PROTOMOL Version 2.0.3 - How-To Guide

April 26, 2006

Abstract

We present a How-To manual for the Molecular Dynamics (MD) simulation engine, PROTOMOL . This manual is intended to function as a guide to extending the framework to encapsulate new models, *i.e.* for phase space propagation, force calculations, *etc.* It is thus a middle tier between simply running the software for which the User Guide is sufficient, and developing the software where the Programmers Guide is helpful.

PROTOMOL makes extensive use of the Factory [3] design pattern for framework extensions, which allows most extensions to be encapsulated within a single class addition. We explain how these simple extensions should be designed, for the purposes of several different types of features.

Acknowledgements

- Special thanks to the Center For Applied Mathematics at the University of Notre Dame for helping to fund researchers on the project.
- Parts of the development were performed at the Norwegian super-computing facilities in Bergen through a Norges Forskningsråd grant.
- Research was partially funded by an Arthur J. Schmitt Fellowship.

Contents

Chapter 1

Introduction

Molecular Dynamics (MD) describes a molecular system as a function of time based on integration of equations of motion and interacting forces. PROTOMOL employs many techniques for attacking the most expensive part of MD simulations in the nonbonded force calculations, including implementation of fast algorithms, library optimizations and parallelism. However, PROTOMOL has also been carefully designed to ensure extensibility of the framework for new model encapsulation. The framework makes use of object-oriented programming and design patterns to accomplish this. As a result, many extensions can be encapsulated by the addition of a single class and minor one-line modifications to other source files in the framework.

There are several extensions that could be made to an MD framework. We describe how to accomplish four useful types of extensions. These deal with: (1) adding new *integrators*, or methods for propagating a molecular system with time, (2) adding new *forces*, which should be calculated within an integrator to propagate the system and can be bonded or nonbonded, (3) adding new integrator *modifiers* to be invoked at certain points in the integration scheme, (4) adding new *outputs*, and (4) adding new topologies or structures. We also explain how to reference each of these extensions in the PROTOMOL configuration file, once the framework has been compiled and linked with the new extensions.

Chapter 2

Adding a New Integrator

The responsibility of MD integrators is to *propagate* a molecular system with time. That is, given a set of atomic position and velocity vectors (P, V), applying an integrator Ψ results in an updated set (P', V'):

$$(\mathbf{p}', \mathbf{v}') = \Psi(\mathbf{p}, \mathbf{v})$$

Or more generally, we can refer to phase space as the collection of atomic positions and momenta $\Gamma = (\mathbf{p}, \mathbf{q})$ and use an integrator to move to a new phase space Γ'.

$$\Gamma' = \Psi(\Gamma)$$

The propagator function Ψ involves calculating all forces. There are several different types of forces which can be accounted for in a molecule, *i.e.* bond stretching, angle or torsion rotations, van der Waals, electrostatic, *etc.* Some of these, like stretching or rotations, are *bonded* forces because they occur among atoms which are connected by bonds. Others, such as van der Waals and electrostatic, are *nonbonded* forces. Once a force vector is calculated for all atoms, it can be used to update atomic velocities and positions by solving Newton's equations. For example, accelerations are obtainable through knowledge of atomic forces and masses, which can subsequently be used to determine velocities by multiplying by a timestep Δt. Positions can be updated through another multiplication of this timestep.

Integration of a molecular system can have multiple hierarchies. Not all intramolecular motion occurs at the same timescale. For example, bond motion is extremely fast, operating on the order of femtoseconds. However, significant performance can be saved by evaluating nonbonded forces less frequently. Since these forces change less significantly, there is only slight accuracy loss by holding them constant while short-range forces are evaluated. This brought about the idea of multiple-timestepping (MTS), which involves multiple integrators. In a simple case, a 2-level MTS scheme will include 2 integrators: Ψ_1 to evaluate short-range forces and Ψ_2 for longer range. Ψ_2 would then calculate its own forces and update its own positions and velocities, then invoke Ψ_1 within a loop, at each iteration Ψ_1 would calculate short-range forces and update its positions and velocities. In this way, forces associated with Ψ_1 get evaluated more frequently than those associated with Ψ_2. This of course can be extended to encapsulate several integrators to form a chain or *hierarchy*.

Adding a new integrator to the PROTOMOL framework involves the following steps:

1. Change to the directory `framework/integrators`.

2. Create a new class, called <*integrator name*>`Integrator`.

3. Decide if this integrator should be an innermost integrator in a hierarchy. If yes, inherit from the class `STSIntegrator`. Otherwise, there are two options. You can inherit from `MTSIntegrator`, or if you want *mollification* [4] in your integrator which improves stability by eliminating slow force portions in the direction of already computed fast forces, inherit from the special integrator `MOLLYIntegrator`. If you are unfamiliar with mollification or do not want to use it, we recommend inheriting from `MTSIntegrator`.

4. Each integrator should define the following data members:

   - A `static const std::string keyword`. This should represent the configuration file reference for this new integrator, and should be initialized in the implementation file (`.cpp`), with a line like the following:
     `const string` <*integrator name*>`Integrator::keyword("`<*integrator name*>`");`
   - Any 'special' parameters unique for your integrator. Thus this excludes timestep, positions, velocities, *etc.*

5. Each integrator should define the following methods:

   - A default constructor. If you have no extra parameters, this can just be a call to the parent default constructor; either `STSIntegrator()`, `MTSIntegrator()` or `MOLLYIntegrator()`. Otherwise extra parameters should be initialized properly.
   - A constructor which accepts parameters. If this integrator is STS, the parameters are minimally a `timestep` of type `Real` and a `ForceGroup *overloadedForces`. If the integrator is MTS or MOLLY, change the first parameter to `int cycles` and in addition pass a parameter `StandardIntegrator *nextIntegrator`. This is if the integrator does not define any extra parameters. If there are no extra parameters the parent constructor can simply be called with these parameters and the function body can be empty. If there are extra parameters, they should be placed after parameter one (`timestep` or `cycles`), and be initialized accordingly within the function body.
   - A method `initialize()`, which accepts the following parameters: `GenericTopology *topo`, `Vector3DBlock *positions`, `Vector3DBlock *velocities`, and `ScalarStructure *energies`. This method should include any functionality that should be invoked once for this integrator before the simulation runs. Most often it can just contain the following two lines (substitute `MTS` and `MOLLY` for `STS` if necessary):
     `STSIntegrator::initialize(topo, positions, velocities, energies);`
     `initializeForces()`
     which tells the integrator to set the structural, phase space and energy information for the system and initialize all forces according to this information.
   - A method `run()` which accepts a single integer argument for the number of steps to run. This is the point where the algorithm is implemented. Positions, forces, and velocities are accessed through arrays `myPositions`, `myVelocities` and `myForces`; to access data for a specific atom, pass the atom number (*i.e.* `myPositions[0]` to access the first atom's positions). Timesteps can be accessed through the method `getTimestep()`, implicitly defined for all integrators. Mass for a particular atom can be accessed through the topology, *i.e.* `myTopo->atoms[i].scaledMass` for the first atom.
   - A method `doMake()` which creates the integrator object. For STS, this method accepts three parameters: a `string&`, a `const vector<Value>& values`, and a `ForceGroup*`

`fg`. For MTS or MOLLY, the method will take one more parameter `StandardIntegrator *nextIntegrator`. The method will return an `STSIntegrator*`, `MTSIntegrator*` or `MOLLYIntegrator*`. The `values` vector will hold all parameters, beginning with `timestep` (for STS) and `cycles` (for MTS or MOLLY), followed by extras. The function need only contain one `return` statement in the body, to return a new object by calling the constructor with appropriate parameters. For example, for an MTS integrator, with one extra parameter:
`return new` <*integrator name*>`Integrator(values[0], fg, nextIntegrator);`

- If extra parameters are necessary, a `const` method `getParameters()` should be defined, whose only formal parameter is a `vector<Parameter>& parameters`. This method defines all extra parameters, how they should be referenced in the configuration file, any constraints on their values, and any default values. The first line should be a call to `getParameters()` of the parent `STSIntegrator`, `MTSIntegrator`, or `MOLLYIntegrator`. To show by example, suppose you are constructing an STS integrator with one extra parameter for the temperature. Temperature should be non-negative, of course. In addition, you want its default value to be 300. The body of this function would appear as follows:

```
STSIntegrator::getParameters(parameters);
parameters.push_back(Parameter("temperature",
Value(myLangevinTemperature,ConstraintValueType::NotNegative()),
300));
```

- Auxiliary methods and variables can be defined as necessary to be used internally by the integrator, we recommend that all these be defined as `private`. However, the above are all of the requirements to add a new integrator to the framework. You may also find it useful to refer to any of the integrators within the directory as a template, just to get a feel for their structure. For a description of configuration file integrator definitions, please refer to the PROTOMOL User Guide.

6. Modify `Makefile.am` in the same directory, and add the new `.cpp` and `.h` files under the appropriate macros (it should be clear).

7. Modify `framework/factories/registerIntegratorExemplars.cpp` to include a call to the newly created integrator, as follows:
`IntegratorFactory::registerExemplar(new` <*Integrator name*>`Integrator());`

2.1 Example: Nosé Poincare

The Nosé Poincare propagation scheme [2] is a single-timestepping method based on a Hamiltonian which has been extended with an additional degree of freedom, or thermostat. This Hamiltonian is represented by the following equation:

$$H = (\sum_i \frac{p_i^2}{2m_i s^2} + V(q) + gkTlns - H_0)s, \tag{2.1}$$

where H_0 is a constant, chosen such that the Hamiltonian is zero at the initial conditions. s is an extended position variable (from Nosé dynamics) and π is its canonical momenta. p is the canonical momenta associated with current positions, such that $p = ps$ where p is the real momenta. T is temperature and k is Boltzmann's constant, and g is equal to the number of degrees of freedom in the system.

Bond *et al.* present a symplectic and time-reversible integration scheme which we implement in PROTOMOL as a new STS integrator. This is given by the following equations:

$$p_i^{n+1/2} = p_i^n - \frac{\Delta t}{2} s^n \frac{\delta}{\delta q_i} V(q^n) \tag{2.2}$$

$$\pi^{n+1/2} = \pi^n + \frac{\Delta t}{2} (\sum_i \frac{1}{m_i} (\frac{p_i^{n+1/2}}{s^n})^2 - gkT) - \frac{\Delta t}{2} \Delta H(q^n,\ p^{n+1/2}, s^n, \pi n + 1/2) \tag{2.3}$$

$$s^{n+1} = s^n + \frac{\Delta t}{2} (s^{n+1} + s^n \frac{\pi^{n+1/2}}{Q} \tag{2.4}$$

$$q_i^{n+1} = q_i^n + \frac{\Delta t}{2} (\frac{1}{s^{n+1}} + \frac{1}{s^n}) \frac{p_i^{n+1/2}}{m_i} \tag{2.5}$$

$$\pi^{n+1} = \pi^{n+1/2} + \frac{\Delta t}{2} (\sum_i \frac{1}{m_i} (\frac{p_i^{n+1/2}}{s^{n+1}})^2 - gkT) - \frac{\Delta t}{2} \Delta H(q^{n+1},\ p^{n+1/2}, s^{n+1}, \pi n + 1/2) \tag{2.6}$$

$$p_i^{n+1} = p_i^{n+1/2} - \frac{\Delta t}{2} s^{n+1} \frac{\delta}{\delta q_i} V(q^{n+1}) \tag{2.7}$$

Equation 2.3 is implicit an requires solving the following equation using the quadratic formula:

$$\frac{\Delta t}{4Q} (\pi^{n+1/2})^2 + \pi^{n+1/2} + C = 0, \tag{2.8}$$

where

$$C = \frac{\Delta t}{2} (gkT(1 + lns^n) - \sum_i \frac{(p_i^{n+1/2})^2}{2m_i (s^n)^2} + V(q^n) - H_0) - \pi^n \tag{2.9}$$

We solve these equations in the integrator we add to PROTOMOL . We allow the user to specify the temperature to run the method *T*, the canonical momenta associated with *s* in `myPi`, the canonical momenta associated with the current position in `myS` and the extended mass *Q* in `myQ`, which is typically made small to increase sample speed. In the header file, we include `STSIntegrator.h` since we are adding a new single-timestepping integration scheme. We define a new class `NosePoincareIntegrator` inheriting from `STSIntegrator` and declare two constructors, one a default which does nothing and the second accepting five user parameters (the four mentioned plus the timestep of the integrator, which is necessary for all STS schemes). We then declare `getParameters()` which will later populate an array with values of these five parameters from the PROTOMOL configuration file, and `getParameterSize()` which returns 5 since there are five parameters. The final two publicly defined methods we declare are `initialize()` and `run()`. All integrators must define these. We define a privately used method `doMake()` which accepts configuration file values from the user and returns a new `NosePoincareIntegrator` object, parameterized properly.

Since `myTimestep` is implicitly defined for all integrators which inherit from `STSIntegrator`, we only have to define data members for the other four user parameters. However, we also choose to define three other utility parameters `myPot`, `myH0` and `myGKT`. These will hold commonly used identifiers in the above equations, to avoid code repetition and decrease complexity. Finally, we publicly define a `keyword` data member which will hold the unique identifier for this integrator in the PROTOMOL configuration file. Before deciding on what this value should hold, all other integrators in the `framework/integrators` directory should be checked accordingly to ensure no keyword conflicts.

```
#ifndef NOSEPOINCAREINTEGRATOR_H
#define NOSEPOINCAREINTEGRATOR_H

#include "STSIntegrator.h"

namespace ProtoMol {

   class ScalarStructure;
   class ForceGroup;

   class NosePoincareIntegrator : public STSIntegrator {

      public:
         NosePoincareIntegrator() {}
         NosePoincareIntegrator(Real timestep,
                                Real temperature,
                                Real pi,
                                Real s,
                                Real q);
         virtual void getParameters(std::vector<Parameter>& parameters) const;
         virtual unsigned int getParameterSize() const{return 5;}
         virtual void initialize(GenericTopology *topo,
         Vector3DBlock   *positions,
         Vector3DBlock   *velocities,
         ScalarStructure *energies);
         virtual void run (int numTimesteps);

         static const std::string keyword;

      private:
         virtual STSIntegrator* doMake(std::string& errMsg,
                                       const std::vector<Value>& values,
                                       ForceGroup* fg)const;
         Real myNPTemperature;
         Real myPi;
         Real myS;
         Real myQ;

         Real myPot;
         Real myH0;
         Real myGKT;
   };

}
```

In the implementation file, we define all member methods of the class `NosePoincareIntegrator`. We first set the `keyword` data member, indicating that use of this propagation scheme should be indicated by `NosePoincare` in the integrator definition of the PROTOMOL configuration file. We then define the parameterized constructor and set all member variables that are user-defined from the configuration file.

We then define the `initialize()` routine. Usually integrators will need to initialize atomic forces as a first step, which can be accomplished by invoking the method `initializeForces()` once. We then set our utility variables. `myPot` is set to the initial potential energy, `myH0` to the initial total energy (potential + kinetic, satifsying Equation 2.1 above), and `myGKT` to the number of degrees of freedom times the Boltzmann constant times the user-specified temperature, subsequently representing the term *gkT* in the above equations.

In our implementation of `run()`, velocities are initially updated with a Leapfrog half-kick. We then obtain the value of *C* in the above equations and store it in `tempC`, then use it to update the canonical momenta of *s*, then update *s* itself. Then positions are updated, followed by another update of the canonical momenta of *s*, and a final Leapfrog half-kick on velocities.

The method `getParameters()` inserts the PROTOMOL configuration file keywords for the four user-defined parameters into the `parameters` array along with the corresponding data members of the class `NosePoincareIntegrator`. We constrain temperature to be non-negative. Subsequently, when these keywords are found by the PROTOMOL configuration file reader their values will be assigned to data members accordingly when a `NosePoincareIntegrator` object is created. This is done by `doMake()`, which returns a call to the parameterized constructor. User-defined parameters are contained in the formal parameter array `values`, in the order they were inserted in `getParameters()`. In an STS integrator, the timestep is implicitly the first parameter, followed by all user-defined parameters.

```
#include "NosePoincareIntegrator.h"

const string NosePoincareIntegrator::keyword( "NosePoincare" );

NosePoincareIntegrator::NosePoincareIntegrator(Real timestep,
                                                Real temperature,
                                                Real pi,
                                                Real s,
                                                Real q) :
                                               myTimestep(timestep),
                                               myNPTemperature(temperature),
                                               myPi(pi),
                                               myS(s),
                                               myQ(q) {}


void NosePoincareIntegrator::initialize(GenericTopology *topo,
                Vector3DBlock   *positions,
                Vector3DBlock   *velocities,
                ScalarStructure *energies)
{
   initializeForces();
   myPot = energies->potentialEnergy();
   myH0 = myPot + kineticEnergy(topo,velocities);
   myGKT = 3.0*(positions->size()-1.0)*Constant::BOLTZMANN*myNPTemperature;
}

void NosePoincareIntegrator::run(int numTimesteps)
```

```
{
   int step = 1;

   while (step < numTimesteps)
   {
      Real h = getTimestep() * Constant::INV_TIMEFACTOR;
      for (unsigned int i = 0; i < myVelocities->size(); i++)
         (*myVelocities)[i] += (*myForces)[i] * h / myTopo->atoms[i].scaledMass;
      Real tempC = 0.5*h*(myGKT*(1.0 + log(myS))
                     - kineticEnergy(myTopo,myVelocities)+myPot-myH0)-myPi;
      myPi = -2*tempC / (1.0 + sqrt(1.0 - tempC*h/myQ));
      Real mySold = myS;
      Real tempF = 0.5*h*myPi/myQ;
      myS *= (1.0 + tempF)/(1.0 - tempF);
      for (unsigned int i = 0; i < myPositions->size(); i++)
         (*myPositions)[i] += (*myVelocities)[i]*mySold*0.5*h*
                              (1.0/myS+1.0/mySold);
      calculateForces();
      myPot = myEnergies->potentialEnergy();
      Real tempS = mySold/myS;
      tempS *= tempS;
      myPi += 0.5*h*(kineticEnergy(myTopo,myVelocities)*tempS-myGKT*
                     (1.0+log(myS))-myPot+myH0-0.5*myPi*myPi/myQ);
      tempS = mySold/myS;
      for (unsigned int i = 0; i < myVelocities->size(); i++)
         (*myVelocities)[i] = (*myVelocities)[i]*tempS
                     + myForces*0.5*h/myTopo->atoms[i].scaledMass;
      step = step + 1
   }
}

void NosePoincareIntegrator::getParameters(vector<Parameter>& parameters)
 const {
    STSIntegrator::getParameters(parameters);
    parameters.push_back(Parameter("temperature",
            Value(myNPTTemperature,ConstraintValueType::NotNegative())));
    parameters.push_back(Parameter("pi"), Value(myPi));
    parameters.push_back(Parameter("s"), Value(myS));
    parameters.push_back(Parameter("q"), Value(myQ));
}

STSIntegrator* NosePoincareIntegrator::doMake(string&,
 const vector<Value>& values,ForceGroup* fg)const{
    return new
          NosePoincareIntegrator(values[0],values[1],values[2],
                                 values[3],values[4],fg);
}
```

2.2 Example: LN

The LN algorithm is based on the combination of force splitting and Langevin Dynamics [6]. PROTOMOL has a single-timestepping Langevin impulse scheme, which can be used to evaluate fast forces. The outer slow force evaluation calculates the slow forces and updates atomic positions by a half-kick. Thus, the LN scheme is based on Position Verlet over Velocity Verlet, for reasons of stability in large timescale operations.

The header file is set up in exactly the same fashion, except that the integrator inherits from STS instead of MTS, and defines an integer `cyclelength` and opposed to `timestep`. As in STS integrators with the timestep, the cyclelength is implicitly defined as the first parameter for an MTS scheme. Also, the constructor must accept one extra parameter, for the next integrator in the hierarchy (when using LN, this will most often be PROTOMOL 's `LangevinImpulse`. Illustrating this through the class declaration and constructor:

```
class LNIntegrator : public MTSIntegrator
{
   ...

   public:
      LNIntegrator(int cycles,
                   ForceGroup* overloadForces,
                   StandardIntegrator* nextIntegrator);

   ...

}
```

In initialization, MTS integrators must also initialize their inner integrator(s). Similarly, these inner integrators must be run at some point in the `run()` method to ensure that faster forces are evaluated appropriately and at the right times. In `initialize()`, this can all be saved through a call to `MTSIntegrator::initialize()` and passing the same four parameters: topology, atomic positions and velocities, and energies. In our `run()` routine, we update positions, calculate slow forces and then run the next integrator. The next integrator is stored in the variable `myNextIntegrator`, and the MTS integrator implicitly defined parameter `myCycleLength` gets passed as the number of steps. `getParameters()` and `doMake()` are specified similarly for MTS and STS, as is the `keyword`, so we leave them out here. We show the `initialize()` and `run()` methods:

```
void LNIntegrator::initialize(GenericTopology *topo,
                              Vector3DBlock *positions,
                              Vector3DBlock *velocities,
                              ScalarStructure *energies){
   MTSIntegrator::initialize(topo, positions, velocities, energies);
   initializeForces();
}
```

```
void LNIntegrator::run(int numSteps) {
   int step = 1;

   while (step < numTimesteps) {
      for (unsigned int i = 0; i < myPositions->size(); i++)
         (*positions)[i] += getTimestep()/2 * (*velocities)[i];
      calculateForces();
      myNextIntegrator->run(myCycleLength);
      step++;
   }
}
```

Chapter 3

Adding a New Force

In an MD simulation, forces can in general be grouped into two subclasses: Bonded and Nonbonded. The former occurs between atoms which are connected in some fashion by bonds. These forces are thus short-range and associated motion tends to be faster. Examples include bond stretching and angle or torsion rotation forces. Nonbonded forces like van der Waals and electrostatic can occur between atoms which are not connected by bonds and are thus long range.

Adding a force to PROTOMOL involves the following steps:

1. Create a new class, called <*force name*>`Force`. Decide if the force should operate on just positions (a system force) or on unprocessed positions and velocities (an extended force). If the former is true, inherit from `SystemForce` and <*force name*>`ForceBase`. If the latter is true, inherit from `ExtendedForce`. If the force is bonded, template the class using one parameter: `TBoundaryConditions`. If the force is nonbonded, template the class using minimally two parameters: `TBoundaryConditions` and `TCellManager`. If the force is nonbonded and requires a cutoff, template a third parameter `TSwitchingFunction`.

2. Create a `void` method `evaluate()`, which accepts minimally four arguments:

   - `const GenericTopology* topo`
   - `const Vector3DBlock* positions`
   - `Vector3DBlock* forces`
   - `ScalarStructure* energies`

   If you are created an extended force, a fifth argument `const Vector3DBlock* velocities` should be placed between `positions` and `forces`. `positions` is a vector of atomic coordinate positions, and `forces` is the corresponding vector of forces. This vector should be modified by `evaluate()` to update atomic forces. `topo` and `energies` include data that can be helpful for force calculations, for their interfaces view `framework/topology/GenericTopology.h` and `framework/base/ScalarStructure.h`.

3. Create a `void` method `parallelEvaluate()`, which accepts the same arguments. If you are running in parallel and have an algorithm to evaluate this force in parallel, you can implement it here. PROTOMOL employs the *Message Passing Interface* (MPI) for parallel execution. `AngleSystemForce` provides an example where every angle is evaluated on a separate processor if possible, or if there are too few processors as much parallelism is exploited as possible. Two useful routines are `Parallel::next()`

and `Parallel::getAvailableNum()`. The latter returns the number of processors on which PROTOMOL is currently running, and `Parallel::next()` can be a conditional clause which surrounds a block of code that should be executed on individual CPUs in parallel (again, see `AngleSystemForce`). If there is nothing special that should be done for this force when running in parallel, simple include an invocation of `evaluate()` in the function body.

4. Define any helper variables for calculating this force in the class header. This includes any variables that the user should be able to define in the configuration file.

5. Define a method `getParameters()`, which is `const` and returns `void`. This method should accept one argument: a `vector<Parameter>& parameters`. The purpose of this method is to designate how parameters for this force should be referenced in the configuration file, their data types, constraints and default values. For example, suppose we wanted to define a data member `myAlpha` for this force, and have it initialized in the configuration file using the tag `"-alpha"`. And we wanted to constrain the value to be positive with a default of 1.0. This can be accomplished by adding the following line to this function body:
   ```
   parameters.push_back(Parameter("-alpha",Value(myAlpha,
      ConstraintValueType::Positive()),1.0));
   ```
   The constraints and default values can be left out if they are non-applicable for a parameter.

6. A method `doMake()` which accepts two parameters. The first is a `string &` for an error message, and the second a `vector<Value> values`. This method should also be `const` but this time should return a `Force*`. The error message string can be modified within the function, if a parameter violates some sort of constraint. If a constraint was programmed into the above `getParameters()` method, that can simply be checked by a call to `values[i].valid()` where `i` is some integer. `values` is an array which stores user-specified configuration file values for all parameters registered in the `getParameter()` method. This function should return a call to the constructor for this force, passing all parameters from `values`. For example if there are two configuration file parameters for this force, the return statement should be:
   `return new` <*force name*>`Force(values[0], values[1]);`

7. Create a default constructor, which sets default values for all member variables.

8. Create a constructor which accepts as arguments all parameters that the user should specify in the configuration file. Set the corresponding member variables.

9. Create a destructor that cleans up an dynamically allocated memory, if this does not apply it can be empty.

10. Create a class <*force name*>`ForceBase` with one single data member, a `static const string keyword`.

11. In the implementation file for this class, insert the line:
    `const string` <*force name*>`ForceBase::keyword("`<*force name*>`");`
    This allows the newly created force to be referenced by the passed keyword in the configuration file.

12. Modify `Makefile.am` in the same directory, and add the new `.cpp` and `.h` files under the appropriate macros (it should be clear).

13. Register the new force with a PROTOMOL force factory. For a bonded force, change to the directory `framework/factories` and find the file `registerForceExemplarsBonded.cpp`. You

will find two methods there, one which accepts periodic boundary conditions and the other accepts vacuum. To the first method, add the line:

```
ForceFactory::registerExemplar(
   new <force name>Force<PeriodicBoundaryConditions>());
```

And to the second:

```
ForceFactory::registerExemplar(
   new <force name>Force<VacuumBoundaryConditions>());
```

For a nonbonded force, change to the directory `framework/forces` and find the file `registerForceExemplar`
You will find two methods there, one which accepts periodic boundary conditions and the other accepts vacuum. To each of these methods, addd enough calls to `registerExemplars()` to cover all possible template parameters. For `TCellManager`, there is only one build into PROTOMOL currently, `CubicCellManager`. For switching functions, PROTOMOL supports `UniversalSwitchingFunction`
(no cutoff), `C1SwitchingFunction` (continuous first derivative), `C2SwitchingFunction` (continuous second derivative), `ShiftSwitchingFunction` (shifting the force by a constant to avoid sudden changes [6]), and `CutoffSwitchingFunction` (a simple cutoff, no smoothing). Also available are `CnSwitchingFunction` (accepts an arbitrary *n*) for its smoothness, and `CmpCnCnSwitchingFunction` for its complement. So for example, if the new force is nonbonded and compatible with only universal, C1 and C2 switching functions, add the following lines to the first method (which accepts periodic boundary conditions):

```
  new <force name>Force<PeriodicBoundaryConditions, CubicCellManager,
UniversalSwitchingFunction>());
  new <force name>Force<PeriodicBoundaryConditions, CubicCellManager,
C1SwitchingFunction>());
  new <force name>Force<PeriodicBoundaryConditions, CubicCellManager,
C2SwitchingFunction>());
```

And to the second method (accepting vacuum boundary conditions):

```
    new <force name>Force<VacuumBoundaryConditions, CubicCellManager,
UniversalSwitchingFunction>());
    new <force name>Force<VacuumBoundaryConditions, CubicCellManager,
C1SwitchingFunction>());
    new <force name>Force<VacuumBoundaryConditions, CubicCellManager,
C2SwitchingFunction>());
```

3.1 Example: Morse Potential

In MD, the Morse potential [5] models negligible bond fluctuations around equilibrium states, represented by the energy equation:

$$E(x) = D(1 - e^{-S_m(x-\bar{x})})^2 \tag{3.1}$$

where S_m represents the width of the well, and D is a dissociation energy at which the Morse term levels off for a large *x*.

By computing the negative gradient we obtain the force term:

$$F(x) = 2DS_m[e^{-2S_m(x-\bar{x})} - e^{-S_m(x-\bar{x})}] \tag{3.2}$$

We now show how to add this term to PROTOMOL . We will create a new class, called `MorseBondSystemForce`. The full implementation must reside in the header file `MorseBondSystemForce.h`, since the class will be templated with the boundary conditions. We will implement this force both in serial and parallel and thus define methods `evaluate` and `parallelEvaluate`, and an auxiliary method `calcMorseBond` to be invoked by both methods which evaluates the Morse potential for a single bond passed as parameter two, to avoid code replication. `getParameterSize()` in this case will return `2` (for S_m and D), ad we define `getParameters` and `doMake` which will define how these parameters are represented in the configuration file and how the object should be initialized. The class must inherit from `SystemForce` and `MorseBondSystemForceBase`, a simple interface which we later define. We define both default and parameterized constructors, setting the values for S_m and D accordingly.

In the implementation of `calcMorseBond`, we first need to know the current distance between the two atoms in the bond being calculated. Thus we obtain the (x, y, z) coordinates of their two positions in `atom1` and `atom2`, and then compute a vector between these two positions, stored in `x12`. The Euclidean distance, stored in `x`, is the normal of this vector. The magnitude of the force `F` is computed using the above equation, with $\bar{x}$ represented by the rest length of the current bond. We then set the forces on each atom by scaling the magnitude of the Morse force by the position vector `x12`, and then updating the bond energy with the above equation.

Our implementation of `evaluate` is simple. We obtain the boundary conditions from the topology and then iterate through every bond in the molecular system, invoking `calcMorseBond` for each one. For `parallelEvaluate`, we start the same way by obtaining the boundary conditions, but we now want to optimally evaluate individual bonds on different processors. Ideally, we could submit one bond per processor, but there may very well be more bonds than processors. Thus, we obtain the number of bonds and store it in an integer `n`, and the minimum of either the number of processors available or the number of bonds and store it in `count`. `count` thus represents the maximum number of parallel evaluations we can execute. We then loop `count` times, and at each iteration a new processor becomes available, checked with `Parallel::next()`. On that processor we evaluate a *range* of bonds, divided as even as possible based on the value of `count`.

We set configuration file parameters by defining their tags and corresponding data members in `getParameters` and sinvoking the parameterized `MorseBondSystemForce` constructor passing the parameters in the correct order in `doMake` and returning the object.

```
#ifndef MORSEBONDSYSTEMFORCE_H
#define MORSEBONDSYSTEMFORCE_H

#include "SystemForce.h"
#include "MorseBondSystemForceBase.h"
#include "ScalarStructure.h"
#include "Parallel.h"

namespace ProtoMol {
  template<class TBoundaryConditions>
  class MorseBondSystemForce : public SystemForce,
                               private MorseBondSystemForceBase{
   public:
    MorseBondSystemForce() : mySm(0.0), myD(0.0) {}
    MorseBondSystemForce(double sm, double d) : mySm(sm), myD(d) {}
    virtual void evaluate(const GenericTopology* topo,
                          const Vector3DBlock* positions,
                          Vector3DBlock* forces,
                          ScalarStructure* energies);

    virtual void parallelEvaluate(const GenericTopology* topo,
                                  const Vector3DBlock* positions,
                                  Vector3DBlock* forces,
                                  ScalarStructure* energies);
    void calcMorseBond(const TBoundaryConditions &boundary,
                       const Bond& currentBond,
                       const Vector3DBlock* positions,
                       Vector3DBlock* forces,
                       ScalarStructure* energies);
    virtual unsigned int getParameterSize() const{return 2;}
    virtual void getParameters(std::vector<Parameter>&) const;

  private:
    virtual Force* doMake(std::string&, std::vector<Value>) const;
    double mySm, myD;
  };


  template<class TBoundaryConditions>
  inline void MorseBondSystemForce<TBoundaryConditions>::evaluate(
                                          const GenericTopology* topo,
                                          const Vector3DBlock* positions,
                                          Vector3DBlock* forces,
                                          ScalarStructure* energies)
  {
    const TBoundaryConditions &boundary =
      (dynamic_cast<const SemiGenericTopology<TBoundaryConditions>& >(*topo))
           .boundaryConditions;

    for (unsigned int i = 0; i < topo->bonds.size(); i++)
      calcMorseBond(boundary, topo->bonds[i], positions, forces, energies);
  }

  template<class TBoundaryConditions>
  inline void MorseBondSystemForce<TBoundaryConditions>::calcMorseBond(
                                          const TBoundaryConditions &boundary,
                                          const Bond& currentBond,
                                          const Vector3DBlock* positions,
                                          Vector3DBlock* forces,
                                          ScalarStructure* energies)
  {
    Vector3D atom1((*positions)[currentBond.atom1]);
    Vector3D atom2((*positions)[currentBond.atom2]);
    Vector3D x12(boundary.minimalDifference(atom2,atom1));
    Real x = x12.norm();

    Real F = 2.0 * myD * mySm *
            (exp(-2*mySm*(x - currentBond.restLength))
             - exp(-mySm*(x - currentBond.restLength)));

    (*forces)[currentBond.atom1] += x12*F;
    (*forces)[currentBond.atom2] -= x12*F;

    (*energies)[ScalarStructure::BOND] += myD*pow(
                              1 - exp(-mySm*(x - currentBond.restLength)), 2);
  }

  template<class TBoundaryConditions>
  inline void MorseBondSystemForce<TBoundaryConditions>::parallelEvaluate(
                                          const GenericTopology* topo,
                                          const Vector3DBlock* positions,
                                          Vector3DBlock* forces,
                                          ScalarStructure* energies)
  {
    const TBoundaryConditions &boundary =
      (dynamic_cast<const SemiGenericTopology<TBoundaryConditions>& >(*topo))
                   .boundaryConditions;

    unsigned int n = topo->bonds.size();
    unsigned int count = min(Parallel::getAvailableNum(),
                             static_cast<int>(topo->bonds.size()));
    for(unsigned int i = 0;i<count;i++)
      if(Parallel::next())
        for (int j = (n*i)/count; j < min((n*(i+1))/count, n); j++)
           calcMorseBond(boundary, topo->bonds[j], positions, forces, energies);
  }

  template<class TBoundaryConditions>
  inline void MorseBondSystemForce<TBoundaryConditions>::getParameters(
                                                 std::vector<Parameter>&) const
  {
     parameters.push_back(Parameter("-Sm", Value(mySm)));
     parameters.push_back(Parameter("D", Value(myD)));
  }

  template<class TBoundaryConditions>
  virtual Force* MorseBondSystemForce<TBoundaryConditions>::doMake(std::string&,
                                                         std::vector<Value>) const
  {
     return MorseBondSystemForce(values[0], values[1]);
  }
}

#endif
```

We also must define the interface for `MorseBondSystemForceBase`. This as mentioned is a simple class, defining a keyword by which the force is identified in the configuration file and recognized by the force factory of PROTOMOL . We can call the keyword `MorseBond`. Thus the header and implementation files appear as follows:
`MorseBondSystemForceBase.h`:

```
#ifndef MORSEBONDSYSTEMFORCEBASE_H
#define MORSEBONDSYSTEMFORCEBASE_H

#include<string>

namespace ProtoMol {
  class MorseBondSystemForceBase {
  public:
    virtual ~MorseBondSystemForceBase(){}
    static const std::string keyword;
  };
}
#endif
```

`MorseBondSystemForceBase.cpp`:

```
#include "BondSystemForceBase.h"
using std::string;

namespace ProtoMol {
  const string BondSystemForceBase::keyword("MorseBond");
}
```

Finally, we add this force to the PROTOMOL source file `framework/factories/registerForceExemplarsBonded.cpp`, adding it to the methods for periodic and vacuum boundary conditions:
`registerForceExemplarsBonded.cpp`:

```
void registerForceExemplarsBonded(const PeriodicBoundaryConditions*){
   ...
   ForceFactory::registerExemplar(
      new MorseBondSystemForce<PeriodicBoundaryConditions>());
   ...
}
```

```
void registerForceExemplarsBonded(const VacuumBoundaryConditions*){
   ...
   ForceFactory::registerExemplar(
      new MorseBondSystemForce<VacuumBoundaryConditions>());
   ...
}
```

3.2 Example: Pairlist Algorithm

Suppose we want to add an algorithm for all pairwise forces which restricts their computation to pairs of atoms within a rectangular box with corners at r_0 and r_c. This can be done by implementing a special class with four templates – one for the cell manager, the second for a pair of atoms, the third for an abstract force class and the fourth for a specific implementation of the force class. This is shown below, in the new class `NonbondedRangeForce`. `NonbondedRangeForce` inherits from the `TForce` template parameter which represents the abstract pairwsise force class, along with `NonbondedRangeForceBase` which is implemented analogous to `MorseBondSystemForceBase` with a keyword as the data member set in the implementation file. We can use the keyword `NonbondedRange`, to allow representation in the configuration file as *-algorithm NonbondedRange* under LennardJones or Coulomb Force descriptions.

As an algorithm, the implementation contains a method `doEvaluate()` which only accepts two parameters - the first for the topology and the second for the number of atom pairs to evaluate `n`. `getParameters()` and `doMake()` have special implemenations as well, a bit different from simply adding a new force. In `getParameters()`, we obviously need values for `myR0` and `myRC`, but we also need to invoke the atom pair `getParameters()` method as well, as there may be further parameters to collect. In `doMake()`, the force that is returned is an object of the specific implementation of the force class which was passed in the template (parameter 4). Thus that `doMake()` is invoked, accepting as a parameter an invokation to `doMake()` for the atom pair. This latter method can essentially be copied into any algorithm implementation. In addition, we provide two new data members: the first to contain atom pairs and the second an enumerator which loops through all atom pairs, if periodic boundaries are used then all atom pairs within one periodic cell.

In `doEvaluate()`, the opening section performs enumeration through all atom pairs within a cell, and can be copied for any algorithm. The key statement for this algorithm is the condition which compares atomic positions to the values of r_0 and r_c. If the positions of atoms *i* and *j* are within this range, calculations are performed on the atom pair.

```
#ifndef NONBONDEDRANGEFORCE_H
#define NONBONDEDRANGEFORCE_H

#include "Force.h"
#include "NonbondedRangeForceBase.h"

namespace ProtoMol {
  template<class TCellManager, class TOneAtomPair,
           class TForce, class TImplForce>
  class NonbondedRangeForce : public TForce,
                  private NonbondedRangeForceBase {
   public:
    NonbondedRangeForce() : myR0(Vector3D(0.0,0.0,0.0)),
                            myRc(Vector3D(0.0,0.0,0.0)) {}
    NonbondedRangeForce(Vector3D r0, Vector3D rC, TOneAtomPair oneAtomPair) :
       TForce(), myR0(r0), myRC(rC), myOneAtomPair(oneAtomPair) {}
    virtual std::string getKeyword() const {return keyword;}
    virtual void getParameters(std::vector<Parameter>& parameters) const;
    virtual unsigned int getParameterSize() const
           {return 2+TOneAtomPair::getParameterSize();}

   protected:
    void doEvaluate(const GenericTopology* topo, unsigned int n);

   private:
    virtual Force* doMake(std::string& errMsg, std::vector<Value> values) const;
    Vector3D myR0, myRC;
    TOneAtomPair myOneAtomPair;
    Topology<TOneAtomPair::BoundaryConditions, TCellManager>::Enumerator
        enumerator;
};


template<class TCellManager, class TOneAtomPair, class TForce, class TImplForce>
void NonbondedRangeForce<TCellManager, TOneAtomPair,
  TForce, TImplForce>::doEvaluate(const GenericTopology* topo, unsigned int n) {

    ////////////////////////////////////////////////
    // COPY FOR ALL ALGORITHMS

    Topology<TOneAtomPair::BoundaryConditions,
             TCellManager>::Enumerator::CellPair thisPair;
    unsigned int count = 0;
    for (; !enumerator.done(); enumerator.next()) {
      enumerator.get(thisPair);
      bool notSameCell = enumerator.notSameCell();

      if(!notSameCell){
        count++;
        if(count > n)
          break;
      }
    ////////////////////////////////////////////////
      for(int i=thisPair.first; i!=-1; i=topo->atoms[i].cellListNext){
        for(int j=(notSameCell ? thisPair.second:topo->atoms[i].cellListNext); j!=-
                j=topo->atoms[j].cellListNext){
          if ( (*(myOneAtomPair.myPositions)[i] <= myRc) &&
               (*(myOneAtomPair.myPositions)[i] >= myR0) &&
               (*(myOneAtomPair.myPositions)[j] <= myRc) &&
               (*(myOneAtomPair.myPositions)[j] >= myR0) )
     myOneAtomPair.doOneAtomPair(i,j);
          }
      }
    }
}

  template<class TCellManager, class TOneAtomPair, class TForce, class TImplForce>
  void NonbondedCutoffForce<TCellManager,TOneAtomPair,
   TForce,TImplForce>::getParameters(std::vector<Parameter>& parameters) const {
    myOneAtomPair.getParameters(parameters);
    parameters.push_back(Parameter("-R0",Value(myR0)));
    parameters.push_back(Parameter("-Rc",Value(myRc)));
  }

  template<class TCellManager, class TOneAtomPair, class TForce, class TImplForce>
  Force* NonbondedCutoffForce<TCellManager,TOneAtomPair,
   TForce,TImplForce>::doMake(std::string& errMsg,
                            std::vector<Value> values) const{
    return ( new TImplForce(values[values.size()-1],
        TOneAtomPair::make(errMsg,
            std::vector<Value>(values.begin(),values.end()-1))));
  }

#endif
```

Chapter 4

Adding a New Modifier

The behavior of PROTOMOL integrators can be slightly changed or *modified* to encapsulate new functionality to be executed at specific points in time. As an example, Nose-Hoover dynamics requires computation of a frictional force which is added to Newton's equations of motion to establish atomic forces. MOLLY integrators require position averaging before computation and mollification while forces are computed. There are several different types of modifiers supported by PROTOMOL :

- **PreStep**: Executed before each integration step.
- **PreDriftOrNext**: Executed before updating positions (for STS integrators) or running the next integrator in the hierarchy (for MTS integrators).
- **PostDriftOrNext**: Executed after updating positions (for STS integrators) or running the next integrator in the hierarchy (for MTS integrators).
- **PreForce**: Executed before calculating any forces.
- **MediForce**: Executed after calculating system forces and before calculating extended forces.
- **PostForce**: Executed after calculating all forces.
- **PostStep**: Executed after each integration step.

Pseudocode for execution of these modifiers is shown in Figure 4.1.
A modifier is added to PROTOMOL using the following steps:

1. Create a class called <*modifier name*>`Modifier`.
2. Create a constructor, it can be default, but often an `Integrator*` is passed so that functionality of the integrator being modified is accessible.
3. Create a function `doExecute()`, accepting no parameters and returning `void`. In the function body, specify exactly what this modifier should do.
4. In the integrator class, if it is not already there, create a member method `addModifierAfterInitialize()`, returning `void`.
5. In the body of `addModifierAfterInitialize()`, if it is not already there add the statement: `STSIntegrator::addModifierAfterInitialize();`

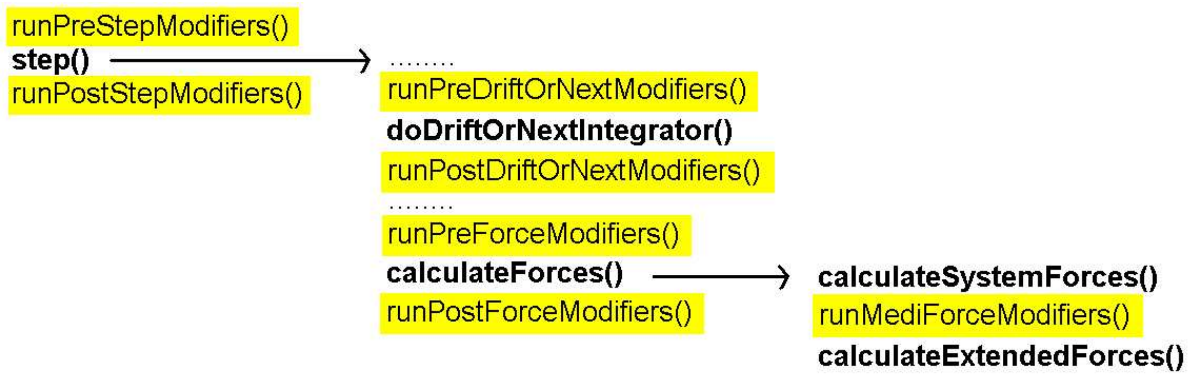

Figure 4.1: Pseudocode for modification of integrators. The `step()` method is shown, which propagates the molecular system one timestep. `doDriftOrNextIntegrator()` either updates positions for an STS integrator or calls the next integrator in the hierarchy for MTS.

6. Add the following statement to the body of `addModifierAfterInitialize()` and before the statement `STSIntegrator::addModifierAfterInitialize();`
`adopt`<*modifier type*>`Modifier(new Modifier`<*modifier name*>`(`<*constructor parameters*>`));`
where *modifier type* is one of the seven types mentioned above, *modifier name* is the name you provided for the modifier, and *constructor parameters* are passed to the constructor you created in step 2. To pass the integrator object, use `this`.

4.1 Example: RATTLE

The minimum usable timestep in molecular dynamics simulation without introducing instabilities is limited by the magnitude of the fastest frequency. In biomolecules, this fastest frequency occurs in bond motion. As a result, some algorithms have been developed which *constrain* bond motion with the goal of increasing the timestep of simulations. One such algorithm is RATTLE [1], which also constrains velocities of atoms involved in constrained bonds. The constraint equation to be followed for two constrained bonded atoms *a* and *b* is:

$$(v_a(t) - v_b(t))\dot{(}r_a(t) - r_b(t)) = 0 \tag{4.1}$$

In our implementation of RATTLE, we allow this above constraint to be satisfied within a small constant ϵ, which is set in the constructor and calculated using an auxiliary method `calcError()`. We also allow a maximum number of velocity updates with the above constraint unsatisfied before automatically stopping, another parameter. The new modifier inherits the `Modifier` interface which defines a method for `doExecute()`. This is all shown in the header file below:

```
#ifndef MODIFIER_RATTLE_H
#define MODIFIER_RATTLE_H

#include "Modifier.h"
#include "Bond.h"
#include "Vector3DBlock.h"
```

```
namespace ProtoMol {

   class Integrator;

   class ModifierRattle : public Modifier {
      ModifierRattle(Real eps, int maxIter, const Integrator* i);

   private:
      virtual Real calcError();
      virtual void doExecute();

      const Integrator* myTheIntegrator;
      Real myEpsilon;
      int myMaxIter;
      const std::vector<Bond::Constraint>* myListOfConstraints;
   };
}

#endif
```

In the implementation file, we first define the constructor and initialize our parameters: for ϵ, the maximum number of iterations and the integrator being modified. `myListOfConstraints` is populated with constrained bonds and angles by a separate PROTOMOL routine. The helper routine `calcError()` computes the above equation for RATTLE and returns the absolute value which is used for ϵ. This error is normalized by the number of constraints.

We update velocities of atoms *a* and *b* using the following equations:

$$v_a(t+\delta t) = v_a(t) + (1/m_a)\lambda_{ab}(t)r_{ab}(t)v_b(t+\delta t) = v_b(t) - (1/m_b)\lambda_{ab}(t)r_{ab}(t) \tag{4.2}$$

where $r_{ab}(t) = (r_a(t) - r_b(t))$, and $\lambda_{ab}(t)$ is a Lagrange multiplier, used to compute the change in momentum:

$$\lambda_{ab}(t) = \frac{-r_{ab}(t)v_{ab}(t)}{\delta t(1/m_a + 1/m_b)||r_{ab}||^2} \tag{4.3}$$

We iterate through this process, repeatedly updating velocities until either the constraint is satisfied within ϵ or the maximum number of iterations has been attained. This is implemented in `doExecute()`.

```
#include "ModifierRattle.h"
#include "Topology.h"
#include "topologyutilities.h"

namespace ProtoMol {

  ModifierRattle::ModifierRattle(Real eps, int maxIter, const Integrator* i):
                myEpsilon(eps), myMaxIter(maxIter), myTheIntegrator(i){}

  Real ModifierMetaRattle::calcError() const {
    Real error = 0;
    for(unsigned int i=0;i<myListOfConstraints->size();i++){
      int a1 = (*myListOfConstraints)[i].atom1;
      int a2 = (*myListOfConstraints)[i].atom2;
      Vector3D vab = (*myVelocities)[a1] - (*myVelocities)[a2];
      Vector3D rab = (*myPositions)[a1] - (*myPositions)[a2];
      Real err = fabs(rab * vab);
      error += err;
    }
    return error /= myListOfConstraints->size();
  }

  void ModifierRattle::doExecute(){
    Real error = calcError();
    Real dt = myTheIntegrator->getTimestep() / Constant::TIMEFACTOR;
    int iter = 0;
    while(error > myEpsilon) {
      for(unsigned int i=0;i<myListOfConstraints->size();i++) {
        int a1 = (*myListOfConstraints)[i].atom1;
        int a2 = (*myListOfConstraints)[i].atom2;
        Real rM1 = 1/myTopology->atoms[a1].scaledMass;
        Real rM2 = 1/myTopology->atoms[a2].scaledMass;
        Vector3D rab = (*myPositions)[a1] - (*myPositions)[a2];
        Real rabsq = rab.normSquared();
        Vector3D vab = (*myVelocities)[a1] - (*myVelocities)[a2];
        Real rvab = rab * vab;
        Real gab = -rvab / (dt * (rM1 + rM2) * rabsq);
        Vector3D dp = rab * gab;
        (*myVelocities)[a1] += dp * dt * rM1;
        (*myVelocities)[a2] -= dp * dt * rM2;
      }
      error = calcError();
      iter++;
      if(iter > myMaxIter)
        break;
    }
  }

}
```

Chapter 5

Adding a New Output

For complete results examination it is a common requirement to view a diverse set of parameter values. PROTOMOL can output data to file(s) or to the screen, and can write *trajectory files*, which resemble a table containing appropriate parameter values along with the simulation step. Files in particular can be convenient for graphing data values using plotting libraries such as Matlab.

To add a new type of output to PROTOMOL , follow these steps:

1. Change to the directory `framework/frontend`.
2. Create a class `Output`<*name of output*>. If the output will be to the screen, inherit from `Output`. Otherwise, if the output will be to a file(s), inherit from `OutputFile`.
3. Add to the class a data member `static const string keyword;`. Initialize this in the implementation file:
   `const string Output`<*name of output*>`::keyword("`*config keyword*`");`
   Where *config keyword* is how this output should be referenced in the configuration file.
4. Add other data members necessary for this type of output.
5. Define a method `getParameters` which accepts one parameter `vector<Parameter> &parameter`. This method should be declared `const`. The body of this method should contain one line constructing each parameter that should be specified in the configuration file associated with this output. These lines specify the configuration file keyword, the corresponding member variable, any constraints on parameter values and finally default values. It is often convenient to use the keyword for the output to refer to the file name through a call to `getId()`, defined for all children of `OutputFile`. For example, suppose we wanted to specify a file name, and an output frequency (this is very common, for example if we only want to output this data every certain number of simulation steps). Of course, any output frequency would have to be positive, and the file name cannot be empty, so those are our constraints. All `OutputFile` children implicitly have a `myFilename` defined. To hold the output frequency, we'll define `myOutputFreq` (see step 4). The following two lines would accomplish this:
   ```
   parameter.push_back(Parameter(getId(),
      Value(myFilename,ConstraintValueType::NotEmpty())));
   parameter.push_back(Parameter("TheOutputFreq",
      Value(myOutputFreq,ConstraintValueType::Positive())));
   ```
6. Define a constructor which accepts and sets all parameters set through the configuration file.

7. Define a default constructor, which sets any default values for member variables.

8. Define a method `doMake()` which returns an `Output*` and is `const`. This should accept a `string &` for an error message that can be modified within the function, and a `const vector<Value> & values`. Each of these values contains one parameter from the configuration file, in the order they were registered in `getParameters` – therefore, in the constructor formal parameters corresponding variables should appear in the same order. For example, if there were two configuration file parameters:
`return new Output`<*output name*>`(values[0], values[1]);`

9. Define three methods: `doInitialize()` (to be invoked once at the beginning of a simulation), `doRun(int step)` (to be invoked at every simulation step), and `doFinalize(int step)` (to be invoked at the last simulation step). These methods can be empty if there is nothing to do at their appropriate times.

10. Finally, change to the directory `framework/factories`. Open the file `registerOutputExemplars.cpp`. In that file you will find a method `registerOutputExemplars()`. Add the line:
`OutputFactory::registerExemplar(new Output`<*output name*>`());`

5.1 Example: Selective Kinetic Energy

Suppose we wanted to add a new output to PROTOMOL which can accept a list of atom names from the configuration file, and output the kinetic energy to a file just taking into account these names of atoms. The user can specify `SelectiveKineticFile` for the output file name, `SelectiveKineticFreq` for the frequency, and finally `SelectiveKineticAtoms` for the atom names. We can assume `SelectiveKineticAtoms` will have a string of atom names associated with it. The header file would appear as follows:

```
#ifndef OUTPUTSELECTIVEKINETIC_H
#define OUTPUTSELECTIVEKINETIC_H

#include "OutputFile.h"
#include <vector>
using std::vector;

namespace ProtoMol {

   class OutputSelectiveKinetic : public OutputFile {
      public:
         OutputSelectiveKinetic() {}
         OutputSelectiveKinetic(const std::string& filename, int freq,
std::string atomnamelist);
         virtual ~OutputSelectiveKinetic() {}

         virtual void getParameters(std::vector<Parameter> &parameter) const;
         virtual unsigned int getParameterSize() const {return 3;}

         static const std::string keyword;
      private:
         virtual Output* doMake(std::String& errMsg,
                                const std::vector<Value>& values) const;
         virtual void doInitialize() {}
         virtual void doRun(int step);
         virtual void doFinalize(int step);

         std::string myAtomNameList;
         vector<std::string> myAtomNames;
}
#endif
```

We will inherit from `OutputFile`, since we are writing to a file. This means that we must include `OutputFile.h`. We will represent the list of atoms by an STL `vector` of `strings`, so we include the `vector` interface and also declare a helper variable `myAtomNames` of this datatype. We define two constructors, one a default which in this case does nothing, and a second that is parameterized with all expected user parameters for this type of output (in this case, there are three – the file name, the frequency and the list of atoms which in the configuration file will be a string delineated in some way, either by comma, dashes, *etc*). Since we inherit from `OutputFile` there are already data members `myFile` (Output stream), `myFileName` (output file name), and `myOutputFreq` (frequency). Therefore we just need one more data member for the string parameter which specifies the list of atom names. We also declare a destructor which does nothing.

We declare a data member `keyword` to be `static const`, we will initialize this later in the implementation file. This is how low level factories recognize this output (therefore it is important to make sure no two outputs have the same keyword). We then declare the methods `doMake`, `doInitialize`, `doRun` and `doFinalize` as specified. `doInitialize` in this particular case is empty.

Finally, we declare a method `getParameters` and also `getParameterSize()` which returns in our case 3, since there are three configuration file parameters associated with this output.

Now let's view the implementation file:

```
#include "OutputSelectiveKinetic.h"

using std::vector;

namespace ProtoMol {
   const string OutputSelectiveKinetic::keyword("OutputSelectiveKinetic");

   OutputSelectiveKinetic::OutputSelectiveKinetic(const std::string& filename,
                                                  int freq,
                                                  std::string atomnamelist)
   {
      myFilename = filename;
      myOutputFreq = freq;
      open();
      myAtomNames = parseOutStrings(atomnamelist);
   }
   void OutputSelectiveKinetic::doRun(int step)
   {
      Real kineticEnergy = 0.0;
      for (unsigned int i = 0; i < myTopology->atoms.size(); i++) {
        if (findIn(myAtomNames, myTopology->atoms[i].atomTypes[i].name))
           kineticEnergy += myTopology->atoms[i].scaledMass
                                 * ((*velocities)[i]).normSquared();
      }
      myFile << step << " " << kineticEnergy << endl;
   }

   void OutputSelectiveKinetic::doFinish() {close();}

   Output* OutputSelectiveKinetic::doMake(string&,
                                         const vector<Value>& values) const {
      return (new OutputSelectiveKinetic(values[0],values[1],values[2]));
   }

   void OutputSelectiveKinetic::getParameters(vector<Parameter> &parameter) const {
      parameter.push_back(Parameter("SelectiveKineticFile",
                          Value(myFilename,ConstraintValueType::NotEmpty())));
      parameter.push_back(Parameter("SelectiveKineticFreq",
                          Value(myOutputFreq,ConstraintValueType::Positive())));
      parameter.push_back(Parameter("SelectiveKineticAtoms",
                          Value(myAtomNameList, ConstraintValueType::NotEmpty())));
   }
}
```

We include the corresponding header file and set the `keyword` variable to `OutputSelectiveKinetic`. Next, we implement all remaining methods. First is the parameterized constructor, where we can set the values for `myFileName` and `myOutputFreq`. Inheriting from `OutputFile` gives the unique ability to invoke `open()`, and have `myFile` automatically set to an output stream writing to `myFileName`. `doFinish()` then just has to call `close()` to close this stream. Finally we set our helper `vector` for the list of atom names from the passed string. We assume some sort of string parsing routine has been implemented for this conversion and for now call it `parseOutStrings()`.

We specify the names of parameters to expect in the configuration file in `getParameters`. Note that we push the parameter objects and their constraints into the `parameter` vector in the order that we initialize them in the parameterized constructor. This way when `doMake` is called it can just pass these parameter values in order to the constructor and return a new object. The constraint on our file name and list of atom names is that they not be empty, and on our frequency that it not be negative or zero.

Finally the method `doRun` computes the total kinetic energy between all atom names selected by the user and outputs it to the output file along with step number. All atoms in the topology are looped through and the corresponding atom name is checked to see if kinetic energy should be counted - we assume there is a `findIn` method defined which returns `true` or `false` depending if a passed `string` is in a `vector` of `strings`. Kinetic energy is computed by multiplying the masses of these atoms by their normalized velocities squared.

Chapter 6

Adding a New Topology

This type of extension will never be done unless you are adding a new type of boundary conditions or cell manager. But if either of these are necessary, follow the following steps once you've accounted for these new boundary conditions/manager in the framework and added them as new classes in `framework/topology`.

1. Change to the directory `framework/factories`.
2. Open the file `registerTopologyExemplars.cpp`.
3. Ensure that in every line in this method which invokes `TopologyFactory::registerExemplar()`, that all possible pairs of boundary conditions and cell managers are present. Currently PROTOMOL supports two boundary conditions (periodic and vacuum) and one cell manager (cubic). Thus for example if adding a second cell manager to PROTOMOL as it stands, you will need to add two more calls to `registerExemplar`, one pairing the manager with periodic boundary conditions and the other with vacuum.

Chapter 7

Conclusions

We have presented a guide on how to extend PROTOMOL to encapsulate new algorithms for force calculation, system integration (including modifiers), outputs and topologies. Our hope is that these guidelines will help PROTOMOL to be a more useful framework for prototyping new methods for running molecular dynamics simulations.

Full guidelines for how to setup a configuration file can be found in the PROTOMOL user guide, and a full description of back end classes in the programmer's guide. Further questions can be directed to `protomol@cse.nd.edu`.

Bibliography

[1] H. C. Andersen. Rattle: A 'velocity' version of the Shake algorithm for molecular dynamics calculations. *J. Comput. Phys.*, 52:24–34, 1983.

[2] Stephen D. Bond, Benedict J. Leimkuhler, and Brian B. Laird. The Nosé–Poincaré method for constant temperature molecular dynamics. *J. Comput. Phys*, 151(1):114–134, 1999.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, 1995.

[4] Q. Ma and J. A. Izaguirre. Targeted mollified impulse — a multiscale stochastic integrator for long molecular dynamics simulations. *Multiscale Model. Simul.*, 2(1):1–21, 2003.

[5] P. M. Morse. Diatomic molecules according to the wave mechanics. ii. vibrational levels. *Phys. Rev.*, 34:57–64, 1929.

[6] Tamar Schlick. *Molecular Modeling and Simulation - An Interdisciplinary Guide*. Springer-Verlag, New York, NY, 2002.