glsim: A general library for numerical simulation

Tomás S. Grigera [1]

Instituto de Investigaciones Fisicoquímicas Teóricas y Aplicadas (INIFTA) and Departamento de Física, Facultad de Ciencias Exactas, Universidad Nacional de La Plata and CONICET La Plata, Consejo Nacional de Investigaciones Científicas y Técnicas, Argentina

A R T I C L E   I N F O

A B S T R A C T

We describe glsim, a C++ library designed to provide routines to perform basic housekeeping tasks common to a very wide range of simulation programs, such as reading simulation parameters or reading and writing self-describing binary files with simulation data. The design also provides a framework to add features to the library while preserving its structure and interfaces.

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

Numerical computation or numerical processing of data with digital computers is an ever-increasing part of day-to-day scientific work, from statistical analysis of experimental data to numerical solution of systems of nonlinear integral or differential equations, to large scale simulations of many-particle systems (from electron gases to galaxies to social systems [1]). Many packages and libraries, both commercial and open-source, exist that can perform, or aid in performing, a wide range of more or less specialized tasks. However, due to the very nature of scientific endeavor, existing software is not always useful for the task at hand. New problems are studied which require variations or combinations of known techniques, or new techniques or solutions are developed for old problems. Thus the practicing scientist often finds him or herself writing computer code.

One of the difficulties faced is that a useful working program requires much more code than the lines needed for the main algorithm, due to the need for user interaction, data input/output, and possibly data preprocessing or format conversion. For example, the code for the LBFGS algorithm for minimization of a function of many variables [2] amounts to about 500 FORTRAN lines, excluding comments.[2] A fully working program to find potential energy minima of a particle system based on LBFGS put together by the author included an additional 300 lines for user interaction and interfacing between the energy routines and LBFGS plus 300 for input/output of simulation trajectory files, plus the routines for evaluation of the energy. This additional code is "clerical": it expresses relatively simple tasks and straightforward algorithms, and

generally takes a small fraction of execution time. But it must be written, debugged and maintained alongside the core of the program. When flexibility is added to the program, the clerical code typically grows quickly. Maintenance and debugging effort grows quickly with size unless the code is well structured [4], but well structured code requires thought and planning. Either way, clerical code ends up requiring a fair amount of attention.

This issue can be more or less sidestepped by writing very rigid, user-unfriendly software (e.g. Monte Carlo programs needing recompilation to change the temperature), which are highly unlikely to be useful to anyone but the original author in the original situation. Though such disposable software may sometimes make sense, most of the time a little more foresight is desirable. Especially because if the program is disposable, so tend to be the data it produces: ad-hoc file formats difficult or impossible to read without access to the source code that created them.

Ideally, the scientist needing to program some new algorithm should be able concentrate in writing and debugging the code for the main algorithm (on which he/she is an expert), while resorting to some sort of library for the clerical (but indispensable) tasks which are not part of the main algorithm, and likely not within the scientist's main expertise.

But can such a general library be developed, or even defined? Not if the "algorithm" and the "clerical" tasks remain so vaguely described. But we show below that one can define a simulation program very generally yet precisely, in a way that allows to identify the basic administrative tasks such a program will need, and build a library to perform such tasks. We also describe a particular implementation of such library, called glsim, which is available for download under an open-source license (see Section 6).

It turns out, not surprisingly, that algorithms can be designed more generally the more abstractly the problem to be solved is defined. We thus begin (Section 2) giving an abstract definition of a simulation, and listing a series of features that should be included

---

in a good simulation program. This allows us to write an outline simulation algorithm (Section 2.3). After commenting briefly on the programming techniques most useful for designing a library of the kind we are after (Section 3), we describe the glsim library in Section 4. We conclude in Section 6.

2. Definition of a simulation

2.1. A simulation: an abstract view

Take a molecular dynamics or Monte Carlo simulation, or an optimization technique (conjugate gradient minimization, annealing, genetic algorithm), or an iterative solution of a system of differential equations. All of these have in common that they start from a set of numbers and "evolve" this set according to some rules. The full history of the evolution ("trajectory") may or not be interesting in itself, but this is not relevant. The point is that although very different in aims, these (and other) simulations can be described under a common scheme.

We shall define *simulation* quite generally as the repeated application of a transformation to a set of numbers. Let's define two spaces $\mathcal{X}$ and $\mathcal{E}$, which we can assume to be subsets of $\Re^n$. $\mathcal{X}$ is the *configuration space*, and a vector $\mathbf{x} \in \mathcal{X}$ is a *configuration*. $\mathcal{E}$ is the *environment space* and $\mathbf{e} \in \mathcal{E}$ is an *environment*.

To perform a *simulation step* means to apply the transformations

$$\mathbf{e}_{n+1} = \mathbf{E}(\mathbf{e}_n), \tag{1}$$

$$\mathbf{x}_{n+1} = \mathbf{X}(\mathbf{x}_n, \mathbf{e}_{n+1}). \tag{2}$$

The configurations and environments thus form an ordered sequence. We can define a *simulation time* $t(n)$ through any monotonically increasing function of the number of steps n. The separation into configuration and environment is somewhat arbitrary, but note that while $\mathbf{x}_{n+1}$ can depend on $\mathbf{e}_{n+1}$, $\mathbf{e}_{n+1}$ is always obtained independently of $\mathbf{x}_n$.

The ordered pair $(\mathbf{x}_n, \mathbf{e}_n)$ is the *state* of the simulation at step n (or time $t(n)$). To start the simulation, we must specify the initial state $(\mathbf{x}_0, \mathbf{e}_0)$. This state can be constructed from another real vector γ through

$$\mathbf{e}_0 = \mathbf{E}_0(\gamma), \tag{3}$$

$$\mathbf{x}_0 = \mathbf{X}_0(\gamma). \tag{4}$$

The components of γ are called *control parameters*.

As the simulation progresses, it may be useful or convenient to compute subsidiary quantities, called *observables*, along the simulation. These quantities depend only on the configuration, and their value is not used at all in computing the successive environment or configuration, so that their computation can be omitted without changing the final state. We note them $\mathbf{O}_i(\mathbf{x}_n)$. To define the observable, a number of parameters will in general be needed, and these could in principle also evolve, so there will be an environment associated with each observable. In practice it is often convenient to merge these environments with the main simulation environment, and we do so below; the important point is that the environment variables associated with the observables do not interact with the rest of the environment.

2.2. A good simulation program

A computer program that can iteratively apply the transformations $\mathbf{E}(\mathbf{e})$ and $\mathbf{X}(\mathbf{x})$ is a simulation program. To be deemed a good simulation program, it should fulfill a number of requirements (Fig. 1).

1. Algorithms of the highest quality
2. Bit-level run reproducibility
3. Invisible run splitting or joining
4. Full human-readable record of simulation conditions
5. Easy user control over simulation parameters
6. Safe early interruption before programmed number of steps
7. Easy continuation after early interruption
8. Minimization of losses due to hardware failure (checkpointing)
9. Ability to read files from earlier versions of the program
10. Easy code maintenance

Fig. 1. Requirements for a good simulation program.

Some comments on these requirements:

1. This is an obvious requirement, the fulfillment of which of course depends on the particular technique being coded. However, a general advise is to write oneself the algorithms on which one is an expert (or close to it), and borrow the rest. This means using good libraries. Many are available freely over the Internet (e.g. Boost [5], the GNU Scientific Library [6,7], the Netlib collection [3]).
2. Computers are good at doing exactly the same things when given the same data, so this is not very difficult to achieve. The point is to stress that the user must have a way to completely specify *all* initial conditions, some of which may not be apparent at first sight (like the internal state of the random number generator, for instance), so that a given final state can be reproduced bit-to-bit. This kind of reproducibility is on the other hand very difficult to achieve across architectures, but this is not very often a crucial need.
3. This (together with the previous requirement) is most useful in debugging or tracking the origin of anomalous behavior that might manifest itself under particular circumstances. This can be achieved by saving the final state as binary data using the machine's internal representation, avoiding conversions e.g. to ASCII decimal numbers.
4. The program must produce a human-readable log file with all relevant information to allow the reproduction of the run. This is easy but time-consuming to program. A way to automate the production of the log is desirable.
5. Programs that require recompilation to change control parameters are all too common. This is unacceptable because it means that part of the information required to reproduce a run is buried in the executable. Of course there is some common-sense imposed limit on what should be parametrizable in an algorithm, but quantities that are expected to be changed (for tuning or to cover a relevant domain of the parameter space) should be stored in a file. For easy user control, a text file with an intuitive syntax (.ini-like) is preferable. Terminal input is generally not a good idea, as programs with a long runtime are likely to be scheduled for remote or background execution.
6. If for some reason (like an unexpected need to shutdown the machine) the simulation must be interrupted before completion, it should be possible to tell the program to save its internal state and terminate. This requires some means of communicating with a process which might not be associated with a terminal, such as Unix signals.
7. On launching the program again after an early interruption, it should automatically recognize a partially completed run and pick up from where it left.
8. The above saving of the internal state could be performed automatically every few hours, so that the simulation can resume with minimal loss of CPU time after hardware failure or power outage.

9. When the algorithm is improved or new features are added, it often becomes necessary to incorporate new data into the state files. These data will not be available when starting a simulation from a state written by an earlier version of the program, but the new version should be able to read the older files and supply appropriate default values for the missing data. This needs the use of files with self-describing structure.

10. The code should be organized in a way that it is easy to understand, debug and extend, using a modular design. Good documentation of source code is essential.

Clearly a good simulation program implementing the above features will require many lines of code apart from those implementing the specific algorithm of interest. Desirable though these features are for a program that will typically run for many hours, implementing them all is probably out of the question for a small (often one-man) team. Ideally, one would want a library that allows implementation of all these requirements automatically or with minimal effort, leaving the scientist-programmer to concentrate on point 1, specific to his/her problem.

2.3. A basic simulation algorithm

Taking into account our definitions and requirements, we are in position to write an outline simulation algorithm (Listing 1).

```
read γ
if partially completed run is found on disk then
    read n, eₙ, xₙ
else
    create x₀, e₀ (perhaps from a saved state)
    n = 0
endif

repeat
    n = n + 1
    compute eₙ = E(eₙ₋₁)
    compute xₙ = X(xₙ₋₁, eₙ)
    for all observables compute Oᵢ(xₙ, eₙ)
    write Oᵢ
    write log
until n = requested steps or early termination requested

write eₙ and xₙ
log termination
end
```

Listing 1. General simulation algorithm.

The fact that we have been able to say so much about a simulation without giving any details of the functions $\mathbf{X}(\mathbf{x})$ and $\mathbf{E}(\mathbf{e})$ might make us wonder how much of what we have said can be conveyed to a computer at this level of abstraction. So-called languages (OOLs) normally offer a set of features that allow us to code a simulation as so far defined in an actual programming language, compile the code and build it into a library. Obviously we will not have functional executable until $\mathbf{X}(\mathbf{x})$ and $\mathbf{E}(\mathbf{e})$ are completely specified and coded, but we shall be close to a "fill-in the blanks" situation where $\mathbf{X}(\mathbf{x})$ and $\mathbf{E}(\mathbf{e})$ can be just plugged in and a full-featured simulation program will result.

3. Design principles and programming techniques

A more or less obvious requirement for a useful library of the kind we are after is that it be built in modules with well-defined interfaces, and with the minimum possible interaction among them, so that they can be plugged in as necessary and combined in different ways. Note however that a powerful modularization is generally not one based on processes, or tasks to be performed on some data, but rather one that represents the division of the problem in abstract parts [8]. These abstract parts will necessarily embody procedures *and* data: for example, in glsim

there is a module for the concept of environment, which holds data associated with the environment as well as the procedures to read and write that data to a file, among others. The interface of a module (how it is seen from the outside) should reflect the abstraction, while the implementation details (i.e. the design decisions) are kept within the module. Modules so built are likely to be useful in a wider range of situations, and keeping design decisions local allows implementation improvements to be easily integrated into existing code. Thus modules can be characterized as keeping to themselves one or more design decisions that they hide from the others. This is known as *information hiding* [8].

The bundling together of data of different types and procedures operating on these data is called *encapsulation* [9]. If several instances of the data so aggregated can be (easily) created by the user, these instances are usually called *objects* (sometimes described as "intelligent data"). In OOLs, objects are variables of a user-defined type, thus the procedure of encapsulation in an OOL amounts to the creation of a new datatype. Languages that include syntactical support for encapsulation typically offer facilities to enforce information hiding to some degree, by allowing to make some data and procedures inaccessible from outside the module where they are defined. Information hiding requires encapsulation at the logical design level (even if not supported syntactically by the language), and perhaps for this reason the two terms are often used interchangeably [9].

Another requirement is that it be easily possible to refine, or *specialize* the modules provided by the library, adding to them new capabilities without breaking the interface. Adding data and procedures to an existing module or object "specializes" the object in the sense that to add capabilities one must typically make more assumptions on how the object will be used and/or impose restrictions on the operations allowed: the specialized object has additional, more specific properties [10] and thus represents a less general concept. In OOLs, it is generally convenient to implement specialization through the mechanism of *inheritance*, which is the possibility to define a datatype based on an existing datatype, only defining explicitly the properties desired for the former that differ from those of the latter [10]. Note however that inheritance and specialization are not isomorphic concepts, and that there are uses of inheritance other than specialization [10].

A bit more subtly, our insistence on building the library as far as possible around abstract concepts requires in turn to be able to write algorithms abstractly, or generically, in the sense that it must be possible to include in the library modules using undefined procedures and/or operating on unspecified data types or data types not completely defined. For this glsim relies heavily on *polymorphism*.

Polymorphism [11] refers to the ability of handling different data types with a uniform interface, which can also be described as using a single name to call different functions, based on the types of the data to be passed to that function. This includes a wide variety of situations. *Ad-hoc* polymorphism [12] refers to the case where different data types are processed by calling different functions, given explicitly for each combination of the allowed types. This includes overloading (writing many functions of the same name but different argument types) and coercion (automatic conversion of some types to other types). Ad-hoc polymorphism is found to some degree in all common programming languages (for instance, one uses the symbol + for addition of two integers or two floating-point variables). For our goal of writing abstract algorithms, we need a language supporting *universal* polymorphism [12], which means that the same code, or code generated using the same rule, is used for all admissible types. One way to achieve this is writing functions where types are not specified but left as a parameter. This is called *parametric polymorphism* [12], or generic programming in OOL jargon. Another,

perhaps more powerful, possibility is *inclusion* (or *subtype*) polymorphism [12], afforded by the concept of inheritance: objects of datatype B derived (i.e. defined by inheritance) from datatype A can be thought as being of type B or type A. Thus code written to operate on objects of type A can also operate on objects of type B.

Polymorphism is what allows us to write and compile our algorithm: clearly $\mathbf{X}(\mathbf{x})$ is a name that refers to different (unknown at compile time) functions, which we can distinguish by looking at the type of the argument. We shall be using mostly inclusion polymorphism because, at least in the language of our implementation (C++), it allows to explicitly express the assumptions one is making about the type that the function will be handling. Put in another way, inheritance guarantees a minimum set of operations that can be performed on the object to be processed.

Substituting different functions for one name is something that has been possible even in very old languages, by combining different modules at the linking stage. This can perhaps be thought of as a rather primitive substitute for polymorphism, as one could compile a module coding the basic algorithm and substitute the appropriate $\mathbf{X}(\mathbf{x})$ at link time. Indeed, the design of glsim profits from experience gained in developing a modular system written in C using such a scheme. However, this is not polymorphism, since the function is not selected based on argument type, but arbitrarily, outside the language itself. It thus cannot benefit from language features such as type checking. Also, static binding (i.e. mapping names to functions at compile or link time) has limitations, since it is not always possible to know at compile- or link time which function it is desired to call (which can be alternatively expressed by stating that the type of the argument is not always known at compile time). For instance, when linking a simulation program one typically wants to include only one of the possible $\mathbf{X}(\mathbf{x})$ functions. However, the situation is different with observables. Similarly to the simulation algorithm, it is best to write some generic code for the observation only once (see Section 4.4) and leave to the user just the task of writing code for the specific quantity required, by supplying some function $\mathbf{O}(\ldots)$. Since one may want more than one observable computed during the same simulation, the generic code will need to call different $\mathbf{O}(\ldots)$ functions at different times in the same point of the program. This requires *dynamic binding*, or the ability to select the appropriate function at run time. Dynamic binding is necessary to exploit the full power of inclusion polymorphism.

3.1. Object-oriented programming and C++

So-called languages (OOLs) provide syntax constructs that allow to easily express the techniques mentioned above. "Easily" means that concepts such as inheritance (which could conceivably be used in a program written in, say, C) can be expressed in the syntax of the language and thus more conveniently, with less work on the part of the programmer, and in a way such that the compiler "understands" what the programmer is trying to do and can help with compile time checks and diagnostics (e.g. type checking).

glsim is written in C++ [13], which is a standardized language with static typing that supports encapsulation, inheritance and polymorphism, and for which compilers are available in a wide variety of platforms. In C++ encapsulation is achieved by defining *classes*, which are user-defined datatypes. The procedures bundled with the data within a class are called *methods* or *member functions*, and data and procedures can be classified as public or private, in which case are inaccessible from functions defined outside the scope of the class. Instances of a class are called *objects*. Polymorphism is supported through explicit overloading (ad-hoc), *templates* (parametric), and inheritance (inclusion). Binding is static by default, but dynamic binding can be requested for specific methods by declaring them *virtual*.

When a method is virtual, C++ allows the programmer to declare it but leave it undefined (i.e. not coded). These methods are called *pure virtuals*. Classes with pure virtual methods cannot be instantiated, because the compiler would not know what to do when these methods are called. They can only be specialized by defining classes derived from them (at some point the pure virtuals will be defined and it will be possible to create instances of those derived classes). For this reason they are called *abstract base classes*, or ABCs. Classes like Simulation and Configuration (see Fig. 3 below) are ABCs in glsim. ABCs are useful for interface specification.

glsim can be said to use object-oriented programming (OOP) to the extent that it uses the techniques we have described [13,14]. However, OOP is sometimes described as a way to match "real-world" objects to software entities in a way that allows more convenient manipulation of them for computing purposes. The author's experience suggests that this view may be misleading, or too narrow, as a class hierarchy design tends to be more useful when built around rather abstract concepts, which may be hard to trace to "real world" objects. Also, the language may force or induce the programmer to define classes to benefit from features such as encapsulation, resulting in objects that may not be the most intuitive. For example, glsim defines a class for the simulation. It is hard to make the case that the simulation itself is a "real world" entity. However the simulation class turns out to be a good programming solution that allows to use C++ support for dynamic binding.

In summary, we simply claim that glsim is written making use of encapsulation and polymorphism as techniques for dealing with the present problem in an abstract way, and is thus written in a widely-available language with good support for them.

A final word about language choice. C++ is often criticised as being slow (although the criticism is contested [13]). It might thus seem a poor choice for a simulation package. Without entering the speed discussion, let us simply point out that the clerical code glsim mainly deals with is not performance-critical. Most of the CPU time will likely be spent computing the transformation $\mathbf{X}(\mathbf{x})$ (think of the force loop of molecular dynamics, for instance). If the programmer deems C++ too slow for the core part of the simulation, he is free to chose any other language; interfacing with glsim will still be possible in reasonable platforms. While glsim strives of course to be as efficient and fast as possible, it is clear that the (likely small) performance penalty introduced will be more than offset by the savings in expert human time required to produce a good simulation program.

3.2. Templates vs. virtual functions

As we said, glsim relies heavily on universal polymorphism. In C++ this means using templates (parametric polymorphism) or virtual functions (subtype polymorphism). The advantages of templates are that they produce slightly smaller objects (because objects with virtual functions need to store a table of virtuals, the so-called vtable), and that they result in faster code, because virtual functions, being resolved at run-time, are called via an indirection. In particular, virtual functions cannot be inlined.

Polymorphism through templates should thus be preferred where speed is critical. If the function implementing the transformation $\mathbf{X}(\mathbf{x})$ relies on polymorphism, it probably should use templates. glsim however uses virtual functions because when implementing non-performance-critical tasks polymorphism through virtual functions has the following advantages over templates:

1. It allows explicit, compiler-checkable interface specification: by writing ABCs, the pure virtuals make explicitly obvious what functions the user of the class expect to find implemented.

2. It allows for dynamic polymorphism: all objects of a derived type can be accessed through pointers to the base type. One can thus make containers that hold objects of different type. This feature is used in glsim for instance to deal with observables: all observables descend from **class** Observable and are accessed from the simulation class through a list of pointers to the base class (Section 4.4). This does not work with templates because instantiating a template creates a completely different type.

3. Virtual functions stay virtual for ever down the class hierarchy: one can specialize a class by deriving and defining or overriding the virtuals in the base, then further specialize the second class by deriving again and overriding only some of the virtuals. This is not easily and conveniently done with templates.

3.3. Literate programming

We have said that we would like a program as easy as possible to maintain and debug, and that this requires among other things a good documentation of the source code. In an attempt to achieve good and up-to-date source-code documentation, glsim is written using the *literate programming* style of programming [15]. The idea is to write the program and documentation simultaneously, shifting the focus from instructing a computer what to do to explaining to a human being what we want the computer to do [15]. The result should be a sort of "essay" that combines code and documentation.

In practice, the programmer writes a file containing documentation chunks written in a text-formatting language (in our case LaTeX) and code chunks written in some programming language (C++ in our case). A literate-programming tool is needed that on one hand extracts the source code and prepares a file suitable for the compiler, and on the other adds the necessary formatting commands to produce a LaTeX source file. glsim uses noweb, a freely-available literate programming tool [16]. After processing with LaTeX, the result is a document (Fig. 2) that reads roughly like the description given below, except of course that all details and the full source are included.

Registration. Built-in types can be registered directly through the appropriate `reg_var()` method. Registration requires a name, an address, and a `pa::kind` value that specifies whether the variable should be read from the file when `load()` is called. This behaviour can be overridden through global setting `read_from_file_always()`. This is useful for checkpointing (see how this feature is used in `environment`).

The `reg_var()` methods return a `const Iobase*` pointer to be used as a handle to the registered variable, which is required for unregistering.
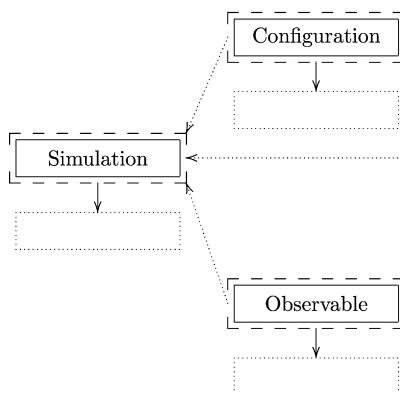
```
46a   ⟨registration methods 46a⟩≡
      const Iobase* reg_var(const std::string& name,bool*,
                            pa::kind rwsetting=pa::read_write);
      const Iobase* reg_var(const std::string& name,int*,
                            pa::kind rwsetting=pa::read_write);
      const Iobase* reg_var(const std::string& name,long*,
                            pa::kind rwsetting=pa::read_write);
      const Iobase* reg_var(const std::string& name,float*,
                            pa::kind rwsetting=pa::read_write);
      const Iobase* reg_var(const std::string& name,double*,
                            pa::kind rwsetting=pa::read_write);
      const Iobase* reg_var(const std::string& name,std::string&,
                            pa::kind rwsetting=pa::read_write);
```

This definition is continued in chunks 46b and 47a.
This code is used in chunk 45.

Untyped fixed- and variable- length buffers can be registered with the next two functions. The third is used internally by all version of `reg_var`, and can be called directly when registering an oject through a class derived from `Iobase`. In this case, note that `Persist_aid` takes ownership of the `Iobase*` object.

```
46b   ⟨registration methods 46a⟩+≡
      const Iobase* reg_var(const std::string& name,void*,int,
                            pa::kind rwsetting=pa::read_write);
      const Iobase* reg_var(const std::string& name,void** int,
```

Fig. 2. The documented source after processing with noweb and LaTeX.

---

4. An overview of the `glsim` library

Let us give an overview of the library from the point of view of the user. We shall leave out many details and show some parts as pseudo-code, so that to actually write code using the library it will be necessary to read the documentation accompanying the package. However the present description should give the reader an idea of the internal organization, and of what the user can expect from glsim.

glsim uses classes to represent the concepts of configuration, environment and observable introduced in Section 2. It also turns out to be useful to introduce a simulation class implementing our main algorithm (Listing 1) in one of its methods. This method is virtual so that it can be eventually overridden. With the exception of **class** Environment, these classes are abstract, because they include at least one pure virtual function. A few additional classes are defined to read the configuration file (**class** Parameters) and to automate the production of self-describing files (**class** Persist_aid). Fig. 3 gives an overview of the class structure.

If the user wants to write, say, a Monte Carlo simulation of the Ising model, (s)he would inherit from Configuration to declare an appropriate spin lattice, inherit from Simulation to write the Metropolis sweep, and from Environment to add the necessary parameters, such as temperature and coupling, and optionally
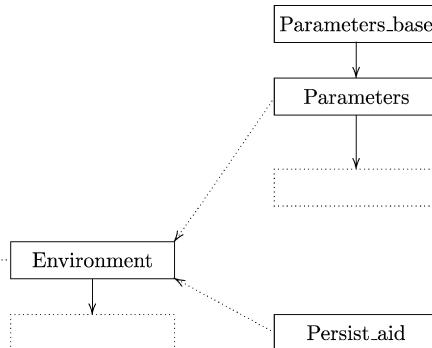

Fig. 3. Class hierarchy diagram. Full arrows indicate inheritance and dotted arrows composition (the class pointed by the arrow includes a pointer or reference to an instance of the other class). Dashed borders indicate abstract classes; empty dotted boxes represent classes the user should define, inheriting from the library as shown, in order to produce a functional simulation program.

define one or more observables inheriting from **class** Observable. To run the simulation, the main program then simply creates the objects, tells the configuration and environment to initialize themselves from disk and calls the Simulation run method. On completion of the run, the configuration and environment save methods are called. main() would read something like the following:

```
extern Parameters *create_parameters();
extern Environment *create_environment(int
    argc,char *argv[],Parameters *par);
extern Configuration *create_configuration(Environment *env);
extern Simulation *create_simulation(Environment*,Configuration*);

int main(int argc,char *argv[])
{
  Parameters *par=create_parameters();
  Environment *env=create_environment(argc,argv,par);
  Configuration *conf=create_configuration(env);
  load configuration;
  Simulation *sim=create_simulation(env,conf);
  load environment;
  steps=sim->run();
  save environment and configuration;
  delete sim;
  delete conf;
  delete env;
  delete par;
}
```

Listing 2. Simulation main().

This main (with the addition of exception catching and timing functions) is included in glsim, since its structure should not need alteration, and since the objects should be created in the order shown (in particular, the environment must be loaded after the simulation has been created, because the environment reads all variables registered for automatic saving, as discussed below). Different simulations can simply link this main with the appropriate create_xxx() functions.

4.1. Simulation

The Simulation class implements our abstract simulation algorithm (Listing 1). A Simulation object is created by passing pointers to suitable Configuration and Environment objects. The required observables are created (defining objects from a separate hierarchy described below, Section 4.4) and registered by calling add_obs(). This guarantees that the simulation is aware of them, and that the methods to compute the observables will be called when appropriate. Finally, the public run method is called to run the simulation as shown above. To produce a working simulation, the user must inherit from **class** Simulation and define the functions init_sim(), step(), and the logging functions.

```
class Simulation {
public:
  Simulation(Environment &e,Configuration &c);
  virtual const char *name()=0;
  virtual long run();
  void add_obs(observer* o);

protected:
  virtual ~Simulation();
  virtual void step()=0;
  logging functions;
  (…)
  virtual void start_observation();
  void obs();
  (…)
} ;
```

Listing 3. Declaration of class Simulation (excerpt).

The private part holds references to the environment and configuration objects (Sections 4.2 and 4.3) and a list of pointers to the observables. We have omitted the declarations and functions to install a signal handler for the Unix termination signals. The handler simply sets the termination_requested flag and returns, so that the current step is completed. The main simulation loop, below, checks this flag and stops the simulation if set. In this way the simulation can be safely interrupted with the kill command, or with ctrl+C if running interactively.

```
volatile sig_atomic_t Simulation::termination_requested=0;

long Simulation::run()
{
  install signal handler;
  check_for_partial_run(); /* This functions sets rmode */
  init_sim(rmode);
  if (env.obs_step>0) start_observation();
  log_start_run();

  env.completed_run=false;
  long steps_completed=env.requested_steps-env.steps_remaining;
  long actual_steps=0;

  /* Simulation loop; if a signal is received the handler will set
     termination_requested to 1 */
  while (env.steps_remaining>0 && termination_requested==0) {
    env.step();
    step();
    if (env.total_steps%env.log_step==0) log();
    if (env.obs_step>0 && env.total_steps%env.obs_step==0) obs();
    env.steps_remaining--;
    actual_steps++;
  }

  steps_completed+=actual_steps;
  if (termination_requested>0)
    std::cout << "\nWARNING: Terminating on signal " << signal_received <<
      "\nCompleted" << steps_completed << " steps.\n\n";
  else
    env.completed_run=true;
  log_stop_run();
  return actual_steps;
}
```

Listing 4. Simulation run method.

Observables are handled by keeping a list (structure from the library (STL)) of pointers to them, std::Observable* observables. **class** Observable (Section 4.4) provides methods for initialization and observation (i.e. computation and saving of the desired quantities), so that Simulation::obs() simply goes through the list, calling the appropriate method for each observable.

4.2. Configuration

A configuration is required to have a name, to know how to load and save itself to disk, and to be able to initialize itself to some default (say, a random but valid configuration).

```
class Configuration {
public:
  std::string name;
  Configuration(const std::string& name_) : name(name_) {}
  virtual ~Configuration() {}
  virtual void deflt()=0;
  virtual void load(const char*)=0;
  virtual void save(const char*)=0;
} ;
```

Listing 5. Declaration of class Configuration.

A working (i.e. instantiable) configuration class must define the three pure virtuals above plus the appropriate access interface, through which the Simulation::step() method will update it. It is important to keep it light, since the configuration will be accessed and updated many thousands of times during a run. In many cases,

public data members are probably the best alternative. See Section 5 for an example.

If desired, loading and saving in self-describing files can be done using **class** Persist_aid below. However, configurations must be saved in files physically distinct from environment files.

4.3. Environment

The environment holds all the data relevant to the simulation which is not reasonable to include in the configuration, including the number of steps completed and requested for the run, and filenames to read and save environment and configuration. An environment object can be created passing it the names of those files, or alternatively the argc and argv arguments of function main plus a reference to an object of **class** Parameters (Section 4.3.1). In this last way, it will parse the command line and initialize itself from a control file.

The declaration of **class** Environment is shown below (many variables omitted for brevity). Since the data are typically to be manipulated from outside the class, public data access has been preferred over get/set methods.

```
class Environment {
public:
  std::string title;
  int requested_steps;
  int log_step;
  std::string configuration_file_ini,configuration_file_fin;
  long total_steps;
  (...)

  Environment(int argc,char *argv[],Parameters &param);
  virtual ˜Environment() {}
  virtual void step();
  void load();
  void load_all();
  void save();

protected:
  Persist_aid persist;
  (...)
} ;
```

Listing 6. Declaration of class Environment (excerpt).

The environment is updated (i.e. the action of the function **E**(**e**) of Eq. (1) is performed) by calling step() (which is done from Simulation::run()). At this level the only action required is to increment the number of steps (total_steps), but more complicated things can be done by overriding this virtual.

The i/o methods are load() and save(), for normal reading/ writing to the environment file, and load_all(), which is the load function to be called when continuing a previously interrupted run (see discussion in Section 4.3.2). Variables are read and written through a Persist_aid object (Section 4.3.2). **class** Persist_aid does i/o to a self-describing binary file, so that variables are read by name. In this way it is possible to read old versions of environment files, because when variables are missing, a warning is printed and a default value (typically set from the configuration file) is used.

Persist_aid works through a simple registration mechanism as illustrated in the constructor below, making it easy to incorporate to the environment file variables defined by the user. This is done inheriting from Environment. If the new variables are to be initialized from the control file, a class derived from **class** Parameters (Section 4.3.1) is first defined which declares the necessary variables to the parameter file parser. This object is passed to Environment's derived constructor, which reads the parameter file values calling Parameters::value(), and registers the variables to be saved with the Persist_aid object.

```
Environment::Environment(int argc,char *argv[],Parameters &param) :
  ignore_partial_run(false),
  completed_run(false),
  obs_step(0),
  total_steps(0),
  par(&param)
  (...)
{
  // Parse the control file
  // N.B. this parses *all* defined variables; must not be called again
  by derived constructors
  par->parse(argc,argv);

  // Read values from control file
  ignore_partial_run=par->value("ignore-partial-run").as<bool>();
  title=par->value("title").as<std::string>();
  requested_steps=par->value("steps").as<int>();
  (...)
  register_vars();
}
(...)
void Environment::register_vars()
{ // Tell persist which variables must be saved in the environment file
  // (derived classes should register their own variables)
  persist.reg_var("environment.title",title,pa::write_only);
  persist.reg_var("environment.requested_steps",&requested_steps,
              pa::write_only);
  persist.reg_var("environment.total_steps",&total_steps);
  (...)
}
```

Listing 7. Environment constructor (excerpt).

4.3.1. Parameters

Simulation parameters are read form a control file with a straightforward ".ini" syntax (`variable=value`). Parsing is done with the `program_options` library, a part of Boost [5], which can also do command-line parsing. class Parameters_base provides a simple interface to Boost::program_options. Basically Parameters_ base defines an object, ctrl_file_options, through which configuration-file parameters can be defined as shown below, and a method value(const std::string &parameter) which returns the value of the requested parameter as a Boost::program_options::variable_value object (see the Environment constructor in Listing 7 for sample usage and the Boost documentation [5] for details).

To define parameters, the user inherits from **class** Parameters:

```
class Parameters : public Parameters_base {
public:
  Parameters();
protected:
  void parse_command_line(int argc,char *argv[]);
  (...)
} ;
```

Listing 8. Class Parameters declaration (excerpt).

The parameters to be read must be defined before the file is parsed. The parser is called from parse_command_line, which in turn is called by the Environment constructor. The most convenient place to define the variables is in the constructor of the class derived from parameters, like it is done in **class** Parameters itself:

```
Parameters::Parameters() : Parameters_base()
{
  ctrl_file_options.add_options()
    ("title",po::value<std::string>()->default_value("[untitled]"),
    "simulation title")
    ("steps",po::value<int>()->default_value(1),"number of steps to run")
    ("log_step",po::value<int>()->default_value(0),"write to log every ... steps")
  (...)
  ;
}
```

Parameters are defined by giving a name, a type, a default value and a description, using the syntax of the Boost::program_options library (ctrl_file_options is an object of type Boost::program_options::options_description).

Command-line parsing is done from parse_command_line, which is a protected virtual so that it can be overridden if needed. The version implemented in Parameters recognizes a command-line of the form

```
simprog [options] control_file initial_infix final_infix,
```

where control_file is the file with the control parameters and initial and final infix are used to build the input and output filenames according to the based on the patterns given in the control file (in the variables env_file, conf_file and obs_file). The special initial infix +++ is interpreted as a request to generate a default environment and configuration. The options -c and -e are also recognized, which allow to override the infix-generated configuration and environment files, respectively.

4.3.2. Persistence

Class Persist_aid is designed to easily implement our requirements 7, 8 and 9, namely to be able to read old versions of simulation files and to transparently resume execution after early interruption. The object to be placed under Persist_aid management is registered by calling reg_var. The user can then essentially forget about loading or saving: all registered variables are saved and loaded through Persist_aid::save() and Persist_aid::load(), which are typically called at the end or start of the simulation by Environment load or save methods. Say the user wants to save the temperature in the environment file to have it automatically restored on resuming the simulation, (s)he would simply do

```
double temperature;
persist.reg_var("my_environment.temperature",&temperature);
or
persist.reg_var("my_environment.temperature",&temperature,
    Persist_aid::write_only);
```

(the meaning of the second form is explained below). The "(scope-or-namespace).(variable-name)" convention is suggested to help keep variable names unique. If this is done in the constructor of a class inherited from Environment, then persist is the Persist_aid object defined in Environment. Its load and save methods are called when appropriate and no further action is needed. On the other hand, if it is desired to keep these data in a file separate from the global environment file, a different Persist_aid object needs to be created and its save and load methods called as needed.

The save method writes all the registered variables to a self-describing binary file (at this time managed through the NetCDF library [17]). The self-describing nature of the file means that on reading, variables are looked up by name (the name given on registering), rather than based on their position in the file. Thus if a new version of the program attempts to read a file produced by an earlier version which used to register fewer variables, the variables common to both versions will be retrieved by load() without problems. The user controls what happens when attempting to read a variable missing in the file by calling one of the methods on_absence_ignore, on_absence_warn or on_absence_throw: the missing variable is silently ignored, a warning is printed on standard output (in both cases the variable is not changed, so that if it was initialized to a reasonable default the simulation can proceed) or an exception is thrown.

There is an additional subtlety on reading: some variables (for instance, the number of steps completed so far) must be kept during a run, but once the run is finished and a new run is requested starting from the previously achieved state, they must be reinitialized to new values. In principle, they should not be saved with the rest of the environment. However, if the run is interrupted early, those values are needed to correctly resume the simulation when required. For this reason, a third argument can be given to reg_var, taking one of the values read_write (the default) or write_only. In the default reading mode (read_from_file_ad_hoc), write_only variable are not read. When resuming a run, Simulation calls Environment::load_all(), which temporarily sets the read mode to read_from_file_always, ensuring that even variables flagged as write_only are loaded.

All simple (built-in) types, plus C- and C++-style strings can be registered. It is also possible to register save and load functions requiring a pointer to a C or C++ file stream. In this case, the provided save function is called, the data is placed in a buffer and it is written as a single variable. This is mainly intended to allow for the use of third-party libraries that provide read/write methods of their own. Finally, notification functions can be registered which are called on i/o on a variable, so that for instance derived quantities can be recomputed when a variable is read from disk.

4.4. Observables

The final component is **class** Observable, intended as a base class for objects representing observables.

```
class Observable {
public:
    Observable(const std::string& name, Environment &e,Configuration &c,int st);
    virtual ˜Observable() {}
    virtual void start(Simulation::run_mode rmode);
    void observe();
    virtual void register_for_persistence(Persist_aid&);
protected:
    virtual void do_observation();
    (...)
} ;
```
Listing 9. Declaration of class Observable (excerpt).

To produce a working observable object, the user must write (apart from constructor and destructor), the methods start() and do_observation() and optionally register_for_persistence(). The first of these is passed a parameter telling it whether it should initialize for a new or a continuation run. It is expected that this function will open a file to record the observations, so this information is important. Typically, in a normal run the file is opened in overwrite mode and a header is written, while a continuation run requires opening in append mode, and the header is omitted. After initializing, the parent start() must be called. do_observation() must do the actual calculation of the observable, accessing the relevant data through the references to the environment and configuration stored in the object.

If required, variables can be registered with the persist object by overriding register_for_persistence(), but it must be remembered to call the corresponding method in the parent class.

4.5. Checkpointing and disk files

At present glsim still lacks support for checkpointing. The reason is that the existence of disk files to which information is added as the simulation proceeds (those produced by **class** Observable's descendants) make checkpointing a harder problem than continuation after interruption with a signal. It is fairly easy, using Unix alarm signals to make the program save the state (configuration and environment) periodically (say every two or three hours). However, if the system fails, the observable files will be out of synchronization, because the observable is typically written more often than the configuration (writing the configuration

after each step is not feasible because it is in general too expensive). Thus restarting after system failure requires a way to restore the observable files to the state they were in at the time the last configuration and environment were written. A convenient mechanism to do this is still missing, so at this point checkpointing with `glsim` is not easily achievable. Alternative ways to provide this mechanism are being considered, and it is expected that checkpointing support will be added in the near future.

5. Example use of `glsim`

Let us sketch how a user would proceed to write a working simulation program based on `glsim`. Assume one wants to implement a dynamics (MD) simulation of monoatomic particles.

First we need to decide how the state of the system (here the mechanical state of particles in 3-*d* space) will be represented. We then define a suitable class, inheriting from Configuration:

```
class MD_configuration : public Configuration {
public:
  int N;
  double time;
  double box_length[3];

  short *id;
  double (*r)[3];
  double (*v)[3];
  double (*a)[3];

  olconfig();
  ~olconfig();

  void load(const char* fname);
  void save(const char *fname);
  (...)
} ;
```

The MD algorithm will need additional parameters, such as the integration time step. These would be added inheriting from Parameters:

```
class MD_parameters : public Parameters {
public:
  MD_parameters() : Parameters()
  {
    ctrl_file_options.add_options()
    ("deltat",po::value<double>,"integration time step") ;
  }
} ;
```

These would be kept in an appropriate Environment descendant, together with additional information that makes sense to store, e.g. the energy:

```
class MD_environment : public Environment {
public:
  Environment(int argc,char *argv[],MD_parameters& par);
  double deltat,energy;
  (...)
} ;

MD_environment::MD_environment(int argc,char *argv[],MD_parameters& par) :
  Environment(arg,arv,par)
{
  deltat=par->value("deltat").as<double>();
  persist.reg_var("MD_environment.deltat",&deltat);
  persist.reg_var("MD_environment.energy",&energy);
  (...)
}
```

We now inherit from Simulation to define the simulation step and the appropriate initialization:

```
class MD_simulation : public Simulation {
public:
  MD_simulation(MD_environment&,MD_configuration&);
  void step();
private:
  MD_environment env;
  MD_configuration conf;
} ;

MD_simulation::MD_simulation(MD_environment& e,MD_configuration& c) :
  Simulation(e,c), env(e), conf(c)
{
  compute initial energy;
  substract center-of-mass motion;
  (...)
}

void MD_simulation::step()
{
  compute forces for configuration conf;
  perform Verlet step of conf;
}
```

Additional logging (e.g. periodically writing the energy) and observation (e.g. periodically saving the configuration to obtain a trajectory) can be added to Simulation and deriving from Observable. Finally, one writes the create_xx functions that create the configuration, environment, and simulation objects and return pointers to them (see Listing 2), e.g.:

```
Environment *create_environment(int argc,char *argv[],Parameters *par)
{
  return new MD_environment(argc,argv,*dynamic_cast<MD_parameters*>(par));
}
```

This code is then linked with `glsim` and the `glsim`-provided main to produce a working MD simulation. Based on the author's experience, it is estimated that this new code amounts to between 300 and 600 source lines, to be compared with 2200+ lines in `glsim` (providing command line parsing, configuration file parsing, self-describing environment files and orderly interruption through Unix signals). About half of the new code will be concerned with i/o of configurations.

5.1. Writing step() in another language

Although the definition of class MD_simulation must include the function step() (otherwise the class would remain abstract), it could be just a wrapper that calls routines in another language, if that is convenient. The details of building a mixed-language program depend on the language and platform (operating system, linker, compiler) and can be rather tedious [18]. However, C++ can be easily linked together with C and FORTRAN (in most platforms). Linking with C is directly supported by the standard through the **extern** "C" {...} construct. FORTRAN can be linked easily with the aid of the cfortran.h header [19], which supports a large number of compilers and linkers. To continue the above example, if one wishes to use FORTRAN routines to compute the forces and to perform the Verlet step, the above step() would look something like this:

```
#include "cfortran.h"

PROTOCCALLSFFUN3(FORCE,force,DOUBLEVV,DOUBLEVV,INTV)
PROTOCCALLSFFUN3(VERLET,verlet,DOUBLEVV,DOUBLEVV,DOUBLEVV)

#define force(r,a,t) CCALLSFFUN3(FORCE,force,DOUBLEVV,DOUBLEVV,INTV,r,a,t)
#define verlet(r,v,a)
CCALLSFFUN3(VERLET,verlet,DOUBLEVV,DOUBLEVV,DOUBLEVV,r,v,a)

void MD_simulation::step()
{
  force(conf.r,conf.a,conf.type);
  verlet(conf.r,conf.v,conf.a);
}
```

cfortran.h also supports calling C or C++ from FORTRAN. For details we refer to the cfortran.h documentation [19].

6. Final remarks and how to obtain `glsim`

We have described a library designed around an abstract definition of a simulation, understood in a very general way, and built using information hiding to provide convenient modules isolating implementation details from the user. It is expected that it will be useful to the developer of a program whose task can be described with the basic simulation algorithm, helping him/her to fulfill most or all of the requirements of a good simulation program with minimum effort.

`glsim` is being used in actual research projects, and is under development. Features (in particular checkpointing support) will be added in the future. Also, since the design is open to addition of more specialized modules, it is expected that the number of classes will grow with modules adding support for more specific simulations and for data analysis. It is hoped that the design of `glsim` will encourage its users to write modular, reusable code that can eventually contribute to the growth of `glsim`. The author has written code for manipulating off-lattice configurations and trajectories (along the lines sketched in Section 5). This code is useful for a wide variety of situations, including analysis programs outside the simulation itself, and it is planned to add this to `glsim` as soon as the interface is polished. Hopefully others will start using `glsim` to later become contributors as well.

The `glsim` source code is distributed under the GNU General Public License version 3. It can be downloaded at no cost from SourceForge (http://sourceforge.net/projects/glsim).

Acknowledgements

References

[1] S. Hartmann, The world as a process: Simulations in the natural and social sciences, in: R. Hegelsman, et al. (Eds.), Modelling and Simulation in the Social Sciences from the Philosophy of Science Point of View, in: Theory and Decision Library, Kluwer, Dordrecht, 1996, pp. 77–100.
[2] D.C. Liu, J. Nocedal, On the limited memory BFGS method for large scale optimization, Mathematical Programming 45 (1989) 503–528.
[3] Netlib, Netlib software repository, http://www.netlib.org, 1985.
[4] R.D. Banker, S.M. Datar, C.G. Kemerer, D. Zweig, Software complexity and maintenance costs, Communications of the ACM 36 (1993) 81–94.
[5] Boost, Boost C++ libraries, http://www.boost.org, 1999.
[6] GSL, GNU scientific library, http://www.gnu.org/software/gsl/, 1996.
[7] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, F. Rossi, GNU Scientific Library Reference Manual, 3rd edition, Network Theory, 2009.
[8] D.L. Parnas, On the criteria to be used to decompose a system into modules, Communications of the ACM 15 (1972) 1053–1058.
[9] E.V. Berard, Abstraction, encapsulation, and information hiding, in: [14].
[10] A. Taivalsaari, On the notion of inheritance, ACM Computing Surveys 28 (1996) 438–479.
[11] C. Strachey, Fundamental concepts in programming languages, Higher-Order and Symbolic Computation 13 (2000) 11–49.
[12] L. Cardelli, P. Wegner, On understanding types, data abstraction, and polymorphism, ACM Computing Surveys 17 (1985) 471–523.
[13] B. Stroustrup, The C++ Programming Language, 3rd edition, Addison–Wesley, 1997.
[14] E.V. Berard, Essays on Object-Oriented Software Engineering, Prentice–Hall, 1992.
[15] D.E. Knuth, Literate Programming, Stanford University Press, California, 1992.
[16] N. Ramsey, Literate programming simplified, IEEE Software 11 (1994) 97–105.
[17] Netcdf, NetCDF (network common data form) libraries http://www.unidata.ucar.edu/software/netcdf/, 2004.
[18] B.D. Burow, Mixed language programming, in: R. Shellard, T. Nguyen (Eds.), Computing in High Energy Physics (CHEP'95), World Scientific, 1995.
[19] B. Burow, cfortran.h, http://wwwasd.web.cern.ch/wwwasd/cernlib/cfortran.html, 1998.