Interactive Molecular Dynamics
Dylan Brandtner, Trevor Cickovski, and Jesús A. Izaguirre
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556

1 Introduction and Motivation

The high-performance molecular dynamics (MD) framework ProtoMol [11] can prototype novel MD algorithms, and has been designed using object-oriented programming, inheritance and design patterns for flexibility. Techniques such as efficient performance-critical code generation through templates and optimizations enabled by parallelism increase ProtoMol’s speed. In addition, the design is based on fast solvers for force computation, further enhancing performance.

Though ProtoMol is powerful and useful, it does have shortcomings. First, results from a simulation are deposited into multiple output files of different formats which makes analysis difficult. Also, simulations are non-interactive and require a static configuration file containing a list of keywords and values that are initialized before a simulation is run. Finally, new integrators and force calculation methods cannot be added to ProtoMol without writing new C++ classes and recompiling. In short, ProtoMol is not user-friendly nor extensible for non-programming experts of the scientific community. Therefore, our goal is to create a high-level, user-friendly interface called MDSimLab for initializing and running interactive simulations using ProtoMol, and allow users to more easily extend ProtoMol with new techniques.

The developers of the parallel MD framework SPaSM [14] faced similar problems with their software. Their solution, called computational steering [3,4], combines simulation, visualization, and data analysis into a single interactive system. This technique required low-level SPaSM code to be abstracted to a higher level scripting (control) language, making it easier to perform and analyze complex simulations with user-friendly commands and defaults. They chose Python [12] as their scripting language, which has object-oriented syntax, easily extended with new data types and modules, and is highly portable. To access C++ functionality through Python, a user needed to write special “wrapper” functions that provided a connection between the scripting language and the underlying functionality; however, this process was technical, tedious, poorly documented, and highly error-prone. Thus, they developed a tool called SWIG (Simplified Wrapper and Interface Generator) [2] to automate this process. SWIG can be used to build a scripting interface from C/C++ functionality, and supports classes, functions, variable declarations, and all built-in types. SWIG can wrap to Python, Tcl, Perl, and Guile-III and automatically generates documentation.

Using the same idea, we wrapped ProtoMol functionality to Python using SWIG. We then created a high-level interface for MDSimLab using Python scripts to invoke ProtoMol functionality, giving ProtoMol more interactive and analytical power. Eventually we will convert this syntax to a domain-specific language, which will provide users with a more natural vocabulary for running MD simulations. This language can also be optimized accordingly for MD simulations [7], yielding better performance. We describe our work in two stages: (1) wrapping, and (2) testing, abstraction, and expansion. Finally, we discuss future work and development of our project.

2 Stage I: Wrapping

2.1 A Simple Example

The first step was to wrap the entire ProtoMol framework to Python using SWIG. Here is a simple example of SWIG wrapping:

Step 1: Gather Files. Suppose you have the following C++ files for the factorial function:

The header file:

```c
//fact.h
int fact(int n);
```
and the source file:

```cpp
//fact.cpp
int fact(int n){
    if (n <= 1) return 1;
    else return n*fact(n-1);
}
```

**Step 2: The Interface File.** SWIG requires an interface file as its input, which would look like this for the above files:

```i
//fact.i

#include fact.h
%
#include fact.h
```

**Step 3: Compile and Link.** SWIG must then be run on the interface file using this command:

```
$ swig -python fact.i
```

which generates the fact_wrap.cxx file. Then the source code must be compiled and linked using the following commands (assuming Python 2.2 is installed, and g++ is an installed compiler):

```
$ g++ -c fact.cpp fact_wrap.cxx -I/usr/include/python2.2
$ g++ -shared fact.o fact_wrap.o -o _fact.so
```

which generates the shared object file _fact.so, and the importable Python code fact.py. To make sure everything is linked correctly, one can run Python on fact.py.

**Step 4: Import to Python.** Now the factorial function can be imported and used in Python:

```
$ python
>>>import fact
>>>fact.fact(5)
120
```

### 2.2 Wrapping ProtoMol

The process above is simple for a trivial case, but wrapping a framework as complex as ProtoMol is much more difficult. We now outline some of the significant problems we faced.

**Problem 1: Need for Automation.** The ProtoMol framework contains roughly 300 header files that needed to be wrapped. For each of these, we had to create an interface file, and run the commands for compiling and linking. Ideally, this process should be automated. We automated the sequence of commands for compiling and linking using simple scripts called .wrapper files, and created a C++ program which input a module name and generated an interface file template, along with this .wrapper script. Another problem that arose was that it was impossible to see all the files that needed to be linked into a module without actually running Python on the generated file to check for linking errors, then going back to the script and adding the source and object files to the compile and link commands. This proved too time-consuming when modules needed to be linked to dozens of others. We lessened this overhead by creating another C++ program that input a script name and a new source file to be added, and made appropriate changes to the script. In addition, SWIG does not support templates in the same way as C++. In SWIG, every possible instance of a template must be renamed in the interface file as a separate instantiation using the SWIG %template command.
ProtoMol, possible template instantiations are registered in files called register<\textit{typename}>Exemplars. For example, forces are often templated and accept different boundary conditions or switching functions as their parameters, and the possible instantiations of these forces are specified inside registerForceExemplars files. Since the number of possible parameter combinations can grow quickly, we created another C++ program which read these exemplar files and added corresponding instantiations to the provided interface file. All of these extra programs we created significantly sped up the wrapping process but they are time-consuming to create. Problems also arose due to other C++ features that SWIG does not support. For example, operator overloads are not supported. If one wishes to overload a Python operator, the overloaded function must be renamed to the appropriate Python function. For example, to overload the + operator in Python, one must rename \texttt{operator+} to \texttt{\_add\_} in the interface file with the \texttt{\_rename} command. SWIG also does not support nested structs or classes. Therefore we needed to change the source code to un-nest certain structs and classes. When other problems arose that forced smaller changes in the source code, we used SWIG macros (#ifdef SWIGPYTHON) so that the changes would only be made when compiling with SWIG. As a future goal, we propose creating a tool to automate this wrapping process [8].

\textbf{Problem 2: Module Interference and Combined Modules.} When running larger tests, we often imported several modules, some of which contained function calls with the same name. Python had problems knowing which function to call. To solve this, we created importable super-modules that combined functionality of several sub-modules. In the interface file for the super-module, we included multiple interface files instead of header files, allowing linking of all appropriate sub-modules into one large library. This solved the interference problem, and also made interactive scripting easier for the user by only requiring one imported module to access functionality of several sub-modules.

3 Stage II: Testing, Abstraction, and Extension

We then needed to test the modules we wrapped, and began laying the foundation for MDSimLab. This included importing wrapped modules and testing basic functionality as well as extending and modifying these modules for Python compatibility. After the modules were functional, we began to abstract function calls and produce interactive and useful results, resulting in a user-friendly and powerful interface. We now describe this process for some of the core modules of ProtoMol.

3.1 Vector3D and Vector3DBlock

The first modules we tested were the ProtoMol Vector3D and Vector3DBlock interfaces. A Vector3D is a simple data structure that holds three values corresponding to 3D coordinates (x, y, and z). There are several operations that can be performed on these vectors such as addition, subtraction, dot and cross products, etc. A Vector3DBlock is a Standard Template Library (STL) [13] vector of Vector3Ds with additional functionality. These data structures are essential to ProtoMol's basic functionality and required extensive testing. First, we renamed operator overloads to Python operators in the interface file for compatibility:

\begin{verbatim}
%rename(_-copy_) Vector3D::operator=;
%rename(_-add_) Vector3D::operator+;
%rename(_-sub_) Vector3D::operator-;
...
\end{verbatim}

Then we extended the functionality of Vector3D using the \texttt{\_extend} command in the interface file to handle a deep copy (instead of Python's default shallow copy), as well as printing to the screen and inputting (since overloaded streams are SWIG-incompatible). We then extended Vector3DBlock to handle printing to the screen and added a function that copies the whole block into a native numerical Python array, which is faster than a Python list because it stores values instead of pointers. In the end, the Vector3DBlock scripting interface could, for example, perform the following:
3.2 Leapfrog Integrator

After the majority of ProtoMol was wrapped and core modules had been tested, we attempted to run a basic simulation. Part of a ProtoMol simulation involves initializing and running an integrator for a given number of steps. An integrator is a method for computing the forces between atoms in a given molecule and then updating their positions and velocities. As a starting point, we decided to test a Leapfrog integrator on the four-atom butane molecule.

**Hard Coded testing.** In order to initialize this integrator, we needed to specify energies, a topology, forces, initial positions, and initial velocities. For ProtoMol, information for the topology, the initial positions, and the forces is contained inside configuration, PDB [5], and CHARMM [6] PSF and PAR input files. A PDB file provides initial atom positions, a PSF file specifies molecular structure, and a PAR file provides initial parameters (such as spring constants for bonds). As a starting point, we hard coded this information into our test script. We created a Vector3DBlock of the initial positions and another of velocities, and hard coded all topology information into a ProtoMol Topology object (including atom, angle, bond, dihedral, and molecule information). We then added two simple forces into a ProtoMol ForceGroup object, which encompasses all forces for a given simulation. Initializing and running the integrator resulted in module interference, which we addressed by creating a library called LeapfrogIntegratorLib containing all necessary modules. This proved to be an adequate solution and we were then able to initialize and run the integrator using the following commands:

```python
... lfi = LeapfrogIntegratorLib.LeapfrogIntegrator(0.1, ForceGroup) lfi.initialize(Topology, Positions, Velocities, Energies) lfi.run(5)
```

This code first constructs a leapfrog integrator with a timestep of 0.1 initialized with a ForceGroup. It then sets the integrator’s initial topology, positions, and velocities from hard coded values earlier in the script, and provides it with a default structure for energy computation. Finally, it runs the integrator for five steps.
After generating random velocities using the same seed as ProtoMol, the results were nearly perfect. Final positions matched those of ProtoMol down to a 4th decimal place, and we attribute differences to value rounding and truncation.

File Reading and Factories. As a next step was to read initial information for the simulation from the PDB, PAR, and PSF files using ProtoMol’s reader modules. Then we read the configuration file, a much more complicated task. In order for a configuration file keyword to be read in ProtoMol, the keyword must first be registered with the back end, placing it into an STL map. In our script, none of the keywords were registered at first, and therefore all keywords in the configuration file were ignored when it was read. Since the keyword registration method of ProtoMol is complicated and contains many templates and macros, we used the SWIG %extend command to create our own simple method which reads a list of keywords from a text file and registered them. Using this method, we were able to read all configuration file information except the integrator, including PDB, PSF, and PAR input file names. Parsing the integrator information proved to be the most complicated reading task. ProtoMol uses factories to dynamically create an integrator object with the proper forces, using the information from the configuration file. Therefore, we tried creating an integrator by invoking the integrator factory [1] which in turn invokes the force factory and dynamically creates the entire integrator, returning a pointer to it. This process requires integrators and forces to be registered. As mentioned, ProtoMol has functions called register<typename>Exemplars that do this, but wrapping and calling them in the script resulted in a local change, which was ineffective. Therefore, we had to call them inside the factory to achieve the desired effect. After everything was registered, the integrator was properly created, and we used it in our script. At this point, all hard-coding of initial data was removed from the script.

Abstraction. Even after all hard coded information was removed, it took 42 lines to run this simple simulation and the commands were somewhat complicated. As a step towards creating the MDSimLab domain-specific language, we abstracted the script syntax into a much higher level interface. First, we encapsulated multiple function calls into single and simpler functions. For example, we encapsulated reading of the PSF file and populating a ProtoMol PSF object to hold the information into a single function defined below:

```python
def read_PSF(psf_filename):
    myPSF=PSFReader.PSF()
    myPSFReader=PSFReader.PSFReader(psf_filename)
    myPSFReader.read(myPSF)
    return myPSF
```

After importing the script that contains this definition, the function can be called as follows:

```python
>>> x = read_PSF("UA_butane.psf")
```

This single function call now constructs a PSF and PSFReader object, and then reads the information from UA_butane.psf into the PSF object and returns it (to x in this case).

Using this function and other similar methods, the test script was reduced to 10 lines, a 76% code size savings. However, this technique still required a lot of information to be saved and passed between functions in the test script:

```python
>>> from LeapfrogLanguage import *
>>> myConfig = read_Config("UA_butane.conf")
>>> myPSF = read_PSF("UA_butane.psf")
>>> myPAR = read_PAR_CHARMM28("UA_butane.par")
>>> myPDB = read_PDB("UA_butane.pdb")
>>> myTop = init_topology_from_config(myConfig,myPSF,myPAR)
>>> myVel = init_velocities_from_config(myConfig,myTop)
>>> myFG = init_force_group_from_config(myConfig)
>>> myLFI = init_leapfrog(myConfig,myTop,myPDB.coords,myFG,myVel)
>>> run_leapfrog(myLFI,5)
```
As can be seen in the above code, there are several objects such as myConfig, myPSF, myPAR, etc. being saved and passed between functions, some of them multiple times. Therefore, we created a single class that encapsulated all these functions as well as data members that were not necessary for the user to see in the script, such as the PSF and PAR objects, which are only needed to build the topology. This method allowed us to store data members in the object, saving the need to constantly pass them as function arguments. Next, we split the class apart to separate the data from the files and the integrator information for better organization. In the end, the simulation could be run with 6 simple commands, an 86% code size savings from our original script:

```python
from LeapfrogLanguage_class import *
SYS=System()
SYS.read_all_from_config("UA_butane.conf")
LF=LeapfrogIntegrator(SYS)
LF.init_all_from_config()
LF.run(5)
```

The first line imports our library module. The second initializes a System object which stores all the information read from the files, and the third reads all initial information and stores it in the object. The fourth line creates a Leapfrog integrator (including its forces) using the information inside the System object, and the fifth initializes the integrator (including its positions, velocities, topology, and energies), once again using this information. The last line runs the integrator for five steps.

**VMD and Plotting.** We were able to achieve interaction with VMD [10] for viewing molecules in 3D. Python scripts can be run from the VMD console, and VMD commands can be used in Python scripts. Therefore, we were able to read initial positions from the PDB file into VMD, and then update the positions onscreen using VMD commands as the simulation was running. This resulted in a moving butane molecule as shown in Fig.1, with atom positions changing as the integrator updated them. Furthermore, after incorporating a Python plotting package called Gnuplot [9], we were able to plot system potential energy as a function of time as the simulation executed, as shown in Fig.2.

![Fig. 1. Series of three snapshots from a simulation of four-atom butane, visualized with VMD. These are interactive shots taken while a simulation is running. Snapshots were taken at timesteps 0, 2000 and 3000.](image)

4 Looking Ahead

The are several immediate areas to explore involving MDSimLab. First we would like to test other integrators and larger molecules like BPTI (we briefly explored this test) and alanine dipeptide. Also, the MDSimLab domain-specific language design is not yet complete. We hope to provide users with a simple way to add, modify, and use their own integrators through this language. Furthermore, we hope to use interactive plotting
Fig. 2. Series of graphs plotted interactively at steps 1, 3 and 5 of Leapfrog Integration on four-atom butane. Plots show system potential energy versus timestep number.

for other simulation data besides potential energy. Finally, this project has opened the door for new projects relating to DSLs and ProtoMol, such as VMD interaction tools, DSL-to-Python translators, DSL metric studies, and a generic automated SWIG wrapping tool [8].

References