monteswitch user manual
(for version 1.0.0)

Tom L. Underwood

April 10, 2016

About *monteswitch*

monteswitch is a package for performing lattice-switch Monte Carlo simulations. It was written by Tom L. Underwood, with the exception of the random number generator, which was written by Takuji Nishimura, Makoto Matsumoto and Josi Rui Faustino de Sousa.

monteswitch is issued under the MIT License. This is a permissive free software license which allows great flexibility regarding use and modification of the software. A copy of the license can be found in the appendix of this document.

Acknowledgements

The development of *monteswitch* was supported by funding from the Engineering and Physical Sciences Research Council (EPSRC), under the supervision of Graeme Ackland. Valuable discussions with Mikhail Mendelev, Nigel Wilding, Andrey Brukhno and Kevin Stratford are gratefully acknowledged.

Disclaimer

While we have endeavoured to ensure *monteswitch* is free from error, we cannot guarantee this. Hence you use *monteswitch* at your own risk.

Conventions used in this manual

Throughout this manual we use:

- `typewriter font` to signify file and program names, command-line arguments to programs, and shell commands and shell output. Shell commands are always provided on stand-alone lines starting with a '`$`', e.g.,

  ```
  $ echo "I am a shell command"
  $ echo "So am I"
  ```

  Furthermore, long shell commands which do not fit on a single line are wrapped on to the following line without the presence of a '`$`', e.g.,

  ```
  $ echo "I am a looooooooooooooooooooooooooooooooooooooooooooo
  ooong shell command"
  $ echo "I am not"
  ```

  Note however that '`$`' is often used in the shell command itself. It should be clear from the context whether this is the significance of any given '`$`' or not.

- **bold font** to signify the names of variables in input and output files for the *monteswitch* programs.

- `UPPER CASE TYPEWRITER FONT` to signify Fortran statements and variables. Note however that the *monteswitch* source code is predominantly lower case (Fortran is case-insensitive). Furthermore, for clarity we sometimes use lower case, e.g., the type of a `REAL` array of size **M_grid_size** is signified as `REAL(M_grid_size)`.

- the usual Unix conventions when describing usage of command-line arguments for a program, e.g., `program [-o] (-c|-d)` signifies that `program` has an optional command-line argument `-o`, and that one of the arguments `-c` or `-d` must be specified.

Contents

Chapter 1

Introduction

Calculating free energies is one of the most fundamental problems in materials science. A plethora of different methods have been developed to this end, each designed with a particlular problem in mind (see, e.g., Ref. [1]). *Lattice-switch Monte Carlo* (LSMC) [2, 3] [1] is a method for calculating the free energy difference between two solid phases. [2] It has been applied to a wide range of systems [2, 4, 5, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14], beginning with the hard-sphere solid, where it was used to determine the free energy difference between the fcc and hcp phases [2, 3] and free energy cost of stacking faults [4, 5]. The method was later applied to soft interatomic potentials [6, 11, 14], systems containing multiple particle species [7, 8], and molecular systems [9, 10, 12, 13]. LSMC has also inspired *phase switch Monte Carlo*, a method for calculating the free energy difference between a solid and a fluid phase [15], which has also seen widespread use [15, 16, 17, 18, 19, 20, 21]. The strength of LSMC is its efficiency: for the purposes of evaluating the free energy difference between pairs of solid phases LSMC is ostensibly more efficient than other existing methods [11, 9]. However it should be noted that this remains somewhat of an open question [4]. Nevertheless one can say that LSMC is *at least* as efficient as existing methods.

Despite its strengths, LSMC has unfortunately yet to have gained widespread popularity. This presumably stems, at least in part, from the lack of an LSMC code which is both versatile and widely available. With this in mind we have developed a package, *monteswitch*, which implements the LSMC method. The

[1] The reader should be aware that LSMC has also been refered to as lattice-*switching* Monte Carlo.

[2] Here we use the term 'phase' rather loosely. By 'phase' here we simply mean 'some system' we are interested in measuring the free energy of relative to 'some other system' via LSMC. A 'phase' could be a solid phase in the conventional sense. For example we could use LSMC to evaluate the free energy difference between the fcc and hcp crystals (taking into account the vibrations of particles about their lattice sites), or the NaCl and CsCl crystals. However one phase could be the fcc crystal while the other could be the fcc crystal exhibiting some concentration c of a certain crystallographic defect. In this case the free energy difference between the two 'phases' would be the free energy cost associated with adding a concentration c of defects to the fcc crystal (where one must be careful to correctly account for the positional entropy of the defects).

package, written in Fortran 95, can be used to evaluate free energy differences between pairs of solid phases in the NVT and NPT ensembles. Furthermore the package contains a version of the main executable which is parallelised using MPI for HPC applications. Note that the two 'phases' under consideration need not be homogeneous crystals. For instance *monteswitch* can be used to evaluate the free energy cost associated with crystallographic defects. Furthermore *monteswitch* can treat systems containing multiple species of particles. However it should be noted that *monteswitch* can only treat 'atomic' systems (i.e., 'non-molecular' systems: those in which the constituent particles do not have rotational degrees of freedom), and pairs of phases which can be represented by orthorhombic unit cells.

While steps have been recently been taken to implement LSMC in existing general-purpose codes, [3] we believe that *monteswitch* will fulfill an important 'gap in the market' for the foreseeable future because it was designed from the outset to be highly-customisable with regards to the interatomic potential. By contrast general-purpose codes tend to have a fixed set of interatomic potentials to draw upon. In *monteswitch* all of the procedures pertaining to the interatomic potential called by the rest of the code are housed within a single Fortran module, which itself is housed in a file `interactions.f95`. Users are encouraged to write their own version of `interactions.f95` which implements the interatomic potential they are interested in. (Of course, in writing their own version of `interactions.f95` users are free to interface with external Fortran modules, or even external programs). Templates are provided with *monteswitch* to assist with this. Furthermore versions of `interactions.f95` are included with *monteswitch* which correspond to some commonly-used interatomic potentials, which can serve as examples. After compiling the package with the desired version of `interactions.f95`, the end result is a version of *monteswitch* in which the desired interatomic potential is 'hard-coded'. (A similar scheme is utilised in the molecular dynamics program MOLDY [23]).

This document describes how to use *monteswitch*. Note that this document does not provide details regarding the structure of the source code. We direct readers interested in such things to the HTML source code documentation – which can be generated from the source code (see Chapter 3) – or failing that the source code itself. Here we assume that the reader is already competent with the Unix shell, including utilities such as `grep`, `sed` and `awk`. Often we provide examples of commands which utilise the shell and the aforementioned utilities to perform certain tasks. However we never describe exactly how these commands work for the sake of brevity; interested users should consult the relevant documentation online or elsewhere. In a similar vein, when we describe Fortran-related issues we assume the reader has an expertise in Fortran 95 . We also assume that the user is familiar with the Monte Carlo method (in the sense of computational chemistry), but not necessarily with LSMC. For further information regarding Monte Carlo simulations in condensed matter physics,

[3] Specifically, LSMC is earmarked for inclusion in the general-purpose Monte Carlo code *DL_MONTE* [22].

see, e.g., Ref. [1].

The layout of this document is as follows. In Chapter 2 we describe the theory behind LSMC to a depth which enables the user to competently perform LSMC simulations using *monteswitch*. In Chapter 3 we describe how to install *monteswitch*, and provide an overview of the package. In Chapter 4 we describe how to use the main Monte Carlo programs within *monteswitch*. In Chapter 5 we describe the various interatomic potentials included with *monteswitch*, as well as how users can implement their own potentials. In Chapter 6 we describe how to use the various utility programs included with *monteswitch* for the creation of input files and post-processing of output files. In Chapter 7 we provide a worked example which elucidates how *monteswitch* can be used to solve a 'real' problem. Finally in Chapter 8 we describe the test cases included with *monteswitch* which, as well as validating *monteswitch*, serve as an archive of examples for users.

Chapter 2

Background theory

In this chapter we describe the LSMC methodology as relevant to *monteswitch*. Further information regarding different incarnations of LSMC can be found in the LSMC studies cited in the previous chapter.

2.1 What do we mean by 'solid phase'?

LSMC is a method to calculate the free energy difference between two solid phases. The term 'solid phase' could mean a number of things; it is worth clarifying exactly what we mean by this term before proceeding. Here we use the following definition, which is convenient for our purposes. We define a solid phase (henceforth we omit the word 'solid' for the sake of brevity) using some state of the system which is *representative* of the phase in question. We refer to such a state as a *representative state*: phase α is the set of all states of the system which are 'reachable' from the chosen representative state for α on the timescale of the simulation. For example, if one is interested in the 'fcc phase', i.e., the set of states corresponding to thermodynamic fluctuations in the particle positions about their average positions (which form an fcc crystal lattice), then a suitable choice of representative state would be that in which the particle positions form an ideal fcc crystal lattice. A simulation initialised in this state would, on the timescale of the simulation, explore the configurations which constitute the fcc phase.

2.2 Calculating free energy differences

Consider now a system which is free to visit two phases 1 and 2 (and only phases 1 and 2). Of phases 1 and 2, the equilibrium phase is that with the lower free energy $\mathcal{F}$, where $\mathcal{F}$ is the Helmholtz free energy F in the NVT ensemble and the Gibbs free energy G in the NPT ensemble. It is the free energy difference between the phases $\Delta\mathcal{F} \equiv \mathcal{F}_1 - \mathcal{F}_2$ which we wish to evaluate, where $\mathcal{F}_1$ and $\mathcal{F}_2$ denote the free energies of phases 1 and 2. We now derive an expression for $\Delta\mathcal{F}$

which allows $\Delta\mathcal{F}$ to be determined from the relative time t_1 and t_2 which the system spends in each phase 1 and 2 during a sufficiently long simulation. It is this expression which is ultimately used in LSMC to calculate $\Delta\mathcal{F}$. Consider the free energy of phase α:

$$\mathcal{F}_\alpha = -\beta^{-1} \ln Z_\alpha, \tag{2.1}$$

where

$$Z_\alpha = \sum_{\sigma \in \alpha} \exp(-\beta \mathcal{E}_\sigma) \tag{2.2}$$

is the partition function for phase α; $\beta \equiv 1/(k_B T)$; k_B is Boltzmann's constant; σ denotes a state of the system; $\mathcal{E}_\sigma$ denotes the energy E_σ of state σ in the NVT ensemble, and the enthalpy $H_\sigma = E_\sigma + PV_\sigma$ of state σ in the NPT ensemble, where V_σ denotes the volume of state σ; and the summation in the above equation is over all states which 'belong' to phase α. From Eqn. (2.1) it follows that

$$\Delta\mathcal{F} = \beta^{-1} \ln\left(\frac{Z_2}{Z_1}\right). \tag{2.3}$$

Now,

$$p_\sigma = \frac{1}{Z} e^{-\beta \mathcal{E}_\sigma}, \quad Z \equiv Z_1 + Z_2 \tag{2.4}$$

is the probability that the system, free to explore both phases 1 and 2, is in state σ. Hence the probability of the system being in phase α is

$$p_\alpha = \sum_{\sigma \in \alpha} p_\sigma = Z_\alpha / Z, \tag{2.5}$$

where we have used Eqn. (2.2). Finally, substituting this into Eqn. (2.3) gives

$$\Delta\mathcal{F} = \beta^{-1} \ln\left(\frac{p_2}{p_1}\right), \tag{2.6}$$

where p_1 and p_2 denote the probability of the system being in phase 1 and 2 respectively.

The above equation suggests the following method for calculating $\Delta\mathcal{F}$ from, e.g. a molecular dynamics simulation: measure the relative time t_1 and t_2 which the system spends in each phase 1 and 2 during the simulation, and substitute these quantities into the above equation – bearing in mind that $t_2/t_1 = p_2/p_1$. However, this method is often intractable in practice for two solid phases, because the time taken for the system to transition between the two phases is too long to allow a reasonable estimate of t_2/t_1 to be deduced in a reasonable simulation time; it may be the case that, regardless of the phase in which the simulation is initialised, the system *never* transitions to the 'other' phase during the course of the simulation. The problem is that, while the regions of phase space corresponding to phase 1 and phase 2 both correspond to probable states of the system at thermodynamic equilibrium, these regions are separated by a *free energy barrier* – a region of phase space associated with states which are

very improbable at thermodynamic equilibrium. This barrier inhibits transitions between the 'islands of stability' in phase space associated with phase 1 and phase 2.

This problem can in principle be circumvented within the Monte Carlo method. In the original incarnation of Monte Carlo, which we refer to as *canonical Monte Carlo*[24] (which we contrast later to *multicanonical Monte Carlo*), the system is evolved throughout the simulation as follows. Each timestep we generate a trial state of the system σ', and attempt to change the system from its current state σ, to the trial state. The traditional approach for NVT ensembles is to perform a 'particle move' to generate a trial state. Here, one particle in σ is moved to yield σ'. In NPT ensembles particle moves are supplemented by 'volume moves', in which the volume, and potentially shape, of the entire system is altered, along with a commensurate rescaling of the particle positions. We accept the change of state from σ to σ' with a probability $p_{\sigma\rightarrow\sigma'}$, which is a function of the energies of the states σ and σ' in the NVT ensemble, and the enthalpies and volumes of the states σ and σ' in the NPT ensemble. The function also depends on the specific scheme used to generate state σ' from σ. For the schemes used in *monteswitch*, $p_{\sigma\rightarrow\sigma'}$ is given by

$$p_{\sigma\rightarrow\sigma'} = \max\Big[1, e^{\beta(E_{\sigma'}-E_{\sigma})}\Big] \tag{2.7}$$

for particle moves and

$$p_{\sigma\rightarrow\sigma'} = \max\Big[1, e^{\beta(H_{\sigma'}-H_{\sigma})+(N+1)\ln(V_{\sigma'}/V_{\sigma})}\Big], \tag{2.8}$$

for volume moves. [1]. The end result is that each state σ is sampled with a probability which reflects the underlying ensemble, i.e., Eqn. (2.4). Hence the equilibrium value of any physical quantity X can be obtained by evaluating the average of X over all timesteps in a sufficiently long simulation. In this manner t_1 and t_2, and hence $\Delta\mathcal{F}$ (via Eqn. (2.6)) can be calculated; t_α is the time average of the quantity θ_α, where $\theta_\alpha = 1$ if the system is in phase α and 0 otherwise. However, in canonical Monte Carlo one has considerable freedom as to how trial states are generated; one is by no means limited to the aforementioned 'traditional' move set. The prospect therefore exists of generating trial states in a manner which results in the system traversing a path in phase space which allows $\Delta\mathcal{F}$ to be calculated relatively quickly. Such a path would involve frequent transitions between both phases 1 and 2 by 'tunnelling' through the free energy barrier separating them.

2.3 Lattice switch Monte Carlo moves

In LSMC a new type of move, a *lattice switch*, is introduced to supplement the traditional move set mentioned above. A lattice switch move takes the system *directly* from one phase to the other, bypassing any free energy barriers

[1] See, e.g, Ref. [1] for expressions for $p_{\sigma\rightarrow\sigma'}$ for other schemes

separating the phases. Every time a lattice switch is accepted, the system transitions to the 'other' phase. The salient feature of the move is that the underlying 'lattice' which characterises the current phase is 'switched' for a lattice which characterises the other phase. For example if the two phases in question are fcc and hcp, then a lattice switch from an fcc state involves switching the underlying fcc lattice for an hcp lattice, while preserving the *displacements* of each particle from its associated lattice site.

Of course this is just a basic description of the lattice switch. We now explain exactly how lattice switches are implemented in *monteswitch*. In *monteswitch* two states are provided at initialisation, one for each phase. The functions of these states are twofold. Firstly, they act as prospective initial states for the simulation: if the system is to be initialised in phase 1, then the phase-1 state will be used as the initial state. Secondly, they are used to determine the nature of the lattice switch move. We refer to these states as *reference states*. Now, in *monteswitch* only orthorhombic systems can be treated. Hence we only consider orthorhombic systems here. For such systems a state amounts to a specification of the following: the dimensions of the supercell L_x, L_y and L_z in each Cartesian direction; either the *fractional* positions $\{\tilde{\mathbf{r}}_i\}$ or the *absolute* positions $\{\mathbf{r}_i\}$ of the particles within the supercell (we denote fractional positions by a '$\sim$'); and the species of the particles $\{S_i\}$. For convenience we gather these quantites into a tuple: $(L_x, L_y, L_z, \{\tilde{\mathbf{r}}_i\}, \{S_i\})$ or $(L_x, L_y, L_z, \{\mathbf{r}_i\}, \{S_i\})$ amounts to a specification of the state of the system. With this in mind, let $(\mathcal{L}_x^{(1)}, \mathcal{L}_y^{(1)}, \mathcal{L}_z^{(1)}, \{\tilde{\mathcal{R}}_i^{(1)}\}, \{\mathcal{S}_i^{(1)}\})$ and $(\mathcal{L}_x^{(2)}, \mathcal{L}_y^{(2)}, \mathcal{L}_z^{(2)}, \{\tilde{\mathcal{R}}_i^{(2)}\}, \{\mathcal{S}_i^{(2)}\})$ describe the reference states for phases 1 and 2 respectively. Without loss of generality consider now a state $\sigma = (L_x, L_y, L_z, \{\mathbf{r}_i\}, \{s_i\})$ belonging to phase 1. We now describe how the state $\sigma' = (L'_x, L'_y, L'_z, \{\mathbf{r}'_i\}, \{s'_i\})$ which results from a lattice switch from state $\sigma$is generated from σ in *monteswitch*. We first construct the underlying phase-1 'lattice' for σ. The positions of the lattice sites, which we denote as $\{\mathbf{R}_i^{(1)}\}$, are constructed from the phase-1 reference state. Specifically, $\{\mathbf{R}_i^{(1)}\}$, transformed into fractional coordinates, is identical to the fractional positions of the particles in the phase-1 reference state:

$$\mathbf{R}_i^{(1)} = \mathrm{diag}(L_x, L_y, L_z)\tilde{\mathcal{R}}_i^{(1)} \tag{2.9}$$

for all particles i, where $\mathrm{diag}(L_x, L_y, L_z)$ denotes the 3×3 matrix with L_x, L_y and L_z along the leading diagonal. We now turn to constructing the trial state σ'. We begin by constructing the dimensions of the supercell for σ'. The reference states for both phases are used to determine how the supercell is transformed during the switch. Specifically, the dimensions of the supercell for σ' are given by

$$L'_x = \frac{\mathcal{L}_x^{(2)}}{\mathcal{L}_x^{(1)}} L_x, \quad L'_y = \frac{\mathcal{L}_y^{(2)}}{\mathcal{L}_y^{(1)}} L_y, \quad L'_z = \frac{\mathcal{L}_z^{(2)}}{\mathcal{L}_x^{(1)}} L_z. \tag{2.10}$$

Hence in the lattice switch the supercell is dilated/contracted in each Cartesian dimension by the same factor as would be the case in transforming the supercell

from the phase-1 reference state to the phase-2 reference state. Thirdly, we construct the underlying lattice for phase 2. As was the case for the underlying lattice for phase 1, the underlying lattice for phase 2, transformed into fractional coordinates, is identical to the fractional positions of the particles in the phase-2 reference state:

$$\mathbf{R}_i^{(2)} = \mathrm{diag}(L'_x, L'_y, L'_z)\tilde{\mathcal{R}}_i^{(2)} \tag{2.11}$$

for all i. Fourthly we construct the particle positions in σ' using the particle positions in σ and the aforementioned underlying lattices for phases 1 and 2 as follows:

$$\mathbf{r}'_i = \mathbf{r}_i - \mathbf{R}_i^1 + \mathbf{R}_i^2 \tag{2.12}$$

for all i. Note that this amounts to a transformation of the particle positions in which the underlying phase-1 lattice $\{\mathbf{R}_i^1\}$ is substituted for a phase-2 lattice $\{\mathbf{R}_i^2\}$ while keeping the *displacement* of each particle from its lattice site unchanged. This can perhaps be seen more clearly by reexpressing the transformation as follows:

$$\mathbf{r}_i = \mathbf{R}_i^{(1)} + \mathbf{u}_i \to \mathbf{r}'_i = \mathbf{R}_i^{(2)} + \mathbf{u}_i \tag{2.13}$$

for all i, where

$$\mathbf{u}_i = \mathbf{r}_i - \mathbf{R}_i^{(1)} = \mathbf{r}'_i - \mathbf{R}_i^{(2)} \tag{2.14}$$

is the displacement of particle i from its lattice site. Finally we transform the species of σ (which initialy corresponding to phase 1) to correspond to phase 2:

$$s_i = S_i^{(1)} \to s'_i = S_i^{(2)} \tag{2.15}$$

for all i.

Note that the lattice switch as just described is an involution: if a lattice switch from state σ yields state σ', then performing a lattice switch from state σ' yields σ.

2.4 Multicanonical Monte Carlo

One might expect that by regularly making lattice switches, the system will regularly transition between phases, allowing $\Delta\mathcal{F}$ to be efficiently evaluated as described above. Unfortunately using canonical Monte Carlo one finds that lattice switches are too rarely accepted for this approach to be useful. The solution to this problem is to use *multicanonical Monte Carlo*[25, 26, 27] instead of canonical Monte Carlo. [2] The former is a straightforward generalisation of the latter in which, instead of sampling from the probability distribution p_σ (Eqn. (2.4)), one samples from the distribution

$$\tilde{p}_\sigma = \frac{1}{\tilde{Z}} e^{-\beta\mathcal{E}_\sigma} e^{\eta_\sigma}, \quad \tilde{Z} \equiv \sum_\sigma e^{-\beta\mathcal{E}_\sigma} e^{\eta_\sigma}, \tag{2.16}$$

[2] Multicanonical Monte Carlo belongs to the class of extended/expanded ensemble techniques; see, e.g., Ref. [28].

where η_σ, known as the *weight function*, is chosen according to the aims of the simulation. Using Eqn. (2.4), it can be shown that

$$\tilde{p}_\sigma = \frac{1}{\tilde{Z}'} p_\sigma e^{\eta_\sigma}, \quad \tilde{Z}' = \sum_\sigma p_\sigma e^{\eta_\sigma}, \tag{2.17}$$

where note that p_σ is the 'true' probability of the system being in state σ. From the above it can be seen that if $\eta_\sigma > 0$ then state σ is *over-sampled* relative to the true distribution. On the other hand if $\eta_\sigma < 0$ then σ is *under-sampled*. The strength of this approach is that, through judicious choice of the weight function, one can 'choose' the path the system traverses through phase space. To elaborate, a multicanonical Monte Carlo simulation can be regarded as a canonical Monte Carlo simulation, but if the energy for each state σ were

$$\tilde{\mathcal{E}}_\sigma = \mathcal{E}_\sigma - \eta_\sigma/\beta \tag{2.18}$$

instead of $\mathcal{E}_\sigma$. Hence a multicanonical Monte Carlo simulation samples states with probabilities corresponding to, say, an NVT ensemble, but if energy were $\tilde{\mathcal{E}}_\sigma$ instead of E_σ. Noting that since the modification to the 'true' energy function, $-\eta_\sigma/\beta$, is proportional to the weight function, it can be seen that the weight function defines an additional 'force' on the system which affects its trajectory through phase space. Therefore by choosing η_σ, one can choose the system's path through phase space.

Of course, in a multicanonical simulation the states are not longer sampled with probabilities corresponding to the true ensemble in question – which *is* the case for canonical Monte Carlo. Accordingly the time average of some physical quantity X throughout a long multicanonical Monte Carlo simulation is not equivalent to the equilibrium value of X for the true ensemble, like it is in a canonical Monte Carlo simulation. Nevertheless one can obtain the equilibrium value of X from a multicanonical simulation by exploiting the fact that, since the weight function is known, then so is the degree of over- or under-sampling of each state. To elaborate, to evaluate the equilibrium value of X in a Metropolis Monte Carlo simulation, we use the following expression:

$$\langle X \rangle \approx \frac{1}{\tau} \sum_{t=1}^{\tau} X(t), \tag{2.19}$$

where $X(t)$ denotes the value of the physical quantity X at timestep t of the simulation, and τ denotes the total number of timesteps. For reasons which will become clear in a moment, $\langle X \rangle$ could equivalently be expressed as

$$\langle X \rangle \approx \frac{\sum_{t=1}^{\tau} w(t) X(t)}{\sum_{t=1}^{\tau} w(t)} \tag{2.20}$$

with $w(t)$ taking the same constant value for all t, where $w(t)$ is the *weight* (not to be confused with the weight function) associated with the state at timestep t. Now, Eqn. (2.17) reveals that state σ is sampled in a multicanonical Monte Carlo simulation a factor of $e^{\eta_\sigma}/\tilde{Z}'$ more often than for the true ensemble. Associating a weight $(e^{\eta_\sigma}/\tilde{Z}')^{-1}$ to σ corrects for this; the effect is that states which are over-sampled are counted less, and under-sampled states are counted more, in the evaluation of $\langle X \rangle$. Hence the expression for $\langle X \rangle$ pertaining to a multicanonical simulation is the same as above, but with $w(t) = (e^{\eta(t)}/\tilde{Z}')^{-1}$:

$$\langle X \rangle \approx \frac{\sum_{t=1}^{\tau} e^{-\eta(t)} X(t)}{\sum_{t=1}^{\tau} e^{-\eta(t)}}, \tag{2.21}$$

where we have canceled the factors of $\tilde{Z}'^{-1}$ from the numerator and denominator.

2.5 Multicanonical Monte Carlo in LSMC

How does multicanonical Monte Carlo resolve the problem that lattice switch moves are too rarely accepted to be useful? To answer this question we must first understand why lattice switches are so rarely accepted. Consider the probability of a lattice switch from state σ to σ' being accepted. Using the lattice switch scheme implemented in *monteswitch*, and described in Sec. 2.3, this is

$$p_{\sigma \to \sigma'} = \max\left[1, e^{\beta(H_{\sigma'} - H_\sigma) + \ln(V_{\sigma'}/V_\sigma)}\right], \tag{2.22}$$

where recall that $H_\sigma = E_\sigma + PV_\sigma$ denotes the enthalpy of state σ, and V_σ denotes its volume. Consider first the case where the lattice switch preserves the volume of the system. This is necessary for the NVT ensemble, but in the NPT ensemble one has freedom to define the switch such that it changes the volume of the system. In this case, since $V_{\sigma'} = V_\sigma$, the above equation reduces to

$$p_{\sigma \to \sigma'} = \max\left[1, e^{\beta(E_{\sigma'} - E_\sigma)}\right]. \tag{2.23}$$

This equation reveals the main cause for lattice switches being unsuccessful: the energy of the trial state σ' generated by a lattice switch is almost always of much higher energy than the current state σ, leading to $p_{\sigma \to \sigma'}$ being very low. This is perhaps obvious in retrospect: one should expect that, from a state which is probable at equilibrium, which accordingly will have a relatively low energy, a large change to the system, such as that brought about by a lattice switch, will typically lead to a huge increase in the system's energy. There are, however, (hopefully) a small number of states 'close' to those found at equilibrium for each phase from which a lattice switch yields a trial state σ' which is of comparable energy to σ. From such states a lattice switch has a

good chance of being accepted. We refer to such states as *gateway states*, since they provide the key to jumping between both phases. It is these states which we wish to over-sample, and we set the weight function accordingly. The idea is that by over-sampling these states, lattice switches are accepted reasonably often, enabling both phases to be explored in a reasonable simulation time. This in turn allows us to determine t_1 and t_2 (which recall are the time averages of θ_1 and θ_2 defined earlier, i.e. $t_1 = \langle\theta_1\rangle$ and $t_2 = \langle\theta_2\rangle$) via Eqn. (2.21), and hence ultimately $\Delta\mathcal{F}$ via Eqn. (2.6).

But how does the above discussion change for lattice switches which alter the volume of the system? The answer is 'not much'. For the case of volume-preserving lattice switches a gateway state is a state from which a lattice switch yields another state of comparable energy. This reflects the fact that $p_{\sigma\to\sigma'} \approx 1$ if $E_{\sigma'} \approx E_\sigma$ in Eqn. (2.23). For the general case of a non-volume-preserving lattice switch the analogous condition for a gateway state, as can be seen from Eqn. (2.22), is that $J_\sigma \approx J_{\sigma'}$, where

$$J_\sigma \equiv H_\sigma + \ln(V_\sigma)/\beta. \tag{2.24}$$

Note that for the case of volume-preserving lattice switches the condition $J_\sigma \approx J_{\sigma'}$ is equivalent to $E_\sigma \approx E_{\sigma'}$.

With the above in mind, how should the weight function be engineered such that gateway states are over-sampled? Consider a state σ, and let σ' denote the state which results from a lattice switch performed from σ. Let us define the state-dependent quantity

$$M_\sigma = \begin{cases} (J_\sigma - J_{\sigma'}) & \text{if } \sigma \text{ belongs to phase 1} \\ -(J_\sigma - J_{\sigma'}) & \text{if } \sigma \text{ belongs to phase 2.} \end{cases} \tag{2.25}$$

This quantity in practice provides a means for resolving gateway states, states corresponding to equilibrium (for the true ensemble under consideration) for phase 1, and states corresponding to equilibrium for phase 2. Accordingly we refer to M as the 'order parameter'. Consider first gateway states. Above we illustrated that gateway states correspond to the condition $J_\sigma \approx J_{\sigma'}$. As can be seen from the above this corresponds to states with $M_\sigma \approx 0$. By contrast, if $|M_\sigma| \gg 0$, then the two states have significantly different values of J. In this case, while switching from the state with the higher value of J to that with the lower value of J is guaranteed, the converse is not: the two states are not concordant with switching *to and from* both phases. $|M_\sigma|$ therefore provides a measure of how 'un-gateway-like' state σ is, with zero corresponding to 'very gateway-like'. Consider now phase-1-equilibrium states. From such states we generally expect a lattice switch to be unsuccessful, and hence $J_{\sigma'} \gg J_\sigma$. Therefore for such states $M_\sigma \ll 0$. Finally consider phase-2-equilibrium states. Similarly we generally expect a lattice switch from such states to be unsuccessful, and hence $J_{\sigma'} \gg J_\sigma$. However this time $M_\sigma \gg 0$. We therefore have three regimes: $M_\sigma \ll 0$ corresponds to phase-1-equilibrium states; $M_\sigma \approx 0$ corresponds to gateway states; and $M_\sigma \gg 0$ corresponds to phase-2-equilibrium states.

With this in mind, if we choose the weight function η_σ to take the same value η_M for all states with the same M, and also choose η_M to be peaked at $M = 0$ and to decay monotonically with $|M|$, then the weight function corresponds to a 'force' which drives the system towards gateway states, allowing the system to transition between the phase-1 and phase-2 regions of phase space, corresponding to $M \ll 0$ and $M \gg 0$ respectively, in a reasonable simulation time. This is, of course, just a *qualitative* description of a form for η_M which is sufficient for our purposes. As one might expect, the quantitative details of the weight function η_M strongly affect the efficiency of the path traversed though phase space with regards to calculating $\Delta\mathcal{F}$; a 'bad' weight function might result in the system getting stuck in one phase, or an unimportant region of phase space, for a long time. Furthermore, it is not obvious *a priori* what a suitable weight function for a given system should be. Hence one must *generate* a weight function which leads to an efficient sampling of phase space. After this *weight function generation simulation*, the resulting weight function can be used in a *production simulation* to calculate $\Delta\mathcal{F}$ as described earlier.

However in practice one cannot treat M as an unbounded continuous variable as above; one cannot define an arbitrary weight function in computer memory via this scheme. Hence in practice one considers a finite range of M which is divided into N_{macro} bins, each corresponding to a distinct range of 'M-space'. Each bin itself corresponds to a macrostate: the macrostate is the collection of states corresponding to the range of M-space covered by the bin. We will henceforth explicitly take this discretisation of M into account, and let $\mathcal{M}$ denote the macrostate corresponding to the $\mathcal{M}$th bin, where $\mathcal{M} = 1, 2, \ldots, N_{\text{macro}}$. Accordingly let $\eta_\mathcal{M}$ denote the weight function for macrostate $\mathcal{M}$.

However this raises the question of what range of M should be considered for a given problem, as well as what value N_{macro} should take. With regards to the first question, the range should be large enough to cover all states which are reasonably likely to occur at thermodynamic equilibrium for both phases, as well as the gateway states. This is necessary because allowing the production simulation to visit the equilibrium states for both phases is essential in order to accurately determine the free energy difference between the two phases (as well as any thermodynamic quantities associated with each individual phase), while allowing the simulation to visit gateway states is essential to allow transitions between the two phases via lattice switches. On the other hand the range should not be too large. Assuming that an ideal weight function is used, in which case the considered range of M-space is sampled uniformly in the production simulation, then using too large a range lowers the efficiency of the simulation because it forces the system to spend less time in the regions of order-parameter space associated with equilibrium states – which are the important states with regards to evaluating thermodynamic quantities. Regarding what value N_{macro} should take, note that N_{macro} determines the fineness of the grid on M-space upon which the weight function is defined. Note also that the weight function is flat over the region of M-space corresponding to one macrostate. Furthermore, the higher N_{macro}, the finer the grid. Hence if the grid is coarse, i.e., if N_{macro} is small, then there will be large regions of M-space over which the weight function

is flat, including near $M = 0$. This is problematic because a rapidly-changing weight function is required near $M = 0$ to 'guide' the system over the free energy barrier. Thus if N_{macro} is too small then the free energy barrier may prove insurmountable despite the use of multicanonical sampling. This problem can be avoided if a sufficiently large N_{macro} is used, however the larger N_{macro} the longer the simulation time required to generate the weight function. Hence a balance must be struck when choosing an appropriate value of N_{macro}.

2.6 Weight function generation methods

We will now describe how weight functions can be generated for use in a production simulation. In multicanonical Monte Carlo the 'ideal' weight function, which we denote as $\eta^*_{\mathcal{M}}$, is regarded as that which leads to all macrostates within the considered order parameter range to be sampled with equal probability in the multicanonical Monte Carlo simulation. Let $\tilde{p}_{\mathcal{M}}$ denote the probability that the multicanonical Monte Carlo simulation is in macrostate $\mathcal{M}$; formally,

$$\tilde{p}_{\mathcal{M}} = \sum_{\sigma \in \mathcal{M}} \tilde{p}_\sigma. \tag{2.26}$$

Thus $\eta^*_{\mathcal{M}}$ would yield $\tilde{p}_{\mathcal{M}} = C$ for all $\mathcal{M}$ within the allowed range, where C is some constant. $\eta^*_{\mathcal{M}}$, or at least a close estimate of it, can be determined using various methods; below we describe the methods supported by *monteswitch*. However we emphasise that strictly speaking it is unnecessary to use the ideal weight function, which samples all macrostates with exactly equal probablility, during the production simulation. The choice of weight function affects the accuracy of physical quantities determined during the production simulation (evaluated using Eqn. (2.21)) only in that it determines the efficiency with which phase space is explored. For this reason though a weight function close to the ideal is desirable because it is a convenient way of ensuring that gateway states are visited often, hence allowing both phases to be explored regularly: a weight function which is 'far from ideal' may not result in gateway states being visited often enough to enable useful estimates of t_1 and t_2 to be obtained. However, using the latter weight function would enable useful estimates for t_1 and t_2 to be obtained *eventually*.

2.6.1 Visited states method

The *visited states method* (see Ref. [27] and references therein) is arguably the simplest method for generating the weight function. In this method, the simulation consists of a number of 'blocks', which themselves consist of a large number of Monte Carlo sweeps. Multicanonical sampling is used throughout, and the weight function is updated at the end of each block. The weight function is different – closer to the ideal – in each subsequent block, and information collected during each block is used to inform the weight function to be used in the next block. Eventually the weight function converges on the ideal: it provides

a 'flat' macrostate histogram; the weight function is such that all macrostates are sampled with equal probability.

Let $H_{\mathcal{M}}^{(n)}$ denote the number of states belonging to macrostate $\mathcal{M}$ which were visited during block n. The specific equation used to obtain the weight function for the next block $n+1$ is

$$\eta_{\mathcal{M}}^{(n+1)} = \eta_{\mathcal{M}}^{(n)} - \ln\left\{ \frac{H_{\mathcal{M}}^{(n)} + 1}{\sum_{\mathcal{M}'} \left(H_{\mathcal{M}'}^{(n)} + 1 \right)} \right\} + k, \tag{2.27}$$

where $\eta_{\mathcal{M}}^{(n)}$ denotes the weight function for block n; the summation over $\mathcal{M}'$ on the denominator of the fraction is over all macrostates $1, 2, \ldots, N_{\text{macro}}$; and k is an inconsequential arbitrary constant, which we choose such that the minimum value of $\eta_{\mathcal{M}}^{(n+1)}$ over all $\mathcal{M}$ is 0. [3] The above equation acts to enhance the value of the weight function for macrostates which have not been visited often during the previous block, and diminish the weight function for macrostates which have been visited. Eventually the weight function is such that all macrostates are visited equally often. With regards to the choice of initial weight function for the simulation, i.e., $\eta_{\mathcal{M}}^{(0)}$, we use $\eta_{\mathcal{M}}^{(0)} = 1$ for all $\mathcal{M}$.

In practice the visited states method converges to the ideal weight function very slowly, and is not amenable to parallelisation. For this reason we do not recommend that it is the sole method used to generate weight function with *monteswitch*. We recommend that the other methods for generating weight functions described in a moment are used instead, at least at first. These other methods are much more efficient at obtaining a weight function close to the ideal. However, these methods, as implemented in *monteswitch*, perhaps never converge on the ideal, while the visited states method does [27]. This limitation of the other methods is in practice probably unimportant. As mentioned earlier, it is unnecessary to use the 'exact' ideal weight function, just a weight function 'close' to the ideal, which these methods provide. Nevertheless the visited states method is included within *monteswitch* in order that the exact ideal weight function can be calculated if required, possibly after a weight function close to the ideal has already been obtained using the aforementioned faster methods.

2.6.2 Transition matrix method

$\eta_{\mathcal{M}}^{*}$ can be related to $p_{\mathcal{M}}$, the *canonical* probability of the system being in macrostate $\mathcal{M}$. (Multicanonical quantities are signified by a '$\sim$'). Firstly, note that the multicanonical probability $\tilde{p}_{\mathcal{M}}$ of the system being in macrostate $\mathcal{M}$ given a weight function $\eta_{\mathcal{M}}$ is

$$\tilde{p}_{\mathcal{M}} \propto e^{\eta_{\mathcal{M}}} p_{\mathcal{M}}. \tag{2.28}$$

[3] Like energies, only differences in the values of the weight function between states are 'physically significant'; the absolute values of the weight function are not. Hence performing a global shift to the weight function for our own convenience is inconsequential to the multicanonical distribution $\tilde{p}_{\sigma}$. This can be seen by applying the transformation $\eta_{\sigma} \to \eta_{\sigma} + C$ to Eqn. (2.16), where C is ab arbitrary constant

This follows from Eqn. (2.17): summing over all $\sigma \in \mathcal{M}$ on the left-hand expression, and recalling that $\eta_\sigma = \eta_\mathcal{M}$ for such σ. Applying the condition $\tilde{p}_\mathcal{M} = C$ for $\eta_\mathcal{M} = \eta^*_\mathcal{M}$ yields

$$C \propto e^{\eta^*_\mathcal{M}} p_\mathcal{M}, \tag{2.29}$$

which can be rearranged to give

$$\eta^*_\mathcal{M} = C' - \ln p_\mathcal{M}, \tag{2.30}$$

where C' is a constant. Eqn. (2.30) reveals that $\eta^*_\mathcal{M}$ can be derived from $p_\mathcal{M}$. $p_\mathcal{M}$ in turn can be determined from the *macrostate transition probability matrix* $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$, which describes the probability that the system, currently in macrostate $\mathcal{M}$, transitions to macrostate $\mathcal{M}'$ *in the canonical ensemble*. In the *transition matrix method* [27, 29] we determine $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$, then use this to obtain $p_\mathcal{M}$, and finally the ideal weight function $\eta^*_\mathcal{M}$ via Eqn. (2.30).

How do we determine $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$, and how do we obtain $p_\mathcal{M}$ from $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$? Let us first deal with how $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$ is determined. Surprisingly $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$, a canonical quantity, can be obtained from a simulation utilising canonical sampling, multicanonical sampling, or even more exotic methods. Consider the case of canonical sampling first. Here, $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$ can be determined in an intuitive manner as follows. During the simulation we keep track of the number of observed transitions between all pairs of macrostates, which we store in a matrix $H_{\mathcal{M}\mathcal{M}'}$ – where $H_{\mathcal{M}\mathcal{M}'}$ denotes the number of observed transitions from macrostate $\mathcal{M}$ to macrostate $\mathcal{M}'$. We then use $H_{\mathcal{M}\mathcal{M}'}$ to obtain an estimate for $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$ via the equation

$$\mathcal{T}_{\mathcal{M}\mathcal{M}'} \approx \frac{H_{\mathcal{M}\mathcal{M}'} + 1}{\sum_{\mathcal{M}''} \left(H_{\mathcal{M}\mathcal{M}''} + 1 \right)}. \tag{2.31}$$

However, this procedure is never used in *monteswitch* because the following superior alternative exists. Consider a trial state σ' generated from σ which, if accepted, would take the system from macrostate $\mathcal{M}$ to macrostate $\mathcal{M}'$, and let the probability of the move being accepted be $p_{\sigma\to\sigma'}$. Instead of performing the update $H_{\mathcal{M}\mathcal{M}'} \to H_{\mathcal{M}\mathcal{M}'} + 1$ if the move is accepted and $H_{\mathcal{M}\mathcal{M}'} \to H_{\mathcal{M}\mathcal{M}'}$ if it is not, one can perform the update

$$\begin{aligned} H_{\mathcal{M}\mathcal{M}'} &\to H_{\mathcal{M}\mathcal{M}'} + p_{\sigma\to\sigma'}, \\ H_{\mathcal{M}\mathcal{M}} &\to H_{\mathcal{M}\mathcal{M}} + 1 - p_{\sigma\to\sigma'}, \end{aligned} \tag{2.32}$$

regardless of whether it is accepted or not. In this case $H_{\mathcal{M}\mathcal{M}'}$ is no longer the number of observed transitions from macrostate $\mathcal{M}$ to $\mathcal{M}'$, but is instead the number of *inferred* transitions. In fact, as alluded to above, this procedure can be used in conjunction with a multicanonical simulation, so long as one bears in mind that $p_{\sigma\to\sigma'}$ is the *canonical* probability of transitioning from state σ to σ' – a quantity which is trivial to calculate during a multicanonical simulation. Why would one wish to use multicanonical sampling instead of canonical sampling to obtain $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$, which after all is a canonical quantity? If multicanonical sampling

is used, while the canonical transition probability $p_{\sigma\to\sigma'}$ is used to update $H_{\mathcal{M}\mathcal{M}'}$ via the above equation, it is the multicanonical transition probability $\tilde{p}_{\sigma\to\sigma'}$, which depends on the weight functions of σ and σ' ($\eta_{\mathcal{M}}$ and $\eta_{\mathcal{M}'}$ respectively), which is actually used to determine whether the move is accepted or not. This is useful because canonical sampling alone may not be sufficient to explore the full range of macrostates $\mathcal{M}$, which is necessary to 'fill up' the matrix $H_{\mathcal{M}\mathcal{M}'}$ and ultimately enable $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$ to be calculated (Eqn. (2.31)). For instance, typically canonical sampling will never result in the macrostates containing gateway states being visited during a reasonable simulation time. One can take this idea further and even go beyond multicanonical sampling to sample phase space. We discuss methods for exploring the phase space with the aim of determining $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$, and ultimately $\eta^*_{\mathcal{M}}$, in a moment.

Having determined $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$, our task is to now calculate $p_{\mathcal{M}}$. It can be shown that the macrostates obey the following detailed balance condition [27]:

$$\mathcal{T}_{\mathcal{M}'\mathcal{M}}p_{\mathcal{M}'} = \mathcal{T}_{\mathcal{M}\mathcal{M}'}p_{\mathcal{M}}. \tag{2.33}$$

Setting $\mathcal{M}' = \mathcal{M} + 1$ and rearranging the above gives

$$p_{(\mathcal{M}+1)} = \frac{\mathcal{T}_{\mathcal{M}(\mathcal{M}+1)}}{\mathcal{T}_{(\mathcal{M}+1)\mathcal{M}}}p_{\mathcal{M}}. \tag{2.34}$$

Using this equation, $p_{\mathcal{M}}$ can be obtained from the matrix $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$ via the following procedure. Firstly, one chooses some arbitrary value for p_1. With this p_2 can be obtained from the above equation ($\mathcal{M} = 1$ in Eqn. (2.34)). This in turn can be used to obtain p_3 ($\mathcal{M} = 2$ in Eqn. (2.34)), which in turn can be used to obtain p_4, etc., until $p_{N_{\text{macro}}}$ is obtained. Finally, one normalises the resulting function $p_{\mathcal{M}}$ such that

$$\sum_{\mathcal{M}=1}^{N_{\text{macro}}} p_{\mathcal{M}} = 1, \tag{2.35}$$

as is required. The final step is to use $p_{\mathcal{M}}$ to obtain an estimate for the ideal weight function. This is done simply by substituting $p_{\mathcal{M}}$ into Eqn. (2.30). We refer to this method for obtaining the weight function as the *shooting method*. The accuracy of the resulting ideal weight function depends on how good our statistics are with regards to the matrix $H_{\mathcal{M}\mathcal{M}'}$ (specifically the elements in the diagonals immediately above and below the main diagonal), which is used to estimate $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$ via Eqn. (2.31), and ultimately $\eta^*_{\mathcal{M}}$.

Methods for exploring phase space

As alluded to above, since the updates to $H_{\mathcal{M}\mathcal{M}'}$ always use the canonical probabilities of transitioning between states, with the transition matrix method one can *choose* how phase space is explored. Effectively the only constraint is that there is fast local equilibration within each macrostate. We will now describe the various methods which can be implemented in *monteswitch*.

The first method is to use multicanonical sampling to explore phase space with an evolving weight function, where the weight function at a given time

is the current estimate for the ideal weight function derived from the current $H_{\mathcal{M}\mathcal{M}'}$ as described above. This is the 'natural' way of applying the transition state method.

The second method is to use what we refer to as *artificial dynamics* to force the system to explore all macrostates in a reasonable amount of time. In this method, the system is first locked into a macrostate for a certain period of time. After that period of time has elapsed, the 'barriers' preventing the system from moving into an adjacent macrostate is moved such that the system is free to transition into an adjacent macrostate. Once this occurs, the system is locked into this new macrostate, and the procedure starts again. There is of course the question of which adjacent macrostate to 'open' to the system. Assuming we are not in macrostate $\mathcal{M} = 1$ or N_{macro}, then there are two options: $(\mathcal{M} + 1)$ and $(\mathcal{M} - 1)$. In *monteswitch* one can specify whether to select the new macrostate at random [30], or whether to sweep through the macrostates systematically, e.g., to explore macrostates $3, 4, 5, \ldots, (N_{\mathrm{macro}} - 1), N_{\mathrm{macro}}, (N_{\mathrm{macro}} - 1), \ldots, 3, 2, 1, 2, 3, \ldots$. This method is faster than the 'natural' method just described because one does not have to wait for the weight function to evolve such that it pushes the system to explore macrostates which are unlikely to be visited in the canonical ensemble.

Chapter 3

Preliminaries

3.1 Compiling *monteswitch*

monteswitch is provided as an archive. This should be extracted to create a directory which consitutes the *monteswitch* package. The package should then be compiled using the Make utility: see the 'README' section in the file `Makefile`, which contains details regarding how to specify the Fortran and MPI compiler and any flags used during compilation, as well as how to compile only the serial (i.e., non-MPI) programs if the user's platform does not contain MPI. The default paths to the Fortran and MPI compilers in `Makefile` are `gfortran` and `mpif90` respectively. For platforms in which these paths apply *monteswitch* can be compiled 'out of the box' by invoking the command

```
$ make serial
```

to compile only serial executables, or

```
$ make mpi
```

to compile *all* executables.

3.2 Transferability between platforms and compilers

monteswitch has not yet been tested on many platforms, and hence there may be unforeseen problems when implementing *monteswitch* on the user's platform. We have overwhelmingly used GFortran and Open MPI compilers with *monteswitch*; hence these compilers can be considered to be the most 'safe'.

One issue we have found with regards to using *monteswitch* on different platforms stems from the fact that *monteswitch* uses Fortran's list-directed input and output. This means the nature of the output from *monteswitch* programs is noticeably compiler-dependent. This in itself is not a problem, though the

post-processing commands given in forthcoming chapters may need to be subtly modified to work with programs compiled using something other than GFortran.

Another issue we have found is that the usage of the utility `sed` is platform-dependent. In particular we have found that appropriate command-line argument to `sed` to invoke in-place editing of a file, without saving a backup, is '`-i ''`' on OS X (note the '`''`'), while it is simply '`-i`' on other Unix platforms (where adding '`''`' results in an error being returned by `sed` on these platforms). This has a bearing on the transferability of some of the `run.sh` shell scripts in the directory `Tests` – which correspond to the OS X version of `sed` – to the user's platform. Hence the user should ensure that these scripts are appropriate for their platform, and modify them accordingly if not.

3.3 Package overview

After compilation, the *monteswitch* directory will contain the following programs:

- `monteswitch`
- `monteswitch_mpi`
- `monteswitch_post`
- `lattices_in_hcp_fcc`
- `lattices_in_bcc_fcc`
- `lattices_in_bcc_hcp`

`monteswitch` and `monteswitch_mpi` are the key programs of the package: they perform Monte Carlo simulations. By contrast `monteswitch_post`, `lattices_in_hcp_fcc`, `lattices_in_bcc_fcc` and `lattices_in_bcc_hcp` are utility programs: `monteswitch_post` is for post-processing one of the output files created by the main programs; and `lattices_in_hcp_fcc`, `lattices_in_bcc_fcc` and `lattices_in_bcc_hcp` are for generating one of the input files for the main programs. We will elaborate upon the function of each of these programs later.

3.3.1 Test cases and examples

There are a number of subdirectories within the package. The directory `Tests` contains a suite of test cases for the main programs. Chapter 8 contains descriptions of the tests, including how to run each test and what results to expect. We recommend that the tests be performed by the user to ensure that the Monte Carlo programs, as compiled by their system, work correctly. Furthermore, these tests are also instructive, providing examples of how to perform various tasks which the typical user would be interested in performing. Finally, the tests

serve to validate the Monte Carlo programs by illustrating that they reproduce known results.

While the test cases serve as examples, more pedagogical examples can be found in the directory `Examples`, including the worked example of Chapter 7.

3.3.2 Specifying the interatomic potential

The directory `Interactions` contains Fortran files corresponding to various interatomic potentials. However, only one of these can be implemented by `monteswitch` at a time. The module which is implemented is stored in the file `interactions.f95` in the main directory; it is only the contents of this file which is used in the package – after compilation. Hence to implement a specific interatomic potential one must copy the desired `interactions.f95` file in `Interactions` (which will be named `interactions_X.f95`, where '`X`' labels the type of interatomic potential the file corresponds to) to `interactions.f95` in the main directory, and then compile the package.

Note that by default the `interactions.f95` is identical to the file `Interactions/interactions_EAM.f95`, and hence compiling *monteswitch* out-of-the-box yields a version which implements EAM potentials.

Further information regarding interatomic potentials is provided in Chapter 5.

3.4 Source code documentation

HTML documentation for the source code can be generated from 'marked-up' comments in the source code files. Invoking the command

```
$ make srcdocs
```

generates this documentation: each HTML file corresponds to a particular source code file.

Chapter 4

Monte Carlo simulation programs

The key programs in the package are `monteswitch` and `monteswitch_mpi`, which run Monte Carlo simulations. The latter is the MPI-parallelised analogue of the former, and the two programs are almost identical in terms of usage. In this chapter we describe the usage of `monteswitch` in detail. Unless otherwise stated, the following information is also pertinent to `monteswitch_mpi`. Information specific to `monteswitch_mpi` can be found in the final section of this chapter.

The command-line arguments passed to `monteswitch` determine the nature of the simulation. Usage of `monteswitch` is:

```
monteswitch [-seed <seed>] -new [-wf]
monteswitch [-seed <seed>] (-resume|-reset)
```

Note that one, and only one, of the arguments `-new`, `-resume` and `-reset` must be specified, while the arguments `-seed` and `-wf` are optional (where moreover `-wf` can only be used in conjunction with `-new`) . Note also that the order of the arguments is important: if `-seed` is specified then it must be the first argument; if `-wf` is specified it must be the last argument. Below we describe the function of each of these arguments.

4.1 Seeding the random number generator: the argument `-seed`

`monteswitch` uses the Mersenne Twister random number generator (MT19937) [31], and a single non-zero integer is used to seed it. The command-line argument `-seed` allows the user to specify the seed for the forthcoming simulation explicitly. If `-seed` is present, then the following argument `<seed>` is taken to be the seed. For example, to set the seed to 12345 for a simulation using the `-new` argument (whose function will be explained in a moment) the appropriate

command is `monteswitch -seed 12345 -new`. If the argument `-seed` is absent then a seed is generated using the system clock.

4.2 Starting a new simulation: the argument `-new`

The command-line argument `-new` invokes a new simulation 'from scratch'. In this case the simulation is initialised using information contained in input files located in the current directory. The specific input files required by `monteswitch` will depend on the specific version of `interactions.f95` utilised when compiling *monteswitch* (see Section 3.3.2). However the input files which are version-dependent only contain information pertaining to the interatomic potential. We defer discussion of such files to Chapter 5; here we focus on the input files which are used in *all* versions of `monteswitch`, namely `lattices_in`, `params_in` and `wf_in`. The first two of these are compulsory: they are read by all new simulations. By contrast `wf_in` is optional, only being read if the command-line argument `-wf` is present (see above for usage). We now describe the format and purpose of these files.

4.2.1 Input file: `lattices_in`

The input file `lattices_in` contains specifications for two states (i.e., supercell dimensions and particle positions), one for each phase. These two states are the reference states described in Section 2.3. Recall that they serve two purposes. Firstly, they act as prospective initial state for the simulation: if the system is to be initialised in phase 1, then the phase-1 state will be used as the initial state, and similarly if the system is to be initialised in phase 2. Secondly they determine the nature of the lattice switch; see Section 2.3 for details. The format of `lattices_in` is as follows. The 1st line is a comment line, and is ignored by the main programs. The 2nd line contains the number of particles N in both phases (and must be the same for both phases). The 3rd, 4th and 5th lines contain the dimensions of the supercell for phase 1 in the x-, y- and z-directions respectively. The next N lines contain the species and positions of the particles within the supercell for phase 1 *in fractional coordinates*: each line contains the fractional x-, y- and z-coordinates for the particle, followed by an integer which signifies its species. (A particle's species potentially determines how it interacts with other particles; *monteswitch* allows different interatomic potentials between sets of particles belonging to different species. The specifics of this are determined by the file `interactions.f95` – see Chapter 5). The remaining lines in `lattices_in` similarly specify the state for phase 2: the next 3 lines contains the dimensions of the supercell for phase 2 in the x-, y- and z-directions respectively; and the next N lines contain the fractional positions and species of the particles for phase 2.

Below is an example of a `lattices_in` file corresponding to phase 1 being an 8-particle bcc supercell and phase 2 being an 8-particle hcp supercell, where

the phase-1 state consists entirely of particles belonging to species '1'and the phase-2 state consists of a mixture of species '1' and '2'. Further examples of `lattices_in` files can be found in the `Tests` and `Examples` directories.

```
bcc-hcp, rho = 0.5, nx,ny,nz = 1, 1, 2
  8
2.2449241
1.5874012
4.4898482
  0.0000000   0.0000000   0.0000000   1
  0.5000000   0.5000000   0.0000000   1
  0.5000000   0.0000000   0.2500000   1
  0.0000000   0.5000000   0.2500000   1
  0.0000000   0.0000000   0.5000000   1
  0.5000000   0.5000000   0.5000000   1
  0.5000000   0.0000000   0.7500000   1
  0.0000000   0.5000000   0.7500000   1
2.4494897
1.4142136
4.6188021
  0.0000000   0.0000000   0.0000000   1
  0.5000000   0.5000000   0.0000000   2
  0.3333333   0.0000000   0.2500000   1
  0.8333333   0.5000000   0.2500000   2
  0.0000000   0.0000000   0.5000000   1
  0.5000000   0.5000000   0.5000000   2
  0.3333333   0.0000000   0.7500000   1
  0.8333333   0.5000000   0.7500000   2
```

Bearing in mind our description of the lattice switch utilised in *monteswitch* given in Section 2.3, it can be seen that the above file results in the following lattice switch from phase 1 to 2:

- A scaling of the x-dimension of the supercell by a factor of $\mathcal{L}_x^{(2)}/\mathcal{L}_x^{(1)} = 2.2449/2.2449 = 1$, i.e., not at all.
- A scaling of the y-dimension of the supercell by a factor of $\mathcal{L}_y^{(2)}/\mathcal{L}_y^{(1)} = 1.414/1.587 = 0.891$.
- A scaling of the z-dimension of the supercell by a factor of $\mathcal{L}_z^{(2)}/\mathcal{L}_z^{(1)} = 4.619/4.490 = 1.029$.
- A transformation of the fractional position of the underlying lattice site for particle i from the ith fractional position specified in the file for phase 1 to the ith fractional position specified for phase 2. E.g. the underlying lattice site for particle 3 has its fractional position $(0.5000000, 0.0000000, 0.2500000)$ transformed to $(0.3333333, 0.0000000, 0.2500000)$.

- A transformation of the species of particles 2, 4, 6 and 8 from '1' to '2'.

The lattice switch from phase 2 to 1 is the inverse of the above transformation.

4.2.2 Input file: `params_in`

The input file `params_in` contains the variables which determine the nature of the simulation. Each variable corresponds to a specific single line in the file, and each line must consist of an arbitrary string (we recommend the name of the variable followed immediately by an '=' character with no spaces), followed by whitespace, followed by the value of the variable. The variables which must appear in a `params_in`, as well as a description of their function, are given in Table 4.2.2. *Note that the variables must appear in params_in in the order that they appear in the table.* E.g., the first variable must be **init_lattice**, the second must be **M_grid_size**, etc. Examples of `params_in` files can be found in the `Tests` and `Examples` directories, which can serve as templates for the user. Note that by 'move' we mean one of either a: particle move, in which one particle is moved to generate a trial microsate; a lattice move, in which the lattice is switched; and a volume move, in which the unit cell itself is altered. Furthermore if the move is rejected then it is still deemed to have taken place. For an NVT ensemble with lattice moves enabled the following move 'cycle' is performed: particle move, lattice move. For an NPT ensemble with lattice moves enabled the move cycle is: particle move, lattice move, (volume move, lattice move), where the set of moves in the brackets occur on average **vol_freq** times per sweep, where a 'sweep' consists of N move cycles, where N is the number of particles in the system.

Table 4.1: Control variables for `monteswitch`, and descriptions of their functions. These variables must be specified in the `params_in` file, one per line, in the order specified in the table. 'Line' refers to the line number in `params_in` on which the corresponding variable must appear.

Line	Variable	Type	Description
1	**init_lattice**	`INTEGER`	Starting phase for the simulation (1 or 2).
2	**M_grid_size**	`INTEGER`	Number of macrostates to divide the considered order parameter range (**M_grid_min** to **M_grid_max**) into.
3	**M_grid_min**	`REAL`	Minimum of considered order parameter range. Note that moves which take the system outwith the considered range are automatically rejected. In other words, the system is constrained to have an order parameter between **M_grid_min** and **M_grid_max**. Hence if one wants to perform a simulation in which the order parameter is unconstrained, one should set **M_grid_min** and **M_grid_max** to values which would never be realised by the system, e.g., -1×10^{100} and 1×10^{100} respectively. This would be the case if one wishes to use `monteswitch` to perform a conventional Monte Carlo simulation.
4	**M_grid_max**	`REAL`	Maximum of considered order parameter range. (See also comments for **M_grid_min**).
5	**enable_multicanonical**	`LOGICAL`	`T` enables multicanonical sampling using the current weight function; `F` enables canonical sampling.
6	**beta**	`REAL`	Inverse temperature: $\beta = 1/(k_B T)$, where k_B is the Boltzmann constant.

7	**P**	`REAL`	Pressure (only relevant in NPT ensemble simulations).
8	**enable_lattice_moves**	`LOGICAL`	`T` enables lattice switch moves (performed after evey particle and volume move).
9	**enable_part_moves**	`LOGICAL`	`T` enables particle moves. This should probably be set to `T`; however disabling particle moves and enabling volume moves may be useful for 'relaxing' the volume of the system before simulations with particle moves are performed.
10	**enable_vol_moves**	`LOGICAL`	`T` enables volume moves and selects the NPT ensemble; `F` selects the NVT ensemble. A volume move will be attempted on average **vol_freq** times per sweep.
11	**part_select**	`CHARACTER(30)`	Flag determining how the next particle to move is selected: `"cycle"` selects particles sequentially, `"rand"` selects particles at random.
12	**part_step**	`REAL`	Particle move maximum size; particles are moved according to a random walk, with a maximum move size of **part_step** in any Cartesian direction.
13	**enable_COM_frame**	`LOGICAL`	`T` performs the simulation in the centre-of-mass reference frame; `F` uses the lab frame. Using the centre-of-mass frame prevents 'drift' in the centre-of-mass, which is convenient because it keeps particles close to their lattice sites. However there is a computational overhead associated with using the centre-of-mass frame.

14	**vol_dynamics**	`CHARACTER(30)`	Flag determining which type of volume moves are performed: `"FVM"` (fixed volume move) keeps the supercell shape unchanged during a volume move; `"UVM"` (unconstrained volume move) allows the x-, y- and z-dimensions to move independently.
15	**vol_freq**	`INTEGER`	Number of volume moves performed per sweep on average if volume moves are enabled. We recommend that this be set to 1.
16	**vol_step**	`REAL`	Volume move maximum step size; the volume is moved according to a random walk in '$\ln(V)$-space', with a maximum move size of **vol_step**.
17	**stop_sweeps**	`INTEGER`	Total number of sweeps to perform in the simulation. If this is set to 0 then no Monte Carlo moves are performed, but tasks performed periodically during the Monte Carlo loop (i.e, updating the weight function, checking whether or not the system has melted, checking for 'divergences' in the energy, calculating equilibrium quantities and their uncertainties) are performed.
18	**equil_sweeps**	`INTEGER`	Number of sweeps to disregard before the system is considered to be equilibrated; statistics are not gathered during these sweeps for block averaging (see below).

19	**enable_melt_checks**	`LOGICAL`	`T` enables periodic checks of whether the system has 'melted', i.e., if one or more of the particles has moved more than a distance of **melt_threshold** from its lattice site in any Cartesian direction then the system is considered to have 'melted'. We recommend that this feature is used, since the LSMC method relies upon the fact that the two phases under c onsideration are metastable, and that particles will not move too far from their lattice sites on the timescale of the simulation.
20	**melt_sweeps**	`INTEGER`	Period (sweeps) to check for melting.
21	**melt_threshold**	`REAL`	See **enable_melt_checks**.
22	**melt_option**	`CHARACTER(30)`	Flag determining what the simulation does if the system has 'melted': `zero_1` and `zero_2` move the system to the zero-displacement states in phases 1 and 2, respectively; `zero_current` does the same but for the current phase; `stop` stops the simulation with an exit status of 2. For `zero_1`, `zero_2`, `zero_current` the system is allowed to re-equilibrate before statistics are gathered for bock averaging. Also, the current block is disregarded for the purposes of block averaging.

23	**enable_divergence_checks**	LOGICAL	T enables periodic checks of whether the energy of the system is correct. To elaborate, for particle moves, the energy of the trial state is not calculated exactly because it is very computationally expensive and unnecessary. Instead the energy *change* with respect to the current state is calculated. This is far less demanding to calculate. If the move is accepted this change is *amended* to the total energy. However, over time it is possible that this 'running total' approach will yield incorrect energies due to the finite precision of the computer. Hence one should periodically recalculate the energy exactly. This is done during volume moves. If **enable_divergence_checks** is set to T, then this is also done every **divergence_sweeps** sweeps, after which the recalculated energy is compared to the 'current' energy, and the simulation is stopped with an exit status of 3 if they are different – outwith a tolerence of **divergence_tol**. Note that the order parameter is also ammended after the energy (for phases 1 and 2) is recalculated. We recommend that this feature is used in NVT simulations, since it forces the energy of the system to be recalculated from scratch every so often. For NPT simulations this should not be an issue since the energy is calculated from scratch every accepted volume move.
24	**divergence_sweeps**	INTEGER	Period (sweeps) to check for 'energy divergences' as just mentioned. Note that checking entails recalculating the energy from scratch.

25	**divergence_tol**	REAL	See **enable_divergence_checks**.
26	**output_file_period**	INTEGER	Period (sweeps) at which information is output to the file `data`. See Section 4.3.
27	**output_file_Lx**	LOGICAL	See Section 4.3.
28	**output_file_Ly**	LOGICAL	See Section 4.3.
29	**output_file_Lz**	LOGICAL	See Section 4.3.
30	**output_file_V**	LOGICAL	See Section 4.3.
31	**output_file_R_1**	LOGICAL	See Section 4.3.
32	**output_file_R_2**	LOGICAL	See Section 4.3.
33	**output_file_u**	LOGICAL	See Section 4.3.
34	**output_file_lattice**	LOGICAL	See Section 4.3.
35	**output_file_E**	LOGICAL	See Section 4.3.
36	**output_file_M**	LOGICAL	See Section 4.3.
37	**output_file_eta**	LOGICAL	See Section 4.3.
38	**output_file_moves_lattice**	LOGICAL	See Section 4.3.
39	**output_file_accepted_moves_lattice**	LOGICAL	See Section 4.3.
40	**output_file_moves_part**	LOGICAL	See Section 4.3.
41	**output_file_accepted_moves_part**	LOGICAL	See Section 4.3.
42	**output_file_moves_vol**	LOGICAL	See Section 4.3.
43	**output_file_accepted_moves_vol**	LOGICAL	See Section 4.3.
44	**output_file_rejected_moves_M_OOB**	LOGICAL	See Section 4.3.
45	**output_file_M_OOB_high**	LOGICAL	See Section 4.3.
46	**output_file_M_OOB_low**	LOGICAL	See Section 4.3.
47	**output_file_barrier_macro_low**	LOGICAL	See Section 4.3.
48	**output_file_barrier_macro_high**	LOGICAL	See Section 4.3.
49	**output_file_rejected_moves_M_barrier**	LOGICAL	See Section 4.3.
50	**output_file_moves_since_lock**	LOGICAL	See Section 4.3.
51	**output_file_melts**	LOGICAL	See Section 4.3.

52	**output_file_equil_DeltaF**	LOGICAL	See Section 4.3.
53	**output_file_sigma_equil_DeltaF**	LOGICAL	See Section 4.3.
54	**output_file_equil_H_1**	LOGICAL	See Section 4.3.
55	**output_file_sigma_equil_H_1**	LOGICAL	See Section 4.3.
56	**output_file_equil_H_2**	LOGICAL	See Section 4.3.
57	**output_file_sigma_equil_H_2**	LOGICAL	See Section 4.3.
58	**output_file_equil_V_1**	LOGICAL	See Section 4.3.
59	**output_file_sigma_equil_V_1**	LOGICAL	See Section 4.3.
60	**output_file_equil_V_2**	LOGICAL	See Section 4.3.
61	**output_file_sigma_equil_V_2**	LOGICAL	See Section 4.3.
62	**output_file_equil_umsd_1**	LOGICAL	See Section 4.3.
63	**output_file_sigma_equil_umsd_1**	LOGICAL	See Section 4.3.
64	**output_file_equil_umsd_2**	LOGICAL	See Section 4.3.
65	**output_file_sigma_equil_umsd_2**	LOGICAL	See Section 4.3.
66	**output_file_equil_L_1**	LOGICAL	See Section 4.3.
67	**output_file_sigma_equil_L_1**	LOGICAL	See Section 4.3.
68	**output_file_equil_L_2**	LOGICAL	See Section 4.3.
69	**output_file_sigma_equil_L_2**	LOGICAL	See Section 4.3.
70	**output_stdout_period**	INTEGER	Period (sweeps) at which information is output to stdout. See Section 4.3.
71	**output_stdout_Lx**	LOGICAL	See Section 4.3.
72	**output_stdout_Ly**	LOGICAL	See Section 4.3.
73	**output_stdout_Lz**	LOGICAL	See Section 4.3.
74	**output_stdout_V**	LOGICAL	See Section 4.3.
75	**output_stdout_R_1**	LOGICAL	See Section 4.3.
76	**output_stdout_R_2**	LOGICAL	See Section 4.3.
77	**output_stdout_u**	LOGICAL	See Section 4.3.
78	**output_stdout_lattice**	LOGICAL	See Section 4.3.

79	**output_stdout_E**	LOGICAL	See Section 4.3.
80	**output_stdout_M**	LOGICAL	See Section 4.3.
81	**output_stdout_eta**	LOGICAL	See Section 4.3.
82	**output_stdout_moves_lattice**	LOGICAL	See Section 4.3.
83	**output_stdout_accepted_moves_lattice**	LOGICAL	See Section 4.3.
84	**output_stdout_moves_part**	LOGICAL	See Section 4.3.
85	**output_stdout_accepted_moves_part**	LOGICAL	See Section 4.3.
86	**output_stdout_moves_vol**	LOGICAL	See Section 4.3.
87	**output_stdout_accepted_moves_vol**	LOGICAL	See Section 4.3.
88	**output_stdout_rejected_moves_M_OOB**	LOGICAL	See Section 4.3.
89	**output_stdout_M_OOB_high**	LOGICAL	See Section 4.3.
90	**output_stdout_M_OOB_low**	LOGICAL	See Section 4.3.
91	**output_stdout_barrier_macro_low**	LOGICAL	See Section 4.3.
92	**output_stdout_barrier_macro_high**	LOGICAL	See Section 4.3.
93	**output_stdout_rejected_moves_M_barrier**	LOGICAL	See Section 4.3.
94	**output_stdout_moves_since_lock**	LOGICAL	See Section 4.3.
95	**output_stdout_melts**	LOGICAL	See Section 4.3.
96	**output_stdout_equil_DeltaF**	LOGICAL	See Section 4.3.
97	**output_stdout_sigma_equil_DeltaF**	LOGICAL	See Section 4.3.
98	**output_stdout_equil_H_1**	LOGICAL	See Section 4.3.
99	**output_stdout_sigma_equil_H_1**	LOGICAL	See Section 4.3.
100	**output_stdout_equil_H_2**	LOGICAL	See Section 4.3.
101	**output_stdout_sigma_equil_H_2**	LOGICAL	See Section 4.3.
102	**output_stdout_equil_V_1**	LOGICAL	See Section 4.3.
103	**output_stdout_sigma_equil_V_1**	LOGICAL	See Section 4.3.
104	**output_stdout_equil_V_2**	LOGICAL	See Section 4.3.
105	**output_stdout_sigma_equil_V_2**	LOGICAL	See Section 4.3.
106	**output_stdout_equil_umsd_1**	LOGICAL	See Section 4.3.

107	**output_stdout_sigma_equil_umsd_1**	LOGICAL	See Section 4.3.
108	**output_stdout_equil_umsd_2**	LOGICAL	See Section 4.3.
109	**output_stdout_sigma_equil_umsd_2**	LOGICAL	See Section 4.3.
110	**output_stdout_equil_L_1**	LOGICAL	See Section 4.3.
111	**output_stdout_sigma_equil_L_1**	LOGICAL	See Section 4.3.
112	**output_stdout_equil_L_2**	LOGICAL	See Section 4.3.
113	**output_stdout_sigma_equil_L_2**	LOGICAL	See Section 4.3.
114	**checkpoint_period**	INTEGER	Period (sweeps) at which the simulation is checkpointed, i.e., how often all simulation variables are output to the file `state`. If **checkpoint_period** is ≤ 0 then `state` will be an empty file. See Section 4.3.
115	**update_eta**	LOGICAL	`T` results in the weight function being periodically updated every **update_eta_sweeps** sweeps, according to the method specified in **update_eta_method**; `F` results in the weight function not being updated – it remains frozen at its current state.
116	**update_eta_sweeps**	INTEGER	Period (sweeps) at which the weight function is updated.
117	**update_trans**	LOGICAL	`T` results in the transition matrix being updated; `F` results in it not being updated.
118	**update_eta_method**	CHARACTER(30)	Method used to update the weight function: `"VS"` uses the visited states method; `"shooting"` uses the shooting method (using the current transition matrix). (See Section 2.6).
119	**enable_barriers**	LOGICAL	`T` enables artificial dynamics; for `F` the system is free to explore any macrostate, but is constrained to reside within the considered order parameter range (**M_grid_min** to **M_grid_max**). (See Section 2.6).

120	**barrier_dynamics**	`CHARACTER(30)`	Flag determining how the macrostate barriers will evolve during artificial dynamics. All methods lock the system into a single macrostate for **lock_moves** moves, before unlocking an adjacent macrostate. Once the system has moved into the adjacent macrostate, the system is then locked into that macrostate, and the procedure starts again. `"random"` evolves the macrostate the system is locked into via a random walk: the next macrostate is decided with equal probability to be that above or that below the current macrostate. `"pong_up"` moves to increasingly higher macrostates until the upper limit of the supported order parameter range is encountered, at which point it reverses direction and proceeds to increasingly lower macrostates until it reaches the lower limit of the order parameter range, at which point it reverses direction, etc. `"pong_down"` instead moves initially to increasingly lower macrostates.
121	**lock_moves**	`INTEGER`	The number of moves to lock the system into one macrostate for if artificial dynamics is used. This should be greater than 0.
122	**calc_equil_properties**	`LOGICAL`	`T` enables internal calculation of various physical quantities, and associated uncertainties, via block averaging. (See Section 4.3).

123	**block_sweeps**	`INTEGER`	The number of sweeps which comprise a 'block' which will be used to evaluate physical quantities and their associated uncertainties via block averaging (see Section 4.3). This should be greater than 0.

4.2.3 Input file: `wf_in`

The command-line argument `-wf` allows one to specify the initial weight function to be used in the simulation: if `-wf` is present, then the initial weight function is read from the file `wf_in`. If `-wf` is absent then `wf_in` is not read, and the weight function is initialised to 0 for all macrostates. `wf_in` must contain **M_grid_size** lines (where **M_grid_size** is specified in `params_in`), each containing two tokens (extra lines and tokens are ignored), which both should be of type `REAL`. The first token on each line is ignored, while the second tokens are treated as the weight function: the value of the weight function for macrostate i is initialised to the value of the second token on line i in `wf_in`. Note that the format of `wf_in` is analogous to that output by the program `monteswitch_post` in conjunction with the `-extract_wf` argument – see Section 6.2.

4.3 Output: stdout, `state` and `data`

During a simulation,information is periodically output to stdout and the file `data`. The variables with names beginning with **output_file_** and **output_stdout_** determine which variables are output to `data` and stdout respectively. The variable **output_file_X**, where **X** is the name of a simulation variable (a list is given below), when set to `T`, will result in a line consisting of `X:` followed by the number of completed sweeps, followed by the current value of **X** being printed to `data` every **output_file_period** sweeps. However, if **output_file_period** is set to 0, then instead the output is after every move; and if **output_file_period** is a negative integer, then there is no output to the file, i.e., the output is suppressed. The above also applies to **output_stdout_X**, but for the output to stdout. The only exceptions to the aforedescribed significance of variables beginning with **output_file_** and **output_stdout_** are **output_file_eta** and **output_stdout_eta**. For these quantities **eta** refers to the value of the weight function *for the current state* – as opposed to the entire weight function. This point is reiterated below. Note that there is no internal variable named **eta** in `monteswitch`: the entire weight function is refered to as **eta_grid** in `monteswitch`; **eta** is calculated on-the-fly when required from **eta_grid** and never stored in memory.

The file `data` can be used to deduce how the system evolves with time during the simulation. For instance, one can use the information contained within `data` to check whether or not the system has equilibrated within a certain number of Monte Carlo sweeps. In a production simulation it can also be used to store the 'observations' performed on the system, which could be used in subsequent analysis.

In addition to `data`, a file `state` is also created by the program periodically throughout a simulation. This file contains all of the simulation variables, and can be used to 'resume' the simulation by running `monteswitch` with the `-resume` or `-reset` argument (see Sections 4.4 and 4.5), or to extract the 'results' of the simulation, e.g., equilibrium quantities, the current weight function,

the number of accepted vs. rejected Monte Carlo moves of a certain type – perhaps using the `monteswitch_post` program (see Section 6.2). The variable **checkpoint_period** determines how often the simulation is checkpointed: a file `state` is created every **checkpoint_period** sweeps. It is also *always* created at the completion of the program. Note that if **checkpoint_period** is ≤ 0, then there is no output to `state`, i.e., the file will be empty. In the `state` file, similarly to `params_in`, each line corresponds to a particular variable: for variable X the corresponding line contains `X=` followed by the value of that variable. The relevant line can be extracted from `state` using the utility `grep`. E.g. the command

```
$ grep "E=" state
```

can be used to extract the current energy of the system from `state`. Note that `state` contains *all* the simulation variables, including those pertaining to the interatomic potential. Hence the format of `state` depends on the details of the `interactions.f95` file. However, all variables pertaining to the interatomic potential are stored at the end of `state`. Hence the format of `state` files down to the point where the variables pertaining to the interatomic potential are stored is universal.

Table 4.3.1 contains a list of `monteswitch` internal variables, not already covered by Table 4.2.2, which have corresponding **output_file_** or **output_stdout_** flags, or can be found in `state` and could possibly be of interest to the user. Note that there are other variables in `state` which are not mentioned in Tables 4.2.2 or 4.3.1 – aside from those stemming from `interactions.f95`. We do not mention them because these are not likely to be of interest to the user; for more information see the HTML documentation for `monteswitch_mod.f95`. Note that the internal variable names output in `state` are case-sensitive, and exactly reflect the names given in Tables 4.2.2 and 4.3.1.

4.3.1 Block averaging

`monteswitch` has the ability to use block averaging to provide estimates of various physical quantities on-the-fly during a simulation. This feature is activated by setting the variable **calc_equil_properties** to `T`. In block averaging one calculates an estimate for some arbitrary physical quantity X, which we denote as $\bar{X}$, as well as an associated uncertainty in $\bar{X}$, which we denote as σ_X. $\bar{X}$ and σ_X are calculated as follows. After the equilibration period, the simulation time is partitioned into N_{blocks} 'blocks' each consisting of an equal number of moves. For each block b we obtain an estimate for X by averaging over the values of X corresponding to all states visited during that block. Let X_b denote the estimate for X for block b obtained in this manner. Now, by taking the average of the values of X_b over all blocks, one can obtain an even more accurate estimate for X. This is $\bar{X}$:

$$\bar{X} = \frac{1}{N_{\text{blocks}}} \sum_{b=1}^{N_{\text{blocks}}} X_b. \tag{4.1}$$

The uncertainty σ_X is simply the standard error in $\bar{X}$:

$$\sigma_X = \frac{1}{\sqrt{N_{\text{blocks}}}} \sqrt{\frac{1}{(N_{\text{blocks}} - 1)} \sum_{b=1}^{N_{\text{blocks}}} (X_b - \bar{X})^2}. \tag{4.2}$$

In `monteswitch` the internal variable **block_sweeps** determines the size of a block in sweeps. Internal variables with names beginning with '**equil_**' or '**sigma_equil_**' contain the results of block averaging. To elaborate, **equil_X** corresponds to $\bar{X}$ and **sigma_equil_X** corresponds to σ_X calculated by `monteswitch` for physical quantity X. The variables with this format can be found in Table 4.3.1. These variables are updated every time a block is 'completed' by the simulation after the equilibration time. Note that the variable **block_counts_X** contains the number of blocks which have been used so far in evaluating **equil_X** and **sigma_equil_X**. Hence **equil_X** will be meaningless (i.e., uninitialised) if **block_counts_X** takes the value 0, and **sigma_equil_X** will be meaningless if **block_counts_X** is less than 2 (in which case there have not yet been enough blocks to evaluate σ_X). Blocks may be 'disregarded' by `monteswitch`, i.e., not used in the evaluation of **equil_X** and **sigma_equil_X** for a number of reasons. Firstly, a block is disregarded if the system was deemed to have 'melted' during the block (see **enable_melt_checks** in Table 4.2.2). Secondly, a block is disregarded if during the block the system never visits the phase or phases necessary to calculate X. For instance if X is the volume of phase 1, then X can only be evaluated if phase 1 is actually visited during the block. Accordingly in this case a block in which phase 1 is never visited is disregarded for the purposes of calculating X. Another important example is if X is the free energy difference between the two phases. In this case X can only be evaluated if both phases are visited during the block. Hence the block is disregarded in this case if only one phase is visited during the block. The user should be aware that all blocks are given equal weight in the block averaging. This could lead to misleading results for, say, quantities pertaining to phase 1, if in some blocks phase 1 is rarely visited and in other blocks phase 1 is often visited. This problem can be avoided by ensuring that the block size is larger than the correlation time of the simulation, in which case phase 1 would be visited approximately equally often for all blocks.

Table 4.2: Useful variables found in the file `state`.

Variable	Fortran type	Description
n_part	`INTEGER`	Number of particles in the system.
Lx	`REAL(2)`	Dimension of supercells in x-direction: the first value pertains to phase 1 while the second pertains to phase 2.
Ly	`REAL(2)`	Dimension of supercells in y-direction: the first value pertains to phase 1 while the second pertains to phase 2.
Lz	`REAL(2)`	Dimension of supercells in z-direction: the first value pertains to phase 1 while the second pertains to phase 2.
V	`REAL`	Current volume of the system.
lattice	`INTEGER`	Current phase of the system (1 or 2).
E_1	`REAL`	Energy of phase 1 for the current displacements.
E_2	`REAL`	Energy of phase 2 for the current displacements.
E	`REAL`	Current energy of the system. This is **E_1** if we are in phase 1 and **E_2** if we are in phase 2.
M	`REAL`	Current order parameter of the system.
switchscalex	`REAL`	Scaling of the supercell in the x-dimension when performing a lattice switch from phase 1 to phase 2. The reciprocal of this is the scaling when performing a lattice switch from phase 2 to phase 1.
switchscaley	`REAL`	Scaling of the supercell in the x-dimension when performing a lattice switch from phase 1 to phase 2. The reciprocal of this is the scaling when performing a lattice switch from phase 2 to phase 1.

switchscalez	`REAL`	Scaling of the supercell in the x-dimension when performing a lattice switch from phase 1 to phase 2. The reciprocal of this is the scaling when performing a lattice switch from phase 2 to phase 1.
sweeps	`INTEGER`	Number of sweeps performed so far, including over previous simulations if we have used the `-resume` argument.
moves	`INTEGER`	Total number of moves performed so far in total, including over previous simulations if we have used the `-resume` argument.
moves_lattice	`INTEGER`	Number of lattice moves performed so far, including over previous simulations if we have used the `-resume` argument.
accepted_moves_lattice	`INTEGER`	Number of accepted lattice moves so far, including over previous simulations if we have used the `-resume` argument.
moves_part	`INTEGER`	Number of particle moves performed so far, including over previous simulations if we have used the `-resume` argument.
accepted_moves_part	`INTEGER`	Number of accepted particle moves so far, including over previous simulations if we have used the `-resume` argument.
moves_vol	`INTEGER`	Number of volume moves performed so far, including over previous simulations if we have used the `-resume` argument.
accepted_moves_vol	`INTEGER`	Number of accepted volume moves so far, including over previous simulations if we have used the `-resume` argument.

rejected_moves_M_OOB	INTEGER	Number of moves rejected because the order parameter of the trial state was outwith the considered range ('OOB' means 'out of bounds'), i.e., **M_grid_min** to **M_grid_max**.
M_OOB_high	INTEGER	The highest order parameter value to be rejected because the order parameter of the trial state was outwith the considered range.
M_OOB_low	INTEGER	The lowest order parameter value to be rejected because the order parameter of the trial state was outwith the considered range.
melts	INTEGER	The number of times the system has melted.
barrier_macro_low	INTEGER	The macrostate number corresponding to the lowest currently allowed macrostate (relevant only when artificial dynamics is enabled).
barrier_macro_high	INTEGER	The macrostate number corresponding to the highest currently allowed macrostate (relevant only when artificial dynamics is enabled).
rejected_moves_M_barrier	INTEGER	The number of moves rejected because the order parameter of the trial state was outwith the range corresponding to the macrostate barriers (relevant only when artificial dynamics is enabled).
block_counts	INTEGER	The total number of blocks considered so far for block averaging (see Section 4.3.1.
equil_DeltaF	REAL	The free energy difference between the phases ($F_1 - F_2$; extensive) evaluated using block averaging (see Section 4.3.1).
sigma_equil_DeltaF	REAL	The uncertainty in **equil_DeltaF** (see Section 4.3.1).

block_counts_DeltaF	INTEGER	The number of blocks used in evaluating **equil_DeltaF** and **sigma_equil_DeltaF**. Note that blocks are disregarded if during the block the system melts, or if the system does not visit both phases during the block (see Section 4.3.1).
equil_H_1	REAL	The energy (for NVT simulations) or enthalpy (for NPT simulations) of phase 1 evaluated using block averaging (see Section 4.3.1).
equil_H_2	REAL	The energy (for NVT simulations) or enthalpy (for NPT simulations) of phase 2 evaluated using block averaging (see Section 4.3.1).
sigma_equil_H_1	REAL	The uncertainty in **equil_H_1** (see Section 4.3.1).
sigma_equil_H_2	REAL	The uncertainty in **equil_H_2** (see Section 4.3.1).
block_counts_H_1	INTEGER	The number of blocks used in evaluating **equil_H_1** and **sigma_equil_H_1**. Note that blocks are disregarded if during the block the system melts, or if the system does not visit phase 1 during the block (see Section 4.3.1).
block_counts_H_2	INTEGER	The number of blocks used in evaluating **equil_H_2** and **sigma_equil_H_2**. Note that blocks are disregarded if during the block the system melts, or if the system does not visit phase 2 during the block (see Section 4.3.1).
equil_V_1	REAL	The volume of phase 1 evaluated using block averaging (see Section 4.3.1).
equil_V_2	REAL	The volume of phase 2 evaluated using block averaging (see Section 4.3.1).
sigma_equil_V_1	REAL	The uncertainty in **equil_V_1** (see Section 4.3.1).

sigma_equil_V_2	`REAL`	The uncertainty in **equil_V_2** (see Section 4.3.1).
block_counts_V_1	`INTEGER`	The number of blocks used in evaluating **equil_V_1** and **sigma_equil_V_1**. Note that blocks are disregarded if during the block the system melts, or if the system does not visit phase 1 during the block (see Section 4.3.1).
block_counts_V_2	`INTEGER`	The number of blocks used in evaluating **equil_V_2** and **sigma_equil_V_2**. Note that blocks are disregarded if during the block the system melts, or if the system does not visit phase 2 during the block (see Section 4.3.1).
R_1	`REAL(n_part,3)`	The current lattice vectors for phase 1.
R_2	`REAL(n_part,3)`	The current lattice vectors for phase 2.
u	`REAL(n_part,3)`	The current displacement vectors.
M_grid	`REAL(M_grid_size)`	Array containing the minimum order parameter for each macrostate: macrostate n corresponds to order parameters between **M_grid**(n) and **M_grid**$(n+1)$.
M_counts_1	`INTEGER(M_grid_size)`	**M_counts_1**(n) is the number of times macrostate n has been visited while the system was in phase 1 so far, including over previous simulations if we have used the `-resume` argument.
M_counts_2	`INTEGER(M_grid_size)`	**M_counts_2**(n) is the number of times macrostate n has been visited while the system was in phase 2 so far, including over previous simulations if we have used the `-resume` argument.
eta_grid	`REAL(M_grid_size)`	**eta_grid**(n) is the value o the weight function for macrostate n.

trans	`REAL(M_grid_size,M_grid_size)`	**eta_grid**(m, n) is the number of inferred transitions from macrostate m to macrostate n; it is the matrix $H_{\mathcal{M}\mathcal{M}'}$ in Section 2.6.2.
equil_umsd_1	`REAL(n_part)`	**equil_umsd_1**(n) is the mean-squared displacement of particle n from its lattice site in phase 1, evaluated using block averaging (see Section 4.3.1).
equil_umsd_2	`REAL(n_part)`	**equil_umsd_2**(n) is the mean-squared displacement of particle n from its lattice site in phase 2, evaluated using block averaging (see Section 4.3.1).
sigma_equil_umsd_1	`REAL(n_part)`	**sigma_equil_umsd_1**(n) is the uncertainty in **equil_umsd_1**(n) (see Section 4.3.1).
sigma_equil_umsd_2	`REAL(n_part)`	**sigma_equil_umsd_2**(n) is the uncertainty in **equil_umsd_2**(n) (see Section 4.3.1).
block_counts_umsd_1	`INTEGER`	The number of blocks used in evaluating **equil_umsd_1** and **sigma_equil_umsd_1**. Note that blocks are disregarded if during the block the system melts, or if the system does not visit phase 1 during the block (see Section 4.3.1).
block_counts_umsd_2	`INTEGER`	The number of blocks used in evaluating **equil_umsd_2** and **sigma_equil_umsd_2**. Note that blocks are disregarded if during the block the system melts, or if the system does not visit phase 2 during the block (see Section 4.3.1).
equil_L_1	`REAL(3)`	The 1st, 2nd and 3rd values in the array **equil_L_1** are the mean x-, y- and z-dimensions of the supercell in phase 1, evaluated using block averaging (see Section 4.3.1). Note that blocks are disregarded if during the block the system melts.

equil_L_2	REAL(3)	The 1st, 2nd and 3rd values in the array **equil_L_2** are the mean x-, y- and z-dimensions of the supercell in phase 2, evaluated using block averaging (see Section 4.3.1). Note that blocks are disregarded if during the block the system melts.
sigma_equil_L_1	REAL(3)	**sigma_equil_L_1**(n) is the uncertainty in **equil_L_1**(n) (see Section 4.3.1).
sigma_equil_L_2	REAL(3)	**sigma_equil_L_2**(n) is the uncertainty in **equil_L_2**(n) (see Section 4.3.1).
block_counts_L_1	INTEGER	The number of blocks used in evaluating **equil_L_1** and **sigma_equil_L_1**. Note that blocks are disregarded if during the block the system melts, or if the system does not visit phase 1 during the block (see Section 4.3.1).
block_counts_L_2	INTEGER	The number of blocks used in evaluating **equil_L_2** and **sigma_equil_L_2**. Note that blocks are disregarded if during the block the system melts, or if the system does not visit phase 2 during the block (see Section 4.3.1).

4.4 Resuming a simulation: the argument `-resume`

The command-line argument `-resume` continues an 'old' simulation, whose variables are contained in the file `state` in the current directory. The 'resumed' simulation is run for the number of Monte Carlo sweeps specified in the variable **stop_sweeps** in `state`. By default this is the number of sweeps which were performed in the old simulation, though one of course this can be manually altered if one wants the resumed simulation to be of a different length to the old simulation.

For a simulation invoked using the argument `-resume`, the file `data` is ammended: the resumed simulation does not overwrite the `data` file; all information from the old simulation is retained in it.

4.5 'Resetting' a simulation: the argument `-reset`

The command-line argument `-reset` invokes a simulation from an old `state` file similarly to `-resume`, except that it resets all 'counter variables' to zero. This has the effect of starting a 'new' simulation whose nature corresponds to the old simulation, but instead uses the state of the system specified in `state`. By contrast, the argument `-new` initialises the state to be such that the particles form a perfect crystal lattice, which usually does not correspond to an equilibrated state. By 'counter variables' we mean those such as variables describing the number of moves performed for each move type, the number of accepted moves for each move type, and variables pertaining to equilibrium quantities. However, note that the weight function (**eta_grid**) and the matrix (**trans**) are *not* regarded as counter variables, and are as such retained from the old simulation if one uses the `-reset` argument.

Note that for a simulation invoked using the argument `-reset`, the file `data` is overwritten, i.e., the information from the 'old' simulation is not retained.

4.6 Exit statuses

`monteswitch` exits with a non-zero exit status of 1 for most errors. However, if the system has melted, and **melt_option** is set to 'stop', then the exit status is 2, and if the energy has diverged from its true value then the exit status is 3. Note though that exit statuses are not part of the Fortran standard, and may not work for all operating systems or compilers.

4.7 MPI simulations: `monteswitch_mpi`

As mentioned at the beginning of this chapter, the program `monteswitch_mpi` is the MPI-parallelised analogue of `monteswitch`. It is identical to the program `monteswitch`, except that instead of a single simulation for **stop_sweeps** Monte Carlo sweeps, `monteswitch_mpi` runs n simulations – replicas – in parallel using MPI, each being approximately **stop_sweeps**$/n$ sweeps in length. Accordingly, during the simulation multiple `data`- and `state`-format files are created – one for each replica. These are named `state_0`, `state_1`, `state_2`, etc., and similarly for the `data`-format files. At the completion of all replicas, the results of block averaging from all replicas are combined and stored in the file `state`. The state stored in `state` corresponds to the state in `state_0`. The user should bear in mind that `state`-format files can be large, and hence one may need to ensure that storage limits are not exceeded when invoking `monteswitch_mpi` with a large number of replicas n.

Further details of the parallelisation are as follows. All replicas are always initialised to be in the same state. For a new simulation this is determined by the **init_lattice** variable in `params_in` similarly to `monteswitch`. For a resumed simulation this is the state contained in the `state` file from which the simulation is to be resumed. We emphasise that *all replicas of the system are initialised with the same state* when the `-resume` argument is used with `monteswitch_mpi`. The files `data_`n are therefore always overwritten by `monteswitch_mpi` when the `-resume` argument is invoked since, given the nature of the parallelisation, there is no continuity between the replicas in subsequent simulations. The excpetion is replica '0', whose state is always stored in the file `state`, as well as `state_0`.

To expand upon what was just said, in a resumed simulation, only replica 0 – the 'master' replica – inherits the counter variables from the previous simulation via the file `state`. All other replicas have counter variables initialised to zero at the start of the resumed simulation. During the simulation, the counter variables for each replica are of course ammended according to the evolution of that replica. Once all replicas have completed their allocated number of sweeps, the variables for the nth replica are exported to the file `state_`n. Then the counter variables of all replicas, other than replica 0, are summed and ammended to replica 0. At the end of the simulation replica 0 thus has variables which corresponding to an evolution of, if there are N tasks, approximately **stop_sweeps**$/N$ sweeps from the starting state, but with counter variables which correspond to **stop_sweeps** sweeps worth of information. The variables in replica 0 are then updated to reflect its new counters, e.g., the weight function is recalculated using the information from the new counters. Finally, the variables are exported to the file `state`.

Regarding the seeding of the random number generator for each replica, `monteswitch_mpi` takes steps to ensure that the seeds are different for each replica. To elaborate, if a seed is specified via the command-line argument `-seed`, then this seed is used as the 'parent seed' for the simulation. The parent seed is used to derive different seeds for each replica via a simple function. If the argument `-seed` is not present, then the seed for each replica is

initially taken from the system clock, before being transformed in a manner which ensures that no two replicas have the same seed. See the source code of `monteswitch_mpi.f95` or its documentation for more details.

Chapter 5

Interatomic potentials

As mentioned in Section 3.3.2, the file `interactions.f95` determines the interatomic potential implemented within *monteswitch*. The `Interactions` directory contains files which correspond to a number of interatomic potentials. Recall that copying one of these files to `interactions.f95` in the main directory, and then compiling *monteswitch*, results in a version of *monteswitch* in which the particles interact via the corresponding interatomic potential. For example, copying `interactions_LJ.f95` to `interactions.f95` and then compiling *monteswitch* results in a version of *monteswitch* in which the particles interact via a Lennard-Jones potential. In this chapter we first describe the potentials implemented by each of the `interactions.f95` included with *monteswitch* in the `Interactions` directory. We also describe the input files for `monteswitch` and `monteswitch_mpi` which must be in conjunction with each in order to specify the free parameters for the interatomic potential under consideration. Then in Section 5.3 we describe how users can write their own `interactions.f95` files to implement their own potentials.

Note that most `interactions.f95` files included with *monteswitch* are applicable only for single-species systems. The exception is `interactions_HS_multi.f95`, which can treat multiple-component hard-sphere mixtures. Furthermore, the potentials included with *monteswitch* all use a single input file named `interactions_in` for `monteswitch` and `monteswitch_mpi` to specify the potential's free parameters. In time we expect to include more interatomic potentials with *monteswitch*.

5.1 Embedded atom model

The file `interactions_EAM.f95` implements the embedded atom model (EAM) [32]. Here the energy of the system is given by

$$E = \frac{1}{2} \sum_{i,j \neq i} \phi(r_{ij}) + \sum_i F(\rho_i), \tag{5.1}$$

$$\rho_i = \sum_{j \neq i} \rho(r_{ij}), \tag{5.2}$$

where r_{ij} is the separation between particles i and j, and ϕ, F and ρ are functions which must be specified and constitute the parametrisation of the EAM potential. If `interactions_EAM.f95` is used then the `interactions_in` file must be a description of the potential in DYNAMO/LAMMPS 'setfl' format. A file with this format is often indicated with the suffix '`.eam.alloy`', and a description of the format can be found at `http://lammps.sandia.gov/doc/pair_eam.html`.

We emphasise that `interactions_EAM.f95` currently only supports single-component systems, and hence `interactions_in` must not correspond to a multicomponent system. An error is returned if this is the case. Furthermore, at initialisation three additional files are created by the Monte Carlo programs in *monteswitch* if `interactions_EAM.f95` is used: `F.dat`, `rho.dat` and `rphi.dat`. `F.dat` and `rho.dat` contain plots of the functions F and ρ read from `interactions_in`, while `rphi.dat` contains a plot of $r \times \phi(r)$ vs. r read from `interactions_in`.

By default `interactions.f95` is the same as `interactions_EAM.f95`.

5.2 Soft pair potentials

Table 5.2 gives a list of soft pair potentials included with *monteswitch*, and the name of the corresponding `interactions.f95` file in `Interactions`. All of these potentials are implemented in the same way. Firstly, the pair potential $\phi(r)$ is assumed to be 0 for inter-particle separations r greater than some cut-off distance r_c. In other words the potential is *truncated* at r_c. Secondly, only pairs of particles within a distance r_list of each other *in the reference states of the simulation* interact with each other throughout the entire simulation. To elaborate, a list of particles which particle i interacts with in phase p is created at the start of a simulation. This list consists of all those particles which are initially within distance r_list of i in the reference state for phase p. Note that *monteswitch* stores two sets of lists, one for each phase, and hence it is possible that particle i interacts with different sets of particles in different phases. We emphasise that the same same cut-off r_list is used for both phases. To clarify the difference between r_c and r_list: the former is the distance at which the potential is truncated, while the latter determines which pairs of particles 'see each other' throughout the simulation. Normally one would set $r_\mathrm{c} > r_\mathrm{list}$.

The format of the `interactions_in` files for these potentials is similar to `params_in` described in Section 4.2.2: each variable which parametrises the potential corresponds to a specific line in `interactions_in`, and each line must contain a string (we recommend the name of the variable followed immediately by an '=' character with no spaces), followed by whitespace, followed by the value of the variable. The final column of Table 5.2 gives the order of variables as they should appear, one per line, in `interactions_in` for each potential. In this column the names of the variables are those used by *monteswitch*, as opposed

to those used in the preceding mathematical expression for the potential $\phi(r)$. Where it is not obvious to which variable in the mathematical expression a variable in the last column relates, the analogous variable in the mathematical expression is given in parenthesis immediately after it. In all cases the variables **cutoff** and **list_cutoff** correspond to r_c and r_{list} described above. **list_size** is the size of the array which stores the list of particles which any particle in the system interacts with. For particle i, the list includes all particles within r_{list} of particle i, *including particle i itself*, in the reference state for phase p. All of these particles are stored in the array. However for technical reasons the array must have at least one additional element. Thus **list_size** should be at least the largest number of particles within r_{list} of any particle i (including i itself) over the reference states for both phases, +1.

To illustrate the above, here is an example `interactions_in` file for the Lennard-Jones potential (`interactions_LJ.f95`):

```
lj_epsilon= 1.0
lj_sigma= 1.0
lj_cutoff= 1.5
list_cutoff= 1.000000001
list_size= 14
```

This file is designed for a LSMC simulation in which phase 1 is hcp and phase 2 is fcc, both with nearest neighbour distance $r_{nn} = 1.0$, in which we wish only nearest neighbours to interact with each other interact via a Lennard-Jones potential with $\epsilon = \sigma = 1$ truncated at distance $r_c = 1.5$. We have specified that only nearest neighbours ever interact by setting **list_cutoff** to be just above the nearest neighbour distance. Furthermore we have set **list_size** to 14 because there are 12 nearest neighbours of any particle, which means that there are 13 particles within a distance r_{nn} of any particle i including particle i itself, and, as mentioned above, we must add 1 to this for technical reasons. Note that 14 is the minimal value of **list_size** suitable for this system. However one could set **list_size** to be greater than 14. The result would be that the array which stores the list is larger than necessary, and accordingly some of the array would remain 'unused'. This becomes a problem if one sets **list_size** to a massive value, resulting in the array taking up a lot of memory unnecessarily.

Table 5.1: Soft interatomic pair potentials included with *monteswitch*

Potential	File name	Expression for potential	Order of variables in `interactions_in`
12-10	`interactions_12-10.f95`	$A/r^{12} - B/r^{10}$	**A**, **B**, **cutoff**, **list_cutoff**, **list_size**
12-6	`interactions_12-6.f95`	$A/r^{12} - B/r^{6}$	**A**, **B**, **cutoff**, **list_cutoff**, **list_size**
Buckingham	`interactions_Buckingham.f95`	$A\exp(-r/\rho) - C/r^6$	**A**, **rho**, **C**, **cutoff**, **list_cutoff**, **list_size**
Gaussian	`interactions_Gaussian.f95`	$-A\exp(-Br^2)$	**A**, **B**, **cutoff**, **list_cutoff**, **list_size**
Lennard-Jones	`interactions_LJ.f95`	$4\epsilon\left[(\sigma/r)^{12} - (\sigma/r)^6\right]$	**lj_epsilon** (ϵ), **lj_sigma** (σ), **cutoff**, **list_cutoff**, **list_size**
9-6 Lennard-Jones	`interactions_LJ_9-6.f95`	$4\epsilon\left[(\sigma/r)^{9} - (\sigma/r)^6\right]$	**lj_epsilon** (ϵ), **lj_sigma** (σ), **cutoff**, **list_cutoff**, **list_size**
Morse	`interactions_Morse.f95`	$E_0\left\{\left[1-\exp\left(-k(r-r_0)\right)\right]^2 - 1\right\}$	**E0**, **k**, **r0**, **cutoff**, **list_cutoff**, **list_size**
n-m	`interactions_n-m.f95`	$\left[E_0/(n-m)\right]\left[m(r_0/r)^n - n(r_0/r)^m\right]$	**E0**, **npot** (n), **mpot** (m), **r0**, **cutoff**, **list_cutoff**, **list_size**
Yukawa	`interactions_Yukawa.f95`	$A\exp(-kr)/r$	**A**, **k**, **cutoff**, **list_cutoff**, **list_size**

5.3 User-defined pair potentials

The file `interactions_TEMPLATE_pair.f95` is a template which can be used to easily create `interactions.f95` files for user-defined pair potentials. In fact this template was used to create the files for almost all of the potentials described in Section 5.2. Instructions are provided in the file regarding how the file should be modified to realise the user's potential of interest. The resulting implementation of the potential, and the required format of the `interactions_in` file, will be the same as described in Section 5.2. Regarding the order in which the variables which parametrise the potential are read from `interactions_in`, the user-defined variables are read first, and then the variables **cutoff**, **list_cutoff**, **list_size** are read.

Note that one is by no means limited to using `interactions_TEMPLATE_pair.f95` to create 'analytical' pair potentials. For instance piecewise pair potentials could be created from `interactions_TEMPLATE_pair.f95`, e.g., the soft sphere model:

$$\phi(r) = \begin{cases} \epsilon(\sigma/r)^n & \text{if } r \leq \sigma \\ 0 & \text{if } r > \sigma. \end{cases} \tag{5.3}$$

5.4 Hard/penetrable spheres

There are two implementations of the penetrable (including hard) spheres model included with *monteswitch*. `interactions_HS.f95` implements the penetrable spheres model for single-species systems only; while `interactions_HS_multi.f95` supports multiple-species systems. The latter file supercedes the former, but we retain the former for legacy reasons – some of the test cases included with *monteswitch* utilise `interactions_HS.f95`. We now describe both implementations.

5.4.1 `interactions_HS.f95`

In `interactions_HS.f95` the pair potential is given by

$$\phi(r) = \begin{cases} \epsilon & \text{if } r < \sigma \\ 0 & \text{otherwise,} \end{cases} \tag{5.4}$$

where ϵ is the energy cost of a pair of spheres overlaping, and σ is the diameter of the spheres. The format for `interactions_in` here, and the implementation of the potential, is similar to the analytical pair potentials described earlier in Section 5.2; the order of variables as they should occur in `interactions_in` is **epsilon**, **sigma**, **list_cutoff**, then **list_size**, where the significance of **list_cutoff** and **list_size** is the same as for the analytical pair potentials.

Note that hard spheres are realised in the limit $\beta\epsilon \to \infty$, where $\beta = 1/(k_B T)$. Thus hard spheres can be modeled by setting either ϵ or β, or both, to a very

high value. We recommend setting $\beta = 1$ and $\epsilon = 100000$ to realise hard spheres. In this case ϵ is not so high that numerical issues occur, but also not so small that moves which lead to overlapping spheres – which are always possible unless ϵ is actually ∞ – are likely to happen during even a very long simulation.

It is noteworthy that the order parameter M has a straightforward physical interpretation for penetrable spheres: if the system is currently in phase 1(2) then $-(+)M/\epsilon$ is the number of overlapping spheres which result from performing a lattice switch from the current state. This is similar to the definition of M used in the original LSMC papers (Refs. [2, 3]).

5.4.2 `interactions_HS_multi.f95`

In `interactions_HS_multi.f95` the sphere diameter is allowed to vary with species. Here, the pair potential between two particles belonging to species s and t is given by

$$\phi_{st}(r) = \begin{cases} \epsilon & \text{if } r < \frac{1}{2}(\sigma_s + \sigma_t) \\ 0 & \text{otherwise,} \end{cases} \tag{5.5}$$

where σ_s denotes the diameter of spheres belonging to species s. The format of the `interactions_in` and the implementation of the potential is similar to `interactions_HS.f95` described above, except an additional `INTEGER` variable **n_species** is introduced, which specifies the number of different species in the system, and **sigma** is now a `REAL(n_species)` array which contains the hard sphere diameters for each species. Note that the ith value in **sigma** is the diameter for species i: the species labels range from 1 to **n_species**, e.g., species '0' or '-2' are not used). The order of variables as they should occur in `interactions_in` is **epsilon**, **n_species**, **sigma**, **list_cutoff**, then **list_size**.

5.5 General user-defined potentials

The file `interactions_TEMPLATE_minimal.f95` is a template which can be built upon by users to implement their own potentials in *monteswitch*. The template includes empty Fortran procedures which must be filled in by the user. Comments within the template act as a guide for the user, who may also find it useful to examine the `interactions.f95` files included with *monteswitch* in the `Interactions` directory as examples. The user is of course free to add their own functions and variables within their custom `interactions.f95` file. Furthermore they can use external Fortran modules in the usual manner. Note that there is no universal procedure for determining the separation between two particles in *monteswitch*. Separations are required only in that they are used to determine the energy of the system, something is within the remit of the file `interactions.f95`. Hence when making a custom `interactions.f95` file it is up to the user to calculate the separations between pairs of particles themselves, possibly by adding their own procedure which does this (which takes into account the periodic boundary conditions).

We emphasise that there is considerable flexibility with regards to potentials which can be implemented within an `interactions.f95` file. Possibilities include different Hamiltonians for each phase, position-dependent 'external' potentials, and many-body potentials. However recall that *monteswitch* is limited to 'atomic' systems, i.e., particles cannot have rotational degrees of freedom.

5.6 Other potentials included with *monteswitch*

Some other `interactions.f95` files are included in the directory `Interactions`, namely `interactions_EC.f95`, `interactions_EC_NPT.f95`, `interactions_LJ_hcp_fcc.f95`. These other files correspond to potentials which we do not anticipate will be of any interest to most users, and are used in the test cases used to validate *monteswitch* which are described in Chapter 8. Hence we do not describe them here. Interested users should inspect the files themselves for further information, as well as Chapter 8.

Chapter 6

Utility programs

In this chapter we describe the utility programs included in the package, which assist with post-processing of the data and the construction of input files.

6.1 Programs for generating `lattices_in` files

The programs `lattices_in_hcp_fcc`, `lattices_in_bcc_fcc` and `lattices_in_bcc_hcp` create `lattices_in` files containing reference states corresponding to, respectively, hcp–fcc, bcc–fcc and bcc–hcp lattice switches. Usage of these programs is:

```
lattices_in_hcp_fcc <rho> <nx> <ny> <nz>
lattices_in_bcc_fcc <rho> <nx> <ny> <nz>
lattices_in_bcc_hcp <rho> <nx> <ny> <nz>
```

The command-line arguments `<rho>`, `<nx>`, `<ny>` and `<nz>` constitute the free parameters for the `lattices_in` file to be created. The first argument `<rho>` is the density (i.e., the number of particles per unit volume) of the reference states to construct. (Note that both states necessarily have the same density). The second, third and fourth arguments `<nx>`, `<ny>` and `<nz>` are integers which correspond to the number of unit cells (described in a moment) which will be tiled in the x-, y- and z-directions respectively to construct the supercell for each phase.

The programs output the desired `lattices_in` file to stdout. Hence one must redirect the output to create the required `lattices_in` file. For example, to generate a `lattices_in` file for hcp–fcc corresponding to a density of 0.25, consisting of 2, 3 and 5 unit cells tiled in the z-, y-, and z-directions, the command is (assuming the `lattices_in_hcp_fcc` executable is in the current directory)

```
$ ./lattices_in_hcp_fcc 0.5 2 3 5 > lattices_in
```

Note that the `lattices_in` files created by these programs have the species of all particles set to '1'.

What follows is a description of the unit cells for each pair of phases for each program. More specific information regarding the unit cells can be obtained by invoking the programs with `<nx>`, `<ny>` and `<nz>` all set to 1.

6.1.1 `lattices_in_hcp_fcc`

The unit cell here contains 12 particles. The particles are spread over 6 planes in the z-direction; each plane contains two particles. The positions of the particles corresponds to a stacking sequence for the planes of ABCABC for the fcc unit cell, and ABABAB for the hcp unit cell. Note that the unit cell is far longer in the z-directon than the x- and y-directions. Hence one should normally use more unit cells in the x- and y-directions than the z-direction to construct the supercells. In the output of this program phase 1 corresponds to hcp and phase 2 corresponds to fcc.

6.1.2 `lattices_in_bcc_fcc`

The unit cell here is the conventional 2-particle body-centred tetragonal (bct) unit cell; for the bcc(fcc) lattice the relative dimensions of the bct unit cell in each Cartesian direction correspond to the bct representation of the bcc(fcc) lattice. In the output of this program phase 1 corresponds to bcc and phase 2 corresponds to fcc.

6.1.3 `lattices_in_bcc_hcp`

The unit cell here contains 4 particles. The bcc unit cell is the 4-particle face-centred tetragonal (fct) corresponding to the fct representation of the bcc lattice. The hcp unit cell is the 'fct-like' representation of the hcp lattice. In the output of this program phase 1 corresponds to bcc and phase 2 corresponds to hcp.

6.2 Post-processing `state` files: `monteswitch_post`

`monteswitch_post` is for post-processing a file `state` generated by `monteswitch` or `monteswitch_mpi`, and can be used to extract useful information from that file. The `state` file which the program operates on is that in the current directory. The command-line arguments determine the task performed by the program. Usage of `monteswitch_post` is as follows, where the function of each command-line argument is described below:

```
monteswitch_post -extract_wf
monteswitch_post -extract_M_counts
monteswitch_post -extract_pos [<species>]
```

```
monteswitch_post -extract_R_1 [<species>]
monteswitch_post -extract_R_2 [<species>]
monteswitch_post -extract_u [<species> <phase>]
monteswitch_post -calc_rad_dist <bins>
monteswitch_post -merge_trans <state_in_1> <state_in_2> <state_out>
monteswitch_post -extract_lattices_in <vectors_in_1> <vectors_in_2>
monteswitch_post -extract_pos_xyz
monteswitch_post -set_wf <wf_file>
```

6.2.1 -extract_wf

Extract the weight function from `state` and output it to stdout. In the output the first token on each line is the order parameter, and the second is the corresponding value of the weight function.

6.2.2 -extract_M_counts

Extract order parameter histograms from `state` and output them to stdout. In the output the first token on each line is the order parameter, the second is the corresponding number of counts for phase 1, and the third is the corresponding number of counts for phase 2.

6.2.3 -extract_pos [<species>]

Extract the current positions of the particles, and output them to stdout. In the output the first, second and third tokens on each line are the x-, y- and z-coordinates respectively for a particle. If the optional argument `<species>` is present then only the positions for particles belonging to species `<species>` are output.

6.2.4 -extract_R_1 [<species>]

Extract the current positions of the lattice sites for phase 1, and output them to stdout. In the output the first, second and third tokens on each line are the x-, y- and z-coordinates respectively for a particle. If the optional argument `<species>` is present then only the sites for particles belonging to species `<species>` in phase 1 are output.

6.2.5 -extract_R_2 [<species>]

Extract the current positions of the lattice sites for phase 2, and output them to stdout. In the output the first, second and third tokens on each line are the x-, y- and z-coordinates respectively for a particle. If the optional argument `<species>` is present then only the sites for particles belonging to species `<species>` in phase 2 are output.

6.2.6 -extract_u [<species> <phase>]

Extract the displacements of the particles, and output them to stdout. In the output the first, second and third tokens on each line are the x-, y- and z-displacements respectively for a particle. If the optional arguments <species> and <phase> are present then only the displacements for particles belonging to species <species> in phase <phase> are output.

6.2.7 -calc_rad_dist <bins>

Calculate the radial distribution function, based on the current state, and output it to stdout. In the output, each line corresponds to a distance, which is the first token, and the second token contains the average number of particles at this distance from a particle. The output is like a histogram, with each line corresponding to a bin, the first token corresponding to the minimum of the range covered by the bin, and the second token corresponding to the number of counts for that bin. The range of the bin is inclusive at its minimum, and exclusive at its maximum. The upper distance considered for the radial distribution function is the lowest of $L_x/2$, $L_y/2$ and $L_z/2$, where L_x denotes the length of the current supercell in the x-direction, and similarly for L_y and L_z; and the number of bins for the histogram is specified in the second argument <bins>. Note that the upper distance corresponds to the 'limit of periodicity' for the system.

6.2.8 -merge_trans <state_in_1> <state_in_2> <state_out>

Combine the **trans** matrices from the files <state_in_1> and <state_in_2>, and store the result in the file <state_out>, where all variables in <state_out> other than the matrix **trans** are inherited from <state_in_1>. Note that **M_grid_size** must be the same for both input files (in which case the matrices **trans** for each are of the same size). This argument can be used for pooling the results of multiple simulations which utilise the same underlying 'order parameter grid' **M_grid_size**.

6.2.9 -extract_lattices_in <vectors_in_1> <vectors_in_2>

Output the geometrical properties of the system in the format of a lattices_in file. The arguments <vectors_in_1> and <vectors_in_2> can be either pos or R. If <vectors_in_1> is pos, then the positions of the particles in phase 1 of the forthcomning lattices_in file will be the positions of the particles in phase 1 in the state file; if <vectors_in_1> is R, then the positions of the particles in phase 1 of the lattices_in file will be the current lattice vectors (i.e., **R_1**) corresponding to phase 1 in the state file. Similar applies for <vectors_in_2> with phase 2.

6.2.10 `-extract_pos_xyz`

Extract the positions of the particles, and output them to stdout in '.xyz' format. In the output the first line contains the number of particles, the second line is a comment line, and the subsequent lines contain the particle positions and species: the first token is the 'element' (set to 'A' for species '1', 'B' for species '2', ..., 'Z' for species 26, and '?' otherwise), and the second, third, fourth and fifth tokens are the x-, y- and z-coordinates respectively.

6.2.11 `-set_wf <wf_file>`

Alters the weight function in `state` to correspond to that specified in the file `<wf_file>`. The format of the file `<wf_file>` must be analogous to the format of the weight function output by this program via the `-extract_wf` argument described above: the file must contain **M_grid_size** lines, each containing two tokens (extra lines and tokens are ignored), which both should be of type `REAL`. The first token on each line is ignored, while the second tokens are treated as the new weight function: the value of the weight function for macrostate i is set to the value of the second token on line i in `<wf_file>`. We recommend care be taken while using this command-line option, since it irretrievably overwrites the weight function in `state`.

Chapter 7

Worked example

We now provide a worked example to elucidate how *monteswitch* can be applied in practice. We investigate the bcc–hcp transition in Zr, modeled using the EAM potential of Ref. [33]. The authors of that study found that the zero-pressure bcc–hcp transition temperature occured at temperature T =1233K. Here we use LSMC to verify this result. To elaborate, we use *monteswitch* to calculate the Gibbs free energy difference ΔG between the bcc and hcp phases of Zr, modeled using the aforementioned potential, at $T = 1233$K and $P = 0$. If this T indeed corresponds to the zero-pressure transition temperature then we should find that $\Delta G = 0$. As well as the transition temperature, Ref. [33] also provides values for the enthalpy change $\Delta H_{\mathrm{hcp}\to\mathrm{bcc}}$ and fractional volume change $\Delta V_{\mathrm{hcp}\to\mathrm{bcc}}/V_{\mathrm{bcc}}$ associated with the transition. These quantities can also be obtained from a *monteswitch* LSMC calculation. Accordingly we compare the values of $\Delta H_{\mathrm{hcp}\to\mathrm{bcc}}$ and $\Delta V_{\mathrm{hcp}\to\mathrm{bcc}}/V_{\mathrm{bcc}}$ obtained from *monteswitch* to those of Ref. [33].

Below we go through the entire procedure we use to obtain ΔG. The procedure includes using artificial dynamics (see Section 2.6.2) to quickly determine a reasonable weight function, and using `monteswitch_mpi` to speed up the calculations via parallelisation (see Section 4.7). Of course, the procedure described here will not be applicable to all situations. For this reason we try to anticipate problems that the user may encounter generally, and provide remarks accordingly.

The relevant `interactions.f95` file for this worked example is `Interactions/interactions_EAM.f95`. Before running these tests one should copy `Interactions/interactions_EAM.f95` to `interactions.f95`, and then compile *monteswitch*. Note that `Interactions/interactions_EAM.f95` implicitly uses eV as the unit of energy and Å as the unit of length, and hence these units will be used throughout this worked example. See Section 5.1 for a detailed description of EAM potentials as applicable to *monteswitch*.

7.1 Overview

The example can be found in the `Examples/EAM_Zr_bcc_hcp` directory of the *monteswitch* package. Within this directory there are a number of subdirectories, each of which corresponds to a different simulation:

- `bcc_preliminary` and `hcp_preliminary` are short *conventional* Monte Carlo simulations of each phase to determine quantities necessary for the forthcoming LSMC simulations;
- `weight_function_generation` is the LSMC simulation to generate the weight function to be used in the forthcoming production simulation (see below);
- `weight_function_verification` is a short LSMC simulation using the weight function obtained from the `weight_function_generation` simulation to ensure that it looks reasonable;
- `production_simulation` is the LSMC simulation used to calculate ΔG, as well as thermodynamic quantities for each phase.

Each of the above subdirectories contains the input and output files for the corresponding simulation – as if the simulation had been run in that subdirectory. Recall that the input files for 'new' `monteswitch` and `monteswitch_mpi` simulations (i.e., simulations invoked using the `-new` argument) are `params_in`, `lattices_in`, and any input files specific to the version of `interactions.f95` file *monteswitch* is used in conjunction with – in this case `interactions_in` (see Section 5.1). Recall also that the output files for a 'new' EAM simulation are `F.dat`, `rho.dat` and `rphi.dat` (see Section 5.1 for a description of these files); `state` and `data` for serial simulations (i.e., using the `monteswitch` program); and additionally `data_0`, `state_0`, `data_1`, `state_1`, `data_2`, `state_2`, etc. for MPI simulations (i.e., using the `monteswitch_mpi` program). The simulations `bcc_preliminary`, `hcp_preliminary` and `weight_function_generation` are 'new' simulations. Note that these simulations all use identical `interactions_in` and `lattices_in` (copies of which can be found in the `Examples/EAM_Zr_bcc_hcp` directory). They differ in that they have different `params_in` files; we have annotated the `params_in` files for each simulation to highlight the salient variable choices.

By contrast the simulations `weight_function_verification` and `production_simulation` are 'resumed' simulations (i.e., simulations invoked using either the `-resume` or `-reset` arguments). Recall that such simulations take an existing `state` file as input for the simulation, which is overwritten periodically throughout and at the completion of the simulation. In subdirectories corresponding to 'resumed' simulations we include both the `state` file used as input to the simulation, as well as the `state` file output at the completion of the simulation. To distinguish between these files we have named the input file `state_start`. To restate, in directories corresponding to

'resumed' simulations, `state_start` is the input file for the simulation, while `state` is the output file.

Finally, each subdirectory contains a number of other files, mainly the result of post-processing performed on the output files for the corresponding simulation. These files, and how they were created, are described in due course.

7.2 The `interactions_in` and `lattices_in` files

Before performing any simulations, one must first construct the input files `interactions_in` and `lattices_in` which specify, respectively, the EAM potential, and the reference states we will use in all simulations. Recall that the `interactions_in` file must be a 'setfl'-format description of the EAM potential of interest. We wish to use the Zr potential of Ref. [33]. It can be seen by inspecting `interactions_in` in the `Examples/EAM_Zr_bcc_hcp` directory – which is the `interactions_in` used for all simulations – that this file corresponds to a 'setfl'-format description of the aforementioned potential as required.

Consider now the `lattices_in` file. Recall that this contains specifications of the reference states for each phase; the particle positions specified in `lattices_in` determine the two phases which will be considered, as well as the nature of the lattice switch move between the phases. Given that we are considering the bcc and hcp phases, we can use the `lattices_in_bcc_hcp` program (see Section 6.1) to generate a `lattices_in` file suitable for our purposes, as opposed to creating it from scratch. We know that the density of both phases is $\approx 0.043\text{Å}^{-3}$ for Zr at the T and P we are considering. Hence we use this as the first argument to `lattices_in_bcc_hcp`. We generated `lattices_in` for use in all simulations using the command (invoked from within the directory `Examples/EAM_Zr_bcc_hcp`)

```
$ ../../lattices_in_bcc_hcp 0.042710367 4 6 4 > lattices_in
```

A copy of the resulting file can be found in the directory `Examples/EAM_Zr_bcc_hcp`. Recall that the second, third and fourth arguments to `lattices_in_bcc_hcp` (4, 6 and 4 in the above command) correspond to how many unit cells to tile in the x-, y- and z-directions to create the supercell for each phase. As can be seen from inspection of the file `Examples/EAM_Zr_bcc_hcp/lattices_in`, our simulations utilise supercells of reasonable size (384 atoms), with dimensions which are greater than twice the cut-off of the EAM potential (i.e., 7.6Å, the last value on the second line of `interactions_in`) as is required to avoid finite size effects associated with 'missing' neighbours.

7.3 Preliminary simulations

Before performing LSMC simulations, it is necessary to first perform preliminary conventional Monte Carlo simulations for each phase under consideration, in

this case bcc and hcp. The directories `bcc_preliminary` and `hcp_preliminary` correspond to these simulations. The aim of these simulations is to determine:

1. appropriate Monte Carlo maximum particle and volume move sizes;
2. an estimate for the equilibration time for the simulation;
3. the appropriate order-parameter range to use in subsequent simulations.

7.3.1 `params_in` files

The salient features of the `params_in` files for these simulations are as follows:

- **init_lattice** = 1 for the bcc simulation and 2 for the hcp simulation, since in the `lattices_in` file phase 1 corresponds to bcc and phase 2 corresponds to hcp.
- **M_grid_min** and **M_grid_max** are set to be extremely low and extremely high values respectively. This is necessary because the simulation automatically constrains the system such that its order parameter lies between **M_grid_min** and **M_grid_max**; setting them to very low and high values respectively allows the system to have total freedom in order-parameter space – which is what we desire for a single phase conventional Monte Carlo simulation. This should always be done if one wishes to perform a conventional Monte Carlo simulation using *monteswitch*.
- **enable_multicanonical** and **enable_lattice_moves** are set to `F` for a conventional Monte Carlo simulation. Note that by disabling lattice switch moves the system is 'locked in' to the phase in which it is initialised.
- **enable_vol_moves** is set to `T`, in which case the simulation corresponds to the NPT ensemble.
- **part_step** and **vol_step** are the maximum particle and volume move sizes. One of the aims of the preliminary simulations is to determine a reasonable value for these variables. The values in the `params_in` files are 0.3 and 0.03 respectively. These were obtained by trial and error. To elaborate, we started the simulation with a guess for these values, and examined the information output to stdout as the simulation was running, specifically the total and accepted number of particle and volume moves (discussed in more detail in a moment). If **part_step** or **vol_step** are too high then the number of accepted particle or volume moves will be very low relative to the total number of moves; if they are too low then the number of accepted particle or volume moves will be close to the total number of moves. If either is the case we simply stopped the simulation, and restarted it again with a better guess for **part_step** or **vol_step**. The whole process does not take very long since it becomes obvious very quickly whether the acceptance rates are reasonable. In principle of course one could automate the process of determining the appropriate values of **part_step** and **vol_step**.

- **stop_sweeps** is set to 1000; the simulation will consist of 1000 sweeps, which is very short, though sufficient (in this system, though not in general) to obtain an estimate of 1., 2. and 3. listed above.

- **output_file_period** is set to 1, which means information is output to the `data` file every sweep. Normally outputing information so frequently would be excessive, but we need such high-resolution information regarding the time evolution of the simulation to determine the equilibration time.

- **output_file_V**, **output_file_E** and **output_file_M** are set to `T`: the system's volume, energy and order parameter are periodically output to the file `data`. We will examine the time-evolution of these quantities to deduce an estimate for the equilibration time, as well as the range of order parameters exhibited by each phase at equilibrium. The latter will in turn inform the values of **M_grid_min** and **M_grid_max** to be used in subsequent LSMC simulations.

- **output_stdout_period** is set to 10: information is output to stdout every 10 sweeps. The information we are interested in while the simulation is running, for reasons mentioned above, is the acceptance rates of the particle and volume moves for the chosen values of **part_step** and **vol_step**. Accordingly we set **output_stdout_moves_part**, **output_stdout_accepted_moves_part**, **output_stdout_moves_vol** and **output_moves_accepted_moves_vol** to `T`. With the corresponding variables one can calculate the acceptance rate for the chosen **part_step** and **vol_step** (acceptance rate = number of accepted moves / total moves)

- Finally, **enable_barriers** is set to `F`. Otherwise artificial dynamics would be used, something we don't want to realise a conventional Monte Carlo simultion.

7.3.2 Running the simulations

Each of the preliminary simulations is run using the `-new` argument. Given that each simulation is so short, it is unnecessary to use MPI: the `monteswitch` executable should be used. The following command invokes the executable (from within the `bcc_preliminary` or `hcp_preliminary` directories):

```
$ ../../../monteswitch -new
```

7.3.3 Results

Plots of the system's energy, volume and order parameter vs. simulation time (where by simulation 'time' here we mean the number of completed Monte Carlo sweeps) can be created by post-processing the file `data` output by each preliminary simulation using the following commands (invoked in each of the directories `bcc_preliminary` and `hcp_preliminary`):

```
$ grep 'E:' data | awk '{print $2,$3}' > E_vs_t.dat
$ grep 'V:' data | awk '{print $2,$3}' > V_vs_t.dat
$ grep 'M:' data | awk '{print $2,$3}' > M_vs_t.dat
```

Fig. 7.1 shows `E_vs_t.dat`, `V_vs_t.dat` and `M_vs_t.dat` for each phase. As can be seen from the figure the system equilibrates quickly: within 200 sweeps. We will use this observation later. Fig. 7.1 also reveals that for bcc the order-parameter range exhibited at equilibrium in the preliminary simulations is about -72 to -22, while for hcp it is about 15 to 42. Recall (see Section 2.5) that we wish to use an order-parameter range which is 'large enough to cover all states which are reasonably likely to occur at thermodynamic equilibrium for both phases, as well as the gateway states'. Hence an order-parameter range of -72 to 42 would seem appropriate. However, our preliminary simulations are relatively short, and hence only provide *estimates* of the order-parameter range likely to be exhibited by each phase at equilibrium. With this in mind we use a slightly larger range than the -72 to 42 observed in the preliminary simulations, namely -82 to 48, to be on the safe side. But we still cannot be certain that this range is large enough. Fortunately though we can check the validity of the chosen range *a postiori* using the results of later simulations (see Section 7.4.3 below).

7.4 Weight function generation

The preliminary simulations have provided us with appropriate values for the maximum particle and volume move sizes, an estimate for the equilibration time, and an appropriate order-parameter range. With this information we can now begin LSMC simulations. The first such simulation is to generate the weight function. The directory `weight_function_generation` corresponds to this simulation. We use artificial dynamics to generate the weight function. Furthermore we use `monteswitch_mpi` to perform a parallelised calculation. Specifically we used 4 threads on a 4-core desktop machine. The following command was used to invoke the simulation (from within the `Examples/weight_function_generation` within the *monteswitch* package itself) on our desktop machine:

```
$ mpiexec -n 4 ../../../monteswitch_mpi -new
```

The appropriate command may be different for the user's platform.

7.4.1 The `params_in` file

The salient features of the `params_in` files for this simulation are as follows:

- **init_lattice** is set to 1, which means that the simulation is initialised in the bcc phase. However, the choice of initial phase is unimportant since both phases will be explored during the simulation.

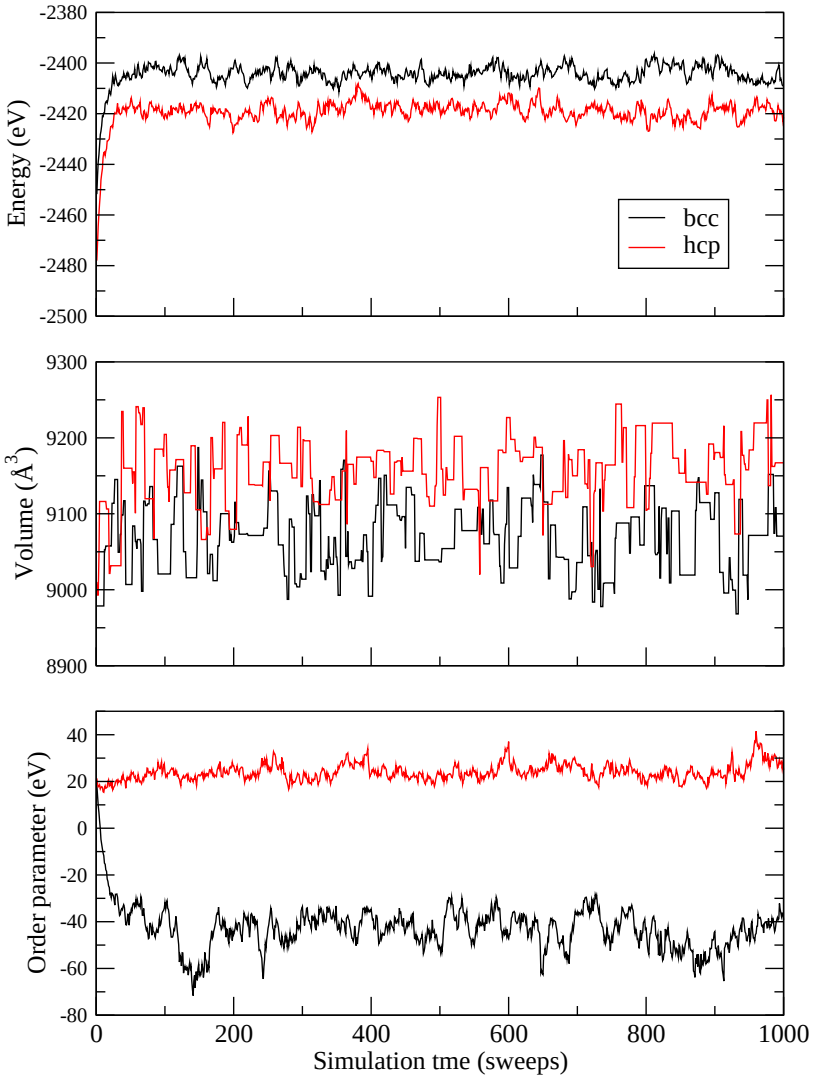

Figure 7.1: Plots of the system's energy (top panel), volume (middle panel) and order parameter (bottom panel) vs. simulation time for the preliminary simulations. The black curves correspond to the bcc preliminary simulation, while the red curves correspond to the hcp simulation.

- **M_grid_size**, **M_grid_min** and **M_grid_max** define the macrostates. We set **M_grid_min** and **M_grid_max** to -82 and 48, which corresponds to the appropriate range determined from our preliminary simulations. Furthermore, we set **M_grid_size** to 100: the order-parameter range will be divided into 100 macrostates. The choice of 100 is based upon our previous experience, and may not be appropriate for all systems. Recall that if the number of macrostates is too high, then the weight function will take longer to generate. On the other hand if the number of macrostates is too low, then the system is unable to explore the whole range of order-parameter space in a reasonable simulation time – regardless of the weight function. This occurs because the order-parameter grid is too coarse to be able to guide the system over any free energy barriers. (See Section 2.5).

- **enable_multicanonical** is set to `T`, which means that the continually updated weight function is used to bias the dynamics of the system. This is actually unnecessary here since we elect to use artificial dynamics to guide the system through order-parameter space. Artificial dynamics would work well with canonical sampling. However, using multicanonical sampling in theory should result in the system transitioning between adjacent macrostates more easily, speeding up the exploration of order-parameter space slightly.

- **enable_lattice_moves** is set to `T`: the system can explore both phases using lattice moves.

- **part_step** and **vol_step** are informed by our preliminary simulations

- **stop_sweeps** is set to 160000. Note that since we are using 4 MPI threads, each thread will perform 40000 sweeps.

- **output_file_period** is set to 250. We do this mainly to avoid creating massive output files. At this point we do not need to know information regarding the time-evolution of the system during the simulation on the scale of less than 250 sweeps, given that the equilibration time is $\lesssim 200$ sweeps. Note that since we are using `monteswitch_mpi` there is one output file for each thread; for each thread n, every 250 sweeps information is output to the file `data_n`.

- **output_stdout_period** is set to -1, which suppresses all output to stdout. This is normally desirable for MPI simulations, since each thread will itself output to stdout every **output_stdout_period** sweeps, which can result in an overwhelming amount of (and confusing) information to stdout. Furthermore, for long simulations we have no interest in watching the simulation variables during the running of the simulation – which is the primary purpose of the simulation outputting information to stdout. Instead we let the simulation run, say, overnight, and extract the information we require from the `data` and `state` files once it is completed –

no information is output to stdout which cannot be obtained from these files.

- **checkpoint_period** is set to 2000; every 2000 sweeps the state of each MPI thread n is output to the file `state_n`.
- **update_eta** is set to `T`, which means that the simulation periodically updates/generates the weight function
- **update_eta_sweeps** is set to 2000; we update the weight function (in this case using the shooting method, see below) every 2000 sweeps.
- **update_eta_method** is set to `"shooting"`, which means that the weight function is determined from the transition matrix via the shooting method (see Section 2.6.2). This obviously requires the matrix **trans** to be updated during the simulation. Accordingly we set **update_trans** to `T`, otherwise the transition matrix is not updated during the simulation and updating the weight function using the shooting method will not work.
- **enable_barriers** is set to `T` since we wish to use artificial dynamics to force the system to quickly explore the whole of order-parameter space. Specifically, we elect to have the system sweep through the macrostates sequentially, proceeding first towards macrostate 1, then from there to macrostate 100, thenback to macrostate 1, etc. Accordingly we set **barrier_dynamics** to `"pong_down"`.
- **lock_moves** controls how long the system is 'locked' into each macrostate by the 'macrostate barriers' during artificial dynamics, before the 'next' macrostate is opened to the system. This period should be long enough that the system has enough time to equilibrate locally within the macrostate, but not so long that the system never ends up exploring all macrostates during the simulation. Our choice of 38400 (=100 sweeps) seems to do the job. Note that while the equilibration time in our preliminary simulations was $\lesssim 200$ sweeps, the time to equilibrate locally *within one macrostate* is expected to be far shorter. Furthermore, the time spent in each macrostate will actually be longer than **lock_moves**: **lock_moves** is the number of moves before a new macrostate becomes available to the system; the system still must move into that new macrostate from the 'old' macrostate by its own accord. As expected, it takes longer for the system to move from old macrostates into new macrostates if the free energy difference difference between the macrostates is high.

7.4.2 Results

At the completion of the simulation the file `state` (not `state_0`, `state_1`, `state_2` or `state_3`) contains the final results of the simulation – pooled from all 4 MPI threads. We are obviously interested in the weight function. This can be obtained from the `state` file using `monteswitch_post`. After the following

command (invoked from within the `weight_function_generation` directory), the file `wf.dat` contains the weight function vs. order parameter:

```
$ ../../../monteswitch_post -extract_wf > wf.dat
```

`wf.dat` is plotted in Fig. 7.2. Note that the weight function is a smooth curve with two minima separated by a high peak near $M = 0$. This is the expected form for a weight function. If the curve were not smooth, or was constant for large regions of order-parameter space, then it is probable that the weight function has not properly been generated. This is perhaps an indication that more time is needed to generate a reasonable weight function. One possible solution is to continue the simulation with `monteswitch_mpi` using the `-resume` flag, e.g.

```
$ mpiexec -n 4 ../../../monteswitch_mpi -resume
```

Of course, just because the weight function has the form described above does not guarantee that it is good. Recall that the ideal weight function leads to the whole considered range of order-parameter space being sampled uniformly. A weight function with the form described above could easily lead to certain regions of order-parameter space being significantly over- or under-sampled. In principle this is not a problem because the final results – given a long enough production simulation – do not depend on the specific weight function used. However the quality of the weight function does determine how efficiently phase space is explored in the production simulation. Using a weight function of high quality samples order-parameter space almost uniformly, resulting in many accepted lattice switches, and both phases being explored over short timescales. On the other hand if a low-quality weight function is used then the system would spend more time in certain regions of order-parameter space than others, and the time between lattice-switches would be less. Hence it is prudent to perform a short simulation using the weight function to verify that it is of good quality. We do this below.

7.4.3 Checking the chosen order-parameter range

The ideal weight function is related to the canonical macrostate probability distribution via Eqn. (2.30). Using this equation one can obtain the probability distribution from the weight function. Of course, this will only be the 'true' probability distribution if the weight function is ideal, which it will not be near given the short length of our weight function generation simulation. Nevertheless we can obtain an *approximate* probability distribution from our weight function using Eqn. (2.30). This is useful for checking that our chosen order-parameter range is appropriate. We use the following command to obtain the probability distribution from the weight function, outputting it to the file `prob.dat`:

```
$ awk 'FNR==NR {sum=sum+exp(-$2); next} {print $1,exp(-$2)/sum}'
wf.dat wf.dat > prob.dat
```

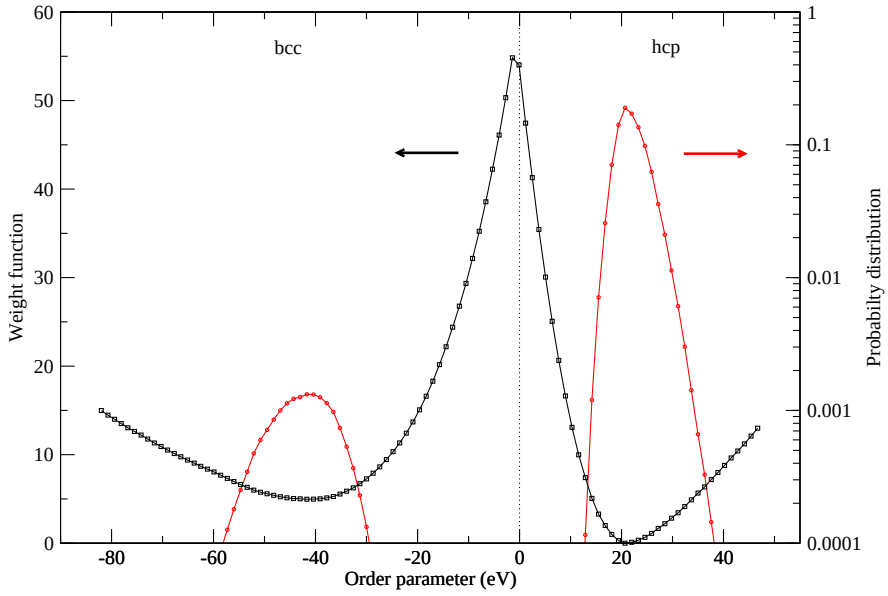

Figure 7.2: Results of the weight function generation simulation. The black curve and square symbols correspond to the weight function, while the red curve and circles corresponds to the probability distribution inferred from the weight function (as described in the main text). Note that the left vertical axis corresponds to the weight function while the right axis corresponds to the probability distribution – as indicated by the arrows. The regions of order-parameter space corresponding to bcc and hcp are indicated.

(Note that the appearance of `wf.dat` twice in the above command is deliberate). `prob.dat` is plotted in Fig. 7.2. Note that the probability distribution has two peaks in order-parameter space, each corresponding to one phase. Now, the order-parameter range is suitable only if the probability distribution is effectively zero at the maximum and minimum order parameters considered. If this is not the case, then some states which are significantly likely at equilibrium would be 'cut out' of the calculation of thermodynamic quantities, because moves which take the system outside the order-parameter range are automatically rejected. Fortunately here the probability distribution decays to essentially zero within the considered range as required. If this were not the case then we would have to re-run the weight function generation simulation using a larger order-parameter range.

7.4.4 Estimating ΔG from the weight function

With the aforementioned probabilty distribution one can obtain an estimate of the free energy difference between the phases. Recall that the free energy difference ΔG is related to the time spent in each phase via Eqn. (2.6), where p_2 and p_1 are the (canonical) probabilities that the system is in phase 1 and 2 respectively. p_1 and p_2 can be deduced by integrating over the corresponding peaks in the probability distribution in `prob.dat`. Noting that the peak in the negative region of order-parameter space corresponds to bcc, and that the peak in the positive region corresponds to hcp, it can be seen that the command applies Eqn. (2.6) to obtain the free energy difference indirectly from the weight function:

```
$ awk '{if($1<0){sum1=sum1+$2}; if($1>0){sum2=sum2+$2}} END{print
-log(sum1/sum2)/9.403 }' prob.dat
```

Recall that 9.403 is the value of β we are considering. The intensive value of ΔG is obtained by dividing by the extensive value by the number of atoms in the system, which in this case is 384:

```
$ awk '{if($1<0){sum1=sum1+$2}; if($1>0){sum2=sum2+$2}} END{print
-log(sum1/sum2)/(9.403*384) }' prob.dat
```

This command gives ΔG to be 0.00113081eV per particle, which corresponds to the hcp phase being very slightly more favoured than the bcc phase (since in *monteswitch* ΔG is the free energy of phase 1, which is bcc here, relative to phase 2, which is hcp here). We will compare this value with the 'true' value we obtain for ΔG later.

We emphasise that this method provides only an estimate for ΔG. Crucially, unlike ΔG obtained from the forthcoming production simulation, the method does not provide an uncertainty for ΔG. Hence one should not rely upon this method for accurate results, though the method could be used, e.g., to quickly determine estimates for ΔG over a wide range of T, perhaps with the aim of deducing the approximate position of the transition temperature (where $\Delta G = 0$).

7.4.5 Using other weight function generation methods

In the above example we used the transition-matrix method in conjunction with artificial dynamics to generate the weight function. This method gives a reasonable weight function very quickly. However, *monteswitch* supports other methods for generating weight functions. These were described in Section 2.6, namely the visited states method, and the transition-matrix method in conjunction with *natural* dynamics. The variables in `params_in` required to implement each of the aforementioned methods within *monteswitch* are shown in Table 7.4.5. Recall that for the transition-matrix method with artificial dynamics there are two possibilities: evolving the macrostate barriers randomly or systematically. Guidelines are also provided in the table regarding values for **update_eta_sweeps** and **lock_moves** for each method. The reasoning behind these guidelines is as follows. For the visited states method, **update_eta_sweeps** corresponds to the block size mentioned in Section 2.6.1. This should be long enough such that a block is representative of a 'long' multicanonical simulation with the current weight function. Furthermore, the closer to the ideal weight function one wishes to get, the longer the blocks should be. For the transition-matrix method, if multicanonical sampling is used, then one may as well update the weight function 'continuously'. In this case the weight function used always reflects all the information gathered so far during the weight function generation simulation, thus enabling the system to explore order parameter space as widely as possible in as short a simulation time as possible. Regarding the choice of **lock_moves**, see the discussion in Section 7.4.

Control variable	VS	TM-ND	TM-AD
enable_multicanonical	`T`	`T`	`F` or `T`
update_eta	`T`	`T`	`T`
update_eta_sweeps	$\gg$ multicanonical correlation time (in sweeps)	~ 1	$\sim$ 1 if **enable_multicanonical**=`T`; $\leq$ **stop_sweeps** if **enable_multicanonical**=`F`
update_trans	N/A	`T`	`T`
update_eta_method	`"VS"`	`"shooting"`	`"shooting"`
enable_barriers	`F`	`F`	`T`
barrier_dynamics	N/A	N/A	`"random"` for random macrostate barrier evolution; `pong_up` or `pong_down` for systematic evolution
lock_moves	N/A	N/A	$\gtrsim$ time to equilibrate *within a macrostate* (in moves)

Table 7.1: Values for variables in `params_in` or `state` required to implement various weight function generation methods. 'VS' refers to the visited states method; 'TM-D' refers to the transition-matrix method with natural dynamics; 'TM-AD' refers to the transition-matrix method with artificial dynamics. (See Section 2.6 for descriptions of these methods). 'N/A' signifies that the variable is not used in the method, and hence its value is unimportant

7.5 Weight function verification

Assuming that we have a good weight function, one thing remains which is useful to know before we perform our production simulation: the correlation time for the multicanonical simulation using this weight function. [1] Recall that we obtained an estimate for the equilibration time earlier, and that it was $\lesssim 200$ sweeps. However, this applies only to a one-phase canonical simulation. As we will see in a moment, the correlation time in the two-phase multicanonical LSMC simulation is much longer. We need to know the multicanonical correlation time because it determines how large the blocks should be for block averaging to calculate thermodynamic quantities, in particular the free energy difference. To determine this correlation time we perform a short multicanonical simulation using the weight function. This simulation also acts to check verify that the weight function is indeed sensible, i.e., that it leads to the entire range of order-parameter space being explored approximately uniformly.

The directory `weight_function_verification` corresponds to the weight function verification simulation. Recall that we can 'resume' a simulation whose variables are stored in the `state` file by invoking `monteswitch` with the `-resume` argument. Furthermore we can resume a simulation from `state` but with all counter variables reset to zero by invoking the `-reset` argument. With this in mind, we use the `state` file from our weight function generation simulation as a starting point. This file contains the weight function we wish to verify. We will modify this file to suit our needs, and then run the simulation using the command (from within the directory `weight_function_verification`)

```
$ ../../../monteswitch -reset
```

Note that the file `state_start` in the directory `weight_function_verification` contains the `state` file from the directory `weight_function_generation` after it has been modified for the weight function verification simulation; and the file `state` in the directory `weight_function_verification` corresponds to the output of the weight function verification simulation.

7.5.1 Creating the input `state` file

The modifications to the `state` file before the weight function verification simulation is run can be seen by invoking the following command in the `weight_function_verification` directory:

```
$ diff state_start ../weight_function_generation/state
```

The key changes are as follows:

[1] Presumably one could make the multicanonical simulation more efficient by optimising the maximum particle and volume move step sizes used in the multicanonical simulation, and not simply using the same values as for the canonical simulation. We have never done this, though it is something worth investigating.

- We have changed **stop_sweeps** to be 10000, which corresponds to a short simulation.
- We have set **update_eta** to `F`, so that the weight function is fixed throughout the simulation.
- We have set **enable_barriers** to `F`, since we want 'natural' dynamics for the weight function verification simulation.
- We have set **output_file_period** to 10 so we have high-resolution information about how order-parameter space is explored.
- We have set **output_stdout_period** to 10 so we can view the progress of the simulation: information is output to stdout every 10 sweeps.

7.5.2 Results

After the simulation is complete we can use `data` to generate a plot of the order parameter vs. time in the same manner as for the preliminary simulations:

```
$ grep 'M:' data | awk '{print $2,$3}' > M_vs_t.dat
```

`M_vs_t.dat` is plotted in Fig. 7.3 – see the results for the first 10000 sweeps. From the figure it can be seen that the entire range of order-parameter space was explored within the simulation. However, a 10000-sweep simulation in this case is too short to estimate the correlation time: within the 10000 sweeps the system only makes one significant transition between the phases, from hcp to bcc. For this reason we continued the simulation for another 10000 sweeps by invoking the command

```
$ ../../../monteswitch -resume
```

Hence the output files `state` and `data`, and `M_vs_t.dat` in the `weight_function_verification` directory correspond to 20000 sweeps, not 10000 sweeps. (A copy of the output `state` file after the first-10000 sweep simulation can be found in the file `state_after_10000_sweeps`). As can be seen from the figure, there are additional transitions between the phases in the second 10000 sweeps. Hence the correlation time – which in this case involves both phases being explored – is $\sim$ 10000 sweeps. The block size we use in the forthcoming production simulation must be much greater than this. Note that after the production run one can retrospectively check that the block size was appropriate. We will do this later. Furthermore, one could use the informtion contained in the `data` file output by the production simulation to perform block averaging for a variety of block sizes, though we do not explain how to do this here.

It is instructive to plot how often each macrostate was visited during the weight function verification simulation. `monteswitch_post` provides an easy means of doing this: the following command creates a file `counts_vs_M.dat`, which is a plot of the order parameter for each macrostate (the lower bound for

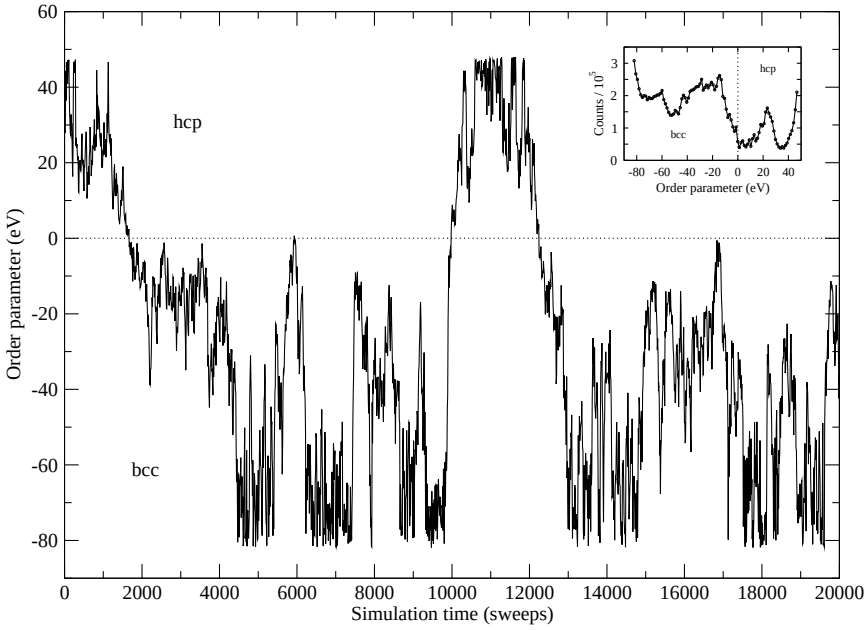

Figure 7.3: Plot of the order parameter vs. simulation time for the weight function verification simulation. The regions of order-parameter space corresponding to bcc and hcp are indicated. The inset is a histogram of how often each macrostate (identified by its order parameter as described in the main text) was visited during the simulation.

the order parameter range covered by the macrostate) vs. the number of times the macrostate was visited during the simulation:

```
$ ../../../monteswitch_post -extract_M_counts | awk '{print
$1,$2+$3}' > counts_vs_M.dat
```

`counts_vs_M.dat` is plotted in the inset of Fig. 7.3. From this we see that all macrostates were visited. It is conspicuous that the histogram is not flat. We would in fact be very lucky to get a flat histogram, even with the ideal weight function, given that we only performed 20000 sweeps – which is the order of the correlation time.

7.6 Production simulation

We are now finally ready to perform the production simulation. The corresponding directory is `production_simulation`. As with the weight function

verification simulation, we use the `state` file output from the weight function generation simulation as the starting point for the production simulation. We will modify this file to suit our needs, and then run the simulation, using four MPI threads with `monteswitch_mpi`, using the command (invoked from within the `production_simulation` directory)

```
$ mpiexec -n 4 ../../../monteswitch_mpi -reset
```

Note that again we have used the `-reset` flag to reset the counter variables at initialisation.

7.6.1 Creating the input `state` file

The modifications to the state file can be seen by invoking the following command in the `production_simulation` directory:

```
$ diff state_start ../weight_function_generation/state
```

The key changes are as follows:

- We have changed **stop_sweeps** to be 700000, which corresponds to 175000 sweeps to be performed for each MPI thread.
- We have set **equil_sweeps** to 20000, which is larger than the correlation time we determined via the weight function verification simulation.
- We have set **update_eta** to `F` so that the weight function is fixed throughout the simulation.
- We have set **enable_barriers** to `F` since we want 'natural' dynamics for the production simulation.
- **calc_equil_properties** is set to `T`. This is required for the simulation to calculate thermodynamic quantities (in particular ΔG, and the enthalpy and volume of each phase) and their uncertainties using block averaging.
- **block_sweeps**, the number of sweeps which constitute a block in the block averaging is set to 155000, which is far larger than the correlation time of ~ 10000 sweeps determined in the weight function verification simulation. Note that, since each MPI thread performs 175000 sweeps, and since each thread will be given 20000 sweeps to equilibrate – during which time the states visited by the system are not used in block averaging - setting the block size to 155000 sweeps corresponds to each MPI thread performing exactly 1 block. Hence the simulation in total considers 4 blocks.

7.6.2 Results

At the completion of the simulation the `state` file contains the thermodynamic quantities evaluated using block averaging. It is these quantities we are interested in, especially ΔG. The variables **equil_DeltaF** and **sigma_equil_DeltaF** in `state` contain ΔG and its uncertainty evaluated using block averaging. The following command extracts these variables from the `state` file:

```
$ grep -E '( equil_DeltaF| sigma_equil_DeltaF)' state
```

Recall that all quantities in the `state` file are extensive. To get the intensive values we must divide them by the number of atoms in the system, which in this case is 384. Hence the following command gives the intensive free energy difference between the phases:

```
$ grep -E '( equil_DeltaF| sigma_equil_DeltaF)' state | awk
'{print $1,$2/384}'
```

Using this we find that ΔG is 0.00104(2)eV per particle, which implies that hcp is slightly favoured over bcc at this T and P. Note that this agrees favourably with the ΔG we obtained from the weight function earlier, namely 0.00113 eV per particle. However, note that ΔG is $\sim$1 meV per particle, which is very small. Thus T =1233K is very close to the zero-pressure bcc–hcp transition temperature: we are in agreement with Ref. [33].

We will now show that we are also in agreement with Ref. [33] with regards to $\Delta H_{\mathrm{hcp}\to\mathrm{bcc}}$ and $\Delta V_{\mathrm{hcp}\to\mathrm{bcc}}/V_{\mathrm{bcc}}$. Using similar commands to the above, one can extract the volume and enthalpy per atom for each phase evaluated using block averaging, as well as their associated uncertainties:

```
$ grep -E '( equil_V_1| sigma_equil_V_1)' state | awk '{print $1,
 $2/384}'
```

gives the volume per atom for phase 1;

```
$ grep -E '( equil_V_2| sigma_equil_V_2)' state | awk '{print $1,
  $2/384}'
```

gives the volume per atom for phase 2;

```
$ grep -E '( equil_H_1| sigma_equil_H_1)' state | awk '{print $1,
 $2/384}'
```

gives the enthalpy per atom for phase 1; and

```
$ grep -E '( equil_H_2| sigma_equil_H_2)' state | awk '{print $1,
 $2/384}'
```

gives the enthalpy per atom for phase 2. Using these quantities one can calculate $\Delta H_{\mathrm{hcp}\to\mathrm{bcc}}$ and $\Delta V_{\mathrm{hcp}\to\mathrm{bcc}}/V_{\mathrm{bcc}}$, which are presented in Table 7.6.2 alongside the results of Ref. [33]. As can be seen from the table the quantities obtained from the production simulation are in excellent agreement with those of Ref.

	monteswitch	Ref. [33]
$\Delta H_{\mathrm{hcp}\to\mathrm{bcc}}$ (eV)	0.0394(2)	0.039
$\Delta V_{\mathrm{hcp}\to\mathrm{bcc}}/V_{\mathrm{bcc}}$ (%)	-0.812(7)	-0.8

Table 7.2: Thermodynamics quantities obtained from the production simulation, and analogous quantities obtained in Ref. [33]. Note that the quoted $\Delta H_{\mathrm{hcp}\to\mathrm{bcc}}$ is intensive, i.e., 'per particle'.

[33]. Of course, it is unnecessary to use an LSMC simulation to extract such one-phase thermodynamic quantities; a pair of conventional Monte Carlo simulations will do the job more efficiently. Here the quantities are somewhat of a by-product of the LSMC simulation, which *is* however required to calculate the *two-phase* quantity ΔG.

7.6.3 Final checks

Once the production simulation is complete it is prudent to perform some checks to ensure that it has gone as expected, and hence that the values for the thermodynamic quantities obtained from the simulation are meaningful. Comparing the thermodynamic quantities obtained from the simulation to 'known' values, as was done above, is an excellent consistency check. However this will not always be possible. A less obvious check to perform is as follows. The free energy difference for a block is given by Eqn. (2.6), where in this case t_1 and t_2 are the number of moves which the system spent in phase 1 and phase 2 respectively during the block. Note that this equation presupposes that both phases are explored during the block. Now, in evaluating ΔG, *monteswitch* only considers blocks during which both phases are explored. This notwithstanding, one should check that the block size is significantly larger than the time it typically takes the system to explore the whole of order-parameter space – which in practice is an equivalent condition to both phases being thoroughly explored during each block. To do this we plot the order parameter vs. simulation time for each MPI thread using the following commands:

```
$ grep 'M:' data_0 | awk '{print $2,$3}' > M_vs_t_0.dat
$ grep 'M:' data_1 | awk '{print $2,$3}' > M_vs_t_1.dat
$ grep 'M:' data_2 | awk '{print $2,$3}' > M_vs_t_2.dat
$ grep 'M:' data_3 | awk '{print $2,$3}' > M_vs_t_3.dat
```

where `M_vs_t_n.dat` is a plot of order parameter vs. simulation time for thread n. These plots are shown in Fig. 7.4. As can be seen from the figure, for each thread the system traverses the entire range of order-parameter space often within the block size (175000 sweeps), as required. Of course, the aim of the weight function verification simulation was to ensure that this would happen: in that simulation we deduced the correlation time, and used that to inform our choice of block size. Still, it is good to double-check.

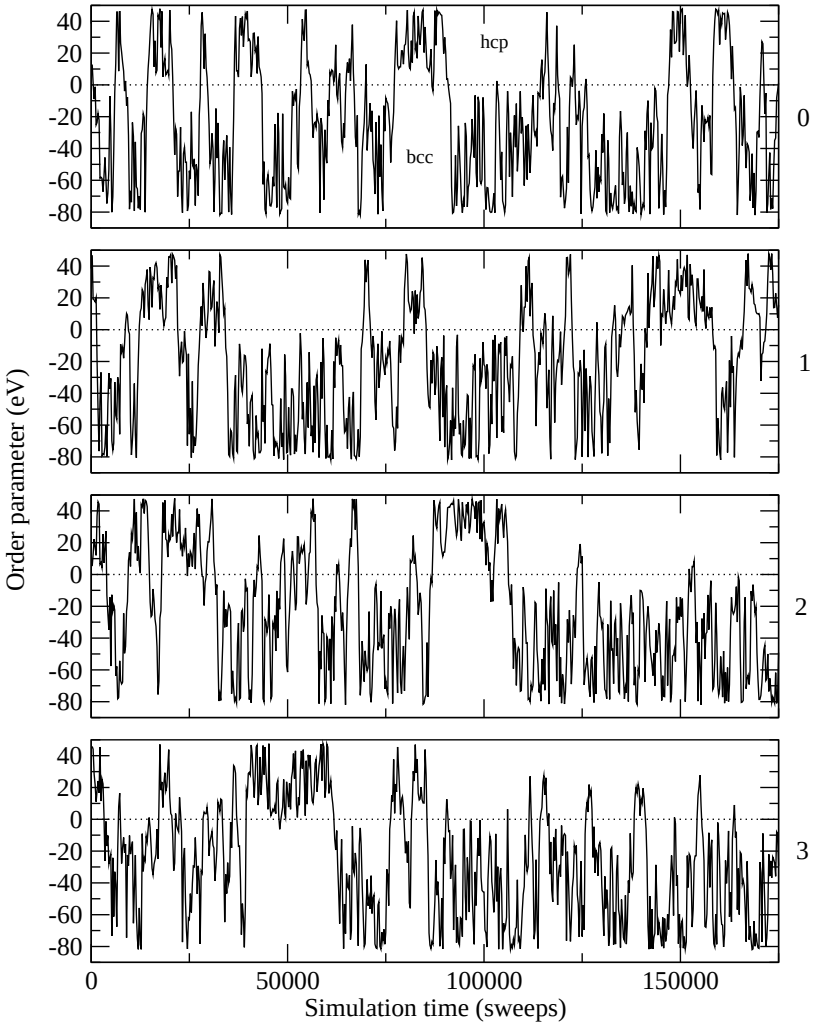

Figure 7.4: Plots of the order parameter vs. simulation time for each MPI thread for the production simulation. The number to the right of each panel corresponds to the thread index. The regions of order-parameter space corresponding to bcc and hcp are indicated.

Chapter 8

Test cases

The directory `Tests` contains a suite of test cases for the main programs in *monteswitch*. Of course, these tests can also be used as examples.

Each test can be executed by executing the `run.sh` shell script in the corresponding directory after copying the appropriate `interactions.f95` file from `Interactions` to `interactions.f95`, and then compiling the package (see Chapter 3). Note that the `run.sh` scripts assume that the main programs `monteswitch` and `monteswitch_mpi` are located in the same directory as the source code, which should be the case by default. Furthermore the `run.sh` files may require some modification to suit your platform (see Chapter 3).

8.1 Einstein crystal (`EC`)

In the Einstein crystal the energy of the system is given by

$$E = \sum_{i=1}^{N} \alpha u_i^2, \tag{8.1}$$

where u_i is the displacement of particle i from its lattice site, N denotes the number of particles in the system, and α is the free parameter of the model. The Einstein crystal is an excellent testing ground for *monteswitch* because analytical results can be derived for this system. For instance, the free energy of the system is given by [1]

$$F = -\frac{3N}{2\beta} \ln\left[\frac{\pi}{\alpha\beta}\right], \tag{8.2}$$

where β is the thermodynamic beta. Furthermore the mean-squared displacement of each particle is [1]

$$\langle u^2 \rangle = \frac{3}{2\beta\alpha}, \tag{8.3}$$

and the mean energy of the system is

$$\langle E \rangle = \alpha N \langle u^2 \rangle = \frac{3N}{2\beta}. \tag{8.4}$$

Here we test *monteswitch* against these analytical results. Our 'phases' here are two Einstein crystals with different values of α. We denote the α pertaining to phase 1 as α_1, and similarly for phase 2. Using Eqn. (8.2), the following expression for the free energy difference between the phases can be derived:

$$\Delta F \equiv F_1 - F_2 = \frac{3N}{2\beta} \ln\left[\frac{\alpha_1}{\alpha_2}\right]. \tag{8.5}$$

We use Eqns. (8.3), (8.4) and (8.5) below.

The `interactions.f95` file corresponding to the Einstein crystal is `interactions_EC.f95`. The Hamiltonian for each phase realised in *monteswitch* in conjunction with `interactions_EC.f95` is as described above, where note that the orign constitutes the lattice site *for all particles*. α_1 and α_2 are specified for new simulations via the input file `interactions_in`. The format of this file is as follows. On the first line there are two tokens. The first token is an arbitrary `CHARACTER(LEN=20)` variable (we recommend: `alpha_1=`). The second token is the value of α_1 (stored internally as **alpha_1**), which is of type `REAL`. The second line is similar, but for α_2.

8.1.1 `test_EC_1`

This test calculates ΔF, and $\langle u^2 \rangle$ and $\langle E \rangle$ for each phase, for the case $\alpha_1 = 1$, $\alpha_2 = 2$, $\beta = 100$ and $N = 1$. The test uses canonical (i.e., not multicanonical) sampling, and consists of 10,000,000 sweeps using **part_step**=0.12.

The following checks should be performed at the completion of the test:

1. ΔF should be $0.015 \ln(1/2) = -0.0103972$: see the variables **equil_DeltaF** and **sigma_equil_DeltaF** output to stdout.
2. $\langle u^2 \rangle$ should be 0.015 for phase 1 and 0.0075 for phase 2: see the variables **equil_umsd_1**, **sigma_equil_umsd_1**, **equil_umsd_2** and **sigma_equil_umsd_2** output to stdout.
3. $\langle E \rangle$ should be 0.015 for both phases. Check that this is the case by examining the variables **equil_H_1**, **sigma_equil_H_1**, **equil_H_2** and **sigma_equil_H_2** in the output to stdout.

8.1.2 `test_EC_1_MPI`

This test is the same as `test_EC_1`, except that an MPI simulation is run using 4 threads. In this case one should check that 1, 2 and 3 described in `test_EC_1` applies to the variables listed in the `state` file, as opposed to the output to stdout. This test verifies that `monteswitch_mpi` works correctly, especially that it correctly combines the results from all threads before outputting the final results to `state`.

8.1.3 test_EC_2

This test is the same as test_EC_1, except that 10 simulations consisting of 1,000,000 sweeps each are performed instead of one simulation of 10,000,000 sweeps, using both the -resume and -new arguments to the executable monteswitch instead of simply the -new argument. This test verifies that checkpointing works correctly within monteswitch. Checks 1, 2 and 3 described in test_EC_1 should be performed on the state file.

8.1.4 test_EC_2_MPI

This test is the same as test_EC_2, except that 10 4-thread MPI simulations are performed. This test verifies that checkpointing works correctly for monteswitch_mpi. Checks 1, 2 and 3 described in test_EC_1 should be performed on the state file.

8.1.5 test_EC_3

This test calculates the thermodynamic quantities mentioned in test_EC_1, but uses multicanonical sampling instead of canonical sampling, with a weight function generated using the visited states method. To elaborate, a weight function generation simulation is performed which consists of 10,000,000 sweeps. Then the state file is altered in preparation for the production simulation. Then the production simulation is performed, which consists of 10,000,000 sweeps. Checks 1, 2 and 3 described in test_EC_1 should be performed on the state file.

8.1.6 test_EC_4

This test is the same as test_EC_3 except that the weight function is generated using the shooting method over 1,000,000 sweeps. Checks 1, 2 and 3 described in test_EC_1 should be performed on the state file.

8.1.7 test_EC_4_MPI

This test is the same as test_EC_4, but uses 2 MPI threads. Checks 1, 2 and 3 described in test_EC_1 should be performed on the state file.

8.1.8 test_EC_5

This test is the same as test_EC_3, however the weight function is generated using the shooting method in conjunction with '"pong_down" artificial dynamics' and canonical sampling over 1,000,000 sweeps. Checks 1, 2 and 3 described in test_EC_1 should be performed on the state file. However, two further files are created during this test: M_vs_t.dat contains the value of the order parameter vs. simulation time, and macro_vs_t.dat contains the macrostate number vs. simulation time. Both plots should resemble a sawtooth.

8.1.9 `test_EC_5_MPI`

This test is the same as `test_EC_5`, except that a 2-thread MPI simulation is performed, and '`"pong_up"` artificial dynamics' are used. In this case the additional files created by the test are `M_vs_t_0.dat` and `macro_vs_t_0.dat`, which pertain to MPI thread '0', and `M_vs_t_1.dat` and `macro_vs_t_1.dat`, which pertain to thread '1'. All files should have a sawtooth pattern. Furthermore checks 1, 2 and 3 described in `test_EC_1` should be performed on the `state` file.

8.1.10 `test_EC_6`

This test is the same as `test_EC_5`, except that '`"random"` artificial dynamics' are used. The files `M_vs_t.dat` and `macro_vs_t.dat` in this case should resemble random walks. Furthermore checks 1, 2 and 3 described in `test_EC_1` should be performed on the `state` file.

8.1.11 `test_EC_7`

This test is the same as `test_EC_5`, except that 10 simulations are performed (each consisting of 100,000 sweeps) instead of 1. This tests verifies that checkpointing works correctly with artificial dynamics. Checks 1, 2 and 3 described in `test_EC_1` should be performed on the `state` file.

8.1.12 `test_EC_7_MPI`

This test is similar to `test_EC_7`, except that 2 2-thread MPI simulations are performed such that the total number of sweeps is 1,000,000. Checks 1, 2 and 3 described in `test_EC_1` should be performed on the `state` file.

8.1.13 `test_EC_8`

This test is similar to `test_EC_1`, except that $N = 4$ instead of 1.

The following checks should be performed at the completion of the test:

1. ΔF should be $0.06 \ln(1/2) = -0.0415888$: see the variables **equil_DeltaF** and **sigma_equil_DeltaF** output to stdout.

2. $\langle u^2 \rangle$ should be 0.015 for phase 1 and 0.0075 for phase 2: see the variables **equil_umsd_1**, **sigma_equil_umsd_1**, **equil_umsd_2** and **sigma_equil_umsd_2** output to stdout.

3. $\langle E \rangle$ should be 0.06 for both phases. Check that this is the case by examining the variables **equil_H_1**, **sigma_equil_H_1**, **equil_H_2** and **sigma_equil_H_2** in the output to stdout

8.1.14 `test_EC_8_MPI`

This test is the same as `test_EC_8`, except that an MPI simulation is run using 4 threads. In this case checks 1, 2 and 3 described in `test_EC_8` should apply to the variables listed in the `state` file – as opposed to the output to stdout. This test verifies that `monteswitch_mpi` works correctly for systems containing multiple particles, especially that it correctly combines the results from all threads before outputting the final results to `state`.

8.1.15 `test_EC_9`

This test is the same as `test_EC_8`, except that 10 simulations consisting of 1,000,000 sweeps each are performed instead of one simulation of 10,000,000 sweeps, using both the `-resume` and `-new` arguments to the executable `monteswitch` instead of simply the `-new` argument. This test verifies that checkpointing works correctly for systems containing multiple particles. Checks 1, 2 and 3 described in `test_EC_8` should be performed on the `state` file.

8.1.16 `test_EC_9_MPI`

This test is the same as `test_EC_9`, except that 10 4-thread MPI simulations are performed. This test verifies that checkpointing works correctly for `monteswitch_mpi` for systems containing multiple particles. Checks 1, 2 and 3 described in `test_EC_8` should be performed on the `state` file.

8.2 NPT Einstein crystal (`EC_NPT`)

The Einstein crystal model described in the previous section cannot be used to validate *monteswitch* for the NPT ensemble: all of the test cases described so far were for the NVT ensemble. Accordingly we consider here a generalisation of the Einstein crystal model suitable for testing the implementation of the NPT ensemble in *monteswitch*. We refer to this model as the *NPT Einstein crystal*. In the NPT Einstein crystal the energy of the system is given by

$$E = \sum_{i=1}^{N} \alpha V^{-\gamma} u_i^2, \tag{8.6}$$

where V is the volume of the system, α and γ are the free parameters of the model, and u_i and N have the same significance as above for the Einstein crystal. The following analytical results can be derived for this model. Firstly, the NPT partition function is given by

$$Z = \left[2\pi\Gamma(1.5)\right]^N (\beta\alpha)^{-1.5N} (\beta P)^{-(1.5N\gamma+1)} \Gamma(1.5N\gamma + 1), \tag{8.7}$$

where $\Gamma(x)$ is the gamma function, P is the pressure, and β is the thermodynamic beta. Using this the Gibbs free energy difference ΔG between two NPT

Einstein crystal 'phases' with free parameters α_1 and γ_1 (for phase 1), and α_2 and γ_2 (for phase 2) can be derived:

$$\Delta G = G_1 - G_2 = -\frac{1}{\beta} \ln\left[\frac{Z_1}{Z_2}\right], \tag{8.8}$$

where

$$\frac{Z_1}{Z_2} = \left(\frac{\alpha_2}{\alpha_1}\right)^{1.5N} (\beta P)^{1.5N(\gamma_2-\gamma_1)} \frac{\Gamma(1.5N\gamma_1+1)}{\Gamma(1.5N\gamma_2+1)}. \tag{8.9}$$

Furthermore, it can be shown that the mean-squared displacement of a particle in an NPT Einstein crystal is given by

$$\langle u^2 \rangle = 1.5(\beta\alpha)^{-1}(\beta P)^{-\gamma} \frac{\Gamma(1.5N\gamma+1+\gamma)}{\Gamma(1.5N\gamma+1)}, \tag{8.10}$$

and that the mean volume of the system is

$$\langle V \rangle = \frac{1}{(\beta P)} \frac{\Gamma(1.5N\gamma+2)}{\Gamma(1.5N\gamma+1)}. \tag{8.11}$$

We use Eqns. (8.8), (8.10) and (8.11) below.

The `interactions.f95` file corresponding to the NPT Einstein crystal is `interactions_EC_NPT.f95`. The Hamiltonian for each phase realised in *monteswitch* in conjunction with `interactions_EC_NPT.f95` is the NPT Einstein crystal as described above. α_1, α_2, γ_1 and γ_2 are specified for new simulations via the input file `interactions_in`. The format of this file is as follows. On the first line there are two tokens. The first token is an arbitrary `CHARACTER(LEN=20)` variable (we recommend: `alpha_1=`). The second token is the value of α_1 (stored internally as **alpha_1**), which is of type `REAL`. The second, third and fourth lines are similar, but for α_2, γ_1 and γ_2 respectively.

8.2.1 `test_EC_NPT_1`

This test calculates ΔG, and $\langle u^2 \rangle$ and $\langle V \rangle$ for each phase, for the case $\alpha_1 = 1000$, $\gamma_1 = 2$, $\alpha_2 = 2000$, $\gamma_2 = 1.8$, $\beta = 10$, $P = 0.25$ and $N = 1$. This test uses canonical sampling (i.e., not multicanonical sampling), and consists of 10,000,000 sweeps using **part_step**=0.03 and **vol_step**=3.0.

The following checks should be performed at the completion of the test:

1. $\langle u^2 \rangle$ should be 0.00048 for phase 1 and 0.00018089 for phase 2: see the variables **equil_umsd_1**, **sigma_equil_umsd_1**, **equil_umsd_2** and **sigma_equil_umsd_2** output to stdout.

2. $\langle V \rangle$ should be 1.6 for phase 1 and 1.48 for phase 2: see the variables **equil_V_1**, **sigma_equil_V_1**, **equil_V_2** and **sigma_equil_V_2** output to stdout.

3. ΔG should be -0.11285: see the variables **equil_DeltaF** and **sigma_equil_DeltaF** output to stdout.

8.2.2 test_EC_NPT_2

This test is the same as `test_EC_NPT_1`, except that 5 simulations consisting of 2,000,000 sweeps each are performed instead of one simulation of 10,000,000 sweeps, using both the `-resume` and `-new` arguments to the executable `monteswitch` instead of simply the `-new` argument. This test verifies that checkpointing works correctly for `monteswitch`. Checks 1–3 described in `test_EC_NPT_1` should be performed on the `state` file.

8.2.3 test_EC_NPT_2_MPI

This test is similar to `test_EC_NPT_2`, except that 5 2-thread MPI simulations are performed, each consisting of 4,000,000 sweeps. This test verifies that checkpointing works correctly for `monteswitch_mpi`. Checks 1–3 described in `test_EC_NPT_1` should be performed on the `state` file.

8.2.4 test_EC_NPT_3

This test calculates the thermodynamic quantities mentioned in `test_EC_NPT_1`, but uses multicanonical sampling instead of canonical sampling, with a weight function generated using the visited states method. To elaborate, a weight function generation simulation is performed which consists of 10,000,000 sweeps. Then the `state` file is altered in preparation for the production simulation. Then the production simulation is performed, which consists of 10,000,000 sweeps. Checks 1–3 described in `test_EC_NPT_1` should be performed on the `state` file. However, this test also creates a file containing the weight function, as well as a file containing a histogram of the macrostates visited during the production simulation. These can be found in the files `wf.dat` and `M_hist.dat` respectively. The histogram in `M_hist.dat` should be fairly flat.

8.2.5 test_EC_NPT_4

This test is the same as `test_EC_NPT_2`, but uses $N = 5$ instead of $N = 1$, as well as a lattice switch which changes the system volume. This test verifies that lattice switches which do not preserve the system volume sample the NPT ensemble correctly in *monteswitch*. It also verifies that checkpointing and sampling via particle and volume moves are correct for systems containing multiple particles.

The following checks should be performed at the completion of the test:

1. $\langle u^2 \rangle$ should be 0.006528 for phase 1 and 0.001862841 for phase 2: see the variables **equil_umsd_1**, **sigma_equil_umsd_1**, **equil_umsd_2** and **sigma_equil_umsd_2** in the file `state`.

2. $\langle V \rangle$ should be 6.4 for phase 1 and 5.8 for phase 2: see the variables **equil_V_1**, **sigma_equil_V_1**, **equil_V_2** and **sigma_equil_V_2** in the file `state`.

3. ΔG should be -0.78606733: see the variables **equil_DeltaF** and **sigma_equil_DeltaF** in the file `state`.

8.3 Lennard–Jones solid: hcp vs. fcc (`LJ_hcp_fcc`)

A solid in which particles interact via the Lennard-Jones potential is a more realistic test for *monteswitch* than those described above. Here we use *monteswitch* to calculate the Gibbs free energy difference ΔG between the hcp and fcc phases of the Lennard-Jones solid, in an effort to reproduce the results of Refs. [30, 6].

The relevant `interactions.f95` file for these tests is `interactions_LJ_hcp_fcc.f95`. Note that the Hamiltonian realised in *monteswitch* in conjunction with `interactions_LJ_hcp_fcc.f95` is applicable only to to the hcp–fcc problem; `interactions_LJ_hcp_fcc.f95` does not implement the Lennard-Jones potential in the usual way. To elaborate, it is common when implementing interatomic potentials to truncate the potential at a predetermined distance, or to have particles only interact with those close by via 'neighbour lists'. This is done to speed up simulations. However, it also leads to incorrect results for the problem we are considering: see Refs. [30, 6] for details. The solution is to evaluate the *difference* of the energy of the current state relative to the ground state for the current phase and density, and then apply a truncation to this difference. This is what is done in `interactions_LJ_hcp_fcc.f95`. Explicitly, the energy for particle positions $\{\mathbf{r}\}$ in the hcp phase, given that the system currently has density ρ, is given by:

$$E = \Phi_{\mathrm{LJ,trunc}}(\{\mathbf{r}\}) - \Phi_{\mathrm{LJ,trunc}}(\{\mathbf{R}_{\mathrm{hcp}}\}) + E_{\mathrm{GS,hcp}}(\rho), \tag{8.12}$$

where $\{\mathbf{R}_{\mathrm{hcp}}\}$ are the lattice vectors corresponding to the hcp lattice at the current density ρ, $\Phi_{\mathrm{LJ,trunc}}(\{\mathbf{r}\})$ is the Lennard-Jones energy for the set of particle positions $\{\mathbf{r}\}$ using truncated interactions (in the conventional sense, e.g., ignore interactions between particles i and j if their separation is greater than some cut-off), and $E_{\mathrm{GS,hcp}}(\rho)$ is the *exact* energy of the hcp lattice at density ρ, i.e., what $\Phi_{\mathrm{LJ,trunc}}(\{\mathbf{R}_{\mathrm{hcp}}\})$ would be if the cut-off distance were taken to infinity. Similar applies for the fcc phase.

8.3.1 `test_LJ_hcp_fcc_1`

This test calculates ΔG for a 216-particle supercell at $P = 0$ and $\beta = 10$ in the NPT ensemble, using Lennard-Jones parametrisation $\epsilon = \sigma = 1$. Two 4-thread MPI simulations are used. The first generates the weight function using `"pong_down"` artificial dynamics over 500,000 sweeps. The second simulation calculates ΔG. Note that this test may take a while to run. The final value for ΔG should match that in Ref. [30], Fig. 6.21, the data-point corresponding to $k_B T = 0.1$, i.e., ΔG should be -0.00055 per particle. (Note that in Ref. [30] ΔG

is defined as the free energy of fcc relative to hcp, while in *monteswitch*, with hcp corresponding to phase 1 and fcc corresponding to phase 2 as is the case here, ΔG is the free energy of hcp relative to fcc – hence the change of sign here relative to the result in Ref. [30]). Therefore the test should yield an *extensive* value for ΔG of -0.12.

The following checks should be performed at the completion of the test:

1. ΔG should be -0.12: see the variables **equil_DeltaF** and **sigma_equil_DeltaF** in the `state` file.

8.4 Hard spheres (single species) (`HS`)

The hard-sphere system has been well studied, and makes an excellent testing ground for *monteswitch*. The relevant `interactions.f95` file for these tests is `interactions_HS.f95`, whose usage is described in Chapter 5. Note that here we set the energy cost of two spheres overlapping, ϵ, to 100,000, as well as setting $\beta = 1$, in order to realise hard spheres as described in Chapter 5.

8.4.1 `test_HS_1`

This test calculates the free energy difference ΔG between the hcp and fcc phases for a 216-particle system in the NPT ensemble at $P\beta\sigma^3 = 14.58$ (where σ denotes the hard-sphere diameter), using `"UVM"` volume moves. An analogous calculation has been performed in Refs. [30, 3], the results of which will act as a benchmark. Two 4-thread MPI simulations are used. The first generates the weight function using `"pong_down"` artificial dynamics over 2,000,000 sweeps. The second simulation calculates ΔG using a total of 125,000,000 sweeps. Note that this test may take a while to run. The final value for ΔG should match Refs. [30, 3]: ΔG should be $0.00113(4)\beta$ per particle. (Note that in Refs. [30, 3] ΔG is defined as the free energy of fcc relative to hcp, while in *monteswitch*, with hcp corresponding to phase 1 and fcc corresponding to phase 2, ΔG is the free energy of hcp relative to fcc – hence the change of sign here relative to the result of Refs. [30, 3]).

The following checks should be performed at the completion of the test:

1. ΔG obtained from the simulation, when divided by 216 (to yield the intensive value), should agree with 0.00113(4): see the variables **equil_DeltaF** and **sigma_equil_DeltaF** in the `state` file.

8.4.2 `test_HS_2`

This test calculates the properties of the hcp phase for a 216-particle system at $P\beta\sigma^3 = 14.58$ in the NPT ensemble. A 4-thread MPI conventional (one-phase, not LSMC) Monte Carlo simulation is performed, using `"UVM"` volume moves. The results should correspond to Table 4.1 in Ref. [30]; specifically, the row beginning '6^3, hcp'. Note that there is a typo in this reference: the results in

Table 4.1 correspond to a reduced pressure of 14.58, *not* 18.74. (The densities quoted for the latter are not in agreement with those quoted in the table; however they do correspond to other hard-sphere simulations the author performed at a reduced pressure of 14.58, which also agree with Speedy's equation of state – Ref. 70 in Ref. [30] – at that pressure).

The following checks should be performed at the completion of the test:

1. The density obtained from the simulation, divided by $\sqrt{2}$, should agree with the value 0.7776(1). To determine the density note that the system contains 216 particles, and that the hcp volume obtained from block averaging and its uncertainty are stored in the variables **equil_V_1** and **sigma_equil_V_1** in the file `state`.

2. The hcp c/a ratio should agree with 1.6323(7). The c/a ratio corresponding to MPI thread '0' can be obtained via the following command, which extracts a measure of the instantaneous c/a ratio from **Lx** and **Lz** in the `data_0` file:

   ```
   $ awk 'NR%14==1 {Lx=$3}; NR%14==3
   {Lz=$3; print $2,(Lz/6)/(Lx/3)}' data_0
   | awk '$1>0 {sum=sum+$2;count=count+1};
   END{print sum/count}'
   ```

   Doing similar for `data_1`, `data_2` and `data_3` will give 4 independent results, one for each MPI thread, which can be combined to give a c/a to be compared against 1.6323(7).

8.5 Embedded atom model (`EAM`)

The relevant `interactions.f95` file for this test is `interactions_EAM.f95`, whose usage is described in Chapter 5.

8.5.1 `test_EAM_1`

This test is exactly the same calculation as the worked example; see Chapter 7 for details.

The following checks should be performed at the completion of the test:

1. The free energy difference between the two systems should be very close to 0: see the variables **equil_DeltaF** and **sigma_equil_DeltaF** in the `state` file.

8.6 Lennard-Jones fluid (`LJ`)

The relevant `interactions.f95` file for these tests is `interactions_LJ.f95`, whose usage is described in Chapter 5.

8.6.1 `test_LJ_1`

This test calculates the average energy per particle $\langle E \rangle$ of the Lennard-Jones fluid in the NVT ensemble using a conventional Monte Carlo simulation. The simulation is performed at reduced temperature $k_B T/\epsilon = 0.85$ and reduced density $\rho\sigma^3 = 0.003$, using a cut-off of 3σ.

The following checks should be performed at the completion of the test:

1. $\langle E \rangle$, after adding the tail correction (see Ref. [1]), should match that of Ref. [34]: -0.03102(6). Note that the test outputs $\langle E \rangle$ with the tail correction to stdout at the completion of the simulation.

8.7 Hard spheres (multiple species) (`HS_multi`)

The relevant `interactions.f95` file for these tests is `interactions_HS_multi.f95`, whose usage is described in Chapter 5.

8.7.1 `test_HS_multi_1`

This test calculates the free energy difference ΔF between two simple multicomponent hard-sphere 'phases'. Both phases have identical supercell dimensions L_x, L_y and L_z, and contain two particles. In phase 1 the two particles both have diameter σ_A, while in phase 2 one particle has diameter σ_A and the other has diameter σ_B; phase 1 consists of two particles belonging to species 'A', while phase 2 consists of one particle belonging to species A and one belonging to species 'B'. ΔF can be evaluated analytically here:

$$\Delta F \equiv F_1 - F_2 = -\frac{1}{\beta} \ln \left[\frac{V - 4\pi\sigma_A^3/3}{V - 4\pi\sigma_{AB}^3/3} \right], \tag{8.13}$$

where

$$\sigma_{AB} = \frac{1}{2}(\sigma_A + \sigma_B), \tag{8.14}$$

β is the thermodynamic beta, and V is the volume of either phase.

Specfiically, we evaluate ΔF for $V = 1$, $\sigma_A = 0.2$, $\sigma_B = 0.1$ and $\beta = 1$, in which case $\Delta F = 0.019846611$. We use canonical sampling, in conjunction with a lattice switch which consists of simply changing the species of one of the particles from A to B in going from phase 1 to 2, and *vica versa* in going from phase 2 to 1.

The following checks should be performed at the completion of the test:

1. ΔF should be 0.019846611: see the variables **equil_DeltaF** and **sigma_equil_DeltaF** in the `state` file. **equil_DeltaF** and **sigma_equil_DeltaF** are also periodically output to stdout.

8.7.2 `test_HS_multi_2`

This test is identical to `test_HS_multi_1` except that multicanonical sampling is used instead of canonical sampling. Note that for this problem canonical sampling is more efficient than multicanonical sampling, and hence the simulation may take a little longer than `test_HS_multi_1` to converge upon the correct value of ΔF.

Appendix A

License

monteswitch

Copyright (c) 2016 Tom L. Underwood

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Bibliography

[1] D. Frenkel and B. Smit, *Understanding Molecular Simulation*. Academic Press, Inc., 2nd ed., 2001.

[2] A. D. Bruce, N. B. Wilding, and G. J. Ackland, "Free energy of crystalline solids: A lattice-switch monte carlo method," *Phys. Rev. Lett.*, vol. 79, pp. 3002–3005, Oct 1997.

[3] A. D. Bruce, A. N. Jackson, G. J. Ackland, and N. B. Wilding, "Lattice-switch monte carlo method," *Phys. Rev. E*, vol. 61, pp. 906–919, Jan 2000.

[4] S. Pronk and D. Frenkel, "Can stacking faults in hard-sphere crystals anneal out spontaneously?," *J. Chem. Phys.*, vol. 110, no. 9, pp. 4589–4592, 1999.

[5] S.-C. Mau and D. A. Huse, "Stacking entropy of hard-sphere crystals," *Phys. Rev. E*, vol. 59, pp. 4396–4401, Apr 1999.

[6] A. N. Jackson, A. D. Bruce, and G. J. Ackland, "Lattice-switch monte carlo method: Application to soft potentials," *Phys. Rev. E*, vol. 65, p. 036710, Mar 2002.

[7] A. N. Jackson and G. J. Ackland, "Lattice-switch monte carlo simulation for binary hard-sphere crystals," *Phys. Rev. E*, vol. 76, p. 066703, Dec 2007.

[8] M. Yang and H. Ma, "Effect of polydispersity on the relative stability of hard-sphere crystals," *The Journal of Chemical Physics*, vol. 128, no. 13, 2008.

[9] M. Marechal and M. Dijkstra, "Stability of orientationally disordered crystal structures of colloidal hard dumbbells," *Phys. Rev. E*, vol. 77, p. 061405, Jun 2008.

[10] P. Raiteri, J. D. Gale, D. Quigley, and P. M. Rodger, "Derivation of an accurate force-field for simulating the growth of calcium carbonate from aqueous solution: A new model for the calcitewater interface," *The Journal of Physical Chemistry C*, vol. 114, no. 13, pp. 5997–6010, 2010.

[11] D. Wilms, N. B. Wilding, and K. Binder, "Transitions between imperfectly ordered crystalline structures: A phase switch monte carlo study," *Phys. Rev. E*, vol. 85, p. 056703, May 2012.

[12] D. Quigley, "Communication: Thermodynamics of stacking disorder in ice nuclei," *The Journal of Chemical Physics*, vol. 141, no. 12, p. 121101, 2014.

[13] S. Bridgwater and D. Quigley, "Lattice-switching monte carlo method for crystals of flexible molecules," *Phys. Rev. E*, vol. 90, p. 063313, Dec 2014.

[14] T. L. Underwood and G. J. Ackland, "Lattice-switch monte carlo: the fccbcc problem," *Journal of Physics: Conference Series*, vol. 640, no. 1, p. 012030, 2015.

[15] N. B. Wilding and A. D. Bruce, "Freezing by monte carlo phase switch," *Phys. Rev. Lett.*, vol. 85, pp. 5138–5141, Dec 2000.

[16] J. R. Errington, "Solidliquid phase coexistence of the lennard-jones system through phase-switch monte carlo simulation," *J. Chem. Phys.*, vol. 120, no. 7, pp. 3130–3141, 2004.

[17] G. C. McNeil-Watson and N. B. Wilding, "Freezing line of the lennard-jones fluid: A phase switch monte carlo study," *J. Chem. Phys.*, vol. 124, no. 6, p. 064504, 2006.

[18] N. B. Wilding, "Freezing parameters of soft spheres," *Mol. Phys.*, vol. 107, no. 4-6, pp. 295–299, 2009.

[19] N. B. Wilding, "Solid-liquid coexistence of polydisperse fluids via simulation," *J. Chem. Phys.*, vol. 130, no. 10, p. 104103, 2009.

[20] P. Sollich and N. B. Wilding, "Crystalline phases of polydisperse spheres," *Phys. Rev. Lett.*, vol. 104, p. 118302, Mar 2010.

[21] N. B. Wilding and P. Sollich, "Phase behavior of polydisperse spheres: Simulation strategies and an application to the freezing transition," *J. Chem. Phys.*, vol. 133, no. 22, p. 224102, 2010.

[22] J. Purton, J. Crabtree, and S. Parker, "Dl_monte: a general purpose program for parallel monte carlo simulation," *Molecular Simulation*, vol. 39, no. 14-15, pp. 1240–1252, 2013.

[23] G. Ackland, K. DMellow, S. Daraszewicz, D. Hepburn, M. Uhrin, and K. Stratford, "The {MOLDY} short-range molecular dynamics package," *Computer Physics Communications*, vol. 182, no. 12, pp. 2587 – 2604, 2011.

[24] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *J. Chem. Phys.*, vol. 21, no. 6, pp. 1087–1092, 1953.

[25] B. A. Berg and T. Neuhaus, "Multicanonical algorithms for first order phase transitions," *Phys. Lett. B*, vol. 267, no. 2, pp. 249–253, 1991.

[26] B. Berg and T. Neuhaus, "Multicanonical ensemble: A new approach to simulate first-order phase transitions," *Phys. Rev. Lett.*, vol. 68, pp. 9–12, Jan 1992.

[27] G. R. Smith and A. D. Bruce, "A study of the multi-canonical monte carlo method," *J. Phys. A: Math. Gen.*, vol. 28, no. 23, p. 6623, 1995.

[28] Y. Iba, "Extended ensemble monte carlo," *International Journal of Modern Physics C*, vol. 12, no. 05, pp. 623–656, 2001.

[29] M. Fitzgerald, R. R. Picard, and R. N. Silver, "Canonical transition probabilities for adaptive metropolis simulation," *Europhys. Lett.*, vol. 46, no. 3, p. 282, 1999.

[30] A. N. Jackson, *Structural Phase Behaviour Via Monte Carlo Techniques.* PhD thesis, University of Edinburgh, Edinburgh, UK, 2001.

[31] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, pp. 3–30, Jan. 1998.

[32] M. S. Daw and M. I. Baskes, "Embedded-atom method: Derivation and application to impurities, surfaces, and other defects in metals," *Phys. Rev. B*, vol. 29, pp. 6443–6453, Jun 1984.

[33] M. I. Mendelev and G. J. Ackland, "Development of an interatomic potential for the simulation of phase transformations in zirconium," *Phil. Mag. Lett.*, vol. 87, no. 5, pp. 349–359, 2007.

[34] `http://www.nist.gov/mml/csd/informatics_research/srsw.cfm` (accessed 3rd April 2016).