0)` calls the second function, thus providing a more uniform and intuitive interface to the function.

Dynamic polymorphism involves resolving calls to functions related by an inheritance hierarchy at run-time as opposed to the compile-time resolution provided by static polymorphism. By virtue of inheritance a "parent" parameter object can be replaced by anyone of its "child" objects in a function or subroutine call. Inheritance guarantees that a derived class object will have at least the same functionality as its base class if not more. This implies that different types of objects related by inheritance can now be passed to the same routine. In OOP since the functional implementations are closely tied to the objects, this in turn implies that depending on the object passed to the routine at run-time different functions may have to be called. This ability of a routine to take different *types* of objects as parameters and resolve the function calls at run-time is referred to as dynamic polymorphism. This sort of polymorphism, while not directly supported in F90, can be simulated with the use of pointers as described by Decyk *et al.* [8,9].

Object-oriented program design requires the programmer to think about the problem at a more fundamental level to identify relevant objects and then define the relationships among them. Language features such as inheritance and polymorphism then facilitate translation of these ideas into computer code. Admittedly OOP ideas take some getting used to. There are a number of excellent papers and books [2,9,10] available to provide a deeper understanding of these concepts and their applications.

FEATURES OF FORTRAN 90

Fortran 90 was designed by the Fortran committee with some of the following goals in mind as described by Reid [11].

- Provide greater expressive power;
- Enhance safety;
- Provide extra fundamental features (such as dynamic storage);
- Exploit modern hardware better;
- Improve portability between different machine architectures.

In this section, we give a very brief overview of some of the key differences between F77 and F90, and we start building up some objects of use in molecular simulations. For more details about F90 please refer to Refs. [9,12].

F90 makes a number of ornamental changes to improve the source code formatting. F90 supports free-formatting, which disables any special significance that was attached to columns in F77. An "&" at the end of a line indicates that the line is continued. The comment character "!" can be placed anywhere on a line with the compiler ignoring anything that follows. We use this improved formatting in the code listed in this paper.

User-defined Type Definitions

One of the biggest improvements over F77 is the ability to abstract data into user-defined types (also known as derived types). For instance, we can associate a number of characteristics to the terms "atoms" and "molecules". Abstractions of these terms can be defined using the *Type* keyword as shown below:

```
Type AtomParams
  Character(len=2)    :: symbol     ! Atomic Symbol
  Character(len=100)  :: name       ! Name
  Real                :: wt         ! Atomic Weight
  Integer             :: atno       ! Atomic Number
End Type AtomParams

Type MoleculeParams
  Character(len=100)  :: name    ! Name of the molecule
  Real                :: wt      ! Molecular Wt.
  Integer             :: natoms  ! No. of atoms
  Type(AtomParams), Dimension(MAX_ATOMS) :: atoms
! MAX_ATOMS is the maximum expected number of atoms
End Type MoleculeParams
```

Now we can use these types just as we would use a primitive data type to define our variables. To declare an array of three molecules and then access its individual fields we use the % symbol as in:

```
Type(MoleculeParams), Dimension(3) :: moList

moList(1)%name = 'Methane' ! Assign name Methane to 1st molecule
moList(1)%wt   = 16.0
moList(1)%natoms = 5
moList(1)%atoms(1)%name = 'Carbon' ! assign name carbon to 1st atom
   ! of 1st molecule
```

Also note that we can use a derived type for the definition of fields of another derived type. In this case we used *AtomParams* in *MoleculeParams*. Thus, we can start to build a hierarchy of data structures with relationships that reflect the order in the physical problem space.

Modules

In F77 the basic building blocks of a program were functions and subroutines. F90 adds another building block called a *module*. A large part of the OOP functionality of F90 comes from the use of modules since they provide a way to couple the data and its associated functionality into one programming unit. We will use the skeleton implementation of a three-dimensional vector object given in Appendix 1 to illustrate a number of features of F90 in general and modules in particular.

A module is defined using the keyword *Module*. Modules can be broken up into two major parts separated by the *Contains* keyword. The section preceding this keyword primarily describes the data type definitions and the *interface* to the routines defined in the module. The routines are then defined after the *Contains* keyword.

Modules provide a mechanism for supporting encapsulation in F90 through the keywords *Public*, *Private* and *Use*, *Only*. Just as we can build a hierarchy of user-defined type definitions, it is possible to build a hierarchy of modules. The *Use* keyword provides access to the variables and routines defined in another module. As discussed before it is a good practice to hide the implementation details from the users of modules. The keywords *Private* and *Public* can be used to change the visibility of the variables and routines defined in a module. It is also good programming to "use" only what is needed from a module. This list is specified with the *Only* keyword. This makes the code more maintainable by immediately conveying to the reader the source of variables and routines used in a module.

In addition to data encapsulation, modules provide explicit interfaces to the routines defined inside them. This means that the compiler has complete information regarding the type of parameters being passed between routines, enabling it to do parameter type-checking. Another advantage is when passing arrays as arguments there is no longer any need to pass their dimensions. This can be seen in the subroutine `vector3D_init1` in Appendix 1 where the dimension of the variable `arr` is not specified. Explicit interfaces thus help to reduce the number of arguments and make the code less prone to insidious errors arising from parameter type incompatibility.

The explicit interface provided inside modules is also useful in supporting overloading. By using the *interface* keyword a generic name can be assigned to routines with different parameters defined in a module. Thus, a call to `vector3D_init` will be resolved to either `vector3D_init1` or `vector3D_init2` depending on whether the second argument is an array or a number. In addition to overloading subroutines and functions, it is also possible to overload operators as shown with the `vector3D_add` method. The *Optional* and *Present* keywords provide another mechanism for designing routines with flexible argument lists. The *Optional* keyword makes the argument optional to the routine. To check whether the optional argument was passed, the keyword *Present* is used as illustrated in the routine `vector3D_display`.

Arrays, Pointers and Dynamic Memory

A much sought after feature in F77 was the ability to dynamically allocate memory. This feature was added to F90. The dynamic memory is managed by using the keywords *Allocate*, *Allocatable* and *Deallocate*. The dynamic allocation is done as shown below:

```
Type(MoleculeParams), Allocatable, Dimension(:) :: moList
Integer     :: err

Read(*,*)  nmoles                    ! Read the no. of molecules
Allocate(moList(nmoles), STAT=err)  ! Get space for 'nmoles' molecules
If (error /= 0) Then
   Write(0,*) ''Could not allocate memory for moList''
   Stop
End If
```

Allocate creates memory on the heap and it stays for the length of the program. It is good programming practice to return the memory to the operating system once we are done with it. This is done by using the *Deallocate* command.

```
Deallocate(moList, STAT=error)
If (error /= 0) Then
 ! We had problems returning the memory
  Write(0,*) 'Could not deallocate ''moList'''
  Stop
End IF
```

Allocatable arrays make for more flexible code while minimizing memory waste.

Often we want to have allocatable fields in an abstract data type. For instance, in `moleculeParams` we may want to make the atom array an allocatable type to reduce memory waste.

```
Type MoleculeParams
  Character(len=100)  :: name    ! Name of the molecule
  Real                :: wt      ! Molecular Wt.
  Integer             :: natoms  ! No. of atoms
  Type(AtomParams), Dimension(:), Allocatable :: atoms ! * ILLEGAL *
End Type MoleculeParams
```

Unfortunately, this is illegal in F90 since *Allocatable* is not an allowed attribute for fields in the user defined types (this will be fixed in Fortran 2000). Instead we have to use the *pointer* attribute. Pointer variables hold the location of memory as their value. Depending on the type of the pointer, this value could refer to the location of an integer, an array or a derived data type. Another way to think about pointers is as aliases to a particular variable, e.g.

```
Integer, Target  :: a
Integer, Pointer :: ptr
a = 5
ptr => a   ! ptr is now an alias for a
ptr =  7   ! a is now  7
```

The variable `ptr` is said to point to `a` or be *Associated* with the variable `a`. To dissociate the pointer we use `Nullify(ptr)`. To find out if a pointer is associated we use the keyword *Associated* as in

```
Integer, Target  :: a
Integer, Pointer :: ptr
a = 5
ptr => a
If (Associated(ptr)) Then ! Pointer is Associated
   Write(*,*) 'ptr is associated'
End If
Nullify(ptr)
If (.Not. Associated(ptr)) Then  ! Pointer is
 Not Associated anymore
   Write(*,*) 'ptr is NOT associated'
End If
```

Pointers can be used to solve the problem of having dynamically allocatable fields in derived data types as shown below. This piece of code will initialize the number of molecules and number of atoms as specified by the user.

```
Type MoleculeParams
  Character(len=100)  :: name    ! Name of the molecule
  Real                :: Wt      ! Molecular Wt.
  Integer             :: natoms  ! No. of atoms
  Type(AtomParams), Dimension(:), Pointer :: atoms
End Type MoleculeParams



Type(MoleculeParams), Allocatable, Dimension(:) :: moList

Read(*,*) nmoles                        ! No. of molecules
Allocate(moList(nmoles), STAT=error)
Do i=1, nmoles
  Read(*,*) natoms      ! No. of atoms in molecule 'i'
  Allocate(moList(i)%atoms(natoms), STAT=error)
End Do
```

Pointers also allow us to implement polymorphism in F90 [9] as we shall see later.

In the style of Matlab ® [13], F90 provides functionality so that arrays may be used in whole expressions with scalars as in,

```
Real, Dimension(10) :: arA, arB, arC
Real                :: scD
scD = 1.0
arA = arB + scD*Cos(arC)
```

In addition, subarrays of larger arrays called "sections" [11] may be used as arrays. For instance,

```
Real, Dimension(10, 10) :: arA, arB, arC
arA(:,10)  = arB(1,:)
  ! set 10th column of A to 1st row of B
arC(1:3,1) = arA(2:6:2,1)
  ! C(1,1)=A(2,1), C(2,1)=A(4,1), C(3,1)=A(6,1)
```

Not only is this more succinct but it is also more computationally efficient than writing `Do` loops since the optimization is better [11].

Portability

With the proliferation of a large number of architectures and operating systems it is difficult to ensure portability especially with regard to numeric operations. Towards this end, F90 provides a way to specify the precision and range of numbers that remains constant across platforms. This is done using the *Selected_real_kind*, *Selected_int_kind* and *Kind* keywords. The usage is illustrated below:

```
! Define a Real Kind with Precision of 9 digits and
! Range of 10^50
  Integer, Parameter :: RDbl = Selected_real_kind(9, 50)
! Define an Integer Kind with 10 significant digits
  Integer, Parameter :: IDbl = Selected_int_kind(10)

! Now use the kind values to declare variables
  Real(kind=RDbl)    :: a ! This variable has 9 digits of
                          ! precision and range of 10^50
  Integer(kind=IDbl) :: b ! This integer has 10 significant figures
```

The result of intrinsic functions *Selected_real_kind* and *Selected_int_kind* is a kind type that meets, or minimally exceeds, the requirements specified by the precision and range [12]. This mechanism ensures program portability across platforms since different architectures have different precisions for primitive data types.

MOLECULAR MODELING WITH OOP

Two important aspects of molecular modeling that are constantly evolving are move types and forcefields. Examples of move types include molecular dynamics integration, Monte Carlo translation of a molecule, etc. New types of moves are developed to accelerate the speed of the simulation, whether it is to achieve equilibrium more quickly or to lower the ratio of CPU time spent per unit of physical time while tracking the system dynamics. While the ability to sample a phase space more efficiently increases the precision of the simulation result, the accuracy of the simulation is determined by the forcefield and the parameters employed. Consequently, new and improved forcefields are also constantly being developed.

As computer simulations become more reliable there is an increased demand on the researcher to cater to a wider array of problems. This often requires the ability to deal with many different kinds of forcefields, and as the problems get more complex, greater functionality is also required to incorporate different kinds of move types into one simulation. The integration of various forcefields and move types brings with it new challenges for data representation. The different move types often require different forms of representation for the current configuration of the system. For instance, it is usually most convenient to work in Cartesian coordinates for molecular dynamics simulations whereas generalized coordinates are preferred for doing Monte Carlo simulations of molecules with constraints. This is also true for various terms comprising a forcefield. While terms for pairwise interactions require Cartesian distances, terms involving intramolecular interactions work with various forms of generalized coordinates.

The problem then is to find a "clean," easily extensible approach to represent the data in the simulation, making the programs more flexible and ensuring faster code development. The OOP paradigm lends itself naturally to such an approach by enabling the programmer to cast the data objects as more direct representations of real world objects. Recently, a number of groups [14–16] have described the design of some basic objects that can be used as building blocks for various types of molecular simulations. They illustrate how relatively straightforward it is to extend the existing building blocks to design objects for new simulations. The idea then is to distribute these objects to other groups who can customize them according to their needs. Huber *et al.* [14] and Sadus [15] implement the objects using C++, whereas Kofke *et al.* [16] use Java as their language of choice. While F90 does not support OOP in the full traditional sense of the word, it does lend itself to a data-centric programming style. We illustrate the design and implementation of OOP objects in F90 by implementing a number of forcefields describing pairwise interactions.

Since F90 does not have an OOP syntax, we define a few programming style rules that enable us to enforce the OOP paradigm more effectively. These rules are given below:

1. All routines in a module start with the name of the module to tie the function more strongly with the module name.
2. Each module has defined within it a user defined data type that is representative of the data in the module.
3. All the public routines in the module take as their first parameter a variable of this representative data type. This enables a close coupling with the data object and the functionality that is provided by the routine.
4. Each module has an *init* routine to initialize the data fields of the variable of the representative data type.
5. Each module has a display routine to report the values of data fields.
6. Each module has a cleanup routine to take care of deallocating memory if necessary.

A molecular simulation program can be viewed as consisting of two distinct parts, the initialization and the actual run. The initialization is responsible for setting up the initial configuration, the forcefield parameters and the parameters for the different move types. One of the difficulties of designing a general purpose simulation code is that parameters and initialization for the different forcefields and move types vary widely. OOP allows one to provide a *generic interface* that is used by any driver program to initialize different move types and forcefields.

We have used the ideas discussed above in the development of the *Mu*ltipurpose *si*mulation *c*ode (Music). All the forcefield calculations (estimation of energies and forces resulting from interparticle interactions) are wrapped into a `forcefield` module. Similarly, all the calculations associated with move types (molecular dynamics integration, Monte Carlo insertions etc.) are wrapped under the `moves` module.

The interface for `forcefield` is defined so that it receives positions of all atoms in the system and returns the energies and forces on particles of interest. The calculation of forces is optional because it is not required during Monte Carlo simulations. Also it is not necessary to perform calculations on all molecules every time. During a Monte Carlo insertion move it is only necessary to calculate the interaction energy of the newly inserted molecule with rest of the system, whereas during an MD simulation we need the forces on all particles during each step of the integration. The `forcefield` object contains all information regarding intermolecular and intramolecular interactions. The intermolecular interactions are usually calculated using pairwise potentials like Lennard Jones potentials. The `pair-model` module which does this is described later in the section.

The interface for `moves` is defined so that it takes a particular configuration of the system and changes it to a different one. It returns the energies of old and new configurations. For example a Monte Carlo translation move will pick one of the molecules and translate it into a different position and then return the old and new energies. This new trial configuration can be accepted or rejected by the main calling program based on the Monte Carlo acceptance criterion [17].

The actual run usually consists of a number of iterations where the phase space is sampled by perturbing the configuration according to move types derived from statistical mechanics. This process can be abstracted as illustrated in Fig. 1. Now if we can define a *generic interface* that can be used by all move types to "communicate" with any type of forcefield then we can quickly create new simulations by "plugging" together move types and forcefields as we desire.

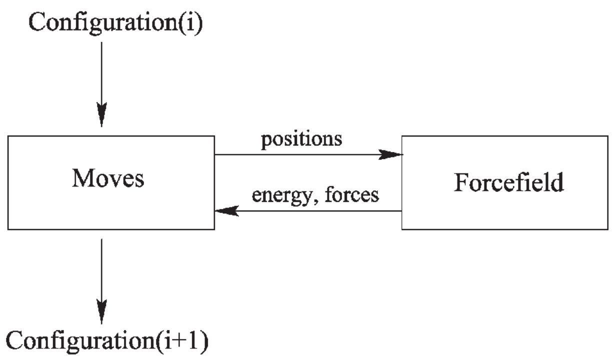

FIGURE 1 The anatomy of a simulation run.

F90 enables us to create such generic interfaces. To illustrate the methodology we implement an interface that is used to initialize various models for calculating pairwise atomic interactions. This object will be used by the `forcefield` module. The object is implemented in the module called `PairModels` and consists of two main routines, `PairModels_init` and `PairModels_getinteraction` (Appendix 2), as well as display and cleanup routines.

The responsibility for initializing parameters and getting interactions is then passed on to the relevant potential model object as specified in `PairModel_init`. This makes sense since each type of potential model object "knows" what parameters it needs and how to use them. The Lennard-Jones potential model object called lj is shown in Appendix 3. This scheme enables us to specify the kind of potential model we want to use as an input parameter string without changing any code. Also, to add another pairwise potential model, all we need to do is modify the module `PairModels`, making it easy to extend the code. These changes will not affect other objects dealing with move types or intramolecular potentials.

The module `PairModels` presented here can be downloaded from our website [18]. Along with the source code in this paper the site has also a number of other potentially useful modules for download.

HYBRID GCMC USING MUSIC

In this section an example of the use of Music for studying adsorption in zeolites is given. This example highlights how object oriented programming facilitated the incorporation of new capabilities into the code with minimal programming effort. Zeolites are crystalline microporous materials with cavities and pores about 3–12 Å in diameter. They are used for separation and catalysis applications. In these processes, adsorption and diffusion of guest molecules within the zeolite pores play an important role.

GCMC simulations can be used to study the adsorption of the guest molecules in the zeolite cavities [19]. The regular GCMC procedures are applicable only when the guest molecule being adsorbed is considered rigid. We have used a method called hybrid GCMC in which regular GCMC is combined with the hybrid Monte Carlo method to take care of the flexibility of the guest molecule.

Hybrid GCMC

The hybrid Monte Carlo (HMC) method was introduced by Duane *et al.* [20] in 1987. A detailed description is given in Ref. [21]. This method is usually used for simulating an NVT ensemble, but in the current work we have used HMC combined with GCMC to simulate a grand canonical ensemble. HMC uses overall system moves instead of the usual translation and rotation moves of individual molecules for exploring the configurations of the system. In these hybrid moves, the particles in the simulation volume are given velocities taken from a Maxwell–Boltzmann distribution and then integrated for a specific amount of time (τ) using a specific integration step ($\Delta\tau$). The resulting configuration is accepted or rejected based on a Metropolis acceptance criterion [21]. The hybrid GCMC simulation thus consists of HMC moves, insertion moves and deletion moves.

Prior to the hybrid GCMC simulation a library of configurations of a single guest molecule in an ideal gas phase is generated by doing HMC at the same temperature as the main simulation. For insertion of a molecule into the system during hybrid GCMC, one of the configurations is selected from the library and then inserted into the zeolite. The Eulerian angles of orientation for the molecule are selected randomly during each insertion. The insertions into the simulation volume can be done randomly or with bias towards the low energy regions of the zeolite [22]. It can be shown that this scheme leads to the same acceptance criterion for insertion as in regular GCMC.

$$P_{\mathrm{Acc}}(N \rightarrow N+1) = \min\left[1, \frac{fV}{(N+1)kT}\exp(-\Delta\mathscr{V}/kT)\right]$$

where $\Delta\mathscr{V}$ denotes the potential energy of interaction between the whole system and the sorbate molecule being inserted and f is the fugacity of the sorbate at the given pressure.

Combining MD and GCMC

Using the Music framework, it was extremely easy to develop the hybrid GCMC code. The integration code (MD code) and the GCMC code had already been developed. The different move types used for

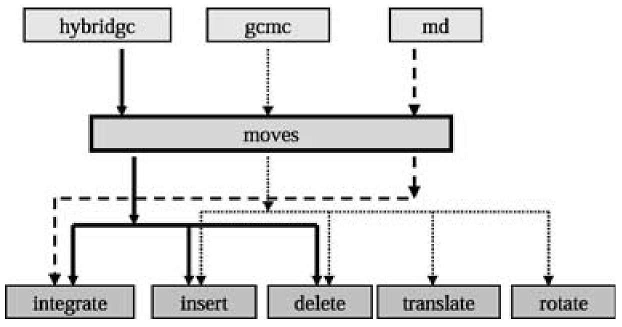

FIGURE 2 The hierarchy of move type modules. Dotted lines: call tree for a GCMC simulation. Dashed lines: call tree for an MD simulation. Solid lines: call tree for hybrid GCMC simulation.

these simulations were already developed as move type objects and placed in the modules. See Fig. 2 for an arrangement of the modules in Music. The individual move types are wrapped under the `moves` module which provides a common interface for accessing all move types. This common interface makes the process of calling individual move types easier. The main calling programs (top level of Fig. 2) need not know much about the interface for the individual move types. During an MD simulation the main calling program calls the `moves` module repeatedly and `moves` subsequently calls the `integrate` module. During a GCMC simulation `moves` is called repeatedly and `moves` calls the `insert`, `delete`, `translate` or `rotate` modules. All of these move types access `forcefield` which contains details about the interaction between molecules to calculate the energies and accelerations. To create a hybrid GCMC simulation a main driver module called `hybridgc` was written, which calls `integrate`, `insert` and `delete` move types. Since all of these modules were well tested previously and their API was well defined, the creation of `hybridgc` and testing became very easy. The flexibility added by defining the move types and placing them under the common interface called `moves` can be explained with the help of an example. Suppose one of the developers in the Music developers group wants to do a hybrid GCMC simulation. If the developer also wants to include a Monte Carlo translation move in the simulation in addition to the `insert`, `delete` and `integrate` moves, all that needs to be done is to add a few lines of code in the `hybridgc` module which makes a call to `translate` through the `moves` module. The module `translate` was probably developed by some other developer interested in regular GCMC. However since the module is placed under the common interface `moves`, the hybrid GCMC developer can also access it and use it. This makes code sharing among developers working on different move types easier.

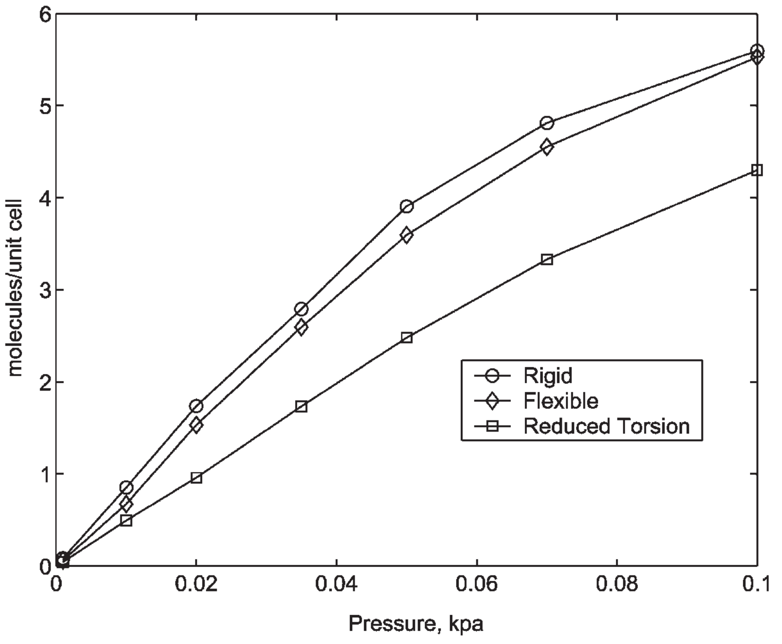

FIGURE 3 Change of loading with flexibility for butane in silicalite at 300 K.

The potential for a decrease in computational speed may make some researchers cautious about adopting OOP. However, with improving compilers, differences in speed between F77 and F90 are becoming quite small. To quantify this for our case, we performed MD simulations of butane in the zeolite silicalite, using 64 butane molecules (as described below) in 16 unit cells of zeolite with periodic boundary conditions. Comparing our older, highly-optimized F77 code against the new Music code, we found that the OOP code was 2.6 times slower. We believe that there is room for optimization of the Music code, which would reduce this factor, and again improved compilers will also help. The reduced time in writing and maintaining the code, however, fully compensates for any increased run-time in our experience.

Effect of Sorbate Flexibility on Adsorption

Using hybrid GCMC we calculated the adsorption isotherms of butane in the zeolite silicalite at 300 K. The butane molecule was modeled using a united atom model with bond stretching, bond bending and bond torsion potentials. The forcefield parameters were the same as the ones used by Macedonia and Maginn [23]. In addition to those parameters (which kept bond lengths fixed), we also included the bond stretching potential energy using the Morse equation [24]. We also performed regular GCMC simulations of completely rigid butane in silicalite for comparison. One might expect to see larger number of molecules adsorbed in the silicalite with the flexible butane model compared to the rigid model, since a flexible molecule will be able to adjust its shape and fit inside the zeolite cavities more easily. However, it was found that the loadings obtained from hybrid GCMC are slightly lower as shown in Fig. 3.

To find out the cause of this effect, we conducted additional simulations with varying flexibility of the butane molecule. It should be noted that in all simulations the butane–zeolite interactions were modeled using the same potential model. Figure 3 also shows the loading for a simulation where the magnitude of the torsional potentials was reduced by a factor of 1/10 (Reduced Torsion), making the molecule even more flexible. The loading here is even lower than the previous cases. We also tried changing the magnitudes of the bond stretching and bond bending potentials. They did not have any significant effect on the adsorption characteristics. So, we conclude that as the torsion potential is made more flexible the loading of butane in silicalite decreases (at fixed temperature and pressure). The distribution of the torsion angle is plotted in Fig. 4. In the gas phase there are two peaks in the distribution: at 0° (*anti*-conformer) and 120° (*gauche*-conformer). In case of the hybrid GCMC simulation in silicalite, the zeolite cavities further restrict the allowed conformations of butane molecules. When a molecule is transported from the gas phase to the adsorbed phase it is forced to remain more at the *anti*-conformation than was required in the gas phase. As a result the peak corresponding to *anti*-conformations increases and the one corresponding to *gauche* conformations decreases. This constraining effect of the zeolite pore walls which makes the *anti*-conformation more preferable was also observed by

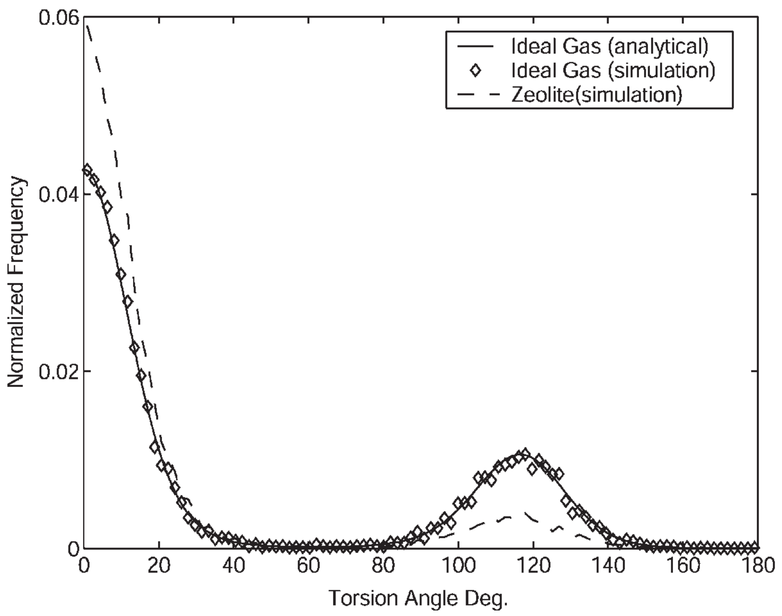

FIGURE 4 Distribution of dihedral angle of butane for adsorption in silicalite at 300 K. Also shown are the simulated and analytical distributions for butane in an ideal gas at 300 K.

June *et al.* [25]. This entropic loss leads to the decreased loading observed when the molecule is made flexible. The entropic loss on the torsional degree of freedom is absent in the case of regular GCMC simulation using the rigid butane model. A similar effect was observed by Gupta *et al.* [26] while comparing GCMC simulations of rigid and flexible cyclohexane in silicalite.

CONCLUSIONS

OOP is difficult. Indeed, one of the challenges of OOP is to create a one-to-one mapping between the elements in the problem and objects in the program. However, the solution is more readable, extensible, and reusable. We have illustrated that though F90 does not have all the features traditionally associated with an OOP language it does provide enough functionality to facilitate object-oriented design.

The ideas discussed in this paper have been used to develop a multipurpose simulation code (Music). This code provides functionality for performing molecular dynamics simulations, Monte Carlo in a number of different ensembles, minimizations, free energy calculations and other classical simulations in bulk phase and adsorbed phases using a variety of forcefields. The fact that a hybrid GCMC simulation code could be very easily developed by mixing already existing modules that were used for GCMC and MD simulations demonstrates the usefulness of an object-oriented framework. Music has been used extensively in our group for a number of projects [27,28] and is continually being developed and improved.

Acknowledgements

This work has been supported by the National Science Foundation CAREER program.

References

[1] Dijkstra, E.W. (1976) *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ).
[2] Discussions of object-oriented programming issues concerning the numerical computing community (www http://www.oonumerics.org).
[3] Moreira, J.E. and Midkiff, S.P. (1998) "Fortran 90 in cse: a case study", *IEEE Comput. Sci. Eng.* **5**, 39.
[4] Akin, J.E. (1999) "Object oriented programming via Fortran 90", *Eng. Comput.* **16**, 26.
[5] Carr, M. (1999) "Using FORTRAN 90 and object-oriented programming to accelerate code development", *IEEE Antennas Propagation Magazine* **41**, 85.
[6] Cary, J.R., Shasharina, S.G., Cummings, J.C., Reynders, J.V.W. and Hinker, P.J. (1997) "Comparison of C++ and Fortran 90 for object-oriented scientific programming", *Comput. Phys. Commun.* **105**, 20.
[7] Gray, M.G. and Roberts, R.M. (1997) "Object-based programming in Fortran 90", *Comput. Phys.* **11**, 355.
[8] Decyk, V.K., Norton, C.D. and Szymanski, B.K. (1998) "How to support inheritance and run-time polymorphism in Fortran 90", *Comput. Phys. Commun.* **115**, 9.
[9] Excellent web site explaining how to support inheritance, polymorphism in F90 (http://www.cs.rpi.edu/~szymansk/oof90.html).
[10] Eckel, B. (2000) *Thinking in C++*, 2nd Ed. (Prentice Hall, Englewood Cliffs, NJ).
[11] Reid, J. (1992) "The advantages of Fortran 90", *Computing* **48**, 219.
[12] Ellis, T.M.R., Philips, I.R. and Lahey, T.M. (1994) *Fortran 90 Programming*, 1st Ed. (Addison-Wesley Publishing Company, Reading, MA).
[13] The Mathworks, Inc. (www http://www.mathworks.com).
[14] Huber, G.A. and McCammon, J.A. (1999) "OOMPAA—Object-oriented model for probing assemblages of atoms", *J. Comp. Phys.* **151**, 264.
[15] Sadus, R.J. (1999) *Molecular Simulation of Fluids: Theory, Algorithms And Object-orientation* (Elsevier, Amsterdam).

[16] Kofke, D.A. and Mihalick, B.C. (2002) "Web-based technologies for teaching and using molecular simulation", *Fluid Phase Equilibria* **194–197**, 327.
[17] Allen, M.P. and Tildesley, D.J. (1987) *Computer Simulation of Liquids* (Oxford University Press, Oxford).
[18] Web site for obtaining source code given in this paper and other potentially useful modules (www http://zeolites.cqe.northwestern.edu/Music/music.html).
[19] Fuchs, A. and Cheetham, A. (2001) "Adsorption of guest molecules in zeolitic materials: computational aspects", *J. Phys. Chem. B* **105**, 7375.
[20] Duane, S., Kennedy, A.D., Pendleton, B.J. and Roweth, D. (1987) "Hybrid Monte Carlo", *Phys. Lett. B* **195**, 216.
[21] Mehlig, B., Heermann, D. and Forrest, B. (1992) "Hybrid Monte Carlo method for condensed-matter systems", *Phys. Rev. B* **45**, 679.
[22] Snurr, R.Q., Bell, A.T. and Theodorou, D.N. (1993) "Prediction of adsorption of aromatic hydrocarbons in silicalite from grand canonical Monte Carlo simulations with biased insertions", *J. Phys. Chem.* **97**, 13742.
[23] Macedonia, M.D. and Maginn, E.J. (1999) "A biased grand canonical Monte Carlo method for simulating adsorption using all-atom and branched united atom models", *Mol. Phys.* **96**, 1375.
[24] Demontis, P., Suffritti, G.B. and Tilocca, A. (1996) "Diffusion and vibrational relaxation of a diatomic molecule in the pore network of a pure silica zeolite: a molecular dynamics study", *J. Chem. Phys.* **105**, 5586.
[25] June, R.L., Bell, A.T. and Theodorou, D.N. (1990) "Prediction of low occupancy sorption of alkanes in silicalite", *J. Phys. Chem.* **94**, 1508.
[26] Gupta, A., Clark, L.A. and Snurr, R.Q. (2000) "Grand canonical Monte Carlo simulations of non-rigid molecules: siting and segregation in silicalite zeolite", *Langmuir* **16**, 3910.
[27] Gupta, A. and Snurr, R.Q. "A study of pore blockage in silicalite zeolite using minimum energy path simulations", In Preparation.
[28] Gupta, A. and Snurr, R.Q. "A study of pore blockage in silicalite zeolite using free energy perturbation calculations", In Preparation.

APPENDIX 1

```
!--------------------------------------------------------------------
! Contains the object and routines required to work with 3D vectors
!--------------------------------------------------------------------
Module vector3D
  !------------------------------------------------------------
  ! Specify which modules to use here. There none in this case
  !------------------------------------------------------------
  !  Use module_xyz, Only: functions_in_xyz, variables_in_xyz

  Implicit None
  Save

  !------------------------------------------------------------------
  ! Make everything private by default, then specify which variables
  ! and functions are allowed for Public Use by other modules
  !------------------------------------------------------------------
  Private
  Public :: VecType, Operator(+), Operator(*), Assignment(=), &
      vector3D_init, vector3D_display, vector3D_getnormsq, &
      vector3D_mult_scalar

  !---------------------------------------------------------------
  ! Data Type Definition of the object associated with this module
  !---------------------------------------------------------------
  Type VecType
    Real*8    :: x, y, z
  End Type VecType

  !----------------------------------
  ! Variable Declarations
  !----------------------------------
  ! None required in this module

  !----------------------------------------------------------
  ! Interface Specification for the Subroutines and Functions
  !----------------------------------------------------------
  ! Static Polymorphism
  Interface vector3D_init
    Module Procedure vector3D_init1
    Module Procedure vector3D_init2
  End Interface

  ! Operator Overloading
  Interface Operator(+)
    Module Procedure vector3D_add
  End Interface
  Interface Operator(*)
    Module Procedure vector3D_mult_scalar
  End Interface

  ! Operator Overloading
  Interface Assignment(=)
    Module Procedure vector3D_vec_eq_scalar
  End Interface


  !-------------------------------------------------------
  ! Definition of Associated Subroutines and Functions
  !-------------------------------------------------------
Contains
  !-----------------------------------------------------------------
  ! Initialize the vec "object" to the values stored in the array arr
  !-----------------------------------------------------------------
  Subroutine vector3D_init1(vec, arr)
    Type(VecType), Intent(out) :: vec
    Real*8, Dimension(:), Intent(in) :: arr

    ! Assign the value from the array
    vec%x = arr(1)
    vec%y = arr(2)
    vec%z = arr(3)
  End Subroutine vector3D_init1

  !--------------------------------------------------------------
  ! Initialize all the components of the vec "object" to the value
  ! in "num"
  !--------------------------------------------------------------
  Subroutine vector3D_init2(vec, num)
    Type(VecType), Intent(out) :: vec
    Real*8, Intent(in)         :: num

    ! Assign "num" to elements of "vec"
    vec%x = num
    vec%y = num
    vec%z = num
  End Subroutine vector3D_init2

  !-------------------------------------------------------
  !Performs " vec1 + vec2 ".  This is operator overloaded
  !-------------------------------------------------------
  Type(VecType) Function vector3D_add(vec1, vec2)
    Type(VecType), Intent(in) :: vec1, vec2

    ! Do the addition
    vector3D_add%x = vec1%x + vec2%x
    vector3D_add%y = vec1%y + vec2%y
    vector3D_add%z = vec1%z + vec2%z
  End Function vector3D_add


  !-----------------------------------------------------
  ! Multiply vector "vec" by scalar "num"
  !-----------------------------------------------------
  Type(VecType) Function vector3D_mult_scalar(vec, num)
    Type(VecType), Intent(in) :: vec
    Real, Intent(in) :: num
    vector3D_mult_scalar%x=num*vec%x
    vector3D_mult_scalar%y=num*vec%y
    vector3D_mult_scalar%z=num*vec%z
  End Function vector3D_mult_scalar

  !-------------------------------------------------------
  ! returns the square of the magnitude of the vector "vec"
  !-------------------------------------------------------
  Real Function vector3D_getnormsq(vec )
    Type(VecType), Intent(in) :: vec
    vector3D_getnormsq=(vec%x**2)+(vec%y**2)+vec%z**2
  End Function vector3D_getnormsq


  !------------------------------------------------
  ! Initialize all of a vector's components to "num"
  !------------------------------------------------
  Subroutine vector3D_vec_eq_scalar(vec, num)
    Type(VecType), Intent(out) :: vec
    Real*8, Intent(in) :: num
    vec%x=num
    vec%y=num
    vec%z=num
  End Subroutine vector3D_vec_eq_scalar
  !-------------------------------------------------------------------
  ! Write contents of "vec" structure to the logical unit no "unitno"
  !-------------------------------------------------------------------
  Subroutine vector3D_display(vec, optunitno)
    Type(VecType), Intent(in) :: vec
    Integer, Optional, Intent(in) :: optunitno
    Integer :: unitno
    If (Present(optunitno)) Then
      unitno = optunitno
    Else
      unitno = 6 ! Write to standard output
    End If
    Write(unitno,*) "Components of vector : ", vec%x, vec%y, vec%z
  End Subroutine vector3D_display
End Module vector3D
```

APPENDIX 2

```
!------------------------------------------------------------------
! This is a "proxy" module to access different pairwise forcefields
! using manually implemented polymorphism.  It is used to provide
! a generic interface to initialize different forcefield models and
! then get interactions from them.
! Lennard Jones potentials (lj) and Buckingham potentials (buck) are
! accessible through this module. More potential models can be
! added similarly.
!------------------------------------------------------------------
Module pairmodels
  Use vector3D, Only: VecType, vector3D_getnormsq
  Use lj, Only: LJ_Params, lj_init, lj_getinteraction, lj_display, &
lj_cleanup
  Use buck, Only: BUCK_Params, buck_init, buck_getinteraction, &
        buck_display, buck_cleanup
  Use defaults, Only: RDbl
  Implicit None
  Save

  Private
  Public:: Pairmodels_Type, pairmodels_init, pairmodels_cleanup, &
pairmodels_getinteraction, pairmodels_display

  Type Pairmodels_Type
    Type(LJ_Params), Pointer          :: lj
    Type(BUCK_Params), Pointer        :: buck
  End Type Pairmodels_Type

Contains
  !------------------------------------------------------------------
  ! Initialize the parameters for atom types "atomtype1" and
  ! "atomtype2" to the "modeltype" of forcefield from file
  ! "paramFilename". The parameters for all atoms are stored in
  ! "pmodels".
  !------------------------------------------------------------------
  Subroutine pairmodels_init(pmodel, modeltype, paramFilename)
    Type(Pairmodels_Type),Intent(inout) :: pmodel
    Character(*), Intent(in)            :: modeltype
    Character(*), Intent(in)            :: paramFilename

    !** Initialize the appropriate forcefield
    If (modeltype == 'LJ') Then
      Call lj_init(pmodel%lj, paramFilename)
    Else If (modeltype == 'BUCK') Then
      Call buck_init(pmodel%buck, paramFilename)
    End If
  End Subroutine pairmodels_init

  !------------------------------------------------------------------
```

```
! Get the potential energy "pot" and force "f" between atom types
! "atom1" and "atom2" separated by the vector "sepvec".  This is in
! effect implementing dynamic polymorphism
!--------------------------------------------------------------------
Subroutine pairmodels_getinteraction(pmodel, sepvec, pot, f)
  Type(Pairmodels_Type),Intent(in) :: pmodel
  Type(VecType), Intent(in)        :: sepvec
  Type(VecType), Intent(out)        :: f
  Real(kind=RDbl), Intent(out)     :: pot
  Integer :: ierr

  If (Associated(pmodel%lj)) Then
    Call lj_getinteraction(pmodel%lj, sepvec, pot, f, ierr)
  Else If (Associated(pmodel%buck)) Then
    Call buck_getinteraction(pmodel%buck, sepvec, pot, f, ierr)
  End If
End Subroutine pairmodels_getinteraction

!------------------------------------------------------------
! Displays the parameters
!------------------------------------------------------------
Subroutine pairmodels_display(pmodel, optunitno)
  Type(Pairmodels_Type),Intent(in) :: pmodel
  Integer, Optional, Intent(in)  :: optunitno

  Integer    :: unitno

  If (Present(optunitno)) Then
    unitno = optunitno
  Else
    unitno = 6    !Standard output
  End If

  If (Associated(pmodel%lj)) Then
    Call lj_display(pmodel%lj, unitno)
  Else If (Associated(pmodel%buck)) Then
    Call buck_display(pmodel%buck, unitno)
  End If
End Subroutine pairmodels_display

!------------------------------------------------------------
! Call the cleanup routines for the different forcefields
!------------------------------------------------------------
Subroutine pairmodels_cleanup(pmodel)
  Type(Pairmodels_Type),Intent(inout) :: pmodel
      If (Associated(pmodel%lj)) Then
        Call lj_cleanup(pmodel%lj)
      Else If (Associated(pmodel%buck)) Then
        Call buck_cleanup(pmodel%buck)
      End If
End Subroutine pairmodels_cleanup
End Module pairmodels
```

APPENDIX 3

```
!-----------------------------------------------------------------
! This module contains the data structure for Lennard-Jones type
! interactions
!-----------------------------------------------------------------
Module lj
  Use defaults, Only: RDbl, strLen, kcalmole_kb
  Use vector3D, Only: VecType, Assignment(=), Operator(+), &
Operator(*), vector3D_getnormsq, vector3D_mult_scalar

  Implicit None
  Save

  Private
  Public :: LJ_Params, lj_init, lj_getinteraction, lj_display, &
lj_cleanup

  Type LJ_Params
    Character(len=strLen) :: atom_name1, atom_name2
    Real(kind=RDbl)     :: sig, eps
    Real(kind=RDbl)     :: A, B      ! A=4*eps*sig^12, B=4*eps*sig^6
    Real(kind=RDbl)     :: cutrad,cutrad2
    Real(kind=RDbl)     :: lowcut    ! lower bound to prevent overflow
  End Type LJ_Params

Contains
  !---------------------------------------------------------------
  ! Initialize the interaction information from the file line.  This
  ! module "knows" how the information is arranged in the
  ! paramsFilename and is able to read.  This way the file format
  ! can be tailored for each specific forcefield initialization.
  !---------------------------------------------------------------
  Subroutine lj_init(params, paramsFilename)
    Type(LJ_Params), Intent(INOUT)            :: params
    Character(*), Intent(IN)                  :: paramsFilename

    Integer                                   :: ios, paramunit
    Character(len=strLen)                     :: paramFilename
    Character(len=strLen), Dimension(10)      :: fields,chunks

    !** Open the file
    paramunit = 13
    Open(unit=paramunit, file=paramFilename, status='old', IOSTAT=ios)
    If (ios /= 0) Then
      Write(0,'(2a,i4,2a)') __FILE__,": ",__LINE__, &
          '  Could not open file ',Trim(paramFilename)
      Stop
  End If
  !** Read the various parameters
  Read(paramunit, *) params%atom_name1, params%atom_name2
  Read(paramunit, *) params%sig
  Read(paramunit, *) params%eps
  Read(paramunit, *) params%lowcut
  Read(paramunit, *) params%cutrad
  params%cutrad2 = params%cutrad**2

  Close(paramunit)

  !** Generate the AB parameters
  Call lj_calc_AB(params)
End Subroutine lj_init

!-------------------------------------------------------------------
! Calculate the interaction between two atoms
!-------------------------------------------------------------------
Subroutine lj_getinteraction(params, sepvec, pot, fvec, ierr)
  Type(LJ_Params), Intent(IN)      :: params
  Type(VecType), Intent(IN)        :: sepvec
  Real(kind = RDbl), Intent(OUT)   :: pot
  Type(VecType), Intent(OUT)       :: fvec
  Integer, Intent(out)             :: ierr

  Real(kind = RDbl)   :: A,B,r2i,r6i,r12i,r2

  pot = 0.0_RDbl
  fvec = 0.0_RDbl       ! Using the overloaded '=' operator
  ierr = 0

  !** Get the square of the distance
  r2 = vector3D_getnormsq(sepvec)

  !**Check if it is within the cut-off radius
  If (r2 > params%cutrad2) Return

  !** Make sure the atoms are not too close
  If (r2 < params%lowcut) Then
    ierr = 1
    Return
  End If

  A = params%A
  B = params%B

  r2i = 1.0_RDbl/r2
  r6i = r2i*r2i*r2i
  r12i= r6i*r6i

  fvec = sepvec  * Real( (12.0_RDbl*A*r12i - 6.0_RDbl*B*r6i)*r2i )
  pot = A*r12i - B*r6i
End Subroutine lj_getinteraction
```

```
!-----------------------------------------------------------------
! This routine calculates A, B parameters from sigma
! and epsilon.  A = 4*eps*(sig**12), B = 4*eps*(sig**6)
!-----------------------------------------------------------------
Subroutine lj_calc_AB(obj)
  Implicit None
  Type(LJ_Params), Intent(INOUT)         :: obj

  obj%B = 4.0_Rdbl*obj%eps*(obj%sig**6)*kcalmole_kb
  obj%A = obj%b*obj%sig**6
End Subroutine lj_calc_AB

!-----------------------------------------------------------------
! This routine calculates the sigma and epsilon values from A and B
!-----------------------------------------------------------------
Subroutine lj_calc_sigeps(obj)
  Implicit None
  Type(LJ_Params), Intent(INOUT)   :: obj

  Real(kind=RDbl)    :: sig6

  sig6    = (obj%A/obj%B)
  obj%eps = (obj%B/(4.0_Rdbl*sig6*kcalmole_kb))
  obj%sig = sig6**(1/6.0)
End Subroutine lj_calc_sigeps

!-----------------------------------------------------------------
! Display the fields of the LJ structure
!-----------------------------------------------------------------
Subroutine lj_display(params, optunitno)
  Type(LJ_Params), Intent(in)    :: params
  Integer, Optional, Intent(in)  :: optunitno

  Integer    :: unitno

  If (Present(optunitno)) Then
    unitno = optunitno
  Else
    unitno = 6    !Standard output
  End If

  Write(unitno,'(4x,3a)')"Atoms   : ", Trim(params%atom_name1), &
      Trim(params%atom_name2)
  Write(unitno,'(4x,a,f8.3)') "Sigma   : ", params%sig
  Write(unitno,'(4x,a,f8.3)') "Epsilon : ", params%eps
  Write(unitno,'(4x,a,e12.4)')"Sigma   : ", params%A
  Write(unitno,'(4x,a,e12.4)')"Epsilon : ", params%B

  End Subroutine lj_display

  !-------------------------------------------------------
  ! Nothing to do here
  !-------------------------------------------------------
  Subroutine lj_cleanup(params)
    Type(LJ_Params), Intent(inout) :: params
  End Subroutine lj_cleanup

End Module lj
```