JCTC Journal of Chemical Theory and Computation

YUP: A Molecular Simulation Program for Coarse-Grained and Multiscaled Models

Robert K. Z. Tan, Anton S. Petrov, and Stephen C. Harvey*

School of Biology, Georgia Institute of Technology, 310 Ferst Drive, Atlanta, Georgia 30332

Received December 16, 2005

Abstract: Coarse-grained models can be very different from all-atom models and are highly varied. Each class of model is assembled very differently, and some models need customized versions of the standard molecular mechanics methods. The most flexible way to meet these diverse needs is to provide access to internal data structures and a programming language to manipulate these structures. We have created YUP, a general-purpose program for coarse-grained and multiscaled models. YUP extends the Python programming language by adding new data types. We have then used the extended language to implement three classes of coarse-grained models. The coarse-grained RNA model type is an unusual nonlinear polymer, and the assembly was easily handled with a simple program. The molecular dynamics algorithm had to be extended for a coarse-grained DNA model so that it could detect a failure that is invisible to a standard implementation. A third model type took advantage of access to the force field to simulate the packing of DNA in viral capsids. We find that objects are easy to modify, extend, and redeploy. Thus, new classes of coarse-grained models can be implemented easily.

1. Introduction

All-atom molecular dynamics (MD) simulations have progressed from the treatment of small proteins in vacuo for a few picoseconds[1] to simulations in solution covering time scales as long as 1 μs.[2] One major challenge for MD simulations is the treatment of very large systems. Perhaps the largest system treated in all-atom detail is the ribosome,[3] for which about 10^6 CPU h were required for MD simulations totaling about 20 ns.

One way to reduce the computational burden for examining very large systems is to use reduced (coarse-grained) representations, in which pseudoatoms are used to represent pieces of the structure. These pseudoatoms may represent only a few atoms, e.g. methyl groups, or they can represent very large groups, depending on the resolution of the model. At very low resolution, a single pseudoatom might represent a protein, a nucleosome, or even larger structures.

The parametrization of all-atom models is based on the chemical properties of the atoms, and the transferability of parameters to a broad collection of molecules is a major goal. In contrast, coarse-grained models are generally developed for specific problems, so their parametrization tends to be empirical. The advent of multiscale models that contains different levels of resolution in different regions renders parametrization even more idiosyncratic.

We developed the Yammp molecular mechanics package with the specific purpose of facilitating coarse-grained modeling.[4] Although it can be used for all-atom modeling,[5] its principal applications have been aimed at tackling very large systems with reduced representations. Among these are supercoiled DNA,[6−8] the ribosome,[9−11] HIV,[12] and other viruses.[13,14] Yammp contains a number of unique force field terms, developed for different applications.

As coarse-grained and multiscale models become more popular, it would be desirable to have a single, integrated package that facilitates the development and application of such models. The revision of Yammp to Yammp 2 provides such a package. It extends the Python programming language, hence it is also known as YUP (Yammp Under Python). This paper describes the new package.

* Corresponding author e-mail: steve.harvey@biology.gatech.edu.

10.1021/ct050323r CCC: $33.50 © 2006 American Chemical Society
Published on Web 03/18/2006

High-quality molecular simulation programs for all-atom models are widely available, and two such programs that influenced this work are AMBER[15,16] and CHARMM.[17,18]

A molecular simulation program has to provide at least two features. The first is a way for users to construct their model. Essentially, this is the process of converting the user's specification of the model (e.g. a base or amino acid sequence) to the terms of an energy function. This function expresses the interactions among the atoms: which atoms are involved in each interaction, the functional form of the interaction, and the specific parameters that are to be used to calculate the interaction. We call this process force field assembly, and ideally the procedure should be extensible to new classes of models. The second feature is the raison d'être for molecular simulation programs: routines for molecular mechanics calculations, and users should be able to extend existing methods and implement new ones.

Conventional, i.e., all-atom, modeling programs can be awkward tools for coarse-grained models. For example, most conventional programs include a mechanism to build only linear polymers with provisions for simple cross-links. This is sufficient to build proteins, nucleic acids, and most other molecules of computational interest. On the other hand, the best way to build a coarse-grained model may not necessarily be as a linear polymer, even if that is what the original is. In fact, a coarse-grained program will have to be much more versatile than a conventional program because the definition of a coarse-grained model is open-ended.

Version 2 of Yammp, or YUP (http://rumour.biology.gatech.edu/YammpWeb/), is designed for the easy implementation of coarse-grained models. This is to be achieved through two features: [1] most of the data structures of the program are user accessible and [2] a high-level language is available to manipulate the data. YUP is a component of the NIH Research Resource Center for the Development of Multiscale Modeling Tools for Structural Biology (http://mmtsb.scripps.edu/). YUP is available for two computer platforms, LINUX on Intel X86 and MacOS X on PowerPC, and can be downloaded without charge from the YammpWeb site.

Most simulation programs provide a scripting or programming language. Perhaps the most sophisticated example is the CNS[19] language, which can be used to implement many new algorithms. The predecessor of the CNS program package is XPLOR, which also influenced the development of Yammp.

2. Method

The Python[20] language (http://www.python.org/) is a good choice for user scripting because it has a simple and clear syntax that is close to natural languages and programs do not have to be explicitly compiled. Thus, it was chosen to be the scripting language in Yammp 1. The most computationally intensive parts of YUP have to be implemented in a low level language, and C was chosen for this purpose. However, high-level languages are much easier to use, and programmers are more productive with these languages. Thus, Python was chosen as a programming language for YUP.

MMTK[21] is a molecular simulation program that is similarly implemented in Python and C, but it is for all-atom models.

An alternative would be to use a bespoke language, as was the case with the CNS package. However, by using a standard language, we are freed from the burden of designing, implementing, and maintaining a language. Furthermore, YUP users will acquire skills in a standard language that can be used to solve other programming problems.

Limited Access to Internal Data. Python is an object-oriented language with a number of predefined data types such as floating-point numbers and strings of characters. YUP extends Python by adding new data types or objects. Programmers are not granted direct access to internal data structures. Instead, controlled access is provided through objects. Objects are bundles of data and code that define their properties and behaviors. The new data types or objects are designed for molecular simulations and are implemented either in the C language for speed or in Python for productivity. However, all the objects are to be used within the Python environment.

Object-Oriented Programming in Python. (This is a discussion of some of the concepts that will be used in this article.) A defining property of objects is *inheritance*: an object can be used to define another object and the derived object has the properties and behaviors of the parent object without having to be reimplemented. The new object may also extend or modify the properties and behaviors of the parent object. Objects or data types are defined in Python using the *class* statement. The class definition includes an initialization function, also known as the constructor. The class or data type definition provides a template for the construction of any number of objects each of which is an *instance* of the data type. The properties of a Python object are usually accessible from the *data attributes*, which are written using the dot notation, e.g. **a.b** is the data attribute named **b** of the object **a**. Some objects may support assignments to certain data attributes, e.g. **a.b=c** assigns the value **c** to the attribute **b** of the object **a**. Object behaviors are usually implemented as *methods*. These are functions written with the dot notation, e.g. **a.m(**...) applies the method **m** (with the arguments unspecified) to the target object **a**. Some objects can be mapped or subscripted, e.g. **a[k]** is a mapping or subscripting of **a** and **k** is the key or subscript. The key may be of any *immutable* type. A data type may also support assignments to mapping, e.g. **a[k]=v** assigns the value **v** to the mapping **a[k]**. Operators may also be defined for an object. These include the arithmetic and *bitwise* operators. For example, in the expression **a/b** the operator is /, and the expression is correct if the operator has been implemented for the objects **a** and **b**. The creator of an object can implement any action for an operator but would usually stay close to the established meaning. The order in which operators are applied (*precedence*) is never changed. One of the most useful data types in Python is the *tuple*, which is an immutable and heterogeneous list, i.e., a list that can contain any type of data but the *tuple* itself cannot be changed. A *tuple* is written as a comma-separated list

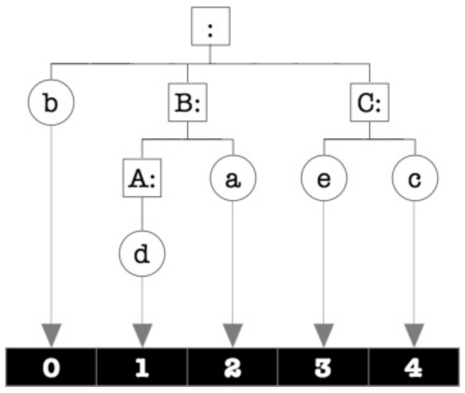

Figure 1. An *AtomMap* hierarchy and an associated *AtomVector* with group *AtomMaps* shown as squares, atom *AtomMaps* as circles, both enclosing the label, and *AtomVectors* are represented by black rectangles enclosing the index.

within parentheses. For example, **(3.142,(0,b),“a string”)** is a *tuple* in which the second item is itself a *tuple*.

Types of Atoms and Interactions. YUP divides the potential energy terms into two categories: *Implicit* and *Explicit*. Interactions of closely bonded atoms such as bonds, angles, and torsions are always explicitly listed; we call such terms *Explicit*. Nonbond interactions involve all atom pairs except those that are explicitly excluded; we call such terms *Implicit*. YUP also provides *Explicit* versions of the nonbond interactions. Each atom can be assigned two atom types: an *inclusion* type and an *exclusion* type. Atom types are used as keys or indices to tables of force field parameters: *inclusion* atom types for parameters of *Explicit* interactions and *exclusion* atom types for parameters of *Implicit* interactions. Thus, parameters are specified separately for each of the two versions of nonbond terms. The parameters are keyed by as many atom types as the number of atoms in the interaction. The list of atom types is called an interaction type. For example, the parameters of the *Implicit* version of the van der Waals term (*VanderWaalsX*) are keyed by two *exclusion* atom types, and the parameters of the torsion term (*Torsions*) are keyed by four *inclusion* atom types.

Every model has to be associated with a specific parameter library. Each library defines a number of *inclusion* and *exclusion* atom types, some *Explicit* interaction types, and all *Implicit* interaction types. An atom can only be assigned an atom type that exists in the parameter library. The only exception is a nameless or *null* atom type. An interaction is assumed to have the *interaction* type that is formed from the atom types of the constituent atoms, but it is possible to assign a specific type to the interaction. In either case the interaction type must have been defined in the library. If at least one atom of an interaction has the *null* atom type, that interaction will not be evaluated. An atom that has a null *inclusion* type does not participate in any of the *Explicit* interactions. Atoms with null *exclusion* atom type are automatically excluded from all *Implicit* interactions.

The *AtomMap* Extension Data Type. This was implemented in the C language and compiled into a shared object (dynamic link library). This is closely associated with the *AtomVector* type that will be discussed later.

There are two varieties of *AtomMaps*. The “atom” variety is used to store atomic properties such as mass, charge, and so on. The “group” variety is used as a container of other *AtomMaps*. *AtomMaps* can be linked to form a hierarchy including the standard chain/residue/atom hierarchy. An *AtomMap* hierarchy resembles a file hierarchy in which the atom *AtomMaps* correspond to files and the group *AtomMaps* are like folders or directories. Thus, an *AtomMap* hierarchy is a filing system for atoms. Figure 1 shows a hierarchy containing five atoms and four groups.

The hierarchy shown in Figure 1 may be constructed as follows:

```
grR = AtomMap( datasource( ″rrDNAv1″ ) )

atb = grR.atom( ″b″, mass=79, … )

grB = grR.group( ″B″ )

grA = grB.group( ″A″ )

atd = grA.atom( ″d″, atomitype=″cnt″, … )

ata = grB.atom( ″a″, atomxtype=″CNT″, … )

grC = grR.group( ″C″ )

ate = grC.atom( ″e″, charge=-1.0, … )

atc = grC.atom( ″c″, mask=25, … )
```

A special group called the root always occupies the top of the hierarchy. It has no name, and it is created by a function (the constructor) instead of methods such as the rest of the hierarchy. The *AtomMap* constructor function requires the path to a library of force field parameters. The example makes use of the **datasource** function to construct the path to the standard parameter library named *rrDNAv1*.

The **atom** method adds a new atom *AtomMap* to an existing target that must be a group *AtomMap*. The expression a.**atom**(“b”) means to the group *AtomMap* a (the target), create and return a new atom *AtomMap* labeled “b”. Additional arguments specify atomic properties such as mass, charge, *inclusion* type, and *exclusion* type. The **group** method is used to create and return a group *AtomMap* that is contained in the target *AtomMap*. In both cases, the returned *AtomMaps* will be printed if they are not assigned to variables. (These variables will be used to illustrate further examples.)

The name of the path between two *AtomMaps* is called the pathname of the destination *AtomMap*. It is formed by concatenating the names of the *AtomMaps* along the path and separating each name with a colon “:”. *AtomMaps* support the mapping operation. The key must be a pathname, and the result of the mapping is the *AtomMap* of the destination. For example, the following are three references to the atom *AtomMap* labeled “d” in Figure 1:

The first line references the variable that was used to hold the atom *AtomMap* at its creation (in the earlier example). The second line shows an example of an absolute path, while the third line shows a relative path.

```
atd

grR[":B:A:d"]

grA["d"]
```

The *AtomVector* Extension Data Type. This is also implemented in the C language. *AtomVectors* contain arrays of floating-point numbers that can be manipulated in a limited number of ways, by using operators, methods, and functions. The operations include scalar and vector arithmetic, rotations and other coordinate transformations, and file operations. Generally, binary operations require conformal *AtomVectors*, i.e., the operands must have the same number of rows and columns. The contents of an *AtomVector* can be packed into or unpacked from a Python *tuple*. This allows for operations beyond those that have been defined for the *AtomVector* data type.

AtomVectors are used to represent such data as coordinates, velocities, gradients, atomic mass, atomic charges, and so on. An *AtomVector* can be associated with an *AtomMap* hierarchy.

The *AtomVector* shown in Figure 1 can be created as follows:

```
C = AtomVector( grR, numdimen = 3 )
```

This creates an *AtomVector* that is associated with the *AtomMap* hierarchy whose root is referenced by the variable *grR*. The *AtomVector* is assigned to the variable *C*, and it provides storage for five atoms and three dimensions. The number of dimensions is assumed to be three if it is not specified; thus the second argument in this example is redundant.

Any *AtomMap* from an *AtomMap* hierarchy can be used to map an *AtomVector* that is associated with this hierarchy. The result is another *AtomVector* that shares storage with the parent *AtomVector*. The derived *AtomVector* is called a slice or extract. Mapping always results in a single contiguous slice. For example

```
C[grB]
```

produces the slice with the indices 1 and 2 in Figure 1. This can also be accomplished with a function:

```
Extract( C, 1, 3 )
```

This is the only way to slice an *AtomVector* that is not associated with an *AtomMap* hierarchy.

Both the original and the extract are *AtomVectors*, and they have many properties and behaviors in common. (One of the few differences is as follows: an original *AtomVector* can be sliced many ways; extracted *AtomVectors* cannot be sliced at all.)

The molecular mechanics algorithms are implemented using *AtomVector* operations. The following is the statement that is at the heart of the steepest descent procedure for energy minimization

```
…
C.shift( G * stepsize )
…
```

where the ellipses represent the code that have been excised from the example: these include energy and gradient evaluations and such housekeeping codes as updating the nonbond list and printing intermediate results.

C is the *AtomVector* containing the coordinates, *G* is the *AtomVector* of the first derivatives, and *stepsize* is the negative step size, a scalar quantity. This example shows the use of an operator and a method.

For *AtomVectors*, the * operator is programmed to multiply the value from each cell of the left operand with the value from the corresponding cell of the right operand. Thus, the *AtomVectors* must be conformal, and the results are placed in a new *AtomVector* that is conformal with the operands. The right operand can be a scalar variable, in which case it is equivalent to having a conformal *AtomVector* that has the same value in every cell. The **shift** method adds the *AtomVector* specified in the argument to the target *AtomVector*, i.e., in-place addition. As with operators, the target and argument *AtomVectors* must be conformal, and the argument can also contain a scalar.

This code fragment applies equally to original *AtomVectors* or to their extracts, as long as the operands are conformal. All the current molecular mechanics methods are implemented in Python using *AtomVector* operations. Thus, these methods can be applied to an entire model or to contiguous slices of the model. Each method accepts a *limit* argument containing an *AtomMap*, and the calculations will be limited to the atoms contained in this *AtomMap*. The parts of the model that are not in the slice are not affected by these methods. If no limit is specified, then the calculations will be carried out on the entire model.

To take advantage of this capability, the *AtomMap* hierarchy has to be constructed so that slices of the model can be selected. The slices may overlap. This might be useful for piecewise refinement for example.

A YUP program runs under the control of a Python interpreter. The latter runs Python extension modules directly, while C extension modules are dynamically linked for execution. We will illustrate this with a simple example. The following defines a function to calculate and return the center of mass of an *AtomVector*.

```
def CenterOfMass( V ):

    from Yup.Taro.AtomVect import Reduction

    return Reduction( V, 0 ) / V.numatom
```

The **Reduction** function sums the rows or columns of the *AtomVector* specified in the first argument depending on the value specified in the second argument. The results are returned in a new *AtomVector*. The **Reduction** function is part of the *AtomVect* module, which was implemented in C and compiled into native machine code. The **CenterOfMass** function takes the *AtomVector* returned by **Reduction** and divides it by the number of atoms. The latter is available for any *AtomVector* as the *numatom* data attribute.

As an example of how the new function might be used, we will move the coordinates *R* so that the center of mass is at the origin:

```
R.shift( -CenterOfMass( R ) )
```

The Python interpreter will process the arguments first. *R* is a simple expression that needs no processing. Next, the **CenterOfMass** function will be evaluated. Like any extension module, *AtomVect* contains a table that maps each syntax element (such as the dot operator, the division operator, and the **Reduction** function in the definition of **CenterOfMass**) to an appropriate function. The value returned by **CenterOfMass** is an *AtomVector*. This is negated before it is passed to the **shift** method. These operations are also implemented as functions in the *AtomVect* module. In essence, the Python interpreter translates a YUP program into a list of functions.

The Energy Extension Data Types. These were implemented in the C language. These types are called *Bonds*, *Angles*, *Torsions*, and *VanderWaalsX*, and there are seven others. We will use the term represented by the *Angles* object for this discussion. This has the form

$$\sum_{i=1}^{N_\theta} k_{\theta,i}[\theta_i - \theta_{0,i}]^2$$

There are N_θ terms in total, and for the *i*th term, the force constant is $k_{\theta,i}$, the prevailing angle is θ_i while the equilibrium angle is $\theta_{0,i}$. To use any particular potential energy term in a model, the appropriate object must be created first:

```
Eangle = Angles( grR, "Angles" )
```

This creates an *instance* of the *Angles* data type. It is associated with the *AtomMap* hierarchy *grR*. The force field parameters will be obtained from the library that was named when *grR* was created. This object is given a label "Angles" to identify it; it can also be used to label printouts.

At this point, the *Angles* object is empty. In a moment, we will be adding the individual interactions.

The *Angles* object can be mapped with a string as key. For example

```
Eangle[":frnG:cntG:lftG:"]
```

The string is an *interaction* type, and it contains the names of three *inclusion* atom types, "frnG", "cntG", and "lftG", set off by colons ":". This mapping returns a *tuple* of numbers, the values (k_θ,θ_0) that have been defined for the *Angles* type ("frnG", "cntG", "lftG").

We can assign new values to an existing *Angles* type. For example, to preserve the force constant and change the equilibrium angle to 108°:

```
atype = ":frnG:cntG:lftG:"

ktheta, theta0 = Eangles[atype]

Eangles[atype] = ( ktheta, 108.0 )
```

(It is easier to assign the complicated *interaction* type string to a variable instead of having to type it twice.) This changes the parameters for this *interaction* type just for this *Angles* object.

The heart of force field assembly is the addition of the individual interactions. For example, to add an *Angles* interaction that involves the atoms in Figure 1, labeled "b", "e", and "d":

```
Eangle.add( ( atb, ate, atd ) )
```

This interaction is added to the *Angles* object *Eangle* using the **add** method. Note that the argument contains only one item, a *tuple* that happens to contain three values. The atoms are represented by their *AtomMaps*, which were assigned to variables in an earlier example.

When an interaction is added like the above, it is assumed that the interaction type is to be constructed from the atom types of the constituent atoms. The parameter library must contain data for this interaction type. If any one atom has a null atom type, then the interaction will not be evaluated even though it will remain in the *Angles* object. A specific interaction type can be specified as the second argument like this:

```
Eangle.add( ( atb, ate, atd ),
        ( "frnG", "cntG", "lftG" ) )
```

(Note that the *Angles* type is expressed differently from the key used in the earlier mapping.) If any one of the atom type names is replaced with a null string, the angle will still be registered but will not be evaluated.

When all the interactions have been added, the *Angles* object must be compiled before the energy (and gradients) can be evaluated.

```
Eangle.compile( C, G )
```

This registers the *AtomVector C* as atomic coordinates, and the *AtomVector G* will be used for the first derivatives. At this time, the *Angles* type is determined for each interaction: whether it is to be constructed from the atom types, or a specific type is to be used. Compilation fails if any interaction type cannot be found in the parameter library. Null interactions are not added to the list of interactions that are to be evaluated.

To evaluate the *Angles* energy

```
Eangle.evaluate()
```

The energy is evaluated for the conformation defined by the coordinates *AtomVector* registered earlier with the **compile** method. The energy is expressed in internal units.

The force field can be modified on the fly during a simulation, by applying the **modify** method, but the **compile** method must be applied to commit to the changes. For example, by writing

```
Eangle.modify( ( atb, ate, atd ) )

Eangle.compile()
```

This interaction will now be evaluated using the default *interaction* type instead of the specific type that was assigned earlier.

Table 1. Data Attributes of the *Model* Data Type

Name of Attribute	Data Type
Map[a]	*AtomMap*
Energy[a]	*Potential*
Coordinates[a]	*AtomVector*
Gradients[b]	*AtomVector*
Velocities[b]	*AtomVector*
InverseMass[b]	*AtomVector*

[a] The first three attributes must be added to a new *Model* object. [b] The remaining attributes will be created as the need arises.

It is possible to have more than one Energy object of the same kind. One reason to do this is to be able to isolate the interactions whose parameters are to be modified dynamically.

Other Energy objects are used in much the same way. Thus, for the *Bonds* object, each interaction requires two *AtomMaps* and two *inclusion* atom types. For the *Torsions* and *Impropers* types, four *AtomMaps* and four *inclusion* atom types are required. These are all *Explicit* Energy types; the interactions have to be added explicitly.

Some terms are also available in an *Implicit* version for example *VanderWaalsX*. An *Implicit* Energy object is completely full as soon as it is created. Exclusions have to be registered, by applying the **exclude** method on each atom pair that is to be excluded.

All nonbond Energy terms, whether *Implicit* or *Explicit*, are evaluated from a list of candidate pairs that have been explicitly added or have not been excluded or are not *null* interactions and are within a certain threshold distance. This list is updated periodically by applying the **update** method.

The *Potential* and *Models* Extension Data Type. These objects are implemented in Python. They are container objects, to hold lists of related objects so that they can be manipulated together. For instance, the *Potential* object holds a list of Energy objects. Thus, the **compile**, **evaluate**, and **update** methods can be applied to all the Energy objects without knowing what these objects are.

Defined for both data types are methods to add, delete, and find the component objects.

The *Model* object always consists of the root *AtomMap*, a *Potential* object, and four *AtomVectors*. These components define a complete model. The components are available as data attributes as listed in Table 1.

A *Model* data type may contain additional attributes. The *Draw* attribute specifies how a model is to be represented visually. If a *Model* object has this attribute, YUP can produce input files for the Kinemage[22] and POV-Ray (http://www.povray.org/) programs. This is especially useful for coarse-grained models, as most molecular graphics programs cannot handle idiosyncratic stereochemistry.

A *Model* type may also define attributes that are unique to a model type.

Thus, we have extended the Python language to be able to write programs to simulate coarse-grained models. Some simulation algorithms require users to define a target or objective function. The function will depend on the model and the problem, and therefore it is impractical to incorporate all possible functions into a program. For example, the CONTRA[23] algorithm to construct a minimally biased path between two conformations requires a user-defined function to provide a score for each conformation along the path. YUP is the ideal platform for such algorithms: many internal data structures are accessible to the user, and the user can use a powerful programming language to access the data and write a scoring function.

The YUP Package. YUP is a Python package, a collection of code modules organized in a file hierarchy. When a YUP program runs, the Python interpreter links each code module as it is needed. Code modules can be added to the package without disturbing existing code. These modules take care of memory management. Object size is theoretically limited only by the word size of the computer (typically 32 bits, increasingly 64 bits). The actual limits are determined by the amount of memory available on a system.The extension objects that were discussed earlier are in two subdirectories: *Yup/Energy/* for the Energy objects and *Yup/Taro/* for the remaining objects.

The molecular mechanics routines are in *Yup/Methods/*. There are routines for two energy minimization methods, a molecular dynamics method and three variants of the Monte Carlo method. The molecular mechanics algorithms are not implemented as functions but are objects derived from the *MolMech* parent object.

General-purpose utilities are kept in the *Yup/Tools/* directory. These are usually implemented as objects instead of procedures. Two of these modules are particularly important. The *FFA* module implements a template for a minimal force field assembler. The *Chains* module defines the *Chains* class, based on *FFA*, and it is a template for a force field assembler that links atoms into a simple unbranched chain. These two objects can be used as the basis for force field assemblers for more complex model types.

Three model types have been implemented so far, and they will be discussed shortly. Each model type requires a standard parameter library. The parameter library for a model type *ModelType* can be found in *Yup/Data/ModelType/*, and the code will be in *Yup/Models/ModelType/*. These models are implemented as objects that can be used to create future models. Programs that make use of the molecular simulation objects without reference to a specific model type can use the empty parameter library *Yup/Data/Null/*.

3. Results

The *rrRNAv1* Model Type. This is a coarse-grained model of RNA where each nucleotide is represented by one pseudoatom.[10] Helices are explicitly represented by semirigid structures in which hydrogen bonds between the strands are replaced by unbreakable interactions. The *rrRNAv1* model type cannot be easily assembled in conventional modeling programs, but a simple procedure has been developed for YUP.

A small RNA will be used as an example: the *rrRNAv1* model of the tRNA^{Phe} molecule is shown in Figure 2.

This model can be viewed as a linear polymer of 76 monomers (each consisting of one pseudoatom), but the helices would have to be built separately (by cross-linking the monomers within each helix to give the proper three-

Table 2. YUP Package Directory[a]

Yup/					
Taro/	Energy/	Tools/	Methods/	Data/	Models/
AtomMap.so	Angles.so	Atoms.py	EnerMinim.py	rrDNAv1/	rrDNAv1/
AtomVect.so	Bonds.so	AutoGen.py	MolDynam.py	rrRNAv1/	rrRNAv1/
Model.py	ElectroI.so	Chains.py	MolMech.py	VirPack/	VirPack/
Potential.py	ElectroX.so	ChemNames.py	MonteCarlo.py	Null/	
	Impropers.so	FFA.py			
	Noens.so	Groups.py			
	SoftSphereI.so	LinSeq.py			
	SoftSphereX.so	MakeGraph.py			
	Torsions.so	MakeLine.py			
	VanderWaalsI.so	ParmTop.py			
	VanderWaalsX.so	TraceChain.py			
		Topology.py			
		misc.py			

[a] *.so files are shared objects or dynamic link libraries; *.py files are Python modules and directory names end in "/".

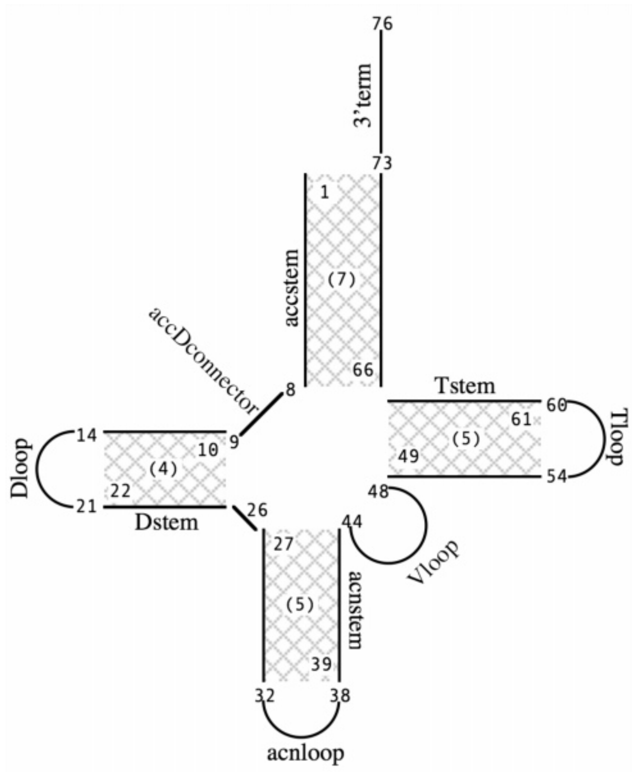

Figure 2. The *rrRNAv1* model of a tRNA^{Phe} molecule: cross-hatched regions represent helices, the nucleotide indices are shown as unadorned numbers, numbers within parentheses indicate the lengths of helices, and each monomer is named except for nucleotide 26.

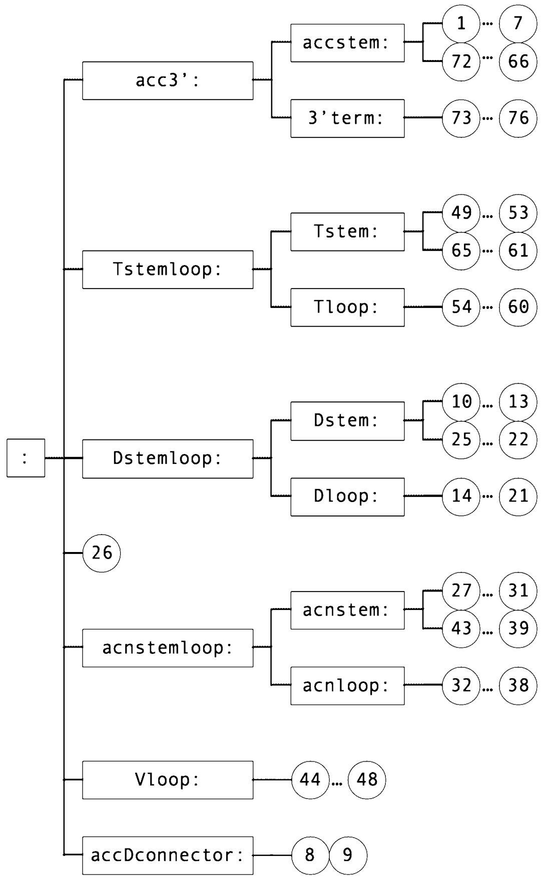

Figure 3. An *AtomMap* hierarchy of tRNA^{Phe} with atom *AtomMaps* represented by circles, group *AtomMaps* by rectangles.

dimensional geometry dictated by base pairing), which would be a difficult task even for such a small RNA. A more systematic approach would be to treat the model as a nonlinear polymer of four helices and seven single-stranded tracts. The procedure in YUP relies on an *AtomMap* hierarchy in which each monomer is represented by a group *AtomMap*. Figure 3 shows one of many feasible hierarchies.

Note the additional levels in the hierarchy, e.g. *Dstem* and *Dloop* are placed in the *Dstemloop* group. Further groupings are possible, e.g. *Dstemloop*, nucleotide 26, and *acnstemloop* could all be placed in a new group; *acc3′* and *Tstemloop* could be placed in another. A hierarchy can be chosen to divide the model into possibly overlapping domains. Each domain can then be selected and treated as a rigid unit or as a focus of molecular mechanics calculations.

The desired hierarchy can be expressed as a *tuple* that contains components of three types. Each type is indicated by a keyword (*TRACT*, *HELIX*, or *DOMAIN*), followed by a label, and the specifications of a *TRACT* or *HELIX* or the contents of a *DOMAIN*. (The secondary structure may be read from a file, such as an RNAML[24] file, but the higher levels of the hierarchy must be added manually.) The hierarchy of Figure 3 can be expressed as

```
( DOMAIN, “”, (
     ( TRACT, “accDconnector”, ( 8, 9 ) ),
     ( TRACT, “Vloop”, ( 44, 48 ) ),
     ( DOMAIN, “acnstemloop”, (
          ( HELIX, “acnstem”, ( 27, 5, 39 ) ),
          ( TRACT, “acnloop”, ( 32, 38 ) ) )),
     ( TRACT, “”, ( 26, 26 ) ),
     ( DOMAIN, “Dstemloop”, (
          ( HELIX, “Dstem”, ( 10, 4, 22 ) ),
          ( TRACT, “Dloop”, ( 14, 21 ) ) )),
     ( DOMAIN, “Tstemloop”, (
          ( HELIX, “Tstem”, ( 49, 5, 61 ) ),
          ( TRACT, “Tloop”, ( 54, 60 ) ) )),
     ( DOMAIN, “acc3′”, (
          ( HELIX, “accstem”, ( 1, 7, 66 ) ),
          ( TRACT, “3′term”, ( 73, 76 ) ) ))))
```

The contents of a *DOMAIN* are indented for clarity. This definition is constructed in one step, but larger RNAs will have to be defined in stages. The program to unravel an RNA definition, such as this, is very simple as the following code outline shows:

```
def adddomain( parent, parts ):
     G = makegroup( parent, parts[1] )
     if parts[0] == TRACT: addtract( G, parts[2] )
     elif parts[0] == HELIX: addhelix( G, parts[2] )
     elif parts[0] == DOMAIN: adddomain( G, parts[2] )
     else: raise RuntimeError, “unknown keyword”
```

The **adddomain** function constructs the *AtomMap* hierarchy, defined in the *parts* argument, underneath the group *AtomMap*, defined in the *parent* argument. The *parts* argument is a tuple containing three items: *parts[0]*, the keyword; *parts[1]*, the label; and *parts[2]*, the specification of *HELIX* or *TRACT* or the contents of *DOMAIN*. If the label is not a blank string, the **makegroup** function constructs a new group *AtomMap* under *parent* and returns the new group. Otherwise, the function returns the value of *parent*. Then, the keyword is parsed to build either a *HELIX* or a *TRACT* using the specification in the final part. The functions **addtract** and **addhelix** add the specifications for one and two tracts respectively to a buffer. The buffer is later used to add the interactions that are needed to join the tracts into a continuous chain. When the keyword is *DOMAIN*, the **adddomain** function is called to unravel *its* content. The program is short and easy to understand.

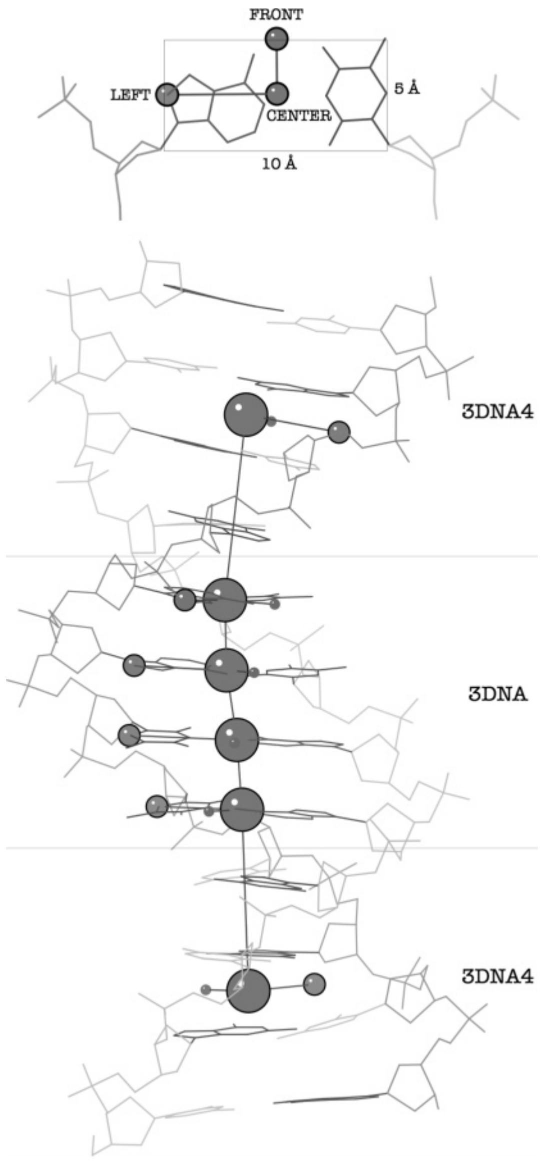

Figure 4. The conventional (all-atom) and *rrDNAv1* models of an isolated base pair (top) and a DNA duplex (bottom) are shown as wire-frames and spheres, respectively. The top panel shows how the pseudoatoms that make up the *rrDNAv1* base plane are related to the atoms of a base pair. The bottom panel shows the conventional and a multiscaled *rrDNAv1* model of a dodecamer. This figure was made using KiNG, a Kinemage[22] display program.

The *rrDNAv1* Model Type. This model type represents double-helical DNA at two levels of detail or scale: three pseudoatoms per base pair (*3DNA*)[6] and three pseudoatoms for every four base pairs (*3DNA4*). A model may contain both representations. The *rrDNAv1* model is used in the study of large pieces of DNA of hundreds to thousands of base pairs. Figure 4 shows a multiscaled *rrDNAv1* model of a dodecamer.

The *rrDNAv1* pseudoatoms, *CENTER*, *FRONT*, and *LEFT*, mark out a quadrant of a base plane. The latter is an idealized rectangle that lies in the average plane of the base pair. Idealized base pairs can be reconstituted from the *rrDNAv1* base plane. Note that the *3DNA4* base plane lies between the second and third base pairs of the tetramer that these pseudoatoms represent. The base plane is also twisted equally between these base pairs. Only the *CENTER* pseudoatoms have bulk; they are connected to form a flexible rod.

The *rrDNAv1* force field assembler is based on the *Chains* template in *Yup/Tools/Chains.py*, which provides the code to assemble the *CENTER* atoms. The additional code is for the assembly of the *LEFT* and *FRONT* atoms. The *rrDNAv1* model is of a linear polymer (cyclized in closed circular DNA), and force field assembly is a straightforward process.

The nonbond interactions are somewhat unusual. As stated earlier, only the *CENTER* atoms have bulk, i.e., only one out of every three atoms takes part in nonbond interactions. Therefore, eight out of nine atom pairs never need to be considered. The list of interactions in the *Explicit* version of a nonbond term will be much shorter than the list of exclusions in the *Implicit* version. Thus, the *rrDNAv1* model type has historically used the *Explicit* version of the repulsion-only term (*SoftSphereI*). With support for *SoftSphereI* still in place, the *rrDNAv1* model now uses the *Implicit* version (*SoftSphereX*). Tests showed that *SoftSphereX* is significantly slower than *SoftSphereI* if a conventional exclusion list is used. However, if the *LEFT* and *FRONT* atom are assigned null *exclusion* atom types, these atoms will be automatically excluded from all *Implicit* interactions. Without the need to process a long exclusion list, *SoftSphereX* is now only slightly slower than *SoftSphereI*. (The switch to *SoftSphereX* allows *rrDNAv1* models to be used in *AMBER*, where all the nonbond terms are *Implicit*.)

The two strands of a circular DNA wind around each other, and we can count the number of times each strand crosses the other. This count is called the Linking Number.[25] Underwinding the DNA by *N* full turns (360*N*°) and closing the circle reduces the linking number by *N*. Overwinding increases the linking number.

The twist of two successive base pairs is modeled by an improper torsion angle between the equivalent *rrDNAv1* base planes. An underwound circular DNA can be constructed by setting all improper torsion angles below the equilibrium values. (The necessary offset at each base plane is 360° multiplied by the number of turns desired and divided by the number of base pairs in the model.) Given the chance, a linear model will relax, and the torsions will be returned to the equilibrium values. On the other hand, a circular model can only relieve the torsional stress by bending the helix. If the linking number is in sufficient deficit or excess, the DNA supercoils, i.e., the double helix winds around itself.

Supercoiling can occur only as long as the *CENTER* atoms form an impenetrable barrier to strand crossing. If permitted, strand crossing would relieve torsional stresses, and the linking number would revert to the equilibrium value. Further calculations would be unproductive since the model is no longer correct. We would like to detect failed calculations and terminate them quickly.

The problem is one of incorporating a nonstandard feature (calculation of the linking number) into a standard molecular mechanics method. The standard molecular dynamics class *Motors*, defined in *Yup/Methods/MolDynam.py*, applies the method **userafter** every *UserAfterInterval* steps of the procedure to solve the equations of motion. The solution is to customize the *Motors* class to have the properties and behaviors that are relevant to *rrDNAv1* models. Thus, we first derive a new class, *_motors*, from the *Motors* class:

```
class _motors( Motors ):

    def __init__( self, dnamodel ):

        Motors.__init__( self, dnamodel )

        self.UserAfterInterval = 10000

        self.__model = dnamodel

        self.__0link = CountLink( dnamodel )

        #… other settings

    def userafter( self ):

        if CountLink( self.__model ) != self.__0link:

            raise RuntimeError, “strands crossed, quitting…”
```

The new object ***_motors*** inherits the properties and behaviors of *Motors*, including the all-important methods to integrate the equations of motion. A function and a method are defined for the new class. The function **__init__** is special: it is called the constructor function, and it initializes a new instance of this class. Note that the parent class is initialized as part of the creation and initialization of the new class. The initialization routine sets a default value for the *UserAfterInterval* attribute (the user can set it to any value later on). The linking number is calculated (using the **CountLink** function), and the value is saved in the instance variable ***__0link***. The **userafter** method definition overrides the method of the same name in the parent class. This method, called every *UserAfterInterval* steps, is now made to calculate the linking number and to compare it with the value saved by the initialization routine. If the numbers differ, then strand crossing has occurred, and the simulation is interrupted.

The *rrDNAv1* module can now use the ***_motors*** class, instead of *Motors*, to define molecular dynamics and simulated annealing (used for structural refinement) calculations. Thus, both structural refinement and simulations can benefit from the strand-crossing check, and failed calculations can be detected and terminated quickly.

The key feature of this implementation is the ability to store the initial linking number for the lifetime of the simulation. It would be much harder to do this if the simulation method were to be implemented as a traditional procedure.

The *VirPack* Model Type. This model type is used to study the packaging process in DNA viruses. The model represents a double-stranded DNA as a continuous chain containing one pseudoatom for every six base pairs, and the capsid is represented either as a sphere or a polyhedron. The shape and the size of the model depend on the virus that is being modeled. The *VirPack* model type makes use of many of the special features of YUP: access to internal data structures, null atom types, an unconventional *AtomMap* hierarchy, and molecular dynamics on slices of the model.

The *VirPack* force field assembler is derived from two classes: *Chains* to assemble the DNA and *Shapes* to assemble the capsid.

The class *Chains* is derived from the *Chains* template defined in *Yup/Tools/Chains.py*. The derived *Chains* class

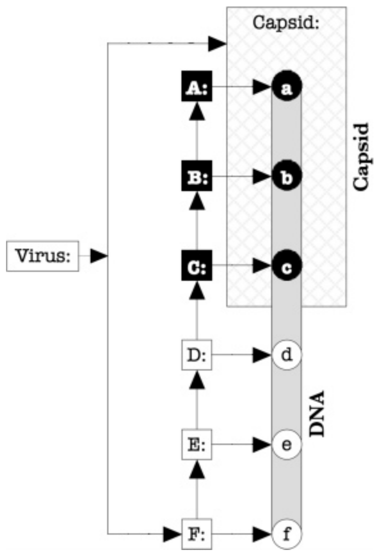

Figure 5. The *VirPack* model of a virus containing six DNA pseudoatoms (circles) linked to groups (rectangles) into a stepped hierarchy, showing the *AtomMaps* that are enclosed by the group labeled "C" (filled circles and rectangles).

links the DNA pseudoatoms into a stepped hierarchy, which allows the selection of an increasing length of the molecule. At this point in the assembly, the DNA pseudoatoms are assigned null atom types, i.e., these atoms contribute nothing to the force field. Figure 5 shows the stepped *AtomMap* hierarchy of a small system.

The *Shapes* class provides methods to assemble the capsid. The simplest type of capsid is a spherical cavity sized to match the volume of a virus, and it requires only one additional atom, the focal atom at the center of the sphere. The pseudoatoms of the DNA are subject to forces directed toward the focal atom for only as long as they are outside the capsid. The *Shapes* class is also capable of assembling more realistic capsids. Two polyhedra (icosahedron and a standard or elongated pentakisdodecahedron) of any size are available now. Barrier atoms are placed on the vertices, edges, and faces of the selected polyhedron. The barrier atoms are spaced as closely as it is required to prevent the passage of the DNA pseudoatoms. An entry hole through the capsid DNA (made by assigning null atom types to the barrier atoms in the hole) allows the passage of DNA. A more elaborate entry channel can also be constructed to mimic the portal proteins of the virus. The process of placing the barrier atoms on an icosahedron is illustrated in Figure 6.

As described earlier, the initial virus model contains a capsid part and a DNA component that is invisible to the force field. The packing procedure starts with the DNA pseudoatoms aligned on a straight line just outside the capsid. The coordinates of the exterior DNA are translated toward the capsid by half the length of the DNA−DNA bond. This pushes one DNA pseudoatom into the capsid, and this pseudoatom is now assigned the proper *inclusion* and *exclusion* atom types. The **compile** method is applied to all the Energy objects, and the newly injected DNA pseudoatom is now subject to the force field, as are all the pseudoatoms that are already in the capsid. All the interior pseudoatoms are selected for a molecular dynamics simulation. This relaxes the compressed bond near the entry as the DNA explores new conformations.

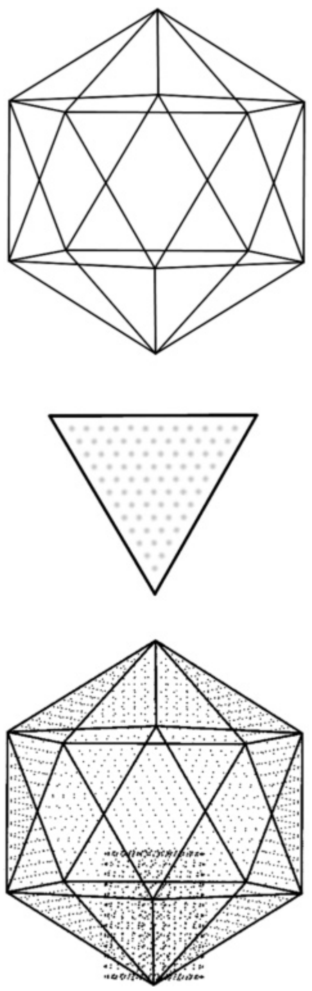

Figure 6. Construction of an icosahedral capsid: starting from the bare icosahedron (top), a template is triangulated to the necessary spacing, the grid is populated with barrier atoms (middle), and the template is copied and transformed into all the faces (bottom). The pseudoatoms defining the cylindrical core associated with the portal are also shown (bottom).

In Figure 5, the group labeled "A:" is shown linked to the DNA pseudoatom with the unique label "a". The next group up the hierarchy, labeled "B:", contains two pseudoatoms labeled "a" and "b". By going up the hierarchy, an increasing length of the DNA can be selected for molecular dynamics simulation. The group labeled "F:" encloses all the DNA pseudoatoms (for this simple system). The packing simulation is complete once the last DNA pseudoatom has been injected into the capsid and the last molecular dynamics simulation is finished. The capsid atoms, either the focal atom of a spherical capsid or the barrier atoms of a polyhedral capsid, are never subjected to molecular dynamics.

The results of a study of virus packing in spherical capsids are reported elsewhere.[26] Work on the polyhedral capsids is underway; these calculations are now taking two to four times longer to complete. Another elaboration of the model is to account for electrostatic interactions; these simulations are taking about three times the time needed for the earlier

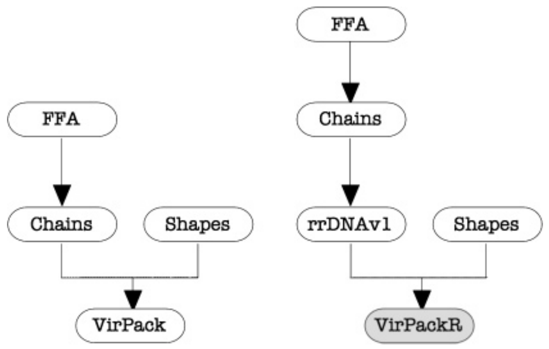

Figure 7. The DNA in the current *VirPack* model type is represented by *Chains* (left) or by *rrDNAv1* (right). The former is the simplest elastic model, with no torsional stiffness, while the latter is parametrized to match the torsional stiffness of DNA, thus mimicking the writhing and twisting of real DNA.

simulations (that lack electrostatics). Clearly, the virus model will have to remain coarse-grained even as it is enhanced.

4. Discussion

Access to internal data structures makes it possible to implement a diverse group of coarse-grained model types in YUP. Each model type has to be assembled in very different ways: *rrRNAv1* models as nonlinear polymers, *rrDNAv1* as conventional models but with very unusual nonbond interactions, and the atoms of *VirPack* models are organized in an unorthodox hierarchy. The data structures that represent the potential energy function were constructed in very different ways to suit each model. Access to internal data structures also makes it possible to customize the molecular simulation algorithms. For example, the *rrDNAv1* model can break down during a molecular dynamics simulation, but the failure is not detectable in a standard algorithm. The algorithm was extended to monitor strand crossing, and unproductive *rrDNAv1* simulations can now be avoided. The molecular dynamics algorithm is customized for the *VirPack* model to modify the force field on the fly in order to simulate the injection of DNA into a viral capsid.

The internal data structures are not directly accessible but are presented to the programmer as molecular simulation objects. This limits how the data are manipulated, but it also makes programming easier. The objects represent a higher level of abstraction than is possible with the underlying data structures. For example, the underlying data structure for the *AtomVector* object is an array of floating-point numbers. A large number of operations, including vector arithmetic, have been implemented for this object type. Thus, we can write more compact programs without the details that can obscure the algorithm.

The biggest benefit from the object-oriented approach is the ease by which objects can be extended, modified, and redeployed. Earlier, we showed how the molecular dynamics object was extended and modified for the *rrDNAv1* model type. As an example of object redeployment, consider the *VirPack* model type. The current model combines the DNA object (*Chains*) and the viral capsid object (*Shapes*). The DNA model lacks torsional stiffness, a defining feature of the molecule. This feature exists in *rrDNAv1*, a coarse-grained model of DNA. If we combine the *rrDNAv1* object with the *Shapes* object, we would then have a more realistic virus packing model (Figure 7).

Some code revisions will be required. A method has to be added to organize the *3DNA* pseudoatoms into the stepped hierarchy depicted in Figure 5. The customized packing procedure has to assign legal *rrDNAv1* atom types to each DNA unit (now consisting of three pseudoatoms) that is injected into the viral capsid. This is still a lot less effort that is required with a traditional approach.

Whenever possible, existing objects should be used to implement new model types. If it is necessary to write new code, it should be implemented as reusable objects. As the number of base objects increase, new models should be easier to implement.

Acknowledgment. This work is supported by NIH Grant P42 RR 12255 (C. L. Brooks III, P.I.).

References

(1) McCammon, J. A.; Gelin, B. R.; Karplus, M. Dynamics of folded proteins. *Nature* **1977**, *267*, 585−590.

(2) Duan, Y.; Kollman, P. A. Pathways to a protein folding intermediate observed in a 1-microsecond simulation in aqueous solution. *Science* **1998**, *282*, 642−643.

(3) Sanbonmatsu, K. Y.; Joseph, S.; Tung, C. S. Simulating movement of tRNA into the ribosome during decoding. *Proc. Natl. Acad. Sci. U.S.A.* **2005**, *102*, 15854−15859.

(4) Tan, R. K. Z.; Harvey, S. C. Yammp: Development of a Molecular Mechanics Program Using the Modular Programming Method. *J. Comput. Chem.* **1993**, *14*, 455−470.

(5) De Loof, H.; Harvey, S. C.; Segrest, J. P.; Pastor, R. W. Mean Field Stochastic Boundary Molecular Dynamics Simulation of a Phospholipid in a Membrane. *Biochemistry* **1991**, *30*, 2099−2113.

(6) Tan, R. K. Z.; Harvey, S. C. Molecular Mechanics Model of Supercoiled DNA. *J. Mol. Biol.* **1989**, *205*, 573−591.

(7) Sprous, D. Tan, R. K. Z.; Harvey, S. C. Molecular Modeling of Closed Circular DNA Thermodynamic Ensembles. *Biopolymers* **1996**, *39*, 243−258.

(8) Tan, R. K. Z.; Sprous, D.; Harvey, S. C. Molecular Dynamics Simulations of Small DNA Plasmids: Effects of Sequence and Supercoiling on Intramolecular Motions. *Biopolymers* **1996**, *39*, 259−278.

(9) Malhotra, A.; Harvey, S. C. A Quantitative Model of the *Escherichia coli* 16S RNA in the 30S Ribosomal Subunit. *J. Mol. Biol.* **1994**, *240*, 308−340.

(10) Malhotra, A.; Tan, R. K. Z.; Harvey, S. C. Modeling Large RNAs and Ribonucleoprotein Particles using Molecular Mechanics Techniques. *Biophys. J.* **1994**, *66*, 1777−1795.

(11) Valle, M.; Zavialov, A.; Li, W.; Stagg, S. M.; Sengupta, J.; Nielsen, R. C.; Nissen, P.; Harvey, S. C.; Ehrenberg, M.; Frank, J. Incorporation of aminoacyl-tRNA into the ribosome as seen by cryo-electron microscopy. *Nat. Struct. Biol.* **2003**, *10*, 899−906.

(12) Elgavish, T.; VanLoock, M. S.; Harvey, S. C. Exploring Three-Dimensional Structures of the HIV-1/tRNALys3 Initiation Complex. *J. Mol. Biol.* **1999**, *285*, 449−453.

(13) Arsuaga, J.; Tan, R. K. Z., Vazquez, M.; Sumners, de W.; Harvey, S. C. Investigation of viral DNA packaging using molecular mechanics models. *Biophys. Chem.* **2002**, *101*−*102*, 475−484.

(14) LaMarque J. C.; Le, T. V.; Harvey, S. C. Packaging double-helical DNA into viral capsids. *Biopolymers* **2004**, *73*, 348−355.

(15) Case, D. A.; Darden, T. A.; Cheatham, T. E. III; Simmerling, C. L.; Wang, J.; Duke, R. E.; Luo, R.; Merz, K. M.; Wang, B.; Pearlman, D. A.; Crowley, M.; Brozell, S.; Tsui, V.; Gohlke, H.; Mongan, J.; Hornak, V.; Cui, G.; Beroza, P.; Schafmeister, C.; Caldwell, J. W.; Ross, W. S.; Kollman, P. A. *AMBER 8*; University of California: San Francisco, CA, 2004.

(16) Pearlman, D. A.; Case, D. A.; Caldwell, J. W.; Ross, W. S.; Cheatham, T. E. III; DeBolt, S.; Ferguson, D.; Seibel, G.; and Kollman, P. AMBER, a package of computer programs for applying molecular mechanics, normal mode analysis, molecular dynamics and free energy calculations to simulate the structural and energetic properties of molecules. *Comput. Phys. Commun.* **1995**, *91*, 1−41.

(17) Brooks, B. R.; Bruccoleri, R. E.; Olafson, B. D.; States, D. J.; Swaminathan, S.; Karplus, M. CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations. *J. Comput. Chem.* **1983**, *4*, 187−217.

(18) MacKerell, A. D., Jr.; Brooks, B.; Brooks, C. L. III; Nilsson, L.; Roux, B.; Won, Y.; Karplus, M. In *The Encyclopedia of Computational Chemistry*; von Ragué Schleyer, P., Ed.; Wiley: Chichester, 1998; Vol. 1, CHARMM: The Energy Function and Its Parameterization with an Overview of the Program, pp 271−277.

(19) Brunger, A. T.; Paul, D.; Adams, G.; Clore, M.; DeLano, W. L.; Gros, P.; Grosse-Kunstleve, R. W.; Jiang, J. S.; Kuszewski, J.; Nilges, M.; Pannu, N. S.; Read, R. J.; Rice, L. M.; Simonson, T.; Warren, G. L. Crystallography & NMR System: A New Software Suite for Macromolecular Structure Determination. *Acta Crystallogr.* **1998**, *D54*, 905−921.

(20) van Rossum, G.; Drake, F. L., Jr. *Python Reference Manual*; Python Software Foundation: 2005.

(21) Hinsen, K. The molecular modeling toolkit: A new approach to molecular simulations. *J. Comput. Chem.* **2000**, *21*, 79−85.

(22) Richardson, D. C.; Richardson, J. S. The kinemage: a tool for scientific communication. *Protein Sci.* **1992**, *1*, 3−9.

(23) Harvey, S. C.; Gabb, H. A. Conformational Transitions Using Molecular Dynamics with Minimum Biasing. *Biopolymers* **1993**, *33*, 1167−1172.

(24) Waugh, A.; Gendron, P.; Altman, R.; Brown, J. W.; Case, D.; Gautheret, D.; Harvey, S. C.; Leontis, N.; Westbrook, J.; Westhof, E.; Zuker, M.; Major, F. RNAML: a standard syntax for exchanging RNA information. *RNA* **2002**, *8*, 707−717.

(25) Bauer, W. R.; Crick, F. H.; White, J. H. Supercoiled DNA. *Sci. Am.* **1980**, *243*, 100−113.

(26) Locker, C. R.; Fuller, S. D.; Harvey, S. C. DNA organization and thermodynamics during viral packing. Submitted for publication.

CT050323R