

# C++ and Generic Programming for Rapid Development of Monte Carlo Simulations

Gregory Brown<sup>1,2</sup>, Hwee Kwan Lee<sup>3</sup>, and Thomas C. Schulthess<sup>1</sup>

<sup>1</sup> Center for Computational Science, Oak Ridge National Laboratory, Oak Ridge, TN 37831-6164, USA

<sup>2</sup> School of Computational Science and Information Technology, Florida State University, Tallahassee, FL 32306-4120, USA

<sup>3</sup> Department of Physics, Tokyo Metropolitan University, Tokyo 192-0397, Japan

**Summary.** Flexible coding techniques are important for enabling the solution of many problems with one application or the rapid construction of new applications through code reuse. Generic programming offers the high-performance computing community an excellent method for code reuse without loss of efficiency. Here, solutions to both problems from the  $\Psi$ -*Mag* toolset are presented. The class `EHModel` and its associated `EHTerms` can be used to calculate energies in a wide variety of spin models. An allied generic programming solution for applying Metropolis Monte Carlo to estimation of thermodynamic quantities is also given.

## 1.1 Introduction

In a wide variety of science and engineering topics, rapid advances in computational resources make possible calculations that were impossible only a few years earlier. Frequently, the codes performing these calculations have been developed by one or two scientists interested in the calculation. While such codes may enable reuse of an entire application through flexibility in input, these codes rarely reuse existing code fragments or even produce fragments suitable for later reuse. Achieving better efficiency often requires the efforts of several programmers, some skilled in computational science but not the science investigated through the calculation. In this environment, production of efficient codes requires development of flexible code fragments that can be reused quickly by people who did not develop them.

Here we discuss a generic programming approach to high-performance computing. In the definition popularized by Austern [1], generic programming is an extension of object-oriented programming that focuses on “the set of requirements” which a data type must fulfill to qualify as an implementation of a particular abstraction. The set of requirements defines a *concept*, and an implemented data type which meets those requirements is called a

*model* of the concept. A good example of a concept is **Container**, which is a collection of elements that can be iterated over. The addition of requirements leads to the *refinement* of one concept into another. **Container** can be refined, for example, into concepts such as **Sequence**, in which elements can only be accessed in some particular order, **Random-Access Container**, in which the elements can be indexed by an integer type, and **Associated Container**, in which elements can be indexed by an arbitrary type. Concepts such as these, along with algorithms such as **Count**, **Copy**, and **Sort** can be recognized as commonly recurring themes in computational science. The Standard Template Library (STL) defines many such concepts and algorithms. STL provides both the tools to simplify the implementation of high-performance codes and the inspiration to solve problems in a generic, problem-independent manner. The result is code that can easily be reused to create new applications.

To be useful for high-performance computing, the flexibility of generic programming must come without sacrificing efficiency or clarity. Using templates in C++ [2] is one programming paradigm in which this can be achieved. In fact, STL was first implemented as templated C++ classes [1]. Direct comparison has shown that, in general, modern C++ compilers generate executables with no performance penalty with respect to procedural languages such as FORTRAN. See, *e.g.*, Ref. [3]. Templates, in particular, defer the specification of types associated with a function or class until compilation, but at no cost to execution speed. For example, a templated sorting function can use a template parameter for the type being sorted in the same manner as subroutines use variable arguments. Whether the type being sorted is a floating point number or an integer is specified at compilation, but the resulting machine code is not affected by the fact that a templated function was used to generate it.

The  $\psi$ -Mag project [4] is an effort to foster a community approach to code sharing and reuse in computational materials science. One goal of the project is developing specifications for important concepts for computational materials science in general and computational magnetism in particular. These concepts are then translated into implemented C++ models. Here we discuss the effort within the  $\psi$ -Mag project that is concerned with the generic implementation of Monte Carlo methods [5], which can be used in the context of computational magnetism to quickly develop new applications.

The natural flexibility in scientific applications based on making as few assumptions as possible is important. A thorough discussion of this point is necessary for a proper comparison to generic programming. Applications built with such flexibility can simulate many different problems, often with the differences specified only in the input files. There are often significant trade-offs between flexibility and computational efficiency, and making a few assumptions can often greatly simplify an algorithm. Much of the art of scientific programming involves optimizing among flexibility, speed, and coding complexity.

The flexible classes in  $\psi$ -Mag that use object-oriented programming to describe the energy and fields associated with spin models are described

in Sec. 1.2. A set of generic-programming concepts for Metropolis-sampling Monte Carlo is outlined in Sec. 1.3. In that section, some implementation details particular to  $\psi$ -Mag are also discussed. Sec. 1.4 is a summary of this work.

## 1.2 Flexible Energy Calculation

Flexibility built on avoiding assumptions is not limited to a particular programming language and has been utilized throughout the history of computer programming. Object-oriented programming facilitates this flexibility through polymorphism [2], by which a set of types can be treated as equivalent even though the members of the set implement very different behaviors. An object-oriented approach, augmented by a trivial application of templates, is outlined here for the calculation of energies in spin models.

The collective behavior of spin systems, such as the Ising and Heisenberg models, has been studied extensively, often in the context of developing fundamental principles in statistical mechanics. See, *e.g.*, Ref. [6]. In part, this interest stems from the relative simplicity of the models. For example, the Zeeman energy of a system of spins in a field is

$$E_z = - \sum_{i=1}^N M_i \mathbf{H} \cdot \mathbf{s}_i, \quad (1.1)$$

where the sum is over all spins,  $M_i$  is the moment of the  $i$ -th spin,  $\mathbf{s}_i$  is the orientation of that spin, and  $\mathbf{H}$  is the applied field. The difference between the two models lies principally in the nature of  $\mathbf{s}_i$  and  $\mathbf{H}$ . For the Ising model  $\mathbf{s}_i \in \{-1, +1\}$ , but for the Heisenberg model  $\mathbf{s}_i \in \mathfrak{R}^d$  with the constraint  $|\mathbf{s}_i|=1$ . Likewise,  $\mathbf{H} \in \mathfrak{R}$  for the former and  $\mathbf{H} \in \mathfrak{R}^d$  for the latter. It is thus possible to create one flexible C++ class to handle the Ising model and the family of  $d$ -dimensional Heisenberg models if the class is templated on the `spin_type` and `field_type`. In fact, maximum flexibility is achieved if different containers are allowed. Therefore, templating on the type of container holding the orientations of the spins and the type of container holding fields provides maximum flexibility. The types of the individual spins and fields are then the `value_types` of the containers.

As an example of avoiding assumptions, consider exchange interactions. In most simple models, each spin interacts with a well-defined (often small) subset of the other spins. If  $J_{ij}$  is the interaction energy between the  $i$ -th and  $j$ -th spins, then the set of all interactions for a particular problem is  $\{J_{ij}\}$ , and the exchange energy is

$$E_x = -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N J_{ij} \mathbf{s}_i \cdot \mathbf{s}_j. \quad (1.2)$$

Frequently  $\{J_{ij}\}$  can be simply constructed, *e.g.* nearest-neighbor interactions on a cubic lattice. A function which uses such an assumption can calculate the energy using a geometry-dependent rule that calculates the neighbors  $\{j\}$  on the fly quite easily. However, each particular geometry will require a separate function. A more flexible approach is to explicitly store  $\{j\}$  and  $J_{ij}$  for each spin  $i$ , so that one function can calculate the energy for any geometry. All information about the dimension and symmetry of that lattice, as well as boundary conditions, are conveniently incorporated into the set  $\{j\}$  for which  $J_{ij}$  are nonzero, and any other variations in exchange strength into an explicit list of  $\{J_{ij}\}$ . Such flexibility greatly enhances code reuse-ability and decreases development time for new codes.

With these two contributions to the energy of spin models in mind, it is possible to define a generic-programming concept of **Energy**. First define **orient\_container** to be the type used to contain the orientations of an entire system of spins, **field\_type** to be the type used to represent the direction and strength of a field, and **field\_container** to be the type used to contain the field at the location of each spin. The required member functions are as follows.

```
Real energy(Real time, const orient_container& spin)
Calculates the total energy for the configuration of the system specified in spin.
```

```
Real energy(Real time, const orient_container& spin,
            size_type ispin)
Calculates the energy associated with the  $i$ -th spin for the system configuration specified in spin.
```

```
void field(Real time, const orient_container& spin,
           field_container& h)
Calculates the field affecting each spin in the system. The calculated fields are added to h. The form of the field affecting the  $j$ -th spin should be the negative of the functional derivative of the energy with respect to  $s_j$ , which enforces  $E(\mathbf{s}_i) = -\mathbf{h}_i \cdot \mathbf{s}_i$ .
```

```
void field(Real time, const orient_container& spin,
           size_type ispin, field_type& h)
Calculate the field affecting only the  $i$ -th spin. Add the calculated field to  $\mathbf{h}(\mathbf{s}_i)$ .
```

Here **Real** is a floating-point type, and **size\_type** is an integer type appropriate for indexing (it may be nonnegative). Note that the simulation time is made a parameter of the function call to allow for the possibility of, for

example, time-dependent applied fields  $\mathbf{H}(t)$ . Two method functions are required for calculating energies. The first one calculates the total energy of the system, while the second calculates only the energy associated with the  $i$ -th spin. The latter method is required for efficiency in algorithms that consider only one spin at a time, such as the local update Monte Carlo discussed in the next section. Note that summing the  $E(\mathbf{s}_i)$  of the latter function over all the spins in the system will not, in general, give the  $E$  of the former function. Double counting in  $E_x$ , which is related to the two-bodied nature of exchange, is a prime example of this fact. Methods for calculating the field at each spin site  $\mathbf{h}(\mathbf{s}_i)$  are also included in the concept definition. Some algorithms for highly efficient Monte Carlo sampling, such as overrelaxation [7], and numerical approaches like micromagnetics [8, 9] use this field. The inclusion of both quantities in the **Energy** concept is also intended to enforce the fundamental definition  $\mathbf{H}_i = -\delta E / \delta \mathbf{s}_i$ . The flexibility outlined here is maximized when the inclusion or exclusion of particular contributions is not determined until the code is executed. Outside the object-oriented paradigm, such flexibility has been mimicked through logical flags or by setting all interactions, such as  $\{J_{ij}\}$ , to zero. Using object-oriented polymorphism is much more efficient. The *Ψ-Mag* toolset contains a class `EHModel` which is a model of **Energy**, but sums over contributions calculated by other models of the **Energy** concept.

Since the purpose of `EHModel` is to accumulate different energy contributions, it is derived from a container of pointers to those contributions. See Fig. 1.1. Since all of the pointers must be of the same type, a base class `EHTerm`, which is a model of **Energy**, is the target of those pointers. Each particular energy contribution is constructed as a derived class which inherits from `EHTerm`. For this polymorphism to work properly, the member functions of `EHTerm` must be virtual functions. The key property of a virtual function is that it actually calls the equivalent function in the derived class. For `EHModel`, this means that the appropriate function will be called for each term. The flexibility inherent in this approach is that different energy contributions can be incorporated into `EHModel`'s container as the program is executed.

Templating is also required for `EHModel`, since different types may be employed to represent spins and fields in different models. Since virtual functional calls must be employed in calculating the particular energy contributions, and virtual member functions cannot be templated, the classes `EHModel`, `EHTerm`, and the derived classes must all be templated. The templating of so many classes is undesirable as it increases code-complexity, but in this situation it is unavoidable.

`EHModel` and its associated classes obviously allow any number of different contributions to the energy. In addition to the Zeeman and exchange energies defined above, common contributions are the uniaxial anisotropy

$$E_u = - \sum_{i=1}^N K_i \left( 1 - [\mathbf{s}_i \cdot \hat{\mathbf{n}}_i]^2 \right) \quad (1.3)$$

where  $\hat{\mathbf{n}}_i$  is the axis of anisotropy for the  $i$ -th spin, and  $K_i$  is the strength of that anisotropy, and dipole-dipole interactions

$$E_m = -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N M_i M_j \mathbf{s}_j \cdot \left( \frac{3\hat{\mathbf{r}}_{ij}\hat{\mathbf{r}}_{ij} - 1}{r_{ij}^3} \right) \cdot \mathbf{s}_i \quad (1.4)$$

with  $\mathbf{r}_{ij}$  the displacement vector between the  $i$ -th and  $j$ -th spins. Extensibility is easily achieved, because new derived classes can be created without modifying the existing code. In addition to energy contributions not mentioned here, this can be exploited to increase the generality or the specificity of the energy contributions. As an example of the former, the toolset already contains an `EHTerm`-derived class for which  $J_{ij}$  can be a tensor. This is appropriate for anisotropic exchange. As an example of the latter, a derived class could be created for which  $J_{ij}$  is the same for all nearest neighbors. The advantage of this kind of specificity is that it can enhance the efficiency of the program. Such general or specific classes can be added to the toolset as the need for each is identified.

While the particular class system associated with `EHModel` and `EHTerm` outlined in the previous section is itself quite flexible, the real power of generic programming is associated more with the `Energy` concept itself. A complete class system for lattice-gas simulations (`LGModel`) or for Lennard-Jones fluids (`LJFluid`) are also possible. With generic programming, algorithms and classes that use `EHModel`, `LGModel`, or `LJFluid` interchangeably are then possible.

### 1.3 Monte Carlo Concepts

By defining a set of generic-programming concepts associated with Monte Carlo sampling, it is possible to create a large set of models which become interchangeable tools and can be assembled quite easily into new Monte Carlo simulations. This section provides one such set of concept definitions.

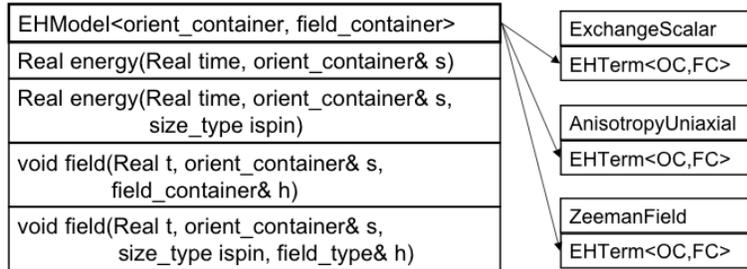
At its most fundamental, Monte Carlo is a technique for estimating the integral

$$I = \int g(x)f(x)dx \quad (1.5)$$

by randomly drawing  $N$  values of  $x$  with probability density function (pdf)  $f(x)$  and evaluating the mean

$$E = (1/N) \sum_{i=1}^N g(x_i). \quad (1.6)$$

In the limit of large  $N$ ,  $E \approx I$  [5]. Monte Carlo techniques usually outperform deterministic quadrature techniques when the dimensionality of  $x$  is large, as



**Fig. 1.1.** Schematic representation of the polymorphism used in `EModel`. `EModel` is a container of pointers to `ETerm`, and the various energy terms inherit from `ETerm`. Virtual function calls, with the same interface as those in `EModel`, enable `EModel` to access the method functions of the derived classes

is usually the case in statistical mechanics where one is interested in high-dimensional phase spaces.

Randomly drawing values of  $x$  from  $f(x)$  is, in general, not easy. Although it is possible to find simple algorithms for particular forms of  $f(x)$ , one often has to resort to importance sampling techniques. One important example is a technique due to Metropolis, *et al.*, [10] which produces a Markov chain of  $x_i$  distributed proportionally to  $f(x)$ . The Metropolis algorithm is based on the idea of *detailed balance*, specifically that the probability of making a transition from state  $x$  to  $x'$  in the Markov chain must be equal to the probability of making the transition from  $x'$  to  $x$ . In the Metropolis approach, see Ref. [5],

$$A(x'|x)T(x'|x)f(x) = A(x|x')T(x|x')f(x'), \tag{1.7}$$

where  $T(x'|x)$  is the probability of “proposing” the transition from  $x$  to  $x'$ ,  $A(x'|x)$  is the probability of accepting the transition, and  $f(x)$  is the desired pdf. This can be rearranged to give a constraint on  $A$ ,

$$A(x'|x) = \frac{T(x|x')f(x')}{T(x'|x)f(x)}A(x|x') \equiv q(x'|x)A(x|x'). \tag{1.8}$$

It is obvious from Eq. (1.8) that  $q(x'|x) = 1/q(x|x')$ . This constraint does not completely specify  $A$ . Two commonly used forms that satisfy Eq. (1.8) are the Metropolis acceptance probability  $A_M = \min(q, 1)$  and the Glauber acceptance probability  $A_G = q/(1+q)$ . In condensed-matter physics, the situation is often much simpler because the transitions in the Markov chain are chosen such that  $T(x'|x) = T(x|x')$ , then  $q = f(x')/f(x)$ .

For a generic approach to Metropolis sampling, several concepts need to be defined. The first concept covers Monte Carlo sampling in general, and Markov-chain importance sampling in particular. In the  $\Psi$ -Mag toolset, we define a class `MetropolisSampling` to implement a generic Monte Carlo concept, labeled `MCSample`, as follows.

```
Real mc_time(Real val)
```

This sets the “simulation time” in the sampling algorithm. The time is given in units of Monte Carlo steps (mcs). The simulation time is frequently needed. For instance, it is needed for measuring the correlation between the samples generated by the Metropolis method, and may be employed in algorithms such as simulated annealing.

```
Real mc_time() const
```

Returns the “simulation time” associated with the sampling object in units of mcs.

```
int operator()(RNG& urng, TransitionOp& op, Weight& f,
              System& x, int ntransition=1)
```

Performs `ntransition` Monte Carlo steps to generate a new configuration of the system to be used in Monte Carlo integration. The parameter `x` contains the initial configuration when the function is called and the new configuration upon return. Here a function object  $f(t, x)$  is used to determine the weight of configuration  $x$  at time  $t$ , with  $t$  the time (in mcs) supplied by the `MCSample` class. For example, for the canonical ensemble, the weight is  $\exp[-\beta E(t, x)]$ . The transitions associated with each mcs are accomplished via a model of `TransitionOp`, and random numbers are supplied by a model of `RNG`. The return value of the function is the number of accepted mcs, which can be used to calculate the acceptance rate.

Essentially `MCSample` is a function object that uses one function object as a uniform random number generator on  $[0, 1)$  for making choices and another function object for determining the weight  $f(t, x)$  of particular configurations. Note that the acceptance probability  $A$  is not passed as a function parameter. Instead it assumed to be an integral part of the sampling concept. In the C++ implementation of  $\Psi$ -Mag it is a template parameter. `MCSampling` also defines a nontrivial transition concept which can generate the sequences of

the Markov chain. The `MTransition` concept, with `element_type` the type used to represent the individual components of the system, is as follows.

```
Real step_size(Real val)
```

Set the size of change associated with each transition. The meaning of this parameter is highly dependent on the individual transitions being implemented.

```
Real step_size() const
```

Return the size of the change associated with each transition

```
size_type pick_dof(RNG& urng, Real time, const System& x)
```

Choose the index of a particular degree of freedom within  $x$ . This usually becomes the trial move.

```
Real operator()(RNG& urng, Real time, System& x, size_type idof)
```

Perform on  $x$  a transition associated with the  $i$ -th degree of freedom. The return value is the ratio  $T(x|x')/T(x'|x)$ , *i. e.* the ratio of the probability of attempting a move from  $x'$  to  $x$  to the probability of attempting a move the other way.

```
void undo(System& x)
```

Restores  $x$  to its configuration before the last move was made. This can only be done for the last transition, repeated calls are *not* guaranteed to move the system backwards through all configurations visited.

`MTransition` provides for selecting a degree of freedom (dof) within the system, performing a transition associated with that dof, and undoing that transition if it is not accepted.

Based on these concepts, the core of the generic single-site-update Metropolis sampling code looks like

```
int dof = op.pick_dof(urng,time,x);
Real fPrev = f(time,x,dof);
Real tRatio = op(urng,time,x,dof);
Real fNew = f(time,x,dof);
Real q = tRatio*fNew/fPrev;
if( urng() < A(q) )
    isuccess++;
else
    op.undo(x);
```

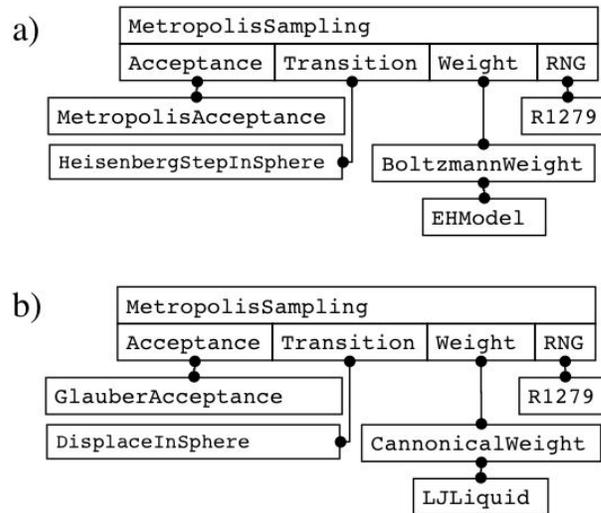
This algorithm should work as well for quadrature as it does for simulations of Ising models and compressible Lennard-Jones fluids. A cluster update algorithm may require a different implementation, but could be a model of `MCSample` and be able to easily use all of the concepts employed here.

In condensed-matter physics the weight function is usually a Boltzmann factor. In spin systems frequently  $f(t, x) = \exp[-\beta E(t, x)]$ , where  $E(t, x)$  is the energy of configuration  $x$  at time  $t$ , and  $\beta = (k_B T)^{-1}$  is the inverse temperature. For compressible fluids, however,  $f(x) = \exp(-\beta[E(x) + PV(t, x)])$ , where the pressure  $P$  and volume  $V(t, x)$  have to be introduced. Since the weight concept only calls for a function object that takes  $t$  and  $x$  as parameters, both possibilities are allowed for, but the weight concept is then refined into two cases. The incompressible refinement requires methods for setting and accessing the temperature  $1/\beta$  and uses a model of the `Energy` concept to calculate the energy associated with  $x$ . The compressible refinement requires setting and accessing methods for both  $1/\beta$  and  $P$ , and needs to supplement the `Energy` concept with a way to calculate  $V(t, x)$ . This can be very easily generalized to the grand canonical ensemble, and is a prime example of the power of generic programming.

The structure of the Boltzmann-weight functions is particularly simple. The various functions can be coded generically using the appropriate `Energy`, `Volume`, and `Number` concepts as template parameters which set the policies [11] appropriate to specific models. In fact, in this decomposition of the importance-sampling problem, all of the system-specific details have been pushed into the weight function. This is not a particularly large burden, care must simply be taken that the `System` passed to the Metropolis sampling object be compatible with the `Energy` model that is waiting for it at the other end of the function-call sequence. If the `System` and `Energy` objects are instantiated together in the application code, the potential for problems will be minimized.

To this point, the discussion has focused on the generic-programming concepts associated with Monte Carlo sampling. To have an actual application code, classes that are models of these concepts must be written, instantiated, and assembled. Schematic representations of two different applications are shown in Fig. 1.2. Figure 1.2(a) represents the simulation of a Heisenberg spin model. It indicates that `R1279`, a lagged-Fibonacci generator, is used for random numbers. The simulation is done in the canonical ensemble using a class `BoltzmannWeight` with `EHModel`, as defined in Sec. 1.2, used to actually calculate the energies. The transition model is a class that displaces a Heisenberg spin within some sphere around its current direction. The transition is a property of the simulation that can be expected to change most frequently. Finally, the Metropolis form of the acceptance probability  $A$  is used, in the `Psi-Mag` toolset we have found it convenient to make the form of  $A$  a template parameter of the `MetropolisSampling` class.

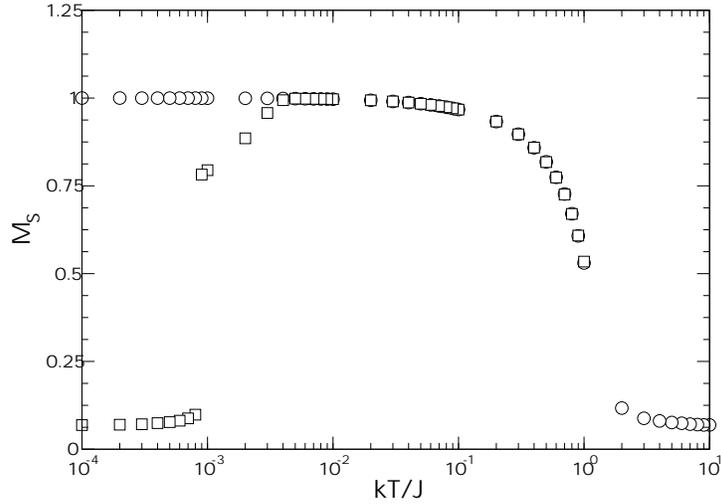
Figure 1.2(b) represents canonical-ensemble simulations on a Lennard-Jones fluid at constant pressure. Compared to part (a), the same random



**Fig. 1.2.** Schematic representation of the particular classes used in to different Monte Carlo simulations assembled from the  $\Psi$ -Mag toolset. (a) Simulation of a Heisenberg model. (b) Simulation of a Lennard-Jones liquid

number generator is used and a trivial change to the Glauber form of  $A$  has been made. However, this simulation requires a new class for calculating the weight  $f(x)$ , and `LJLiquid` must therefore be capable of calculating both  $E$  and  $V$ . Finally, since the rotation of a Heisenberg spin has no meaning in this context, a class which displaces particles within a sphere centered on the previous location has been used.

There is another indication of the power of generic programming techniques. The schematics shown in Fig. 1.2 were inspired by a graphical interface used for the Common Component Architecture (CCA) [12, 13]. CCA is based on the Scientific Interface Description Language (SIDL), and can be used to bind together code not originally written to interoperate; even code written in different languages. There is a very strong correlation between the SIDL and generic programming concepts. Using a simple wrapping method [14], it is possible to turn the Monte Carlo classes of the  $\Psi$ -Mag toolset into several components which can be combined easily in a graphical format similar to that shown in Fig. 1.2.



**Fig. 1.3.** Staggered magnetization,  $M_S$ , for a  $6 \times 6 \times 6$  Heisenberg model with  $K=0.01$  calculated at various temperatures using the scheme described here. The implementation using `MetropolisSampling` and `BoltzmannWeight` as the model for the `Weight` concept (squares) suffers from underflow at low temperatures, while an implementation using `MetropolisSamplingBoltzmann` and `EnergyWeight` for the model (circles) does not

The concepts developed here have been verified with actual simulations. The results of simulations of a  $6 \times 6 \times 6$  Heisenberg antiferromagnet with  $K_i=0.01$ ,  $\hat{\mathbf{n}}_i=\hat{\mathbf{z}}$ , and  $J_{ij}=-1$  for  $i$  and  $j$  nearest neighbors are presented in Fig. 1.3. The squares are the staggered magnetization estimated using the Monte Carlo scheme outlined here. The loss of order indicated by the decrease in the staggered magnetization at very low temperatures is caused by an underflow error in the exponentiation that occurs because in `MetropolisSampling`  $q=\exp[-\beta E(x')]/\exp[-\beta E(x)]$ . The creation of two new classes, one an `MCSampling` model that calculates using  $q=\exp(-\beta[E(x') - E(x)])$  and one that makes  $f(x)=E(x)$ , solves this problem. The error-free results are the circles in Fig. 1.3.

The results shown here highlight some cautions that must be kept in mind. The most important is that generic programming will not enable nonexperts to naively make calculations that would have been infeasible for them otherwise. In fact, the opposite may be true. Expertise and a thorough examination of the results may be required to validate an application assembled in this way. It is precisely the knowledge that calculations can go wrong in many subtle ways that makes the expertise so important.

## 1.4 Summary

Applications of object-oriented and generic programming to Monte Carlo simulation of spin models have been presented. Using object-oriented polymorphism and explicit descriptions of details such as  $\{J_{ij}\}$ , flexible calculations of the energy and fields associated with the spin models are possible. The concepts needed for a generic-programming implementation of Metropolis-sampling Monte Carlo have been outlined. The  $\Psi$ -Mag toolset implements both solutions and has been used to successfully calculate thermodynamic properties of spin systems.

## Acknowledgments

This work was supported by the DOE Office of Science through ASCR-MICS and the Computational Material Science Network of BES-DMSE as well as the Laboratory Directed Research and Development program of ORNL under contract DE-AC05-00OR22725 with UT-Battelle LLC.

## References

1. M. H. Austern: *Generic Programming and the STL* (Addison-Wesley, Reading, Mass., 1999)
2. B. Stroustrup: *The C++ Programming Language* (Addison-Wesley, Reading, Mass., 2000)
3. J. R. Cary, S. G. Shasharina, J. C. Cummings, J. V. W. Reynders, and P. J. Hinker: *Comp. Phys. Comm.* **105**, 20 (1997)
4. <http://www.ccs.ornl.gov/mri/psimag>
5. M. H. Kalos and P. A. Whitlock: *Monte Carlo Methods* (Wiley, New York, 1986)
6. N. Goldenfeld: *Lectures of Phase Transitions and the Renormalization Group* (Addison-Wesley, Reading, Mass., 1992)
7. F. R. Brown and T. J. Woch: *Phys. Rev. Lett.* **58**, 2394 (1987)
8. W. Brown: *Micromagnetics* (Wiley, New York, 1963)
9. W. F. Brown: *IEEE Trans. Magn.* **15**, 1196 (1979)
10. N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller: *J. Chem. Phys.* **21**, 1087 (1953)
11. A. Alexandrescu: *Modern C++ Design* (Addison-Wesley, Reading, Mass. 2001)
12. R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, B. Smolinski: in *Eighth IEEE International Symposium on High Performance Distributed Computing* (IEEE, 1998) [Lawrence Livermore National Laboratory technical report UCRL-JC-134475]
13. D. E. Bernholdt, W. R. Elwasif, J. A. Kohl, T. G. W. Epperly: in *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries*. Available at <http://www.ece.lsu.edu/jxr/ics02workshop.html>
14. W. R. Elwasif, G. Brown, D. E. Bernholdt, and T. C. Schulthess: (unpublished)