INTERACTING DOMAIN-SPECIFIC LANGUAGES WITH BIOLOGICAL PROBLEM SOLVING ENVIRONMENTS

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Trevor M. Cickovski

______________________________

Jesús A. Izaguirre, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
April 2008
INTERACTING DOMAIN-SPECIFIC LANGUAGES WITH BIOLOGICAL
PROBLEM SOLVING ENVIRONMENTS

Abstract

by

Trevor M. Cickovski

Iteratively developing a biological model and verifying results with lab observations has become standard practice in computational biology. This process is currently facilitated by biological Problem Solving Environments (PSEs), multi-tiered and modular software frameworks which traditionally consist of two layers: a computational layer written in a high level language using design patterns, and a user interface layer which hides its details. Although PSEs have proven effective, they still enforce some communication overhead between biologists refining their models through repeated comparison with experimental observations in vitro or in vivo, and programmers actually implementing model extensions and modifications within the computational layer.

I illustrate the use of biological Domain-Specific Languages (DSLs) as a middle-level PSE tier to ameliorate this problem by providing experimentalists with the ability to iteratively test and develop their models using a higher degree of expressive power compared to a graphical interface, while saving the requirement of general purpose programming knowledge. I develop two radically different biological DSLs: XML-based BioLogo will model biological morphogenesis using a cell-centered stochastic cellular automaton and translate into C++ modules for an object-oriented
PSE CompuCell3D, and MDLab will provide a set of high-level Python libraries for running molecular dynamics simulations, using wrapped functionality from the C++ PSE ProtoMol. I describe each language in detail, including its roles within the larger PSE and its expressibility in terms of representable phenomena, and a discussion of observations from users of the languages. Moreover I will use these studies to draw general conclusions about biological DSL development, including dependencies upon the goals of the corresponding PSE, strategies, and tradeoffs.
This thesis is dedicated to my parents and Lilly for all their love and support. However, I also would like to make the following acknowledgments. Special thanks to my advisor Dr. Jesús Izaguirre for his outstanding mentorship, and to my committee, Dr. Greg Madey, Dr. Pat Flynn and Dr. Mark Alber. I also am thankful for the collaborations with the Bioinformatics group at Indiana University and the LCLS at Notre Dame, and particularly Dr. Chris Sweet for providing me with top-notch education in both Math 999 and Jokes 101. Someday I’ll be able to pass the second. Finally, thanks to Scott, Margaret, Jen, and Nice Save! for the friendships and unforgettable times for which I will always be grateful.
APPENDIX G: OTHER MATHEMATICAL MODELS . . . . . . . . . . . . . 197
  G.1 Basic Cell Sorting Type Automaton . . . . . . . . . . . . . . . . . . 197
  G.2 Schnakenberg PDEs . . . . . . . . . . . . . . . . . . . . . . . . . . 200

APPENDIX H: BIOLOGICAL TERMS: MOLECULAR DYNAMICS . . . . . 203

APPENDIX I: MDL API DESCRIPTION . . . . . . . . . . . . . . . . . . . 205
  I.1 Package src . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 206
    I.1.1 Modules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 206
  I.2 Package src.factories . . . . . . . . . . . . . . . . . . . . . . . . . 207
    I.2.1 Modules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 207
  I.3 Module src.factories.ForceFactory . . . . . . . . . . . . . . . . . . 208
    I.3.1 Class ForceFactory . . . . . . . . . . . . . . . . . . . . . . . . 208
  I.4 Module src.factories.PropagatorFactory . . . . . . . . . . . . . . . 215
    I.4.1 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 215
    I.4.2 Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 215
    I.4.3 Class PropagatorFactory . . . . . . . . . . . . . . . . . . . . . 215
  I.5 Package src.forces . . . . . . . . . . . . . . . . . . . . . . . . . . . 220
  I.5.1 Modules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 220
  I.6 Module src.forces.HDForce . . . . . . . . . . . . . . . . . . . . . . . 221
    I.6.1 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 221
    I.6.2 Class HDForce . . . . . . . . . . . . . . . . . . . . . . . . . . 221
  I.7 Package src.modifiers . . . . . . . . . . . . . . . . . . . . . . . . . 222
  I.7.1 Modules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 222
  I.8 Module src.modifiers.averagePositions . . . . . . . . . . . . . . . . 223
    I.8.1 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 223
    I.8.2 Class ReducedHessAngleList . . . . . . . . . . . . . . . . . . 223
  I.9 Module src.modifiers.friction . . . . . . . . . . . . . . . . . . . . . 225
    I.9.1 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 225
  I.10 Module src.modifiers.mollify . . . . . . . . . . . . . . . . . . . . . 226
    I.10.1 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 226
  I.11 Package src.propagators . . . . . . . . . . . . . . . . . . . . . . . . 227
    I.11.1 Modules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 227
  I.12 Package src.propagators.methods . . . . . . . . . . . . . . . . . . . 228
    I.12.1 Modules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 228
  I.13 Module src.propagators.methods.bbk . . . . . . . . . . . . . . . . . 229
    I.13.1 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 229
    I.13.2 Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 229
  I.14 Module src.propagators.methods.impulse . . . . . . . . . . . . . . . 230
    I.14.1 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 230
    I.14.2 Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 230
  I.15 Module src.propagators.methods.leapfrog . . . . . . . . . . . . . . . 231
    I.15.1 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 231
    I.15.2 Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 231
  I.16 Module src.propagators.methods.takahashi . . . . . . . . . . . . . . 232
    I.16.1 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 232
1.1 Two-tier design of a PSE. The experimentalist interacts with the software through Tier 2, the user layer, usually through graphical tools for observation and analysis. Computational models are implemented within Tier 1, the computational layer, typically in a general purpose programming language. Tier communication in a PSE is always top down, meaning that any subset of lower level tiers can be executed standalone without requiring any of the higher level tiers. In this case, the user layer submits commands and invokes accessor methods to retrieve information from the computational back end.

1.2 Three-tier design of a PSE, with a domain-specific language as a middle-level tier. Users now have the option of interacting through the more graphical user layer (now tier 3), or to gain more flexibility and expressive power by interacting through the domain-specific language (tier 2).

1.3 Modified control flow diagram for the plugin pattern, with a DSL generating plugin extensions for the computational back end, dynamically loadable through a runtime user specification.

1.4 Control flow diagram illustrating selective precompilation, which offers a continuous process of PSE extension and execution.

2.1 The CPM grid showing cells and ECM. The shading denotes the cell type. Different cells (for example cells 1 and 3) may have the same type. A site $S$ connects up to fourth-neighbor pixels ($N_1, \ldots, N_4$). From [37].

2.2 Architecture of CompuCell3D. Tier 1 (the developer layer) contains modules (in blue) to accomplish various tasks and run different mathematical models, such as the CPM (in Potts3D) and superimposed chemical fields (in Field3D), etc. Dynamically loaded plugins (in violet) are the standard method for extending Tier 1. Tier 2, the Python interface, consists of Python modules which interface to precompiled shared object binaries of the computational back end, wrapped with SWIG. Finally, Tier 3 or the Presentation Layer provides a GUI which outputs data in the form of screenshots. From [36].
2.3 Screenshot of the CompuCellPlayer, visualizing results of a simulation in 3D. From [190].

2.4 Architecture of CompuCell3D, with BioLogo. This DSL adds one more tier to the framework, becoming Tier 3 and interacting with Tiers 2 and 1. CompuCellPlayer, the highest level of interaction with the CompuCell3D PSE, is still the uppermost tier.

2.5 Translation of a BioLogo program using embedded Python to a CompuCell3D Python plugin.

2.6 Diagram of translation of a BioLogo program, and execution time control flow of the CompuCell3D PSE.

2.7 Developmental timeline of chick-limb skeletal patterning. Drawings show transverse sections of wing buds. For all panels, proximal is left, distal right, anterior up and posterior down. From [146], with modifications.

2.8 Cell type automaton for an avian limb simulation. A NonCondensing cell becomes Condensing when exposed to an activator concentration $A(x, y, z)$ above a threshold, and a Condensing cell becomes NonCondensing when exposed to an activator concentration below a threshold. From [36], with modifications.

2.9 Avian limb development, visualizing only Condensing cells for clarity. The limb forms at the center of a 3D box of cells, and the limb is surrounded by mostly NonCondensing cells. Formation begins with the humerus after 2300 MCS, followed by the radius and ulna after 3250 MCS and finally digits after 4250 MCS. This agrees with the results of [37]. Note: This is 2D a cross section taken at $y=15$.

2.10 (a) Output of in vitro capillary development with elongated cells ($L = 30$) in the absence of cell adhesion, showing the capillary networks found in [135]. (b) Chemical concentration from the same simulation as (a), visualizing the field added through Biologo. (c) Vascular islands forming due to the enforcement of rounded cells ($L = 10$).

2.11 Useful FiPy terms for representing reaction-diffusion equations. From [60].

2.12 Abstract XMLSerializable interface, defined by individual concrete classes correspond to BioLogo tags.

2.13 Control flow illustrating the error checking steps involved in a BioLogo module from initial translation to execution as a CompuCell3D plugin. Initial error checking is performed syntactically at the XML level (proper tag sequence and balancing using external libraries), followed by expression level checking which involves variable lookups, type checks, etc. Runtime exceptions are caught as typical in C++: either by handling routines in the computational back end or as traps to the operating system.
2.14 Architecture of BioLogo C++ code generator. The abstract class Statement defines an interface from which parsing and generation functionality for all keywords inherit. ........................................ 56

2.15 Results of an August 2007 survey about the ease of incorporating mathematical models into CompuCell3D. ................................. 70

2.16 Results of the same survey in August 2007 of BioLogo preference vs. Python, based on programming experience. ............................... 71

3.1 Diagram showing ProtoMol computational back end modules and their interaction. From [133]. ................................................................. 81

3.2 Architecture of MDL and ProtoMol. Each can be developed independently, using an IDE such as Eclipse. I use SWIG to wrap ProtoMol modules for MDL, producing (dark arrow) a set of pre-compiled shared object binaries and Python wrappers which invoke (clear arrow) functionality within these binaries. MDL commands will either be fed directly into the Python interpreter or invoke the Python wrappers generated by SWIG. Simulation observable data can then be passed to external Python tools for plotting, visualization, etc. ........................................ 84

3.3 Usage of a subversion repository on simtk.org for concurrent access to ProtoMol and MDL, and the process of incorporating precompiled binaries into MDL on a client. ........................................ 85

3.4 Diagram illustrating the control flow of MDL propagator objects, and the points at which modifiers are called. ................................. 119

3.5 Interactive plot of total energy while running an MDL simulation of united-atom butane. ................................................................. 125

3.6 The registry data member of the MDL class PropagatorFactory, populated within its constructor. In this particular case, Leapfrog will be there automatically, while NosePoincare and takahashi will be added through subsequent package scans. Once populated, this data member maps uniquely identified Python strings (used within the call to propagate()), to another dictionary which contains the method for constructing the object or calling the function, the type of the propagator, and a tuple containing alternating parameters and default values if necessary. ........................................ 128

3.7 Instantiation of the outer propagation function of a two-level MTS Impulse-BBK scheme. The registry is indexed with the identifier name of the propagator (Python string), obtaining the constructor, type of the propagator, and a tuple containing parameters and default values (in this case none). Then the number of steps, cyclelength and corresponding force field are appended to its argument list. ................................. 132
3.8 Instantiation of the inner propagator in the two-level Impulse-BBK scheme. Once again, the propagator name is used to index the registry. This time, the function handle is appended to arglist, and thus will eventually be a formal parameter for the invocation to impulse(). The timestep and ForceField are also passed. Since the bbk propagator contains parameters and corresponding default values, user-supplied values are bound to parameters if supplied; if not supplied then defaults are assumed.

4.1 Modified waterfall model for codevelopment of the DSL and computational back end tiers of a PSE. The requirements and design should happen together with open communication lines, and the computational back end should be implemented and tested before the DSL to minimize dependency propagation overhead between the implementations. Finally, both can be maintained together.

4.2 Modified incremental model for codevelopment of a DSL and an interfaced computational back end. Like the waterfall model, global requirements and maintenance can be performed together. For each increment, localizing and assessing dependencies can be performed together, and actual modifications should be distinct sequential stages, with the computational back end coming first.

G.1 Series of images from a cell sorting experiment [184, 186] in the chicken embryo. Epithelial cells are pigmented and dark, and retinal cells are light. The two types of cells are initially randomly distributed (a). Over time the cells cluster into groups by type as a result of differential adhesion, into a tight blob. From [19].

G.2 Results of the cell sorting simulation with CompuCell3D using the BioLogo-generated cell type automaton, starting from a randomly initialized two-cell type aggregate. Overtime the more adhesive Dark cells cluster together in the center of the aggregate, with less adhesive Light cells on the outside.

K.1 Wall clock execution times of the parallelized averaging stage of FTSM with various processor counts.

K.2 Overall wall clock execution times of parallel FTSM.

K.3 Our initial diagonal string between two energetically favorable conformations of alanine dipeptide, and our finite temperature string after 64 updates. To illustrate convergence, we superimpose our results on those of [167] (dotted curve).
TABLES

2.1 SELECTED COMPUCELL3D PLUGINS. I GROUP PLUGINS INTO TWO SETS: A HIGHER-LEVEL SET OPERATES ON CHEMICAL FIELDS (LISTED FIRST), AND A LOWER-LEVEL SET WHICH IMPLEMENTS CELL PROPERTIES AND BEHAVIORS. . . . . . . 74

2.2 MY EXTENSIONS AND THEIR CODE SIZE SAVINGS. . . . . . . 75

2.3 FILE AND C++ CLASS COUNTS FOR EXTENSIONS. . . . . . . 75

3.1 WALL CLOCK EXECUTION TIMES (S) OF PROPAGATION SCHEMES USING VARYING DEGREES OF PRECOMPILATION. . . . . . . 150

3.2 WALL CLOCK EXECUTION TIMES (S) OF FORCE ALGORITHMS USING VARYING DEGREES OF PRECOMPILATION. . . . . . . 150

L.1 PROPAGATOR KEYWORDS, PARAMETERS AND DEFAULTS. . 305

M.1 PARAMETERS, DATATYPES AND DEFAULT VALUES FOR FORCE COMPUTATION. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 307

N.1 PROTOMOL FILES AND CORRESPONDING MDL MODULES. . 309
ACKNOWLEDGMENTS

- I acknowledge the following individuals for contributions to the BioLogo and CompuCell3D projects: Dr. Jesús Izaguirre, Dr. Maciej Swat, Dr. James A. Glazier, Dr. Chengbang Huang, Dr. Mark Alber, Dr. Stuart A. Newman, Dr. George Hentschel, Dr. Tilmann Glimm, Dr. Roeland Merks, Dr. Rajiv Chaturvedi, Mr. Kedar Aras, Mr. Troy Raeder, and Mr. Brian Bien.

- I acknowledge the following contributors to the MDLab and ProtoMol projects: Dr. Jesús Izaguirre, Dr. Chris Sweet, Dr. Thierry Matthey, Dr. Paul Brenner, Dr. Scott Hampton, Mr. Santanu Chatterjee, Mr. Dylan Brandtner, and Mr. Chun Bong Benjamin Yau.

- I acknowledge support from the Center for Applied Mathematics at the University of Notre Dame, and NSF grants IBN-0083653, IBN-0313730, and ACI-0135195.

"When the power of love overcomes the love of power the world will know peace.” - Jimi Hendrix
CHAPTER 1

INTRODUCTION

Recent experience illustrates the significance of mathematical models in representing large-scale and potentially complex biological phenomena. For example, the stochastic, lattice-based Cellular Potts Model (CPM) of Glazier and Graner [76] can reproduce observed developmental behavior such as limb bud growth in the chicken [37], multiple life cycles (i.e. migration, thermotaxis, and phototaxis) of the slime mould *Dictyostelium Discoideum* [129], and even capillary formation of Human Umbilical Vascular Endothelial Cells (HUVEC, [135]). The success of this model often involved coupling to external partial differential equation (PDE) solvers for reaction-diffusion equations [82, 68], following the idea of Turing [202] who boldly claimed reaction-diffusion to be expressible by a set of PDEs, which when solved produce instability patterns which form the basis for morphogenesis, a stage of cellular pattern formation in early embryonic development. Mathematical representation of natural processes can actually be traced back to the laws of physics and Sir Isaac Newton. Obtaining a mathematical model and implementing it computationally (a computational model) is beneficial to understanding the mechanisms of life using *in silico* techniques instead of *in vitro* or *in vivo* experimentation, although the model will require verification through these latter schemes. By running these computational models as simulations, results are obtainable to varying degrees of accuracy at abnormal conditions on materials and biological systems not yet made [64].
The concept of using Problem Solving Environments (PSEs) as simulation tools was proposed by Gallopoulos et al. [67] in 1994. They suggested PSEs as essentially highly domain-specific virtual machines, encapsulating models for potentially complex phenomena "behind the scenes" and providing adequate mechanisms for results analysis. These models can be controlled, combined or parameterized without low-level implementation knowledge. Twelve years later, Merks et al. [137] provided a refined set of guidelines to developing a biological morphogenesis PSE, which are in principle applicable to other domains within biology:

1. Implement behaviors of single modules, corresponding to biological entities such as cells or organs.
2. Provide the infrastructure for interaction of these modules.
3. Offer necessary tools for setting up simulations on parallel architectures transparently, and visualize and analyze the resulting structures.

Clearly, PSEs facilitate computational modeling. Beyond containing potentially complex mathematical model implementations for various biological phenomena (point (1)), through a solid infrastructure for entity interaction (point (2)) a biologist could view collective behavior effects resulting from individual entity behavioral changes. This is facilitated by point (3). Recent examples of biological PSEs [137, 37] illustrate the importance and merits of a **multi-tiered** design, which divides PSE functionality into layers designated by abstraction level and flexibility, which are inverse properties. The **computational back end** is the lowest tier and can be developed using different design strategies, for example performance-enhancing algorithms or design patterns for extensibility. The **user layer** is another tier, containing user interaction and data analysis tools such as graphical interfaces and plotting libraries as shown in Figure 1.1. Although Merks et al. [137] used additional tiers such as web servers, clients and lab tools, the computational back
end and user layer are the two minimal necessary tiers to accomplish the above guidelines for PSEs.

Figure 1.1. Two-tier design of a PSE. The experimentalist interacts with the software through Tier 2, the user layer, usually through graphical tools for observation and analysis. Computational models are implemented within Tier 1, the computational layer, typically in a general purpose programming language. Tier communication in a PSE is always top down, meaning that any subset of lower level tiers can be executed standalone without requiring any of the higher level tiers. In this case, the user layer submits commands and invokes accessor methods to retrieve information from the computational back end.

PSEs also provide collaboration tools between experimentalists and programmers regarding the addition and modification of biological models. An experimentalist provides a model description to a programmer, the programmer implements the model in Tier 1 which if carefully designed should not take long, and the experimentalist runs the new framework alongside lab experiments, comparing results. Subsequent model extensions and refinements can be submitted to the programmer,
repeating the process. The root cause of this overhead from repeated communication between experimentalist and programmer is caused by each being constrained to their appropriate tier according to their domains of expertise. Ideally however, the PSE could be brought to the lab bench by providing experimentalists the ability to test and update models alongside verification experiments \textit{in vitro} or \textit{in vivo}, avoiding this communication overhead.

We could accomplish this goal by providing a fourth guideline for PSE design: \textit{Offering necessary tools for a user to add or change functionality of implemented models}. In this thesis I explore biological domain-specific languages (DSLs) which fill this void as another PSE tier, shown in Figure 1.2. As before, the experimentalist can submit commands through graphical tools in the user layer (now tier 3), but also gains the option of prototyping models through the DSL, which provides more low level control than a graphical interface while still facilitating reasoning and operation in their domain of expertise. The DSL can interact with the computational back end in various ways, which I explore in this thesis. The user layer can also generate DSL modules, but I leave this for future exploration.

DSLs solve the root cause of this overhead, namely the lack of a natural vocabulary for expressing complex phenomena fundamental to the biology domain [26]. As put by van Deursen \textit{et al.} [204], a DSL is \textit{"a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain"}. We have seen such examples in domains such as context-free grammar development with Lex [116] and Yacc [102], equation and matrix solving with Matlab [55] and Mathematica [131], and even in the music industry with Csound [42]. These and many other DSLs include tools such as compilers, interpreters and IDEs and are not part of a multi-tiered PSE. This is also true for many biological DSLs like the Cell
Figure 1.2. Three-tier design of a PSE, with a domain-specific language as a middle-level tier. Users now have the option of interacting through the more graphical user layer (now tier 3), or to gain more flexibility and expressive power by interacting through the domain-specific language (tier 2).

Programming Language [2] and the Molecular Modeling Toolkit (MMTK, [85]).

As a PSE tier, a biological DSL would need to provide functionality analogous to Visual Basic for Applications (VBA) within Microsoft Excel, which can represent phenomena difficult to express with Excel itself. CellML [43] started this ball rolling in biology by providing a generic and extensible modeling language for which people can write Application Programmer Interfaces (APIs) to interface to their own PSEs. This interface between the DSL and the computational back end can take multiple approaches. I specifically compare and contrast two techniques for this interface:
plugin generation and selective precompilation. Each of these capitalize on the fact that often when interfacing a DSL to a computational back end, only a small subset of functionality needs to be prototypable in the DSL, and the rest can be considered core or predefined.

The Plugin [63] software design pattern facilitates separation of core and optional functionality, by encapsulating the latter within plugin libraries which are compiled and linked separately from the computational core, having machine code representation as shared objects and dynamically loaded at runtime during a simulation. Dynamic loading of a subset of plugin libraries is often triggered through user input, such as a configuration file reference, as shown in Figure 1.3. If the computational back end makes use of the plugin pattern, a DSL can subsequently be a plugin generator, with mathematical models expressed in the DSL translated to plugin source code. DSL-generated plugins can in turn be dynamically loaded in similar fashion. This approach offers a DSL the ability to extend a computational back end with minimal to no explicit performance penalty versus hand-generated plugin extensions. Second, the translation process although typically non-trivial becomes localized to a subset of computational back end functionality without modifying a computational core, or necessitating the addition of new hooks for DSL compatibility. Finally, extensions fully tested and debugged become readily available in the PSE for future simulations, requiring no further translation and recompilation.

Plugin generation thus offers a discrete extension and execution algorithm which involves language translation, plugin compilation, and dynamic loading. Selective precompilation takes a different approach, merging PSE extension and execution as a continuous process through the DSL by invoking a subset of precompiled computational back end functionality. This process is shown in Figure 1.4. The DSL provides a high-level and domain-specific API for experimentalist interaction, which
invokes more complex low-level functionality, represented either as helper routines or precompiled libraries from the computational back end. To achieve performance, selective precompilation can take advantage of the well known 90/10 rule in software development, which states that 90% of execution time is spent in 10% of code. If this 90% is precompiled, even prototyping analogously large portions of computational back end tasks in a DSL with a sizable performance decline will not significantly reduce collective runtime performance. Thus selective precompilation involves dividing back end functionality into two subsets. The first contains functionality which
should not be prototypable in the DSL, which will typically include (1) computational core functionality, (2) computationally intensive tasks and (3) utility libraries such as I/O modules. The second set includes the types of methods that should be prototypable in the DSL. This functionality should not be precompiled unless some methods have been created already in the back end, which can then be predefined in the DSL. With selective precompilation the computational back end need not use the plugin design pattern or even general design patterns for that matter, although they would be helpful in terms of modularization since the granularity of control offered by the DSL is proportional to division of functionality in the computational back end.

![Figure 1.4. Control flow diagram illustrating selective precompilation, which offers a continuous process of PSE extension and execution.](image)

The two techniques are radically different and offer distinct benefits and trade-offs, some of which they share but others they complement. I will outline advantages and disadvantages in this thesis, and conclude with a guideline to biological DSL developers describing when each technique is appropriate. There are a large num-
ber of considerations, including software design modularity, division of functionality, information hiding, error handling, performance, memory, etc. These are coupled to experimentalist goals with respect to prototypable models in the DSL. DSLs are unique with respect to this latter goal in that widespread expressive power is generally not a requirement, but writability with respect to a smaller set of operations is usually the main concern for experimentalists. Ease of use also must occupy a high priority. I analyze these techniques through case studies by designing two DSLs and interfacing them to developed PSEs. These include (associated technique and PSE in parentheses):

1. BioLogo (Plugin generation, CompuCell3D): The CompuCell3D PSE runs 3D morphogenesis [105] simulations using a lattice-based cellular automaton [5] coupled with PDE solvers and cell type automata as its core mathematical models. BioLogo generates CompuCell3D extensions for (1) energy terms which drive the cellular automaton, and (2) specific types of PDE solvers and cell type automata as plugins which follow CompuCell3D design patterns and have little to no explicit performance penalty. BioLogo is XML-based, with some embedded Python for enhanced expressive power.

2. MDLab (Selective precompilation, ProtoMol): ProtoMol is an object-oriented, modular C++ PSE which conducts molecular dynamics (MD) simulations. MDLab builds on Python and executes commands within a standard Python interpreter or PyMPI [163] for parallel simulations, providing a scripting interface to MD. This is accomplished through a domain-specific API which through a non-trivial translation instantiates objects and invokes functionality of the ProtoMol back end wrapped as precompiled shared objects using SWIG [17], in addition to some less computationally intensive tasks in pure Python. In this way MDLab is a useful prototyping, testing and debugging tool for MD numerical methods and simulation protocols.

For each application, I begin with a domain description and a set of mathematical models which I desire to prototype. I then present each PSE design with the addition of the DSL and method for applying the associated interface technique. I then illustrate each DSL’s ability to prototype biologically relevant models, and finally provide an evaluation of each DSL from design and user perspectives. In my conclusions, I use these results to evaluate the two techniques and provide a sum-
mary of situations where each are appropriate, along with advantages, disadvantages and tradeoffs.

1.1 Contributions of This Thesis

First and foremost, I intend this thesis to be a future DSL developer reference, particularly for those interested in biological PSEs and using a DSL as a middle-level tier to facilitate mathematical model prototyping. After reading this thesis, a developer should have a clear understanding of the techniques of plugin generation and selective precompilation, along with situations where each are applicable and appropriate tradeoffs. Their usefulness will be demonstrated through their applications to relevant and currently researched biological processes, and these principles and practices can be followed on future applications. Moreover I will use case studies to develop software engineering strategies for codevelopment of the DSL and computational back end tiers.

As a side benefit, interaction flexibility of two PSEs is increased through the development and interfacing of these two DSLs. I illustrate this through several examples mathematical models expressible in a higher level and simpler fashion in the DSLs compared to their analogous representation in a general purpose language (GPL). Experimentalists will subsequently be able to extend each PSE with new models through greater expressive power than available in the user layer, without venturing outside their domain of expertise by modifying source code in the computational back end or entailing communication overhead with a software developer or programmer. In this way domain experts will be able to express themselves and most importantly their mathematical models using appropriate constructs in each DSL.
CHAPTER 2

BioLogo, A DOMAIN-SPECIFIC LANGUAGE FOR MORPHOGENESIS

2.1 Introduction

I illustrate plugin generation by constructing an XML-based DSL BioLogo for morphogenesis which interacts with a PSE CompuCell3D. Morphogenesis is a phase in embryonic development of a multicellular organism where cells cluster into patterns that later differentiate into muscle, bone and organs. During morphogenesis, cells undergo various behavioral mechanisms such as collective or independent movement, division, growth, and adhesion. Cells can also secrete or resorb chemicals or follow chemical gradients, and also differentiate into behavior-specialized cell types. I begin by outlining mathematical models for morphogenesis and describing their representation in BioLogo, and describe the translation process from BioLogo XML modules to CompuCell3D plugins. I test BioLogo using biologically relevant verification simulations, and conclude with a discussion of observations from design and user perspectives, including surveys taken at a CompuCell3D workshop.

2.2 Mathematical Morphogenesis Models

A particularly interesting aspect of nature that continues to be researched is the development of a complex multicellular organism from a unicellular zygote. After rapid mitotic divisions, the fertilized zygote forms a multicellular morula cluster.
of cells which after further cleavage forms a blastula. Morphogenesis occurs after formation of the blastula, involving cellular aggregation into patterns that spatiotemporally differentiate into organs, muscle, or bone. A particular challenge in post-genomic biology is relating morphogenesis to the expression of gene products by cells, specifically studying how these gene products drive morphogenetic subprocesses such as differentiation, growth, mitosis, or secretion of extracellular materials.

One method of creating this deeper understanding is to develop mathematical models for these subprocesses which can be iteratively refined and verified through comparison with lab experiments. Merks and Glazier [136] argue for a cell-centered approach to modeling morphogenesis, with individual cells as the lowest level of abstraction, studying the collective influence of individual cellular behaviors on macroscopic behavior such as tissues and organisms and avoiding the computationally intractable approach of modeling $10^5 \sim 10^6$ gene products. By iteratively refining such a model to a minimal set of behaviors which generate behavior corresponding to experimental results, gene networks which drive such behaviors can be inferred. I now give background on some submodels which follow the cell-centered approach and can be integrated for collective modeling of morphogenesis.

2.2.1 Turing Reaction-Diffusion PDEs

Reaction-diffusion (RD) is a process by which multiple substances react with one another and become chemically converted, and also diffuse through space. The idea of using a partial differential equation (PDE) to model RD began in 1937 with the Komogorov-Petrovsky-Piscounov equation [111], temporally modeling one substance in one dimension:

$$\frac{\partial u}{\partial t} = R(u) + D \nabla^2 u,$$  \hspace{1cm} (2.1)
where $\frac{\partial u}{\partial t}$ represents the changes in concentration of substance $u$ with time $t$, $R(u)$ the local change in concentration of $u$ (reaction) and $D \nabla^2 u$ the diffusion term with $D$ governing the diffusion rate of $u$. Turing would extend this idea to multiple reacting substances in 1952 [202], substituting $u$ in the above equation with a vector of concentrations $\mathbf{u}$, with $D$ now becoming a diagonal matrix of diffusion constants and $R$ a corresponding vector of functions of all substance concentrations. In this same study, Turing suggested that this coupled RD of multiple chemicals is sufficient to account for morphogenesis in embryonic development, through concentration patterns arising from system instabilities upon application of specific parameter sets. Twenty years later, Gierer and Mienhardt identified an activator molecule along with an inhibitor which suppresses this activator as the key components of Turing’s model [72].

Although Turing’s propositions remained hidden for a long time, scientists in the late twentieth century began observing these so called ”Turing patterns” within chemical mixtures [46] and aqueous media [114]. Today, Turing models are still researched, having been shown to reproduce convincingly observed patterns such as leopard and jaguar spots [122], butterfly wing stripes [175], and even semi-arid vegetation [41].

2.2.2 Cellular Potts Model

The Cellular Potts Model (CPM) [76] is a lattice-based model for cellular dynamics, and can model morphogenetic events such as cell clustering, volume and surface area constraints, and chemotaxis (or haptotaxis, cell-type dependent motion) towards a chemical gradient. The CPM uses a lattice to describe cells and associates an integer index with each lattice site (voxel) to identify the spatial extent and location of each cell at any instant. The index value at a lattice site is $\sigma$ if
the site lies in cell $\sigma$. Domains in the lattice (the collection of lattice sites with the same index) represent cells, as shown in Figure 2.1.

$$E_{CM} = 0 \quad N_4 N_3 N_4 \quad N_4 N_2 N_1 N_2 N_4 \quad N_3 N_1 S N_1 N_3 \quad N_4 N_2 N_1 N_2 N_4 \quad N_4 N_3 N_4$$

Figure 2.1. The CPM grid showing cells and ECM. The shading denotes the cell type. Different cells (for example cells 1 and 3) may have the same type. A site $S$ connects up to fourth-neighbor pixels ($N_1, \ldots, N_4$). From [37].

A cell is thus a set of discrete components that can rearrange to produce cell shape changes and motion. Instead of representing the forces which cause cells to rearrange directly, the CPM aggregates them into an effective energy $E$ (containing terms corresponding to real and ‘mimicked’ energies), the gradient of which is the force acting at any point. Eq. (2.2) shows a typical effective energy $E$.

$$E = E_{Contact} + E_{Volume} + E_{Surface} + E_{Chemical}. \quad (2.2)$$

The first term, $E_{Contact}$, is shown in Eq. (2.3) and describes the net adhesion/repulsion of two cells $\sigma$ and $\sigma'$. 

14
$E_{\text{Contact}} = \sum_{(i,j,k), (i',j',k') \text{neighbors}} J_{\tau(\sigma), \tau'(\sigma')} (1 - \delta(\sigma(i,j,k), \sigma'(i',j',k')))$, \hspace{1cm} (2.3)

where $J_{\tau,\tau'}$ represents the binding energy per unit area of the cell and depends on the types of interacting cells. The Kronecker delta $\delta(\sigma, \sigma') = 0$ if $\sigma \neq \sigma'$ and $\delta(\sigma, \sigma') = 1$ if $\sigma = \sigma'$, ensuring that only two unique cells can contribute to this effective energy term. Terms 2 and 3 in Eq. (2.2) respectively represent volume and surface area fluctuations in the cell membrane. A cell of a given type $\tau$ is assumed to have an associated target volume $v(\sigma, \tau)$ and target surface area $s(\sigma, \tau)$ around which it fluctuates, obtainable experimentally by fixing CPM lattice point and domain length ratios. These two terms enforce an energy penalty for deviations from these target values. $E_{\text{Volume}}$ depends on two parameters: volume elasticity, $\lambda$, and target volume, $v_{\text{target}}(\sigma, \tau)$. Analogous to this, $E_{\text{Surface}}$ depends on membrane elasticity, $\lambda'$, and target surface area $s_{\text{target}}(\sigma, \tau)$:

$$E_{\text{Volume}} = \sum_{\text{cells}} \lambda \sigma (v(\sigma, \tau) - v_{\text{target}}(\sigma, \tau))^2$$
$$E_{\text{Surface}} = \sum_{\text{cells}} \lambda' \sigma (s(\sigma, \tau) - s_{\text{target}}(\sigma, \tau))^2.$$ \hspace{1cm} (2.4)

These terms also allow modeling of cell growth by gradually increasing these target values with time, cell division by growing a cell to a "doubling volume" equal to $2v_{\text{target}}$ and splitting the dividing cell into two cells each with target volume $v_{\text{target}}$, and cell death by setting these targets to zero.

The final term models cellular responses to a diffusible or substrate-bound external chemical signal. Cellular chemotaxis involves motion along gradients of diffusible chemical signals, and haptotaxis involves motion along gradients of substrate-bound chemical signals. In Eq. (2.5) which represents this final term, $C(x, y, z)$ is the local concentration of the morphogen molecule in extracellular space and $\mu(\sigma)$ is an
effective chemical potential which governs strength of chemotaxis or haptotaxis:

\[ E_{\text{Chemical}} = \sum_{x,y,z} \mu(\sigma(x, y, z))C(x, y, z). \]  

(2.5)

For chemotaxis, the chemical field \( C \) must be superimposed on the CPM lattice and also interpenetrate the cells, simplifications shown to be valid if movement is slow compared to the rate of diffusion (the case in most simulations of developmental biology) [126, 129, 170, 130, 136, 138, 89, 88, 87, 127, 128]. Solving the afore-mentioned reaction-diffusion PDEs can evolve the chemical field, although the spatial and temporal scaling may vary from the CPM lattice.

2.2.3 Cell Type Automata

During morphogenesis, cells have been observed to undergo *differentiation* from initial multipotent stem cells into specialized cell *types* of the developed organism, where cell type classification identifies broadly similar behaviors between cells and associating them with a specific cell *type*. Cell differentiation from one type to another is in general abrupt and irreversible and involves a comprehensive and qualitative change in cell behavior. Biologically, this could correspond to turning on or off genetic pathways [106]. In addition, cells of the same type can exist in different *states*, which correspond to a set of associated parameter values for a cell. States allow for finer-grained classification of cell behavior; if two cells possess the same type and state, their behavior would be equivalent in the same external environment.

One can model differentiation using *cell type automata* which are deterministic and finite, with cell *types* representing computational states, and the input alphabet represented by some form of internal or external signalling ([53, 165] specify formal issues involved in creating type automata from continuum and discrete approximations of complex gene regulatory networks). Each type in the automaton
corresponds to a biological cell type which exists during morphogenesis, and successful type transitions correspond to cell differentiation.

2.3 CompuCell3D Design and Interfacing of BioLogo

CompuCell3D is a modular, multi-tiered PSE written in C++ which simulates morphogenesis in three dimensions and is an ongoing project involving the Laboratory for Computational Life Sciences at the University of Notre Dame (headed by Jesus Izaguirre), and the Biocomplexity Institute at Indiana University (http://www.biocomplexity.indiana.edu/). The modular design of the CompuCell3D PSE is shown in Figure 2.2. The Developer Layer consists of the computational back end or simulation engine. The module Potts3D encapsulates the CPM and Field3D manages the CPM lattice; also containing functionality for superimposing chemical fields onto this lattice and enforcing spatial boundary conditions using the Boundary module. The Factory Method ([69], 107) promotes loose coupling between the CPM lattice and boundary strategy [10], and the Automaton module provides generic functionality for type differentiation, using a variation of the State ([69], 305) design pattern. Plugins, shown in violet, are the standard method for extending the simulation engine to encapsulate new models, localized to a particular directory in the framework and dynamically loaded at runtime upon reference in an input configuration file. Table 2.1 lists features available as CompuCell3D plugins, with some having multiple versions targeted to specific applications. Finally, the Simulator invokes functionality of Potts3D and Plugins, to run a complete morphogenesis simulation. The Python Interface provides Python modules which invoke functionality of wrapped precompiled binaries of the modules in the computational back end using the Simplified Wrapper and Interface Generator (SWIG, [17]). The user choice of whether or not to use Python is a runtime
decision, specified in **CompuCellPlayer** (the third tier). If Python is desired, the user can specify at runtime (1) a Python simulation file which includes Python plugin object instantiations, and (2) an XML configuration file containing C++ plugin instantiations. Here I show an example of a Python plugin which adds an attribute to every cell in the simulation (from the **CompuCell3D Manual**, [191]):

```python
class ExtraAttributeCellsort(SteppablePy):
    def __init__(self, _simulator, _frequency=10):
        SteppablePy.__init__(self, _frequency)
        self.simulator = _simulator
        self.inventory = self.simulator.getPotts().getCellInventory()

    def step(self, mcs):
        invItr = CompuCell.STLPyIteratorCINV()
        invItr.initialize(self.inventory.getContainer())
        invItr.setToBegin()
        cell = invItr.getCurrentRef()
        while (1):
            if invItr.isEnd():
                break
            cell = invItr.getCurrentRef()
            pyAttrib = CompuCell.getPyAttrib(cell)
            pyAttrib[0:2] = [cell.id * mcs, cell.id * (mcs - 1)]
            invItr.next()
```

Each simulation which uses Python plugins also requires a Python *simulation file* which invokes back end modules at the appropriate times. I include an example in Appendix C. The Presentation Layer consists of **CompuCellPlayer**, which uses Qt [200] and VTK [110] to interactively visualize **CompuCell3D** simulations in three dimensions as shown in Figure 2.3, and can output data as .png screenshots.

Program 1 shows an example **CompuCell3D** XML configuration file. Parameters such as the dimensions of the CPM lattice, number of Monte Carlo Steps (MCS), temperature and the ratio between lattice points and flip attempts are specified in the **Potts** module. This is followed by specifications of plugins to include in the
Figure 2.2. Architecture of COMPUCELL3D. Tier 1 (the developer layer) contains modules (in blue) to accomplish various tasks and run different mathematical models, such as the CPM (in Potts3D) and superimposed chemical fields (in Field3D), etc. Dynamically loaded plugins (in violet) are the standard method for extending Tier 1. Tier 2, the Python interface, consists of Python modules which interface to precompiled shared object binaries of the computational back end, wrapped with SWIG. Finally, Tier 3 or the Presentation Layer provides a GUI which outputs data in the form of screenshots. From [36].

Simulation along with user-defined parameters which can customize their behavior, called *instantiation* of a plugin. For example, the **Volume** plugin implements the volume constraint energy term of the CPM in Eq. (2.4), thus requiring a target volume for cells and also a scaling factor **LambdaVolume**. Plugin parameters can also be cell-type dependent, for example the **Energy** parameter of the **Contact** plugin represents the variable $J$ in Eq. (2.3), which is a function of a cell’s type and the type of its neighbor. These are passed as XML tag attributes **Type1** and **Type2**.

Although carefully designed with good software practices, COMPUCELL3D as it stands cannot be brought to the lab bench for model development alongside ex-
Program 1: Example COMPUCELL3D configuration file.
Figure 2.3. Screenshot of the CompuCellPlayer, visualizing results of a simulation in 3D. From [190].

experiments. The computational back end is extensible for a programmer to add new model implementations, but testing and debugging becomes difficult as C++ does not cater well, enforcing recompilation upon every change and producing cryptic runtime errors such as segmentation faults. But most importantly, extension also requires knowledge of a general purpose language (C++), including moderately complex concepts like classes, inheritance and design patterns. Adding a new CPM energy term requires minimally three C++ files and maximally five: a header and implementation for the plugin, a header and implementation for an energy calculator, and a proxy [69]. Energy terms coupled to a cell attribute (such as the volume constraint, which requires cell volume updates) require these five. For plu-
gins which only require energy term computation only three are needed, but this is still burdensome. Moreover, appropriate CMake files require modification to encapsulate the new plugin in CompuCell3D compilation and installation scripts. Other extension types similarly require a heavy burden. Adding a new RD PDE solver requires three C++ files plus CMake modifications, and adding a new cell type automaton requires three C++ files plus one file to implement each automaton transition, along with CMake modifications.

CompuCellPlayer is the other extreme, allowing interaction without imposing knowledge of a general-purpose language, other than a domain-specific XML configuration file. The interface is window and menu-based, with available functionality clearly delineated. However, CompuCellPlayer cannot extend or modify CompuCell3D mathematical models, but can only combine and run existing models. The closest tier to achieving this lab bench availability is the Python tier, which provides advantages to prototyping and testing new mathematical methods because of inherent advantages due to interpretation (better error checking, no recompilation required upon changes). The level of abstraction is higher, interface to complex functionality is simpler, and a plugin can be coded as one class. A disadvantage of the Python tier is the lack of a domain-specific API enforcing knowledge of some (SWIG-wrapped) computational structures and their available data members, as well as some general Python glue code which must be written for all extensions. Moreover, using Python plugins requires a Python simulation file which registers and instantiates that plugin, which is a disadvantage versus C++ plugins which only need instantiation.

With BioLogo as a middle-level tier interacting with the computational back end through plugin generation, an experimentalist could express mathematical morphogenesis models using a more natural vocabulary, and this model would trans-
parently generate a C++ plugin and modify compilation scripts such that one PSE recompilation encapsulates the new plugin extension. Testing and debugging becomes easier because the syntax analysis process would catch errors before C++ generation displaying noncryptic and domain-specific messages, and thus generated C++ code should be syntactically bug free. Moreover, once a generated plugin is a part of the CompuCell3D PSE, it can be reused repeatedly by the user without regeneration, and even recommended to CompuCell3D developers for potential inclusion in future PSE releases. Of most importance the experimentalists will now be directly developing and testing their models, as opposed to separate programmer development and experimentalist testing. I develop BioLogo as an XML-based DSL for morphogenesis and a tool for constructing, testing and debugging mathematical morphogenesis models. I add BioLogo as a new tier 3 in the CompuCell3D architecture, as shown in Figure 2.4. Tier 4, the uppermost tier, remains the CompuCellPlayer since this provides the highest abstraction level. BioLogo interacts with the two lower tiers. Although its primary purpose is to generate plugin extensions for the computational back end, to increase expressive power BioLogo supports embedded Python in some XML modules, which can be translated to Python plugins with transparently generated glue code. The extensibility of XML helps in language maintenance.

BioLogo XML code is first translated into a simpler intermediate format using a compiler front end which avoids the requirement of Document Type Definitions (DTDs) and XML schemas by using an external parsing library, Xerces-C++ [9]. Error handling is performed at this stage including type and scope checks, with appropriate error messages generated for the experimentalist. BioLogo assumes a floating point data type for all user-defined variables, and there are some predefined objects with 'special' types (for example, to represent a cell in a CPM energy
Figure 2.4. Architecture of COMPUCELL3D, with BioLogo. This DSL adds one more tier to the framework, becoming Tier 3 and interacting with Tiers 2 and 1. COMPUCELLPLAYER, the highest level of interaction with the COMPUCELL3D PSE, is still the uppermost tier.

The generated intermediate code is simpler and is imperative as opposed to declarative, expressing BioLogo constructs in terms of their operational semantics and simplifying the translation to C++. In the future when BioLogo generates C++ plugins for multiple platforms, the compiler front end will implement machine-independent optimizations [3]. Embedded Python is not translated but passed directly into the new Python plugin, so only the plugin glue code is generated by BioLogo tools. Thus exceptions and error checking for embedded Python are performed at runtime by the Python interpreter when launched by the
COMPUCELL3D application. The compiler back end converts the intermediate code to new C++ (or Python, if applicable) plugins for COMPUCELL3D. The intermediate form is also assumed to be bug free, so the compiler back end only performs translation and no error checking.

2.4 Syntax and Examples

A BIOLOGO program begins with a `<CompuCell3D>` tag and ends with a `</CompuCell3D>` tag, and contains multiple XML blocks which each translate to a new COMPUCELL3D plugin. Some BIOLOGO tags require XML attributes enclosed within double quotes ("). These blocks can represent CPM energy terms, cell type automata and RD PDEs. I now provide examples.

2.4.1 CPM Energy Terms

Representation of a CPM energy term is contained within `<Hamiltonian>` and `</Hamiltonian>` tags. The term Hamiltonian references the CPM Hamiltonian function (or energy function), of the cell and other superimposed lattice configurations. An energy term block contains the syntactic structure shown in Program 2.

```xml
<Hamiltonian name="..." />
... inputs ...
... fields ...
... step module ...
... equation ...
</Hamiltonian>
```

Program 2: Template for a Hamiltonian term in BIOLOGO.

The name attribute corresponds to the name of the generated C++ plugin, and should correspond somehow to the type of energy being computed. Since COMPUCELL3D plugins are customizable, a user can instantiate a plugin through the
configuration file by providing a specific set of input values which affect behavior. Names of these inputs are specified immediately after the opening tag, in the format:

```xml
<Input name="..." />
```

where the specified value for name will be used as a tag in the COMPUCELL3D configuration file to provide corresponding values for this input. For cell type-dependent inputs (such as the $J$ term of Eq. (2.3)), one can pass a params argument which accepts an integer for a cell type count, for example:

```xml
<Input name="..." params="2" />
```

All BioLogo Inputs are assumed to contain floating point values, with the exception of filename inputs which I now cover. BioLogo energy terms allow external coupled chemical field superimposition, which can influence the computed result (for example, chemotaxis models). A chemical field becomes superimposed using Field tags:

```xml
<Field name="..." />
```

In addition, a chemical field can be static or dynamic. A static field is populated when the simulation starts by a binary file containing floating point concentrations with $x$ as the innermost loop, by coupling the field to an Input tag for the filename, i.e.:

```xml
<Input name="..." type="file" />
<Field name="..." filename="same as the name attribute of the above Input" />
```

When the user instantiates this plugin, they provide the binary file name and instantiate the input above, and that file populates the coupled chemical field. If a field is dynamic, it is populated in the Step block, enclosed within `<Step>` and `</Step>` tags and consisting of multiple secrete or resorb tags with the following
format:

\[
<\text{secrete field=\"...\" amount=\"...\" condition=\"...\"} /> \\
<\text{resorb field=\"...\" amount=\"...\" condition=\"...\"} />, \\
\]

where the field attribute contains the name of a field, amount contains an arithmetic expression for the amount of chemical to secrete or resorb, and condition is a boolean condition specifying when to secrete or resorb. BioLogo arithmetic and boolean expressions can use Inputs as variables. Arithmetic operators are the standard +, -, *, /, ^ (power), and parentheses; for boolean operators full words are used: less, greater, lessequal, greaterequal, equal, notequal. Trigonometric functions which accept an arithmetic expression as an argument are available for sin, cos, tan, asin, acos, and atan; as well as sqrt. So for example, assuming we had defined two inputs \( x \) and \( y \) and a field \( c \), to secrete into \( c \) the quantity \( x*y \) for all \( c \) locations with a concentration above 0.5:

\[
<\text{secrete field=\"c\" amount=\"x*y\" condition=\"c greater 0.5\"} /> \\
\]

BioLogo also abstracts functionality for biological cells, which have associated attributes for volume, type, surface, xCM, yCM, zCM. Each of these, with the exception of type, contains integer values. type contains a string. BioLogo string literals are surrounded by single quotes. The cell thus constitutes the one and only object-oriented construct of BioLogo. The variable cell, defined within secrete and resorb tags as the cell located at the point of secretion or resorption, contains a '. ' operator for attribute access. These attributes, in turn, can be used in arithmetic or boolean expressions within the tags. For example, for type A cells to resorb a chemical concentration of 0.4 from \( c \):

\[
<\text{resorb field=\"c\" amount=\"0.4\" condition=\"cell.type equal \text{\char'13\"A\char'13\"}\"} /> \\
\]

Finally, the actual equation for the energy term is implemented within Equation tags and expressed using one or more pixelsum, cellsum, and neighborsum tags.
BioLogo thus makes the assumption that CPM energy terms consist of a summation of some sort, either (1) over all lattice points (a `pixelsum`), (2) over all cells (a `cellsum`), or (3) over all lattice points and their neighbors (a `neighborsum`). The syntax for these three tags is:

```
<pixelsum exp="..." condition="..." />
<cellsum exp="..." condition="..." />
<neighborsum exp="..." condition="..." limit="..."/>
```

`exp` corresponds to an arithmetic expression for the equation and `condition` to an optional boolean expression for counting a value in the sum. These tags also have access to the `cell` variable and its attributes. In a `pixelsum` over all lattice points (call them `x`), the `cell` corresponds to the cell located at `x`. In a `cellsum` over all cells (call them `c`), the `cell` corresponds to the cell `c` being summed over. In a `neighborsum` over all lattice points (call them `x`) and neighboring points (call them `x'`), `cell` corresponds to the cell at `x` and `cell2` to the cell at `x'`. Finally, a `neighborsum` has a Euclidean distance `limit` within which neighbors are considered, in CPM lattice points.

For example, Eq. (2.5) is representable by the `pixelsum` shown in Program 3. The field `c` can be populated through `myFile`, and the Hamiltonian can accept another input for the scaling factor `mu`. The arithmetic expression for the equation is simply `mu*c`, and ECM points (associated with a `Medium` cell type) are excluded from the sum. Note that in `pixelsums` and `neighborsums`, indexing is implicit for fields since the sum is over lattice points.

Once generated, we could then instantiate this plugin in the COMPUCELL3D configuration file using Program 4, providing for example a value of 100.0 for `mu` and chemical.dat for `myFile`, which populates `c`.

To represent a surface area constraint analogous to Eq. (2.4) but with a cell-type
2.4.2 Cell Type Automata

A cell type automaton is represented in BioLogo within a `cellmodel` block with an associated `name` attribute, which corresponds to the name of the generated C++ plugin. Program 7 shows a template for a BioLogo cell type automaton. Each `cellmodel` defines a set of cell `types`, and also a set of `cell state variables` which govern cellular behavior at a finer scale. Cell state variables must be defined at the

---

**Program 3:** Chemotaxis in BioLogo.

```
<Hamiltonian name="Chemotaxis" />
  <Input name="mu" />
  <Input name="myFile" type="file" />
  <Field name="c" filename="myFile" />
  <Equation>
    <pixelsum exp="mu*c" condition="cell.type notequal 'Medium'" />
  </Equation>
</Hamiltonian>
```

**Program 4:** Instantiation of the plugin generated by BioLogo Program 3 in the COMPUCELL3D configuration file.

```
<Plugin name="Chemotaxis" />
  <mu>100.0</mu>
  <myFile>chemical.dat</myFile>
</Plugin>
```

dependent surface area, we could use a `cellsum` as shown in Program 5. Note that we do not need a condition for ECM exclusion, since a `cellsum` by definition sums over cells. We can then instantiate this plugin in a COMPUCELL3D configuration file using Program 6, assuming cell types A and B are available in the simulation.

```
<Hamiltonian name="Surface" />
  <Input name="targetsurface" params="1" />
  <Input name="lambda" />
  <Equation>
    <cellsum exp="lambda*(cell.surface-targetsurface)^2" />
  </Equation>
</Hamiltonian>
```

**Program 5:** Surface area constraint in BioLogo.

---
Program 6: Instantiation of the plugin generated by BioLogo Program 5 in the CompuCell3D configuration file.

beginning of a cellmodel definition. Each cell type contains a creation block which initializes cell state variables when a cell becomes this type, a updatevariables block which updates cell state variables at every MCS, and a updatecelltypes block which defines type transitions in the automaton. Each of these blocks is optional; for example if cell state should remain static for a particular cell type, that type may only include the updatecelltypes block.

Program 7: Template of a BioLogo cell type automaton.

Cell state variables are declared within a declare block using statevar tags, with attributes for the variable name and an optional value. State variables are assumed to contain floating point data:

<declare>
  <statevar name="..." value="..." />
  ....
</declare>
Upon declaration, the state variables can be used in arithmetic and boolean expressions within the cell type automaton. The creation block contains multiple BioLogo copy statements of the following syntax:

```xml
<copy name="..." value="..." condition="..." />
```

where name is a cell state variable and value is an arithmetic expression which evaluates to a floating point value. The condition (optional) is a boolean expression which must be true for the binding to occur. These expressions can contain other cell state variables, or intrinsic cell attributes such as volume, surface, etc. However, intrinsic attributes are immutable within the automaton definition. updatevariables also contains zero or more copy statements, in similar fashion. Finally, the updatecelltypes block contains zero or more BioLogo changeif statements:

```xml
<changeif currenttype="..." condition="..." />
```

Each of these tags corresponds to a type transition where the destination cell type is the encompassing cell type. The source cell type is contained within the attribute currenttype and must be a valid cell type, transitioning to the destination type if the boolean condition attribute is true. Clearly, it is possible to design a non-deterministic type automaton by defining multiple changeif clauses containing simultaneously true conditions, with the same source cell type. BioLogo enforces determinism by prioritizing destination type by order of appearance in the cellmodel. Thus transitions to the first defined cell type would have priority over the second defined cell type, and so on. Also, the first defined cell type is assumed to be the initial type of all cells.

### 2.4.3 Reaction-Diffusion Partial Differential Equation Solvers

A reaction-diffusion (RD) PDE system can be represented using Eq. (2.1), extended for coupled chemicals using a vector of concentrations \( \vec{u} \), a reaction term
which is a vector of functions of the concentrations, and a diagonal matrix $D$:

$$
\frac{\partial \vec{u}}{\partial t} = \vec{R}(\vec{u}) + D \Delta^2 \vec{u},
$$

(2.6)

A BioLogo PDE system can be implemented using a PDESolver. The corresponding generated plugin will solve the system of equations using the finite difference method which was shown to be stable for solving RD systems by [120], and assumes scalar boundary conditions with respect to time and a resulting unique solution, as Turing did in his original model [202]. The generated solver enforces periodic boundary conditions with respect to the lattice. Mixed boundary conditions [49] were later shown to produce different phenomena and potentially multiple solutions, which could be a potential future extension to BioLogo.

The template for a PDESolver is shown in Program (8). A PDESolver tag contains attributes for its name which corresponds to the generated plugin name, and normalize (optional) which contains either true (default) or false. If normalize is true, chemical concentrations are normalized to real values between zero and one.

```xml
<PDESolver name="..." normalize="..." />
... inputs ...
... fields ...
... equations ...
</PDESolver
```

Program 8: Template of a BioLogo PDE solver.

As with Hamiltonians, generated BioLogo PDE solver plugins are customizable, meaning users can instantiate them with input values. The syntax of inputs and fields are the same as with Hamiltonians, except that there is no file-field coupling since chemical fields coupled to for PDE solvers are assumed to be evolved by the solver itself. Once again, inputs are assumed to be bound to floating point values, and chemical fields are assumed to contain floating point concentrations.
Each PDE is defined within a **DiffEq** block, with a template shown in Program 9. The **fieldname** attribute contains the field that this equation temporally evolves. Each equation can have multiple **terms**, of the form:

```
<Term exp="..." condition="..." />
```

where **exp** is an arithmetic expression and **condition** is a boolean expression (optional), providing a condition for inclusion. Multiple **Term** tags may be necessary if some PDE terms should be included under different conditions than others. Defined **Inputs** and **Fields** can be used in these expressions, as well as a predefined variable **Kronecker** for the Kronecker delta, which is bound to the value 0 for the ECM lattice points and 1 for cell lattice points, and also a function **Laplacian(f)**, which in Turing’s general format corresponds to \( \nabla^2 f \) computing the gradient of a defined field \( f \) twice (used in the diffusion term).

```
<DiffEq fieldname="..." />
... terms ...
</DiffEq>
```

Program 9: Template of a BioLogo partial differential equation.

In addition, there are also some implicitly defined inputs for the timestep (\( DT \)), the spatial discretization (\( DX \)), and the ratio between steps of finite difference and MCS (\( Steps \)), which the user must provide at instantiation time and can also be used within expressions. Turing systems expressed with pure XML translate to a relatively fast C++ PDE solver, but there are some limitations, particularly in terms of expressive power. For example the only mathematical functionality available is **Kronecker** and **Laplacian**, limiting its applicability to more trivial Turing systems. Moreover, many models of this sort [126, 82] use auxiliary functions which return different values depending on concentration at a particular point, for instance. This is not possible using strictly the current XML capabilities. To provide this ability (with some performance sacrifice), I offer the option of embedding Python into a
PDESolver as an alternative to DiffEq tags, following the template shown in Program 10 using a Python block. Instead of generating a C++ plugin, a Python plugin is generated, shown pictorially in Figure 2.5. Python bindings are established for fields and both explicitly and implicitly defined inputs so they can be used in the embedded Python code. Inputs are bound to floating point values provided at instantiation time and the fields are bound to Python one-dimensional arrays with wraparound, easing compatibility with useful external PDE solving libraries such as FiPy [60], imported by default in generated Python plugin glue code. Instantiation of the Python plugin is the same as for C++ plugins.

The entire process of plugin generation and subsequent PSE execution can be summed up in Figure 2.6. C++ and Python plugins (PyPlugins) are contained in their respective locations within the CompuCell3D PSE. BioLogo PDE solvers which use embedded Python are transparently converted to Python plugins, and
all other modules to C++ plugins. When the user runs the CompuCell3D PSE, they can set an option in CompuCellPlayer as to whether or not Python should be used. If Python will be used, the Simulator module of the CompuCell3D back end loads Python plugins and instantiates them through configuration file references, otherwise it loads and instantiates C++ plugins. Two current limitations which will be addressed in the future are the preassumption that the user has FiPy installed, and the inability to run Python plugins with C++ plugins. The first can be fixed through a configurable code generator for BioLogo, and the second is more of a global PSE design issue. Initially I was able to use Boost [21] within generated plugins for a runtime invocation of a Python interpreter on embedded Python from BioLogo, but the performance decline resulting from data transmission between Python structures and CompuCell3D data structures for processing was impractical for these types of simulations (for vasculogenesis which was 2D, almost nine-fold with an implicit solver, and still six-fold with an explicit).

I now verify BioLogo functionality by generating new plugins which encapsulate previously published biological models and comparing results of CompuCell3D execution with those published.

2.4.4 Avian Limb Bud Growth

Skeletal pattern formation in the avian (or chicken) limb bud is an experimentally well studied biological system and is often termed chondrogenic condensation as a
Figure 2.6. Diagram of translation of a BioLogo program, and execution time control flow of the CompuCell3D PSE.
result of cellular clustering into patterns which form cartilage before differentiating into bone cells. During growth of the vertebrate, increasing numbers of cartilage elements form proximo-distally (away from the body of the chicken), in-order forming the humerus, followed by the radius and ulna, and finally the phalanges. A developmental timeline [146] of this skeletal pattern formation in the chicken limb is shown in Figure 2.7.

Figure 2.7. Developmental timeline of chick-limb skeletal patterning. Drawings show transverse sections of wing buds. For all panels, proximal is left, distal right, anterior up and posterior down. From [146], with modifications.

The dynamical, bare-bones reactor-diffusion mechanism of the RD equations of [82] governs the spatiotemporal dynamics of multiple growth factors. One of these growth factors, Transforming Growth Factor Beta (TGF-β), has been shown to act as an activator molecule in pattern formation [141]. An inhibitor molecule in these equations suppresses the production, or downstream effects, of the activator [134, 142, 202]. Since there is no backcoupling of cells to either chemical field in this case [33], I can assume a static chemical field and to save time it makes sense
to presolve the equations to establish these underlying patterns. Activator concentration stimulates cellular secretion of a glycoprotein fibronectin, which establishes adhesive patches in the ECM to which cells can adhere [65]. [33] merged these models with a cell type automaton for differentiation of cells from non-condensing cells into condensing cells which would eventually form the bone of the limb using the activator as a stimulus as shown in Figure 2.8, later extended to 3D in [37]. By using fibronectin as the chemical field $C$ in Eq. (2.5) but restricting reactions to cells of type condensing, they were able to model haptotaxis, or cell type-dependent motion along the chemical gradient.

Figure 2.8. Cell type automaton for an avian limb simulation. A NonCondensing cell becomes Condensing when exposed to an activator concentration $A(x, y, z)$ above a threshold, and a Condensing cell becomes NonCondensing when exposed to an activator concentration below a threshold. From [36], with modifications.

To validate BioLogo, I extend CompuCell3D to include two new plugins: one for the cell type automaton shown in Figure 2.8, and another for haptotaxis. This latter extension will work similar to the default chemotaxis term in the CPM energy equation, with the exceptions that it will involve coupled chemical fields (one for the activator TGF-$\beta$, solved externally and populated from a file; and the other for fibronectin, whose secretion is stimulated by TGF-$\beta$). Note also that there will be coupling between these two extensions, since the cell type automaton contains
a dependency on the activator TGF-β for transitions between non-condensing and condensing cells, and only condensing cells can react to fibronectin.

I define a new CPM energy term \textit{LimbChemical} for haptotaxis in Program 11(a). This term will require a few user inputs:

1. A \textit{Threshold} for TGF-β concentration which if exceeded will stimulate fibronectin secretion.
2. A value for \( \mu \) in Eq. (2.5).
3. A \textit{rate} of fibronectin secretion, which defines the amount of fibronectin secreted at a lattice location if the TGF-β threshold is crossed.

In addition, the term requires two chemical fields:

1. The activator TGF-β, static and populated from a file. This requires the user to supply a fourth input, which I will name \textit{ConcentrationFile}.
2. A dynamic field for fibronectin, populated through secretion.

In the \textbf{Step} module, I supply the \textit{secrete} tag to govern fibronectin secretion. My rate of secretion is supplied by the user in the input \textit{FibroRate}, and my \textit{condition} for secretion is that the activator threshold at this location is above the user-supplied \textit{Threshold}. Finally in the \textbf{Equation} module, I supply a mathematical expression analogous to Eq. (2.5) using a \textit{pixelsum}, and my fibronectin field for \( C \). However, I restrict this sum to be counted only for lattice locations containing a condensing cell, by passing a condition that \textit{cell.type} is Condensing.

Once the plugin has been generated and \textsc{CompuCell3D} recompiled with the extensions, I instantiate my new plugin using Program 11(b), in my configuration file argument to \textsc{CompuCell3D}. After solving the equations of [82] which establish underlying patterns without backcoupling of cells to the chemical justifying a static chemical field to save performance, I \textit{normalized} the concentrations of the activator and stored them in \texttt{bnewSys123_71x31x281.dat}. I supply this as the \textit{ConcentrationFile} input, and specify the other three inputs as shown.

39
Program 11: (a) *LimbChemical* effective energy or *Hamiltonian* for the avian limb-bud growth simulation. The effective energy is associated with two chemical fields: *activator* (populated through a *ConcentrationFile*) and *fibronectin* (populated by cell secretion and resorption) superimposed on the CPM lattice. (b) Sample instantiation of the BIOLOGO-generated *LimbChemical* plugin in a COMPUCELL3D configuration file. The file bnewSys123_71x31x281.dat populates *activator*. 
I then couple this Hamiltonian term to a cell type automaton, which I define in Program 12(a) and instantiate in Program 12(b), through the useplugin tag in the automaton definition and passing LimbChemical as the name. This provides me with access to its inputs and fields through the '.' operator, using for example LimbChemical.activator and LimbChemical.Threshold. There are no cell state variables in this particular case, I only need to define transitions. A cell becomes Condensing if it currently NonCondensing and exposed to an activator concentration above the threshold, and I use the same Threshold with which the user instantiates the LimbChemical plugin. A cell becomes NonCondensing if it is currently Condensing and exposed to an activator concentration below this same threshold.

Program 12: (a). Cell type automaton for a chondrogenic condensation simulation with two cell types Condensing and NonCondensing. Condensing cells are more adhesive than NonCondensing cells. (b). Instantiation of this cell type automaton in the COMPUCELL3D configuration file.

The full COMPUCELL3D configuration that I used for verification can be found in the Appendix, Program 19. In addition to my generated plugins I use several
CPM energy terms, such as the adhesion energy of Eq. (2.3) which is included in 
COMPUCELL3D by default. In order to obtain higher adhesion for condensing cells, 
I set $J$ between two cells of this type to be very low, as shown in Program 13. I also 
use out of the box plugins for volume constraints (Eq. (2.4)), mitosis, and domain 
growth. Mitosis rates are obtained from [119]. For domain growth, the algorithm 
in COMPUCELL3D is density dependent. Each time the cell density (percentage 
of lattice points containing a cell) exceeds 5 percent, the grid grows by 4 rows in 
positive $z$ and then delays 10 MCS to let the CPM gradually drive cells into the 
empty rows. I start the domain at about one-twentieth of the maximum length of 
the limb bud, and simulate until the domain extends to full length.

```xml
<CompuCell3D>
  <Plugin Name="Contact">
    <Energy Type1="Medium" Type2="Medium">2.9</Energy>
    <Energy Type1="NonCondensing" Type2="NonCondensing">7.0</Energy>
    <Energy Type1="Condensing" Type2="Condensing">0.5</Energy>
    <Energy Type1="NonCondensing" Type2="Condensing">7.0</Energy>
    <Energy Type1="NonCondensing" Type2="Medium">2.9</Energy>
    <Energy Type1="Condensing" Type2="Medium">2.9</Energy>
  </Plugin>
</CompuCell3D>

Program 13: Instantiation of the Contact plugin for the avian limb simulation.
```

Results are shown in Figure 2.9. This figure is a 3D simulation and would 
normally be surrounded by a box of non-condensing cells, but I used the GUI of 
COMPUCELL3D to turn off visualization of these cells for clarity purposes. Con- 
densing cells are visualized at three points: the first at about 2300 MCS when the 
humerus is completely formed, the second after radius and ulna formation at about 
3250 MCS and finally complete formation including digits at 4250 MCS. This is 
in accordance with the limb formation timetable shown in Figure 2.7 and also in 
accordance with the results of [37].
Figure 2.9. Avian limb development, visualizing only *Condensing* cells for clarity. The limb forms at the center of a 3D box of cells, and the limb is surrounded by mostly *NonCondensing* cells. Formation begins with the humerus after 2300 MCS, followed by the radius and ulna after 3250 MCS and finally digits after 4250 MCS. This agrees with the results of [37]. Note: This is 2D a cross section taken at $y=15$.

2.4.5 In Vitro HUVEC Vasculogenesis

Vasculogenesis in Human Umbilical Vascular Endothelial Cells (HUVEC) involves cells lining the inner walls of blood vessels organizing into polygonal patterns of cell cords during embryonic development. The existing blood vessels sprout and split, and this models the initial vascular network through the formation of new blood vessels. Merks *et al.* [135] developed an *in silico* model of HUVEC vasculogenesis which reproduced experimentally observed results *in vitro*, by making the following three assumptions about HUVEC:

1. They secrete a morphogen which the ECM inactivates.
2. They preferentially extend filopodia up morphogen gradients.
3. They elongate in response to angiogenic growth factors.
Clearly, assumption (2) can be accomplished through a CPM chemical energy term. Merks et al. actually used a slightly different term for chemotaxis, which depends upon the concentration difference between a cell and its neighbor (candidate for lattice point flipping) [170]. This ability is available within COMPUCELL3D out of the box. There is also a plugin available for elongation (assumption (3)).

Assumption (1) requires a chemical field for the morphogen, and a mathematical model for evolving this chemical field. Merks et al. used a version of the Gamba-Serini PDE model [68] to simulate RD of the chemoattractant, which I refer to as $c$:

$$\frac{\partial c}{\partial t} = \alpha \delta_{\sigma_x,0} - (1 - \delta_{\sigma_x,0})\epsilon c + D \nabla^2 c, \quad (2.7)$$

Note that this equation uses the Kronecker delta $\delta_{\sigma_x,0}$ which equals 1 if $\sigma_x$ defines a cell point and 0 for an ECM point. Thus this PDE consists of (1) a positive cellular secretion term, (2) a negative ECM resorption term, and (3) the standard diffusion term. I generate a finite difference solver plugin for this PDE using BioLogo in Program 14(a). My three inputs are for constants in Eq. (2.7): $\alpha, \epsilon, D$ (diffusion constant, or DiffConst). I do not want to normalize the chemical field in this case. The PDE evolves one field, $c$, and I specify the equation in the DiffEq module, which I indicate should evolve field $c$ in the fieldName attribute. My Term uses predefined functionality for the Kronecker delta as defined above and the Laplacian. I instantiate the plugin using the COMPUCELL3D configuration file snippet shown in Program 14(b), where I provide values for my three inputs, my timestep DT, spacestep DX, and I evolve $c$ using twenty steps of finite difference per MCS. I then couple this chemical field to an existing COMPUCELL3D Chemotaxis plugin by passing the GambaSerini as a plugin name to the ChemicalField tag, and $c$ as the chemical field for chemotaxis.
Program 14: (a) CPM-PDE hybrid model of angiogenesis [135], derived from the Gamba and Serini PDEs [68], implemented in BIOLOGO as an evolver of chemical fields. (b) Instantiation of this solver in the CompuCELL3D configuration file. The CompuCELL3D Chemotaxis plugin also accepts ChemotaxisByType tags with attributes for cell type and associated Lambda values, providing control over which cell types chemotact and the strength of chemotaxis.

Merks et al. [135] experienced cell adhesion or dissociation based on the value of $\gamma = J_{Cell,Medium} - J_{Cell,Cell}/2$, where a positive $\gamma$ resulted in adhesion and a negative $\gamma$ resulted in dissociation. The value for $J$ comes from the default adhesion plugin, or Eq. (2.3). In addition, they used a length constraint which operates similar to the volume constraint, using a target length $L$ and an energy contribution proportional to the difference between a cell’s current length and $L$. This length constraint is also available in CompuCELL3D. I use these metrics to validate if my solver (running in the background) is working properly. My results are shown in Figure 2.10. Merks et al. showed that cells in the absence of adhesion could form networks resulting from elongation and chemotaxis. My results corresponded in Figure 2.10(a), where I set $\gamma$ to zero and using 10x5 cells set their $L$ value to 30, making them very long. The chemoattractant $c$ (evolved by my generated solver) is shown in Figure 2.10(b). When I set $L$ to favor more rounded cells at a value of 10, networks do not form
and I get vascular islands, also in accordance with the results of Merks et al.

Figure 2.10. (a) Output of *in vitro* capillary development with elongated cells ($L = 30$) in the absence of cell adhesion, showing the capillary networks found in [135]. (b) Chemical concentration from the same simulation as (a), visualizing the field added through Biologo. (c) Vascular islands forming due to the enforcement of rounded cells ($L = 10$).

Another option is to use embedded Python. In this example I use the FiPy libraries [60] to implement the different terms of Eq. (2.7). The three terms of this equation are expressible in FiPy using terms in Figure 2.11. For this example I use one term for secretion, one source term for resorption, and one diffusion term. The secretion amount is independent of the current concentration, while resorption is linearly dependent on $\epsilon$ corresponding to the FiPy source term. The FiPy diffusion term computes the gradient of the concentration, scaled by a diffusion factor provided to its constructor. I represent the entire PDE as a FiPy transient term, solved three-dimensionally using implicit finite volume, the discretization of which reduces to finite difference on Cartesian grids [206], with Program 15. Although solving with implicit finite difference vs. explicit will result in some performance sacrifice, it avoids instabilities at high timesteps. Either can be used in an embedded Python specification. With Python bindings created for all inputs and fields along with
predefined bindings for the timestep \( \Delta t \) and spacestep \( \Delta x \), I now use them to set up these terms. My diffusion coefficient (passed to the \texttt{ImplicitDiffusionTerm}) is the input \texttt{DiffConst}, and I use implicitly defined functionality such as \texttt{kronecker} and \( \Delta t \) to solve the PDE and evolve \( c \).

\[
\frac{\partial (\rho \phi)}{\partial t} = S_\phi + \left[ \nabla \cdot (\Gamma_i \nabla) \right]^n \phi
\]

Figure 2.11. Useful FiPy terms for representing reaction-diffusion equations. From [60].

Program 15: Expression of Eq. (2.7) using BIOLOGO and embedded Python.
2.5 BioLogo Tools Design

2.5.1 Compiler Front End

I implement the BioLogo compiler front end using C++, a convenient option due to the availability of XML parsing libraries such as Xerces-C++ [9] and a convenient method for recursive descent parsing [174] using the XMLSerializable interface provided by XMLCereal [40]. I illustrate functionality inherited from XMLCereal using the Unified Modeling Language (UML, [151]) in Figure 2.12. XMLSerializable provides event handling routines for comments, starting and ending tags, etc. and an abstract method readXML(). Each BioLogo tag has an analogous C++ concrete class which inherits from XMLSerializable and overloads readXML(). Each readXML() implementation invokes functionality from XMLPullParser, which provides methods for checking opening and closing tags, attribute availability, etc. The BioLogo compiler front end consists of:

1. A set of abstract data types implemented as C++ classes, one for each BioLogo tag, which inherit the XMLSerializable interface and overload readXML(). XML parsing and error checking is performed using XMLPullParser functionality and invoking other readXML() routines if the tag corresponds to an XML block.

2. A set of helper routines for mathematical expression parsing. These routines perform error checking, including type and scope checks, mismatched parentheses and dangling mathematical operators.

3. A central class BioLogoReader, whose instance is constructed as a Singleton [6]. This object contains pointers to every object in (1), and conversely every object in (1) contains a reference to the Singleton. The Singleton API contains methods to invoke routines in (2). Also the variable symbol table is encapsulated as a member Standard Template Library (STL, [176]) deque of maps, containing pairs of variable names and BioLogo data types. Possible BioLogo data types include: float, boolean, string, field, and cell. Type checking and inference are performed in mathematical expression parsing routines. BioLogo is statically typed but all user-defined Inputs are assumed to contain floating point data and translate to double-precision floating point values in C++, removing type specification from user requirements.
XMLSerializable

+ virtual void readXML(XMLPullParser&)

XMLPullParser

+ bool check(int)
+ string getName()
+ XMLAttrib getAttribute(int)
+ int findAttribute(string)
+ void match(int, int)
+ void skip(int)
......

Figure 2.12. Abstract XMLSerializable interface, defined by individual concrete classes correspond to BioLogo tags.

BioLogo tags can contain multiple attributes, all of which are strings but some are simple variable names and others are mathematical expressions. The process of block parsing thus consists of the following steps:

1. Retrieve a new tag from the BioLogo program by invoking XMLPullParser::check() and updating the state of the XMLPullParser.

2. Push a new variable scope. In some situations, there will be variables implicitly defined for this tag block, if so insert them into the top-level scope.

3. Invoke the readXML() routine bound to the subclass of XMLSerializable corresponding to the name of this tag, accessible through XMLPullParser::getName().

4. Retrieve tag attributes by invoking XMLPullParser::getAttribute() and passing the attribute name as a string. If the attribute is optional,
XMLPullParser::findAttribute() returns -1 if the attribute was unspecified.

5. For mathematical expressions, call BioLogoReader::parseExpression() which performs error checking including variable definitions, type and scope checks.

6. Generate intermediate code (if necessary).

7. Repeat steps (1) - (5) for each tag contained within this block. If the tag does not correspond to an XML block, continue.

8. Match this element's ending tag.


10. Generate intermediate code (if necessary).

Program 16 illustrates the Secrete class implementation, which parses the BioLogo secrete tag. As we recall, secrete has two associated attributes, amount and condition, both mathematical expressions. The former is arithmetic, and the latter is boolean. An implicitly defined variable cell is inserted into the symbol table, which is usable in these mathematical expressions. Note that when attributes amount and condition are retrieved, since condition is optional findAttribute() is invoked. Error checking on both expressions is performed in parseExpression(). The intermediate code takes the form of a conditional IF clause, with an assignment statement signified by a COPY keyword. The earlier secrete example:

   <secrete field="c" amount="x*y" condition="c greater 0.5" />

would take the following format in intermediate code:

   IF c[] > 5
   COPY c[] c[]+x*y
   END IF

In intermediate code, brackets ([]) follow all field references in mathematical expressions so that when converting to C++, the translator will know that these fields require indexing. If a tag contains a block structure (such as cell-type for instance), a loop is necessary to parse internal tags and invoke analogous
readXML() methods. Program 17 contains a code snippet from the implementation of Celltype::readXML() which checks for internal creation, updatevariables and updatecelltypes blocks. Error handling is performed in three stages, as shown in Figure 2.13. The first two, XML syntax checking and expression parsing, are performed within the compiler front end. Runtime exception handling is performed in some generated code and also within modules of the CompuCell3D computational back end.

![Figure 2.13](image)

Figure 2.13. Control flow illustrating the error checking steps involved in a BioLOGO module from initial translation to execution as a CompuCell3D plugin. Initial error checking is performed syntactically at the XML level (proper tag sequence and balancing using external libraries), followed by expression level checking which involves variable lookups, type checks, etc. Runtime exceptions are caught as typical in C++: either by handling routines in the computational back end or as traps to the operating system.
Program 16: Implementation of a class which parses the BioLogo secrete tag, and generates intermediate code.
in.match(START_ELEMENT, -TEXT);
while (in.check(START_ELEMENT)) {
    if (in.getName() == "creation")
        myReader->getCreation()->readXML(in);
    else if (in.getName() == "updatevariables")
        myReader->getUpdateVariables()->readXML(in);
    else if (in.getName() == "updatecelltypes")
        myReader->getUpdateCellTypes()->readXML(in);
    in.match(END_ELEMENT, -TEXT);
    in.skip(TEXT);
}

Program 17: C++ snippet from the readXML() member function of the CellType class, which parses and generates code for the BioLogo CellType tag.

BioLogo expression parsing follows a recursive algorithm for infix expressions using standard recursive descent [148]. A full description of its context-free grammar in Backus-Naur Form [14, 145], a convenient notation originally used to describe ALGOL [157], is provided in Appendix B. Associativity and operator precedence rules are the same as C++, with parentheses the highest followed by exponentiation, multiplication and division, addition and subtraction, logical comparators, logical AND and finally logical OR. These rules along with rules of associativity are specified outside of the grammar, within the parser. A parse tree is constructed left to right, where expected types are inherited attributes passed down the tree [174] and actual types are synthesized attributes and passed up the tree. Type inference for identifier lexemes is based on a symbol table lookup (this will crash with an Undefined variable if the identifier is undefined within the current scope). Otherwise a literal value will either be a number of a set of characters terminated by single quotes, for the former type float is inferred, and for the latter type string. One unique case is if the type is a field. In mathematical expressions, a field can either be indexed with three sets of brackets ([[]]) to access a specific floating point value in the field, or can be directly accessed. If the latter is performed, generated C++ will loop over
all points in the field and perform this operation. Thus in the case of secrete:

<secrete field="c" amount="x*y" condition="c greater 0.5" />

when scanning the condition attribute, c is interpreted as type float, since we are actually referencing every point in c; this BioLogo statement thus says “for every point in c, if the floating point value is above 0.5 add the quantity x*y.”

To perform type inference for expressions, the compiler front end looks at the corresponding operator (node) in the tree. For binary operations, the only types which can be combined are float and boolean, but all other operations with different operand types will generate a Type Mismatch error. After checking for type mismatches, the compiler front end checks type compatibility with the operator. The BioLogo boolean operators equal and notequal are compatible with operands of any data type (including cell). Boolean operators lessequal, greaterequal, less, greater are compatible with float, string, and boolean operators. Arithmetic operators are compatible with type float or boolean, other than + which can also concatenate two strings. If an operator is used with non-compatible types, an Incompatible operand types error is produced, along with a description containing the operator name and inferred types of its operands. If types are compatible, then the type of the overall expression must be inferred:

1. If the operator is boolean, the expression is boolean.

2. If the operator is arithmetic and the operands are of the same type, the expression type is the type of the operands. Otherwise, the expression is of type float (since a float combined with a boolean will still be a floating point expression).

Once a mathematical expression has been fully error checked, intermediate code is generated for the expression (also in infix notation), with the following changes:

1. If a field is referenced in a mathematical expression without indices, empty brackets [] get appended to the end. This signifies to the code generator to perform this operation for every point in the field.
2. Whitespace is removed.

3. Boolean operators which are words (lessequal, etc) are changed to their corresponding C++ symbols.

4. Single quotes are changed to double quotes (BioLogo strings use single quotes).

To illustrate the structure of BioLogo intermediate syntax, I have placed in Appendix F the intermediate code generated for the BioLogo verification simulations. This includes the LimbChemical Hamiltonian term, the Avian cell type automaton, and the scripted and unscripted GambaSerini PDE solvers.

2.5.2 Compiler Back End (Plugin Generator)

The BioLogo compiler back end converts a bug-free intermediate program to COMPUCELL3D plugins. Its framework is organized similar to the compiler front end, in that each keyword defines an interface which inherits functionality from an abstract class Statement, as shown using UML in Figure 2.14. DiffEq and CellModel are just two of many derived classes of Statement. Meanwhile, Statement supplies abstract methods read() and generate() for each keyword. Thus, reading and generating are discrete steps in the plugin generation process. Since BioLogo modules are not always independent (i.e. coupled energy terms and type automata), in some cases a module A must know functionality of a different module B before generating C++ code for A. Also in many cases code generation is context sensitive, for example the glue code will vary for each type of mathematical model extension. The data members of Statement provide a list of intermediate code lines, a list of Statement pointers for code blocks, the corresponding plugin output file, and finally the keyword. Accessor methods are provided for these data members, but for clarity I do not show them in the UML.
Figure 2.14. Architecture of BioLogo C++ code generator. The abstract class `Statement` defines an interface from which parsing and generation functionality for all keywords inherit.

Context sensitivity is controlled by a static variable `MODE` which at any time is set to one of three enumerated values `HAMILTONIAN`, `CELLMODEL` or `EVOLVER`. This variable is accessible to all `Statement` subclass implementations. Similar to the compiler front end there is a top level `Statement` called `Program` from which parsing and generation begins; although unlike the compiler front end there is no symbol table, error-checking functionality, etc. so the structure is much simpler. If we look at its `read()` method:

```c++
void Program::read()
{
    ifstream itmFile(myInputFile.c_str(), ios::in);
    string line;
    while (!itmFile.eof())
    {
        getline(itmFile, line);
        if (getParameter(line, 1) == "HAMILTONIAN")
        {
            MODE = HAMILTONIAN;
            myStatements.push_back(new Hamiltonian(
```
we can see the recursive descent parsing. The first statement encountered will either be a HAMILTONIAN, EVOLVER (generated from a BioLogo PDESolver) or CELLMODEL. First the intermediate file is opened for input; then lines of intermediate code are retrieved. If a HAMILTONIAN is found the MODE is set accordingly and a new Hamiltonian object is allocated and referenced in the member array of Statement pointers. Next, addPluginToCMake() modifies compilation scripts by adding this new plugin. The helper getParameter() routine accepts a whitespace delimited string of strings and a number n, and returns string n. Finally, the read() method of the new Hamiltonian object is invoked. The class Hamiltonian in turn implements its own read() method, and so on. The generate() method for Program is very straightforward; since a Program is not actually an intermediate statement, the implementation simply contains a counter-controlled repetition over
all internal Statements, and calls their `generate()` routines:

```cpp
void Program::generate()
{
    for (unsigned int i = 0; i < myStatements.size(); i++)
        myStatements[i]->generate();
}
```

For a more detailed `generate()` implementation, we can look at the COPY statement, which will either look like:

COPY x x+4*y

or:

COPY c[] c[]+x*y

if `c` is a chemical field. Recall that COPY is be used in the context of Hamiltonian clauses within BioLogo neighbor, cell or pixel sums. The corresponding C++ statement which should be generated in case 1 is:

```cpp
x = x+4*y;
```

And in case 2:

```cpp
c->set(pt, c->get(pt)+x*y);
```

Thus back end chemical field accesses occur through member `set()` and `get()` methods. Each computational back end plugin corresponding to a Hamiltonian contains a method `changeEnergy()` which accepts a back end `Point3D` object `pt`, which is a struct of three data members `x`, `y` and `z`. The back end is implemented in this way because the CPM automaton only updates maximally one lattice point every MCS, avoiding computing energy contributions for all CPM lattice points at every MCS, where all but one would have not changed from the previous step.
The Copy::generate() method is thus implemented as follows, with a boolean
fieldflag set if the type of the destination variable is a field, inferred through the
presence of square brackets []. myVar and myValue are populated through reading,
similar to the case of Hamiltonians:
void Copy::generate()
{
unsigned int i = 0;
bool fieldflag = false;
while (i < myVar.size())
{
if (myVar[i] == ’[’)
{
fieldflag = true;
myVar.erase(myVar.begin()+i);
myVar.erase(myVar.begin()+i);
myVar.insert(i, "->set(pt ");
}
i++;
}
if (!flag)
*myOutfile << myVar << " = " << parseExpression(myValue)
<< ";" << endl;
else
*myOutfile << myVar << ", " << parseExpression(myValue)
<< ");" << endl;
}
Note that when generating C++ for myValue, its string is passed to a helper
routine parseExpression, which contains two main responsibilities:
1. Perform a similar scan for field references which use [], and convert them to
a get(pt) invocation.
2. Handle predefined cell variables by referencing the appropriate plugin in generated code. As an example, consider the following expression in a cell sum:
COPY x x*cell.volume
In C++, this should translate to the following for back end compatibility:

59


x = x*((VolumePlugin*)Simulator::pluginManager.get(\"Volume\"))->
  getClassNode().get(cell)->volume;

In other words, CompuCell3D defines a plugin which holds and updates cell volumes, called "Volume". The plugin manager, which holds a collection of plugin pointers, must be referenced to obtain the correct plugin. Then a DynamicClassNode object is returned [39, 37, 10] corresponding to a linear set of cell attributes in memory. CompuCell3D was written in this manner to save page faults and cache misses. Volume is then retrieved for the particular cell object in the expression. All implicitly defined cell attributes can be referenced in this way. In addition, cell state variables specified in BioLogo could also be referenced, if a BioLogo cellmodel was defined in the program being translated. For example, if all cells of model X were defined to contain state variable y, then it is feasible to have an expression:

```
set a a+cell.y
```

In this case the actual generated plugin for the cell model is referenced.

Glue code for a CompuCell3D Hamiltonian plugin is generated by class Hamiltonian:

```cpp
class Hamiltonian : public Statement {

public:
    Hamiltonian(string);
    ~Hamiltonian();
    void read();
    void generate();

private:
    string myName;
    bool myDynamic;
    vector<Input*> myInputs;
    vector<Field*> myFields;
    vector<std::string> myFieldNames;

    ofstream* myPluginH;
    ofstream* myPluginCpp;
    ofstream* myPluginProxy;
    ofstream* myOutfileH;
    ofstream* myCMake;

    void generateCMake();
};
```
In this case, the file stream `myOutfile` is an inherited attribute from `Statement` and references for some Hamiltonian $H$, the file `HEnergy.cpp`. `myOutfileH` references `HEnergy.h`. There are three other output streams for `HPlugin.h`, `HPlugin.cpp` and `HPluginProxy.cpp` (registers the new energy plugin with computational back end factories). Finally, an output stream is opened for `CMakefiles.txt`, which informs CMake of the COMPUCELL3D plugin. These are opened in the constructor:

```cpp
Hamiltonian::Hamiltonian(string name)
{
    myName = name;

    system(("mkdir "+Program::path+"/plugins/"+myName).c_str());
    myPluginH = new ofstream((Program::path+string("/plugins/")
        +myName+string("/")+myName+"Plugin.h").c_str(), ios::out);
    myPluginCpp = new ofstream((Program::path+string("/plugins/")
        +myName+string("/")+myName+"Plugin.cpp").c_str(), ios::out);
    myPluginProxy = new ofstream((Program::path+string("/plugins/")
        +myName+string("/")+myName+"PluginProxy.cpp").c_str(), ios::out);
    myOutfileH = new ofstream((Program::path+string("/plugins/")
        +myName+string("/")+myName+"Energy.h").c_str(), ios::out);
    myOutfile = new ofstream((Program::path+string("/plugins/")
        +myName+string("/")+myName+"Energy.cpp").c_str(), ios::out);
    myMakefileAm = new ofstream((Program::path+string("/plugins/")
        +myName+string("/")+"CMakeLists.txt").c_str(), ios::out);

    myDynamic = false;

    myKeyword = "Hamiltonian";
}
```

Each of these files are subsequently populated in the `generate()` routine. This structure is similar for cell models and PDE solvers.
2.6 Observations

2.6.1 Contributions

My goal in developing BioLogo was to bring the CompuCell3D PSE to the lab bench for experimentalist prototyping and verification of mathematical models. I argue the success of BioLogo in this manner by illustrating the following points.

Declarative Semantics

Through the use of declarative semantics, BioLogo can accurately represent complex biological phenomena using a natural vocabulary for biologists. I have demonstrated the accuracy of BioLogo through biologically relevant verification simulations, duplicating the results of previously published simulation results and transitively, biological lab observations. While obtaining extensibility in the back end required design patterns and thinking in terms of classes, objects, and methods; BioLogo allows reasoning in terms of cellular behaviors, differentiation, and reaction-diffusion. Moreover, BioLogo tags are domain-specific and the programs very structured as a result of the piggyback on XML.

Code Savings

BioLogo can represent complex biological phenomena in a way that saves code size vs. C++. I illustrate code size savings for BioLogo extensions vs. their corresponding C++ plugins, measured in BioLogo tags vs. C++ statements, in Table 2.2. Note that BioLogo also generates input files for CMake for each plugin as well. The savings provided by BioLogo illustrate two things. First, I was successful in encapsulating functionality which requires multiple C++ statements into a more abstract and simplified syntactic sugar. Second, even for a carefully designed
object-oriented back end using design patterns, a great deal of glue code (preprocessor directives, namespaces, inheritance) is still required for plugin extensions. This even holds true for generated Python plugins. Although embedded Python is directly translated, Python plugins in general require about 80 lines of glue code (preprocessing, input bindings, and storing data within SWIG-wrapped back end structures).

Organization

The representation of this phenomena in BioLogo is more organized than its corresponding representation in C++ for the CompuCell3D back end. Each BioLogo extension required one small block of code in one file. The generated C++ plugin entails multiple files and multiple C++ classes to correctly interface with the back end. In Table 2.3, I show each extension type and the number of files (header and implementation) that each requires, plus a number of C++ classes. Because the CompuCell3D back end abstracts transitions in the state pattern, the number of header files (and C++ classes) in the case of type automata grows linearly with the number of transitions in the automaton. In this case, I specify the minimum number possible, with a plus sign to indicate that it could be higher. Since conversion to plugins is transparent, the user need not be concerned with functionality organization in the computational back end. Encapsulating a new mathematical model in CompuCell3D involves writing an XML module and passing it once through the BioLogo language tools.

Performance

The extensions generated by BioLogo are high-performance and have minimal to no explicit performance decline when compared to hand-coded extensions. Us-
ing a lattice-based model like the CPM and attempting pixel index flips hundreds of thousands to billions of times per each lattice update when the lattice contains potentially thousands of cells, performance becomes very important. Although this point is highly subjective (i.e. a developer can always put multiple hours into optimizing a piece of code using techniques specialized for that extension), I ensure that the plugins generated by BioLOGO employ general performance saving techniques in the CompuCell3D back end such as offset neighbor allocation and contiguous cellular attribute allocation [37]. Thus I maximize performance for general extensions which follow the design of CompuCell3D and the methods provided in the CompuCell3D manuals for extending the framework, which would be the most commonly used approach anyway.

2.6.2 Design Perspectives

Modularity

The CompuCell3D computational back end is modular, combining several object-oriented design patterns [10]. For generating extensions, the Plugin [63] and Proxy [69] patterns were the most valuable. In particular, core functionality such as the CPM, 3D fields, boundary conditions, etc. were clearly separated from optional functionality for more specific simulations, stored as plugins and instantiated through a proxy. Since all plugins are independently compiled and dynamically loaded, BioLOGO did not modify core functionality of CompuCell3D. This made plugin generation a very attractive option, since all translated functionality was well localized within the computational back end without necessitating extra hooks in the core.

The only hindrance came when generating C++ code that referenced data members through multiple levels of indirection. The computational back end fol-
allowed good software practices with respect to information hiding, often resulting in method invocation chains when referencing low level data. Any resulting performance penalty could be attributed to the PSE as a whole; however code generation was more challenging as a result.

It is also worth mentioning that using XML and Xerces parsing libraries made the language extensible. For example, changing a BioLogo tag name, attribute names, etc. can be accomplished through the high-level APIs provided by Xerces and the extensibility of XML. Thus encapsulating new functionality has a small cost in the compiler front end; but the ease of incorporating the same functionality into generated plugins depends on back end modularity.

Naming Schemes

Resolving naming conflicts was a difficult hurdle to overcome when generating plugins because naming conflicts could exist at multiple levels. A plugin could preexist in the CompuCell3D framework with the same name; and within plugin definitions a user may specify a variable name (an Input or Field, for instance) which conflicts with an inherited attribute.

With respect to the former, I had to make a decision on the appropriate balance between convenience and safety. In the end, the goals of the experimentalist must be given the highest priority. From their perspective, if they are extending CompuCell3D with a model that they wish to test and debug through BioLogo, it would be inconvenient to repeatedly ask for confirmation on overwriting an existing plugin since erroneous plugins would need to be overwritten multiple times until fully debugged. Moreover, CompuCell3D provides facilities such that a list of preexisting plugins can be easily generated; and all CompuCell3D plugins are contained within the same directory. I thus compromise safety slightly by allowing a
preexisting plugin to be overwritten if an extension with the same name is specified within a BioLogo program. A future addition to BioLogo can be to configure the language at installation time such that an appropriate warning is either enabled or disabled at the user’s discretion.

Information hiding in CompuCell3D did become a valuable asset in terms of avoiding naming conflicts, since data members of abstract classes were frequently restricted to private access as opposed to protected, eliminating a lot of potential conflicts. However, conflicts could still occur with member functions. Addressing this entails a deeper analysis of the CompuCell3D framework, and complete safety will require improvements in future implementations of BioLogo. Since CompuCell3D followed general C++ practices for naming schemes, the underscore (\_) was never used as an opening character of a member function or variable. Thus for any user-specified Input and Field names; their analogous generated C++ names could begin with two underscores \_\_ as well as any references in mathematical expressions within the plugin. This may work well initially, but imposes some restriction on future CompuCell3D development; if an abstract interface is extended to include member functions which begin with two underscores, the BioLogo code generator would need to be modified to ensure compatibility. An alternative method could be to dynamically rename conflicting user-specified Input and Field members, which would either require (1) a database of preexisting data members and functions which would present the same issue of requiring modification upon CompuCell3D abstract interface changes, or (2) a translation time scan of relevant abstract classes in CompuCell3D which would require a more complex translator tool and potentially increase overhead significantly.
Error and Exception Handling

The BioLogo architecture enabled localization of error checking routines and helped reliability in that regard. XML parsing was straightforward through Xerces and the XMLSerializable interface from XMLCereal, and syntax errors such as unbalanced tags, unexpected elements, etc. were caught by error handling routines from BasicUtils [39]. This functionality was localized to a pure virtual method from readXML() which subclasses (one for each BioLogo tag) override to retrieve attributes, call other readXML() routines for block parsing, etc. Since these libraries are generic, customized error messages were more difficult to generate, although I used them during mathematical expression parsing, localized to a single routine which also performed type and scope checking. Generated code had to be completely bug free, so very thorough error checking was required at the compiler front end stage. Designing robust error checking schemes occupied a great deal of time at the design and implementation phases of the language as well as compile-time overhead. By restricting all error checking to the compiler front end and subsequently producing bug-free C++ extensions to CompuCell3D, an experimentalist can work and debug in their own domain without receiving generic, obscure compile-time errors and warnings from a GPL compiler. Runtime exception handling depends largely on the computational back end although some power is provided through code generation, and the compiler back end does generate some calls to exception handlers for NULL pointer checks, uninitialized plugins and file I/O failures.

Codependent Evolution

An important issue to consider when interacting a DSL with a computational back end is their ability to evolve codependently. In other words, if the computational back end of CompuCell3D were to change (a new module added, API for
existing modules modified), would BioLogo still be compatible and if not, would there be sufficient warnings to indicate a necessary redesign of translation tools? Through this experience, I have learned that the plugin generation approach coupled with the current BioLogo design which involves no translation time analysis of computational back end structures, does not facilitate this well. To illustrate this point, consider the generation of a PDE solver. An implicitly defined mathematical function Kronecker returns 1 for cells and 0 for the medium. This requires a C++ runtime invocation of the CompuCell3D routine Potts3D::getCellG(x, y, z) to retrieve either a cell object or NULL pointer (ECM) at lattice location (x, y, z). Now suppose CompuCell3D developers changed the name of this routine to getCell(). Instantly the BioLogo translator is noncompatible, but even more worrisome is that without proper communication between computational back end and DSL developers, the DSL could feasibly be broken for an elongated period of time without warning. This implies that the DSL would need to be tested thoroughly and frequently if the computational back end is continuously evolving. The design could be extended to include compile time analysis of computational back end structures to track changes and avoid generated bugs, but the large scale scan coupled with inference algorithms for back end data members and variables would only add to an already sizable translational overhead. A consequence of this is that plugins generated the BioLogo tier of a specific release of the CompuCell3D PSE would need to be regenerated for compatibility with future releases of the software.

2.6.3 User Perspectives

A presentation and tutorial on BioLogo was presented at the CompuCell3D workshop at Indiana University (Bloomington, IN) in August of 2007. Although BioLogo has been available open source on the web for quite some time, this was
the first chance to educate potential users (most of whom were experimentalists) on BioLogO and provide them with hands-on experience, allowing them to use the language for mathematical models developed by their research groups. The workshop was a week long effort to illustrate the COMPUCELL3D PSE, and administer a survey to obtain an idea of preferred extension methods. In the end, out of ten randomly selected experimentalists, eight preferred using BioLogO for extending COMPUCELL3D over directly writing C++ plugins, indicating that the language is flexible enough to express mathematical models for several different biological processes within various organisms. One even ranked BioLogO ahead of the COMPUCELL3D XML configuration file as the easiest method for interacting with the global PSE. These ten experimentalists also rated the four methods for representing mathematical models in COMPUCELL3D: XML configuration file, Python, BioLogO and C++. The ratings ranged from 4 (for the most convenient) to 1 (for the least convenient). Figure 2.15 shows the results. Not surprisingly, experimentalists almost unanimously picked the XML configuration file as the easiest method for interaction (although this cannot be used to extend COMPUCELL3D but only to test existing plugins), and C++ as the most difficult.

There are multiple conclusions that can be drawn from this graph. First, it is clear that a simplified, high level interface to complex back end functionality is desirable for experimentalists. Almost two-thirds of those surveyed ranked C++ below both BioLogO and Python in terms of ease of use. Interestingly, Python (although close) beat out BioLogO as the second-easiest method of incorporating a mathematical model into COMPUCELL3D. Being that the syntax of BioLogO is clearly simpler, more domain-specific and better abstracted compared to that of Python, this result provides a general indicator that the level of overhead involved in code recompilation and package reinstallation is a turn-off to some experimentalists.
Figure 2.15. Results of an August 2007 survey about the ease of incorporating mathematical models into CompuCell3D.

Moreover, supplemental comments indicated a general concern about BioLogo expressive power limitations and the lack of an imperative control flow, resulting in a language that is too high level and abstract to be flexible. I tried to ameliorate this problem with embedded Python but due to direct translation the benefits become less over writing a Python plugin, other than saving the generation of roughly 80 lines of glue code at the beginning and end. Those who ranked Python low complained more about the structure (specifically indentation of code blocks), also unfamiliarity with object-oriented syntax.

Subsequently I decided to delve further into the survey results, and separate
rankings by classifying some of those surveyed, based on their responses to other survey questions, as "programmers". Although this classification necessitated some subjection, I based this on the experimentalists’ indicated comfortability with scripting and object-oriented notation (in particular the member access ‘.’ operator), which were survey questions. Based on responses to these questions, I classified six of the experimentalists as having programming experience. The results, as shown in Figure 2.16, did not indicate clear correlation between programming experience and preference.

![Diagram](image)

**Figure 2.16.** Results of the same survey in August 2007 of BIOLOGO preference vs. Python, based on programming experience.

Overall, there are several conclusions that can be drawn from these survey re-
results. First, considering the use of a DSL as a middle-level PSE tier involves more than simply evaluating the programming expertise of PSE users (i.e., experimentalists). A deeper evaluation must be performed, and questions such as the following must be answered:

1. Since using the DSL will still involve programming although at a higher level, which styles of programming are preferred in general? A language, no matter how high level, is still a language and involves programming concepts. Even BioLogo, which was component-based and used declarative semantics, with plugin generation still involved large-scale and discrete error-checking, translation and compilation steps which turned off some experimentalists due to extension overhead. There were even experimentalists who were familiar enough with object-oriented language design to interact with the CompuCell3D back end directly.

2. What, if any, is an acceptable simulation runtime performance decline as a result of using the DSL? BioLogo-generated C++ extensions experienced little to no explicit performance penalty, yet many experimentalists remained unconvinced and were willing to accept penalties associated with Python byte code interpretation. For scripting the question then becomes, what percentage of computational back end functionality should be contained within precompiled binaries?

In any case it is clear that for some, despite some performance decline and more general purpose structure, a solid set of domain-specific Python libraries invoking SWIG-wrapped back end functionality (which provides an additional opportunity to save performance) would be highly useful and in some cases, superior to this XML-based syntax that I have provided. This motivates the second DSL technique that I explore, selective precompilation. Although some of the techniques involved in the Python interface to CompuCell3D are applicable in selective precompilation, it should be clear that this Python interface is NOT a DSL; for the primary reason that by definition the layer is not domain-specific, and no nontrivial translation to computational back end functionality is involved. I develop and study a language MDLab which uses selective precompilation for a different domain, molecular dynamics. This provides an excellent contrast in style such that I can compare the
DSL design and user perspectives of MDLab with BioLogo to draw even more general conclusions about the application of these DSL interface strategies.
TABLE 2.1

SELECTED COMPUCELL3D PLUGINS. I GROUP PLUGINS INTO TWO SETS: A HIGHER-LEVEL SET OPERATES ON CHEMICAL FIELDS (LISTED FIRST), AND A LOWER-LEVEL SET WHICH IMPLEMENTS CELL PROPERTIES AND BEHAVIORS.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdvectionDiffusionSolver</td>
<td>Solves an advection-diffusion equation on a cell field.</td>
</tr>
<tr>
<td>FlexibleDiffusionSolver</td>
<td>A customizable solver of diffusion equations, which also allows secretion and absorption and restriction of diffusion by cell type. Allows variable space step, time step, diffusion constants, secretion rates and number of fields.</td>
</tr>
<tr>
<td>BoundaryPenalty</td>
<td>Enforces an energy penalty if a cell is close to a boundary, to prevent cells spreading on domain boundaries.</td>
</tr>
<tr>
<td>CellBoundaryTracker</td>
<td>Provides locations of cell boundary voxels.</td>
</tr>
<tr>
<td>CellVelocity</td>
<td>Tracks cell speeds.</td>
</tr>
<tr>
<td>CenterOfMass</td>
<td>Tracks cell centers of mass.</td>
</tr>
<tr>
<td>Chemotaxis</td>
<td>Implements cell chemotaxis to an external chemical field, with forces proportional to chemical gradients.</td>
</tr>
<tr>
<td>ExternalPotential</td>
<td>Imposes a directed potential, or force, on cells.</td>
</tr>
<tr>
<td>Growth</td>
<td>Implements a cell density-dependent algorithm for domain growth, described in [37]. The lattice will maintain its current dimensions until cell density reaches a user-specified threshold, then will grow in positive $z$ by a user-specified amount.</td>
</tr>
<tr>
<td>LengthConstraint</td>
<td>Implements anisotropic cells.</td>
</tr>
<tr>
<td>Mitosis</td>
<td>Implements cell division.</td>
</tr>
<tr>
<td>SimpleClock</td>
<td>Implements an internal timer for cells. Provides the ability to start a timer and decrement until its hits zero.</td>
</tr>
<tr>
<td>Viscosity</td>
<td>Implements cell viscosity (useful in fluid flow simulations).</td>
</tr>
</tbody>
</table>
### TABLE 2.2
MY EXTENSIONS AND THEIR CODE SIZE SAVINGS.

<table>
<thead>
<tr>
<th>Extension</th>
<th>BioLogo Tags</th>
<th>C++ Statements</th>
<th>Pct Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>LimbChemical</td>
<td>11</td>
<td>151</td>
<td>93%</td>
</tr>
<tr>
<td>Avian</td>
<td>8</td>
<td>97</td>
<td>92%</td>
</tr>
<tr>
<td>GambaSerini</td>
<td>7</td>
<td>181</td>
<td>96%</td>
</tr>
</tbody>
</table>

### TABLE 2.3
FILE AND C++ CLASS COUNTS FOR EXTENSIONS.

<table>
<thead>
<tr>
<th>Extension</th>
<th>Header Files</th>
<th>Impl Files</th>
<th>C++ Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamiltonian</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Cell Type Automaton</td>
<td>1+</td>
<td>2</td>
<td>1+</td>
</tr>
<tr>
<td>PDE Solver (C++)</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
3.1 Introduction

I apply selective precompilation to MDLab (MDL for short), which I design as a DSL for molecular dynamics (MD). MD simulations involve solving Newton’s equations of motion for a system of atoms and propagating the system through time-dependent updates of atomic positions and velocities. Although a great deal of computing power is available today, a current severe limitation of MD simulations is the inability to model systems of reasonable size for a biologically relevant period of time. The main reasons for this are that (1) computation of certain types of pairwise forces require quadratic algorithms and thus slow simulations down considerably, and (2) the size of the timestep is often limited by the fastest fluctuations \[171\] which can occur on the order of femtoseconds (fs). A growing area of research is thus to develop numerical methods for MD simulations with improved time complexity, often involving deemed acceptable approximations for selected calculations. These numerical methods may involve propagating the system over a time \(\Delta t\), computing a force, smoothing a force to zero after a certain pairwise distance, etc. Developing these numerical methods demands the ability to prototype the methods in a clear and concise manner, and test the methods for accuracy and stability using a wide range of physical systems and parameter sweeps. Once a numerical method has
been developed and thoroughly tested, it can always be reimplemented in a general purpose, compiled and optimized language within a larger framework. However, obtaining these methods which are stable at longer timesteps is most of the battle. I demonstrate the use of MDL to prototype mathematical models such as propagation schemes and force calculation algorithms, its interface to MD PSE PROTOmOL [133] libraries wrapped using SWIG [17], and the ability to interface to other Python-based tools such as Matplotlib [132] and PyMPI [163]. Finally, I include a set of user and design observations, and conclusions regarding the applicability of selective precompilation.

3.2 Molecular Dynamics Mathematical Models

In Molecular Dynamics (MD), a system of atoms is propagated with time by solving Newton’s equations of motion, or \( F = ma \), for each atom in the system and subsequently updating atomic positions and velocities over a timestep \( \Delta t \) using the resulting acceleration \( a \). The above equation represents Newtonian dynamics and can be characterized by Eq. (3.1):

\[
F(x) = M \frac{\partial^2 x}{\partial t^2} \tag{3.1}
\]

Or more generally by Eq. (3.2), where \( U(x) \) is the potential energy of the system.

\[
F(x) = -\nabla U(x). \tag{3.2}
\]

MD conformational space, or phase space \( \Gamma = (\vec{x}, \vec{p}) \) is represented by the set of \((x, y, z)\) positions \( \vec{x} \) and momenta \( \vec{p} = M\vec{v} \), where \( M \) is the diagonal atomic mass matrix and \( \vec{v} \) is the atomic velocity vector, also as \((x, y, z)\). Thus overall, each atom has six degrees of freedom, and a conformational space containing \( N \) atoms has \( 6N \)
degrees of freedom. A **propagator** $\Psi$ is used to map conformational space $\Gamma_t$ at some time $t$ to a conformational space $\Gamma_{t+\Delta t}$ after some timestep $\Delta t$. As an example, the Leapfrog [86] propagator updates positions and velocities of the system using the following time-discretization of Newton’s equations, where $n$ is the propagation step. If each step is propagated using a timestep of $\Delta t$, then $\vec{x}^n$ represents atomic positions at time $n\Delta t$:

\begin{align}
\vec{v}^{n+1/2} &= \vec{v}^n + \frac{\Delta t}{2} M^{-1} \vec{f}^n \quad \text{(3.3)} \\
x^{n+1} &= x^n + \Delta t v^{n+1/2} \quad \text{(3.4)} \\
\vec{v}^{n+1} &= \vec{v}^{n+1/2} + \frac{\Delta t}{2} M^{-1} \vec{f}^n. \quad \text{(3.5)}
\end{align}

In this case, a full timestep update of positions (a *kick*) is sandwiched between two half-timestep updates of velocities (*half-kicks*) to improve accuracy. The Leapfrog method is symplectic [31] in that it conserves energy $E$, so we call this an *NVE* method (also conserving the number of atoms $N$ and the system volume $V$). Also note that the positional update is based on a Taylor-series approximation truncated after the first derivative term, this method is deemed second-order accurate with respect to $\Delta t$, with an error proportional to $\Delta t^2$. The Leapfrog method is also time-reversible [117], and can be used to propagate a system backwards in time. There are many different propagation schemes in existence today, each of which strive for enhanced accuracy and/or stability. For example, some schemes are designed for Langevin dynamics [47], governed Eq. (3.6) which more accurately approximates non-perfect vacuum conditions in nature by including a frictional term and a random white noise force $R(t)$ [171]. Instead of conserving total energy, Langevin dynamics conserves temperature (NVT) and as a result, average kinetic energy as shown by
Eq. (3.7), where $k_B$ is the Boltzmann constant. This allows better sampling of phase space by reducing the likelihood of trapping inside energy wells.

$$F(x) = -\nabla U(x) - \gamma M \frac{\delta x}{\delta t} + R(t).$$

(3.6)

$$<KE> = \frac{3}{2} k_B T$$

(3.7)

The Brünger-Brooks-Karplus (BBK, [27]) scheme generalizes the above Leapfrog equations to operate using Langevin dynamics. Later, the Langevin Impulse propagation scheme [97, 177] performed a similar task but improved stability for longer timesteps. During propagation, different types of forces can be accounted for, which occur at multiple frequencies. For example, bond fluctuations vary at femtosecond frequencies, whereas forces between nonbonded pairs of atoms like van der Waals and electrostatic forces are much more slowly varying [171]. An unfortunate consequence is that the timestep for propagation is limited by the fastest motion, because a timestep which is too high fails to account for multiple fluctuations which occur at higher frequencies and introduces unacceptable inaccuracies in computation. This is the motivation behind the development of multiple-timestepping (MTS) propagation schemes, where multiple propagation algorithms are invoked at different frequencies, calculating different types of forces [1, 187]. The execution of an MTS scheme $\Psi_{outer}(\Gamma)$ proceeds as follows, using a second single-timestepping (STS) propagator which updates fast frequency forces more often:

- Execute $\Psi_{outer}(\Gamma)$:
  - Update slow frequency forces
  - Execute $\Psi_{inner}(\Gamma)$ $c$ times:
    * Update fast frequency forces
    * Update $\Gamma$ using $\Delta t$
This propagator would update phase space over total time $c\Delta t$, and with MTS propagators $c$ is known as the cyclelength. We can wrap $\Psi_{\text{outer}}(\Gamma)$ with another MTS propagator which evaluates even slower frequency forces, etc. which can eventually form a chain of MTS integrators, terminated by an STS. An example MTS propagation scheme is Verlet-I/r-RESPA [79, 201], which permits timestep increases but imposes some non-linear instability. MOLLY [70] propagators counter potential instabilities within this scheme by perturbing potential energy using time-averaged positions. Moreover, propagation algorithms can be modified to make acceptable approximations and constraints such as SHAKE [205] and RATTLE [8] of fast frequency forces and as a result afford larger timesteps.

3.3 ProtoMOL Design and Interfacing of MDLab

ProtoMOL [133] is an object-oriented, C++ framework for conducting MD simulations. The modular design of ProtoMOL employs several design patterns [6, 69] which results in a highly extensible software package. The computational back end can be developed standalone with its own executable application. Structurally, ProtoMOL source code is grouped into modules which accomplish a specific portion of MD simulation and are compiled into shared object machine code which is subsequently linked to form the protomol application. ProtoMOL modules are contained within their own directory within the tree structure. The ProtoMOL modules and dependencies are shown in Figure 3.1. The frontend layer handles all communication with the user, including screen and file I/O and initialization of computational back end data. The integrators provide a middle layer which runs a propagation scheme on data within the computational back end, and initialized through the front end. Computation is performed by base (basic functionality of atoms, bonds, etc. along with some constants, parsing/type converting utilities,
(etc.), forces library, topology (boundary conditions, cell management, switching functions, etc.) and parallel libraries (for parallel computation through MPI [77]) libraries. The PROTO-MOL PSE lacks an explicit user interface, but different types of observable data can be sent to Matlab-plottable files, or common MD formats compatible with Visual Molecular Dynamics (VMD, [93]).

Figure 3.1. Diagram showing PROTO-MOL computational back end modules and their interaction. From [133].

Software which runs MD simulations often tends to focus on performance. PROTO-MOL is just one example, another example is NAMD [104]. The reason for this focus is the performance bottleneck associated with determining electrostatic forces between atom pairs, which can be very expensive for large molecular systems. A central focus on performance coupled with design patterns can often result in extensibility difficulties for domain experts due to lack of readability in the source code. ProtoMol and NAMD require a knowledge of a general-purpose language to add new propagation schemes, plus some knowledge of the structure of the framework, particularly in the case of PROTO-MOL where polymorphism and inheritance are used. Moreover, without a propagation scheme that can operate at larger timesteps, high performance computing only goes so far. Even with the massively parallel super-
computers we have today running highly optimized software, it is extremely difficult to run MD simulations for a total time that exceeds the microsecond timescale on a system of reasonable size, and a direct cause of this situation is the timestep limitations of current propagation schemes required for accurate modeling of fast forces.

To address these concerns, I developed the Molecular Dynamics Lab (MDL) as an MD tool which allows for efficient prototyping, testing, and debugging of new propagation schemes, force computation algorithms, and simulation protocols. Selecting a base language for MDL required a careful consideration of the overall goals of the language. Flexibility became a high priority, because my goal was to design a language for constructing arbitrarily complex numerical methods and protocols. Ease of use for nonprogrammers was also an important consideration, as is the case with most DSLs by definition. Performance in my case was not as much of a concern, since as I outlined above I intend MDL for the purposes of prototyping, testing, and debugging numerical methods and not necessarily running them on large biological systems for long periods of time, as this can be done on one of several high-performance MD frameworks. Thus although I cannot altogether neglect performance, I associate it with a lower priority compared to BioLogo.

With goals of flexibility and ease of use holding a priority over performance, a scripting language was the most logical choice as a base. Scripting languages offer the benefits of better testing and debugging through their malleability [75], providing constructs which allow prototyping at a higher level. They also tend to be interpreted, which avoids translational overhead for repeated compilations and conversions to machine code upon small implementation changes. The choice of which scripting language to use was more delicate. Python [164] had several advantages in terms of the audience of this DSL. The dynamic typing of Python helps
substantially with ease of use, saving the user requirement of explicitly associated variable names with types, which is more of a programmer concept. This is in contrast to Java [61] for example, which although interpreted is statically typed. At the same time, Python is not restrictively typed like Tcl [152] for example which limits possible datatypes to strings and lists of strings. Python code is also readable, since block structure is enforced by indentation requirements and its syntax includes the option for object-oriented design if the user is so inclined. Second, Python is portable and cross platform, even compared with other scripting languages like Perl [121, 156] for instance which requires specification of the interpreter binary location at the top of every program. Finally, with Python I can even salvage some performance. Although a Python program is interpreted, it also written to disk as byte-compiled code the first time it passes through the interpreter, and this code will execute faster than straight-up interpretation on subsequent runs.

There is an abundance of general purpose tools composed of Python which as a result become interfaced to other Python tools without much difficulty. MDL provides a scripting interface to MD through a high level API resting upon Python and SWIG-wrapped [17] C++ libraries from ProtoMol. Separation of precompiled, wrapped functionality from PROTO-MOL and prototypable functionality in MDL will be enforced using selective precompilation. The MDL API contains constructs which provide a domain-specific abstraction layer.

To gain access to PROTO-MOL modules in MDL, I use SWIG [17] to convert selected PROTO-MOL functionality to importable Python modules, as shown in Figure 3.2. Although the conversion from C++ to Python is indeed a large performance sacrifice, some is salvaged by the ability of SWIG to create wrapper routines for class member functions that are imported as precompiled shared object binaries, within which I implement the most computationally intensive functionality such as
pairwise force evaluation. SWIG also generates Python wrappers which invoke these precompiled binaries.

Figure 3.2. Architecture of MDL and PROTOMOL. Each can be developed independently, using an IDE such as Eclipse. I use SWIG to wrap PROTOMOL modules for MDL, producing (dark arrow) a set of precompiled shared object binaries and Python wrappers which invoke (clear arrow) functionality within these binaries. MDL commands will either be fed directly into the Python interpreter or invoke the Python wrappers generated by SWIG. Simulation observable data can then be passed to external Python tools for plotting, visualization, etc.

PROTOMOL is linked to a Subversion [189] repository on Simtk [183]. Users or developers have the option of configuring and compiling PROTOMOL using Automake [73] which generates appropriate Makefiles, or using a special Makefile.swig to wrap PROTOMOL components as Python modules for MDL. This is shown in Figure 3.3. Using a second script, fullCopy.sh, these wrapped shared object modules can be copied into the lib/ directory of the MDL package, assuming the user has the language installed.
3.4 Syntax and Examples

For the domain-specific MDL API, I break the functionality of an MD simulation into a few categories, and implement each within a Python class. The collection of member functions and data members for each of these classes define the MDL user API:

1. **Physical**: The physical system; encompassing all positional and structural information of both the molecule and its external environment. Includes atomic positions, velocities, and masses; as well as temperature, boundary conditions, pressure, volume, and molecular structure (bonds, angles, etc.).

2. **ForceField**: A group of forces for evaluation. Each simulation can contain bonded (between two-atom bonds, three-atom angles, four-atom dihedrals and impropers) or nonbonded (pairwise Van der Waals and electrostatic) forces. This module also contains algorithms for the nonbonded pairwise force evaluations. For van der Waals the user can specify either a direct (full) evaluation or use a cutoff which sets the force to zero after a certain pairwise distance. Electrostatic force evaluations can use these two approaches as well, but more efficient Ewald [57, 59], Particle Mesh Ewald (PME) [44] and MultiGrid summations [23, 178, 99] are also available. If a cutoff is used, the user can supply switching functions to smooth the force to zero at the cutoff for more stability by avoiding an abrupt jump in the force value.

3. **Forces**: The atomic force vector, system energies, and a set of force fields to evaluate at different levels of propagation. Single-timestepping (STS) propagations use just one force field, but multiple-timestepping propagators use one per level.
4. **IO**: Input and output functionality, necessary for loading initial simulation data and observing results. Positional and velocity data can be read from Protein Data Bank (PDB, [18]) files or simple XYZ coordinate files. CHARMM [25] Protein Structure Files (PSF) provide the molecular structure and Parameter (PAR) files provide all information necessary to compute potential energy and forces (such as spring constants for bonds, target angles for dihedrals, etc.) The same types of data files can be output throughout a simulation, along with tabular and Matlab [55] plottable data files for observables such as temperature, energies, momenta, etc.

5. **Propagator**: The propagation scheme. Several propagators are available by default in MDL, outlined in Appendix L, but the user can also specify a propagator which they construct. Some of the available propagators model the dynamics of a molecular system, while others run stochastic sampling methods such as Hybrid Monte Carlo (HMC, [51]) and Shadow Hybrid Monte Carlo (SHMC, [81]), or constrained umbrella sampling [140].

The MDL user interacts through the API which invokes more low-level, behind the scenes functionality by either feeding commands to the Python interpreter or calling SWIG-wrapped functionality from the PROTOmOL libraries. Simulation output can subsequently be fed into Python-compatible tools for visualization, plotting, etc. and MDL is already compatible with several of these tools. Development in MDL can be done through the Eclipse Interactive Development Environment (IDE) [52] by downloading PyDev [162] plugins. Eclipse coupled with PyDev allows a user to test/debug MDL modules and subsequently run MDL simulations through its IDE which can invoke a local Python interpreter. Eclipse contains several useful features for Python development, including hyperlinks for all Python modules found in the system PYTHONPATH, the ability to insert breakpoints, etc. MDL simulations can also be run through the IPython [95] interpreter, which expands the functionality of a straight-up Python interpreter by offering command shell access, tab-completion, easy access to the PDB debugger, etc. Although Eclipse cannot invoke IPython as its Python interpreter, IPython can be invoked within Python programs. Development of C/C++ modules for PROTOmOL could potentially be done with the CDT (C/C++ Development Toolkit) [29], although this option has
yet to be explored.

3.4.1 Simulation Protocols in MDL

An MDL user can prototype schemes for propagating phase space \( \Gamma = (\vec{x}, \vec{p}) \) to a new phase space \( \Gamma' = (\vec{x}', \vec{p}') \), over a timestep \( \Delta t \). These schemes can run dynamics or sampling and henceforth I refer to them as propagation schemes. To test the accuracy and stability of a propagation scheme, it is necessary to assemble an MDL simulation protocol, which uses the MDL API and its core modules to set up a physical system, forces to evaluate, observables to record or plot, etc.

Physical System

The first step for any MD simulation will usually be to define a physical system. This is done in MDL by creating an instance of class `Physical`, i.e:

```python
phys = Physical()
```

In this particular case I chose to name my object `phys`, but any valid Python variable name can be used. After execution of this Python statement, `phys` becomes an object of type `Physical` and contains multiple associated data members which can be accessed throughout a simulation, using the Python member access `.' operator. I now list the member variable names and datatypes, primitive data types are simply Python primitives. From this point forward, I refer to \( N \) as the number of atoms in a simulated molecular system, meaning that this system has \( 6N \) degrees of freedom.

1. **positions** (Numpy array of float): Atomic position vector. Always contains \( 3N \) elements, with each atom having an associated \((x, y, z)\) position.

2. **velocities** (Numpy array of float): Atomic velocity vector. Similar to positions, containing \( 3N \) elements, with each atom having an associated \((x, y, z)\) velocity. Atomic velocities are in units of \( \text{Å}/\text{fs} \).
3. **masses** (Numpy array of float): Atomic mass vector, contains \( N \) elements. Although in MD masses are in general considered a diagonal \( N \times N \) matrix with diagonal element \((i, i)\) containing the mass of atom \(i\) in units of AMU, I store this diagonal as simply a vector to conserve space.

4. **invmasses** (Numpy array of float): Inverse atomic mass vector. Similar to the mass vector, but each element of this vector contains one over the AMU mass of the corresponding atom. Units of these contents are thus \(AMU^{-1}\).

5. **time** (float): Current simulation time in units of fs. The simulation is assumed to begin at time zero.

6. **temperature** (float): Kelvin temperature of the system. Each system starts at 300 K unless this value is manually changed.

7. **seed** (integer): Seed for random number generation; may be necessary in case where randomness is involved to ensure cohesivity - for example if no atomic velocities are supplied prior to applying a propagation scheme, random values are generated using a Maxwell distribution based on the Kelvin temperature of the system. This is just one example of where randomness is involved in MDL simulations. Default value is 1234.

8. **exclude** (string: “1-2”, “1-3”, “1-4”, “scaled1-4”, or “none”): When computing pairwise forces, this provides an option of excluding certain pairs of atoms if they are covalently bonded (and thus presumably have had corresponding bonded forces computed). “1-2” excludes all pairs of atoms which are directly bonded, and “1-3” excludes 1-2 plus pairs of atoms which are bonded to the same third atom (a three-atom angle). “1-4” (the default) excludes 1-3 plus pairs of atoms which are bonded within three atoms. “scaled1-4” excludes 1-3 but scales 1-4 using different parameters for van der Waals and electrostatic interactions. “none” does not exclude any pairs.

9. **bc** (string: “Periodic” or “Vacuum”). Simulation boundary conditions. Periodic boundary conditions (PBC) implement a wraparound at the simulation boundaries which are defined by a periodic box that encompasses the system (using the minimum \(x, y,\) and \(z\) of all atoms). When calculating pairwise distances between atoms, PBC determines if the wraparound distance would be shorter than the direct Euclidean distance between the two atoms, and uses that value for force computation if necessary. PBC tend to be more useful for solvated systems, with solvent (for example) water molecules encircling a solute molecule, bonded with weak attractive forces. They also can be useful for simulations of noble gas dynamics (i.e. argon). Vacuum boundary conditions (VBC) implement no wraparound and are better for unsolvated systems where bulk dynamics are not involved.

10. **cellsize** (float). For nonbonded force evaluations, this provides an option for improving efficiency if using a cutoff. When pairwise evaluations are used with a cutoff, it would be highly inefficient to check all pairs of atoms to see which pairwise distances fall outside the cutoff (thus assuming zero force). The internal pairwise force computation divides space into cubic cells, and only atom pairs within neighboring cells are considered for van der Waals and
electrostatic forces. Thus for maximum efficiency, the cellsize should be equal to half the cutoff.

11. **remcom** (string: "yes" or "no"). Specifies whether motion of the center of mass (or *translational motion*) of the collective molecule should be removed when updating velocities. Default is "yes".

12. **remang** (string: "yes" or "no"). Specifies whether angular momentum of the center of mass (or *rotational motion*) of the collective molecule should be removed when updating velocities. Default is "yes".

13. **cB1, cB2, cB3, cO** (Numpy array of float). For PBC, these specify the *cell basis vectors* and the cell *origin*. Each contains three elements for \(x, y, z\). The cell basis vectors provide a set of three linearly independent vectors which span physical space and determine the size of the periodic box. By default, the periodic box is determined by the maximum \(x, y, z\) coordinates of all atoms, so the vectors are initialized to \([cB1 = [x 0 0], cB2 = [0 y 0], cB3 = [0 0 z] \). The cell origin determines the tail location of these vectors and is by default set to \([0 0 0] \). For VBC these are not used.

Each of the data members of a **Physical** object can be set manually. Other types of data can be initialized using input files, readable through the API of the MDL **I0** class. An **I0** instance can be constructed in the same fashion:

\[
i0 = I0()
\]

with the object name following the normal rules of Python variable naming.

The **I0** interface provides several basic member functions for populating physical structures, including:

1. **readPDBPos**: Populates the position vector using a PDB file. A PDB file provides each atom with a unique integer identifier (starting from 1), along with its atom type (alpha-carbon, amide nitrogen, etc.), residue name and a set of \((x, y, z)\) coordinates. Positions can be populated in this way, by reading an XYZ or DCD file as described below, or manually.

2. **readPDBVel**: Populates the velocity vector using a PDB file. The user does not need to populate the velocity vector themselves; they can randomly generate velocities by setting the **temperature** data member of the **Physical** object, and control their behavior using the **seed** data member. This will most often be used to restart a simulation from a previous state.

3. **readXYZPos**: An alternative to reading a PDB file; an XYZ file consists of repeated lines of atom types and three coordinates. The line number specifies the unique identifier for each atom. This function populates the position vector.
4. **readXYZVel**: Similar purpose for the velocity vector.

5. **readDCDTrajectory**: Populate the position vector using a binary CHARMM/X-PLOR [25] DCD trajectory file which contains \((x, y, z)\) coordinates for each atom. This type of file input is unique in that it saves state; so that if `readDCDTrajectory()` is invoked a second time, parsing will begin from the end of the previous set of atomic positions or frame. Thus, a DCD file can contain multiple frames and repopulate the position vector at a certain frequency, for example.

6. **readPSF**: Populates the structure of the system using a CHARMM PSF. This is the only option here; every simulation must provide a PSF, and this provides each atom with its name and type, along with its mass (AMU) and charge (electrons) but no coordinates. The PSF also defines by atom indexing which atom pairs are covalently bonded; also which atom triplets form angles (two atoms bonded to a common third atom), and which quadruplets form impropers (three atoms bonded to a common fourth atom), or dihedrals (a sequence of four bonded atoms). If some atoms are capable of hydrogen bonding, \(\text{H}^+\) donors and acceptors are also specified in the PSF.

7. **readPAR**: Reads a CHARMM parameter (PAR) file, which provides other parameter values necessary for force computation. For example, when computing the energy contribution due to the deviation between the length \(l\) of a bond from equilibrium length \(L\), a spring constant \(k\) is used, as shown in Eq. (3.8).

\[
V_{\text{bond}} = k(l - L)^2. \tag{3.8}
\]

The force contribution is subsequently computed as the gradient of this contribution to the potential energy with respect to positions. The PAR file provides values for \(k\) and \(L\) for every bond. There are several other parameters of this sort provided by the PAR, for more information visit the CHARMM website [32].

8. **readEigenvectors**: MDL provides propagators for normal mode analysis (NMA, [24, 179, 199]) which strives to increase the timestep for MD simulations by generalizing fast frequency motions into a set of vectors governing their harmonic approximations around an equilibrium conformation of the system. Using NMA will require the eigenvectors of an applied mass-reweighted Hessian matrix (matrix of second derivatives of the potential energy), with eigenvalues corresponding to frequencies. These eigenvectors can be stored in a binary file.

For a more complete example, the following protocol initializes a simulation of an 1101 atom solvated Bovine Pancreatic Tripsin Inhibitor (BPTI) system, studied experimentally in [45, 48] and computationally in [147]:
phys = Physical()
io = IO()
io.readPDBPos(phys, "examples/bpti_water_1101/bpti.pdb")
io.readPSF(phys, "examples/bpti_water_1101/bpti.psf")
io.readPAR(phys, "examples/bpti_water_1101/bpti.par")
phys.bc = "Periodic"
phys.cellsize = 4
phys.exclude = "scaled1-4"
phys.temperature = 300
phys.seed = 7536031

Since I am running a solvated system, I use PBC and allow the back end to populate cell basis vectors using a default origin and periodic box. I also choose to initialize the velocity vector randomly using a temperature of 300 K and a seed of 7536031. For nonbonded force evaluation, I will be using a cutoff of 8 angstroms, and so I set my cubic cell size to 4 angstroms to maximize efficiency.

Force Computation

Calculating system forces in MDL involves two main structures: a Forces structures which keeps track of the atomic force vector and system energies (all types of potential, and kinetic), and a ForceField object which contains a collection of force calculators for the system. Forces can be divided into two types: bonded (between covalently bonded atoms) and nonbonded (between atoms which are not covalently bonded). Some bonded forces are due to deviation of two-atom bonds, three-atom angles, and four-atom dihedrals and impropers from equilibrium lengths and angles. Others may be introduced restraining forces which introduce biasing potentials that can for example keep a dihedral harmonically oscillating around an artificial target value. Examples of nonbonded forces include van der Waals and electrostatic forces. A Forces object is constructed in the standard fashion, and its member function makeForceField returns an MDL ForceField object. By passing
the string "charmm" as below, a force field containing four bonded forces (bond, angle, dihedral and improper) and two nonbonded forces (van der Waals and electrostatic) is constructed, which correspond to potential energy terms used in the CHARMM [32] MD package:

```
forces = Forces()
ff = forces.makeForceField(phys, "charmm")
```

Without the "charmm" string, an empty ForceField is returned and must be populated manually using member functions bondedForces and nonbondedForces, passing characters to represent each type of force ('b' for bond, 'a' for angle, 'd' for dihedral, 'i' for improper, 'l' for van der Waals and 'c' for electrostatic). For example the following is equivalent to the above, although subsets of these characters can be specified if the user does not want to evaluate all six forces (for example in noble gas simulations, there are no bonded forces):

```
forces = Forces()
ff = forces.makeForceField(phys)
ff.bondedForces("badi")
ff.nonbondedForces("lc")
```

Van der Waals forces are approximated by a Lennard Jones potential (thus the character 'l' above), which includes a repulsive force at a short pairwise distance $r$ (Pauli Exclusion), and an attractive force at a large pairwise distance (van der Waals), as shown in Eq. (3.9).

$$V_{LJ} = 12\left(\frac{A}{r^{12}}\right) - 6\left(\frac{B}{r^6}\right) \quad (3.9)$$

Electrostatic forces are computed using a Coulombic potential (thus the 'c'), following Coulomb’s Law represented in Eq. (3.10) and treating the two atoms as point charges $q_1$ and $q_2$, with $k$ as the electrostatic constant.
\[ V_{\text{Coulomb}} = \frac{kq_1q_2}{r^2} \]  

(3.10)

Nonbonded pairwise forces can in general be computed directly, or using a cutoff. A direct or simple full computation computes pairwise forces by looping over all pairs of atoms; which is an \(O(n^2)\) algorithm for a system of \(n\) atoms. By using a cutoff, some of this computation can be saved by assuming a pairwise force to be zero if two atoms have a pairwise distance larger than some cutoff in angstroms. These settings can be specified using the params dictionary data member of ForceField, which will map keywords to appropriate values, i.e.:

```python
ff.params["LennardJones"] = {'algorithm':'SimpleFull'}
ff.params["Coulomb"] = {'algorithm': 'Cutoff',
                        'cutoff':8.0}
```

The algorithm key maps to the algorithm for nonbonded calculations, and the cutoff parameter specifies the cutoff length in angstroms. A cutoff algorithm is still quadratic in the worst case; but there have been several more efficient algorithms for computing electrostatic forces that are included in MDL. For example, the Ewald method (algorithm ”Ewald” in MDL, [57, 59]) achieves \(O(N^{3/2})\) complexity by approximating each point charge with a Gaussian charge density, splitting the sum into a real \((O(n))\) and reciprocal (Fourier) \((O(n^2))\) space term, each of which rapidly converge. Particle Mesh Ewald (PME) (algorithm ”PME”, [44]) extends the Ewald summation by interpolating charges onto a mesh and use Fast Fourier Transforms to evaluate the reciprocal space term, yielding a complexity of \(O(n \log n)\). Finally, the MultiGrid method (”MultiGrid”, [23, 178]), evaluates electrostatic forces in \(O(n)\) time through the use of multilevel summations on successively coarser grids. Of course increases in speed result in accuracy compromises.
If a cutoff is used, instabilities can be created by directly dropping the force to zero at the cutoff distance. For this reason, it is standard practice to apply a switching function after some distance \( d < \text{cutoff} \), to smooth the force to zero. These values can also be set using the `params` dictionary, to avoid overwriting the previous we can directly index using keys and set corresponding values. For example, the switching function can be \( n \)-order continuous, where \( n \) is specified using an "order" parameter:

```python
ff.params['Coulomb']['switching'] = 'Cn'
ff.params['Coulomb']['order'] = 2  # Second order continuous
```

Other switching function types include "Universal" (the default, performs no switching), or "Shift" which performs a shift of the potential energy to avoid the sudden change in force that causes this discontinuity. The distance in angstroms at which to start and stop applying the switching function is also controllable through "switchon" (defaults to zero) and "switchoff" (defaults to the cutoff). Most often the default value for "switchoff" will be desired, i.e. there is not much point to applying a switching function after a cutoff or halting its application before the cutoff distance. However, "switchon" is often nonzero to improve accuracy, although must also be chosen carefully to avoid instabilities:

```python
ff.params['Coulomb']['switchon'] = 4.0
```

In addition to these parameters, there are others available for the fast electrostatic computation algorithms. These are outlined alongside the algorithms themselves in Appendix M.

Output

MDL output, controllable through the `IO` class, provides a useful method to interactively view observables. Output can be in the form of files, plots, or screen
output. Some types of output are instantaneous (these are all to files), while others such as trajectory files, plots, and screen can be set to occur at a specific frequency. For example, setting the \texttt{screen} data member:

\begin{verbatim}
io.screen = 2
\end{verbatim}

prints step number, femtosecond time, total energy, temperature, and volume to the screen every 2 steps of propagation, formatted according to the following example:

\begin{verbatim}
Step: 0, Time: 0.000 [fs], TE: 198.0526 [kcal/mol], T: 308.2064 [K]
Step: 2, Time: 40.000 [fs], TE: 198.0234 [kcal/mol], T: 302.3148 [K]
Step: 4, Time: 80.000 [fs], TE: 198.0834 [kcal/mol], T: 284.0183 [K]
Step: 6, Time: 120.000 [fs], TE: 198.2843 [kcal/mol], T: 263.5826 [K]
Step: 8, Time: 160.000 [fs], TE: 198.2702 [kcal/mol], T: 256.1270 [K]
Step: 10, Time: 200.000 [fs], TE: 198.0043 [kcal/mol], T: 262.1633 [K]
\end{verbatim}

The user can also produce observable plots at specific frequencies through the \texttt{plots} data member of the \texttt{IO} module. \texttt{plots} is a Python dictionary which maps observable names (Python strings) to integer frequencies. For example, to plot pressure every five steps and total energy every two:

\begin{verbatim}
io.plots = {'pressure':5, 'totalenergy':2}
\end{verbatim}

Plottable observables include the following:
1. \texttt{'potentialenergy'}: Potential energy (kcal/mol)
2. \texttt{'kineticenergy'}: Kinetic energy (kcal/mol)
3. \texttt{'totalenergy'}: Total energy (potential + kinetic, kcal/mol)
4. \texttt{'pressure'} Total pressure (Pa)
5. \texttt{'volume'} Total volume ($\text{Å}^3$)
6. \texttt{'bondenergy'} Energy between two-atom bonds (kcal/mol)
7. \texttt{'angleenergy'} Energy between three-atom angles (kcal/mol)
8. \texttt{'dihedralenergy'} Energy between four-atom dihedrals (kcal/mol)
9. \texttt{'improperenergy'} Energy between four-atom impropers (kcal/mol)
10. 'ljenergy' Energy between nonbonded pairs, due to the Lennard Jones potential (kcal/mol)

11. 'coulombenergy' Energy between nonbonded pairs, due to electrostatic interactions (kcal/mol)

12. 'shadowenergy' Shadow energy, if computation of a shadow Hamiltonian [54] is desired (kcal/mol).

For non-instantaneous file output, the IO class includes a second Python data member dictionary `files`, which maps Python strings to Python pairs (tuples), with the first element a Python string for the filename and the second an integer frequency. For example, to output the Kelvin temperature, system energies and momenta at respective frequencies of 5, 2, and 1:

```python
io.files = {'temperature':('temperatures.out',5),
            'energies':('energies.out',2),
            'momentum':('momentum.out',1)}
```

Each of these will generate output in tabular format, with the first column containing the step number and subsequent columns containing observables (in the case of temperature and momentum there is only one observable, in the case of energies there are several). This format allows these files to be easily read by an external plotting program (Matlab, for instance). Trajectory files can also be generated using the following:

1. 'dcdtraj': Binary DCD file containing atomic positions. Each output is appended to the user-specified file.

2. 'pdbframepos': PDB file containing a snapshot of atomic positions. Each output generates a new file, with the step number and the .pdb suffix concatenated to the end of the user-specified filename.

3. 'xyztrajforce': XYZ file containing \((x, y, z)\) components of atomic forces (units of kcal/mol Å). Each output is appended to the user-specified file.

4. 'xyztrajpos': XYZ file containing \((x, y, z)\) components of atomic positions. Each output is appended to the user-specified file.

5. 'xyztrajvel': XYZ file containing \((x, y, z)\) components of atomic velocities. Each output is appended to the user-specified file.
One more type of non-instantaneous file output is ‘dihedrals’, which is mapped to a three element tuple: the filename, the frequency, and the dihedral number. If this is specified, the angle value in radians of the corresponding dihedral will be output to the user-specified file:

```python
self.files = {'dihedrals':('dihedrals.out', 4, 0)}
```

Note that modifying the entire dictionary at any point erases previous entries. To add entries to an existing dictionary while keeping previously specified output files and frequencies, use the index operator (brackets, []):

```python
self.files['dihedrals'] = ('dihedrals.out', 4, 0)
```

Instantaneous output can be performed using one of the following routines. Each of these routines accept two arguments: a Physical object, and a filename as a Python string:

1. **writePDBPos**: Write the atomic position vector, including all necessary atomic information, to a PDB file.
2. **writePDBVel**: Same purpose, but for the atomic velocity vector.
3. **writeXYZPos**: Write the atomic position vector, including all necessary atomic information, to an XYZ file.
4. **writeXYZVel**: Same purpose, but for the atomic velocity vector.
5. **writeXYZBinPos, writeXYZBinVel**: Same as their counterparts but write an XYZ file in binary format.
6. **writePAR**: Write a CHARMM parameter file.
7. **writePSF**: Write a CHARMM PSF file.

Below is an example where many of these data files are written at the end of a simulation. In this case, I write CHARMM parameter and PSF files, along with a PDB file and binary XYZ for atomic positions, and a regular XYZ file for atomic velocities:
io.writePAR(phys, "output.par")
io.writePSF(phys, "output.psf")
io.writePDBPos(phys, "final.pdb.pos")
io.writeXYZVel(phys, "final.xyz.vel")
io.writeXYZBinPos(phys, "final.xyz.bin.pos")

Propagation

A physical system is propagated using an MDL Propagator object, whose constructor accepts Physical, Forces and IO instances:

prop = Propagator(phys, forces, io)

declaring that this propagator will be updating the passed physical system with time, populating the passed Forces object with atomic forces and system energies, and outputting data at frequencies specified by the IO object. Its member function propagate() will perform the update of physical space, and accepts the following parameters:

1. scheme: A Python string (for STS) or list of Python strings (for MTS), corresponding to propagation methods defined in MDL. Several propagation methods are predefined in MDL and are listed in Appendix L; others can be user-defined as illustrated in Section 3.4.2. For MTS, the outermost propagation scheme should be specified first (thus the last entry is always a STS propagator). If no STS propagator is specified, 'Leapfrog' is assumed.

2. steps: Number of steps for propagation.

3. cyclelength: An integer or list of integers corresponding to cycle lengths for all MTS propagators. For STS propagation this argument can be simply omitted, for a two-level propagation scheme one integer can be specified, otherwise for three or more levels a list must be used (again, with the outermost cyclelength first).

4. dt: Timestep for propagation in femtoseconds.

5. forcefield: A ForceField object (for STS) or list of ForceField objects (for MTS) to use for each propagator in the scheme. For MTS, the number of elements in the list should correspond exactly to the number of propagation levels. ForceField objects are created as was shown in Section 3.4.1.
6. (optional) **params**: A two-dimensional Python dictionary object, which maps Python strings (propagator scheme names) to a second dictionary, which maps acceptable parameter names to values. A list of available parameters for pre-defined MDL propagators is also shown in Appendix L.

The MDL `ForceField(s)` passed to `propagate()` have not yet been instantiated with force objects. This happens lazily in the call to `propagate()`, to avoid rebuilding the force field each time the user specification changes. For example, the user could feasibly specify bonded forces for a force field, then later before `propagate()` is invoked decide to remove these forces, which without lazy instantiation would result in a redundant build. The instantiation proceeds as follows:

1. For each forcefield `ff`:
   
   (a) For each specified force type (a character = 'b', 'a', etc.):
      
      i. Obtain any user-specified parameter values. For LennardJones ('l') and Coulomb ('c'), this will minimally include an algorithm and switching function, and depending on these values can include more parameters.
      
      ii.Invoke the MDL `ForceFactory` (Section 3.5.2) with the force type and parameters to obtain the appropriate, instantiated force object.
      
      iii. Add this object to `ff`.

I now show some examples of calls to `propagate`:

- **Example: One-level Leapfrog**

  In this case, I am just performing STS propagation. Suppose I wanted to propagate at a timestep of 0.5 fs for 2000 steps, for a total time of 1 ps. Assuming I created a `ForceField` named `ff`, I could accomplish this with the following statement:

  ```
  prop.propagate(scheme='Leapfrog',
                  steps=2000,
                  dt=0.5,
                  forcefield=ff)
  ```

  The MDL ‘Leapfrog’ propagator does not accept any further parameters, so this is enough. Any propagator which accepts parameters contains corresponding default values for them, thus the `params` argument to `propagate()` is always optional.
• Example: Two-level Impulse with STS Langevin Impulse

This is an example of an MTS propagation, containing an outer and inner (STS) propagator. Since my STS propagator is not Leapfrog, I must provide both propagator scheme names in my scheme parameter. I only need one cyclelength integer since I only have one MTS propagator, but I need a list of two ForceField objects (assume I called them \textbf{ff} for the outer and \textbf{ff2} for the inner). Supposing I wanted to propagate for 100 steps at an outer cyclelength of 5 and an inner timestep of 0.1 (note this yields a total propagation time of 100*5*0.1 = 50 fs), my invocation would proceed as follows:

```python
prop.propagate(scheme=['Impulse', 'LangevinImpulse'],
               steps=100,
               cyclelength=5,
               dt=0.1,
               forcefield=[ff, ff2],
               params={'LangevinImpulse': {'temp':250,
                                          'gamma':91,
                                          'seed':100}})
```

The MDL Impulse propagator implements the Verlet-I/r-RESPA method \cite{78, 79, 201} which divides forces into fast and slow components, evaluating the latter at a smaller frequency. In this case, the STS propagation scheme, LangevinImpulse, would evaluate the fast (i.e. bonded forces) every 0.1 fs while the slow forces would be evaluated every 5 simulation steps, or 5*0.1=0.5 fs; saving simulation time and using acceptable timesteps can preserve stability. The Impulse propagator does not accept parameters, but LangevinImpulse does although there are defaults available for each. For illustration purposes, I set each parameter - temp (Kelvin Temperature), the constant gamma for Langevin dynamics, and seed (for the initiation of random white noise).

• Example: Three-level BSplineMOLLY, with Impulse and STS BBK

The mollified impulse method (MOLLY, \cite{70, 71, 100}) can increase the acceptable timestep length of Verlet-I/r-RESPA by 50\% \cite{123} using time-averaged positions obtained over a dynamics run with a set of reduced forces through a process called mollification, coupled with a BSpline weight function when determining the average positions. With a non-default three-level MTS scheme, I must specify three different propagators, and three different ForceField objects (assume I called them ff for the outermost, ff2 for the middle and ff3 for the innermost). If I wanted to propagate 200 steps at respective cyclelengths of 10 and 5 with an inner timestep of 0.1 for a total propagation time of 1 ps:

```python
prop.propagate(scheme=['BSplineMOLLY', 'Impulse', 'BBK'],
               steps=100,
```
cyclelength=[10, 5],
dt=0.1,
forcefield=[ff, ff2, ff3],
params={'BsplineMOLLY': {'type': 'long',
'stepsize': 2},
'BBK': {'temp': 330,
'gamma': 3,
'seed': 1234}})

For BSpineMOLLY, the stepsize is the timestep to use for MOLLY averaging, and the type (a Python string, 'short' or 'long') for the type of BSpline weight function to apply (long is more stable, particularly for large Δt). Parameters for BBK (referencing the Langevin dynamics of Eq. (3.6)) include the seed for random force generation, γ and a Kelvin temperature.

3.4.2 Defining New Propagation Schemes

In addition to the propagation schemes already available in the MDL framework, users can define and test new propagation schemes. MDL expects new propagators to be either in the form of a Python class, or Python function. The Python class can subsequently be used as an abstract datatype to define propagator instances and associate member functions and variables with each instantiation of the class. Propagator functions accept all necessary system data as formal parameters and operate directly on that data, propagating positions and velocities using a formal parameter for the timestep.

When a new propagator is defined (class or function), it is registered with a propagator factory [6], described in Section 3.5.1. This factory maps Python strings to corresponding propagator constructors or function handles. Note that the propagate() member function of MDL class Propagator accepts a Python string (or list of Python strings) for the name of the propagation scheme, and this uniquely defines the propagation scheme for the simulation.
Classes

An MDL propagator class defines three optional routines:

1. **init()**: Invoked once at the beginning of propagation. This should initialize member variables which will be repeatedly updated, in some situations perform an initial force calculation, *etc.*

2. **run()**: Invoked at every step of propagation.

3. **finish()**: Invoked once at the end of propagation.

I will begin by illustrating each method through an example prototype of the BBK [27] propagator. Although BBK has been SWIG-wrapped and is available using fast precompiled binaries, I believe this method provides a good example of a prototype which will fully detail the features and functionality of MDL propagator objects. Afterwards, I will give an example prototype of a more complex propagator object which has since been included in the MDL framework, Nosé-Poincare [20].

To illustrate the power and usefulness of MDL for colleagues, in the appendix I show a very complex implementation of the Recursive Multiple Thermostat method [194] which uses Hamiltonian separation [195].

The first step is to define a Python class **BBK**. The class should be defined in a Python module with the same name **BBK**, stored in a file **BBK.py** in the MDL `src/propagators/objects` directory. All subsequent code snippets should thus be included in this file **BBK.py**. The BBK propagator must be registered with the MDL propagator factory. Upon a call to the member function `propagate()` of the `Propagator` class, the MDL propagator factory searches the `src/propagators/objects` directory for Python modules. Each Python module is in turn expected to define two variables: the first a string `name` bound to a Python string with the name of the propagation scheme (the same name as referenced in the call to `propagate()`), and `parameters`, bound to a tuple containing mappings from Python string parameter names to default values. As BBK uses Langevin
Dynamics, I need to provide a temperature in a parameter I will call "temp" which I will default to 300 K, a friction constant \( \gamma \) which I default to 0.5, and a random number "seed" which defaults to 1234 (this can ensure that multiple runs produce the same results despite the addition of random white noise):

```python
name="BBK"
parameters=("temp", 300,
            "gamma", 0.5,
            "seed", 1234)
```

This propagator can subsequently be used by calling `propagate()` in a simulation protocol, with default parameters:

```python
prop.propagate(scheme="BBK", steps=200, dt=0.5, forcefield=ff)
```

or non-default parameters:

```python
prop.propagate(scheme="BBK", steps=200, dt=0.5, forcefield=ff,
                params={'BBK':{'temp':285,'gamma':1.1,'seed':1000}})
```

Since BBK is a single-timesteping (STS) propagator, the class should inherit from STS; multiple-timesteping propagators would inherit from MTS:

```python
class BBK(_STS):
```

When parameters are specified in a Python propagator module, their names become bound as member variables of the Python propagator class. Thus the class BBK now contains three member variables which can be used within computations inside member functions. The `init()` method of an MDL propagator class accepts an MDL Physical object, Forces object, and Propagator object. For purposes of this method, I simply calculate forces once for initialization, using the calculateForces method of the MDL Propagator class, passing the Forces object:

103
def init(self, phys, forces, prop):
    prop.calculateForces(forces)

    After performing an initial force calculation, I perform all phase space updates in the run() method, which also accepts the same MDL Physical, Forces, and Propagator objects. The run() method will propagate phase space by performing direct updates of the Physical object:

def run(self, phys, forces, prop):

    I then add a random force $f_R$ into the calculations, scaled by a force constant proportional to $\gamma$:

    $$f = f + f_R \sqrt{\frac{2kT \gamma}{\Delta t}},$$

    (3.11)
done through the MDL statements below, using an auxiliary forceconstant:

    forceconstant = 2*Constants.boltzmann()*self.temp*
                    self.gamma/self.dt
    forces.force += forces.randomForce(self.seed)*sqrt(forceconstant)

    Assuming I propagate the system from step $n$ to step $n+1$ (where time corre-
sponds to $n\Delta t$ and $(n + 1)\Delta t$), I can now update the velocity vector by a half kick using Eq. (3.12), which provides some initial damping. Note that scaling by a factor of $-\gamma$ overtime offsets the increase in kinetic energy by the addition of random force.

    $$v^{n+1/2} = v^n (1.0 - \frac{\Delta t}{2} \gamma)$$
    $$v^{n+1/2} = v^{n+1/2} + \frac{\Delta t}{2} M^{-1} f^n,$$

    (3.12)
phys.velocities *= 1.0-0.5*self.dt*self.gamma
phys.velocities += 0.5*self.dt*phys.invmasses*forces.force

    I then perform a kick on positions:
phys.positions += phys.velocities*self.dt

And another calculation of forces followed by another random force:

prop.calculateForces(forces)
forceconstant = 2*Constants.boltzmann()*self.temp*
               self.gamma/self.dt
forces.force += forces.randomForce(self.seed)*sqrt(forceconstant)

And then a final half kick on velocities, using Eq. (3.13),

\[
v^{n+1} = v^{n+1/2} + \frac{\Delta t}{2} M^{-1} f^{n+1}
\]

\[
v^{n+1} = v^{n+1}(1/(1.0 - \frac{\Delta t}{2} \gamma)),
\]

(3.13)

and implemented in MDL as follows:

phys.velocities += 0.5*self.dt*phys.invmasses*forces.force
phys.velocities *= (1/(1.0-0.5*self.dt*self.gamma))

Note that in this particular case, I do not need and thus do not provide a
finish() method. Like Leapfrog, this method is second-order accurate due to the
similar kick on positions.

Example: Nose-Poincaré

Sampling canonical (NVT) ensembles can be particularly useful when modeling
a system at a fixed temperature, or for obtaining conformational states outside of
energy 'wells' which are not overcomable by an NVE method. Constant tempera-
ture simulations began with artificial scaling of velocities after each iteration of an
NVE method such as Leapfrog to keep the average kinetic energy \(< KE > = \frac{3}{2} kT\)
constant [207], and later with Gaussian thermostatting [56]. The Andersen ther-
mostat [7] runs standard Leapfrog propagation but after the final half kick, exposes
the system to a heat bath and simulates random collisions with the bath using a
Poisson distribution, while providing atoms which do collide with a Gaussian velocity distribution using a zero mean and a standard deviation equal to the square root of the temperature. The Andersen thermostat along with Langevin dynamics provided stochastic methods for modeling isothermal dynamics but in these cases careful control over random collision frequency must be ensured. Later Nosé [149] illustrated a deterministic method which actually extends the system being modeled with artificial heat bath coordinates $s$ with associated masses $Q$ and momentum $p_s$.

Nosé’s resulting method conserved the following quantity:

$$H_{\text{NOSE}} = \sum_i \frac{\tilde{p}_i^2}{2m_is^2} + U(x_i) + \frac{p_s^2}{2Q} + gkT\ln s$$  \hspace{1cm} (3.14)

where $x_i$ corresponds to real positions and $\tilde{p}_i$ as their canonical momenta, which is related to the real momenta $p$ by $p = \tilde{p}/s$. $p_s$ is the canonical momenta associated with $s$, and $g$ is the number of degrees of freedom in the system plus one. Phase space updates which conserve this quantity can be shown to sample the canonical ensemble with respect to $x$ and $p$ under the assumption of ergodicity, which states that if we were to allow a molecular system to evolve indefinitely, it will eventually reach all possible conformations, i.e.:

$$\int A(x,p)\rho(x,p)\,dq\,dp = \lim_{\tau \to \infty} \frac{1}{\tau} \int_0^\tau A(x(t),p(t))dt$$  \hspace{1cm} (3.15)

In the Nosé scheme, the new momenta $\tilde{p}$ scaled the real momenta by a factor of $\frac{1}{s}$, which imposes the same scaling on the timestep $\Delta t$. Hoover [90] later modified the method through a change of variables using a friction coefficient which avoids this timestep scaling but also resulting in a non-Hamiltonian formulation. As an alternative, Bond et al. [20] proposed a method which applies a Poincaré time transformation to Nosé’s approach, resulting in a Hamiltonian formulation that avoids
timstep rescaling and samples the canonical ensemble under the same ergodicity assumption:

\[
H_{\text{NOSE-POINCARE}} = s \left( \sum_i \frac{\tilde{p}_i^2}{2m_is^2} + U(x_i) + \frac{p_s^2}{2Qs} + gkTlns - H_0 \right), \tag{3.16}
\]

which keeps the difference between the value of the Nose-Hoover Hamiltonian and its initial value \(H_0\) constant. I can set up a Nosé-Poincaré propagator in MDL by creating a file `NosePoincare.py`. The Nosé-Poincaré propagator requires the following inputs:

1. Initial Kelvin temperature.
2. Initial values of thermostat position and momentum.
3. Thermostat mass \(Q\).

I can register the propagator, these associated parameters, and defaults with the MDL `PropagatorFactory` as follows:

```python
name="NosePoincare"
parameters=('temp', 500,
          'pS', 0.0,
          's', 1.0,
          'Q', 10.0)
```

I define my class, once again as a concrete implementation of `STS`:

```python
class NosePoincare(STS):
```

In the initialization routine, I set some helper member variables, and calculate forces once. I save the Boltzmann constant, the old thermostat position, a conversion from kinetic energy to temperature, the product of the degrees of freedom, Boltzmann constant and Kelvin temperature, initial potential energy and initial value for \(H_0\):
def init(self, phys, forces, prop):
    self.k = Constants.boltzmann()
    self.sold = self.s  # Old thermostat value
    self.gkT = 3.0*(phys.numAtoms()-1.0)*self.k*self.temp
    self.KEtoT = 2. / (3.0*(phys.numAtoms()-1.0)*self.k)
    prop.calculateForces(forces)
    self.PE = forces.energies.potentialEnergy()
    self.h0 = self.Potnl + forces.energies.kineticEnergy(phys)

Nose-Poincare propagation occurs in seven steps:

1. A half-timestep update to conformational momenta.
2. A half-timestep update to thermostat momenta.
3. A full-timestep update to thermostat and conformational positions.
4. Calculate forces and save potential energy.
5. A half-timestep update to thermostat momenta.
6. A half-timestep update to conformational momenta.

I write each step as a method of class NosePoincare, and invoke each method
from the run() routine below:

def run(self, phys, forces, prop):
    self.updateP(phys, forces, prop)
    self.updatePs(phys, forces, prop)
    self.updateSandX(phys, prop)

    prop.calculateForces(forces)
    self.Potnl = forces.energies.potentialEnergy()

    self.updatePs2(phys, forces, prop)
    self.updateP2(phys, forces, prop)

The first step to Nose-Poincare is a half-step update of conformational momenta,
represented in the following equation with respect to $\tilde{p}$:

$$\tilde{p}_i^{n+1/2} = \tilde{p}_i^n + \frac{\Delta t}{2}s^n F(q)$$  \hspace{1cm} (3.17)
Dividing both sides of the equation by $s^n$, we obtain an update of the canonical momenta $p$:

$$p_{i}^{n+1/2} = p_{i}^{n} + \frac{\Delta t}{2}F(q)$$ \hspace{1cm} (3.18)

and then multiplying each side by $M^{-1}$, the update for velocity:

$$v_{i}^{n+1/2} = v_{i}^{n} + \frac{\Delta t}{2}M^{-1}F(q)$$ \hspace{1cm} (3.19)

Expressable in MDL as:

```python
def updateP(self, phys, forces, prop):
    phys.velocities += forces.force*0.5*self.dt*phys.invmasses
```

The second step is an update of the thermal momenta:

$$p_{s}^{n+1/2} = p_{s}^{n} + \frac{\Delta t}{2} \left( \sum_{i} \frac{1}{m_{i}} \left( \frac{\tilde{p}_{i}^{n+1/2}}{s^{n}} \right)^{2} - gkT \right) - \frac{\Delta t}{2}\Delta H(q^n, \tilde{p}^{n+1/2}, s^n, p_{s}^{n+1/2})$$ \hspace{1cm} (3.20)

Equivalent to an expression involving $C$, the solution to a scalar quadratic equation:

$$p_{s}^{n+1/2} = \frac{-2C}{1 + \sqrt{1 - C\Delta t/Q}}$$ \hspace{1cm} (3.21)

where:

$$C = \frac{\Delta t}{2} (gkT(1 + ln s^n) - \sum_{i} \frac{(\tilde{p}_{i}^{n+1/2})^2}{2m_{i}(s^n)^2} + V(q^n) - H_0) - p_{s}^{n}$$ \hspace{1cm} (3.22)

In MDL, I can first compute $C$ and then use it to compute the new thermal momenta:
def updatePs(phys, forces, prop):
    C = 0.5*self.dt*(self.gkT*(1.0 + log(self.s))
    -forces.energies.kineticEnergy(phys)+self.Potnl-self.h0)
    -self.pS
    self.pS = -2*tempC / (1.0 + sqrt(1.0 - C*self.dt/self.Q))

Next, I update both sets of positions. The thermostat positions $s$ are updated first:

$$s^{n+1} = s^n + \frac{\Delta t}{2} \left( s^{n+1} + s^n \right) \frac{p_s^{n+1/2}}{Q}$$  \hspace{1cm} (3.23)

This can be algebraically transformed into the following:

$$(1 - \frac{\Delta t p_s^{n+1/2}}{2Q})S^{n+1} = (1 + \frac{\Delta t p_s^{n+1/2}}{2Q})s^n$$ \hspace{1cm} (3.24)

Or:

$$s^{n+1} = \frac{1 + F}{1 - F} s^n$$ \hspace{1cm} (3.25)

where:

$$F = \frac{\Delta t p_s^{n+1/2}}{2Q}$$ \hspace{1cm} (3.26)

I write this in MDL using the following code snippet. Since I will need the old thermostat positions to update conformational positions, I save them in a new member variable bathPold:

def updateSandX(self, phys, prop):
    self.sold = self.s
    F = 0.5*self.dt*self.pS/self.Q
    self.s = (1.0 + tempF)/(1.0 - tempF)

Positions are then updated using the previous and current thermostat positions:
\[ x_{i}^{n+1} = x_{i}^{n} + \frac{\Delta t}{2} \left( \frac{1}{s_{n+1}} + \frac{1}{s^{n}} \right) \tilde{p}_{i}^{n+1/2} \] (3.27)

which is expressible in terms of the conformational velocity of particle \( i \) by multiplying \( \tilde{p}_{i} \) by \( \text{sm}_{i} \):

\[ x_{i}^{n+1} = x_{i}^{n} + \frac{\Delta t}{2} \left( \frac{1}{s_{n+1}} + \frac{1}{s^{n}} \right) v_{i}^{n+1/2} \] (3.28)

And implemented in MDL:

```python
self.positions += self.sold*0.5*self.dt*self.sold*
(1.0/self.s+1.0/self.sold)*phys.velocities
```

The next step is another half-timestep update of thermostat momenta, according to the following:

\[ p_{s}^{n+1/2} = p_{s}^{n+1/2} + \frac{\Delta t}{2} \left( \sum_{i} \frac{1}{m_{i}} \left( \tilde{p}_{i}^{n+1/2} \right)^{2} - gkT \right) - \frac{\Delta t}{2} \Delta H(q^{n+1}, \tilde{p}_{s}^{n+1/2}, s^{n+1}, p_{s}^{n+1/2}) \] (3.29)

Note this equation is slightly different than the first half-timestep update of thermostat momenta, whose right hand side also contained the new value for the momenta. In this case, if I expand \( \Delta H \):

\[ p_{s}^{n+1} = p_{s}^{n+1/2} + \frac{\Delta t}{2} \left( \sum_{i} \frac{1}{m_{i}} \left( \tilde{p}_{i}^{n+1/2} \right)^{2} - gkT \right) - \frac{\Delta t}{2} \left( \sum_{i} \frac{1}{2m_{i}} \left( \tilde{p}_{i}^{n+1/2} \right)^{2} + V(x^{n+1}) \right) + \frac{p_{s}^{n+1/2}}{2Q} + gkT \ln s^{n+1} - H_{0} \] (3.30)

Then combine like terms:
\[ p_s^{n+1} = p_s^{n+1/2} + \frac{\Delta t}{2} \left( \sum_i \frac{1}{2m_i} \left( \tilde{p}_i^{n+1/2} \right)^2 - gkT \right) - \frac{\Delta t}{2} \left( V(q^{n+1}) + \frac{p_s^{n+1/2}^2}{2Q} \right) + gkT \ln s^{n+1} - H_0 \] (3.31)

And factor:

\[ p_s^{n+1} = p_s^{n+1/2} + \frac{\Delta t}{2} \left( \sum_i \frac{1}{2m_i} \left( \tilde{p}_i^{n+1/2} \right)^2 - gkT \left( 1 + \ln s^{n+1} \right) - V(x^{n+1}) - \frac{p_s^{n+1/2}^2}{2Q} \right) + H_0 \] (3.32)

If we then recognize the expression for total kinetic energy \( KE \):

\[ KE = \sum_i \frac{1}{2} m v_i^2 = \sum_i \frac{1}{2m_i} p_i^2 = \sum_i \frac{1}{2m} \tilde{p}_i^2 \] (3.33)

The final update can be expressed as:

\[ p_s^{n+1} = p_s^{n+1/2} + \frac{\Delta t}{2} \left( KE \left( \frac{s^n}{s^{n+1}} \right)^2 - gkT \left( 1 + \ln s^{n+1} \right) - V(x^{n+1}) - \frac{p_s^{n+1/2}^2}{2Q} \right) + H_0 \] (3.34)

And in MDL:

```python
def updatePs2(self, phys, forces, prop):
    self.pS += 0.5*self.dt*(forces.energies.kineticEnergy(phys) - (self.sold/self.s)**2 - self.gkT*(1.0+log(self.s))-self.Potnl+self.h0-0.5*self.pS*2/self.Q)
```

Finally I update conformational momenta according to the following:

\[ \tilde{p}_i^{n+1} = \tilde{p}_i^{n+1/2} + \frac{\Delta t}{2} s^{n+1} F(x^{n+1}) \] (3.35)
Which, I can once again convert to equations involving system velocities through
\[ \dot{p} = ps \] and \[ p_i = mv_i : \]

\[ v_i^{n+1} = v_i^{n+1/2} \frac{s_i^{n+1/2}}{s_i^{n+1}} + \frac{\Delta t}{2} M^{-1} F(x^{n+1}) \] (3.36)

Represented in MDL as:

```python
def updateP2(self, phys, forces, prop):
    phys.velocities *= (self.sold/self.s)
    phys.velocities += forces.force*0.5*phys.dt*phys.invmasses
```

The above implementation of Nosé-Poincaré has been shown to sample canonically for systems believed to be ergodic. For non-ergodic stiff systems, sampling can be improved through the use of Nosé-Hoover chains [130] which use a different thermostat at each phase space update depending on previous thermostat values such that the average thermostat momentum is kept equal to the value achieved under ergodicity. Sweet [194] illustrated potential timestep and thermostat-controllable subsystem size, and defined the Recursive Multiple Thermostat (RMT) method which applies Nosé thermostat \( M \) times to the original and kinetic terms of Nosé-Poincare:

\[
H_{RMT} = s_1s_2...s_M \sum_{j=1}^{N} \frac{p_j^2}{2m_j s_1^2 s_2^2 ... s_M^2} + V(x) + \sum_{i=1}^{M-1} \frac{p_i^2}{2Q_i s_{i+1}^2 ... s_M^2} + \frac{p_{sM}^2}{2Q_M} + gkTlns_1 + \sum_{i=2}^{M} ((N_f + i - 1)ktlns_i + f_i(s_i)) - H_0
\]

Dr. Sweet implemented and tested his method in MDL, and his implementation is included in Appendix J.
Functions

MDL propagator functions accept user parameters as formal arguments along with MDL top-level objects such as \texttt{Physical}, \texttt{Forces} and \texttt{IO} instances. Propagator methods also expect a number of steps, a timestep and a \texttt{ForceField} instance. In addition, they can be MTS or STS. If they are MTS, they require two more parameters: (1) a Python function handle for the next propagator in the chain, and (2) its arguments; minimally including the number of steps, timestep and a second \texttt{ForceField} but potentially including another propagator, etc. depending on the number of propagation levels. I will begin by illustrating a prototype of BBK using a function for comparison with the earlier class implementation, then prototype and test a more complex method.

For BBK, I begin by creating a file in the directory \texttt{src/propagators/methods}, called \texttt{bbk.py}. The convention in MDL is to begin propagator class names with capital letters and functions with lowercase letters. Like propagator classes, registration with the factory is necessary:

\begin{verbatim}
name="bbk"
parameters=("temp", 300,
          "gamma", 0.5,
          "seed", 1234)
\end{verbatim}

I now define a Python function \texttt{bbk()}, which will perform STS propagation. Parameters are passed to MDL STS propagator methods in the following order:

1. \texttt{Physical} object.
2. \texttt{Forces} object.
3. \texttt{IO} object.
4. Python integer for the number of steps.
5. Python floating point value for the timestep.
6. \texttt{ForceField} object.
7. Extra parameters. As we have seen for BBK, these will include a Kelvin temperature, friction constant $\gamma$ and Python integer seed for random forces. The order in the formal parameter list of these extra parameters must be the same as the order registered with the factory. Note this was not an issue for classes, since parameters became transparently bound as member attributes.

Keeping this in mind, the structure of the function becomes very similar to the propagator class. The `run()` method is equivalently implemented in a simple counter-controlled `for` loop which executes `steps` times, and the initial calculation of forces is performed before this loop. If there were a `finish()` method, it would be placed after this loop.

```python
def bbk(phys, forces, io, steps, timestep, fg, temp, gamma, seed):
    fg.calculateForces(phys, forces)  # initial force calculation
    for step in range(0, steps):
        forceconstant = 2*Constants.boltzmann()*temp*gamma/timestep
        forces.force += forces.randomForce(phys,seed)*sqrt(forceconstant)

        phys.velocities *= (1.0-0.5*timestep*gamma)
        phys.velocities += forces.force*0.5*timestep*phys.invmasses

        forces.force += forces.randomForce(phys,seed)*sqrt(forceconstant)

        phys.velocities *= (1.0/(1.0+0.5*timestep*gamma))
        phys.velocities += forces.force*0.5*timestep*phys.invmasses

        phys.velocities *= (1.0/(1.0+0.5*timestep*gamma))

Example: Simplified Takahashi-Imada

A simplified form of the Takahashi Imada scheme [198] is to compute the force $F$ on modified positions, as shown in Eq. (3.37)

$$Mq^{n+1} - 2q^n + q^{n-1} = F(q^n + \frac{\Delta t^2}{12} M^{-1} F(q^n))$$

(3.37)
I implement this propagation scheme as a Python function `takahashi` which accepts an MDL Physical, Forces, and IO object along with a number of steps `steps`, timestep `dt`, and `ff` which is an MDL ForceField object (the set of forces to compute):

```python
def takahashi(phys, forces, io, steps, dt, ff):
```

I then loop and at each step perform the above computation and collect data using the following sequence of steps:

1. Execute all user-specified IO (plots, files, screen, etc.)
2. Update velocities and positions using Newton’s equations.
3. Calculate forces.
4. Perturb positions using Takahashi and Imada’s method.
5. Calculate forces on perturbed positions.
6. Update physical time.

I use a Python `for` loop and trivially implement steps (1) - (3):

```python
for step in range(0, steps):
    io.run(phys, forces, step, dt)
    phys.velocities += forces.force*dt*phys.invmasses
    phys.positions += phys.velocities*dt
    ff.calculateForces(phys, forces)
```

I then perturb the positions and recompute forces; however after one iteration of the propagation scheme the new positions should not be equal to the perturbed positions, but the position vector computed above. Thus, I first save the current positions in a numpy array `auxpositions`. To avoid copying the pointer to the low level data, I invoke the member function `copy()` of numpy arrays, then perturb and calculate:
auxpositions = phys.positions.copy()
phys.positions += dt*dt/12.0 * phys.invmasses * forces.force
ff.calculateForces(phys, forces)

Finally, I copy members of the auxpositions back into the MDL Physical object. I must avoid a reference copy and losing data once auxpositions loses scope, so I perform an element-by-element copy back into the positions member of phys. Then I update the time member of phys.

    for ii in range(0, phys.numAtoms()*3):
        phys.positions[ii] = auxpositions[ii]
    phys.time = step*dt

The final step is always to register the new propagation scheme with the MDL factory; since this Python module will be placed in the methods directory, it will be successfully registered as a function with the MDL PropagatorFactory. There are no extra parameters for this function, so I register an empty tuple for these:

    name="takahashi"
    parameters=()

I use this method to model the dynamics of 280 argon molecules; as argon is a noble gas there is no bonded forces and only nonbonded van der Waals forces, which I model with the Lennard-Jones potential. I set up the physical system in the standard way, with periodic boundary conditions at a temperature of 106 K:

    phys = Physical()
    io = IO()
    io.readPDBPos(phys, "data/argon_280/argon.pdb")
    io.readPSF(phys, "data/argon_280/argon.psf")
    io.readPAR(phys, "data/argon_280/argon.par")
    phys.bc = "Periodic"
    phys.cellsize = 4.0
    phys.temperature = 106

    io.screen = 1  # WRITE DATA TO THE SCREEN
I then set up the van der Waals interactions, cutting off force computation after 8.0 Å, and smooth using a C1 switching function:

```python
forces = Forces()
ff = forces.makeForceField(phys)
ff.nonbondedForces("l")
ff.params['LennardJones'] = {'algorithm':'Cutoff',
                            'switching':'C1',
                            'cutoff':8.0}
```

As the Takahashi scheme is now registered with the `PropagatorFactory`, I can execute the propagator for 100 steps at timestep of 20 fs:

```python
prop = Propagator(phys, forces, io)
prop.propagate(scheme="takahashi", steps=100, dt=20.0, forcefield=ff)
```

The resulting screen output illustrates energy and volume conservation after roughly 2 ps:

```
... Time: 1800.000 [fs], TE: 197.9445 [kcal/mol],..., V: 21117.72 [Å^3]
... Time: 1820.000 [fs], TE: 197.9404 [kcal/mol],..., V: 21117.72 [Å^3]
... Time: 1840.000 [fs], TE: 197.8980 [kcal/mol],..., V: 21117.72 [Å^3]
... Time: 1860.000 [fs], TE: 197.8832 [kcal/mol],..., V: 21117.72 [Å^3]
... Time: 1880.000 [fs], TE: 197.8878 [kcal/mol],..., V: 21117.72 [Å^3]
... Time: 1900.000 [fs], TE: 197.9373 [kcal/mol],..., V: 21117.72 [Å^3]
... Time: 1920.000 [fs], TE: 197.9368 [kcal/mol],..., V: 21117.72 [Å^3]
... Time: 1940.000 [fs], TE: 197.8759 [kcal/mol],..., V: 21117.72 [Å^3]
... Time: 1960.000 [fs], TE: 197.8733 [kcal/mol],..., V: 21117.72 [Å^3]
... Time: 1980.000 [fs], TE: 197.8972 [kcal/mol],..., V: 21117.72 [Å^3]
... Time: 2000.000 [fs], TE: 197.8771 [kcal/mol],..., V: 21117.72 [Å^3]
```

### Propagator Modifiers

An MDL *modifier* is a Python function which operates on a propagator object and executes at certain points within the scheme. Modifier functions are contained within Python modules in the MDL `src/modifiers` directory. MDL provides six types of modifiers:
1. **Pre-Init:** Executes before the `init()` method.

2. **Post-Init:** Executes after the `init()` method.

3. **Pre-Force:** Executes before forces are calculated (this could happen multiple times within a propagation).

4. **Post-Force:** Executes after forces are calculated.

5. **Pre-Run:** Execute before the `run()` method.

6. **Post-Run:** Executes after the `run()` method.

The control flow of an MDL integrator object is shown in Figure 3.4.

As an example, I could modify Leapfrog to include velocity scaling after every conformational update, keeping the average kinetic energy and the temperature at a constant value, call it $T_0$. This would require a scaling of velocities using Eq. (3.38).

\[ v^{n+1} = v^{n+1} \sqrt{\frac{T_0}{T}} \]  

(3.38)

I can implement this scaling within a modifier `velocitiescale` which would be a `postrun` modifier. After every execution of the propagator `run()` method, I can
invoke the `velocityscale` method, with `obj` as the propagator object assuming
to have a bound member floating point variable `T0`, created during instantiation
through the factories:

```python
def velocityscale(phys, forces, prop, obj):
    phys.velocities *= numpy.sqrt(obj.T0 / phys.temperature())
```

I can then define a new propagator for velocity scaling as just a derived class
of `Leapfrog` assuming that has been implemented elsewhere, adding the modifier.
I provide a default value of 300 K for the target temperature, which the user can
override in a call to `propagate()`:

```python
class VelocityScale(Leapfrog):
    pass

name="VelocityScale",
parameters=('T0', 300)
modifiers = ["velocityscale", "PostRun"]
```

### 3.4.3 Defining New Forces

A new MDL force can be defined as a Python class. This class can subsequently
define any number of member variables and functions, as long as an `evaluate`
member function is provided. This function must not accept any parameters, other
than `self`. If access to core MDL structures is required, these objects can be passed
as formal parameters to the constructor.

For example, suppose I was defining the following harmonic dihedral restraint,
where \( \phi_i(x) \) represents the value of a selected dihedral \( i \) and \( \phi_0 \) represents a `target`
value in radians around which \( \phi_i(x) \) should harmonically oscillate:

\[
V(x) = k(\phi_i(x) - \phi_0)^2
\] (3.39)
The value $k$ controls the strength of the constraint; as the absolute value of $\phi_i(\vec{x}) - \phi_0$ increases, the system becomes less energetically favorable. I define this harmonic dihedral force as a new class, \texttt{HDForce}. For its constructor, I pass the MDL \texttt{Physical} and \texttt{Forces}, along with the dihedral index, target value $\phi$ and $k$, and create analogous bindings as member variables of the class:

```python
class HDForce:
    def __init__(self, phys, forces, phi, num, k):
        self.phys = phys
        self.forces = forces
        self.phi = phi
        self.dihedral = num
        self.k = k
```

After \texttt{evaluate} is called, system energies and forces (now stored in \texttt{self.forces}) must be updated accordingly. Updating the energy is straightforward using the above equation, I first compute the difference between the appropriate dihedral angle and its target \texttt{self.dihedral}, and ensure the difference is between $-\pi$ and $\pi$:

```python
def evaluate(self):
    diff = self.phys.phi(self.dihedral) - self.phi
    if (diff < -numpy.pi):
        diff += 2*numpy.pi
    elif (diff > numpy.pi):
        diff -= 2*numpy.pi
    self.forces.energies.addDihedralEnergy(self.k*diff**2)
```

I then accumulate this energy into the system dihedral energy:

```python
self.forces.energies.addDihedralEnergy(self.k*diff**2)
```

A dihedral is composed of four atoms $i, j, k$ and $l$. To compute the force on atom $i$ for example, I must apply the chain rule when calculating the gradient of the potential:
\[ F_i = \frac{\partial U}{\partial \phi} \frac{\partial \phi}{\partial x_i}, \]  

(3.40)

where \( x_i \) is the position of atom \( i \). Calculating \( \frac{\partial U}{\partial \phi} \) is a straightforward derivative calculation:

\[ \frac{\partial U}{\partial \phi} = 2 \times self.k \times \text{diff} \]

The calculation of \( \frac{\partial \phi}{\partial x_i} \) is more involved, for a complete derivation please view [101]. The next step is to obtain the indices of each individual atom (subtracting one since we will be indexing a Numpy array for the positions):

\[ \text{atomI} = \text{self.phys.dihedral(self.dihedral-1).atom1} - 1 \]
\[ \text{atomJ} = \text{self.phys.dihedral(self.dihedral-1).atom2} - 1 \]
\[ \text{atomK} = \text{self.phys.dihedral(self.dihedral-1).atom3} - 1 \]
\[ \text{atomL} = \text{self.phys.dihedral(self.dihedral-1).atom4} - 1 \]

I then compute \( \vec{r}_{ij} \), \( \vec{r}_{kj} \), and \( \vec{r}_{kl} \) as distances between respective atom pairs.

\[ \vec{r}_{ij} = \text{self.phys.positions[atomJ*3:atomJ*3+3]} - \text{self.phys.positions[atomI*3:atomI*3+3]} \]
\[ \vec{r}_{kj} = \text{self.phys.positions[atomJ*3:atomJ*3+3]} - \text{self.phys.positions[atomK*3:atomK*3+3]} \]
\[ \vec{r}_{kl} = \text{self.phys.positions[atomL*3:atomL*3+3]} - \text{self.phys.positions[atomK*3:atomK*3+3]} \]

I then use the \text{cross} and \text{dot} functions of Numpy and my own \text{norm2} method which calculates a vector normal squared, to compute the force on all four atoms:

\[ m = \text{numpy.cross(rij, rkj)} \]
\[ n = \text{numpy.cross(rkj, rkl)} \]
\[ fi = m \times (-\text{UdPhi} \times \text{norm(rkj)} / \text{norm2(m)}) \]
\[ fl = n \times (\text{UdPhi} \times \text{norm(rkj)} / \text{norm2(n)}) \]
\[ fj = fi \times (-1 + \text{numpy.dot(rij, rkj)}/\text{norm2(rkj)}) \]
\[ - fl \times (\text{numpy.dot(rkl, rkj)}/\text{norm2(rkj)}) \]
\[ fk = -(fi + fj + fl) \]
And finally modify the force vector at the appropriate indices to include these forces:

```python
self.forces.force[atomI*3:atomI*3+3] += fi
self.forces.force[atomJ*3:atomJ*3+3] += fj
self.forces.force[atomK*3:atomK*3+3] += fk
self.forces.force[atomL*3:atomL*3+3] += fl
```

Within a simulation protocol, I can add forces prototyped in Python by first constructing an instance of the object, then invoking the `addPythonForce()` method of the MDL ForceField class:

```python
hd = HDForce(phys, forces, phi=3.0, num=1, k=1)
ff.addPythonForce(hd)
```

### 3.4.4 Interfacing to Other Tools

**Matplotlib**

Matplotlib [132] is a Python 2D plotting library which provides functionality conducive to recording observables and producing professional-style plots. This includes the ability to save plots as image files, zoom in and out, scrolling left and right, etc. To use Matplotlib, the user should set the `useMPL` data member of the IO instance to 1:

```python
io.useMPL = 1
```

Within the IO instance, if this flag is set, four data structures are populated for the purposes of plotting:

1. `xData`: Python dictionary mapping from plot names (same keywords specified by the user in `io.plots`) to a Python list of values for the x axis. Normally `xData` will contain step numbers.

2. `yData`: Python dictionary mapping from plot names to a Python list of corresponding y values for `xData`. Thus `xData` and `yData` will always be the same size.
3. **figures**: Python dictionary mapping from plot names to Matplotlib figure numbers. Each figure is mapped to a unique integer identifier.

4. **mplFigCount**: Number of Matplotlib figures. Each Matplotlib figure for Matplotlib must be constructed by passing an integer which uniquely identifies it. This count is incremented each time a figure is created.

Subsequently, when quantity *name* with value *val* at step *i* is plotted, internally the following steps are followed:

1. If the step is zero, create a Matplotlib figure object, which represents an active Matplotlib figure handle. For information on member functionality, view the Matplotlib documentation. Construct by passing the value of `mplFigCount` as its unique identifier, and then increment `mplFigCount`.

2. Bring `figures[name]` to the front by passing its value to the Matplotlib `figure()` routine.

3. Set the figure’s x-axis label to *Step*, by calling the Matplotlib `xlabel()` routine.

4. Set the figure’s y-axis label to *name*, by calling the Matplotlib `ylabel()` routine.

5. Append *i* to `xData`.

6. Append *val* to `yData`.

7. Invoke the Matplotlib `plot()` routine, passing `xData[name]` and `yData[name]`.

8. Invoke the Matplotlib `draw()` routine.

Figure 3.5 illustrates a plot of total energy, while an MDL simulation of four-atom (or united-atom) butane is running in the background.

**Gnuplot**

Gnuplot Python (Gnuplot.py, [74]) is a Python interface to Gnuplot, an open-source command-driven plotting tool which comes standard on some flavors of Linux. If the `useMPL` data member of the MDL IO module is unset (which is the default), MDL uses Gnuplot.py for plotting purposes. When a user specifies an observable to be plotted, an entry is created within a member Python dictionary `graphs` of
the IO instance, which maps Python string keys (the observables) to corresponding Gnuplot objects. Upon a call to `propagate()`, plots are updated at appropriate frequencies. Axis ranges are determined dynamically by the range of data being plotted using the `set_range()` function from Gnuplot.py, and data is plotted using the `plot()` function.

Numpy

The Numerical Python (Numpy) libraries [150] provide a set of mathematical functionality which include fast multidimensional array operations to Python. MDL requires Numpy version 1.0.3, and uses Numpy arrays to represent matrix and vector data. This includes the following:

1. Atomic positions.
2. Atomic velocities.
Data is exchanged between MDL and precompiled structures through a pointer typecast from a reference to a C array of double-precision values to a Numpy array reference. In general, assuming that the user’s Python interpreter was built with the same compiler as the shared object libraries for MDL, no precision is lost since Python floating-point values are converted to IEEE 754 double precision on most machines [164].

**PyMPI**

PyMPI [163] is a Python interpreter which can execute functionality in parallel through the Message Passing Interface (MPI, [143]). After installing PyMPI, an MDL user can prototype and run simulations through the PyMPI interpreter, gaining access to parallel functionality by importing the `mpi` module:

```python
import mpi
```

A user then gains access to useful functionality for parallel execution, including but not limited to:

- `mpi.size`: Number of processors.
- `mpi.rank`: Unique integer identifier of the current processor (master node is zero).
- `mpi.bcast`: Executed on the master node, accepts a value and broadcasts this value to all slave nodes.
- `mpi.allgather`: Executed on the master node, accepts a variable and collects all corresponding values on slave nodes into a Python list on the master node.

PyMPI is itself a process which is submitted to `mpirun`, i.e.:
mpirun -np 3 pyMPI simulations/FTSM_Alanine.py

would run the simulation protocol specified in the Python file FTSM_Alanine.py on three processors. In Appendix K I include a section from a publication by myself and fellow collaborators, where we used MDL to implement an early version of the Finite Temperature String Method [166], and we currently are using PyMPI to implement an improved version [125].

3.5 MDL Tool Design: Factories

3.5.1 Propagator Factory

The MDL propagator factory provides a mapping from Python strings to methods for instantiating corresponding propagator objects or functions. Each MDL propagator, class or function, must case-sensitively have a unique name. When defining a new propagator object in Python, MDL standard practice defines the module name with the same name as the corresponding class; although this is not required. All that is required is that each module within src.propagators.object define a name attribute and set it equal to a Python string, which uniquely defines the propagator type defined within the module. Similarly with functions, in the src.propagators.methods package.

The propagator factory, an instantiation of the MDL class PropagatorFactory, is a Singleton [6] and performs all propagator registrations in its constructor. The PropagatorFactory contains a Python dictionary data member registry, which maps propagator names (Python strings) to another dictionary, as shown in Figure 3.6. The second dictionary contains all information needed to instantiate a propagator object (or invoke a propagator function); including the constructor or function
name, type (‘object’ or ‘method’; ‘predefined’ for SWIG-wrapped ProtoMol propagators), and parameters passed as a Python tuple of alternating names and default values. If there are no parameters, this tuple is empty.

```
PropagatorFactory

self.registry = {
    'Leapfrog': {'constructor': <class lib.Leapfrog.Leapfrog>,
                 'type': 'protomol',
                 'parameters': ()},
    .... other predefined propagators ....

    'NosePoincare': {'constructor': <class objects.NosePoincare.NosePoincare>,
                     'type': 'object',
                     'parameters': ('temp', 500, 'bathM', 0, 'bathP', 0, 'Q', 10)},
    .... other MDL objects ....

    'takahashi': {'constructor': <class methods.takahashi.takahashi>,
                  'type': 'method',
                  'parameters': ()},
    .... other MDL methods ....
}
....
```

Figure 3.6. The registry data member of the MDL class PropagatorFactory, populated within its constructor. In this particular case, Leapfrog will be there automatically, while NosePoincare and takahashi will be added through subsequent package scans. Once populated, this data member maps uniquely identified Python strings (used within the call to propagate()), to another dictionary which contains the method for constructing the object or calling the function, the type of the propagator, and a tuple containing alternating parameters and default values if necessary.

The steps to registering all propagators are as follows and performed when the Singleton object is created for the factory:

1. Register SWIG-wrapped propagators. The names, parameters and defaults for these propagator objects are hardcoded.

2. Register MDL classes, by loading all Python modules within the src/propagators/objects directory. Any Python file within this directory will be loaded as a module. For each of these modules:
(a) Retrieve the name and parameters attributes from the module. If name corresponds to a registered propagator, display a warning that this registration will be ignored, and continue.

(b) Index the registry data member using name, and set the corresponding entry to another dictionary which has the following mappings:
   - 'constructor' to the constructor method for the propagator class,
   - 'type' to the Python string 'object',
   - 'parameters' to the tuple provided in the parameters attribute of this module.

(c) Repeat step (2), for propagator functions. These are located within the src/propagators/methods directory, and their 'type' entry within the registry will be set to 'method'.

The PropagatorFactory Singleton is referenced again upon a call to the propagate() method of Propagator, which accepts all necessary arguments to perform propagation including the values of any propagator parameters that should not use defaults provided by the factory. For example, I showed an invocation of propagate() using NosePoincaré, passing the number of steps to run, the timestep and a ForceField object - and note the argument provided to the formal parameter scheme is the same as the registered name of the Nose-Poincaré propagator:

prop = Propagator(phys, forces, io)
prop.propagate(scheme="NosePoincare", steps=80, dt=1, forcefield=ff)

I also could have provided values for a subset of parameters:

prop.propagate(scheme="NosePoincare", steps=80, dt=1, forcefield=ff,
               params={'temp':275,
                       'Q':8.0})

Or multiple schemes and parameter sets for MTS:

prop.propagate(scheme=['Impulse', 'LangevinImpulse'],
               steps=100,
               cyclelength=5,
               dt=0.1,
               forcefield=[ff, ff2],
               params={'Impulse':{},
                       'LangevinImpulse':{}},
In fact in the above case, specifying a set of parameters for Impulse is not necessary since it’s empty and that is the default; I just show it for illustration purposes. For the propagate() call the default name for any scheme is 'Leapfrog', and in fact for STS Leapfrog it is not necessary to even specify a scheme; nor is it necessary in MTS when using Leapfrog for the innermost scheme; the number of levels is always governed by the number of force fields in the call to propagate(). MDL currently cannot mix propagator objects with propagator functions; although this will be a future addition. Therefore if using functions anywhere in the chain, the propagator types must uniformly be functions and it will be necessary to specify all names. propagate() invokes the create() method of PropagatorFactory, passing all of the above information and in the case of functions, a Physical, Forces and IO object (provided when the Propagator is constructed) as these are required formal parameters of every propagator function. create() subsequently will return either:

1. The result of a propagator function call, or
2. An instantiated propagator object.

For both cases the registry is indexed; and the corresponding ’type’ is checked to determine if the desired scheme is a propagator function or object.

Obtaining Functions

If the scheme is a function, the factory returns case (1) which is simpler; since all necessary parameters have been passed. A propagator function accepts the following parameters:

{'LangevinImpulse':{"temp":250,
    'gamma':91,
    'seed':100
}}

1. The result of a propagator function call, or
2. An instantiated propagator object.
1. Physical object.
2. Forces object.
3. IO object.
4. Number of steps (Python integer).
5. Timestep (for STS, Python float) or Cyclelength (for MTS, Python integer).
6. ForceField object.
7. Any extra parameters.
8. If MTS, a function handle for the next method in the chain.
9. If MTS, parameters (5)-(8) for the next method in the chain.

Note that indexing the registry using the outermost propagator name provides access to a corresponding 'constructor' (the function itself), along with its corresponding 'defaults' which is a tuple containing alternating parameter names and defaults. The function invocation is thus set up as follows:

1. Set up the function arguments. Start with a tuple arglist of seven arguments consisting of parameters (1)-(6) above.
2. For each parameter $p$ in the 'defaults':
   - If $p$ was given a value in the call to propagate(), append that value to arglist.
   - Otherwise, append the default value for $p$ to arglist, which is the next element in 'defaults'.
3. If there are more functions in the chain, for each function:
   - Append its function handle to arglist.
   - Append its arguments to arglist, as above.
4. Return the application of the outermost function handle to arglist.

This process is illustrated in Figure 3.7, a two-step process where I set up an MTS Impulse-BBK scheme. I illustrate an invocation of the MDL propagate() routine and the method by which its arguments are translated to an argument list arglist for the outer propagator function handle (in this case, `<function impulse>`). As a
first step, the outermost function string identifier indexes the registry data member of the PropagatorFactory, to obtain the <function impulse> handle. Then the arglist is constructed to contain MDL Physical, Forces, and IO member variables followed by the number of steps (100), the outer cyclelength (5), and the outer forcefield ff. Since an MDL impulse propagator does not have any parameters (i.e. the defaults tuple is empty), there are no extra parameters to pass for impulse.

Figure 3.7. Instantiation of the outer propagation function of a two-level MTS Impulse-BBK scheme. The registry is indexed with the identifier name of the propagator (Python string), obtaining the constructor, type of the propagator, and a tuple containing parameters and default values (in this case none). Then the number of steps, cyclelength and corresponding force field are appended to its argument list.

The arglist is completed in Figure 3.8, where the inner propagator identifier string bbk indexes the registry. In this case, the <function bbk> handle is
passed as the next argument in `arglist`. Its arguments subsequently follow; since `bbk` is the innermost method in my MTS chain, the factory appends the timestep `dt`, and its forcefield `ff2`. In this case, the registry associates `bbk` with a non-empty default tuple, and so values for the three parameters `temp`, `gamma`, and `seed` must be appended to `arglist`. The user provided values for `temp` and `seed`, so the default value is used for `gamma`.

\[
\text{prop.propagate(scheme= ["impulse", "bbk"],}
\text{steps=100,}
\text{cyclelength=5,}
\text{dt=0.2,}
\text{forcefield= [ff, ff2],}
\text{params= ("bbk":
\{"temp": 285,
"seed": 101\})}
\]

---

**Figure 3.8.** Instantiation of the inner propagator in the two-level Impulse-BBK scheme. Once again, the propagator name is used to index the registry. This time, the function handle is appended to `arglist`, and thus will eventually be a formal parameter for the invocation to `impulse()`. The timestep and `ForceField` are also passed. Since the `bbk` propagator contains parameters and corresponding default values, user-supplied values are bound to parameters if supplied; if not supplied then defaults are assumed.
Obtaining Objects

For propagator classes, the PropagatorFactory returns a fully instantiated Python object representing the propagator. This includes parameters, force fields, and subsequent propagators in the chain if the scheme is MTS. A separate object is instantiated for each propagator in the chain. Each propagator contains a reference for the next propagator in the chain, which is NULL if the propagator is the innermost.

Recall that the PropagatorFactory contains a registry which maps propagator names to data for a constructor, type and defaults (tuple of alternating parameter names and default values). For propagator classes, constructor maps to a method handle for the actual Python class constructor. SWIG-wrapped propagators are instantiated slightly differently than Python propagators; I describe each in detail below.

For SWIG-wrapped propagators:

1. Initialize an empty argument list arglist.
2. Append the timestep (or cyclelength) to arglist.
3. For each parameter p in the 'defaults':
   - If p was given a value in the call to propagate(), append that value to arglist.
   - Otherwise, append the default value for p to arglist, which is the next element in 'defaults'.
4. Append the corresponding ForceField to arglist.
5. If this is not the innermost propagator, append a recursive call to create() with the next propagation scheme in the chain and the rest of the arguments to create().
6. Return an application of the constructor method handle to arglist.

The main difference between SWIG-wrapped and Python propagators is that parameters specified within the definition of a Python propagator become bound as
member variables of that Python class; whereas with SWIG-wrapped propagators their values are passed to the constructor. So to instantiate a Python propagator object:

1. If this is the innermost propagator:
   - Apply its constructor to create a propagator object, passing the corresponding force field.
   - Create a member variable \( dt \) and assign to it the value passed to \( \text{propagate()} \).

2. Otherwise:
   - Recursively call \( \text{create()} \) to instantiate the next propagator in the chain.
   - Apply its constructor to the corresponding force field and the next propagator object in the chain (now instantiated).
   - Create a member variable \( \text{cyclelength} \) and assign to it the value passed to \( \text{propagate()} \).

3. For each parameter \( p \) in the 'defaults':
   - Create a member variable \( p \) for the propagator object.
   - If \( p \) was given a value in the call to \( \text{propagate()} \), assign this value to the member variable.
   - Otherwise, assign to the member variable the default value of \( p \).

4. Create member variables to hold lists of modifiers.

Obtaining Modifiers

Propagator modifiers become registered with the MDL PropagatorFactory upon encountering a modifiers variable within one of the propagator modules. Note that the encompassing propagator module must contain a propagator class, since with a function there is no clear division of functionality between initialization and execution. The propagator designer provides the name and type (preinit, postinit, etc.) of the modifier. After the propagator itself has been registered, the PropagatorFactory member function registerModifier() is invoked, accepting
Python strings for the name of the modifier, the type of the modifier and the corresponding propagator. A resulting three-element tuple consisting of the modifier handle, type, and propagator name is registered within the member Python dictionary `modifier_registry`:

```python
def registerModifier(name, type, propname):
    myModule = _get_mod('modifiers.'+name)
    self.modifier_registry[name]={
        'constructor':getattr(myModule, name),
        'type':thetype,
        'prop':theprop
    }
```

When a propagator object with modifiers is instantiated, it is automatically provided six member Python lists to hold modifier method handles: `preinitmodifiers`, `postinitmodifiers`, `preforcemodifiers`, `postforcemodifiers`, `prerunmodifiers`, and `postrunmodifiers`. Then the `PropagatorFactory` linearly iterates through the `modifier_registry`, and for each modifier whose `prop` member is equal to the name of the propagator being instantiated, the modifier handle (stored in `constructor`) is appended to the appropriate member list of the propagator object.

The `propagate()` method then proceeds as follows for MDL propagator objects:
1. Execute all `preinitmodifiers`.
2. Invoke the `init()` function.
3. Execute all `postinitmodifiers`.
4. Execute all `prerunmodifiers`.
5. Invoke the `run()` function.
6. Execute all `postrunmodifiers`.
7. Invoke the `finish()` function.

The `calculateForces()` member function of `Propagator` proceeds as follows:
1. Execute all `preforcemodifiers`.
2. Compute forces.
3. Execute all `postforcemodifiers`. 
3.5.2 Force Factory

The MDL ForceFactory is referenced when MDL Forces are built. Upon construction, the ForceFactory sets up Python dictionaries containing mappings from simulation parameters to various SWIG-wrapped force object constructor handles to avoid an object instantiation for every possible force object. For bonded forces the mapping is simple, and uses a Python string for spatial boundary conditions as a key, i.e.:

```python
self.bondForces = {'Vacuum': BondForce.BSF_Vacuum,
                   'Periodic': BondForce.BSF_Periodic}
```

BondForce is a Python module containing Python wrappers, and BSF_Vacuum and BSF_Periodic are both constructors for the appropriate force object depending on the boundary conditions. Similar mappings are performed for covalently bonded three-atom angles, and four-atom dihedrals and impropers. For nonbonded forces the mapping contains more dependencies, namely the algorithm for computing pairwise forces and any switching function(s) that should be applied. For van der Waals forces, the possible algorithms are:

1. SimpleFull: All pairs.
2. Cutoff: All pairs within a certain cutoff distance in angstroms.

For electrostatic forces, in addition to the above three more algorithms are possible:

1. Ewald: The Ewald method [57, 59]. Ewald summations compute pairwise electrostatic forces in $O(N^{3/2})$ time while maintaining long range behavior, an advantage over using simple cutoffs. These summations assume that each atom is surrounded by a Gaussian cloud with a total charge which cancels the point charge $q_i$ of the atom, and compute short range pairs within a distance where $q_i$ is not completely screened, effectively using a cutoff and thus an $O(N)$ method. Long range forces are approximated by assuming a continuous charge density and performing a Fourier transform, which will compute the
total electrostatic potential at any point \( r_i \). There is an underlying assumption that the entire system is neutral, i.e. \( \sum_i q_i = 0 \). This summation is \( O(N^2) \). With an appropriate choice of the Gaussian cloud width, for systems of fewer than ten thousand atoms the algorithm is efficient and quite accurate; running in \( O(N^{3/2}) \) time.

2. **PME**: Particle Mesh Ewald [44]. PME modifies the long range Ewald summation term by approximating the charge continuum with points on a mesh and solving a discretized Poisson equation using Fast Fourier Transform (FFT, [160]) which takes \( O(N \log N) \) time, as opposed to computing interactions between each atom and a set of charge clouds which is quadratic. This savings increases the size of applicable simulations by a factor of ten to one hundred over straight-up Ewald sums, at the cost of some accuracy to the approximation on a mesh.

3. **MultiGrid**: MultiGrid summation [99], extending the \( \theta(N) \) methods of [23, 178] to periodic boundary conditions. Like Ewald, these summations split the electrostatic calculations into a short and long range part based on a cutoff value, the latter of which is termed the smooth part. The short-range part is computed directly. For the long range part, a function for the potential is obtained by a multiple-step process. The atomic point charges are interpolated onto successively coarser grids, each with a granularity decreasing by a factor of two. Then the interpolated point charges on the coarsest grid are used to determine potential values at these same points on the coarsest grid. These are then anterpolated to successively finer grids, eventually reaching the atomic granularity. Since potential computation is performed at the coarsest grid, the total cost is estimated by a recursive function with a linear inter/anterpolation contribution at successive steps each half as large, resulting in a geometric series that is bounded by a first-order term with respect to \( N \).

There are also various switching functions defined, which smooth forces to zero at cutoff points. A switching function \( S(r_{ij}) \) is a function of the pairwise distance \( r_{ij} \) between two atoms \( i \) and \( j \). When calculating a force \( F(r_{ij}) \) between two nonbonded pairs of atoms, the force is scaled by the switching function as shown in Eq. (3.41).

\[
F(r_{ij}) = S(r_{ij})F(r_{ij})
\]  

(3.41)

The following switching functions are available in MDL:

1. **Universal**: No smoothing, no cutoff.

2. **Cutoff**: No smoothing, simple cutoff at the appropriate distance.
3. **C1**: Apply the switching function in Eq. (3.42), which has a continuous first derivative and $r_c$ the cutoff distance:

$$S(\vec{r}_{ij}) = \begin{cases} 1 - \frac{3}{2} |\vec{r}_{ij}| r_c^2 - \frac{1}{2} |\vec{r}_{ij}|^3 r_c^{-3} & \text{if } |\vec{r}_{ij}| \leq r_c \\ 0 & \text{if } |\vec{r}_{ij}| > r_c \end{cases}$$  \hspace{1cm} (3.42)

4. **C2**: Apply the switching function in Eq. (3.43), which has a continuous second derivative. $r_c$ is the cutoff distance and $r_0$ the distance at which to begin apply switching; default is 0 A which will apply switching the entire time. This can be set using the `switchon` parameter.

$$S(\vec{r}_{ij}) = \begin{cases} 1 & \text{if } |\vec{r}_{ij}| \leq r_0 \\ \frac{(|\vec{r}_{ij}|^2 - r_c^2)^2 (r_c^2 + 2 |\vec{r}_{ij}|^2 - 3 r_0^2)}{(r_c^2 - r_0^2)^3} & \text{if } r_0 \leq |\vec{r}_{ij}| < r_c \\ 0 & \text{if } |\vec{r}_{ij}| > r_c \end{cases}$$  \hspace{1cm} (3.43)

5. **Shift**: Apply a shift of the potential energy by a factor of $\frac{1}{r_c^2}(1 - \frac{r^2}{r_c^2})$.

6. **Cn**: A switching function of flexible continuity. This was used in [192] when studying the effects of switching functions of variable smoothness on conservation within Shadow Hamiltonians, and applies the following:

$$S(\vec{r}_{ij}) = \begin{cases} 1 & \text{if } |\vec{r}_{ij}| \leq r_0 \\ \sum_{k=0}^{2n+1} \gamma_k (\frac{r_{ij} - r_c}{r_0 - r_c})^k & \text{if } r_0 \leq |\vec{r}_{ij}| < r_c \\ 0 & \text{if } |\vec{r}_{ij}| > r_c \end{cases}$$  \hspace{1cm} (3.44)

Coefficients $\gamma_k$ for different values of $n$ can be found in their paper. They are based on boundary conditions with values of zero for $S(r)$ and $S'(r)$ at the switchon and cutoff distances.

7. **CmpCnCn**: Complement to the Cn switching function $S(r)$, equal to $1 - S(r)$.

Mappings in MDL nonbonded force dictionaries are from Python strings for boundary conditions, algorithm, and switching function, to force objects. Below I show a portion of the contents of the member `1jForces` dictionary of the MDL `ForceFactory`:
self.ljForces =
{'Vacuum':
{'SimpleFull':SimpleFullForce.NSFSF_V_U_L,
'Cutoff':
{'C1':CutoffForce.NCSF_CCM_OAPVBC_C1SF_LJF,
'C2':CutoffForce.NCSF_CCM_OAPVBC_C2SF_LJF,
'Cutoff':CutoffForce.NCSF_CCM_OAPVBC_CSF_LJF,
'Shift':CutoffForce.NCSF_CCM_OAPVBC_SSF_LJF,
'Cn':CutoffForce.NCSF_CCM_OAPVBC_CNSF_LJF,
'CmpCnCn':CutoffForce.NCSF_CCM_OAPVBC_CCNCNSF_LJF},
'Appropriate force objects are instantiated depending on the user-specified values
for these three Python strings. Subsequently, depending on these values, parameters
can be specified (for example, a Cn switching function will of course need a value
for its order n). Extra parameters are set within the params member dictionary of
MDL ForceFields:

ff.params[‘Coulomb’] = {‘algorithm’:‘Cutoff’,
‘switching’:‘Cn’,
‘order’:2}

For the following algorithm/switching function pairs, the following parameters
are available, with corresponding types and default values:

1. **SimpleFull**
   - blocksize (integer, default 32): This is used for efficiency purposes, as-
     suming all pairs of atoms are represented in an N x N matrix, this controls
     the size of the submatrix which is evaluated at once (if greater than N, a
     simple two-level nested loop is executed looping over all pairs). Tweaked
     properly, this parameter can be used to minimize cache misses.

2. **Cutoff**
   - cutoff (float): Pairwise Å distance at which forces are assumed to be
     zero.
   - switchon (float): Å distance at which to turn on switching if using a
     switching function.
   - switchoff (float): Å distance at which to turn off switching (defaults
to cutoff value).
   - order (integer): If a Cn or CmpCnCn switching function is used, the order.
In addition, there are several user-specified parameters for fast electrostatic algorithms (Ewald, PME and MultiGrid). For a full description, see Appendix M.

3.6 Observations

3.6.1 Contributions

My goals in developing MDLab were to provide MD experimentalists with a tool for development and testing of various types of numerical methods and application to biological systems. I argue that the language has met these goals by illustrating the following.

1. **MDLab can prototype several different types of mathematical models applicable in molecular dynamics simulations.** I have demonstrated the use of MDLab to prototype both propagation schemes and force calculators, and particularly in the case of propagators I offer the user representation flexibility by allowing both Python classes and Python functions, effectively providing a choice between object-oriented and procedural programming. The propagation methods can be prototyped as single or multiple timestepping, and registration with the MDL factories is transparent with a straightforward interface. By using Numpy arrays, mathematical operations are facilitated both syntactically and in terms of performance, since matrix and array operations are optimized. The domain-specific API allows experimentalists to think and develop in their own domain of expertise, and provides a high level of abstraction to more complex functionality implemented both in pure Python and precompiled binaries from ProtoMol.

2. **MDLab provides a suitable testing environment for these methods.** MDL simulation protocols incorporate the standard I/O formats for molecular dynamics, including the Protein Data Bank and CHARMM PSF and Parameter files, and even Matlab-like XYZ files. Several sample biological systems have been included with the framework, and the user can load their own by downloading appropriate input files, etc. Data can easily be displayed and analyzed, particularly through the interfacing of plotting libraries such as Gnuplot-py and Matplotlib, and the standard molecular dynamics file formats are also supported, some of which (PDB, PSF, DCD) can subsequently be analyzed by visualization engines. The scripting interface and low translational overhead makes viewing the effects of small tweaks and parameter sweeps straightforward and efficient.

3. **MDLab offsets the performance penalty of interpretation by providing selective precompilation at the appropriate granularity:** By
encapsulating a great deal of computationally intensive functionality within precompiled shared object machine code from ProtoMol, in particular including the standard force computation which is known to typically consume 98% of a molecular dynamics simulation, there is a more acceptable penalty running a simulation with MDL vs. straight-up ProtoMol than there would have been had the algorithms been implemented in pure Python. I illustrate this further in Section 3.6.3. Thus selective precompilation allowed me to capitalize upon the testing and debugging benefits of an interpreted malleable scripting language, while retaining performance in most cases.

3.6.2 Design Perspectives

Modularity

If selective precompilation is used, the computational back end will in general require abstractions for the types of methods prototypable in the DSL, so that pure DSL implementations of these methods can be invoked at the appropriate times. If a computational back end was not built in this way and contained a great deal of macro functions performing lots of unrelated operations, one of either the user’s granularity of control (since the entire method would require wrapping) or performance (since the finer scale functionality would require a pure Python implementation) would need to be sacrificed. These principles are analogous to those faced by a GUI designer whose framework must invoke computational back end libraries in response to user commands. Since in general PSEs have a user and computational layer, a modular back end is a resulting likely component of a PSE.

Naming Schemes

The risk of naming conflicts dropped in the case of selective precompilation vs. plugin generation, but this still can happen if we are inheriting computational back end functionality, particularly through dynamic method binding. DSL interfaces which inherit from precompiled back end functionality must in general be used
carefully; since instantiating an object with one of these interfaces and then dynamically adding a new associated attribute (which is easy in Python), can shadow a public member function or data member in the DSL environment. This happened to not be a large issue in MDL, since the only back end modules I inherited from were `STSIntegrator` and `MTSIntegrator` in derived classes `STS` and `MTS`, and the only precompiled methods which I invoked from Python were `initialize()` and `calculateForces()`. Thus as long as a user does not define member functions with these names no naming conflicts take place. The odds are very much against this happening; since the user must define an `init()` method which initializes the propagator anyway, and forces generally will be calculated by the precompiled binaries.

However, there were a few other challenges with SWIG. For instance, SWIG does not provide constructors for abstract classes. At first this does not seem problematic since in C++ objects of any class with pure virtual functions cannot be instantiated. However, if I wanted to design an MDL class which inherited the interface of one of the wrapped abstract classes, this caused problems since in Python constructors of parent classes are implicitly invoked. This forced the use of some `#defines` in the back end source code enclosing some default implementations of formerly abstract functions, and an additional compilation flag. Moreover, certain C++ member function names cause naming conflicts in SWIG-wrapped code; one example I needed to deal with in Python was `defined()`. Multiple functions in the same scope with the same name but different prototypes also causes problems. There were a few other small issues but in summary computational back end updates to `ProtoMol` (modifications or extensions) are not necessarily immediately compatible with MDL; although they will be in most situations. Still, this imposes the requirement that taking advantage of readily available new features provided by
selective precompilation will require some compatibility tests, if SWIG will be used.

Generally speaking, naming conflicts can be avoided in selective precompilation if precompiled functionality is only invoked but not inherited. Sometimes this may be impractical, and if that is the case a designer should be conscious of naming conflicts, and should provide experimentalists with a list of names which are off-limits.

Type Compatibility

With the potential performance hurdles of designing numerical methods in Python, it was essential to provide users with Numpy arrays for low-level structures which represent conformational space, such as positions and velocities. Initially however, the analogous structures in the PROTOmOL computational back end for positions and velocities were STL vectors of C structs, each of which contained three coordinates (collectively termed a Vector3DBlock). Since PROTOmOL pairwise force computation algorithms operated on these latter data structures, an MDL position vector could not be passed to the precompiled libraries without a conversion. The user would then be left with two impractical options with respect to performance: (1) perform an \( O(N) \) copy algorithm between structures each time these structures were referenced, or (2) write analogous pairwise functionality in Python. To fix this, I had to create a linear version of PROTOmOL with a slight tweak in the implementation of Vector3DBlock, while encapsulating a low-level C array bound to the name myData. I kept its API mostly the same, so only minimal changes to other computational back end modules were required. Then using SWIG, I extended Vector3DBlock with a setData() method in its interface file:
This routine handles loading Numpy, setting dimensions and the memory address of the back end structure, and any garbage collection for data previously stored in the C array. Then in the MDL Physical class, I created bindings for both a numpy array (positions) and a Vector3DBlock (posvec), and always ensured they referenced the same data by overriding the implicitly defined `__setattr__` method of Physical:

```python
def __setattr__(self, name, val):
    if (name == 'positions'):
        posvec.setData(val)
    elif (name == 'velocities'):
        velvec.setData(val)
    else:
        self.__dict__[name] = val  # Usual
```

Error and Exception Handling

Similar to BioLogo, error and exception handling in MDLab occurred at three levels: the domain-specific level, the Python level, and in the precompiled binaries. Standard Python errors and exceptions are caught by the Python interpreter and these could be customized accordingly through handlers written in Python, and exceptions while executing precompiled functionality are caught by the computa-
tional back end libraries of PROTOmOL. I handled domain-specific error checking by establishing pre- and post- conditions for API functions, and exiting gracefully upon failure to meet one of these criteria. Since Python is dynamically typed, in the future this can be expanded to include parameter type checking before subroutine execution, avoiding the more generic type mismatch error output by the Python interpreter.

However, this selective precompilation did place a higher degree of pressure upon the error handling at the domain-specific level because now only a subset of back end functionality is executed. There were several cases where in a straight-up execution of the computational back end functionality would be executed in the PROTOmOL main() routine which ensured certain exceptions would not take place; for example a NULL propagator object - that were not necessarily guaranteed without execution of main(). A similar case occurred in file I/O failure, where reader routines returned a boolean true/false which was later checked in main(), and then an exception was issued. So I had to catch these with domain-specific checks. A good thing about selective precompilation however is that a second chance is provided to catch exceptions not caught by the computational back end.

Codependent Evolution

One advantage to runtime interaction with the computational back end is that extensions to PROTOmOL become readily available in MDL without changes to its Python modules, if these updates retain compatibility with SWIG [17]. Upon recompilation of these precompiled binaries, new functionality is subsequently accessible and invocable through the high-level API; and only changes to method prototypes, class APIs or inheritance hierarchies would require modifications in the DSL tier.
Although this is a concern, computational back end extensibility is still facilitated much better with selective precompilation than plugin generation because in this case assuming no change in the DSL grammar, DSL prototyped methods will be compatible with future PSE releases and nothing needs recompilation or regeneration. Selective precompilation also provides more safety, since the DSL does not actually modify lower level tiers,

3.6.3 User Perspectives

The ability of MDL to prototype non-trivial propagation schemes such as Nosé-Poincaré, Recursive Multiple Thermostats and the Finite Temperature String Method illustrates the expressive power and flexibility of the language. By piggybacking Python and taking advantage of malleability and imperative semantics, users gained flexibility along with the ability to prototype methods on the fly, make changes quickly, and test and debug. Python also contains object-oriented constructs which along with packages facilitate modularity, so central data was clearly organized in the MDL API in terms of functionality (force computation, propagation, I/O, etc.) Although MDL requires more general purpose programming knowledge compared to BioLOGO, this was more appropriate for its applications in numerical method prototyping, as opposed to BioLOGO where the central model was a cellular automaton evolving through global and local state changes, where mathematical constructs were localized to rules for updating the lattice.

Generally, if we can have a good estimate $n$ of the execution slowdown of a method executing in a DSL versus within the computational back end, we can estimate a performance penalty due to selective precompilation as:
\[ \text{Penalty}_{SD} = c + (1 - c)n, \quad (3.45) \]

where \( c \) is the typical fraction of CPU time occupied by precompiled modules of the computational back end. A very highly regarded and well-respected colleague of mine, Dr. Robert Skeel from Purdue University, once stated with respect to MDL that ”a factor of 2 is all most of us would accept for a performance hit”. Based on this and several related remarks that I have received through my work with MDL, it seems that performance is indeed the big question mark when it comes to prototyping simulations and numerical methods in Python. I have attempted to make the argument that performance is less important in this case since the most important result is stable and accurate numerical methods which can later be implemented using high-performance software running on supercomputers. Still it is clear that performance cannot be completely neglected. Performance will matter in cases where a long simulation or a large scale molecular system will be required to thoroughly test a new method. These situations could for example occur in implicit solvent models which require a parallel explicit model execution to compare long term behaviors, or in a constraint such as Langevin which involves a certain degree of randomness where a point of data decorrelation is of interest.

I made effort to salvage performance by providing the users with options to invoke SWIG-wrapped, precompiled functionality for propagating the system and calculating forces. In subsequent tables, I illustrate performance penalties resulting from different levels of manual Python prototyping, including tests measuring wall clock time of the STS BBK method represented as precompiled binaries, Python wrappers and a Python class, as well as a two-level MTS Impulse and Leapfrog scheme, shown in Table 3.1. Tests were run on a moderately sized 1101-atom sol-
vated BPTI system, on a 32-bit Intel 686 processor running Redhat Linux, version 9. The simulations used periodic boundary conditions and computed pairwise forces using a direct computation with no cutoff, for 1 ps. For STS BBK, the timestep used was 1 fs, for MTS Impulse/Leapfrog the inner timestep was 0.5 fs and the outer cycle length was 4, thus computing nonbonded forces every 2 fs. These results illustrate a relatively small 15 to 20 percent performance decline when using wrapped propagators from MDL as opposed to running ProtoMol directly. Usage of pure Python to prototype these propagators as methods or objects is not particularly relevant when it comes to performance, as a result the choice of which format to use will depend primarily on user preference. Pure Python also does not have a large performance decline from a straight-up ProtoMol run, only about 31%. Since it is well known that force computation (and particularly pairwise force evaluations) generally occupy the largest portion of molecular dynamics simulation CPU time, I also tested multiple levels of precompilation for a linear time harmonic dihedral force (prototyped earlier in Python, also shown in Table 3.1) and a more expensive, quadratic time electrostatic force evaluation in Table 3.2. For this system, I chose a smaller, unsolvated 22-atom alanine dipeptide molecule and ran Langevin Impulse for 1 ps.

These results indicate that a computationally non-intensive force calculator does not suffer greatly from a pure Python implementation, but a more expensive method will have a high penalty. Since force computation is performed very often, implementing (for example) a quadratic algorithm in pure Python occupies a larger percentage of simulation time without the benefits of precompilation. Thus from a user’s point of view, MDL is very convenient as a tool to implement constraining forces for example, or structural fluctuations, but pairwise evaluations should still be
TABLE 3.1

WALL CLOCK EXECUTION TIMES (S) OF PROPAGATION SCHEMES USING VARYING DEGREES OF PRECOMPILATION.

<table>
<thead>
<tr>
<th>Method</th>
<th>ProtoMol</th>
<th>Wrapped</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBK</td>
<td>79.04 +/- 0.02</td>
<td>91.4 +/- 0.1</td>
<td>103.4 +/- 1.1</td>
</tr>
<tr>
<td>Impulse/Leapfrog</td>
<td>41.376 +/- 0.006</td>
<td>49.74 +/- 0.03</td>
<td>50.35 +/- 0.03</td>
</tr>
<tr>
<td>Harmonic Dihedral</td>
<td>82.30 +/- 0.01</td>
<td>87.8 +/- 0.3</td>
<td>88.37 +/- 0.09</td>
</tr>
</tbody>
</table>

TABLE 3.2

WALL CLOCK EXECUTION TIMES (S) OF FORCE ALGORITHMS USING VARYING DEGREES OF PRECOMPILATION.

<table>
<thead>
<tr>
<th>Method</th>
<th>ProtoMol</th>
<th>Wrapped</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electrostatic</td>
<td>0.92 +/- 0.17</td>
<td>1.1 +/- 0.3</td>
<td>30.1 +/- 0.2</td>
</tr>
</tbody>
</table>

performed by machine code. Since the design of PROTOmOL takes advantage of opportunities to optimize pairwise force computations such as keeping track of pairwise cell lists containing nearby atom pairs, a future goal of MDL could be a hook into the PROTOmOL framework such that pairwise force algorithms developed in Python can be coupled with optimization techniques in the back end. I still hypothesize a large performance penalty in such cases, but the currently non-practical punishment could be reduced to acceptable in certain situations (for example, mesoscale systems).
I have demonstrated how MDL can be run through a PyMPI interpreter to incorporate parallelism while prototyping new methods. Parallelism can also be incorporated in low-level computations which are currently embedded within precompiled binaries - such as atom, force, and spatial decomposition [158]. The source code of PROTO-MOL computational back end, when configured appropriately includes invocations of MPI libraries. The fast electrostatic methods Ewald, PME and MultiGrid are all parallelized using either atom decomposition or force decomposition at the user’s discretion [99]. With the former, subsets of the \( N \) atoms which compose a system are assigned to each processor, and with the latter actual pairwise force computations are assigned to each processor. Calls to MPI libraries from PROTO-MOL source code could not be SWIG-wrapped for MDL, since when running parallel Python code the executable for `mpirun` becomes the `pyMPI` interpreter as opposed to PROTO-MOL itself. For MDL to contain parallel implementations of these algorithms, a Python implementation involving references to modules provided by PyMPI would need to coexist. We could not simply replace the SWIG-wrapped calls with pure Python due to the resulting performance decline of serial simulation runs. Accomplishing this would thus require a complete reimplementation of many of these algorithms in Python but would provide more flexibility for users who have access to computing clusters. MDL simulations could then be run through the `pyMPI` interpreter to gain access to this parallelism.

Another challenge involves the prototyping of new forces. The current method with MDL involves manual modification of both the `energy` and `force` data members of an MDL `Forces` object. Ideally, since force is simply the gradient of energy with respect to atomic positions, the user could just specify a new energy term as a function of atomic positions, and have the gradient computed behind
the scenes. I have attempted this using several different libraries, but each have their impracticalities. SymPy [197] is a useful set of Python libraries which can compute derivatives with respect to independent variables by classifying the latter as *symbols*, subsequently taking expressions involving these symbols and recursively computing derivatives, applying chain rule accordingly if the expression involves functions. However, since atomic positions are a vector, very often this vector will require indexing (for example, if a cross product is computed) - and this operation is not supported for SymPy symbols. Swigimac [196] experienced similar difficulties. Others [84] compute derivatives through various approximations, but I discovered that even in a single and simple force such as harmonic dihedral error propagation becomes large very quickly. Even Matlab [55] experiences difficulties of computing derivatives when the independent variable is a vector. Although a specific strategy of incorporating this goal is not currently clear, its availability would make force prototyping much less painful for the user.

While discussing Matlab [55], it should be noted that many experimentalists in Molecular Dynamics are fluent and prefer the use of Matlab for numerical method prototyping for its elegant syntax, friendly IDE and data analysis capabilities. Matlab has limitations in terms of libraries which perform tasks other than numerical operations and an underdeveloped interface to C/C++. MEX files [113] provide one option as a method of invoking C/C++ code from Matlab, but they require a special gateway routine and data structures and the lack of a wrapper tool analogous to SWIG for this purpose makes this option currently less practical when interfacing a DSL to a computational back end. Nevertheless, I believe this study illustrates the merits of such a tool, and future creation and usage in MDL would result in large scale improvements in acceptance and usability within the MD community.
CHAPTER 4

CONCLUSIONS

I have illustrated the application of two techniques, plugin generation and selective precompilation, to the task of encapsulating a DSL as a middle-level tier in a computational biology PSE. The techniques succeeded in the cases of BioLogo and MDLab of providing a vocabulary natural to experimentalists for expressing phenomena relevant to their domain. Language tool design can be a DSL developer decision, but these techniques for computational back end interfacing are applicable across domains. It is also clear from these case studies that the DSL and computational back end must evolve codependently, and thus an important issue to resolve is where in the software life cycle should these interdependent tiers be designed and implemented. It would be beneficial to develop them together to ensure compatibility between computational back end design patterns, modularity, etc. and the DSL, however if the computational back end is highly dynamic an increased risk of DSL impact costs due to naming conflict resolution and tool incompatibility is evident. Since design modifications are almost always less costly than implementation modifications, the most optimal approach is to design the computational back end and DSL at the same time, but implement and test them sequentially. Costs are then reduced to maintenance, but hopefully the back end will be designed carefully and modularly such that a mostly static core can be maintained. This process is
well-facilitated by the waterfall model [169] since each stage in software development (requirements, design, implementation, testing, maintenance) happens sequentially and discretely with no back-communication. Requirements and design of the DSL and computational back end can be developed together, then the back end can be implemented and tested. The DSL can then be implemented, tested, and interfaced to a work design, and both can be maintained together. This process is shown in Figure 4.1.

Figure 4.1. Modified waterfall model for codevelopment of the DSL and computational back end tiers of a PSE. The requirements and design should happen together with open communication lines, and the computational back end should be implemented and tested before the DSL to minimize dependency propagation overhead between the implementations. Finally, both can be maintained together.

A reality however is that due to inflexibilities in the waterfall model, more pro-
cess iterative approaches are currently popular in software engineering [180], such as incremental change [58]. Incremental development merges requirements, design and implementation into *increments* which each deliver part of the system functionality, freezing requirements at the implementation stage. Codeveloping a DSL using this type of approach is more challenging, because the dependency mapping to back end components (i.e., increments) is more critical. The key observation is that global requirements specification still occurs in a somewhat distinct stage in this model, because requirements must be prioritized to determine the increment order. In this sense DSL and back end requirements can be merged, and maintenance can also be viewed as increments. Developing individual increments adds a single new feature to the framework and involves starting from a baseline, localizing where a change should be made, performing impact analysis, prefactoring, implementation, and postfactoring before formulation of the new baseline. Testing is intermittently performed throughout prefactoring, implementation, and postfactoring. Localization and impact analysis can be performed together on the DSL and computational back end, collecting a set of change locations and dependencies in both tiers. Following the same principles as with the waterfall model (design together, implement sequentially), the actual prefactoring, implementation, and postfactoring can be done sequentially; first with the computational back end and then with the DSL. This 'extended' incremental development is shown in Figure 4.2, which is executed for each increment extension to the PSE.

My conclusions regarding interface strategies address the early stages of the development cycle, namely requirements and design. Post-design language tool implementation strategies [50, 107] and testsuite development and generation [11, 83, 209] have been researched and several approaches can be taken using various software
engineering approaches. Requirements engineering for a DSL involves (1) surveying the experimentalist audience, and (2) studying prototypical mathematical models. Pursuing item (1) can yield results for item (2), since of course dialogue with future users is the best way to obtain potential simulations that will be conducted using the DSL. One advantage that I had with respect to item (1) when interfacing my DSLs was that the computational back ends had been developed and released several times prior to the interface, so a good audience base had been established. However I have already discussed the risks of independent DSL and back end development, and so assuming the computational back end and PSE are being developed together, an audience can be cultivated by surveying literature using for example Web of Science [96], PubMed [161], etc. and establishing contacts, making connections, and proposing the new DSL. Once an audience has been established and enough interest generated, requirements can be formulated based on experimentalist preferences, resources, and envisioned models. Certain considerations will always hold a high priority when it comes to DSLs. Ease of use and thorough exception handling are necessary for experimentalists to prototype, test, and debug in their appropriate domain. However, considerations such as performance, runtime memory consumption, syntax style, abstraction level, and desired translational overhead will vary in priority. For example, performance will be important if (1) the mathematical models being developed are computationally intensive, (2) models will be tested on large scale biological systems, (3) a long simulation must be run to verify the models, or (4) experimentalists have a general lack of computing resources.

Once priorities are established, a decision can be made on the technique for interfacing the DSL. Clearly the appropriate level of abstraction to maximize writability is obtainable with either plugin generation or selective precompilation depending on
organization, syntax and availability of domain operations. Expressive power is thus independent of the interface technique, but the following must still be considered:

1. **Performance.** The edge here goes to plugin generation, since generated plugin extensions contain little to no explicit performance penalty and there are many opportunities to optimize generated code if language tools are properly designed. The only potential performance risk with plugin generation results from the imposition of executing the entire computational back end, as opposed to a subset with selective precompilation. However, this is only an issue if the computational back end invokes a great deal of redundant functionality during general simulation runs, becoming a software design issue in that tier. Selective precompilation offers a great deal of flexibility and opportunity to salvage performance by encapsulating computationally intensive functionality as precompiled binaries, but this also comes at a price. Precompiled functionality can be parameterized such that it is callable and customizable, but not prototypable by the user, so control granularity is reduced. With plugin generation, even computationally intensive functionality can be prototyped and translated into an appropriate plugin extension with solid runtime performance.

2. **Memory consumption.** Plugin generation also has a slight edge here, once again due to optimization potential. A DSL developer has a higher level of control over the amount of memory consumed at runtime by a user-prototyped method compared to selective precompilation, where no translation is performed and experimentalist code is run directly. Chances are that an experimentalist prototyping a method will not be thinking about memory consumption, but will be more interested in numerical behavior - accuracy, stability, etc. With plugin generation a prototyped method can be translated into a memory-efficient plugin implementation if language tools are properly designed.

3. **Error and exception handling.** The two techniques are even here, and each are strong because they provide multiple layers of error and exception handling. For plugin generation, error handling can be performed and messages generated at the domain level using language tools, and translation provides the freedom to generate plugin extensions which perform their own runtime exception handling. With selective precompilation, error handling can be performed using pre and post conditions in the domain-specific API, also at the domain level. Selective precompilation will suffer worse if the computational back end is not designed well in terms of handling unexpected runtime behavior, since we do not have the benefit of generating calls to exception handler routines.

4. **Translational overhead.** Clearly selective precompilation has a huge advantage here, since transparent object instantiation and method invocation is much cheaper than generation and compilation of a new computational back end module such as a plugin. The lack of a complete language-to-language translation [182] in the case of selective precompilation offers sizable benefits in terms of testing and debugging mathematical models and particularly those which involve a great deal of arithmetic, since small tweaks and parameter sweeps will often need to be performed before a method is deemed acceptable.
These costs can be summarized in the cost equation of Sebesta [174], Eq. (4.1) for general programming languages. In this case the language cost has seven components - $C_T$ for training programmers, $C_W$ for writability, $C_C$ for compiling, $C_E$ for execution, $C_{IS}$ for the implementation system, $C_R$ for reliability and $C_M$ for maintenance (note this is for programs written in the language and not the language itself).

$$C = C_T + C_W + C_C + C_E + C_{IS} + C_R + C_M.$$ (4.1)

DSLs should by definition minimize $C_T$ and also $C_W$ with respect to phenomena in their corresponding domain. $C_{IS}$ depends on the language tools designed, if the compilers and interpreters are open source and cross platform, and $C_M$ depends on the syntax and semantics of the language itself. The technique for interfacing the DSL to the computational back end will affect the other three terms. For example, in the case of plugin generation $C_C$ is very high due to the multiple translation steps between DSL and machine code, but $C_E$ should be low in terms of speed and memory. The latter will be higher in the case of selective precompilation but $C_C$ will be much lower. $C_R$ is slightly higher in the case of plugin generation due a larger number of potential naming conflicts, although good exception handling capabilities in both cases will provide benefits to $C_R$. Since presumably at the point of deciding on a technique a developer will have performed audience surveys and obtained an idea of goal prototypical mathematical models, priorities to the above considerations can be assessed and one of these techniques can be used as appropriate. The outline above indicates slight advantages to plugin generation in terms of efficiency both from speed and memory perspectives, and large advantages in terms of reduced translational overhead for selective precompilation. Thus in
summary, plugin generation is a useful technique if the following hold:

1. Based on the scale of the biological systems, the execution time of mathematical models, the lengths of typical simulations, and the computational tools available to experimentalists, performance is a very high priority. This will be the case in for example sequence analysis [15] which requires large scale data, and in the case of Lattice Gas Cellular Automata (LGCA, [168]), if the lattice is large and irregularly shaped, or hexagonal [66], or three-dimensional [181].

2. The hypothesized typical simulation execution time is very high compared to how much translational overhead there ever will be. This requires some presumptions about the quantity of translational overhead, but plugin generation requires the plugin design pattern and so translation will localized to a particular area of the computational back end and should not have to scan and modify the core, so chances are translational overhead will not be larger than the minute timescale, and certainly should not take multiple days as some simulations can. Presumptions must also be made on mathematical models since future prototypes may be novel ideas that are not currently implemented anywhere; however comparable methods and similar biological system size scales can likely be obtained. If model verification simulations are projected to take two days to run for example, even a 1.5-fold performance hit resulting in one more day may not be practical, and an experimentalist may gladly accept waiting a few more minutes upon model tweaks for access to the performance benefits of plugin generation. As an example, a Boussinesq wave model [103] of the December 2004 Indian Ocean Tsunami took 27 hours when executing in parallel on eight processors, and in this case even a highly expensive twenty minute translation would not be too cumbersome, especially in a serial execution. Even in the parallel case, a 1.5-fold hit would result in a 40 hour simulation, so in that case the twenty minutes would be negligible to save half a day of runtime.

3. The computational back end will not change a great deal overtime, or if it will, strong communication exists between the developer of the DSL and the developer of the computational back end such that maintenance efforts can be coordinated; and clear notifications and guidelines can be provided to DSL users for updating their plugin libraries when downloading new releases.

However, plugin generation does place more constraints on the computational back end design by explicitly requiring the plugin pattern. If the plugin pattern is inapplicable to the computational back end design, neither is plugin generation applicable to the interfaced DSL. Selective precompilation enforces the need for back end abstractions for prototypable functionality and is a useful technique in the following situations:
1. Based on the typical mathematical models that experimentalists will be prototyping, a long simulation (in terms of wall clock time) is not necessary to develop and test them. This was the case in for example the land surface model of Bosilovich and Sun [22] which included mesoscale numerical methods designed to predict soil moisture and temperature. This was also the case for the Momentum Impulse Relaxation (MIR, [12]) method of predicting fluid viscosities.

2. The steps involved in testing a numerical method for accuracy will involve large scale parameter sweeps, repeated comparison with experiments, and lots of refinements. If this is the case and (1) holds, the translational overhead associated with plugin generation will most likely be non-negligible and will be particularly inconvenient when frequently executing new tests.

Once goal mathematical models are understood and a design strategy is obtained, syntax design can be performed. Mathematical submodels should be divided into entity-relationship diagrams, for a good idea of necessary functionality and dependencies that could potentially translate to DSL modules and features. The BioLogo survey results indicated that a solid number of experimentalists either understand general programming concepts which are simple or are implemented using a simple enough syntax which they could pick up easily; and that about half preferred this resulting higher expressive power. Thus a DSL developer must not assume that the audience are simply nonprogrammers and that their goal should be to remove as many programmer concepts as possible. After a rough initial design of the syntax, the developer can create a few rough examples of code blocks or modules within the perspective DSL, potentially even using different representations if some points are still undecided. Ideally these blocks should represent some biologically relevant phenomena for which domain experts are familiar, and also be complex enough to illustrate the capabilities the DSL could offer. Once these are submitted to an audience, an idea of which format is generally most pleasing to the eye, most understandable, etc. can be obtained. Preferred operating systems should also be surveyed, to obtain an idea of required DSL system compatibilities. This process
can be iterative, repeating until a solid consensus is established and maximizing the likelihood of future embracing of the DSL.

The potential impact of DSLs in computational biology is significant. Models prototyped and tested through a PSE can provide a deeper knowledge of natural processes, drug design, the human genome project, etc. I look forward to continued development of the BioLogo and MDLab projects, and to future projects which provide this flexibility for the experimentalist, catering to the development of well-tested and accurate mathematical models. As a result of this impact, I believe the DSL role in biological PSE usability continue to grow with time, and hope my work has provided useful insight into techniques for facilitating this process.
Figure 4.2. Modified incremental model for codevelopment of a DSL and an interfaced computational back end. Like the waterfall model, global requirements and maintenance can be performed together. For each increment, localizing and assessing dependencies can be performed together, and actual modifications should be distinct sequential stages, with the computational back end coming first.
APPENDIX A

BIOLOGICAL TERMS: MORPHOGENESIS

- **aggregation** - the gathering of cells together in a mass.
- **anisotropic** - of unequal dimensions along different axes.
- **apoptosis** - cell death, occurring by disintegration into membrane-bound particle which are later expelled.
- **avian** - of or relating to birds.
- **biocomplexity** - the study of complex structures and behaviors that arise from biological entity interactions.
- **blastula** - a spherical cluster of cells in embryonic development resulting from multiple mitotic divisions of the zygote after fertilization and cleavage.
- **Cellular Potts Model (CPM)** - a mathematical model for morphogenetic subprocesses such as cell clustering, growth, division, and death.
- **cell type** - an attribute assigned to cells which provides a method for categorizing by behavior.
- **Cell Type Automaton** - a set of cell types and rules for transitioning between types; analogous to a computational automaton. Under the cell type automaton, each cell can only be of one type at a time and can transition between types depending on internal or external conditions.
- **chemotaxis** - movement of a cell towards or away from an external chemical gradient.
- **chondrogenic** - associated with cartilage formation.
- **circadian** - occurring every 24 hours.
- **clustering** - cellular migration into tight groups.
- **cytoskeleton** - the internal framework of a cell.
- **differentiation** - the process by which a cell changes cell type and follows a different set of behaviors cf. **cell type**
• distal - farthest.
• dorsal - back side.
• diffusion - motion of molecules and ions from regions of high concentration to regions of low concentration, until the point of equilibrium.
• elasticity - the ability to return to a stable form after being deformed.
• flip - in the CPM, the change of an integer index at a lattice pixel to that of one of its neighboring pixels, with Metropolis/Monte Carlo probability dependent upon the resulting change in effective energy.
• fusion - a merging together.
• gene transcription - the transfer of genetic information from DNA to mRNA when the latter is synthesized.
• gradient - the change of a value relative to a change in a variable (i.e., chemical gradient in the CPM is change in chemical concentration over distance).
• grid-based - involving within core functionality a coordinate-based set of points in space.
• growth - cellular increase in volume.
• Hamiltonian - a mathematical function used to generate motion in a changing system, in the case of the CPM a function which governs effective energy which in turn plays a role in dynamics within the CPM.
• haptotaxis - movement of a cell towards or away from a chemical gradient, constrained to an area also dependent upon the gradient.
• induction - causing a process to occur (i.e., stimulating production of an enzyme).
• in silico - on a computer.
• in vitro - outside of a living organism.
• in vivo - within a living organism.
• intracellular - occurring within a cell.
• intercellular - occurring between cells.
• isotropic - independent of direction.
• medium/ExtraCellular Matrix (ECM) - area within which cells are contained
• membrane-bound - constrained to the cell membrane.
- **mesenchymal** - a type of embryonic cell contained in the mesoderm, which is a gelatinous substance containing loosely-packed cells which eventually develops into for example cartilage or bone. This is especially found in chicken limbs.

- **migration** - movement in a given direction.

- **mitosis/division** - the splitting of a cell into two cells of roughly equal volume.

- **morula** - a mass of cells formed during embryonic development after mitotic divisions and cleavage of the zygote.

- **morphogen** - a chemical that forms a gradient and has an effect upon cellular behavior during morphogenesis.

- **morphogenesis** - a set of processes in embryonic development of a multicellular organism, involving pattern formation into bone and organs.

- **no flux boundary conditions** - a set of rules that when applied to a grid, discards points outside of the maximum dimensions.

- **osmotic** - relating to the diffusion of a fluid through a semipermeable membrane.

- **periodic boundary conditions** - a set of conditions that when applied to a grid, implements wraparound using modular arithmetic for points outside of the maximum dimensions.

- **pixel/voxel/lattice point** - in the CPM, a location in the lattice specified by a set of coordinates \((x, y, z)\), with which an integer index greater than 0 is associated to specify the cell at its location, or 0 if medium.

- **proximal** - closest.

- **pseudopod** - a projection of cellular cytoplasms during cell motion.

- **reaction-diffusion (RD)** - change in local concentrations of chemical due to reaction (interaction with other chemicals) or diffusion (flow of chemical concentration from cells).

- **resorption** - the process of absorbing a substance that was once contained.

- **secretion** - the process of generating and excreting a substance.

- **sorting** - the clustering of cells by cell type.

- **spatiotemporal** - in space and time.

- **stochastic** - involving randomness.

- **subcellular** - occurring within a cell or one of its interior structures.

- **substrate** - an underlying layer of an environment.
- **supercellular** - occurring outside a cell.
- **target volume/surface area** - the experimental volume/surface area values to which a cell tends.
- **ventral** - lower.
- **zygote** - the single cell that forms after fertilization, by the union of gametes.
APPENDIX B

BACKUS-NAUR FORM OF BioLogo

```
program -> <CompuCell3D> subprogram cellmodel subprogram </CompuCell3D>

subprogram -> hamiltonian subprogram
              | cellmodel subprogram
              | pdesolver subprogram

hamiltonian -> <Hamiltonian name=" str "> inputs fields step equation
              </Hamiltonian>

inputs -> input inputs
          |

input -> <Input name=" str " inputparams />

inputparams -> type="file"
              | params= "1or2 "

1or2 -> 1
      | 2

fields -> field fields
         |

field -> <Field name=" str " fieldparams />

fieldparams -> file=" str "
              |

step -> <Step> react </Step>
```
react -> <secrete field="str" amount="arithexp" optionalcondition react />
    | <resorb field="str" amount="arithexp" optionalcondition react />

optionalcondition -> condition="boolexpression"
    |

equation -> <Equation> summations <Equation>

summations -> neighborsum summations
    | cellsum summations
    | pixelsum summations

neighborsum -> <neighborsum exp="arithexp" optionalcondition optionallimit />

optionallimit -> limit="arithexp" /
    |

cellsum -> <cellsum exp="arithexp" optionalcondition />
    |

pixelsum -> <pixelsum exp="arithexp" optionalcondition />
    |

cellmodel -> <cellmodel name="str"> useplugins declare celltypes </cellmodel>

useplugins -> useplugin useplugins
    |

useplugin -> <useplugin name="str"/>

declare -> <declare> decset </declare>

decset -> statevar decset
    |

statevar -> <statevar name="str"/>

celltypes -> celltype celltypes
    |

celltype -> <celltype name="str"> creation updatevariables updatecelltypes </celltype>

creation -> <creation> conditionalcopies </creation>
conditionalcopies -> copy copies
   |
   copy -> <copy to=" str " from=" exp " optionalcondition />

updatevariables -> <updatevariables> conditionalcopies </updatevariables>
   |
updatecelltypes -> <updatecelltypes> transitions </updatecelltypes>
   |
transitions -> transition transitions
   |
transition -> <changeif currenttype=" str " condition=" boolexp " />

pdesolver -> <PDESolver name=" str " optionalnormalize > inputs fields pdes
   </PDESolver>

optionalnormalize -> normalize=" trueorfalse "
   |
trueorfalse -> true
   | false

pdes -> <DiffEq field=" str " > terms </DiffEq>
   | <Python> embeddedPython </Python>

terms -> term terms
   |
term -> <Term exp=" arithexp " optionalcondition />

exp -> arithexp
   | boolexp

arithexp -> exp + exp
   | exp - exp
   | exp * exp
   | exp / exp
   | exp % exp
   | exp ^ exp
   | ( exp )
   | sin ( exp )
\begin{align*}
\cos ( \ exp ) \\
\tan ( \ exp ) \\
\arcsin ( \ exp ) \\
\arccos ( \ exp ) \\
\arctan ( \ exp ) \\
str \\
\number \\
\' str \\
\end{align*}

\[ \text{boolexp} -> \ exp \ less \ exp \]
\[ \exp \ greater \ exp \]
\[ \exp \ lessequal \ exp \]
\[ \exp \ greaterequal \ exp \]
\[ \exp \ equal \ exp \]
\[ \exp \ notequal \ exp \]
\[ \exp \ and \ exp \]
\[ \exp \ or \ exp \]
\[ \not \ exp \]
\[ ( \ exp ) \\
\str \\
\number \\
\' \ str \]

Some notes on special cases occurring within the BNF:

- Arithmetic expressions (\textit{arithexp}) and boolean expressions (\textit{boolexp}) can contain some enscripted C++, although this should be used minimally and with care now as full C++ parsing has not been built in.

- \texttt{str} is a string of characters, containing zero or more of A-Z, a-z or 0-9 (no symbols).

- \texttt{number} can be an integer or real value.

- The nonterminal \textit{embeddedPython} can be replaced by zero or more valid Python statements.
APPENDIX C

PYTHON SIMULATION IN CompuCell3D
def mainfcn():
    import sys
    # appending path to CompuCell related Python modules
    from os import environ
    from os import getcwd
    sys.path.append(environ["SWIG_LIB_INSTALL_DIR"])  
    sys.path.append(environ["PYTHON_MODULE_PATH"])  
    sys.path.append(getcwd()+"/examples_PythonTutorial")

    import SystemUtils
    SystemUtils.initializeSystemResources()

    import CompuCellPython
    import CompuCellAuxPython
    import PlayerPython
    CompuCellPython.initializePlugins()

    # Create a Simulator. This returns a Python object that wraps
    # Simulator.
    sim = CompuCellPython.Simulator()

    simthread=PlayerPython.getSimthreadBasePtr();
    simthread.setSimulator(sim)
    simulationFileName=simthread.getSimulationFileName()
    print "simulationFileName=",simulationFileName

    # Add the Python specific extensions
    reg = sim.getClassRegistry()
    CompuCellPython.parseConfig(simulationFileName, sim)

    sim.extraInit()
    simthread.preStartInit()
    sim.start()
    simthread.postStartInit()

    screenUpdateFrequency=simthread.getScreenUpdateFrequency()

    from PySteppables import SteppableRegistry
    steppableRegistry=SteppableRegistry()

    # Part unique to this run
    # Here we will add our first Python steppable
    from steppables import ExtraAttributeCellSort
    extraAttributeCellSort=ExtraAttributeCellSort(_simulator=sim,_frequency=10)
    steppableRegistry.registerSteppable(extraAttributeCellSort)

    steppableRegistry.init(sim)

    steppableRegistry.start()

    for i in range(sim.getNumSteps()):
        sim.step(i)
        steppableRegistry.step(i)
        if not i % screenUpdateFrequency:
            simthread.loopWork(i)
            simthread.loopWorkPostEvent(i)
    sim.finish()
    steppableRegistry.finish()

mainfcn()

Program 18: Example CompuCell3D simulation file which adds the ExtraAttributeCellSort Python steppable.
APPENDIX D

CompuCell3D CONFIGURATION FILES OF BioLogo VERIFICATION SIMULATIONS
D.1 Avian Limb Bud Growth

<CompuCell3D>
  <Potts>
    <Dimensions x="71" y="31" z="281"/>
    <Steps>735</Steps>
    <Temperature>2.85</Temperature>
    <Flip2DimRatio>2</Flip2DimRatio>
    <Boundary_x>Periodic</Boundary_x>
    <Boundary_y>Periodic</Boundary_y>
    <Boundary_z>Periodic</Boundary_z>
  </Potts>
  <Plugin Name="Avian"/>
  <TargetVolume Type1="Condensing">32</TargetVolume>
  <TargetVolume Type1="NonCondensing">32</TargetVolume>
  <LambdaVolume>1</LambdaVolume>
  <Plugin Name="Surface">
    <TargetSurface>77</TargetSurface>
    <LambdaSurface>0</LambdaSurface>
  </Plugin>
  <Plugin Name="LimbChemical">
    <Threshold>0.7</Threshold>
    <Mu>10</Mu>
    <FibroRate>0.1</FibroRate>
    <FieldDim x="71" y="31" z="281"/>
    <ConcentrationFile>SM-71x31x281.dat</ConcentrationFile>
  </Plugin>
  <Plugin Name="Contact">
    <Energy Type1="Medium" Type2="Medium">2.9</Energy>
    <Energy Type1="NonCondensing" Type2="Medium">2.9</Energy>
    <Energy Type1="Condensing" Type2="Condensing">0.5</Energy>
    <Energy Type1="NonCondensing" Type2="NonCondensing">7.0</Energy>
    <Energy Type1="Condensing" Type2="NonCondensing">7.0</Energy>
  </Plugin>
  <Plugin Name="Mitosis">
    <DoublingVolume>16</DoublingVolume>
  </Plugin>
  <Plugin Name="CenterOfMass"/>
  <Plugin Name="Growth">
    <Delta>4</Delta>
    <DensityThreshold>5</DensityThreshold>
    <Delay>10</Delay>
    <FGFThreshold>0.7</FGFThreshold>
    <InitialDim x="71" y="31" z="13"/>
  </Plugin>
  <Steppeable Type="UniformInitializer">
    <Gap>16</Gap>
    <Width>2</Width>
    <RangeBox x="71" y="31" z="13"/>
  </Steppeable>
</CompuCell3D>

Program 19: COMPUCELL3D configuration file for avian limb bud growth, using the BioLOGO-generated LimbChemical and Avian plugins.
D.2 HUVEC Vasculogenesis (Unscripted)

<CompuCell3D>
  <Potts>
    <Dimensions x="500" y="1" z="500"/>
    <Steps>10000</Steps>
    <Temperature>50</Temperature>
    <Flip2DimRatio>1</Flip2DimRatio>
    <Boundary_x>NoFlux</Boundary_x>
    <Boundary_y>NoFlux</Boundary_y>
    <Boundary_z>NoFlux</Boundary_z>
    <FlipNeighborMaxDistance>1.5</FlipNeighborMaxDistance>
  </Potts>
  <Plugin Name="CellType">
    <CellType TypeName="Medium" TypeId="0" />
    <CellType TypeName="Cell" TypeId="1" />
    <CellType TypeName="Boundary" TypeId="2" Freeze="" />
  </Plugin>
  <Plugin Name="SimpleClock"/>
  <Plugin Name="ChemotaxisDicty">
    <Lambda>1000</Lambda>
    <ChemicalField Source="GambaSerini">c</ChemicalField>
  </Plugin>
  <Plugin Name="VolumeFlex">
    <VolumeEnergyParameters CellType="Cell" TargetVolume="100" LambdaVolume="50"/>
    <VolumeEnergyParameters CellType="Boundary" TargetVolume="6760" LambdaVolume="10000"/>
  </Plugin>
  <Plugin Name="CenterOfMass"/>
  <Plugin Name="ContactLocalFlex">
    <Energy Type1="Cell" Type2="Cell">40</Energy>
    <Energy Type1="Cell" Type2="Medium">20</Energy>
    <Energy Type1="Cell" Type2="Boundary">100</Energy>
    <Energy Type1="Boundary" Type2="Medium">0</Energy>
    <Depth>1.5</Depth>
  </Plugin>
  <Plugin Name="LengthConstraint">
    <LengthEnergyParameters CellType="Cell" TargetLength="30" LambdaLength="5"/>
    <LengthEnergyParameters CellType="Boundary" TargetLength="30" LambdaLength="0"/>
  </Plugin>
  <Plugin Name="Connectivity">
    <Penalty>100000</Penalty>
  </Plugin>
  <Steppable Type="PIFInitializer">
    <PIFName>capillary5.pif</PIFName>
  </Steppable>
  <Steppable Type="DictyChemotaxisSteppable">
    <ChemotactUntil>0</ChemotactUntil>
    <IgnoreFirstSteps>0</IgnoreFirstSteps>
    <ChemotaxisActivationThreshold>0.0</ChemotaxisActivationThreshold>
    <ChemicalField Source="GambaSerini">c</ChemicalField>
  </Steppable>
  <Steppable Type="GambaSerini">
    <Step>15</Step>
    <DT>2</DT>
    <DX>0.000002</DX>
    <XLOW>83</XLOW>
    <XHIGH>415</XHIGH>
    <ZLOW>83</ZLOW>
    <ZHIGH>415</ZHIGH>
    <alpha>0.00018</alpha>
    <epsilon>0.000018</epsilon>
    <DiffConst>0.000000000001</DiffConst>
  </Steppable>
</CompuCell3D>

**Program 20**: COMPUCELL3D configuration file for HUVEC vasculogenesis, using the BioLOGO-generated *GambaSerini* PDE solver.
D.3 HUVEC Vasculogenesis (Scripted)

<CompuCell3D>
  <Potts>
    <Dimensions x="500" y="1" z="500"/>
    <Steps>10000</Steps>
    <Temperature>50</Temperature>
    <Flip2DimRatio>1</Flip2DimRatio>
    <Boundary_x>NoFlux</Boundary_x>
    <Boundary_y>NoFlux</Boundary_y>
    <Boundary_z>NoFlux</Boundary_z>
    <FlipNeighborMaxDistance>1.5</FlipNeighborMaxDistance>
  </Potts>
  <Plugin Name="CellType"/>
    <CellType TypeName="Medium" TypeId="0"/>
    <CellType TypeName="Cell" TypeId="1"/>
    <CellType TypeName="Boundary" TypeId="2" Freeze=""/>
  </Plugin>
  <Plugin Name="SimpleClock"/>
    <Plugin Name="ChemotaxisDicty" Lambda="1000"/>
    <ChemicalField Source="GambaSeriniSCRIPT"/>
  </Plugin>
  <Plugin Name="VolumeFlex"/>
    <VolumeEnergyParameters CellType="Cell" TargetVolume="100" LambdaVolume="50"/>
    <VolumeEnergyParameters CellType="Boundary" TargetVolume="6760" LambdaVolume="10000"/>
  </Plugin>
  <Plugin Name="CenterOfMass"/>
    <Plugin Name="ContactLocalFlex" />
    <Energy Type1="Cell" Type2="Cell">40</Energy>
    <Energy Type1="Cell" Type2="Medium">20</Energy>
    <Energy Type1="Cell" Type2="Boundary">100</Energy>
    <Energy Type1="Boundary" Type2="Medium">0</Energy>
    <Depth>1.5</Depth>
  </Plugin>
  <Plugin Name="LengthConstraint"/>
    <LengthEnergyParameters CellType="Cell" TargetLength="30" LambdaLength="5"/>
    <LengthEnergyParameters CellType="Boundary" TargetLength="30" LambdaLength="0"/>
  </Plugin>
  <Plugin Name="Connectivity"/>
    <Penalty>100000</Penalty>
  </Plugin>
  <Steppable Type="PIFInitializer">
    <PIFName>capillary5.pif</PIFName>
  </Steppable>
  <Steppable Type="DictyChemotaxisSteppable">
    <ClockReloadValue>5500</ClockReloadValue>
    <ChemotactUntil>0</ChemotactUntil>
    <IgnoreFirstSteps>0</IgnoreFirstSteps>
    <ChemotaxisActivationThreshold>0.0</ChemotaxisActivationThreshold>
    <ChemicalField Source="GambaSeriniSCRIPT"/>
  </Steppable>
  <Steppable Type="GambaSeriniSCRIPT">
    <Step>15</Step>
    <DT>2</DT>
    <DX>-0.000002</DX>
    <XLOW>83</XLOW>
    <XHIGH>415</XHIGH>
    <ZLOW>83</ZLOW>
    <ZHIGH>415</ZHIGH>
    <alpha>0.00018</alpha>
    <epsilon>0.00018</epsilon>
    <DiffConst>0.000000000001</DiffConst>
  </Steppable>
</CompuCell3D>

Program 21: CompuCell3D configuration file for HUVEC vasculogenesis, using the BioLogo-generated GambaSeriniSCRIPT PDE solver.
APPENDIX E

GENERATED C++/PYTHON FOR VARIOUS EXTENSIONS
#ifndef AVIANNONCONDENSINGTRANSITION_H
#define AVIANNONCONDENSINGTRANSITION_H

#include <CompuCell3D/Automaton/Transition.h>
#include <CompuCell3D/plugins/LimbChemical/LimbChemicalEnergy.h>
#include <CompuCell3D/Simulator.h>
#include <CompuCell3D/plugins/LimbChemical/LimbChemicalPlugin.h>
#include <CompuCell3D/plugins/Avian/AvianPlugin.h>

namespace CompuCell3D {
    
    class Cell;

    class AvianNonCondensingTransition : public Transition {
    
    public:
        
        LimbChemicalEnergy* LimbChemical;

        AvianNonCondensingTransition(char cellType) : Transition(cellType) {
            LimbChemical = ((LimbChemicalPlugin*)
            Simulator::pluginManager.get("LimbChemical")]->getLimbChemicalEnergy();
        }

        bool checkCondition(const Point3D &pt, CellG* newCell) {
            if ((((AvianPlugin*)Simulator::pluginManager.get("Avian")]->getTypeName(
            const_cast<CellG*>(newCell)->type) == "Condensing") &
            (LimbChemical->getActivator()->get(pt)<LimbChemical->getThreshold()))
            return true;
        return false;
        }
    }
};
#endif

Program 22: Implementation of the Transition class for the NonCondensing cell type in the Avian cell type automaton. The name of this file is AvianNonCondensingTransition.h.
Program 23: Implementation of the Transition class for the Condensing cell type in the Avian cell type automaton. The name of this file is AvianCondensingTransition.h.
... Preprocessor directives ... 

AvianPlugin::AvianPlugin() {}  

AvianPlugin::~AvianPlugin() {}  

void AvianPlugin::init(Simulator *simulator) {  
  potts = simulator->getPotts();  
  potts->registerCellGChangeWatcher(this);  
  potts->registerAutomaton(this);  
  classType = new CellType();  
  classType->addTransition(new AvianNonCondensingTransition(1));  
  classType->addTransition(new AvianCondensingTransition(2));  
}  

unsigned char AvianPlugin::getCellType(const CellG *cell) const {  
  if (!cell) return 'M';  
  else if (const_cast<CellG*>(cell)->type == 0)  
    const_cast<CellG*>(cell)->type = 1;  
  return const_cast<CellG*>(cell)->type;  
}  

std::string AvianPlugin::getTypeName(const char type) const {  
  switch (type) {  
    case 'M': return "Medium";  
    case 0: return "NonCondensing";  
    case 1: return "NonCondensing";  
    case 2: return "Condensing";  
    default: THROW(string("Unknown cell type ") + BasicString(type) + ");")  
  }  

unsigned char AvianPlugin::getTypeId(const string typeName) const {  
  if (typeName == "Medium") return 'M';  
  else if (typeName == "NonCondensing") return 1;  
  else if (typeName == "Condensing") return 2;  
  else THROW(string("Unknown cell type ") + typeName + ");")  
}  

void AvianPlugin::creation(CellG* newCell) {  
  if (!newCell) return;  
  else if (getCellType(newCell) == 1) {}  
  else if (getCellType(newCell) == 2) {}  
}  

void AvianPlugin::updateVariables(CellG* newCell) {  
  if (!newCell) return;  
  else if (getCellType(newCell) == 1) {}  
  else if (getCellType(newCell) == 2) {}  
}  

void AvianPlugin::readXML(XMLPullParser &in) {}  

void AvianPlugin::writeXML(XMLSerializer &out) {}  

Program 24: Implementation of the plugin for the Avian cell type automaton.  
The name of this file is AvianPlugin.cpp.
```cpp
#ifndef AVIANPLUGIN_H
#define AVIANPLUGIN_H

#include <CompuCell3D/Plugin.h>

#include <CompuCell3D/Automaton/Automaton.h>
#include "AvianVarDynamicClassNode.h"
#include <string>

namespace CompuCell3D {
    class Potts3D;
    class Cell;

    class AvianPlugin : public Plugin, public Automaton {
        Potts3D* potts;
        AvianVarDynamicClassNode varNode;
    public:
        AvianPlugin();
        virtual ~AvianPlugin();

        virtual void init(Simulator *simulator);
        unsigned char getCellType(const CellG *cell) const;
        std::string getTypeName(const char type) const;
        unsigned char getTypeId(const std::string typeName) const;
        void creation(CellG* newCell);
        void updateVariables(CellG* newCell);
        AvianVarDynamicClassNode getVarNode() {return varNode;}

        virtual void readXML(XMLPullParser &in);
        virtual void writeXML(XMLSerializer &out);
    };
}
#endif
```

Program 25: Header file of the plugin for the *Avian* cell type automaton. The name of this file is *AvianPlugin.h*.

```cpp
#include "AvianPlugin.h"

#include <CompuCell3D/Simulator.h>
using namespace CompuCell3D;

#include <BasicUtils/BasicPluginProxy.h>

BasicPluginProxy<Plugin, AvianPlugin>
avianTypeProxy("Avian", "Adds the Avian cell type variables and
updates cell types.", &Simulator::pluginManager);
```

Program 26: Proxy that registers the *Avian* cell type automaton plugin. The name of this file is *AvianTypePluginProxy.cpp*.
#ifndef AVIANVARDYNAMICCLASSNODE_H
#define AVIANVARDYNAMICCLASSNODE_H

#include <BasicUtils/BasicDynamicClassNode.h>
#include "AvianVarNode.h"

namespace CompuCell3D {
    class AvianVarDynamicClassNode :
        public BasicDynamicClassNode<AvianVarNode> {};
};
#endif

Program 27: Dynamic class node for cell attributes in the Avian cell type automaton. The name of this file is AvianVarDynamicClassNode.h.

#ifndef AVIANVARNODE_H
#define AVIANVARNODE_H

namespace CompuCell3D {
    class AvianVarNode {
        public:
            int targetvolume;
            int targetsurface;
    };
};
#endif

Program 28: Simple class to hold cell attributes. Since Avian did not define any cell state variables, just defaults are declared. The name of this file is AvianVarNode.h.
E.2 Avian Limb: Haptotaxis

```cpp
#include "LimbChemicalPlugin.h"
#include <CompuCell3D/Simulator.h>
using namespace CompuCell3D;
#include <BasicUtils/BasicPluginProxy.h>

BasicPluginProxy<Plugin, LimbChemicalPlugin>
LimbChemicalProxy("LimbChemical", "Adds the LimbChemical energy function.",
                  &Simulator::pluginManager);
```

**Program 29**: Proxy to register the plugin which implements the *LimbChemical* energy term. The name of this file is *LimbChemicalPluginProxy.cpp*. 
Program 30: Generated plugin header for the *LimbChemical* energy term. The name of this file is *LimbChemicalPlugin.h*. 

```cpp
#ifndef LIMBCHEMICALPLUGIN_H
#define LIMBCHEMICALPLUGIN_H

#include "LimbChemicalEnergy.h"
#include <CompuCell3D/Plugin.h>

namespace CompuCell3D {
    class LimbChemicalEnergy;
    class Simulator;
    class Potts3D;

    class LimbChemicalPlugin : public Plugin {
        LimbChemicalEnergy *limbChemicalEnergy;
        Potts3D* potts;

        public:
            LimbChemicalPlugin();
            virtual ~LimbChemicalPlugin();

            virtual void init(Simulator *simulator);
            LimbChemicalEnergy* getLimbChemicalEnergy() {return limbChemicalEnergy;}

            virtual void readXML(XMLPullParser &in);
            virtual void writeXML(XMLSerializer &out);
    };
}
#endif
```
Program 31: Generated plugin implementation for the *LimbChemical* energy term. The name of this file is *LimbChemicalPlugin.cpp*.
class LimbChemicalEnergy: public EnergyFunction, public virtual XMLSerializable {
    Simulator* sim;
    Automaton* automaton;
public:
    float getThreshold () {return Threshold;}  
private:
    float Threshold;
public:
    float getMu () {return Mu;}  
private:
    float Mu;
public:
    float getFibroRate () {return FibroRate;}  
private:
    float FibroRate;
public:
    Field3D<float>** getActivator() {return activator;}  
    std::string getConcentrationFile() {return ConcentrationFile;}  
private:
    Field3D<float>* activator;
    std::string ConcentrationFile;
public:
    Field3D<float>* getFibronectin() {return fibronectin;}  
private:
    Field3D<float>* fibronectin;
public:
    LimbChemicalEnergy() : Threshold(0), Mu(0), FibroRate(0){}
    LimbChemicalEnergy(Simulator* simulator) : sim(simulator), Threshold(0), Mu(0), FibroRate(0){}
    LimbChemicalEnergy(Simulator* simulator, float Threshold, float Mu, float FibroRate) : sim(simulator),
        Threshold(Threshold), Mu(Mu), FibroRate(FibroRate){automaton=sim->getPotts()->getAutomaton();}
    int getIndex(const int type1, const int type2) const;
    virtual double localEnergy(const Point3D &pt);
    virtual double changeEnergy(const Point3D &pt, const CellG *newcell, const CellG *oldcell);
    virtual void readXML(XMLPullParser &in);
    virtual void writeXML(XMLSerializer &out);
};

Program 32: Generated header file LimbChemical energy term calculator. The name of this file is LimbChemicalEnergy.h.
Program 33: Generated implementation of the \textit{LimbChemical} energy term calculator. The name of this file is \texttt{LimbChemicalEnergy.cpp}. 

187
E.3 HUVEC Vasculogenesis: C++ PDE Solver
#ifndef GAMBA SERINI EVOLVER_H
#define GAMBA SERINI EVOLVER_H

... preprocessor directives ...

namespace CompuCell3D {
  class Automaton;
  class GambaSeriniEvolver : public DiffusableVector<float> {
    public:
      GambaSeriniEvolver();
      ~GambaSeriniEvolver();

      virtual void init(Simulator* simulator);
      virtual void start();
      virtual void step(const unsigned int _currentStep);
      virtual void finish() {}
      int Kronecker(int x, int y, int z);

      virtual void readXML(XMLPullParser &in);
      virtual void writeXML(XMLSerializer &out);
    private:
      Simulator* sim;
      Automaton* automaton;
      int __step;
      double __r;
      double __dx;
      int __xlow;
      int __xhigh;
      int __ylow;
      int __yhigh;
      int __zlow;
      int __zhigh;
      float*** __currc;
      float*** __nextc;
      int __myActualStep;
      bool __myFirstBreak;
      string __concentrationFieldName;

    public:
      float getAlpha () {return alpha;}
      float alpha;

    private:
      float getEpsilon () {return epsilon;}
      float epsilon;

    public:
      float getDiffConst () {return DiffConst;}
      float DiffConst;

    getIndex(const int type1, const int type2) const;
  }
};

#endif

Program 34: Generated plugin header for the PDE solver for HUVEC vasculogenesis simulation. The name of this file is GambaSeriniEvolver.h.
```cpp
#include "GambaSeriniEvolver.h"

... preprocessor directives ...
... constructor and destructor ...

void GambaSeriniEvolver::init(Simulator *simulator) {
    sim = simulator;
    __myActualStep = 0;
    allocateDiffusableFieldVector(3, Dim3D(sim->getPotts()->getCellFieldG()->getDim().x+2, sim->getPotts()->getCellFieldG()->getDim().y+2, sim->getPotts()->getCellFieldG()->getDim().z+2));
    if (__xhigh == 0)
        __xhigh = concentrationFieldVector[0]->getDim().x;
    if (__yhigh == 0)
        __yhigh = concentrationFieldVector[0]->getDim().y;
    if (__zhigh == 0)
        __zhigh = concentrationFieldVector[0]->getDim().z;
    __myFirstBreak = true;
}

void GambaSeriniEvolver::start() {
    __currc = (float***)malloc((concentrationFieldVector[0]->getDim().x)*sizeof(float**));
    __nextc = (float***)malloc((concentrationFieldVector[0]->getDim().x)*sizeof(float**));
    for (int x = 0; x < concentrationFieldVector[0]->getDim().x; x++) {
        __currc[x] = (float**)malloc(sizeof(float*));
        __nextc[x] = (float**)malloc(sizeof(float*));
        for (int y = 0; y < 1; y++) {
            __currc[x][y] = (float*)malloc((concentrationFieldVector[0]->getDim().z)*sizeof(float));
            __nextc[x][y] = (float*)malloc((concentrationFieldVector[0]->getDim().z)*sizeof(float));
            for (int z = 0; z < concentrationFieldVector[0]->getDim().z; z++) {
                __currc[x][y][z] = 0; __nextc[x][y][z] = 0;}}
    }
}

int GambaSeriniEvolver::Kronecker(int x, int y, int z) {
    if (sim->getPotts()->getCellFieldG()->get(Point3D(x,y,z))) return 0;
    else return 1;
}

void GambaSeriniEvolver::step(unsigned int _currentStep) {
    for (int iter = 0; iter < __step; iter++) {
        for (int x = __xlow; x < __xhigh; x++) {
            for (int y = 0; y < 1; y++) {
                for (int z = __zlow; z < __zhigh; z++) {
                    __nextc[x][0][z] = __currc[x][0][z];
                    __nextc[x][0][z] = __nextc[x][0][z] + __r*((1-Kronecker(x,y,z))*alpha-epsilon*__currc[x][0][z]
                                             *Kronecker(x,y,z)+DiffConst*((__currc[x][0][z==0?concentrationFieldVector[0]->getDim().z-1:z-1]
                                             +__currc[x][0][(z+1)%(concentrationFieldVector[0]->getDim().z)]
                                             +__currc[(x+1)%(concentrationFieldVector[0]->getDim().x)]
                                             +__currc[x][0][z]<0?4*__currc[x][0][z]/(__dx*__dx));
                    if (__nextc[x][0][z] < 0)
                        __nextc[x][y][z] = 0;}}
                }
            if (__myActualStep % __step) {
                for (int i = __xlow; i < __xhigh; i++) {
                    for (int j = __zlow; j < __zhigh; j++) {
                        __currc[i][0][j] = __nextc[i][0][j];}
                }
                __myActualStep++;
            }
        }
    }
}

... readXML and writeXML ...

Program 35: Generated plugin implementation file for the PDE solver for HUVEC vasculogenesis simulation. The name of this file is GambaSeriniEvolver.cpp.
```

190
Program 36: Proxy for the GambaSerini PDE solver, generated in the file GambaSeriniEvolverProxy.cpp

E.4 HUVEC Vasculogenesis: Python PDE Solver
Program 37: Generated glue code for Python PDE solvers (assumes FiPy is installed).
class GambaSeriniSCRIPTEvolver(SteppablePy):
    def __init__(self, simulator, alpha, epsilon, DiffConst, dt, dx, steps, 
xlo=0, xhi=-1, ylo=0, yhi=-1, zlo=0, zhi=-1, frequency=1):
        SteppablePy.__init__(self, frequency)
        self.simulator = simulator
        self.cellFieldG = self.simulator.getPotts().getCellFieldG()
        self.dim = self.cellFieldG.getDim()
        self.alpha = alpha
        self.epsilon = epsilon
        self.DiffConst = DiffConst
        self.dt = dt
        self.dx = self.dy = self.dz = dx
        self.steps = steps
        self.xlo = xlo
        if (xhi == -1):
            self.xhi = self.dim.x - 1
        else:
            self.xhi = xhi
        self.ylo = ylo
        if (yhi == -1):
            self.yhi = self.dim.y - 1
        else:
            self.yhi = yhi
        self.zlo = zlo
        if (zhi == -1):
            self.zhi = self.dim.z - 1
        else:
            self.zhi = zhi
        self.nx = self.xhi - self.xlo + 1
        self.ny = self.yhi - self.ylo + 1
        self.nz = self.zhi - self.zlo + 1
        self.mesh = Grid3D(dx=self.dx, dy=self.dy, dz=self.dz,
x = self.xhi - self.xlo + 1, y = self.yhi - self.ylo + 1,
z = self.zhi - self.zlo + 1)
        self.kronecker = CellVariable(name='kronecker', mesh=self.mesh, value=0)
        self.c = CellVariable(name='c', mesh=self.mesh, value=0)
    def setCField(self, field):
        self.cField = field
    def start(self): pass

Program 38: Generated Python code which creates bindings for implicitly and explicitly defined inputs, and chemical fields.
Program 39: Generated Python plugin for the GambaSerini PDE solver used in simulations of HUVEC vasculogenesis.
HAMILTONIAN LimbChemical
INPUT Threshold float
INPUT Mu float
INPUT FibroRate float
FIELD fibronectin float
INPUT ConcentrationFile file activator float
IF activator[]>=Threshold
COPY fibronectin[] fibronectin[]+FibroRate
END IF
DECLARE double __energy 0.0
IF oldcell.type=="Condensing"
COPY __energy __energy+(Mu*fibronectin[])
END IF
RETURN __energy
END STEP
END HAMILTONIAN

Program 40: Intermediate code for LimbChemical plugin.
CELLMODEL Avian
USEPLUGIN LimbChemical
CELLTYPE NonCondensing
UPDATECELLTYPES
CHANGEIF Condensing LimbChemical.activator[]<LimbChemical.Threshold
END UPDATECELLTYPES
END CELLTYPE
CELLTYPE Condensing
UPDATECELLTYPES
CHANGEIF NonCondensing LimbChemical.activator[]>LimbChemical.Threshold
END UPDATECELLTYPES
END CELLTYPE
END CELLMODEL

Program 41: Intermediate code for Avian plugin.

EVOLVER GambaSerini false
INPUT alpha float
INPUT epsilon float
INPUT DiffConst float
FIELD c float
DIFFEQ c
TERM alpha*Kronecker-epsilon*(1-Kronecker)*c+DiffConst*Laplacian(c)
true
END DIFFEQ
END EVOLVER

Program 42: Intermediate code for GambaSerini unscripted plugin.

EVOLVER GambaSeriniSCRIPT false
INPUT alpha float
INPUT epsilon float
INPUT DiffConst float
FIELD c float
FIPY
differterm = ImplicitDiffusionTerm(coeff = self.DiffConst)
secretion = self.alpha*self.kronecker
resorption = ImplicitSourceTerm(coeff =
  self.epsilon*(1-self.kronecker))
eq = TransientTerm() == secretion - resorption + differterm
eq.solve(self.c, dt=self.dt)
END FIPY
END EVOLVER

Program 43: Intermediate code for GambaSerini scripted plugin.
G.1 Basic Cell Sorting Type Automaton

**BioLogo** cell type automata allow users to define the following:

1. A set of state variables for each cell, whose collective values represent the *state* of the cell.
2. A method for updating the state variables of each cell, executed at every CPM flip attempt.
3. A method for changing the type of each cell, executed at every CPM flip attempt.

One very basic automaton models the sorting of two cell types, one of which are more adhesive to each other. Previous models for cell sorting [185] demonstrate the collective grouping of cells of the same type into *nodules*, or small cell aggregates. This occurs in nature for example between retinal and epithelial cells of the chicken eye, as shown in Figure G.1.

In **BioLogo**, I model the two cell types as *Light* and *Dark*, and assume a 50/50 chance of cells being initialized to either type. **BioLogo** initializes all cells to the same type. If I make this type *Light*, I will need to attempt one type transition per cell upon the first attempt to flip a pixel belonging to that cell. I can accomplish this by giving all cells one integer state variable, call it *flag*, initialized to 0. Upon the first attempt to flip a pixel belonging to the cell, an attempt is made to change
Figure G.1. Series of images from a cell sorting experiment [184, 186] in the chicken embryo. Epithelial cells are pigmented and dark, and retinal cells are light. The two types of cells are initially randomly distributed (a). Over time the cells cluster into groups by type as a result of differential adhesion, into a tight blob. From [19].

the type to Dark. Afterwards, this flag becomes 1. As long as flag is 1, no attempts can be made to change cell type.

Program 44(a) defines this model in BioLogo. A type automaton is specified using the BioLogo cellmodel tag, which accepts an attribute for the name of the automaton which is also the name of the plugin which is generated for COMPU-CELL3D and referenced in the configuration file using Program 44(b). Cell type automata do not need any other configuration file parameters other than the name.

I then define each possible cell type using celltype tags. By default, the ECM is included in every simulation and does not differentiate, requiring no specification in BioLogo. Each cell type also requires an attribute for its name. For the Light cell type, I define a creation module which initializes flag to 0. This module is invoked once when a cell of the corresponding type is created. The updatevariables module is invoked every time a CPM pixel flip attempt is made on a pixel belonging to that cell. I simply set flag to 1 in the updatevariables module of Light.

Finally, the updatecelltypes module specifies the conditions for a cell to become the corresponding type, using a number of changeif tags, which accept a string attribute currenttype for the current cell type (must be defined in the cell type automaton) and a boolean expression for the condition for changing. This is
invoked before **updatevariables**, thus if a cell is *Light*, an attempt to change to *Dark* will be made before *flag* is set to 1. Note in the condition for changing to *Dark*, mathematical C++ syntax has been scripted, enabling me to access `drand48()` to obtain a random number and give *Light* cells a 50/50 chance of changing to *Dark* if *flag* is still 0.

```xml
<cellmodel name="CellSort">
  <declare>
    <statevar name="flag" />
  </declare>
  <!-- Light cell type. -->
  <celltype name="Light">
    <creation>
      <copy to="flag" from="0" />
    </creation>
    <updatevariables>
      <copy to="flag" from="1" />
    </updatevariables>
  </celltype>
  <!-- Dark cell type (adhesive). -->
  <celltype name="Condensing">
    <updatecelltypes>
      <changeif currenttype="Light"
                condition="((flag equal 0) and 
                          (drand48() greaterequal .5))"/>
    </updatecelltypes>
  </celltype>
</cellmodel>
```

**Program 44:** (a). Cell type automaton for basic cell sorting using two cell types *Light* and *Dark*. (b). Instantiation of this automaton in the CompuCell3D configuration file.

CompuCell3D defines a plugin for adhesion energy, defined in Eq. (2.3). This plugin can be instantiated in the configuration file using Program 45, which defines the $J$ values for each cell type, using **Energy** tags. $J$ is set lowest for *Dark-Dark* interactions, making CPM flips which bring these cell types closer together more
likely. Using this plugin coupled with the BioLogo-generated plugin for the cell type automaton, I was able to use CompuCell3D to model cell sorting, as shown in Figure G.2.

```xml
<Plugin Name="Contact" />
  <Energy Type1="Medium" Type2="Medium">0</Energy>
  <Energy Type1="Light" Type2="Light">14</Energy>
  <Energy Type1="Dark" Type2="Dark">2</Energy>
  <Energy Type1="Light" Type2="Dark">11</Energy>
  <Energy Type1="Light" Type2="Medium">16</Energy>
  <Energy Type1="Dark" Type2="Medium">16</Energy>
</Plugin>
```

Program 45: (a). Instantiation of the CPM contact energy in the CompuCell3D configuration file for cell sorting.

![Image](image.png)

Figure G.2. Results of the cell sorting simulation with CompuCell3D using the BioLogo-generated cell type automaton, starting from a randomly initialized two-cell type aggregate. Overtime the more adhesive Dark cells cluster together in the center of the aggregate, with less adhesive Light cells on the outside.

G.2 Schnakenberg PDEs

The dimensionless Schnakenberg PDE model [144] was used in an earlier 2D simulation of avian limb morphogenesis [34]. Although this model experienced diffi-
culties in three dimensions due to parameter sensitivity and lack of diffusive freedom [91], their model performed well in 2D and is simple enough to provide a good example of coupling chemical fields using BioLogo PDESolvers without embedded Python.

Their model evolved two coupled chemical fields, one for the activator $u$ and the other for an inhibitor $v$:

\[
\frac{\partial u}{\partial t} = \gamma(a - u + u^2v) + \nabla^2 u = \gamma f(u,v) + \nabla^2 u, \\
\frac{\partial v}{\partial t} = \gamma(b - u^2v) + d\nabla^2 u = \gamma g(u,v) + d\nabla^2 v.
\]

Using BioLogo, I can design this PDESolver as shown in Program 46 to take four inputs from the user (implicitly floating point), for $\gamma$, $a$, $b$ and $d$. Since there are now two coupled chemical fields $u$ and $v$, I use two different DiffEq modules, each nesting two Terms. Since the exp argument of the Term tag is a mathematical expression, I could have merged the two terms together within each DiffEq, but chose to keep them separate for clarity. Note that any defined input and field names become bound to variables usable in mathematical expressions within the DiffEq blocks.
<PDESolver name="Schnakenberg">
  
  <Input name="gamma" />
  <Input name="a" />
  <Input name="b" />
  <Input name="d" />

  <Field name="u" />
  <Field name="v" />

  <DiffEq fieldname="u">
    <Term exp="gamma*(a-u+u*u*v)" />
    <Term exp="laplacian(u)*u" />
  </DiffEq>

  <DiffEq fieldname="v">
    <Term exp="gamma*(b-u*u*v)" />
    <Term exp="d*laplacian(v)" />
  </DiffEq>
</PDESolver>

Program 46: Schnakenberg equations in BioLOGO.
BIOLOGICAL TERMS: MOLECULAR DYNAMICS

- **amino acid** - subunit of a protein consisting of an amino group (NH$_2$), a uniquely identifying residue, alpha carbon, and carboxyl (COOH) group.
- **angle** - a group of three covalently bonded atoms.
- **atom** - the smallest building block of matter.
- **covalent bond** - a bond between two atoms where electrons are shared.
- **dehydration synthesis** - A chemical reaction resulting in the joining of two molecules by removing a hydrogen (H) from one and a hydroxyl (OH) from the other, producing water.
- **dihedral** - a group of four covalently bonded atoms, in sequence.
- **dipeptide** - molecule resulting from the combination of two amino acids by dehydration synthesis.
- **electrostatic** - a force between charged particles which is attractive between opposite charges and repulsive otherwise; increasing in magnitude at a small distance.
- **Hamiltonian** - an expression for system energy which is always conserved (varies across ensembles).
- **improper** - a group of four covalently bonded atoms, where one of the four atoms is covalently bonded to the other three.
- **ion** - an atom which has lost (+) or gained (-) electrons.
- **ionic bond** - a highly attractive force between a positively and negatively charged ion.
- **molecule** - a collection of two or more atoms, bounded together.
- **Nuclear Magnetic Resonance (NMR)** - the application of an orthogonal magnetic and electric field to a molecule and determining structure based on response along with known atomic magnetic moments and angular momenta.
• **pairwise force** - a force which occurs between a pair of atoms.

• **polypeptide** - two or more amino acids bounded together.

• **phase space** - the set of atomic positions and momenta, completely defining a molecular system.

• **propagator** - a mathematical method for moving a system from one phase space to a new one, by solving equations of motion.

• **van der Waals** - a type of nonbonded, intermolecular weak attractive force. These can include dipole-dipole attraction, London dispersion, and hydrogen bonding.

• **X-ray Crystallography** - a technique for deducing atomic positions by passing X rays through a crystalline solid and observing resulting diffraction patterns.
APPENDIX I

MDL API DESCRIPTION
I.1 Package src

I.1.1 Modules

- **factories** (*Section I.2, p. 207*)
  - ForceFactory (*Section I.3, p. 208*)
  - PropagatorFactory (*Section I.4, p. 215*)

- **forces** (*Section I.5, p. 220*)
  - HDForce (*Section I.6, p. 221*)

- **modifiers** (*Section I.7, p. 222*)
  - averagePositions (*Section I.8, p. 223*)
  - friction (*Section I.9, p. 225*)
  - mollify (*Section I.10, p. 226*)

- **propagators** (*Section I.11, p. 227*)
  - methods (*Section I.12, p. 228*)
    * bbk (*Section I.13, p. 229*)
    * impulse (*Section I.14, p. 230*)
    * leapfrog (*Section I.15, p. 231*)
    * takahashi (*Section I.16, p. 232*)
    * velocitiescale (*Section I.17, p. 233*)
  - objects (*Section I.18, p. 234*)
    * HMCMDL (*Section I.19, p. 235*)
    * MTS (*Section I.20, p. 237*)
    * NosePoincGL (*Section I.21, p. 238*)
    * RMT (*Section I.22, p. 242*)
    * STS (*Section I.23, p. 246*)

- **toplevel** (*Section I.24, p. 247*)
  - Constants (*Section I.25, p. 248*)
  - Energies (*Section I.26, p. 251*)
  - FTSM (*Section I.27, p. 256*)
  - ForceField (*Section I.28, p. 263*)
  - Forces (*Section I.29, p. 266*)
  - IO (*Section I.30, p. 268*)
  - Physical (*Section I.31, p. 281*)
  - Propagator (*Section I.32, p. 290*)
I.2 Package src.factories

I.2.1 Modules

- ForceFactory (Section I.3, p. 208)
- PropagatorFactory (Section I.4, p. 215)
I.3 Module src.factories.ForceFactory

I.3.1 Class ForceFactory

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(self)</td>
<td>Initializes mappings from boundary conditions (for bonded forces) and boundary conditions, algorithms and switching functions (for nonbonded forces) to SWIG-wrapped force object constructors (not instances, saving memory).</td>
</tr>
<tr>
<td>createBondForce(self, bc)</td>
<td>Return a bond force object.</td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>bc</td>
<td>Boundary conditions (Periodic or Vacuum)</td>
</tr>
<tr>
<td>(type=string)</td>
<td></td>
</tr>
<tr>
<td>Return Value</td>
<td>SWIG-wrapped bond force object.</td>
</tr>
<tr>
<td>(type=Force)</td>
<td></td>
</tr>
<tr>
<td>createAngleForce(self, bc)</td>
<td>Return an angle force object.</td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>bc</td>
<td>Boundary conditions (Periodic or Vacuum)</td>
</tr>
<tr>
<td>(type=string)</td>
<td></td>
</tr>
<tr>
<td>Return Value</td>
<td>SWIG-wrapped angle force object.</td>
</tr>
<tr>
<td>(type=Force)</td>
<td></td>
</tr>
</tbody>
</table>
createDihedralForce(self, bc)
Return a dihedral force object.

Parameters

bc: Boundary conditions (Periodic or Vacuum)
(type=string)

Return Value

SWIG-wrapped dihedral force object.
(type=Force)

createImproperForce(self, bc)
Return an improper force object.

Parameters

bc: Boundary conditions (Periodic or Vacuum)
(type=string)

Return Value

SWIG-wrapped improper force object.
(type=Force)

createHarmDihedralForce(self, bc, params)
Return a harmonic dihedral force object.

Parameters

bc: Boundary conditions (Periodic or Vacuum)
(type=string)

params: Mapping from parameter names to corresponding values.
(type=dict)

Return Value

SWIG-wrapped harmonic dihedral force object.
(type=Force)
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Parameters</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createMollyBondForce(self, bc)</code></td>
<td>Return a mollified bond force object.</td>
<td><code>bc</code>: Boundary conditions (Periodic or Vacuum)</td>
<td>SWIG-wrapped mollified bond force object. (&lt;type&gt;Force)</td>
</tr>
<tr>
<td><code>createMollyAngleForce(self, bc)</code></td>
<td>Return a mollified angle force object.</td>
<td><code>bc</code>: Boundary conditions (Periodic or Vacuum)</td>
<td>SWIG-wrapped mollified angle force object. (&lt;type&gt;Force)</td>
</tr>
<tr>
<td><code>createLennardJonesForce(self, bc, params)</code></td>
<td>Return a van der Waals force object.</td>
<td><code>bc</code>: Boundary conditions (Periodic or Vacuum)</td>
<td>SWIG-wrapped van der Waals force object. (&lt;type&gt;Force)</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>params</code>: Mapping from parameter names to corresponding values.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>createCoulombDiElecForce(self, bc, params)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return an coulomb dielectric force object (for implicit solvation).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bc: Boundary conditions (Periodic or Vacuum)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(type=string)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>params: Mapping from parameter names to corresponding values.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(type=dict)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return Value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWIG-wrapped coulomb dielectric force object.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(type=Force)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>createCoulombForce(self, bc, params, fastelectro)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return an electrostatic force object.</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>bc: Boundary conditions (Periodic or Vacuum)</td>
</tr>
<tr>
<td>(type=string)</td>
</tr>
<tr>
<td>params: Mapping from parameter names to corresponding values.</td>
</tr>
<tr>
<td>(type=dict)</td>
</tr>
<tr>
<td>Return Value</td>
</tr>
<tr>
<td>SWIG-wrapped electrostatic force object.</td>
</tr>
<tr>
<td>(type=Force)</td>
</tr>
<tr>
<td>Function Name</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>createLennardJonesCoulombForce(self, bc, params)</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>bc</td>
</tr>
<tr>
<td>params</td>
</tr>
<tr>
<td>Return Value</td>
</tr>
<tr>
<td>createMagneticDipoleForce(self, bc)</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>bc</td>
</tr>
<tr>
<td>Return Value</td>
</tr>
<tr>
<td>switchingFunctions(self, params, number)</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>params</td>
</tr>
<tr>
<td>number</td>
</tr>
</tbody>
</table>
**getParameter** *(self, params, name, defaultval=0)*

Return a parameter value if it exists, otherwise return the passed default value.

**Parameters**

- **params**: Mapping from parameter names to values  
  *(type=dict)*
- **name**: Name of the parameter  
  *(type=string)*
- **defaultval**: Default value for the parameter if it is not found  
  *(type=(any type))*

**applyParameters** *(self, newforce, bc, alg, switch, params, fastelectro=None)*

Depending on the algorithm used, set parameter values for the passed force object.

**Parameters**

- **newforce**: Pairwise force object  
  *(type=Force)*
- **bc**: Boundary conditions (Periodic or Vacuum)  
  *(type=string)*
- **alg**: Algorithm for pairwise evaluation  
  *(type=string)*
- **switch**: Switching function  
  *(type=string)*
- **params**: Mapping from parameter names to values  
  *(type=dict)*
- **fastelectro**: Mapping from fast electrostatic parameter names to values  
  *(type=dict)*

**Return Value**

Pairwise force object with instantiated parameters.  
*(type=Force)*

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bondForces</td>
<td>Mapping from boundary conditions to two-atom bond force object constructors</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>angleForces</td>
<td>Mapping from boundary conditions to three-atom angle force object constructors</td>
</tr>
<tr>
<td>dihedralForces</td>
<td>Mapping from boundary conditions to four-atom dihedral force object constructors</td>
</tr>
<tr>
<td>improperForces</td>
<td>Mapping from boundary conditions to four-atom improper force object constructors</td>
</tr>
<tr>
<td>harmDihedralForces</td>
<td>Mapping from boundary conditions to harmonic dihedral force object constructors</td>
</tr>
<tr>
<td>hd</td>
<td>Number of harmonic dihedral forces</td>
</tr>
<tr>
<td>ljForces</td>
<td>Maps boundary conditions, algorithm and switching function to van der Waals force object constructor</td>
</tr>
<tr>
<td>cdeForces</td>
<td>Maps boundary conditions, algorithm and switching function to coulomb dielectric force object constructor - used for implicit solvation</td>
</tr>
<tr>
<td>coulombForces</td>
<td>Maps boundary conditions, algorithm and switching function to electrostatic force object constructor. For fast electrostatics, an additional mapping is performed for the terms calculated (i.e. for Ewald, real reciprocal and correction).</td>
</tr>
<tr>
<td>ljCoulombForces</td>
<td>Maps boundary conditions, algorithm and switching function pair to a unified van der Waals and electrostatic force object constructor. This saves performance by determining atom pairs just once for both types of pairwise forces.</td>
</tr>
<tr>
<td>magneticDipoleForces</td>
<td>Mapping from boundary conditions to magnetic dipole force object constructors</td>
</tr>
<tr>
<td>mollify</td>
<td>Wrapper function for MOLLY forces. Wraps either a bond or angle force object.</td>
</tr>
</tbody>
</table>
I.4 Module src.factories.PropagatorFactory

I.4.1 Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setPropagator(prop, phys, forces, obj, levelswitch=False)</code></td>
<td>Set and initialize a propagator object.</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td></td>
</tr>
<tr>
<td>prop:</td>
<td>MDL Propagator object</td>
</tr>
<tr>
<td>phys:</td>
<td>The physical system.</td>
</tr>
<tr>
<td>forces:</td>
<td>MDL Forces object</td>
</tr>
<tr>
<td>obj:</td>
<td>Prototyped propagator object</td>
</tr>
<tr>
<td>levelswitch:</td>
<td>True if we are changing levels in the hierarchy.</td>
</tr>
<tr>
<td></td>
<td>Default false.</td>
</tr>
<tr>
<td><code>executePropagator(prop, phys, forces, io, numsteps)</code></td>
<td>Run and finish the propagator.</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td></td>
</tr>
<tr>
<td>prop:</td>
<td>MDL Propagator object</td>
</tr>
<tr>
<td>phys:</td>
<td>The physical system.</td>
</tr>
<tr>
<td>forces:</td>
<td>MDL Forces object</td>
</tr>
<tr>
<td>numsteps:</td>
<td>Number of steps to run</td>
</tr>
</tbody>
</table>

I.4.2 Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>propFactory</td>
<td>PropagatorFactory singleton object</td>
</tr>
<tr>
<td>Value:</td>
<td>PropagatorFactory()</td>
</tr>
</tbody>
</table>

I.4.3 Class PropagatorFactory

Contains mappings from propagator names to creation functions, their types (object or method) and parameters with default values. Given a name and parameter values, the propagator factory can return either a method handle or an instance of a propagator.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__(self)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>query(self, name=&quot;None&quot;)</code></td>
<td>Query the factory for information on all available propagation schemes. This will display their type (object or method) and parameters and default values. You can also query with a name to get information about a specific propagation scheme.</td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>name: Propagation scheme name, default is None (type=string)</td>
<td></td>
</tr>
<tr>
<td><code>registerObject(self, myModule, name, defaults)</code></td>
<td>Register a propagator object prototyped in Python</td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>myModule: Module in which the class definition is contained (type=Python Module) name: Name of propagation scheme (type=string) defaults: Mapping from parameter names to default values (type=tuple)</td>
<td></td>
</tr>
<tr>
<td><code>registerMethod(self, myModule, name, defaults)</code></td>
<td>Register a propagator Python method.</td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>myModule: Module in which the method definition is contained (type=Python Module) name: Name of propagation scheme (type=string) defaults: Mapping from parameter names to default values (type=tuple)</td>
<td></td>
</tr>
</tbody>
</table>
registerPMObject(self, name, defaults)
Register a SWIG-wrapped propagator object

Parameters

name: Name of propagation scheme
    (type=string)
defaults: Mapping from parameter names to default values
    (type=tuple)

registerModifier(self, name, thetype, theprop)
Register a propagator modifier

Parameters

name: Name of propagation scheme
    (type=string)
thesize: Type of modifier (preinit, postinit, preforce, ...)
    (type=string)
thesizep: Name of the propagator object that this modifier
    operates upon.
    (type=string)

getType(self, name)
Return the type (object or method) of a propagator

Parameters

name: Name of propagation scheme
    (type=string)

Return Value

Type of the propagator
    (type=string)
**findArg**(*self, name, pars*)

Find a parameter name in a parameter tuple.

**Parameters**

- **name**: Name of parameter
  
  *(type=string)*

- **pars**: Parameter list
  
  *(type=tuple)*

**Return Value**

Index of the parameter in the list (-1 if not found)

*(type=int)*

**applyModifiers**(*self, obj, name*)

Instantiate modifiers for a propagator.

**Parameters**

- **obj**: Propagator object
  
  *(type=STS/MTS)*

- **name**: Propagator name
  
  *(type=string)*

**Return Value**

Propagator object with instantiated modifiers.

*(type=STS/MTS)*
create(self, *args)

Accept a Python tuple containing the propagator name, timestep, parameter values and force field. This tuple can thus be various sizes depending on the number of parameters. Create and return a corresponding instantiated propagator object, or method handle.

Parameters

args: List of propagator name, dt, parameter values and force field. If the propagation scheme is MTS, this list is followed by the name of the next propagator in the chain and its associated parameters, etc.

(type=tuple)

Return Value

The propagator.

(type=STS, MTS, or Python method handle)

registerAllPM(self)

Register all SWIG-wrapped propagators.

Instance Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objects</td>
<td>A list of propagator objects created.</td>
</tr>
<tr>
<td>registry</td>
<td>Maps propagator names to creation functions and parameters/defaults</td>
</tr>
<tr>
<td>modifier_registry</td>
<td>Maps modifier names to method handles</td>
</tr>
</tbody>
</table>
I.5 Package src.forces

I.5.1 Modules

- **HDForce** *(Section I.6, p. 221)*
I.6 Module src.forces.HDForce

I.6.1 Functions

<table>
<thead>
<tr>
<th>norm(v3d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>norm2(v3d)</td>
</tr>
</tbody>
</table>

I.6.2 Class HDForce

Implement a harmonic dihedral constraining potential: \( U(x) = k(\phi - \phi_0)^2 \)

Methods

```python
__init__(self, phys, forces, phi, num, k)
```

Initialize an object of type HDForce

**Parameters**

| phys: | The physical system. | (type=Physical) |
| phys: | MDL Forces object  | (type=Physical) |
| phi: | Target dihedral value in radians. | (type=float) |
| dihedral: | Dihedral number to constrain. | (type=integer) |
| k: | | (type=float) |

```python
evaluate(self)
```

Modify energy and force vector to include this force term.
I.7 Package src.modifiers

I.7.1 Modules

- **averagePositions** *(Section I.8, p. 223)*
- **friction** *(Section I.9, p. 225)*
- **mollify** *(Section I.10, p. 226)*
I.8  Module src.modifiers.averagePositions

I.8.1  Functions

averagePositions(phys, forces, prop, obj)

Perform position averaging for a MOLLY propagator. This was used in conjunction with an MDL-prototyped Bspline MOLLY propagator and assumes some member functions of the passed object.

Parameters

- **phys**: The physical system. 
  *(type=Physical)*
- **forces**: MDL Forces object 
  *(type=Forces)*
- **prop**: MDL Propagator object 
  *(type=Propagator)*
- **obj**: Prototyped propagator object 
  *(type=MOLLY)*

I.8.2  Class ReducedHessAngleList

object

```plaintext
list

src.modifiers.averagePositions.ReducedHessAngleList
```

 Inherits from the Python list, and stores angle Hessian matrices.

Methods

_init_(self, size)

Initialize a list of the passed size with default matrices.

Parameters

- **size**: Size of the list. 
  *(type=int)*

Return Value

new list

Overrides: list.__init__

identityAll(self)

Set all objects to the identity matrix.
__mul__(self, factor)
Scale all Hessians.

Parameters

factor: Scaling factor.
(type=int)

Return Value

Scaled Hessians.
(type=ReducedHessAngleList)

Overrides: list.__mul__

__add__(self, list2)
Add two Hessian lists.

Parameters

list2: Second operand.
(type=ReducedHessAngleList)

Return Value

Hessian sum.
(type=ReducedHessAngleList)

Overrides: list.__add__

Inherited from list

__contains__(), __delitem__(), __delslice__(), __eq__(), __ge__(), __getattr__(),
__getitem__(), __getslice__(), __gt__(), __hash__(), __iadd__(), __imul__(), __iter__(),
__le__(), __len__(), __lt__(), __ne__(), __new__(), __repr__(), __reversed__(), __rmul__(),
__setitem__(), __setslice__(), append(), count(), extend(), index(), insert(), pop(),
remove(), reverse(), sort()

Inherited from object

__delattr__(), __reduce__(), __reduce_ex__(), __setattr__(), __str__()

Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| Inherited from object
| __class__ |             |
I.9  Module src.modifiers.friction

I.9.1  Functions

<table>
<thead>
<tr>
<th>friction(phys, forces, prop, obj)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modify the force vector to include a frictional force. This can be used for incorporation of Langevin dynamics.</td>
</tr>
</tbody>
</table>

**Parameters**

- **phys**: The physical system.  
  *(type=Physical)*
- **forces**: MDL Forces object  
  *(type=Forces)*
- **prop**: MDL Propagator object  
  *(type=Propagator)*
- **obj**: Prototyped propagator object  
  *(type=STS/MTS)*
I.10 Module src.modifiers.mollify

I.10.1 Functions

<table>
<thead>
<tr>
<th>mollify(phys, forces, prop, obj)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perform mollification. This uses angle filters and thus can only modify a MOLLY object.</td>
</tr>
</tbody>
</table>

**Parameters**

- **phys**: The physical system.
  
  *(type=Physical)*

- **forces**: MDL Forces object
  
  *(type=Forces)*

- **prop**: MDL Propagator object
  
  *(type=Propagator)*

- **obj**: Prototyped propagator object
  
  *(type=MOLLY)*
I.11 Package src.propagators

I.11.1 Modules

- **methods** *(Section I.12, p. 228)*
  - bbk *(Section I.13, p. 229)*
  - impulse *(Section I.14, p. 230)*
  - leapfrog *(Section I.15, p. 231)*
  - takahashi *(Section I.16, p. 232)*
  - velocityscale *(Section I.17, p. 233)*

- **objects** *(Section I.18, p. 234)*
  - HMCMDL *(Section I.19, p. 235)*
  - MTS *(Section I.20, p. 237)*
  - NosePoincGL *(Section I.21, p. 238)*
  - RMT *(Section I.22, p. 242)*
  - STS *(Section I.23, p. 246)*
I.12 Package src.propagators.methods

I.12.1 Modules

- **bbk** *(Section I.13, p. 229)*
- **impulse** *(Section I.14, p. 230)*
- **leapfrog** *(Section I.15, p. 231)*
- **takahashi** *(Section I.16, p. 232)*
- **velocitiescale** *(Section I.17, p. 233)*
I.13 Module src.propagators.methods.bbk

I.13.1 Functions

```
bbk(phys, forces, io, steps, timestep, fg, temp, gamma, seed)
```


**Parameters**

- **phys**: The physical system. 
  \( \text{type}=\text{Physical} \)
- **forces**: MDL Forces object. 
  \( \text{type}=\text{Forces} \)
- **io**: MDL IO object. 
  \( \text{type}=\text{IO} \)
- **steps**: Number of steps to run. 
  \( \text{type}=\text{int} \)
- **timestep**: Timestep for propagation. 
  \( \text{type}=\text{float} \)
- **fg**: MDL force field for evaluation. 
  \( \text{type}=\text{ForceField} \)
- **temp**: Kelvin temperature. 
  \( \text{type}=\text{float} \)
- **gamma**: Friction constant in Langevin dynamics. 
  \( \text{type}=\text{float} \)
- **seed**: Random number seed 
  \( \text{type}=\text{int} \)

I.13.2 Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Propagator name for the factory</td>
<td>&quot;bbk&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Value: &quot;temp&quot;, 300, &quot;gamma&quot;, 0.5, &quot;seed&quot;, 1234</td>
</tr>
</tbody>
</table>
Module src.propagators.methods.impulse

I.14.1 Functions

\textbf{impulse}(\texttt{phys}, \texttt{forces}, \texttt{io}, \texttt{steps}, \texttt{timestep}, \texttt{fg}, \texttt{nextinteg}, *\texttt{args})

Verlet/r-RESPA propagation method. Multiple timestepping.

\textbf{Parameters}

\begin{itemize}
\item \textbf{phys}: The physical system.
  \textit{(type=Physical)}
\item \textbf{forces}: MDL Forces object.
  \textit{(type=Forces)}
\item \textbf{io}: MDL IO object.
  \textit{(type=IO)}
\item \textbf{steps}: Number of steps to run.
  \textit{(type=int)}
\item \textbf{timestep}: Timestep for propagation.
  \textit{(type=float)}
\item \textbf{fg}: MDL force field for evaluation.
  \textit{(type=ForceField)}
\item \textbf{nextinteg}: Method handle for next propagator in the chain
  \textit{(type=function)}
\item \textbf{args}: Parameters for the next propagator in the chain
  \textit{(type=tuple)}
\end{itemize}

I.14.2 Variables

\begin{tabular}{|l|l|}
\hline
\textbf{Name} & \textbf{Description} \\
\hline
name & Propagator name for the factory \\
& \textit{Value: "impulse"} \\
parameters & Parameter names and defaults \\
& \textit{Value: ()} \\
\hline
\end{tabular}
I.15  Module src.propagators.methods.leapfrog

I.15.1  Functions

```
leapfrog(phys, forces, io, steps, timestep, fg)
```

Leapfrog propagation method. Single timestepping.

**Parameters**

- **phys**: The physical system.
  
  (type=Physical)

- **forces**: MDL Forces object.
  
  (type=Forces)

- **io**: MDL IO object.
  
  (type=IO)

- **steps**: Number of steps to run.
  
  (type=int)

- **timestep**: Timestep for propagation.
  
  (type=float)

- **fg**: MDL force field for evaluation.
  
  (type=ForceField)

I.15.2  Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Propagator name for the factory</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong>: &quot;leapfrog&quot;</td>
</tr>
<tr>
<td>parameters</td>
<td>Parameters and defaults</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong>: ()</td>
</tr>
</tbody>
</table>
I.16 Module src.propagators.methods.takahashi

I.16.1 Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>takahashi(phys, forces, io, steps, timestep, fg)</code></td>
<td>Simplified Takahashi-Imada propagation method. Single timestepping.</td>
</tr>
</tbody>
</table>

Parameters

- **phys**: The physical system.  
  *(type=Physical)*
- **forces**: MDL Forces object.  
  *(type=Forces)*
- **io**: MDL IO object.  
  *(type=IO)*
- **steps**: Number of steps to run.  
  *(type=int)*
- **timestep**: Timestep for propagation.  
  *(type=float)*
- **fg**: MDL force field for evaluation.  
  *(type=ForceField)*

I.16.2 Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Propagator name for the factory</td>
</tr>
<tr>
<td></td>
<td>Value: &quot;takahashi&quot;</td>
</tr>
<tr>
<td>parameters</td>
<td>Parameter names and defaults</td>
</tr>
<tr>
<td></td>
<td>Value: ()</td>
</tr>
</tbody>
</table>
I.17 Module src.propagators.methods.velocityscale

I.17.1 Functions

**velocityscale**(phys, forces, io, steps, timestep, fg, t0)

Leapfrog propagation method. Single timestepping.

**Parameters**

- **phys**: The physical system.
  
  *(type=Physical)*

- **forces**: MDL Forces object.
  
  *(type=Forces)*

- **io**: MDL IO object.
  
  *(type=IO)*

- **steps**: Number of steps to run.
  
  *(type=int)*

- **timestep**: Timestep for propagation.
  
  *(type=float)*

- **fg**: MDL force field for evaluation.
  
  *(type=ForceField)*

I.17.2 Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Propagator name for the factory</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong>: &quot;velocityscale&quot;</td>
</tr>
<tr>
<td>parameters</td>
<td>Parameters and defaults</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong>: 'T0', 300</td>
</tr>
</tbody>
</table>
I.18  Package src.propagators.objects

I.18.1  Modules

- HMCMDL (Section I.19, p. 235)
- MTS (Section I.20, p. 237)
- NosePoincGL (Section I.21, p. 238)
- RMT (Section I.22, p. 242)
- STS (Section I.23, p. 246)
I.19 Module src.propagators.objects.HMCMDL

I.19.1 Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of propagation scheme</td>
</tr>
<tr>
<td></td>
<td>Value: &quot;HMCMDL&quot;</td>
</tr>
<tr>
<td>parameters</td>
<td>Tuple of parameters</td>
</tr>
<tr>
<td></td>
<td>Value: ()</td>
</tr>
</tbody>
</table>

I.19.2 Class HMCMDL

src.propagators.objects.MTS.MTS → src.propagators.objects.HMCMDL.HMCMDL

Implements Hybrid Monte Carlo sampling.

Methods

```python
metropolis(self, new, curr, phys)
```

Metropolis function which computes an acceptance probability for sets of positions depending on energy change $dE$. $P = e^{-dE/kt}$

**Parameters**

- `new`: Hypothetical new energy with flip. 
  
  *(type= float)*

- `curr`: Current energy.
  
  *(type= float)*

- `phys`: The physical system.
  
  *(type= Physical)*

**Return Value**

- 0 for reject, 1 for accept
  
  *(type= int)*
### `init(self, phys, forces, prop)`

Initialize propagator: seed the generator, invoke the next propagator in the chain, and compute forces.

**Parameters**

- **phys:** The physical system.  
  *(type=Physical)*
- **forces:** MDL Forces object.  
  *(type=Forces)*
- **prop:** MDL Propagator object.  
  *(type=Propagator)*

### `run(self, phys, forces, prop)`

Run propagator.

**Parameters**

- **phys:** The physical system.  
  *(type=Physical)*
- **forces:** MDL Forces object.  
  *(type=Forces)*
- **prop:** MDL Propagator object.  
  *(type=Propagator)*

### `finish(self, phys, forces, prop)`

Finish propagator; in this case just invoke the finish method of the next propagator in the chain.

**Parameters**

- **phys:** The physical system.  
  *(type=Physical)*
- **forces:** MDL Forces object.  
  *(type=Forces)*
- **prop:** MDL Propagator object.  
  *(type=Propagator)*
I.20  Module src.propagators.objects.MTS

I.20.1  Class MTS

MTSIntegrator.MTSIntegrator

src.propagators.objects.MTS.MTS

Parent class for all multiple time-stepping propagators.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init(self)</td>
<td>Initialize the propagator.</td>
</tr>
<tr>
<td>run(self)</td>
<td>Run the propagator.</td>
</tr>
<tr>
<td>finish(self)</td>
<td>Finalize the propagator.</td>
</tr>
</tbody>
</table>

I.20.2  Class MOLLY

MTSIntegrator.MTSIntegrator

src.propagators.objects.MTS.MTS

src.propagators.objects.MTS.MOLLY

Parent class for all MOLLY propagators.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setMOLLYForceGroups(self)</td>
<td>Initialize MOLLY force fields and energies.</td>
</tr>
<tr>
<td>calculateBondedForces(self)</td>
<td>Evaluate mollified bonded forces.</td>
</tr>
</tbody>
</table>

Inherited from src.propagators.objects.MTS.MTS(Section I.20.1)

finish(), init(), run()
I.21 Module src.propagators.objects.NosePoincGL

I.21.1 Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of propagation scheme.</td>
</tr>
<tr>
<td></td>
<td>Value: &quot;NosePoincGL&quot;</td>
</tr>
<tr>
<td>parameters</td>
<td>Parameters and defaults</td>
</tr>
<tr>
<td></td>
<td>Value: 'temp', 500, 'bathM', 0.0, 'bathP', 1.0, 'Q', 10.0</td>
</tr>
</tbody>
</table>

I.21.2 Class NosePoincGL

src.propagators.objects.STS.STS

src.propagators.objects.NosePoincGL.NosePoincGL

This method is based on an extended Hamiltonian with an additional thermostat, or degree of freedom. The resulting Hamiltonian gives rise to implicit coupling between the variables, requiring implicit symplectic methods. Since this is undesirable the Generalised Leapfrog method is utilised to produce a system which can be solved explicitly. This method is described in detail in:


Methods

\[
\text{init}(self, \text{phys}, \text{forces}, \text{prop})
\]

Initialize propagator.

**Parameters**

<table>
<thead>
<tr>
<th>phys: The physical system.</th>
</tr>
</thead>
<tbody>
<tr>
<td>{type}=Physical</td>
</tr>
<tr>
<td>forces: MDL Forces object.</td>
</tr>
<tr>
<td>{type}=Forces</td>
</tr>
<tr>
<td>prop: MDL Propagator object.</td>
</tr>
<tr>
<td>{type}=Propagator</td>
</tr>
</tbody>
</table>
half1UpdtMom(self, phys, forces, prop)

Update system velocity first 1/2 step

Parameters

phys: The physical system.
  (type=Physical)

forces: MDL Forces object.
  (type=Forces)

prop: MDL Propagator object.
  (type=Propagator)

half1UpdtbathM(self, phys, forces, prop)

Update thermal bath 'momenta' first 1/2 step

Parameters

phys: The physical system.
  (type=Physical)

forces: MDL Forces object.
  (type=Forces)

prop: MDL Propagator object.
  (type=Propagator)

UpdtPosBathP(self, phys, prop)

Update positions and thermal bath variable full step

Parameters

phys: The physical system.
  (type=Physical)

prop: MDL Propagator object.
  (type=Propagator)

half2UpdtbathM(self, phys, forces, prop)

Update thermal bath 'momenta' first 1/2 step

Parameters

phys: The physical system.
  (type=Physical)

forces: MDL Forces object.
  (type=Forces)

prop: MDL Propagator object.
  (type=Propagator)
**half2UpdtMom**(*self, phys, forces, prop*)

Update system velocity second 1/2 step

**Parameters**

- **phys**: The physical system.
  
  *(type=*Physical*)

- **forces**: MDL Forces object.
  
  *(type=*Forces*)

- **prop**: MDL Propagator object.
  
  *(type=*Propagator*)

**run**(*self, phys, forces, prop*)

Run propagator. 1/2 step in momentum followed by 1/2 step in thermostat momentum followed by full step in positions, then 1/2 step in thermostat momentum and 1/2 step in momentum to give a time reversible method.

**Parameters**

- **phys**: The physical system.
  
  *(type=*Physical*)

- **forces**: MDL Forces object.
  
  *(type=*Forces*)

- **prop**: MDL Propagator object.
  
  *(type=*Propagator*)

**finish**(*self, phys, forces, prop*)

Finalize propagator. Append data to temperature and total energy Python lists.

**Parameters**

- **phys**: The physical system.
  
  *(type=*Physical*)

- **forces**: MDL Forces object.
  
  *(type=*Forces*)

- **prop**: MDL Propagator object.
  
  *(type=*Propagator*)

**Instance Variables**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bathPold</td>
<td>Old thermostat value as system stores rescaled velocity number of Dof of system, momentum conserved so atoms*3D-3</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gkT</td>
<td>Product of degrees of freedom, boltzmann constant and Kelvin temperature</td>
</tr>
<tr>
<td>KEtoT</td>
<td>Kinetic to temp. conversion</td>
</tr>
<tr>
<td>Potnl</td>
<td>Potential energy</td>
</tr>
<tr>
<td>h0</td>
<td>Initial ’internal’ Hamiltonian value so that total Hamiltonian always 0</td>
</tr>
<tr>
<td>stepsdone</td>
<td>Number of steps completed</td>
</tr>
<tr>
<td>avTemp</td>
<td>Average Kelvin temperature</td>
</tr>
<tr>
<td>tempers</td>
<td>Holds pairs of step numbers and average temperatures</td>
</tr>
<tr>
<td>Hamiltonian</td>
<td>Holds pairs of step numbers and total energies</td>
</tr>
</tbody>
</table>
I.22 Module src.propagators.objects.RMT

I.22.1 Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the propagator.</td>
</tr>
<tr>
<td>Value:</td>
<td>&quot;RMT&quot;</td>
</tr>
<tr>
<td>parameters</td>
<td>Parameters and defaults.</td>
</tr>
<tr>
<td>Value:</td>
<td>'temp', 500, 'Ps', [0.0, 0.0,</td>
</tr>
<tr>
<td></td>
<td>0.0, 0.0, 0.0], 'S', [1.0, ...</td>
</tr>
</tbody>
</table>

I.22.2 Class RMT

src.propagators.objects.STS.STS  src.propagators.objects.RMT.RMT

This method is based on an extended Hamiltonian with 'm' additional thermostats, or degrees of freedom. The resulting Hamiltonian gives rise to implicit coupling between the variables, requiring implicit symplectic methods. Since this is undesirable a Hamiltonian splitting method is used to produce 2+m Hamiltonian systems which can be solved explicitly. This splitting is described in: http://www.nd.edu/~csweet1/campus/rmtsplit.pdf

Methods

```python
init(self, phys, forces, prop)
```

Initialize propagator.

Parameters

- **phys**: The physical system.
  
  *(type=Physical)*

- **forces**: MDL Forces object.
  
  *(type=Forces)*

- **prop**: MDL Propagator object.
  
  *(type=Propagator)*
halfUpdtH2\((self, \text{typ}, \text{phys}, \text{forces}, \text{prop})\)

H2 half step.

**Parameters**

- **typ**: 0 (first cycle) or 1 (second cycle) 
  \((type=\text{int})\)
- **phys**: The physical system. 
  \((type=\text{Physical})\)
- **forces**: MDL Forces object. 
  \((type=\text{Forces})\)
- **prop**: MDL Propagator object. 
  \((type=\text{Propagator})\)

halfUpdtH3j\((self, \text{dir}, \text{prop})\)

H3 half step.

**Parameters**

- **dir**: 0 direction \(j=1\ldots M\), \(\text{dir}=1\) direction \(j=M\ldots1\) 
  \((type=\text{int})\)
- **prop**: MDL Propagator object. 
  \((type=\text{Propagator})\)

halfUpdtH31\((self, \text{prop})\)

H31 half step.

**Parameters**

- **prop**: MDL Propagator object. 
  \((type=\text{Propagator})\)

UpdtH1\((self, \text{phys}, \text{forces}, \text{prop})\)

H1 full step.

**Parameters**

- **phys**: The physical system. 
  \((type=\text{Physical})\)
- **forces**: MDL Forces object. 
  \((type=\text{Forces})\)
- **prop**: MDL Propagator object. 
  \((type=\text{Propagator})\)
**totalEnergy**(*self, typ, phys, forces*)

Calculate total energy of the system.

**Parameters**

- **typ:** 0 calc h0, 1 calc non time reparam Energy, 2 calc total
  
  (*type=int*)

- **phys:** The physical system.
  
  (*type=Physical*)

- **forces:** MDL Forces object.
  
  (*type=Forces*)

---

**prodSs**(*self, start, end*)

Returns cumulative product of S parameters

**Parameters**

- **start:** Starting index
  
  (*type=int*)

- **end:** Ending index
  
  (*type=int*)

---

**run**(*self, phys, forces, prop*)

Run the propagator. Solves for half step in H2,H31,...,H3m then full step in H1 followed by half steps in H3m,...,H31,H2 so that method is time reversible.

**Parameters**

- **phys:** The physical system.
  
  (*type=Physical*)

- **forces:** MDL Forces object.
  
  (*type=Forces*)

- **prop:** MDL Propagator object.
  
  (*type=Propagator*)
**finish**(self, phys, forces, prop)

Finalize the propagator.

**Parameters**

- **phys**: The physical system.  
  *(type=Physical)*
- **forces**: MDL Forces object.  
  *(type=Forces)*
- **prop**: MDL Propagator object.  
  *(type=Propagator)*

**Instance Variables**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Potnl</td>
<td>Potential energy</td>
</tr>
<tr>
<td>gkT</td>
<td>Number of Dof*kT, momentum conserved so number of atoms * 3D - 3</td>
</tr>
<tr>
<td>KEtoT</td>
<td>Conversion of kinetic energy to temperature</td>
</tr>
<tr>
<td>Nf</td>
<td>number of Dof</td>
</tr>
<tr>
<td>kT</td>
<td>Boltzmann constant times Kelvin temperature</td>
</tr>
<tr>
<td>h0</td>
<td>Initial total energy</td>
</tr>
<tr>
<td>stepsdone</td>
<td>Number of steps completed</td>
</tr>
<tr>
<td>avTemp</td>
<td>Average Kelvin temperature</td>
</tr>
<tr>
<td>tempers</td>
<td>Holds pairs of step numbers and average temperatures</td>
</tr>
<tr>
<td>Hamiltonian</td>
<td>Holds pairs of step numbers and total energies</td>
</tr>
</tbody>
</table>
I.23 Module src.propagators.objects.STS

I.23.1 Class STS

STSIntegrator.STSIntegrator ─── src.propagators.objects.STS.STS

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init(self)</td>
<td>Finalize the propagator.</td>
</tr>
<tr>
<td>run(self)</td>
<td>Finalize the propagator.</td>
</tr>
<tr>
<td>finish(self)</td>
<td>Finalize the propagator.</td>
</tr>
</tbody>
</table>
I.24 Package src.toplevel

I.24.1 Modules

- Constants (Section I.25, p. 248)
- Energies (Section I.26, p. 251)
- FTSM (Section I.27, p. 256)
- ForceField (Section I.28, p. 263)
- Forces (Section I.29, p. 266)
- IO (Section I.30, p. 268)
- Physical (Section I.31, p. 281)
- Propagator (Section I.32, p. 290)
## I.25 Module src.toplevel.Constants

### I.25.1 Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>boltzmann()</td>
<td>Boltzmann constant, units kcal/mol K^-1</td>
</tr>
<tr>
<td>epsilon()</td>
<td>10^-14</td>
</tr>
<tr>
<td>tiny()</td>
<td>10^-20</td>
</tr>
<tr>
<td>timeFactor()</td>
<td>Timestep scaling factor for propagation.</td>
</tr>
<tr>
<td>invTimeFactor()</td>
<td>Inverse of the timestep scaling factor for propagation.</td>
</tr>
<tr>
<td>periodicBoundaryTolerance()</td>
<td>Size of buffer zone for the periodic cell.</td>
</tr>
<tr>
<td>sqrtCoulombConstant()</td>
<td>sqrt(k), for electrostatic energy (kq1q2/r).</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>pressureFactor()</td>
<td></td>
</tr>
<tr>
<td>pdbVelScalingFactor()</td>
<td>Scaling factor for velocities from a PDB file.</td>
</tr>
<tr>
<td></td>
<td>(type=float)</td>
</tr>
<tr>
<td>coulombFactor()</td>
<td>Scaling factor to convert between units of Vm and C.</td>
</tr>
<tr>
<td></td>
<td>(type=float)</td>
</tr>
<tr>
<td>electronCharge()</td>
<td>Charge of an electron in Coulombs.</td>
</tr>
<tr>
<td></td>
<td>(type=float)</td>
</tr>
<tr>
<td>aaTom()</td>
<td>Conversion from Angstroms to meter.</td>
</tr>
<tr>
<td></td>
<td>(type=float)</td>
</tr>
<tr>
<td>avogadro()</td>
<td>Avogadro’s number (atoms per mol).</td>
</tr>
<tr>
<td></td>
<td>(type=float)</td>
</tr>
<tr>
<td>amuToKg()</td>
<td>Scaling factor to convert AMU to SI kg.</td>
</tr>
<tr>
<td></td>
<td>(type=float)</td>
</tr>
<tr>
<td>Function</td>
<td>Return Value</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>kcalToJoule()</td>
<td>Scaling factor to convert kcal to SI Joules.</td>
</tr>
<tr>
<td></td>
<td><em>(type=float)</em></td>
</tr>
<tr>
<td>fsTime()</td>
<td>Number of fs per second <em>(10^-15)</em></td>
</tr>
<tr>
<td></td>
<td><em>(type=float)</em></td>
</tr>
<tr>
<td>siBoltzmann()</td>
<td>Boltzmann constant in SI units <em>(J/K)</em></td>
</tr>
<tr>
<td></td>
<td><em>(type=float)</em></td>
</tr>
</tbody>
</table>
I.26   Module src.toplevel.Energies

I.26.1   Class Energies

ProtoMol::ScalarStructure  

src.toplevel.Energies.Energies

Holds the system energies
These include: potential (bond, angle, ...) and kinetic.

Inherits from precompiled class ScalarStructure
This manages energies in a table. The inherited method getTable() 
will index this table.

This provides several inherited methods which are used:
- potentialEnergy()
- intoAdd()
- intoSubtract()
- molecularVirial()
- computeVirial()

Methods

```
_add_(self, rhs)
```

Overloads the ‘+’ operator to add two energy structures 
This simply adds corresponding energy terms for cumulative bond energy, etc.

**Parameters**

- **rhs**: The second Energies object.
  
  *(type=Energy)*

**Return Value**

The sum of the two Energies objects.

*(type=Energy)*
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| `__sub__(self, rhs)` | Overloads the `-` operator to add two energy structures. This simply subtracts corresponding energy terms for cumulative bond energy, etc.  
**Parameters**  
  - `rhs`: The second Energies object.  
  * (type=Energies)  
**Return Value**  
  - The difference of the two Energies objects.  
  * (type=Energies) |
| `computeMolecularVirial(self)` | Tells the energies structure to include the molecular virial tensor when calculating terms.  
**Return Value**  
  - The previous state without the virial.  
  * (type=Energies) |
| `computeVirial(self)` | Tells the energies structure to include the virial tensor when calculating terms.  
**Return Value**  
  - The previous state without the virial.  
  * (type=Energies) |
| `addBondEnergy(self, r)` | Accumulate into the bond energy  
**Parameters**  
  - `r`: Quantity to accumulate.  
  * (type=float) |
| `addAngleEnergy(self, r)` | Accumulate into the angle energy  
**Parameters**  
  - `r`: Quantity to accumulate.  
  * (type=float) |
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Parameters</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>addDihedralEnergy(self, r)</td>
<td>Accumulate into the dihedral energy</td>
<td>r: Quantity</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to accumulate.</td>
<td></td>
</tr>
<tr>
<td>addImproperEnergy(self, r)</td>
<td>Accumulate into the improper energy</td>
<td>r: Quantity</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to accumulate.</td>
<td></td>
</tr>
<tr>
<td>addShadowEnergy(self, r)</td>
<td>Accumulate into the shadow energy</td>
<td>r: Quantity</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to accumulate.</td>
<td></td>
</tr>
<tr>
<td>addCoulombEnergy(self, r)</td>
<td>Accumulate into the electrostatic energy</td>
<td>r: Quantity</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to accumulate.</td>
<td></td>
</tr>
<tr>
<td>addLJEnergy(self, r)</td>
<td>Accumulate into the van der Waals energy</td>
<td>r: Quantity</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to accumulate.</td>
<td></td>
</tr>
<tr>
<td>coulombEnergy(self)</td>
<td>Return Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Electrostatic energy</td>
<td></td>
<td>float</td>
</tr>
<tr>
<td>Function</td>
<td>Return Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ljEnergy(self)</td>
<td>van der Waals energy (type=float)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bondEnergy(self)</td>
<td>Energy due to two-atom bond deviations from equilibrium lengths. (type=float)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>angleEnergy(self)</td>
<td>Energy due to three-atom angle deviations from equilibrium angles. (type=float)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dihedralEnergy(self)</td>
<td>Energy due to four-atom dihedral deviations from equilibrium angles. (type=float)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>improperEnergy(self)</td>
<td>Energy due to four-atom improper deviations from equilibrium angles. (type=float)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shadowEnergy(self)</td>
<td>Shadow energy. (type=float)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kineticEnergy(self, phys)</td>
<td>Kinetic energy, as a sum of 0.5<em>m</em>v^2 for each atom. (type=float)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>totalEnergy</strong>((self, phys))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total energy, as a sum of potential and kinetic.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>((\text{type}=\text{float}))</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## I.27.1 Functions

**linspace**($xmin$, $xmax$, $N$)

N-element linear space between $xmin$ and $xmax$

**Parameters**

- $xmin$: Lower boundary
  
  (type=float)

- $xmax$: Upper boundary
  
  (type=float)

- $N$: Number of elements
  
  (type=int)

**Return Value**

N-element list of evenly spaced points

(type=list)

**extractPhi**($phipsi$)

Extract phi values of a string.

**Parameters**

- $phipsi$: List of [phi, psi]
  
  (type=list)

**extractPsi**($phipsi$)

Extract psi values of a string.

**Parameters**

- $phipsi$: List of [phi, psi]
  
  (type=list)
### setConstraint(ff, phi, psi)

Find the harmonic dihedral forces, and set their reference dihedrals

**Parameters**

- **ff**: MDL force field
  
  *(type=ForceField)*

- **phi**: Reference phi value in radians
  
  *(type=float)*

- **psi**: Reference psi value in radians
  
  *(type=float)*

### switchPhiPsi(S)

Switch phi and psi in the passed string.

**Parameters**

- **S**: List of [phi, psi]
  
  *(type=list)*

### norm(v3d)

Compute the norm of a 3-element vector.

**Parameters**

- **v3d**: 3-element vector
  
  *(type=numpy.ndarray)*

**Return Value**

The norm

*(type=float)*

### norm2(v3d)

Compute the norm squared of a 3-element vector.

**Parameters**

- **v3d**: 3-element vector
  
  *(type=numpy.ndarray)*

**Return Value**

The norm squared

*(type=float)*
**circshift**(*v*)
Shift all elements of the passed vector by one. Move the last element to the front.

**Parameters**

\[ v: \text{ Vector to shift.} \]
\[(type=\text{list})\]

**Return Value**

Shifted vector
\[(type=\text{list})\]

**cumsum**(*v*)
Return a vector with each element as the sum of all previous elements. Thus if [1 4 5] was passed, [1 5 10] would be returned.

**Parameters**

\[ v: \text{ Vector of elements to sum} \]
\[(type=\text{list})\]

**Return Value**

List of cumulative sums
\[(type=\text{list})\]

**M**(phys, alpha, beta)

**Parameters**

\[ \text{phys: } \text{The physical system} \]
\[(type=\text{Physical})\]

\[ \text{alpha: } \text{Index of the first dihedral} \]
\[(type=\text{int})\]

\[ \text{beta: } \text{Index of the second dihedral} \]
\[(type=\text{int})\]

**Return Value**

Value for M
\[(type=\text{float})\]
reparam\(z\)

High level smoothing/reparameterization routine. Invokes spline functions.

**Parameters**

\[ x: \text{X values}\]
\[ (type=list) \]
\[ y: \text{Y values}\]
\[ (type=list) \]

**Return Value**

Smoothed and reparameterized list of \([x, y]\) pairs
\[ (type=list) \]

---

I.27.2 Class SplineInterpolator

Cubic spline interpolator.

**Methods**

```python
__init__(self)
```

Set up the tridiagonal system

**Parameters**

\[ x: \text{x values}\]
\[ (type=list) \]
\[ y: \text{y values}\]
\[ (type=list) \]
\[ num: \text{Number of points}\]
\[ (type=int) \]

```python
SetTridiagonalSystem(self, x, y, num)
```

Set up the tridiagonal system

**Parameters**

\[ x: \text{x values}\]
\[ (type=list) \]
\[ y: \text{y values}\]
\[ (type=list) \]
\[ num: \text{Number of points}\]
\[ (type=int) \]

```python
FindLU(self, num)
```

Construct two bi-diagonal matrices \(L\) and \(U\).

**Parameters**

\[ num: \text{Number of points}\]
\[ (type=int) \]
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Parameters</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LBiDiSol</strong></td>
<td>Lower Bi-diagonal solver. Solves lower bi-diagonal system Lx = b</td>
<td><code>num</code> (Number of points, type=int)</td>
<td>Solution to upper bi-diagonal system, type=list</td>
</tr>
<tr>
<td><strong>UBiDiSol</strong></td>
<td>Upper Bi-diagonal solver.</td>
<td><code>num</code> (Number of points, type=int)</td>
<td>Solution to upper bi-diagonal system, type=list</td>
</tr>
<tr>
<td><strong>locate</strong></td>
<td>Locate where in the x points the passed value falls</td>
<td><code>x</code> (List of x values, type=list), <code>k</code> (Value, type=number)</td>
<td>int @return Index into x values</td>
</tr>
</tbody>
</table>

260
### EvalSpline(self, x, y, S, nx)
Evaluate approximated function at values nx

**Parameters**

- **x**: X values  
  *(type=list)*
- **y**: Y values  
  *(type=list)*
- **S**: Approximated function  
  *(type=list)*
- **nx**: Values at which to evaluate  
  *(type=list)*

**Return Value**

Approximated function values  
*(type=list)*

### Spline(self, x, y)
Highest level routine for setting up member matrices.

**Parameters**

- **x**: X values  
  *(type=list)*
- **y**: Y values  
  *(type=list)*

**Return Value**

Interpolated values  
*(type=list)*
SmoothingAndReparameterization(self, x, y, xx, yy, num)

Reparameterize by arc length

Parameters

- **x**: X values
  (type=list)
- **y**: Y values
  (type=list)
- **xx**: Linear space
  (type=list)
- **yy**: Corresponding spline values
  (type=list)
- **num**: Number of (x,y) points
  (type=int)

Return Value

- (x,y) points after reparameterization
  (type=list)

Instance Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>diag</td>
<td>Middle diagonal elements</td>
</tr>
<tr>
<td>u_diag</td>
<td>Upper diagonal elements</td>
</tr>
<tr>
<td>l_diag</td>
<td>Lower diagonal elements</td>
</tr>
<tr>
<td>R</td>
<td>Right hand side</td>
</tr>
<tr>
<td>L</td>
<td>Lower bidiagonal matrix</td>
</tr>
<tr>
<td>U</td>
<td>Upper bidiagonal matrix</td>
</tr>
<tr>
<td>xx</td>
<td>Holds solution</td>
</tr>
</tbody>
</table>
I.28  Module src.toplevel.ForceField

I.28.1  Class ForceField

ProtoMol::ForceGroup

src.toplevel.ForceField.ForceField

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>del</strong>(self)</td>
<td>Destructor.</td>
</tr>
<tr>
<td>setDefaults(self)</td>
<td>Set default values for all parameters, for all force evaluation algorithms.</td>
</tr>
<tr>
<td></td>
<td>This includes bonded and nonbonded, pairwise and fast electrostatic.</td>
</tr>
<tr>
<td><strong>setattr</strong>(self, s, t)</td>
<td></td>
</tr>
<tr>
<td>removeBondedForces(self)</td>
<td>Remove all bonded (bond, angle, dihedral, improper, harmonic dihedral) forces from the force field.</td>
</tr>
<tr>
<td>removeNonbondedForces(self)</td>
<td>Remove all nonbonded forces (van der Waals, electrostatic, magnetic dipole) from the force field.</td>
</tr>
<tr>
<td>removeMollyForces(self)</td>
<td>Remove all mollified forces from the force field.</td>
</tr>
<tr>
<td>bondedForces(self, inputstring)</td>
<td>Add bonded forces contained in the input string ('b', 'a', 'd', 'i', or 'h')</td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>inputstring:</td>
<td>Contains the characters representing the bonded forces to instantiate</td>
</tr>
<tr>
<td></td>
<td>(type=\text{string})</td>
</tr>
<tr>
<td>nonbondedForces(self, inputstring)</td>
<td>Add nonbonded forces contained in the input string ('l', 'c', 'm', or 'e'). If 'l' and 'c' are included a coupled (vdW)-electrostatic is instantiated for maximum performance.</td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>inputstring:</td>
<td>Contains the characters representing the nonbonded forces to instantiate</td>
</tr>
<tr>
<td></td>
<td>(type=\text{string})</td>
</tr>
</tbody>
</table>
### mollyForces

Add mollified forces contained in the input string, ('b' or 'a') are available.

**Parameters**

- **inputstring**: Contains the characters representing the mollified forces to instantiate
  
  *(type=string)*

### findForce

Search for a force in the array of force types.

**Parameters**

- **t**: Type of force ('b', 'a', etc. see above). Force can be bonded, nonbonded, or molly.
  
  *(type=char)*

**Return Value**

- Index of this type of force in the types array; -1 if not found.
  
  *(type=int)*

### breakLennardJonesCoulombForce

Breaks a coupled van der Waals - electrostatic force into individual calculations. This is invoked if the user specifies a different algorithm for van der Waals and electrostatic - for example if one is direct and the other uses a cutoff; they will not use the same set of atom pairs.

### calculateForces

Calculate all forces (except molly) in the force field, and populate the passed MDL forces object. This should be called from a Python-prototyped propagation function.

**Parameters**

- **phys**: The Physical system.
  
  *(type=Physical)*

- **forces**: MDL Forces object.
  
  *(type=Forces)*
calculateMollyForces(self, phys, forces)

Similar to calculateForces, but for mollified forces.

Parameters

phys: The Physical system.
(type=Physical)

forces: MDL Forces object.
(type=Forces)

Instance Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>Mapping from parameter names to default values for bonded and nonbonded forces, except fast electrostatics</td>
</tr>
<tr>
<td>fastelectro</td>
<td>Mapping from parameter names to default values for fast electrostatics</td>
</tr>
<tr>
<td>dirty</td>
<td>Dirty bit, allows for lazy building of force objects upon system propagation</td>
</tr>
<tr>
<td>forcetypes</td>
<td>List of force types, can contain 'b' (bond), 'a' (angle), 'd' (dihedral), 'i' (improper), 'h' (harmonic dihedral), 'l' (van der Waals), 'c' (electrostatic), 'e' (implicit solvation dielectric scaling), 'lc' (coupled vdW and electrostatic), 'm' (magnetic dipole), 'mb' (mollified bond) and 'ma' (mollified angle)</td>
</tr>
</tbody>
</table>
I.29 Module src.toplevel.Forces

I.29.1 Class Forces

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(self)</td>
<td>Method for initializing object.</td>
</tr>
<tr>
<td>dirty(self)</td>
<td>Checks if any of the force fields created is dirty.</td>
</tr>
</tbody>
</table>

Return Value

True if any of the force fields has been modified since the last propagation; false otherwise.

(type=boolean)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>getattr</strong>(self, name)</td>
<td>Method for getting attribute.</td>
</tr>
<tr>
<td><strong>setattr</strong>(self, name, val)</td>
<td>Method for setting attribute.</td>
</tr>
<tr>
<td>reset(self)</td>
<td>Reset data members to default values.</td>
</tr>
<tr>
<td>makeForceField(self, phys, *args)</td>
<td>Create a new MDL force field.</td>
</tr>
</tbody>
</table>

Parameters

phys: The physical system.

(type=Physical)

args: Python tuple, may have no values or a string "charmm"

(type=tuple)

Return Value

Newly instantiated MDL force field.

(type=ForceField)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>removeForceField(self, ff)</td>
<td>Method for removing force field.</td>
</tr>
</tbody>
</table>

Parameters

ff: MDL force field.

(type=ForceField)
randomForce(self, phys, seed)

Compute a random (x, y, z) force.

Parameters

phys: The physical system.
    (type=Physical)
seed: Random number generator seed.
    (type=integer)

Return Value

The random force as a three-element array (x,y,z)
    (type=numpy.ndarray)

build(self)

Using an MDL force factory, instantiate all force objects stored in the forcetypes data member of each force field. These will be SWIG-wrapped objects and are appended to the forcearray data member of each force field.

Instance Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>myForceFields</td>
<td>Array of MDL force fields</td>
</tr>
<tr>
<td>energies</td>
<td>Holds system energies</td>
</tr>
</tbody>
</table>
I.30 Module src.toplevel.IO

I.30.1 Class IO

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(self)</td>
<td></td>
</tr>
<tr>
<td><strong>setattr</strong>(self, att, val)</td>
<td></td>
</tr>
<tr>
<td>reset(self)</td>
<td>Reset the state of the IO object.</td>
</tr>
<tr>
<td>checkPath(self, filename)</td>
<td>If the passed filename does not exist, append the MDL root directory to it. This allows users to specify either absolute or relative paths for their input files.</td>
</tr>
</tbody>
</table>

**Parameters**

- **filename**: Absolute or relative path to a file.
  *(type=string)*

**Return Value**

- **string**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>readPSF(self, phys, psfname)</td>
<td>Read a PSF file and populate atomic positions.</td>
</tr>
</tbody>
</table>

**Parameters**

- **phys**: The physical system.
  *(type=Physical)*
- **psfname**: PSF file name.
  *(type=string)*

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>readPAR(self, phys, parname)</td>
<td>Read a CHARMM parameter file and populate the topology.</td>
</tr>
</tbody>
</table>

**Parameters**

- **phys**: The physical system.
  *(type=Physical)*
- **parname**: CHARMM parameter file name.
  *(type=string)*
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>readPDBPos</td>
<td>Read a PDB position file and populate atomic positions.</td>
<td></td>
</tr>
<tr>
<td>(self, phys, pdbname)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>The physical system. (type=Physical)</td>
<td></td>
</tr>
<tr>
<td>phys</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pdbname</td>
<td>PDB file name. (type=string)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Read a PDB velocity file and populate atomic velocities.</th>
</tr>
</thead>
<tbody>
<tr>
<td>readPDBVel</td>
<td>(self, phys, pdbname)</td>
</tr>
<tr>
<td>Parameters</td>
<td>The physical system. (type=Physical)</td>
</tr>
<tr>
<td>phys</td>
<td></td>
</tr>
<tr>
<td>pdbname</td>
<td>PDB file name. (type=string)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Read a XYZ position file and populate atomic positions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>readXYZPos</td>
<td>(self, phys, xyzname)</td>
</tr>
<tr>
<td>Parameters</td>
<td>The physical system. (type=Physical)</td>
</tr>
<tr>
<td>phys</td>
<td></td>
</tr>
<tr>
<td>xyzname</td>
<td>XYZ file name. (type=string)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Read a XYZ velocity file and populate atomic velocities.</th>
</tr>
</thead>
<tbody>
<tr>
<td>readXYZVel</td>
<td>(self, phys, xyzname)</td>
</tr>
<tr>
<td>Parameters</td>
<td>The physical system. (type=Physical)</td>
</tr>
<tr>
<td>phys</td>
<td></td>
</tr>
<tr>
<td>xyzname</td>
<td>XYZ file name. (type=string)</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| readDCDTrajectory(self, phys, dcdname) | Read a DCD trajectory file and populate atomic positions. This routine saves state, so that upon the next invocation the next trajectory will be read. | phys: The physical system.  
(type=Physical)  
dcdname: DCD trajectory file name.  
(type=string) |
| readEigenvectors(self, phys, eigname) | Read a eigenvector file and populate normal mode data.                     | phys: The physical system.  
(type=Physical)  
eigname: Eigenvector file name.  
(type=string) |
| writePSF(self, phys, psfname)  | Write atomic positions to a PSF file.                                      | phys: The physical system.  
(type=Physical)  
psfname: PSF file name.  
(type=string) |
| writePAR(self, phys, parname)  | Write a CHARMM parameter file.                                              | phys: The physical system.  
(type=Physical)  
parname: CHARMM parameter file name.  
(type=string) |
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>writePDBPos(self, phys, pdbname)</td>
<td>Write atomic positions to a PDB file.</td>
<td>phys: The physical system. (type=Physical)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pdbname: PDB file name. (type=string)</td>
</tr>
<tr>
<td>writePDBVel(self, phys, pdbname)</td>
<td>Write atomic velocities to a PDB file.</td>
<td>phys: The physical system. (type=Physical)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pdbname: PDB file name. (type=string)</td>
</tr>
<tr>
<td>writeXYZPos(self, phys, xyzname)</td>
<td>Write atomic positions to a XYZ file.</td>
<td>phys: The physical system. (type=Physical)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pdbname: XYZ file name. (type=string)</td>
</tr>
<tr>
<td>writeXYZVel(self, phys, xyzname)</td>
<td>Write atomic velocities to a XYZ file.</td>
<td>phys: The physical system. (type=Physical)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pdbname: XYZ file name. (type=string)</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Parameters</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>------------</td>
</tr>
</tbody>
</table>
| `writeXYZBinPos` | Write atomic positions to a binary XYZ file. | `phys`: The physical system. *(type=Physical)*  
    `pdbname`: XYZ file name. *(type=string)* |
| `writeXYZBinVel` | Write atomic velocities to a binary XYZ file. | `phys`: The physical system. *(type=Physical)*  
    `pdbname`: XYZ file name. *(type=string)* |
| `runOutput` | Run all registered outputs. For propagator objects this is called automatically, but for propagator functions it needs to be called from within. | `phys`: Physical system *(type=Physical)*  
    `forces`: MDL forces object *(type=Forces)*  
    `step`: Step number *(type=int)*  
    `ts`: Timestep *(type=float)* |
| `setPMV` | Only invoked if Pmv is being used, accepts an object for the molecular viewer. | `pmvobj`: Pmv ViewerFramework object *(type=ViewerFramework)* |
newGraph(self, xlab, ylab)

Create and return a new Gnuplot or Matplotlib graph object (thus return type is flexible depending on which is used).

**Parameters**

- **xlab**: Label for x-axis  
  *(type=string)*
- **ylab**: Label for y-axis  
  *(type=string)*

plotVector(self, prop, graph, vec, rangex=[], rangey=[])

Plot a vector of (x,y) data of the form \([x1,y1],[x2,y2],...\]

**Parameters**

- **prop**: MDL Propagator object  
  *(type=Propagator)*
- **graph**: Gnuplot or Matplotlib graph  
  *(type=Gnuplot or Matplotlib graph object)*
- **vec**: (x, y) data  
  *(type=list)*
- **rangex**: Plotting range for x-axis (default is to dynamically adjust to the data)  
  *(type=pair)*
- **rangey**: Plotting range for y-axis (default is to dynamically adjust to the data)  
  *(type=pair)*

plotQuantity(self, step, quantity, name)

Plot the passed step and quantity using a specific graph name.

**Parameters**

- **step**: Simulation step number  
  *(type=int)*
- **quantity**: Observable value  
  *(type=float)*
- **name**: Observable name  
  *(type=string)*
plotPotential(self, phys, forces, step)

Instantaneously plot the potential energy of the system. Note: this function is invoked automatically if a plot for the potential energy was registered in the plots dictionary. Thus, the user more often than not will not call this explicitly, since this assumes a plot for potential energy has been created already.

Parameters

phys: The physical system
(type=Physical)

forces: MDL Forces object
(type=Forces)

step: Simulation step number
(type=int)

plotKinetic(self, phys, forces, step)

Similar, for kinetic energy

Parameters

phys: The physical system
(type=Physical)

forces: MDL Forces object
(type=Forces)

step: Simulation step number
(type=int)

plotTotal(self, phys, forces, step)

Similar, for total energy

Parameters

phys: The physical system
(type=Physical)

forces: MDL Forces object
(type=Forces)

step: Simulation step number
(type=int)
plotTemperature(self, phys, forces, step)

Similar, for temperature

Parameters

phys: The physical system
    (type=Physical)

forces: MDL Forces object
    (type=Forces)

step: Simulation step number
    (type=int)

plotPressure(self, phys, forces, step)

Similar, for pressure

Parameters

phys: The physical system
    (type=Physical)

forces: MDL Forces object
    (type=Forces)

step: Simulation step number
    (type=int)

plotVolume(self, phys, forces, step)

Similar, for system volume

Parameters

phys: The physical system
    (type=Physical)

forces: MDL Forces object
    (type=Forces)

step: Simulation step number
    (type=int)
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>plotCoulombEnergy</td>
<td>Similar, for electrostatic energy</td>
<td>phys: The physical system (type=Physical)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>forces: MDL Forces object (type=Forces)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>step: Simulation step number (type=int)</td>
</tr>
<tr>
<td>plotLJEnergy</td>
<td>Similar, for van der Waals energy</td>
<td>phys: The physical system (type=Physical)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>forces: MDL Forces object (type=Forces)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>step: Simulation step number (type=int)</td>
</tr>
<tr>
<td>plotBondEnergy</td>
<td>Similar, for energy between two-atom bonds</td>
<td>phys: The physical system (type=Physical)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>forces: MDL Forces object (type=Forces)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>step: Simulation step number (type=int)</td>
</tr>
</tbody>
</table>
### plotAngleEnergy(self, phys, forces, step)

Similar, for energy between three-atom angles

**Parameters**

- **phys**: The physical system  
  *(type=Physical)*
- **forces**: MDL Forces object  
  *(type=Forces)*
- **step**: Simulation step number  
  *(type=int)*

### plotDihedralEnergy(self, phys, forces, step)

Similar, for energy between four-atom dihedrals

**Parameters**

- **phys**: The physical system  
  *(type=Physical)*
- **forces**: MDL Forces object  
  *(type=Forces)*
- **step**: Simulation step number  
  *(type=int)*

### plotImproperEnergy(self, phys, forces, step)

Similar, for energy between four-atom impropers

**Parameters**

- **phys**: The physical system  
  *(type=Physical)*
- **forces**: MDL Forces object  
  *(type=Forces)*
- **step**: Simulation step number  
  *(type=int)*
plotShadowEnergy(self, phys, forces, step)

Similar, for shadow energy

Parameters

phys: The physical system
  (type=Physical)
forces: MDL Forces object
  (type=Forces)
step: Simulation step number
  (type=int)

runPlots(self, phys, forces, step, ts)

Run all plots registered in the plots dictionary.

Parameters

phys: The physical system
  (type=Physical)
forces: MDL Forces object
  (type=Forces)
step: Simulation step number
  (type=int)
ts: Simulation timestep
  (type=float)

build(self)

Instantiate all file I/O and plots.

recache(self, phys, forces)

Restore the output cache.

Parameters

phys: The physical system
  (type=Physical)
forces: MDL Forces object
  (type=Forces)
run(self, phys, forces, step, ts, *args)

Run all plots registered in the plots dictionary.

Parameters

phys: The physical system
  (type=Physical)

forces: MDL Forces object
  (type=Forces)

step: Simulation step number
  (type=int)

ts: Simulation timestep
  (type=float)

args: Extra parameters (if necessary)
  (type=tuple)

Instance Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>myOutputs</td>
<td>LIST OF DESIRED OUTPUTS</td>
</tr>
<tr>
<td>myPlots</td>
<td>LIST OF DESIRED PLOTS (Python functions)</td>
</tr>
<tr>
<td>doMPL</td>
<td>USING MATPLOTLIB? USE GNUPLOT IF FALSE</td>
</tr>
<tr>
<td>pause</td>
<td>FREQUENCY FOR PAUSING (DEFAULT 0 = NEVER)</td>
</tr>
<tr>
<td>graphLabelsX</td>
<td>ARRAY OF X-AXIS GRAPH LABELS</td>
</tr>
<tr>
<td>graphLabelsY</td>
<td>ARRAY OF Y-AXIS GRAPH LABELS</td>
</tr>
<tr>
<td>xyData</td>
<td>MAP FROM GRAPH NAME TO XY PAIRS FOR DATA</td>
</tr>
<tr>
<td>graphs</td>
<td>MAP FROM GRAPH NAME TO GNUPLOT GRAPH OBJECT</td>
</tr>
<tr>
<td>xData</td>
<td>MAP FROM GRAPH NAME TO X DATA</td>
</tr>
<tr>
<td>yData</td>
<td>MAP FROM GRAPH NAME TO Y DATA</td>
</tr>
<tr>
<td>figures</td>
<td>MAP FROM GRAPH NAME TO MATPLOTLIB OBJECT</td>
</tr>
<tr>
<td>mplFigCount</td>
<td>NUMBER OF MATPLOTLIB OBJECT AT THE FRONT</td>
</tr>
<tr>
<td>doPmv</td>
<td>ARE WE USING PMV? (DEFAULT False = NO)</td>
</tr>
<tr>
<td>pmvobj</td>
<td>PMV PYTHON OBJECT, REFERENCING GUI</td>
</tr>
<tr>
<td>cmdlog</td>
<td>ARRAY OF EXECUTED PYTHON COMMANDS</td>
</tr>
<tr>
<td>pmvMODE</td>
<td>PMV MODE (0=STOP, 1=GO, 2=PAUSE)</td>
</tr>
</tbody>
</table>

continued on next page
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcdfiles</td>
<td>Array of DCD filenames</td>
</tr>
<tr>
<td>screen</td>
<td>Frequency to perform screen output</td>
</tr>
<tr>
<td>plots</td>
<td>Map of plot names to frequency; all default to -1 (never)</td>
</tr>
<tr>
<td>files</td>
<td>Map of file output names to (filename freq), default is (”, -1) - no file, never</td>
</tr>
<tr>
<td>dirty</td>
<td>Dirty bit, set to 1 if data members have been modified since the last propagation</td>
</tr>
</tbody>
</table>
I.31 Module src.toplevel.Physical

I.31.1 Class Atom

An atom in the system.

Methods

```
__init__(self, n, s, rs, rn, an, at, c, m)
```

Instance Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Atom number</td>
</tr>
<tr>
<td>seg_id</td>
<td>Segment identifier</td>
</tr>
<tr>
<td>residue_sequence</td>
<td>Residue sequence</td>
</tr>
<tr>
<td>residue_name</td>
<td>Residue name</td>
</tr>
<tr>
<td>atom_name</td>
<td>Atom name</td>
</tr>
<tr>
<td>atom_type</td>
<td>Atom type</td>
</tr>
<tr>
<td>charge</td>
<td>Charge [e]</td>
</tr>
<tr>
<td>mass</td>
<td>Mass [amu]</td>
</tr>
</tbody>
</table>

I.31.2 Class Bond

A two-atom bond.

Methods

```
__init__(self, n1, a1, a2)
```

Instance Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Bond number</td>
</tr>
<tr>
<td>atom1</td>
<td>Atom index 1</td>
</tr>
<tr>
<td>atom2</td>
<td>Atom index 2</td>
</tr>
</tbody>
</table>

I.31.3 Class Angle

A three-atom angle.

Methods

```
__init__(self, n1, a1, a2, a3)
```

Instance Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Bond number</td>
</tr>
</tbody>
</table>

continued on next page
I.31.4 Class Dihedral

A four-atom dihedral.

Methods

```python
__init__(self, n, a1, a2, a3, a4)
```

Instance Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Dihedral number</td>
</tr>
<tr>
<td>atom1</td>
<td>Atom index 1</td>
</tr>
<tr>
<td>atom2</td>
<td>Atom index 2</td>
</tr>
<tr>
<td>atom3</td>
<td>Atom index 3</td>
</tr>
<tr>
<td>atom4</td>
<td>Atom index 4</td>
</tr>
</tbody>
</table>

I.31.5 Class Improper

A four-atom improper.

Methods

```python
__init__(self, n, a1, a2, a3, a4)
```

Instance Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Bond number</td>
</tr>
<tr>
<td>atom1</td>
<td>Atom index 1</td>
</tr>
<tr>
<td>atom2</td>
<td>Atom index 2</td>
</tr>
<tr>
<td>atom3</td>
<td>Atom index 3</td>
</tr>
<tr>
<td>atom4</td>
<td>Atom index 4</td>
</tr>
</tbody>
</table>

I.31.6 Class HDonor

An H+ donor

Methods

```python
__init__(self, n, a1, a2)
```

Instance Variables
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Donor number</td>
</tr>
<tr>
<td>atom1</td>
<td>Atom index 1</td>
</tr>
<tr>
<td>atom2</td>
<td>Atom index 2</td>
</tr>
</tbody>
</table>

I.31.7 Class HAcceptor

An H+ acceptor

Methods

```python
__init__(self, n, a1, a2)
```

Instance Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Acceptor number</td>
</tr>
<tr>
<td>atom1</td>
<td>Atom index 1</td>
</tr>
<tr>
<td>atom2</td>
<td>Atom index 2</td>
</tr>
</tbody>
</table>

I.31.8 Class Physical

Defines a physical system: positions, velocities, temperature, boundary conditions, etc.

Methods

```python
__init__(self)

copy(self)
Perform a deep copy, avoid reference assignment

__getattr__(self, name)

__setattr__(self, name, val)

reset(self)
Reset all member variables to default values
```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Parameters</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pressure</code></td>
<td>Pressure of the system.</td>
<td><code>forces</code>: MDL Forces object</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>(type=Forces)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Return Value</strong></td>
<td>System pressure</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>(type=float)</code></td>
<td></td>
</tr>
<tr>
<td><code>volume</code></td>
<td>Volume of the system.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Return Value</strong></td>
<td>System volume</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>(type=float)</code></td>
<td></td>
</tr>
<tr>
<td><code>numAtoms</code></td>
<td>Number of atoms.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Return Value</strong></td>
<td>Number of atoms.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>(type=int)</code></td>
<td></td>
</tr>
<tr>
<td><code>numBonds</code></td>
<td>Number of bonds.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Return Value</strong></td>
<td>Number of bonds</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>(type=int)</code></td>
<td></td>
</tr>
<tr>
<td><code>numAngles</code></td>
<td>Number of angles</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Return Value</strong></td>
<td>Number of angles</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>(type=int)</code></td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
<td>Return Value</td>
<td>Parameters</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>numDihedrals</td>
<td>Number of dihedrals</td>
<td>Number of dihedrals (type=int)</td>
<td></td>
</tr>
<tr>
<td>numImpropers</td>
<td>Number of impropers</td>
<td>Number of impropers (type=int)</td>
<td></td>
</tr>
<tr>
<td>numDonors</td>
<td>Number of hydrogen donors (for H+ bonding)</td>
<td>Number of hydrogen donors (type=int)</td>
<td>index: Atom index (1 to N) (type=int)</td>
</tr>
<tr>
<td>numAcceptors</td>
<td>Number of hydrogen acceptors (for H+ bonding)</td>
<td>Number of hydrogen acceptors (type=int)</td>
<td></td>
</tr>
<tr>
<td>atom</td>
<td>Get an atom at the passed index.</td>
<td>The atom at the passed index.</td>
<td>index: Atom index (1 to N) (type=int)</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
<td>Parameters</td>
<td>Return Value</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------</td>
<td>-------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>bond(self, index)</td>
<td>Get a bond at the passed index.</td>
<td>index: Bond index (type=int)</td>
<td>The bond at the passed index. (type=Bond)</td>
</tr>
<tr>
<td>dihedral(self, index)</td>
<td>Get a dihedral at the passed index.</td>
<td>index: Dihedral index (type=int)</td>
<td>The dihedral at the passed index. (type=Dihedral)</td>
</tr>
<tr>
<td>improper(self, index)</td>
<td>Get an improper at the passed index.</td>
<td>index: Improper index (type=int)</td>
<td>The improper at the passed index. (type=Improper)</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Parameters</td>
<td>Return Value</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------</td>
<td>---------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td><code>donor</code></td>
<td>Get an H+ donor at the passed index.</td>
<td><strong>index</strong>: H+ donor index (type=int)</td>
<td>The H+ donor at the passed index. (type=HDonor)</td>
</tr>
<tr>
<td><code>acceptor</code></td>
<td>Get an H+ acceptor at the passed index.</td>
<td><strong>index</strong>: H+ acceptor index (type=int)</td>
<td>The H+ acceptor at the passed index. (type=HAcceptor)</td>
</tr>
<tr>
<td><code>mass</code></td>
<td>Mass of an atom.</td>
<td><strong>atom</strong>: Atom index (1 to N) (type=int)</td>
<td>Atom mass [amu] (type=float)</td>
</tr>
<tr>
<td><code>getTemperature</code></td>
<td>System temperature (K)</td>
<td></td>
<td>Kelvin temperature (type=float)</td>
</tr>
</tbody>
</table>
angle(self, index)
Dihedral angle (rad) at passed index

Parameters

    index: Dihedral index
    (type=int)

Return Value

    Dihedral angle in radians
    (type=float)

calculateHessians(self, mollypos, angleFilter)
Calculate bond and angle Hessians for MOLLY propagators

Parameters

    mollypos: Mollified positions
    (type=numpy.ndarray)

    angleFilter: List of angle Hessians
    (type=ReducedHessAngleList)

updateCOM_Momenta(self)
Update center of mass and angular momentum

build(self)
Build the physical data.

Instance Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seed</td>
<td>Random number seed</td>
</tr>
<tr>
<td>exclude</td>
<td>Exclusion Pairs</td>
</tr>
<tr>
<td>cellsize</td>
<td>Cell size</td>
</tr>
<tr>
<td>bc</td>
<td>Boundary conditions</td>
</tr>
<tr>
<td>remcom</td>
<td>Remove COM motion?</td>
</tr>
<tr>
<td>remang</td>
<td>Remove angular momentum?</td>
</tr>
<tr>
<td>defaultCBV</td>
<td>Use default cell basis vectors (PBC)</td>
</tr>
<tr>
<td>time</td>
<td>Current time</td>
</tr>
<tr>
<td>cB1</td>
<td>Cell basis vector 1</td>
</tr>
<tr>
<td>cB2</td>
<td>Cell basis vector 2</td>
</tr>
<tr>
<td>cB3</td>
<td>Cell basis vector 3</td>
</tr>
<tr>
<td>cO</td>
<td>Cell origin</td>
</tr>
<tr>
<td>temperature</td>
<td>Kelvin temperature</td>
</tr>
<tr>
<td>masses</td>
<td>Diagonal mass matrix</td>
</tr>
<tr>
<td>invmasses</td>
<td>Diagonal inverse mass matrix</td>
</tr>
</tbody>
</table>

continued on next page
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>masssum</td>
<td>Sum over all atomic masses</td>
</tr>
<tr>
<td>dirty</td>
<td>Dirty bit</td>
</tr>
</tbody>
</table>
I.32 Module src.toplevel.Propagator

I.32.1 Class Propagator

Methods

```python
__init__(self, phys, forces, io)
reset(self)
Reset the state of the Propagator object.
isMDL(self, integ)
Determine whether or not a propagator has been coded in MDL.

Parameters

integ: MDL propagator object (STS or MTS)
(type=STS/MTS)

Return Value

True if the passed propagator is coded using MDL.
(type=boolean)

addPreInitModifier(self, integ, modifier)
Add a modifier to execute before propagator initialization.

Parameters

integ: MDL propagator object (STS or MTS)
(type=STS/MTS)

modifier: Routine which alters propagator behavior.
(type=function)

addPostInitModifier(self, integ, modifier)
Add a modifier to execute after propagator initialization.

Parameters

integ: MDL propagator object (STS or MTS)
(type=STS/MTS)

modifier: Routine which alters propagator behavior.
(type=function)
```
addPreRunModifier(self, integ, modifier)

Add a modifier to execute before propagator execution.

Parameters

integ: MDL propagator object (STS or MTS)
(type=STS/MTS)

modifier: Routine which alters propagator behavior.
(type=function)

addPostRunModifier(self, integ, modifier)

Add a modifier to execute after propagator execution.

Parameters

integ: MDL propagator object (STS or MTS)
(type=STS/MTS)

modifier: Routine which alters propagator behavior.
(type=function)

addPreForceModifier(self, integ, modifier)

Add a modifier to execute before force calculation.

Parameters

integ: MDL propagator object (STS or MTS)
(type=STS/MTS)

modifier: Routine which alters propagator behavior.
(type=function)

addPostForceModifier(self, integ, modifier)

Add a modifier to execute after force calculation.

Parameters

integ: MDL propagator object (STS or MTS)
(type=STS/MTS)

modifier: Routine which alters propagator behavior.
(type=function)
### runModifiers

#### Function:
```
runModifiers(self, modifiers, phys, forces, prop, integ)
```

Run modifiers of a propagator

#### Parameters

- **modifier**: A set of routines which alternates propagator behavior  
  *(type=list of functions)*
- **integ**: MDL propagator object (STS or MTS)  
  *(type=object)*

### timestep

#### Function:
```
timestep(self, integ)
```

Return the timestep of a propagator, scaled accordingly

#### Parameters

- **integ**: MDL propagator object (STS or MTS)  
  *(type=object)*

#### Return Value

The timestep (dt) of a propagator  
*(type=float)*

### calculateForces

#### Function:
```
calculateForces(self, forces)
```

Calculate forces and update the atomic force vector.

#### Parameters

- **integ**: MDL Forces object  
  *(type=Forces)*
- **forces**: MDL Forces object  
  *(type=Forces)*

### initNext

#### Function:
```
initNext(self, phys, forces)
```

For multiple timestepping, initialize the next propagator in the chain.

#### Parameters

- **integ**: MDL Physical object  
  *(type=Physical)*
- **integ**: MDL Forces object  
  *(type=Forces)*
- **phys**: MDL Physical object  
  *(type=Physical)*
- **forces**: MDL Forces object  
  *(type=Forces)*
runNext(self, phys, forces, cL)

For multiple timestepping, execute the next propagator in the chain.

Parameters

- **integ**: MDL Physical object
- **integ**: MDL Forces object
- **integ**: MDL IO object
- **cL**: Cycle length (number of times to execute the inner propagator) 
  *(type=integer)*
- **phys**: *(type=Physical)*
- **io**: *(type=IO)*
- **forces**: *(type=Forces)*

finishNext(self, phys, forces, prop)

For multiple timestepping, finish the next propagator in the chain.

propagate(self, scheme="Leapfrog", steps=0, cyclelength=-1, dt=0.1, forcefield=[], params={})

Propagate the system.

Parameters

- **name**: Name of the propagator to use. 
  *(type=string)*
- **steps**: Number of steps for execution. 
  *(type=integer)*
- **dt**: Timestep. 
  *(type=float)*
- **ff**: MDL ForceField object. 
  *(type=ForceField)*
- ***args**: Extra parameters unique for this propagation scheme. 
  *(This could be empty).*
  *(type=tuple)*

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>myPropagator</td>
<td>PROPAGATOR OBJECT (0 IF METHOD)</td>
</tr>
<tr>
<td>myStep</td>
<td>CURRENT SIMULATION STEP</td>
</tr>
<tr>
<td>myTimestep</td>
<td>PROPAGATOR TIMESTEP (fs)</td>
</tr>
<tr>
<td>myLevel</td>
<td>CURRENT PROPAGATOR HIERARCHY LEVEL (0 IS STS)</td>
</tr>
</tbody>
</table>
class RMT(STS):
    def init(self, phys, forces, prop):
        prop.calculateForces(forces)
        self.Potnl = forces.energies.potentialEnergy()
        self.gkT = 3.0*(phys.numAtoms())*Constants.boltzmann()
            *self.temp
        self.KEtoT = 2.0 / (3.0*(phys.numAtoms()-1.0)
            *Constants.boltzmann())
        self.Nf = 3.0*(phys.numAtoms())
        self.kT = Constants.boltzmann()*self.temp
        self.h0 = self.totalEnergy(0,phys,forces)

def halfUpdtH2(self,typ,phys,forces,prop):
    prodS = self.prodSs(1,self.NumStats)
    ii = 1
    sumSpot = 0.0
    while (ii < self.NumStats):
        sumSpot += (self.Nf+ii)*self.kT*log(self.S[ii]) + 0.5*
            (1.0-self.S[ii])*(1.0-self.S[ii])/self.C[ii]
        ii += 1
    self.Ps[0] -= 0.5*self.dt*prodS*(self.Potnl+sumSpot)
    ii = 1
    while (ii < self.NumStats):
        self.Ps[ii] -= 0.5*self.dt*phys.invmasses[self.S[ii]]
        self.Ps[ii] -= 0.5*self.dt*self.S[0]*(prodS/self.S[ii])*
            (self.Potnl+sumSpot+(self.Nf+ii)*self.kT
                -self.S[ii]*(1.0-self.S[ii])/self.C[ii])
        ii += 1
    if typ < 1:
        phys.velocities += forces.force*0.5*phys.invmasses
        self.OldProdS = prodS*self.S[0]
    else:
        temps = self.OldProdS/(prodS*self.S[0])

APPENDIX J

RECURSIVE MULTIPLE THERMOSTATTING (RMT)
phys.velocities *= tempS
phys.velocities += forces.force*0.5*self.dt*phys.invmasses

def halfUpdtH3j(self, dir, prop):
    kk = 1
    while (kk < self.NumStats):
        if dir > 0:
            jj = self.NumStats - kk
        else:
            jj = kk
        prodSlj = self.prodSs(0, jj)
        prodSgj = self.prodSs(jj + 1, self.NumStats)
        a = self.dt * prodSlj / (8.0 * self.Q[jj] * prodSgj)
        c = -self.Ps[jj]
        self.Ps[jj] = -2.0 * c / (1.0 + sqrt(1.0 - 4.0 * a * c))
        self.Sn = self.S[jj]
        self.S[jj] *= (1.0 + a) / (1.0 - a)
        ii = 0
        while (ii < jj):
            ii += 1
        ii = jj + 1
        while (ii < self.NumStats):
            ii += 1
        kk += 1

def halfUpdtH31(self, prop):
    prodS = self.prodSs(1, self.NumStats)
    a = self.dt / (8.0 * self.Q[0] * prodS)
    c = -self.Ps[0] - 0.25 * self.dt * prodS * self.h0
    self.Ps[0] = -2.0 * c / (1.0 + sqrt(1.0 - 4.0 * a * c))
    self.Sn = self.S[0]
    a = 0.25 * self.dt * self.Ps[0] / (self.Q[0] * prodS)
    self.S[0] *= (1.0 + a) / (1.0 - a)
    ii = 1
    while (ii < self.NumStats):
```python
self.Ps[ii] += 0.25*self.dt*((self.Sn+self.S[0])/self.S[ii])
    *(0.5*self.Ps[0]*self.Ps[0]/
     (self.Q[0]*prodS)+prodS*self.h0)

ii += 1
self.Ps[0] += 0.25*self.dt*(prodS*self.h0-0.5*self.Ps[0]
    *self.Ps[0]/(self.Q[0]*prodS))

def UpdtH1(self, phys, forces, prop):
    prodS = self.prodSs(0,self.NumStats)
    tempS = self.OldProdS/prodS
    phys.positions += phys.velocities*tempS*self.dt
    sKinetic = forces.energies.kineticEnergy(phys)*self.OldProdS
        *self.OldProdS/prodS
    self.Ps[0] += (self.dt/self.S[0])*(sKinetic-prodS*self.gkT
        *(1.0+log(self.S[0])))

    ii = 1
    while (ii < self.NumStats):
        self.Ps[ii] += (self.dt/self.S[ii])*(sKinetic
            - prodS*self.gkT*log(self.S[0]))
        ii += 1

def totalEnergy(self,typ,phys,forces):
    """
    Calculate total energy of the system.
    """
    prodS = self.prodSs(0,self.NumStats)
    resProdS = prodS/self.S[0]
    tempH = forces.energies.kineticEnergy(phys)+self.Potnl
        +self.gkT*log(self.S[0])+0.5*self.Ps[0]
            *self.Ps[0]/(self.Q[0]*resProdS*resProdS)
    ii = 1
    while (ii < self.NumStats):
        tempH += (self.Nf+ii)*self.kT*log(self.S[ii])+0.5*
            (1.0-self.S[ii])*(1.0-self.S[ii])/self.C[ii]
        resProdS /= self.S[ii]
        tempH += 0.5*self.Ps[ii]*self.Ps[ii]/
            (self.Q[ii]*resProdS*resProdS)
        ii += 1
    if typ > 0:
        tempH -= self.h0
    if typ > 1:
        tempH *= prodS
    return(tempH)
```
def prodSs(self,start,end):
    ""
    Returns cumulative product of S parameters
    ""
    ii = start
    prodS = 1.0
    while (ii < end):
        prodS *= self.S[ii]
        ii += 1
    return(prodS)

def run(self, phys, forces, prop):
    ""
    Run the propagator.
    Solves for half step in H2,H31,...,H3m
    then full step in H1
    followed by half steps in H3m,...,H31,H2
    so that method is time reversible.
    ""
    self.halfUpdtH2(0, phys, forces, prop)
    self.halfUpdtH31(prop)
    self.halfUpdtH3j(0, prop)
    self.UpdtH1(phys, forces, prop)
    prop.calculateForces(forces)
    self.Potnl = forces.energies.potentialEnergy()
    self.halfUpdtH3j(1, prop)
    self.halfUpdtH31(prop)
    self.halfUpdtH2(1, phys, forces, prop)

name = "RMT"
parameters = (’temp’, 500,
              ’Ps’, [0.0,0.0,0.0,0.0,0.0,0.0,0.0],
              ’S’, [1.0,1.0,1.0,1.0,1.0,1.0,1.0],
              ’Q’, [2.0,3.0,4.5,6.8,10.2],
              ’C’, [0.04,0.04,0.04,0.04,0.04],
              ’NumStats’, 5) #: Parameters and defaults.
APPENDIX K

MDL IMPLEMENTATION OF THE FINITE TEMPERATURE STRING
METHOD - FROM [38], WITH MODIFICATIONS.

The Finite Temperature String Method (FTSM, [166, 167]) is an algorithm for estimating the most probable path between two metastable states of a system through an initial guess and multiple refinements of the path. For an alanine dipeptide molecule, the $(\phi, \psi)$ angles of the backbone dihedral can be used to define this path. At each FTSM step, there are several intermediate conformations on the path which require refinement through a constrained MD run, followed by an averaging of data to update the path with new and improved values, closer to the most probable path. Once the path is updated, it is smoothed and reparameterized using normalized arc lengths. A serial `execute()` method for our prototype looks like this, if we were to update the string once (multiple updates could be done in a loop):

```python
def execute():
    slopes = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,
             1,1,1,1,1,1,1,1,1,1,1,1,1,1,
             1,1,1,1,1,1,1,1,1,1,1,1,1,1]
    S = initializeFTSM("alanineInitial.pdb",
                       "alanineFinal.pdb",
                       ff, 40)
    walk(S, 40)
    for conf in range (1, 41):
        S[conf] =
            FTSMAverage(conf, 1000,
```

298
slopes[conf])
S, slopes = reparameterize(smooth(S), 42)

The first call to initializeFTSM() generates an initial diagonal string of 40 equally spaced points between the initial and final conformations, passed as PDB files. Next, walk() generates intermediate PDB files (labeled conf1.pdb, conf2.pdb, etc.) for these 40 intermediate points, by a walk through the conformations using a weak restraint on the backbone dihedral. Then FTSMaverage() runs 1000 steps of constrained dynamics at a timestep of 0.5 fs (for a total of 0.5 ps), starting from each of these PDBs using strong backbone dihedral restraints and storing average $(\phi, \psi)$ in $S$. This is followed by a smoothing and reparameterization of $S$, the former of which uses the steepest decent method.

Averaging starting from a particular conformation can be performed in parallel, since it is independent of the averaging at other conformations. Thus, once we have our PDB files representing each conformation and constraints setup properly in the force field ff, we can run the averaging in parallel, dividing the intermediate conformations as evenly as possible among the nodes. Assuming that numconf holds the number of intermediate conformations, if we want to distribute the conformations sequentially (i.e. for four nodes and 40 conformations node 0 would get conformations 1-10; node 1 would get 11-20, node 2 would get 21-30 and node 3 would get 31-40), we can do the following, using mpi.rank as the node number (root is 0), and mpi.size as the total number of nodes.

```python
start = mpi.rank*(numconf/mpi.size)+1
modVal = numconf % mpi.size
if (mpi.rank <= modVal and mpi.rank != 0):
    start += mpi.rank
elif (mpi.rank != 0):
```

299
start += modVal
end = start+(numconf/mpi.size)-1
if ((mpi.rank+1) <= modVal):
    end += 1

Now start and end respectively hold the starting and ending conformations for each individual processor. We can now run averaging in a loop on each processor:

```python
currS = []
for conf in range(start, end+1):
    currS.append(FTSMaverage(conf, 1000,
                               slopes[conf]))
```

then collect results on the root, which we later will broadcast to all nodes after smoothing. We use `mpi.allgather` to concatenate the lists of updated \((\phi, \psi)\) pairs, then append the starting and ending points of the string, followed by an `mpi.barrier()` which ensures no nodes pass this point until all have finished averaging:

```python
tempS = mpi.allgather(currS)
S = S[0] + tempS + S[S.__len__()-1]
mpi.barrier()
```

We can leave smoothing and reparameterization as root node tasks. The updated string and slopes are returned into temporary arrays, which are subsequently broadcast to all nodes using `mpi.bcast()`. By placing an `mpi.barrier()` at the end, we ensure no nodes go back into running constrained dynamics before the root node finishes this process:

```python
if (mpi.rank == 0):
    tempS, tempslopes =
        reparameterize(smooth(S), 42)
else:
    tempS = []; tempslopes = []
S = mpi.bcast(tempS)
slopes = mpi.bcast(tempslopes)
mpi.barrier()
```
Tests were run on a 32-bit Atipa cluster running GNU Linux 2.6.13.1. Individual nodes contain dual Intel Xeon 2.4 GHz CPUs, with 1 GB RAM. Convergence was obtained in [167] after 60-70 updates. We show the wall clock execution times of 64 averagings of parallel FTSM for processor counts of 1, 2, 4, 6, and 10 in Figure K.1, averaged over four runs.

Figure K.1. Wall clock execution times of the parallelized averaging stage of FTSM with various processor counts.

Results illustrate significant savings between 1 and 4 processors, after which benefits level off slightly. With 10 processors, we were able to achieve about 45.3% parallel efficiency. Because averaging is the only FTSM routine currently parallelized, the overall savings is much less significant, shown in Figure K.2, although we were still able to obtain roughly 30 minute savings on a 90-minute single processor simulation. This graph shows that the overhead of root tasks and subsequent communication with large amounts of processors is currently high, but we expect future improvement when we will attempt to parallelize other phases of the algorithm besides averaging. An initial attempt to parallelize the walking stage using PDB files from the previous iteration before smoothing indicated more painful dragging from
an unsmoothed string to the smooth one as opposed to going point-by-point like we have been, which is non-parallelizable. However we will continue to explore other options, such as selective generation of PDBs while running constrained dynamics for use in subsequent walking stages. We will also look into piecewise smoothing of the string with each processor operating on its individual set of points (ideally, the same set as used in averaging), and study dependability of the resulting strings. Finally, I am in the process of implementing an improved version of the model [125] where string updates are conducted on-the-fly alongside short restrained dynamics runs.

Figure K.2. Overall wall clock execution times of parallel FTSM.

Figure K.3 shows our initial diagonal and final string after 64 updates on alanine dipeptide, superimposed and converging to the results of [167], shown as a dotted curve.
Figure K.3. Our initial diagonal string between two energetically favorable conformations of alanine dipeptide, and our finite temperature string after 64 updates. To illustrate convergence, we superimpose our results on those of [167] (dotted curve).
APPENDIX L

MDL AVAILABLE PROPAGATORS
<table>
<thead>
<tr>
<th>Propagator</th>
<th>S/M</th>
<th>Parameters and Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBK [27]</td>
<td>STS</td>
<td>temp (float, 300), gamma (float, 3.0), seed (int, 1234)</td>
</tr>
<tr>
<td>LangevinImpulse [177]</td>
<td>STS</td>
<td>temp (float, 300), gamma (float, 3.0), seed (int, 1234)</td>
</tr>
<tr>
<td>Leapfrog [86]</td>
<td>STS</td>
<td>none</td>
</tr>
<tr>
<td>NormModeMin [193]</td>
<td>STS</td>
<td>temp (float, 300), inertia (float, 0.5), bathpos (float, 1.0)</td>
</tr>
<tr>
<td>NoseHoover [149, 90]</td>
<td>STS</td>
<td>temp (float, 500), bathM (float, 0.0), bathP (float, 1.0), Q (float, 10.0)</td>
</tr>
<tr>
<td>NosePoincare [20]</td>
<td>STS</td>
<td>temp (float, 300), pressure (float, 100), omegaTo (float, 0.5), omegaTv (float, 2.5), tauP (float, 1.5)</td>
</tr>
<tr>
<td>NPTVerlet [172]</td>
<td>STS</td>
<td>temp (float, 300), omegaTo (float, 0.5)</td>
</tr>
<tr>
<td>PositionVerlet [201]</td>
<td>STS</td>
<td>none</td>
</tr>
<tr>
<td>RMT [195]</td>
<td>STS</td>
<td>temp (float, 500), Ps (list(5), [0.0 0.0 0.0 0.0 0.0]), S (list(5), [1.0,1.0,1.0,1.0,1.0]), Q (list(5), [2.0,3.0,4.5,6.8,10.2]), C (list(5), [0.04,0.04,0.04,0.04,0.04]), NumStats (int, 5)</td>
</tr>
<tr>
<td>SelfconsistentLeapfrog [154]</td>
<td>STS</td>
<td>temp (float, 300), gamma (float, 0.5), seed (int, 1234), iter (int, 5)</td>
</tr>
<tr>
<td>BSplineMolly [123]</td>
<td>MTS</td>
<td>type (string, 'short'), stepsize (float, 0.1)</td>
</tr>
<tr>
<td>EquilibriumMolly [124]</td>
<td>MTS</td>
<td>none</td>
</tr>
<tr>
<td>HMC [51]</td>
<td>MTS</td>
<td>temp (float, 285)</td>
</tr>
<tr>
<td>Impulse [187]</td>
<td>MTS</td>
<td>none</td>
</tr>
<tr>
<td>NormModeInt [193]</td>
<td>MTS</td>
<td>fixmodes (int, 1400), gamma (float, 80), seed (int, 1234), temp (float, 300), nve (int, 0), Berendsen (int, 0), fdof (int, 6)</td>
</tr>
<tr>
<td>NormModeCoarse [193]</td>
<td>MTS</td>
<td>fixmodes (int, 1400), gamma (float, 80), seed (int, 1234), temp (float, 300), nve (int, 0), Berendsen (int, 0), fdof (int, 6), dtrat (int, 1)</td>
</tr>
<tr>
<td>ShadowHMC [81]</td>
<td>MTS</td>
<td>temp (float, 285), order (int, 4), C (float, 4.0), optimize (int, 0), ratio (int, 2)</td>
</tr>
<tr>
<td>Umbrella [140]</td>
<td>MTS</td>
<td>none</td>
</tr>
</tbody>
</table>
APPENDIX M

MDL FORCE COMPUTATION ALGORITHMS AND PARAMETERS
TABLE M.1

PARAMETERS, DATATYPES AND DEFAULT VALUES FOR FORCE COMPUTATION.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Algorithm</th>
<th>Data Type</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>Ewald/PME</td>
<td>boolean</td>
<td>True</td>
<td>Compute the real term of the Ewald sum?</td>
</tr>
<tr>
<td>reciprocal</td>
<td>Ewald/PME</td>
<td>boolean</td>
<td>True</td>
<td>Compute the reciprocal term of the Ewald sum?</td>
</tr>
<tr>
<td>correction</td>
<td>Ewald/PME</td>
<td>boolean</td>
<td>True</td>
<td>Compute the correction term of the Ewald sum?</td>
</tr>
<tr>
<td>alpha</td>
<td>Ewald/PME</td>
<td>float</td>
<td>optimal</td>
<td>Scaling factor</td>
</tr>
<tr>
<td>accuracy</td>
<td>Ewald/PME</td>
<td>float</td>
<td>1e-5</td>
<td>Accuracy</td>
</tr>
<tr>
<td>expansion</td>
<td>Ewald/PME</td>
<td>float</td>
<td>3.0</td>
<td>Expansion factor</td>
</tr>
<tr>
<td>gridsize</td>
<td>PME</td>
<td>integer</td>
<td>5</td>
<td>Mesh size</td>
</tr>
<tr>
<td>interpolation</td>
<td>PME/MultiGrid</td>
<td>string</td>
<td>'BSpline'</td>
<td>Interpolation method (BSpline or Hermite)</td>
</tr>
<tr>
<td>direct</td>
<td>MultiGrid</td>
<td>boolean</td>
<td>True</td>
<td>Compute the direct term of the MultiGrid summation?</td>
</tr>
<tr>
<td>correction</td>
<td>MultiGrid</td>
<td>boolean</td>
<td>True</td>
<td>Compute the correction term of the MultiGrid summation?</td>
</tr>
<tr>
<td>smooth</td>
<td>MultiGrid</td>
<td>boolean</td>
<td>True</td>
<td>Compute the smoothing term of the MultiGrid summation?</td>
</tr>
<tr>
<td>levels</td>
<td>MultiGrid</td>
<td>integer</td>
<td>4</td>
<td>Number of inter/antepolation levels.</td>
</tr>
<tr>
<td>s</td>
<td>MultiGrid</td>
<td>float</td>
<td>1.0</td>
<td>Softening distance.</td>
</tr>
<tr>
<td>order</td>
<td>MultiGrid</td>
<td>integer</td>
<td>4</td>
<td>Interpolation order.</td>
</tr>
<tr>
<td>ratio</td>
<td>MultiGrid</td>
<td>integer</td>
<td>2</td>
<td>Fine-to-coarse grid size ratio.</td>
</tr>
<tr>
<td>coarsegridsize</td>
<td>MultiGrid</td>
<td>integer</td>
<td>5</td>
<td>Size of the coarsest grid</td>
</tr>
<tr>
<td>finagridsize</td>
<td>MultiGrid</td>
<td>integer</td>
<td>3</td>
<td>Size of the finest grid (VBC)</td>
</tr>
<tr>
<td>fingridorigin</td>
<td>MultiGrid</td>
<td>list(3)</td>
<td>[0,0,0]</td>
<td>Origin of the finest grid (VBC)</td>
</tr>
</tbody>
</table>
APPENDIX N

ProtoMol CLASSES WRAPPED FOR MDL
TABLE N.1

PROTOTOMOL FILES AND CORRESPONDING MDL MODULES.

<table>
<thead>
<tr>
<th>Module</th>
<th>C++</th>
<th>MDL Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>matutilities</td>
<td>matutilities</td>
</tr>
<tr>
<td>base</td>
<td>ReducedHessAngle</td>
<td>ReducedHessAngle</td>
</tr>
<tr>
<td>base</td>
<td>ScalarStructure</td>
<td>ScalarStructure</td>
</tr>
<tr>
<td>base</td>
<td>Vector3DBlock</td>
<td>Vector3DBlock</td>
</tr>
<tr>
<td>forces</td>
<td>AngleSystemForce</td>
<td>AngleForce</td>
</tr>
<tr>
<td>forces</td>
<td>BondSystemForce</td>
<td>BondForce</td>
</tr>
<tr>
<td>forces</td>
<td>NonbondedCutofSystemForce</td>
<td>CutofForce</td>
</tr>
<tr>
<td>forces</td>
<td>DihedralSystemForce</td>
<td>DihedralForce</td>
</tr>
<tr>
<td>forces</td>
<td>ForceGroup</td>
<td>ForceGroup</td>
</tr>
<tr>
<td>forces</td>
<td>HarmDihedralSystemForce</td>
<td>HarmDihedralForce</td>
</tr>
<tr>
<td>forces</td>
<td>ImproperSystemForce</td>
<td>ImproperForce</td>
</tr>
<tr>
<td>forces</td>
<td>NonbondedFullSystemForce</td>
<td>FullForce</td>
</tr>
<tr>
<td>forces</td>
<td>NonbondedMultiGridSystemForce</td>
<td>MultiGridForce</td>
</tr>
<tr>
<td>forces</td>
<td>NonbondedPMEwaldSystemForce</td>
<td>PMEwaldForce</td>
</tr>
<tr>
<td>forces</td>
<td>NonbondedFullEwaldSystemForce</td>
<td>FullEwaldForce</td>
</tr>
<tr>
<td>forces</td>
<td>NonbondedSimpleFullSystemForce</td>
<td>SimpleFullForce</td>
</tr>
<tr>
<td>forces</td>
<td>WrapperMetaForce</td>
<td>WrapperMetaForce (used for MOLLY)</td>
</tr>
<tr>
<td>frontend</td>
<td>OutputCache</td>
<td>OutputCache</td>
</tr>
<tr>
<td>frontend</td>
<td>OutputDCDTrajectory</td>
<td>OutputDCDTrajectory</td>
</tr>
<tr>
<td>frontend</td>
<td>OutputDihedrals</td>
<td>OutputDihedrals</td>
</tr>
<tr>
<td>frontend</td>
<td>OutputEnergy</td>
<td>OutputEnergy</td>
</tr>
<tr>
<td>frontend</td>
<td>OutputMomentum</td>
<td>OutputMomentum</td>
</tr>
<tr>
<td>frontend</td>
<td>OutputPDBFramePos</td>
<td>OutputPDBFramePos</td>
</tr>
<tr>
<td>frontend</td>
<td>OutputScreen</td>
<td>OutputScreen</td>
</tr>
<tr>
<td>frontend</td>
<td>OutputTemperatures</td>
<td>OutputTemperatures</td>
</tr>
<tr>
<td>frontend</td>
<td>OutputXYZTrajectory</td>
<td>OutputXYZTrajectory</td>
</tr>
<tr>
<td>integrator</td>
<td>BBKIntegrator</td>
<td>BBK</td>
</tr>
<tr>
<td>integrator</td>
<td>BSplineMOLLYIntegrator</td>
<td>BSplineMolly</td>
</tr>
<tr>
<td>integrator</td>
<td>EquilibriumMOLLYIntegrator</td>
<td>EquilibriumMolly</td>
</tr>
<tr>
<td>integrator</td>
<td>HMCIntegrator</td>
<td>HMC</td>
</tr>
<tr>
<td>integrator</td>
<td>ImpulseIntegrator</td>
<td>Impulse</td>
</tr>
<tr>
<td>integrator</td>
<td>LangevinImpulseIntegrator</td>
<td>LangevinImpulse</td>
</tr>
<tr>
<td>integrator</td>
<td>LeapfrogIntegrator</td>
<td>Leapfrog</td>
</tr>
<tr>
<td>integrator</td>
<td>MTSIntegrator</td>
<td>MTSIntegrator</td>
</tr>
<tr>
<td>integrator</td>
<td>NormModeCoarse</td>
<td>NormModeCoarse</td>
</tr>
<tr>
<td>integrator</td>
<td>NormModeInt</td>
<td>NormModeInt</td>
</tr>
<tr>
<td>integrator</td>
<td>NormModeMin</td>
<td>NormModeMin</td>
</tr>
<tr>
<td>integrator</td>
<td>NoseNVTLeapfrogIntegrator</td>
<td>NoseHoover</td>
</tr>
<tr>
<td>integrator</td>
<td>NPTVerletIntegrator</td>
<td>NPTVerlet</td>
</tr>
<tr>
<td>integrator</td>
<td>NVTVerletIntegrator</td>
<td>NVTVerlet</td>
</tr>
<tr>
<td>integrator</td>
<td>PLeapfrogIntegrator</td>
<td>PositionVerlet</td>
</tr>
<tr>
<td>integrator</td>
<td>DMDLeapfrogIntegrator</td>
<td>SelfconsistentLeapfrog</td>
</tr>
<tr>
<td>integrator</td>
<td>ShadowHMCIntegrator</td>
<td>ShadowHMC</td>
</tr>
<tr>
<td>integrator</td>
<td>STSIntegrator</td>
<td>STSIntegrator</td>
</tr>
<tr>
<td>integrator</td>
<td>UmbrellaIntegrator</td>
<td>Umbrella</td>
</tr>
<tr>
<td>io</td>
<td>PARReader</td>
<td>PARReader</td>
</tr>
<tr>
<td>io</td>
<td>PARWriter</td>
<td>PARWriter</td>
</tr>
<tr>
<td>io</td>
<td>PDBReader</td>
<td>PDBReader</td>
</tr>
<tr>
<td>io</td>
<td>PDBWriter</td>
<td>PDBWriter</td>
</tr>
<tr>
<td>io</td>
<td>PSFReader</td>
<td>PSFReader</td>
</tr>
<tr>
<td>io</td>
<td>PSFWriter</td>
<td>PSFWriter</td>
</tr>
<tr>
<td>topology</td>
<td>Topology</td>
<td>Topology</td>
</tr>
<tr>
<td>topology</td>
<td>topologyutilities</td>
<td>topologyutilities</td>
</tr>
</tbody>
</table>


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


[84] K. Hinsen. Python modules useful in computational chemistry. Available at http://ccl.net/cca/software/SOURCES/PYTHON/HINSEN.


315


dinosaur.compilertools.net/lex/index.html.


[119] J. Lewis. Fate maps and the pattern of cell division: a calculation for the

explicit finite difference method for 2-dimensional reaction-diffusion equations.


pigment patterns on the leopard and the jaguar. Phys. Rev. E, 74:011914–1–8,
2006.

[123] Q. Ma and J. A. Izaguirre. Long time step molecular dynamics using targeted

[124] Q. Ma and J. A. Izaguirre. Targeted mollified impulse — a multiscale stochastic
integrator for long molecular dynamics simulations. Multiscale Model.

[125] L. Maragliano and E. Vanden-Eijnden. On-the-fly string method for minimum


[129] F. M. Marée and P. Hogeweg. How amoeboids self-organize into a fruiting


