PROFASI: A Monte Carlo Simulation Package for Protein Folding and Aggregation*

ANDERS IRBÄCK, SANDIPAN MOHANTY[†]
Complex Systems Division, Department of Theoretical Physics, Lund University, Sölvegatan 14A, SE-223 62 Lund, Sweden

Received 21 December 2005; Accepted 7 February 2006
DOI 10.1002/jcc.20452
Published online in Wiley InterScience (www.interscience.wiley.com).

Abstract: We present a flexible and efficient program package written in C++, PROFASI, for simulating protein folding and aggregation. The systems are modeled using an all-atom description of the protein chains with only torsional degrees of freedom, and implicit water. The program package has a modular structure that makes the interaction potential easy to modify. The currently implemented potential is able to fold several peptides with about 20 residues, and has also been used to study aggregation and force-induced unfolding. The simulation methods implemented in PROFASI are Monte Carlo-based and include a semilocal move and simulated tempering. Adding new updates is easy. The code runs fast in both single- and multi-chain applications, as is illustrated by several examples.

© 2006 Wiley Periodicals, Inc. J Comput Chem 27: 1548–1555, 2006

Key words: protein folding; protein aggregation; Monte Carlo; all-atom model; C++

Introduction

With improved algorithms and faster computers, it is becoming computationally feasible to simulate how small proteins fold to their native states (for a review, see ref. 1). This is an exciting development, which will lead to a better understanding not only of protein folding, but also of protein aggregation and of the interaction of proteins with other molecules and materials. For atomic-level simulations of proteins, there exists a number of program packages that can be used, such as AMBER,[2] CHARMM,[3] GROMACS,[4] and SMMP.[5] The force fields implemented in these packages are typically quite detailed, with parameters that in many cases are estimated using microscopic arguments. We are currently exploring a different approach,[6–8] which starts from a simple ansatz for the interaction potential. The parameters of this potential are calibrated against data pertaining to folding properties of whole chains, rather than properties of groups of one or a few atoms. For this purpose, we consider a set of well-characterized sequences, which we study by high-statistics folding simulations. The idea is to successively refine the potential by studying more and more sequences, which will impose new constraints on the model.

The sequences that the current version of the model is able to fold have about 20 amino acids, and there are known examples of sequences of this size that the model fails to fold. One example is the tryptophan zipper β-hairpins,[9] which make β-hairpins in the model but with the wrong topology. The set of sequences that the model can fold includes both α-helical and β-sheet peptides. Furthermore, for these sequences, the model has been found to provide a good description not only of the folded structure, but also of the folded population and its temperature dependence.[8] Without changing any parameters, the model was also used to study the oligomerization of the seven-residue fragment $A\beta_{16–22}$ of the amyloid-β peptide associated with Alzheimer's disease, with very promising results.[10] Recently, the force-induced unfolding of ubiquitin, with 76 residues, was investigated, again using exactly the same model.[11]

Here we present a program package, PROFASI (PROtein Folding and Aggregation SImulator), that implements this model. To be able to take full advantage of the computationally convenient form of the model, we developed PROFASI from scratch rather than using some existing program package. Anticipating future refinements, PROFASI is built so that the interac-

*The PROFASI package is freely and immediately available to academic users upon signing a user agreement. It can be downloaded from http://www.thep.lu.se/complex/activities/profasi, where also more details on the code can be found, including a documentation of the different classes used.

[†]Present address: John von Neumann Institute for Computing, Forschungszentrum Jülich, D-52425 Jülich, Germany.

Correspondence to: A. Irbäck; e-mail: anders@thep.lu.se

Contract/grant sponsor: Swedish Research Council

tion potential can be easily modified. The modular structure of the code, which is written in C++, also facilitates the introduction of, for example, new capping groups. For the sake of optimization, the code assumes a definite representation of the system, with only torsional degrees of freedom, which cannot be easily replaced. An important design criterion behind PROFASI is that it should be able to handle both single chains and multi-chain systems in a flexible and efficient manner. Using PROFASI, a high-statistics study of the thermodynamics of a 20-residue peptide takes roughly a day on a cluster of ten 2.2 GHz processors. The corresponding time for a multi-chain system of 6 seven-residue peptides is roughly 3 days.

Model and Algorithms

The model implemented in PROFASI contains all atoms of the protein chains, including hydrogen atoms, but no explicit water molecules. The model assumes fixed bond lengths, bond angles, and peptide torsion angles (180°), so that each amino acid has the Ramachandran torsion angles ϕ and ψ and a number of side-chain torsion angles as its degrees of freedom.

The interaction potential

$$E = E_{\text{loc}} + E_{\text{ev}} + E_{\text{hb}} + E_{\text{hp}} \tag{1}$$

is composed of four terms. The term E_{loc} is a local backbone potential which has the form of an electrostatic interaction between adjacent peptide units along the chain. The other three terms are nonlocal in sequence. The excluded volume term E_{ev} is a $1/r^{12}$ repulsion between pairs of atoms. E_{hb} represents two kinds of hydrogen bonds: backbone–backbone bonds and bonds between charged side chains and the backbone. The last term E_{hp} represents an effective hydrophobic attraction between non-polar side chains. It is a simple pairwise additive potential based on the degree of contact between two nonpolar side chains. All the three terms nonlocal in sequence are short range and are evaluated using a cutoff. The precise form of the different interaction terms and the numerical values of all the geometry parameters held constant can be found elsewhere.[6,8]

The most time-consuming and therefore most carefully optimized part of the code is the calculation of E_{ev}, which is split into two parts. The contribution to E_{ev} from local pairs of atoms, connected by three covalent bonds, is calculated separately, through a straightforward summation over all such pairs. The contribution to E_{ev} from all the remaining pairs of atoms, which are less likely to be in close contact, is calculated by using the cell list method.[12]

To avoid introducing fixed boundaries, the chains are assumed to move in a periodic box. With these boundary conditions, it could happen that an atom interacts with another atom from the same chain but from a different periodic copy of the system. To avoid this situation, it is sufficient to make the box length slightly larger than the maximum end-to-end distance, since all the interactions are short ranged. PROFASI uses the same periodic box for single chains as well as multi-chain systems, although it is not needed in the single-chain case. The use of the periodic box slows down the single-chain simulations (by about 30% for the peptides we have studied, about 20 residues long), which we regard as an acceptable price for avoiding duplication of code.

The simulation algorithms available with PROFASI are based on Monte Carlo rather than molecular dynamics. For backbone degrees of freedom, two different updates are provided, one non-local and one semilocal. The full set of conformational updates currently implemented in PROFASI is as follows.

1. Updates of individual angles. Both backbone and side-chain torsion angles can be updated using simple Metropolis[13] single-variable steps. When applied to a backbone angle, this method tends to lead to a highly nonlocal deformation of the chain, which is likely to be rejected if the chain is compact. This simple update can, on the other hand, be a very powerful method for extended chains.[14,15]
2. Biased Gaussian steps.[16] It is desirable to also include a backbone update less drastic than the single-angle update. One possibility is to use a strictly local method like the concerted-rotation algorithm.[17] For long chains, such an update has the advantage of leaving a large part of the energy function unchanged. However, strictly local methods tend to be quite complex, and the proteins amenable to atomic-level simulations are in any case not very long. Instead, we have therefore chosen to use biased Gaussian steps, a semilocal method that is flexible and easy to implement. This method simultaneously turns n adjacent torsion angles along the backbone, where n typically is 7 or 8. A tentative new $\bar{\tau} = (\tau_1, \ldots, \tau_n)$, $\bar{\tau}'$, is generated with a bias toward local deformations. Specifically, using a conformation-dependent $n \times n$ matrix $\mathbf{G}$ such that $(\bar{\tau}' - \bar{\tau})^T \mathbf{G}(\bar{\tau}' - \bar{\tau}) \approx 0$ for changes corresponding to local deformations, $\bar{\tau}'$ is drawn from the Gaussian distribution

$$W(\bar{\tau} \to \bar{\tau}') = \frac{(\det \mathbf{A})^{1/2}}{\pi^3} \exp\left[-(\bar{\tau}' - \bar{\tau})^T \mathbf{A}(\bar{\tau}' - \bar{\tau})\right] \tag{2}$$

$$\mathbf{A} = \frac{a}{2}(\mathbf{1} + b\mathbf{G}) \tag{3}$$

where $\mathbf{1}$ denotes the $n \times n$ unit matrix, and a and b are tunable parameters. The parameter a controls the acceptance rate, whereas b sets the degree of bias toward local deformations. Typical values in our simulations are $a = 300(\text{rad})^{-2}$ and $b = (\text{rad}/\text{Å})^{-2}$. Finally, the new conformation $\bar{\tau}'$ is subject to an accept/reject step, with probability

$$P_{\text{acc}}(\bar{\tau} \to \bar{\tau}') = \min\left(1, \frac{W(\bar{\tau}' \to \bar{\tau})}{W(\bar{\tau} \to \bar{\tau}')} \exp[-(E' - E)/kT]\right) \tag{4}$$

for acceptance (k is Boltzmann's constant and T is the temperature). The factor $W(\bar{\tau}' \to \bar{\tau})/W(\bar{\tau} \to \bar{\tau}')$ is needed for detailed balance to be fulfilled, since $\mathbf{G}$ is conformation-dependent and W thereby asymmetric.
3. Rigid-body translations and rotations of whole chains. These updates are useful for multi-chain systems.

These updates serve different purposes and are meant to be used together. The optimal choice of relative frequencies for the updates is not obvious and depends on the system under study. In our simulations, we used a simple estimate based on the respective numbers of degrees of freedom to set the ratio between backbone and side-chain updates. Single-angle backbone updates were used more frequently than biased Gaussian steps at high temperature, whereas the roles were reversed at low temperature.

In addition to these updates for simulations in the canonical ensemble, simulated tempering[18–20] is also implemented in PROFASI. This method simulates an expanded ensemble defined by the partition function

$$Z = \sum_k e^{-g_k} Z(T_k), \tag{5}$$

where the g_k's are tunable parameters, the T_k's are a set of predetermined temperatures, and $Z(T_k)$ denotes the canonical partition function at temperature T_k. The simulation involves jumps between the different temperatures T_k, which are controlled by a Metropolis accept/reject step. The idea is to make it easier for the system to escape from local free-energy minima by allowing visits to higher temperatures where barriers are lower. The simulation parameters g_k are determined by trial runs.[20] A method closely related to simulated tempering is the parallel-tempering or replica-exchange method.[21–23] In terms of computational efficiency, we expect simulated and parallel tempering to be similar,[24] and so the choice between these methods is largely a matter of taste. Parallel tempering has the advantage of not having any g_k parameters to be tuned. On the other hand, the determination of these parameters is often not difficult, and it provides useful information about the free energy of the system. Also, it might be easier to detect incomplete convergence in simulated tempering, where the temperature distribution is not enforced by hand.

Program Description

The program package PROFASI has been developed on standard desktop computers running a GNU/Linux operating system. Computations have been carried out both on single computers and Linux clusters.

In PROFASI, the key concepts of the model and the algorithms are represented as separate C++ classes. These classes can be divided into three major groups: building blocks (atoms, amino acids, polypeptide chains, etc.), energy terms, and Monte Carlo updates. The idea is to combine these elements in an application program. What the application program should look like depends on the problem at hand, but a few examples are provided with the package. These programs illustrate how the classes are to be used. The user adapts them to what he or she might be interested in. One such application program will be discussed in some detail below. The approach taken here, in which nouns rather than verbs are coded in a program, is native to the C++ programming language.

Building Blocks

A top level building block in PROFASI is the `Protein` class,* which provides an easy interface to define and initialize one chain. A chain object `prn` with the sequence Acetyl-Lys-Leu-Val-Phe-Phe-Ala-Glu-NH_2, for example, can be declared by writing

```
Protein prn("Acetyl","KLVFFAE","Amide");
```

The `Protein` class is, however, not meant to be used directly by the user. For that there is a `Population` class, which represents a population of one or more chains. To declare a system of 10 chains of the above mentioned kind, the user writes

```
Population p;
p.AddProtein("Acetyl","KLVFFAE","Amide",10);
p.Initialize();
```

Using several calls to the `AddProtein` function, a heterogeneous mixture of polypeptide chains can be created and simulated. The `Initialize()` function creates the `Protein` objects according to the specified sequence and assigns random initial values to all degrees of freedom. When required, other initial conditions can be set afterwards on a chain by chain basis. The `Chain(...)` member function of the `Population` class gives access to an individual `Protein`. It is then possible to invoke several different initializing member functions on the chain. For instance, the member function `helical_init()` forces the `Protein` to assume a helical backbone conformation, `trivial_init()` sets all internal degrees of freedom to zero, `randomize()` randomizes all internal degrees of freedom, and `ReadState(...)` reads in a previously saved state from a file. The position and orientation of one or more chains relative to the rest can, if desired, be randomized using the `RandomizeRelConf(...)` functions of the `Population`.

PROFASI contains classes representing structures at different levels of detail all the way from the population of polypeptide chains to simple atoms. Only a little familiarity with the building block classes is necessary to use the program. To be able to modify and make additions to the package however, some knowledge is required about the classes used to represent, for example, the atoms of the system. Each atom carries a unique integer label. The label for the C_α atom of the *j*th amino acid in the *i*th chain, for instance, can be found by writing

```
p.Chain(i)->AA(j)->Calpha().UniqueId();
```

Here `Chain(i)` returns a pointer to the *i*th chain in the population, `AA(j)` returns a pointer to an `Aminoacid` object representing the *j*th amino acid of that chain, and `Calpha()` returns a reference to an `Atom` object representing the C_α atom. Finally, `UniqueId()` returns the integer label of this atom.

The `Atom` class keeps information about the type of atom (hydrogen, carbon, nitrogen, oxygen, or sulfur) and the position, where the position is an object belonging to the `AtomCoordinates` class. The position of an `Atom` object `a` is accessed by writing `a.Pos()`. The usual algebraic operations on three-dimensional vectors are available for `AtomCoordinates`. For instance, if `a1` and `a2` are two `AtomCoordinates` objects, the expression `a1-a2` returns a three-vector representing the

*`Peptide` and `PolypeptideChain` are equivalent names for this class, through `typedefs`.

difference in spatial coordinates, whereas `a1*a2` returns the vector product. So, an `AtomCoordinates` object appears very much like a three-dimensional position vector, but when it is copied, only an integer label is copied. The main reason behind creating the `AtomCoordinates` class was to simplify the implementation of conformational updates, by making it possible to access atom coordinates more directly, without explicit reference to the chain structure. Having this possibility is useful because many of the operations involved are independent of the fact that the position vectors actually represent locations of atoms that constitute amino acids in a chain. By using the `AtomCoordinates` class, these operations can be implemented in a simple and efficient way. In addition, the introduction of the `AtomCoordinates` class makes it possible to reduce the number of copying operations on the three double-precision numbers representing the position in space.

Amino acids are represented using the base class `AminoAcid` for common properties, and derived classes for individual properties of the different amino acids. There is one such derived class for each amino acid type. The somewhat abstract concept represented by the `Node` class simplifies the structure of these amino acid classes. It represents the different types of junction of bonds that occur in protein structures, like tetrahedral and trigonal junctions. A `Node` object is responsible for finding the appropriate positions in three dimensions for all atoms attached to its out-going bonds. The `TetrahedralGroup` class is a derived class of the `Node`, which has one in-coming bond and three out-going bonds with similar properties. Another example is the `ATetGroup` (asymmetric tetrahedral group), which is similar to the `TetrahedralGroup`, except that the out-going bonds can have different bond lengths and bond angles. Yet another example is the `PhenylGroup`, which is responsible for placing a phenyl ring with attached hydrogens in space, given the position of the in-coming bond and the orientation of the ring about that bond. With the computation of coordinates relegated to the various `Node` objects, each amino acid class declaration becomes a simple specification of the different elementary structures in its side chain.

Energy Terms

The different terms of the interaction potential are coded into different classes, each inheriting from a base class called `Energy`. This base class has properties that are necessary for every energy function so that they can be used with the rest of PROFASI. Such properties are declared as virtual functions, and an individual energy class should override these functions. For instance, each energy class must define a way to `evaluate()` the term in question and calculate the change `deltaE(...)` for a given conformational update. The member function `evaluate()` calculates the energy term by summing all possible contributions, whereas `deltaE(...)` calculates the change in energy that would result if a proposed update is accepted. It could be calculated by using `evaluate()` for the new set of coordinates and taking the difference from a stored value corresponding to the state before the update. However, all updates in PROFASI leave large parts of the system unchanged, which makes it much more economical to determine `deltaE(...)` by recalculating only the interactions of moved atoms (with all other atoms, moved or fixed). Optimization of the energy calculations based on specific properties of the different conformational updates is central to the efficiency of the resulting code.

Optimization of a given energy term is a matter of implementation, which may involve any local tricks and stunts, without affecting any other modules. For this to work, all modules that have to deal with energy terms, should do so only through the interface provided by the base class. So long as an energy class inherits from the base `Energy` class, and overrides its virtual functions, it can be used with the other classes of PROFASI.

Six energy classes are provided to represent the energy terms in eq. (1). The classes `Bias` and `Hydrophobicity` represent the terms E_{loc} and E_{hp}, respectively. The classes `HBMM` and `HBMS`, which both inherit from a `HydrogenBond` class, represent the main chain–main chain and the main chain–side chain hydrogen bonds, respectively. `HydrogenBond` in turn inherits from `Energy`. Similarly, the classes `ExVol` and `LocExVol` represent two kinds of contributions to the excluded-volume term E_{ev}. `LocExVol` represents the contribution from atom pairs connected by three covalent bonds. This local contribution is very important to ensure proper torsion angle distributions, but is also computationally simple. When one torsion angle is turned, at most nine local pairs can be affected. Information about which pairs are affected for different updates can be calculated during the initialization and stored, which reduces the number of calculations done during a Monte Carlo step.

Calculation of the rest of the excluded-volume energy is the most expensive part of our model. As mentioned earlier, we use cell lists[12] to speed up this calculation. The environment of our chains, the periodic box, is divided into an integral number of cells along each axis. The dimension of a cell is at least as large as the cutoff used for E_{ev}, so that in any given configuration, the contribution to E_{ev} from atom pairs not located in the same cell or in neighboring cells is guaranteed to be zero. This greatly reduces the number of calculations required to compute E_{ev}. During a simulation, when a Monte Carlo update is proposed, only the contribution of the cells containing affected atoms and their neighbors is recalculated.

The other energy terms in our potential are inherently much less expensive than the two excluded volume contributions. This is because they involve a limited number of atoms, and simple bookkeeping helps in avoiding calculation of unchanged contributions during a Monte Carlo step.

We expect that the implementation of energy classes will change in the future to refine the potential and perhaps also to further speed up the calculations. Care has been taken in the design of PROFASI, so that this can be achieved without having to introduce major changes elsewhere.

Conformational Updates

The conformational updates described in Model and Algortihms are implemented using five classes, each inheriting from the base class `Update`. The `Update` class represents basic properties, like for instance, the first and last atoms affected by an update. The properties of the base class serve as the interface of any derived update class with other modules in PROFASI. The five derived update classes are as follows:

- `Rotation`, which performs a rigid-body rotation of one whole chain.
- `Translation`, which performs a rigid-body translation of one whole chain.
- `Pivot`, which turns a single backbone angle.
- `Rot`, which turns one side-chain angle.
- `BGS`, which performs a biased Gaussian step (see Model and Algorithms).

The first four of these updates divide the system into two rigid parts that move relative to each other, whereas `BGS` divides the system into three parts: the unmoved atoms, a range of atoms in which many atoms move relative to each other, and a third part moving rigidly relative to the other two. The update classes provide information to identify these different parts. This information is used to optimize the energy calculations.

Algorithms

The role of the update classes mentioned above is only to propose new configurations. To actually perform the Markov chain evolution of the system, the `MC` class is used. During initialization, an `MC` object acquires information about different conformational updates, energy functions, and the population of polypeptide chains. The `Step()` member function chooses an update with given probabilities and performs it. It accumulates `deltaE` values from the different energy classes, and accepts or rejects the update with a Metropolis-like probability depending on the energy change and the temperature. There is a provision to incorporate an extra multiplicative factor in the acceptance probability of an update, like the $W(\bar{\tau}' \rightarrow \bar{\tau})/W(\bar{\tau} \rightarrow \bar{\tau}')$ needed in biased Gaussian steps [see eq. (4)].

The `MC` class as such is for constant-temperature simulations. With methods for such simulations available, little extra effort is required to implement simulated annealing[25] for stochastic optimization and generalized-ensemble methods such as simulated[18,19] and parallel[21,22] tempering. In the current version of PROFASI, a `SimAnneal` class for simulated annealing and a `SimTemp` class for simulated tempering are available, both implemented as derived classes of the `MC` class. In simulated tempering, the user has to specify the number and range of temperatures to be used, the simulation parameters g_k [see eq. (5)], and the number of conformational updates between two temperature updates. Given this input, the `SimTemp` class changes the temperature appropriately.

Observables

PROFASI provides a large number of interface functions that can be used to monitor various observables. Documentation for different classes contains information about such interface functions provided by them. Where to look for an interface function

Table 1. Order of Side-Chain Atoms in the Program.

Amino acid	Side-chain atoms
G	H_α^2
A	C_β,H_β^1,H_β^2,H_β^3
V	C_β,H_β,$C_{\gamma1}$,$C_{\gamma2}$,$H_{\gamma1}^1$,$H_{\gamma1}^2$,$H_{\gamma1}^3$,$H_{\gamma2}^1$,$H_{\gamma2}^2$,$H_{\gamma2}^3$
L	C_β,H_β^1,H_β^2,C_γ,H_γ,$C_{\delta1}$,$C_{\delta2}$,$H_{\delta1}^1$,$H_{\delta1}^2$,$H_{\delta1}^3$,$H_{\delta2}^1$,$H_{\delta2}^2$,$H_{\delta2}^3$
I	C_β,H_β,$C_{\gamma1}$,$C_{\gamma2}$,$H_{\gamma2}^1$,$H_{\gamma2}^2$,$H_{\gamma2}^3$,$H_{\gamma1}^1$,$H_{\gamma1}^2$,$C_{\delta1}$,$H_{\delta1}^1$,$H_{\delta1}^2$,$H_{\delta1}^3$
S	C_β,H_β^1 ,H_β^2 ,O_γ,H_γ
T	C_β,H_β,$O_{\gamma1}$,$C_{\gamma2}$,$H_{\gamma2}^1$,$H_{\gamma2}^2$,$H_{\gamma2}^3$,$H_{\gamma1}$
C	C_β,${H_\beta}^1$,H_β^2 ,S_γ,H_γ
M	C_β,${H_\beta}^1$,H_β^2,C_γ,H_γ^1 ,H_γ^2,S_δ,C_ε,H_ε^1 ,H_ε^2 ,H_ε^3
P	C_β,H_β^1,H_β^2,C_γ,H_γ^1,H_γ^2,C_δ,H_δ^1,H_δ^2
D	C_β,H_β^1,H_β^2,C_γ,$O_{\delta1}$,$O_{\delta2}$
N	C_β,H_β^1,H_β^2,C_γ,$O_{\delta1}$,$N_{\delta2}$,$H_{\delta2}^1$$H_{\delta2}^2$
E	C_β,H_β^1,H_β^2,C_γ,H_γ^1,H_γ^2,C_δ,$O_{\varepsilon1}$,$O_{\varepsilon2}$
Q	C_β,H_β^1,H_β^2,C_γ,H_γ^1,H_γ^2,C_δ,$O_{\varepsilon1}$,$N_{\varepsilon2}$,$H_{\varepsilon2}^1$,$H_{\varepsilon2}^2$
K	C_β,H_β^1,H_β^2,C_γ,H_γ^1,H_γ^2,C_δ,H_δ^1,H_δ^2,C_ε,H_ε^1,H_ε^2,N_ζ,H_ζ^1,H_ζ^2,H_ζ^3
R	C_β,H_β^1,H_β^2,C_γ,H_γ^1,H_γ^2,C_δ,H_δ^1,H_δ^2,N_ε,H_ε,C_ζ,$N_{\eta1}$,$H_{\eta1}^1$,$H_{\eta1}^2$,$N_{\eta2}$, $H_{\eta2}^1$,$H_{\eta2}^2$
H	C_β,H_β^1,H_β^2,C_γ,$C_{\delta2}$,$N_{\varepsilon2}$,$C_{\varepsilon1}$,$N_{\delta1}$,$H_{\delta2}$,$H_{\varepsilon1}$,$H_{\delta1}$
F	C_β,H_β^1,H_β^2,C_γ,$C_{\delta1}$,$C_{\varepsilon1}$,C_ζ,$C_{\varepsilon2}$,$C_{\delta2}$,$H_{\delta1}$,$H_{\varepsilon1}$,H_ζ,$H_{\varepsilon2}$,$H_{\delta2}$
Y	C_β,H_β^1,H_β^2,C_γ,$C_{\delta1}$,$C_{\varepsilon1}$,C_ζ,$C_{\varepsilon2}$,$C_{\delta2}$,$H_{\delta1}$,$H_{\varepsilon1}$,O_η,$H_{\varepsilon2}$,$H_{\delta2}$,H_η
W	C_β,H_β^1,H_β^2,C_γ,$C_{\delta1}$,$N_{\varepsilon1}$,$C_{\varepsilon2}$,$C_{\delta2}$,$C_{\varepsilon3}$,$C_{\zeta3}$,$C_{\eta2}$,$C_{\zeta2}$,$H_{\delta1}$,$H_{\varepsilon1}$,$H_{\varepsilon3}$,$H_{\zeta3}$, $H_{\eta2}$,$H_{\zeta2}$

For an `AminoAcid` object `a`, the function `a.sidechain_atom(i)` returns the *i*th side-chain atom, starting from C_β (H_α^2 for Gly). This table shows the correspondence of the side chain atoms with the integer *i*. The heavy atoms are labeled as in the Protein Data Bank (PDB). Hydrogen atoms are labeled according to which heavy atom they are attached to.

depends on the character of the observable. For instance, the end-to-end distance is a description of a polypeptide chain rather than of an amino acid or an atom. So, to access the end-to-end distance, the user should look for an appropriate function in the `Protein` class.

Frequently used and sophisticated observables such as the root-mean-square deviation (RMSD) from a reference structure are implemented as individual classes, for easy use. The RMSD determination involves a minimization with respect to all possible rotations and translations. The `ProteinRMSD` class performs this calculation using closed-form algebraic expressions based on singular-value decomposition, without involving any search. Monitoring of native contacts and native hydrogen bonds can be handled conveniently using the `ContactMap` class.

When adding new observables, it is important to have easy access to information about individual atoms, particular torsion angles, etc. One way of accessing a certain atom was mentioned under Building Blocks, but the `Protein` and `AminoAcid` classes also provide many other ways to address individual atoms. For instance, the *i*th backbone atom of `Protein p` can be accessed by writing `p.backbone_atom(i)`, and the *i*th side-chain atom of `AminoAcid a` can be accessed by writing `a.sidechain_atom(i)`. Table1 summarizes the order in which side-chain atoms are returned for the different amino acids.

To simplify the handling of repeated tasks relating to observables, two structures are provided, the `Observable` base class and the `ObsHandler` class. The `Observable` class represents anything that has a `Name()` and a `Value()` which returns a `double`. All energy classes and the `ProteinRMSD` class discussed above inherit from this base class. The `ObsHandler` class performs tasks such as keeping a record of all observables, writing a specified set of them into the run-time history file (see Getting Started), and averaging and maintaining histograms of the subset of all observables on which such operation is desired. In addition, it can optionally save the state of all the histograms at regular intervals, so that the program can be restarted with the histograms resuming their sampling from the end of one run. A new observable that calculates a property of the system and returns a double precision number can be passed to the `ObsHandler` object `H` by writing

```
H.track(&myobs, ''rt avg his'');
```

The observable handler `H` then tracks this observable with the given options, that is, it puts its value to the run-time history file "rt", prints its average at different temperatures in the "averages" file, and maintains a histogram of this observable and saves it regularly. Each `Observable` also returns an estimate for its range, which is used by the `ObsHandler` object to automatically set the range of the corresponding histograms. It is possible to manually specify the number of bins and the range of the histograms for one observable by writing

```
H.setHist(obsname,nbins,xmin,xmax);
```

The column of the "rt" file corresponding to one observable can be found in the "rtkey" file, created by the `ObsHandler` object.

Performance

To give an idea of the speed of the program, we show in Table2 the time required to perform one billion elementary Monte Carlo steps for a few different single- and multi-chain systems. The runs were carried out on a 64-bit AMD Opteron 2.2-GHz processor, using the gcc 3.3.3 compiler. All the systems were studied using simulated tempering, with the same choice of temperatures (eight temperatures, ranging from 275 to 369 K). The relative frequencies for the different updates were the same in all runs, except that the multi-chain runs contained a small fraction of rigid-body moves which were not used for single chains.

The five single-chain runs in Table 2 contain between 131 and 304 atoms. The computer time per Monte Carlo step shows a roughly linear increase with the number of atoms for these chains. Without the cell list method, this scaling of time with system size would be quadratic for large systems. For the systems of 1, 3, 6, and 10 Aβ_{16-22} peptides, the computer time per Monte Carlo step increases slower than linearly with the number of atoms. This behavior is possible because all the updates act on individual chains, which means that the fraction of atoms moved by an update is never larger than the inverse number of chains. Care has to be taken so that the program takes advantage of this fact. Finally, we note in Table 2 that the computer time per Monte Carlo step per atom is significantly smaller for the system of 10 GNNQQNY peptides than for the system of 10 Aβ_{16-22} peptides. This difference reflects a difference in physical behavior. The GNNQQNY simulation is faster because the propensity to aggregate is lower for this system than for the Aβ_{16-22} system; an update typically takes longer if the atoms are densely packed.

Table 2. Time Taken Per One Billion Elementary Monte Carlo Steps for Some Representative Systems.

Peptide	Chains	Amino acids	Atoms	Hours/billion MC steps
Trp cage	1	20	304	18
GB1p	1	16	247	15
F_s	1	21	266	17
C-peptide	1	13	192	12
Aβ_{16-22}	1	7	131	10
Aβ_{16-22}	3	21	393	25
Aβ_{16-22}	6	42	786	36
Aβ_{16-22}	10	70	1310	62
GNNQQNY	10	70	1070	35

Trp cage (PDB code 1L2Y) is a compact helical "miniprotein"[26]; GB1p is the 41–56-residue fragment from the protein G B1 domain, which makes a β-hairpin[27]; F_s is a designed Ala-based α-helix[28]; the C-peptide is an α-helix that was studied together with GB1p in a recent test of different force fields[29]; Aβ_{16-22} is a fibril-forming seven-residue fragment[30] of the Alzheimer's amyloid-β peptide; GNNQQNY is a fibril-forming seven-residue fragment[31] from the amyloid yeast protein Sup35. Acetyl and amide capping groups were included in the simulations of Aβ_{16-22} and the C-peptide. For the F_s peptide we used succinyl and NH_2 as N- and C-terminal capping groups.

To obtain the computational cost, the time per Monte Carlo step has to be multiplied by the number of steps needed to faithfully sample the conformation space. Our experience is that a run with 1–2 billion steps is sufficient to get a good picture of the thermodynamic behavior of single chains with about 20 amino acids, and that reasonable estimates of statistical errors can be obtained from 10 such runs. A run of this length is, by contrast, somewhat short for the two 10-chain systems in Table 2. To obtain reliable results for these systems, including error estimates, we recommend using 10 runs with at least 5 billion steps. Each such simulation requires 1–2 weeks on the computer that was used for the runs in Table 2.

Installation

Installation of PROFASI is easy: download, unpack, and run make. Small adjustments might have to be made in the Makefile to set the path to one particular compiler, and to specify compiler- or machine-specific optimization options. The package comes with a Makefile assuming gcc as the compiler. But the program has been tested with the Intel compiler and Portland Group's compiler suite. PROFASI is an independent package, and does not depend on any other scientific package to be installed. Only standard C++ libraries are needed.

Getting Started

PROFASI includes a basic set of application programs which can be immediately used without changing anything inside the code. A convenient interface is provided through a configuration file called "settings.prf", in which many program variables can be set. This file is read in by many application programs, and it is normally here that one puts information about the amino acid sequence, the size of the periodic box, the number of temperatures and the temperature range, the length of the simulation, and many other variables. All these variables can be changed without the need to recompile the program. A list of variables that can be set and the syntax to be used can be found in the documentation. It is sufficient to have just this settings file in the directory where the program is being run. For instance, if you wish to start a simulation of five chains of the peptide Acetyl-Lys-Leu-Val-Phe-Phe-Ala-Glu-NH_2 in a periodic box of size 50 Å, the simplest settings file should have the following lines:

```
add_chain 5 < Acetyl KLVFFAE Amide >
box_length 50
```

With this settings file in a directory, one runs the program "simtemp.ex" (located in the "app" directory; run it either with its full path or through a symbolic link) to perform a simple simulation on a single processor. The program performs a simulation of the specified five-chain system using simulated tempering, with default values for all unspecified parameters, and writes information to different files in a directory called "n0". The calculations can also be performed using the executable simtemp.mex, which is meant to be used in the parallel mode through mpirun (from MPICH[32]). This program can be used to collect statistics by running independent simulations of the same system on different nodes. Running it on 10 nodes creates 10 directories "n0", ... , "n9" with the output from each process written in a separate directory.

Among other things, these programs produce a run-time history file called "rt" in their output directories (e.g., "n0"). This file records the state of the system at regular intervals. The first column is the number of Monte Carlo cycles (one cycle corresponds to 100 elementary Monte Carlo steps by default), and the other columns record the temperature, energy terms, and other properties. A key to what the different columns mean is found in the "rtkey" file. The "averages" file generated by each run stores mean and standard deviation for various quantities, like energy and secondary-structure contents, at the different temperatures.

The settings file above, although functional, is only meant to be illustrative. It omits a lot of variables which causes them to fall back to some defaults. In a typical application, the user would control many more aspects through this file.

An Example Application Program

The interface provided through the settings file is powerful enough to do a wide range of simulations without changing any code. For example, all the folding and aggregation simulations mentioned in the Introduction[8,10] can be done by only changing the settings file and running the same program. Changing only the settings file is, on the other hand, insufficient if, for instance, a new measurement or some other statistics is desired. The user then needs to write an application program like those in the "app" directory. In the application programs, an `Interface` class is implemented with the following three important member functions.

- `Run(int rank, int nruns)`, which is called by a very small `main()` function or by the MPICH interface. The `rank` and `nruns` arguments can be used to redirect output from one process to a unique directory and to ensure different random number seeds in parallel runs. This function is intended to do what the `main()` function normally does.
- `Init(...)`, which is called by the `Run()` function, and carries out all required initializations.
- Optionally a `ParseCommands(const char *)` function, which is called by the `Init()` function. This function takes a file name as the argument (normally "settings.prf") and sets up different variables of the program based on the contents of that file.

```
class Interface : public InterfaceBase {
public:
    int Run(int rnk, int nrn);
    int Init();
private:
    ABrandNewUpdate newupdate;
    ANewEnergyTerm newen;
};
int Interface::Init() {
    boxSize(35);
    p.AddProtein("Acetyl","KLVFFAE","Amide",3);
    skipUpdate("Pivot"); //if you don't like Pivot
    useUpdate(&newupdate); //and want to use your own update
    useEnergy(&newen); //add your new energy term
    H.track(&newen,"rt avg"); //or ''rt avg his'', for newen
    prepare_mc();
    auto_track_obs(); //adds default set of observables
    H.setHist("Etot",50,0.0,110.0); //hist. bins and range
    H.initialize(NTMP,"temperature");
}
int Interface::Run(int rnk, int nrn) {
    if (Init()==0) return 1;
    for (unsigned long i=0;i<100000;++i) {
        mc.RunCycle();
        H.sample(i,mc.CurTempIndex());
        if ((i+1)%1000==0) {
            H.writeRTSnapshot();
            SavePDB("tmp_pdb");
        }
        mc.SwitchTemp();
    }
    H.writeAverages();
    H.writeHistograms();
}
```

Figure 1. An example application program for PROFASI. Application programs are written as `Interface` classes. The `main()` function merely transfers control to the `Run()` member function of the `Interface` class, after performing possible system-specific initialization steps not directly related to the model, like for instance, after determining the rank and number of runs in an MPI run. The code shown here is part of the program "example.cc" in the "app" directory of the package.

An example of an application program can be found in Figure 1. This program simulates three chains of the Alzheimer's A$\beta_{16\text{–}22}$ peptide with acetyl and NH_2 groups as N and C terminal capping groups, respectively. The `Interface` class inherits from `InterfaceBase`, in which we have defined the default behavior of the entire model. It has a population object `p` which is empty, a `mc` object `mc`, and instances of the energy and update classes described in this article. By default all these energies and updates are used, and their initialization is handled through this base class. The base class knows nothing about observables, and is meant to be a pure representation of the Monte Carlo engine. The `Interface`, while inheriting its properties, augments it with an `ObsHandler` object, which is set to track the values of all default energy functions, as well as secondary structure of the chains, and RMSD when appropriate. For single-chain systems the rigid-body updates are switched off, unless they are switched on explicitly by the user.

The `Interface` class provides a purpose to the base class. It fills the population with a certain number of `Proteins`. It has a `Run` function in which the user specifies what is to be done with the population. Here one can also make changes such as skipping default updates or energies and adding new ones, adding new observables, etc., as illustrated in Figure 1. With such an `Interface` class defined, one only needs a small `main()` function to instantiate an interface object and transfer control to its `Run()` member function.

Inside the `Run()` function, the two functions `RunCycle()` and `SwitchTemp()` are called. `RunCycle()` performs a predefined number of conformational updates (default is 100), whereas `SwitchTemp()` is responsible for the temperature update.

The `SimTemp` object looks for the parameters g_k of eq. (5) in a file called "gpars.in". When the file is not found, the g_k's are set to zero, which normally leads to uneven probabilities for different temperatures. At any point of a simulation, the "gpars.out" file contains recommendations for these g_k parameters based on the statistics collected so far, which can be used as input values for a new run. Several such iterations may be required before the calculated g_k parameters are reasonably close to the input values. Before this is achieved, the different temperatures are not uniformly scanned, and the evolution of the system is likely to be slow.

Summary

The program package PROFASI implements a model that contains all atoms of the protein chains, but which has a force field that is simpler than the typical all-atom force field. PROFASI provides a C++ platform for folding and aggregation studies of this model, but might also be useful for studies based on other models with a similar structure. Special attention has been devoted to developing a code that runs efficiently in both single- and multi-chain applications.

Acknowledgments

The authors thank Giorgio Favrin, Simon Mitternacht, Björn Samuelsson, Fredrik Sjunnesson, and Stefan Wallin for valuable discussions at different stages of this work.

References

1. Gnanakaran, S.; Nymeyer, H.; Portman, J.; Sanbonmatsu, K. Y.; García, A. E. Curr Opin Struct Biol 2003, 13, 168.
2. Pearlman, D. A.; Case, D. A.; Caldwell, J. W.; Ross, W. R.; Cheatham, T. E., III; DeBolt, S.; Ferguson, D.; Seibel, G.; Kollman, P. Comput Phys Commun 1995, 91, 1.
3. Brooks, B. R.; Bruccoleri, R. E.; Olafson, B. D.; States, D. J.; Swaminathan, S.; Karplus, M. J Comput Chem 1983, 4, 187.
4. Lindahl, E.; Hess, B.; van der Spoel, D. J Mol Model 2001, 7, 306.
5. Eisenmenger, F.; Hansmann, U. H. E.; Hayryan, S.; Hu, C. K. Comput Phys Commun 2001, 138, 192.
6. Irbäck, A.; Samuelsson, B.; Sjunnesson, F.; Wallin, S. Biophys J 2003, 85, 1466.
7. Irbäck, A.; Sjunnesson, F. Proteins 2004, 56, 110.
8. Irbäck, A.; Mohanty, S. Biophys J 2005, 88, 1560.
9. Cochran, A. G.; Skelton, N. J.; Starovasnik, M. A. Proc Natl Acad Sci USA 2001, 98, 5578.
10. Favrin, G.; Irbäck, A.; Mohanty, S. Biophys J 2004, 87, 3657.
11. Irbäck, A.; Mitternacht, S.; Mohanty, S. Proc Natl Acad Sci USA 2005, 102, 13427.
12. Hockney, R. W.; Eastwood, J. W. Computer Simulations Using Particles; McGraw-Hill: New York, 1981.
13. Metropolis, N. A.; Rosenblut, A. W.; Rosenblut, M. N.; Teller, A.; Teller, E. J Chem Phys 1953, 21, 1087.
14. Lal, M. Mol Phys 1969, 17, 57.
15. Madras, N.; Sokal, A. D. J Stat Phys 1988, 50, 109.
16. Favrin, G.; Irbäck, A.; Sjunnesson, F. J Chem Phys 2001, 114, 8154.
17. Dodd, L. R.; Boone, T. D.; Theodorou, D. N. Mol Phys 1993, 78, 961.
18. Lyubartsev, A. P.; Martsinovski, A. A.; Shevkunov, S. V.; Vorontsov-Velyaminov, P. N. J Chem Phys 1992, 96, 1776.
19. Marinari, E.; Parisi, G. Europhys Lett 1992, 19, 451.
20. Irbäck, A.; Potthast, F. J Chem Phys 1995, 103, 10298.
21. Swendsen, R. H.; Wang, J.-S. Phys Rev Lett 1986, 57, 2607.
22. Hukushima, K.; Nemoto, K. J Phys Soc Jpn 1996, 65, 1604.
23. Hansmann, U. H. E. Chem Phys Lett 1997, 281, 140.
24. Irbäck, A.; Sandelin, E. J Chem Phys 1999, 110, 12256.
25. Kirkpatrick, S.; Gelatt, C. D.; Vecchi, M. P. Science 1983, 220, 671.
26. Neidigh, J. W.; Fesinmeyer, R. M.; Andersen, N. H. Nat Struct Biol 2002, 9, 425.
27. Blanco, F. J.; Rivas, G.; Serrano, L. Nat Struct Biol 1994, 1, 584.
28. Lockhart, D. J.; Kim, P. S. Science 1993, 260, 198.
29. Yoda, T.; Sugita, Y.; Okamoto, Y. Chem Phys 2004, 307, 269.
30. Balbach, J. J.; Ishii, Y.; Antzutkin, O. N.; Leapman, R. D.; Rizzo, N. W.; Dyda, F.; Reed, J.; Tycko, R. Biochemistry 2000, 39, 13748.
31. Nelson, R.; Sawaya, M. R.; Balbirnie, M.; Madsen, A. Ø.; Riekel, C.; Grothe, R.; Eisenberg, D. Nature 2005, 435, 773.
32. Gropp, W.; Lusk, E.; Doss, N.; Skjellum, A. Parallel Comput 1996, 22, 789.