The *monteswitch* user manual

Tom L. Underwood

September 11, 2015

Contents

Acknowledgements

The development of *monteswitch* was supported by funding from the Engineering and Physical Sciences Research Council (EPSRC).

Disclaimer

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

List of Abbreviations

LSMC Lattice-switch Monte Carlo

Conventions used in this manual

Throughout this manual we use:

- *italics* to signify file names and directories, including the *monteswitch* package itself;
- `typewriter font` to signify program names, and shell commands and output, including command-line arguments to programs;
- and **bold font** to signify the names of variables within a program.

Chapter 1

Introduction

monteswitch is a package whose primary purpose is to enable lattice-switch Monte Carlo (LSMC) [1, 2] simulations to be performed. It can be used to evaluate free energy differences between pairs of solid phases in NVT and NPT ensembles to a high precision. It can treat single-component atomic (i.e., non-molecular) systems, and pairs of 'phases' which can be represented by an orthorhombic unit cell. Note that the two 'phases' need not be homogeneous crystals, though we expect that this will be the main use for *monteswitch*; possible applications for *monteswitch* include, evaluating the free energy cost of creating point defects and interfaces, as well as evaluating the free energy difference between a crystalline and amorphous solid phase.

The core of *monteswitch* is written in Fortran 95, and there is a version of the main program which is parallelised using MPI for HPC applications. All of the source code pertaining to the calculation of the system's potential energy contained within a single file *interactions.f95* in the main directory. The nature of this file determines which interatomic potential will be used in the main programs, once they are compiled. By default *interactions.f95* pertains to EAM interatomic potentials, though versions of the file are included within the package corresponding to some commonly-used pair potentials. Users are encouraged to create versions of *interactions.f95* which suit their own needs, and hence create bespoke versions of the main programs in which the users' interatomic potentials are 'hard coded'. This approach allows considerable flexibility with regards to the potentials, including their algorithmic efficiency. Of course, the `interactions.f95` file could wrap to external procedures or programs. In this way *monteswitch* can easily be interfaced with other code or programs.

This document describes how to use *monteswitch*, including how to incorporate user-defined interatomic potentials via customisation of the file *interactions.f95*. It is assumed that the user is competent with the unix shell, including the utilities `grep`, `sed` and `awk`. Often we provide examples of commands which utilise the shell and the aforementioned utilities to perform useful *monteswitch*-related tasks. However we never describe why these commands work for the sake of brevity; interested users are referred to the relevant documentation on-

line or elsewhere. In a similar vein, expertise in Fortran 95 is assumed when we describe Fortran-related issues, including how to customise *interactions.f95*. It is also assumed that the user is familiar with the Monte Carlo method (in the sense of computational chemistry), but not necessarily with LSMC. Accordingly in Chapter 2 we describe the theory behind LSMC to a depth which enables the user to competently perform LSMC simulations. The rest of this document petains to *monteswitch* itself. Chapter 3 describes how to compile *monteswitch*. It also provides an overview of the package. Then in Chapters 4 and 5 detailed usage of the programs within *monteswitch* is provided. Chapter 6 is more pedagocial. In this chapter we [WHAT DO WE DO IN THIS CHAPTER] Finally, in Chapter 7 we describe how to customise *interactions.f95*.

Note that this document does not provide details regarding the structure of the source code. In a similar vein, this document does not provide thorough details of how the LSMC method is implemented in *monteswitch*. We direct parties 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. Regarding the LSMC method itself, further information can be found in the references. For further information regarding Monte Carlo simulations in condensed matter physics, see, e.g., [FRENKLE BOOK].

Chapter 2

Background Theory

2.1 Calculating free energy differences

The primary aim of *monteswitch* is two calculate the free energy difference between two solid phases. At this point we should clarify that by 'solid' here we mean that Consider a system which is free to visit two phases 1 and 2 (and only phases 1 and 2) within the NVT ensemble. We consider only the NVT ensemble for now for the sake of simplicity; what follows can easily be adapted to apply to the NPT ensemble. Now, the equilibrium phase is that with the lower Helmholtz free energy F; it is the free energy difference between the phases $\Delta F \equiv F_1 - F_2$ which we wish to evaluate, where F_1 and F_2 denote the free energies of phases 1 and 2. From knowledge of the relative time the system spends in phase 1 and phase 2, which we denote as t_1 and t_2 respectively, one can calculate ΔF using the following equation:

$$\Delta F = \beta^{-1} \ln\left(\frac{t_2}{t_1}\right), \tag{2.1}$$

where $\beta \equiv 1/k_b T$, and k_B denotes Boltzmann's constant. Therefore, in principle, one can calculate ΔF from, e.g. a molecular dynamics simulation, as follows: 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. 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 an *entropic barrier* – a region of phase space associated with states which are very improbable at thermodynamic equilibrium. This entropic 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 *Metropolis Monte Carlo*[3] (which we contrast later to *multicanonical Monte Carlo*), the system is evolved throughout the simulation as follows. Each time step we generate a trial state of the system σ', and attempt to change the system from its current state σ, to the trial state. We change the state from σ to σ' with probability

$$p_{\sigma\to\sigma'} = \max\Big[1, e^{\beta(E_{\sigma'}-E_\sigma)}\Big], \tag{2.2}$$

where E_σ denotes the energy of state σ. The end result is that each state σ is sampled with a probability corresponding to the NVT ensemble:

$$p_\sigma = \frac{1}{Z} e^{-\beta E_\sigma}; \quad Z \equiv \sum_\sigma e^{-\beta E_\sigma} \tag{2.3}$$

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 ΔF (via Eqn. (2.1)) 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. The traditional approach for NVT ensmbles is to perform a 'particle move' to generate a trial state. In this, the trial state differs from the current state in that one of the particle's positions differs. In NPT ensembles particle moves are supplemented by 'volume moves', in which the shape of the entire system is altered, along with a commensurate rescaling of the particle positions. Crucially however, the important properties of the Metropolis algorithm do not rely upon the choice of mechanism used to generate trial states. One has considerable freedom in this regard, and 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 ΔF to be calculated relatively quickly. Such a path would involve frequent transitions between both phases 1 and 2 by 'jumping over' the entropic barrier.

2.2 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. Let $\mathbf{r}_i$ denote the position of particle i. Now, in an NVT emsemble, the set of particle positions $\{\mathbf{r}_i\}$ amounts to a specification of the state of the system. We are interested in evaluating free energy differences between pairs of crystalline solid phases. Accordingly, it makes sense to characterise a given state of the system as belonging to a solid phase α if the positions of the particles 'approximately' form a crystal lattice characteristic of α. In this case one can express the position $\mathbf{r}_i$ of particle i as follows:

$$\mathbf{r}_i = \mathbf{R}_i^\alpha + \mathbf{u}_i, \tag{2.4}$$

where $\mathbf{R}_i^\alpha$ denotes the position of the site in the analogous α crystal lattice which is closest to i, and $\mathbf{u}_i$ denotes the displacement of i from that lattice site. Note that the displacements $\{\mathbf{u}_i\}$ are necessarily small since the particle positions form an approximate α crystal lattice. In a lattice switch from phase α to the 'other' phase α' we transform the underlying lattice $\{\mathbf{R}_i^\alpha\}$ to $\{\mathbf{R}_i^{\alpha'}\}$, *while keeping the particle displacements* $\{\mathbf{u}_i\}$ *unchanged*. [1] The effect is that the trial state belongs to phase α': the positions of the particles in the trial state form an approximate α' crystal lattice. Hence every time a lattice switch is accepted, the system transitions to the 'other' phase.

2.3 Multicanonical Monte Carlo

One might expect that by regularly making lattice switches, the system will regularly transition between phases, allowing ΔF to be efficiently evaluated as described earlier. Unfortunately, if one does this in Metropolis Monte Carlo, one finds that lattice switches are too rarely accepted for this approach to be useful. The problem is that the trial state σ' generated by a lattice switch is almost always of much higher energy than the current state σ, and hence will almost always be rejected by the Metropolis algorithm (Eqn. (2.2)). The solution to this problem is to use *multicanonical Monte Carlo*[4, 5, 6] instead of Metropolis Monte Carlo. The former is a straightforward generalisation of the latter in which, instead of Eqn. (2.2), one has

$$\tilde{p}_{\sigma\to\sigma'} = \max\Big[1, e^{\beta(E_{\sigma'}-E_\sigma)}e^{(\eta_{\sigma'}-\eta_\sigma)}\Big], \tag{2.5}$$

where η_σ, known as the *weight function*, is chosen according to the aims of the simulation. In this case each state σ is sampled with probability

$$\tilde{p}_\sigma = \frac{1}{\tilde{Z}}e^{-\beta E_\sigma}e^{\eta_\sigma}; \quad \tilde{Z} \equiv \sum_\sigma e^{-\beta E_\sigma}e^{\eta_\sigma} \tag{2.6}$$

Using Eqn. (2.3), it can be shown that

$$\tilde{p}_\sigma = \frac{1}{\tilde{Z}'}p_\sigma e^{\eta_\sigma}; \quad \tilde{Z}' = \sum_\sigma p_\sigma e^{\eta_\sigma}, \tag{2.7}$$

where recall that p_σ is the probability of observing state σ in the NVT ensemble. From this it can be seen that if $\eta_\sigma > 0$ then state σ is *over-sampled* relative to Metropolis Monte Carlo. 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. [This can be seen by noting that] [CLUMSY] a multicanonical Monte Carlo

[1] One could in principle alter the particle displacements during a lattice switch. It is conceivable that this would lead to a more efficient exploration of both phases for some systems. However, this is not implemented in *monteswitch*: in *monteswitch* lattice switches always keep the particle displacements unchanged.

simulation can be regarded as a Metropolis Monte Carlo simulation, but if the energy for each state σ were

$$\tilde{E}_\sigma = E_\sigma - \eta_\sigma/\beta \tag{2.8}$$

instead of E_σ. Hence a multicanonical Monte Carlo simulation samples states with probabilities corresponding to, say, an NVT ensemble, but if energy were $\tilde{E}_\sigma$ instead of E_σ. Noting that 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' NVT ensemble – which *is* the case for Metropolis Monte Carlo. Accordingly the time average of X throughout a long multicanonical Monte Carlo simulation is not equivalent to the equilibrium value. Nevertheless one can obtain the equilibrium value of any physical quantity X in 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:

$$\langle X \rangle \approx \frac{1}{\tau} \sum_{t=1}^{\tau} X(t), \tag{2.9}$$

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

$$\langle X \rangle \approx \frac{\sum_{t=1}^{\tau} w(t) X(t)}{\sum_{t=1}^{\tau} w(t)} \tag{2.10}$$

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. (2.7) 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 NVT 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 $w(t) = (e^{\eta(t)}/\tilde{Z}')^{-1}$:

$$\langle X \rangle \approx \frac{\sum_{t=1}^{\tau} e^{-\eta(t)} X(t)}{\sum_{t=1}^{\tau} e^{-\eta(t)}}, \tag{2.11}$$

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

Why is this helpful to us? As mentioned above, most states are such that lattice switches will be almost certainly be rejected from them. There are, however, a small number of states from which a lattice switch yields a trial state which is of comparable energy to σ. From such states a lattice switch has a good chance of being accepted. (Note that the same is also true for the trial state corresponding to a lattice switch: a lattice switch from the trial state, which brings us back to the current state, would also have a good chance of success). 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 result is that lattice switches are accepted reasonably often, encouraging switching between the two phases. Thus the multicanonical simulation samples both phases in a reasonable simulation time, which enables us to obtain 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. (2.11), enabling us to obtain ΔF via Eqn. (2.1).

How should the weight function be engineered such that gateway states are over-sampled? Let us define the quantity

$$M(\{\mathbf{u}_i\}) = E(\{\mathbf{R}_i^1 + \mathbf{u}_i\}) - E(\{\mathbf{R}_i^2 + \mathbf{u}_i\}), \tag{2.12}$$

where $E(\{\mathbf{r}_i\})$ denotes the energy of the state with positions $\{\mathbf{r}_i\}$. The first term on the right-hand side is the energy associated with the state in phase 1 where the displacements are $\{\mathbf{u}_i\}$ for phase 1, and the second term is the analogous quantity for phase 2. Note that the states $\{\mathbf{r}_i\} = \{\mathbf{R}_i^1 + \mathbf{u}_i\}$ and $\{\mathbf{r}_i\} = \{\mathbf{R}_i^2 + \mathbf{u}_i\}$ differ by a lattice switch: performing a lattice switch from the former yields the latter and *vice versa*. Note also that $M(\{\mathbf{u}_i\}) = 0$ if the energies of both these states are identical. In this case the energy cost of a lattice switch is zero, an attempted lattice switch from either state would be successful, and hence both states are gateway states. By contrast, if $|M(\{\mathbf{u}_i\})| \gg 0$, then the two states have significanly different energies. In this case, while switching from the higher-energy state to the lower-energy state is guaranteed, the converse is not: the two states are not concordant with switching *to and from* both phases. $|M(\{\mathbf{u}_i\})|$ therefore provides a measure of how 'un-gateway-like' a state with displacements $\{\mathbf{u}_i\}$ is, with zero corresponding to a gateway state. 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. 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 ΔF; 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 ΔF as described above.

Given that we are constraining the weight function to take the same value for all states with the same M, it is convenient to define a macrostate, which we also denote as M, which is comprised of all such states. Thus in implementing multicanonical Monte Carlo in the computer, it is sufficient to store the value of the weight function for each M considered, instead of for each state - which is intractable. Of course, it is impossible to treat M as a continuous variable in the computer; in reality the considered range of M will be split into N_{macro} bins each corresponding to a range of M. Each bin itself corresponds to a macrostate: the macrostate is the collection of states corresponding to the range of M covered by the bin. We will henceforth explicitly take the discretisation of M into account. Let $\mathcal{M}$ denote the macrostate corresponding to the $\mathcal{M}$th bin, where $\mathcal{M} = 1, 2, \ldots, N_{\mathrm{macro}}$. Accordingly let $\eta_{\mathcal{M}}$ denote the weight function for macrostate $\mathcal{M}$.

2.4 Weight function generation

We will now describe how weight functions can be generated for use in a production simulation. All methods for generating a weight function share the same notion of the 'ideal' weight function, which we denote as $\eta^*_{\mathcal{M}}$: it is such that all macrostates within an 'allowed range' are 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,

$$\tilde{p}_{\mathcal{M}} = \sum_{\sigma \in \mathcal{M}} \tilde{p}_\sigma. \tag{2.13}$$

Thus an ideal weight function would yield $\tilde{p}_{\mathcal{M}} = C$ for all $\mathcal{M}$ within the allowed range, where C is some constant. The ideal weight function, or at least a close estimate of it, can be determined in various ways; below we describe the methods supported by *monteswitch*.

We emphasise that it is unnecessary to use an 'exact' 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. (2.11)) only in that it determines the efficiency with which phase space is explored. A weight function close to the ideal is desirable because it enables gateway states to be visited often, hence allowing both phases to be explored regularly. On the other hand 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*.

2.4.1 Visited states method

The *visited states method* is an iterative 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 Monte Carlo is used, 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

$$\eta_{\mathcal{M}}^{(n+1)} = \eta_{\mathcal{M}}^{(n)} - \ln\left\{ \frac{H_{\mathcal{M}}^{(n)} + 1}{\sum_{\mathcal{M}'} \left(H_{\mathcal{M}'}^{(n)} + 1 \right)} \right\} + k, \tag{2.14}$$

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, [2] which we choose such that the minimum value of $\eta_{\mathcal{M}}^{(n+1)}$ over all $\mathcal{M}$ is 0. 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 $\mathcal{M}$.

The visited states method converges to the ideal weight function very slowly. For this reason we do not recommend that it is the sole method used to generate weight function with *monteswitch*, but primarily, or perhaps entirely, use the other methods for generating weight functions described in a moment. These other methods are much more efficient at obtaining a weight function close to the ideal. However, these methods, as implemented in *monteswitch*, can never converge on the ideal, while the visited states method can. 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] Only differences in the values of the weight function between states are 'physically significant'; the absolute values of the weight function are not. See, e.g., Eqn. (2.5).

2.4.2 Transition matrix method

It can be shown that the ideal weight function obeys [REFERENCE: BRUCE PAPER]

$$\eta^*_{\mathcal{M}} = C' - \ln(p_{\mathcal{M}}), \tag{2.15}$$

where $p_{\mathcal{M}}$ denotes the probability of the system being in macrostate $\mathcal{M}$ in the true NVT ensemble (sampled from by Metropolis Monte Carlo, but not necessarily multicanonical Monte Carlo), and C' is a constant. Hence the ideal weight function can be determined 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 transition matrix method we keep track of the transitions between all pairs of macrostates, use that information to calculate $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$, then $p_{\mathcal{M}}$, and finally $\eta^*_{\mathcal{M}}$ via the above equation.

The first step is to calculate $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$. To do this, we keep a histogram of the number of *supposed* transitions between each pair of macrostates – where the meaning of 'inferred' will become clear in a moment. Let $H_{\mathcal{M}\mathcal{M}'}$ denote the number of inferred transitions between macrostates $\mathcal{M}$ and $\mathcal{M}'$. Again, note that the aforementioned transitions pertain to the true ensemble, *not to the multicanonical one*. We update $H_{\mathcal{M}\mathcal{M}'}$ and $H_{\mathcal{M}\mathcal{M}}$ every time a trial state is generated which, if accepted, would take the system from macrostate $\mathcal{M}$ to macrostate $\mathcal{M}'$. The update scheme is as follows

$$\begin{aligned} H_{\mathcal{M}\mathcal{M}'} &\to H_{\mathcal{M}\mathcal{M}'} + p_{\text{Metro}}, \\ H_{\mathcal{M}\mathcal{M}} &\to H_{\mathcal{M}\mathcal{M}} + 1 - p_{\text{Metro}}, \end{aligned} \tag{2.16}$$

where p_{Metro} is the probability of the move being accepted according to the *Metropolis algorithm* (Eqn. (2.2)) – *not* the analogous algorithm for multicanonical Monte Carlo (Eqn. (2.5)). To restate, for every trial move, we determine the probability of the move being accepted according to the Metropolis algorithm, and update the histogram $H_{\mathcal{M}\mathcal{M}'}$, not with the number of *observed* transitions, but with the number of transitions *supposed* to occur given the probability given by the Metropolis algorithm. The benefit of this approach is that $H_{\mathcal{M}\mathcal{M}}$ can be obtained even if Metropolis sampling is not used, i.e. if multicanonical sampling is used. To elaborate, note that in the above equation while the Metropolis probability p_{Metro} is used to update $H_{\mathcal{M}\mathcal{M}'}$, the actual algorithm used to determine whether the move is accepted or not does not necessarily have to be the Metropolis algorithm – the multicanonical algorithm could be used. This is useful because Metropolis sampling alone would not explore the full range of macrostates $\mathcal{M}$, which is necessary to 'fill up' the matrix $H_{\mathcal{M}\mathcal{M}'}$. We discuss methods for exploring the phase space in a transition matrix simulation in a moment. From the matrix $H_{\mathcal{M}\mathcal{M}'}$ we obtain an estimate for $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$ using the following equation:

$$\mathcal{T}_{\mathcal{M}\mathcal{M}'} \approx \frac{H_{\mathcal{M}\mathcal{M}'} + 1}{\sum_{\mathcal{M}''} \left(H_{\mathcal{M}\mathcal{M}''} + 1\right)}. \tag{2.17}$$

The second step is to use our estimate of $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$ to calculate $p_{\mathcal{M}}$. It can be shown that the macrostates obey a detailed balance condition

$$\mathcal{T}_{\mathcal{M}'\mathcal{M}} p_{\mathcal{M}'} = \mathcal{T}_{\mathcal{M}\mathcal{M}'} p_{\mathcal{M}}. \tag{2.18}$$

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

$$p_{(\mathcal{M}+1)} = \frac{\mathcal{T}_{\mathcal{M}(\mathcal{M}+1)}}{\mathcal{T}_{(\mathcal{M}+1)\mathcal{M}}} p_{\mathcal{M}}. \tag{2.19}$$

Using this equation, $p_{\mathcal{M}}$ can be obtained from the matrix $\mathcal{T}_{\mathcal{M}\mathcal{M}'}$ using 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. (2.19)). This in turn can be used to obtain p_3 ($\mathcal{M} = 2$ in Eqn. (2.19)), 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

$$\sum_{\mathcal{M}=1}^{N_{\text{macro}}} p_{\mathcal{M}} = 1, \tag{2.20}$$

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. (2.15). 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. (2.17), 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 Metropolis probabilities, 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 *macrostate barriers* 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 'barrier' preventing the system from moving into an adjacent macrostate is moved such that the system is free to transition into the 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, 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 that 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 in the true NVT 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 Differences in *monteswitch* between platforms

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'. [ALSO USED CRAY?]

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. For example, [INSERT EXAMPLE: CRAY VS. GFORTRAN]

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 number of test cases for the main programs. The file *GUIDE_TO_TESTS.txt* within this directory contains a detailed description 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*. [GUIDE IN THIS DIRECTORY?]

3.3.2 Modules for various interatomic potentials

The files beginning with *interactions_* (and ending with *.f95*) are the Fortran modules [1] 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* (note that there is no '_' in the file name); it is only the contents of this file which is used in the package –

[1] Strictly speaking these files are not actually modules: the file *interactions.f95* is inserted into the module *monteswitch_mod.f95* using the Fortran command INCLUDE.

after compilation. Hence to implement a specific interatomic potential one must copy the desired '*interactions_* file' to *interactions.f95*, and then (re-)compile the package.

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

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 (LSMC) 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.

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

There are three allowed command-line arguments to `monteswitch`:

- `-new`
- `-resume`
- `-reset`

The command-line argument `-new` (Example usage: `monteswitch -new`) runs a new simulation. The simulation is initialized using information contained in the input files *params_in*, *lattices_in* and *interactions_in* in the current directory. The file *lattices_in* contains specifications of two microstates. This file determines the two phases which will be considered, and defines the lattice switch which will be used in the simulation. The file *interactions_in* contains variables pertaining to the interatomic potential. All other variables which determine the nature of the simulation are contained within the file *params_in*. We will now describe the format of these files.

4.2 Input file: *lattices_in*

The input file *lattices_in* contains specifications of two microstates (i.e., supercell dimensions and particle positions), one for each phase. The format of this file 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 positions of the particles within the supercell for phase 1 *in fractional coordinates*. The remaining lines similarly specify the supercell 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 positions of the particles within the supercell for phase 2. Here is an example of a *lattices_in* file corresponding to phase 1 being an 8-atom bcc supercell and phase 2 being (an 8-atom) fcc supercell: [1]

```
bcc-hcp, rho =   0.50000000000000000      , nx,ny,nz =            1           1           2
         8
  2.2449240966187456
  1.5874010519681991
  4.4898481932374912
  0.0000000000000000        0.0000000000000000        0.0000000000000000
 0.50000000000000000       0.50000000000000000        0.0000000000000000
 0.50000000000000000        0.0000000000000000       0.25000000000000000
  0.0000000000000000       0.50000000000000000       0.25000000000000000
  0.0000000000000000        0.0000000000000000       0.50000000000000000
 0.50000000000000000       0.50000000000000000       0.50000000000000000
 0.50000000000000000        0.0000000000000000       0.75000000000000000
  0.0000000000000000       0.50000000000000000       0.75000000000000000
  2.4494897427831779
  1.4142135623730949
  4.6188021535170050
  0.0000000000000000        0.0000000000000000        0.0000000000000000
 0.50000000000000000       0.50000000000000000        0.0000000000000000
 0.33333333333333331        0.0000000000000000       0.25000000000000000
 0.83333333333333348       0.50000000000000000       0.25000000000000000
  0.0000000000000000        0.0000000000000000       0.50000000000000000
 0.50000000000000000       0.50000000000000000       0.50000000000000000
 0.33333333333333331        0.0000000000000000       0.75000000000000000
 0.83333333333333348       0.50000000000000000       0.75000000000000000
```

The two microstates in *lattices_in* are used in two ways by **monteswitch**. Firstly, they serve as the initial microstate: if the simulation is to be initialised in phase 1, then it will be initialised in the phase 1 microstate specified in *lattices_in*; and similarly for phase 2. Secondly, they define the lattice switch. Let

[1] This example was created using the `lattices_in_bcc_hcp` program

$\tilde{\mathbf{R}}_i^\alpha$ denote the position of the lattice site i for phase α in fractional coordinates. (Recall our earlier discussion of lattice sites in Sec. 2.2). To elaborate, the position of the lattice site in 'real' coordinates is

$$\mathbf{R}_i^\alpha = (L_x^\alpha \tilde{\mathbf{R}}_{i,x}^\alpha, L_y^\alpha \tilde{\mathbf{R}}_{i,y}^\alpha, L_z^\alpha \tilde{\mathbf{R}}_{i,z}^\alpha), \tag{4.1}$$

where L_x^α denotes the current x-dimension of the supercell for phase α – the current phase the system is in – and similarly for L_y^α and L_z^α. Now, the vectors $\tilde{\mathbf{R}}_i^\alpha$ used by the main program are the fractional positions specified in *lattices_in*: $\tilde{\mathbf{R}}_i^\alpha$ is taken to be the ith specified fractional coordinates pertaining to phase α. With this in mind, the lattice switch from phase 1 to 2 in a simulation consists of:

- A scaling of the x-dimension of the supercell of which reflects the relative sizes of the x-dimensions of both lattices specified in *lattices_in*. E.g., in the above example the x-dimension would be scaled by a factor of $2.2449/2.2449 = 1$, i.e., not at all.
- A similar scaling for the y-dimension of the supercell. E.g., in the above example the y-dimension would be scaled by a factor of $1.414/1.587 = 0.891$.
- A similar scaling for the z-dimension of the supercell. E.g., in the above example the y-dimension would be scaled by a factor of $4.619/4.490 = 1.029$.
- A change in the fractional position of each lattice site i from $\tilde{\mathbf{R}}_i^1$ to $\tilde{\mathbf{R}}_i^2$.

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

Note that in a `monteswitch` LSMC simulation, the vectors $\tilde{\mathbf{R}}_i^\alpha$ do not change. This means that if the volume of the system is expanded in an NPT simulation, then there is a commensurate scaling of the lattice sites $\mathbf{R}_i^\alpha$. Furthermore, `monteswitch` only supports lattice-switches which preserve the volume of the system. Accordingly an error is returned by `monteswitch` if one specifies two microstates in *lattices_in* which have different volumes.

4.3 Input file: *interactions_in*

The file *interactions_in* contains variables which parametrise the interatomic potential. The format of this file depends on the specific *interactions.f95* file used when compiling *monteswitch*. [WE DISCUSS THE FORMAT FOR PARTICULAR INTERACTIONS FILES IN SECTION...]

4.4 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 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 variables which must appear in a *params_in*, as well as a description of their function, are given in Table 4.4. *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 *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	Thermodynamic beta.

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. Of course this should 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. We recommend that this be set to 1.
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 center-of-momentum reference frame; 'F' uses the lab frame. Using the centre-of-momentum frame prevents 'drift' in the centre-of-momentum. This is especially important for lattice-switch simulations. Hence this should be set to 'T' unless the user has a very good reason not to.

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. Note that currently lattice moves are forbidden in conjunction with unconstrained volume moves, though this may be relaxed in the future.
15	**vol_freq**	INTEGER	Number of volume moves performed per sweep on average.
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.

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 consideration 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 microstates 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. Also, the current block with regards to the calculation of equilibrium quantities is disregarded.

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 microstate is not calculated exactly because it is very computationally expensive and unnecessary. Instead the energy *change* with respect to the current microstate is calculated. This is far less demanding to calculate. If the move is accepted this change is *ammended* 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 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 4.5.
27	**output_file_Lx**	LOGICAL	See Section 4.5.
28	**output_file_Ly**	LOGICAL	See Section 4.5.
29	**output_file_Lz**	LOGICAL	See Section 4.5.
30	**output_file_V**	LOGICAL	See Section 4.5.
31	**output_file_R_1**	LOGICAL	See Section 4.5.
32	**output_file_R_2**	LOGICAL	See Section 4.5.
33	**output_file_u**	LOGICAL	See Section 4.5.
34	**output_file_lattice**	LOGICAL	See Section 4.5.
35	**output_file_E**	LOGICAL	See Section 4.5.
36	**output_file_M**	LOGICAL	See Section 4.5.
37	**output_file_eta**	LOGICAL	See Section 4.5.
38	**output_file_moves_lattice**	LOGICAL	See Section 4.5.
39	**output_file_accepted_moves_lattice**	LOGICAL	See Section 4.5.
40	**output_file_moves_part**	LOGICAL	See Section 4.5.
41	**output_file_accepted_moves_part**	LOGICAL	See Section 4.5.
42	**output_file_moves_vol**	LOGICAL	See Section 4.5.
43	**output_file_accepted_moves_vol**	LOGICAL	See Section 4.5.
44	**output_file_rejected_moves_M_OOB**	LOGICAL	See Section 4.5.
45	**output_file_M_OOB_high**	LOGICAL	See Section 4.5.
46	**output_file_M_OOB_low**	LOGICAL	See Section 4.5.
47	**output_file_barrier_macro_low**	LOGICAL	See Section 4.5.
48	**output_file_barrier_macro_high**	LOGICAL	See Section 4.5.
49	**output_file_rejected_moves_M_barrier**	LOGICAL	See Section 4.5.
50	**output_file_moves_since_lock**	LOGICAL	See Section 4.5.
51	**output_file_melts**	LOGICAL	See Section 4.5.

52	**output_file_equil_DeltaF**	LOGICAL	See Section 4.5.
53	**output_file_sigma_equil_DeltaF**	LOGICAL	See Section 4.5.
54	**output_file_equil_H_1**	LOGICAL	See Section 4.5.
55	**output_file_sigma_equil_H_1**	LOGICAL	See Section 4.5.
56	**output_file_equil_H_2**	LOGICAL	See Section 4.5.
57	**output_file_sigma_equil_H_2**	LOGICAL	See Section 4.5.
58	**output_file_equil_V_1**	LOGICAL	See Section 4.5.
59	**output_file_sigma_equil_V_1**	LOGICAL	See Section 4.5.
60	**output_file_equil_V_2**	LOGICAL	See Section 4.5.
61	**output_file_sigma_equil_V_2**	LOGICAL	See Section 4.5.
62	**output_file_equil_umsd_1**	LOGICAL	See Section 4.5.
63	**output_file_sigma_equil_umsd_1**	LOGICAL	See Section 4.5.
64	**output_file_equil_umsd_2**	LOGICAL	See Section 4.5.
65	**output_file_sigma_equil_umsd_2**	LOGICAL	See Section 4.5.
66	**output_stdout_period**	INTEGER	Period (sweeps) at which information is output to stdout. See Section 4.5.
67	**output_stdout_Lx**	LOGICAL	See Section 4.5.
68	**output_stdout_Ly**	LOGICAL	See Section 4.5.
69	**output_stdout_Lz**	LOGICAL	See Section 4.5.
70	**output_stdout_V**	LOGICAL	See Section 4.5.
71	**output_stdout_R_1**	LOGICAL	See Section 4.5.
72	**output_stdout_R_2**	LOGICAL	See Section 4.5.
73	**output_stdout_u**	LOGICAL	See Section 4.5.
74	**output_stdout_lattice**	LOGICAL	See Section 4.5.
75	**output_stdout_E**	LOGICAL	See Section 4.5.
76	**output_stdout_M**	LOGICAL	See Section 4.5.
77	**output_stdout_eta**	LOGICAL	See Section 4.5.
78	**output_stdout_moves_lattice**	LOGICAL	See Section 4.5.

79	**output_stdout_accepted_moves_lattice**	LOGICAL	See Section 4.5.
80	**output_stdout_moves_part**	LOGICAL	See Section 4.5.
81	**output_stdout_accepted_moves_part**	LOGICAL	See Section 4.5.
82	**output_stdout_moves_vol**	LOGICAL	See Section 4.5.
83	**output_stdout_accepted_moves_vol**	LOGICAL	See Section 4.5.
84	**output_stdout_rejected_moves_M_OOB**	LOGICAL	See Section 4.5.
85	**output_stdout_M_OOB_high**	LOGICAL	See Section 4.5.
86	**output_stdout_M_OOB_low**	LOGICAL	See Section 4.5.
87	**output_stdout_barrier_macro_low**	LOGICAL	See Section 4.5.
88	**output_stdout_barrier_macro_high**	LOGICAL	See Section 4.5.
89	**output_stdout_rejected_moves_M_barrier**	LOGICAL	See Section 4.5.
90	**output_stdout_moves_since_lock**	LOGICAL	See Section 4.5.
91	**output_stdout_melts**	LOGICAL	See Section 4.5.
92	**output_stdout_equil_DeltaF**	LOGICAL	See Section 4.5.
93	**output_stdout_sigma_equil_DeltaF**	LOGICAL	See Section 4.5.
94	**output_stdout_equil_H_1**	LOGICAL	See Section 4.5.
95	**output_stdout_sigma_equil_H_1**	LOGICAL	See Section 4.5.
96	**output_stdout_equil_H_2**	LOGICAL	See Section 4.5.
97	**output_stdout_sigma_equil_H_2**	LOGICAL	See Section 4.5.
98	**output_stdout_equil_V_1**	LOGICAL	See Section 4.5.
99	**output_stdout_sigma_equil_V_1**	LOGICAL	See Section 4.5.
100	**output_stdout_equil_V_2**	LOGICAL	See Section 4.5.
101	**output_stdout_sigma_equil_V_2**	LOGICAL	See Section 4.5.
102	**output_stdout_equil_umsd_1**	LOGICAL	See Section 4.5.
103	**output_stdout_sigma_equil_umsd_1**	LOGICAL	See Section 4.5.
104	**output_stdout_equil_umsd_2**	LOGICAL	See Section 4.5.
105	**output_stdout_sigma_equil_umsd_2**	LOGICAL	See Section 4.5.

106	**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 4.5.
107	**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.
108	**update_eta_sweeps**	INTEGER	Period (seeps) at which the weight function is updated.
109	**update_trans**	LOGICAL	'T' results in the transition matrix being updated; 'F' results in it not being updated.
110	**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). ELABORATE.
111	**enable_barriers**	LOGICAL	'T' enables macrostate barriers; 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**).

112	**barrier_dynamics**	CHARACTER(30)	Flag determining how the macrostate barriers will evolve. 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.
113	**lock_moves**	INTEGER	The number of moves to lock the system into one macrostate for if macrostate barriers are used. This should be greater than 0.
114	**calc_equil_properties**	LOGICAL	'T' enables calculation of equilibrium quantities using block analysis. ELABORATE.
115	**block_sweeps**	INTEGER	The number of sweeps which comprise a 'block' which will be used to evaluate equilibrium properties and their uncertainties as just described. This should be greater than 0.

4.5 Output: stdout, *state* and *data*

[TIDY UP; NOTE CASE SENSITIVE KEY WORDS]

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. [eta is not an internal variable, only eta_grid is] The only exceptions to all of this are **output_file_eta** and **output_stdout_eta**, for which **eta** does means the value of the weight function for the current microstate as opposed to the entire weight function. This point is reiterated below.

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 4.6 and 4.7), 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 5.2). 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 4.5 contains a list of `monteswitch` internal variables, not already covered by Table 4.4, 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 4.4 or 4.5 – 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*.

Table 4.2: Useful variables found in .

Variable	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. This is **E_1** minus **E_2**.
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 macrostate barriers are enabled).
barrier_macro_high	INTEGER	The macrostate number corresponding to the highest currently allowed macrostate (relevant only when macrostate barriers are 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 macrostate barriers are enabled).
block_counts	INTEGER	The total number of 'blocks' considered so far for evaluating equilibrium quantities and their uncertainties.
equil_DeltaF	REAL	The free energy difference between the phases ($F_1 - F_2$; extensive) evaluated using block analysis.
sigma_equil_DeltaF	REAL	The uncertainty in **equil_DeltaF**.
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.
equil_H_1	REAL	The energy (for NVT simulations) or enthalpy (for NPT simulations) of phase 1 evaluated using block analysis.
equil_H_2	REAL	The energy (for NVT simulations) or enthalpy (for NPT simulations) of phase 2 evaluated using block analysis.

sigma_equil_H_1	REAL	The uncertainty in **equil_H_1**.
sigma_equil_H_2	REAL	The uncertainty in **equil_H_2**.
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.
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.
equil_V_1	REAL	The volume of phase 1 evaluated using block analysis.
equil_V_2	REAL	The volume of phase 2 evaluated using block analysis.
sigma_equil_V_1	REAL	The uncertainty in **equil_V_1**.
sigma_equil_V_2	REAL	The uncertainty in **equil_V_2**.
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.
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.
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.
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.

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 transition matrix mentioned in Section [INSERT SECTION NUMBER].
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 analysis. Note that blocks are disregarded if during the block the system melts.
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 analysis. Note that blocks are disregarded if during the block the system melts.

sigma_equil_umsd_1	REAL(**n_part**)	**sigma_equil_umsd_1**(n) is the uncertainty in **equil_umsd_1**(n).
sigma_equil_umsd_2	REAL(**n_part**)	**sigma_equil_umsd_2**(n) is the uncertainty in **equil_umsd_2**(n).

4.6 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.

4.7 '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 microstate of the system specified in *state*. By contrast, the command-line argument `-new` initialises the microstate to be such that the particles form a perfect crystal lattice, which usually does not correspond to an equilibrated microstate. 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 collection transition 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.

4.8 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.

4.9 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 statistics from all are combined, and the results are stored in the file *state*. The microstate stored in state corresponds to the microstate in *state_0*. Note that, since *state*-format files can be large, by default the files *state_0*, *state_1*, *state_2*, etc. created by **monteswitch_mpi** are empty. This can be disabled if `-explicit` is used as the second command line argument to `monteswitch_mpi`, e.g., `mpiexec -n 4 monteswitch_mpi -new -explicit`. Similar applies to the arguments `-resume` and `-reset`.

Further details of the parallelisation are as follows. All replicas are always initialised to be in the same microstate. 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 microstate 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 microstate* when the `-resume` argument is used with `monteswitch_mpi`. In future we hope to allow 'true' checkpointing whereby a simulation picks up exactly where it left off , i.e., by having the nth replica initialised from the file *state_n* instead of having all replicas initialised from *state*, and to ammend *data_n* with information pertaining to replica n during the resumed simulation. Note that the files *data_n* are thus 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 microstate 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 the 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*.

Chapter 5

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.

5.1 Programs for generating *lattices_in* files

The programs `lattices_in_hcp_fcc`, `lattices_in_bcc_fcc` and `lattices_in_bcc_hcp` generate *lattices_in* files corresponding to, respectively, hcp–fcc, bcc–fcc and bcc–hcp pairs of phases. Each program takes four arguments. The first is the density, i.e., the number of atoms per unit volume, of the pair of microstates to construct. The second, third and fourth arguments are the 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 program outputs the text for the *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 `lattices_in_hcp_fcc 0.5 2 3 5 > lattices_in`.

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 '1' for the second, third and fourth arguments.

5.1.1 `lattices_in_hcp_fcc`

The unit cell here contains 12 atoms. The atoms are spread over 6 planes in the z-direction; each plane contains two atoms. The positions of the atoms 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 lattice 1 corresponds to hcp and lattice 2 corresponds to fcc.

5.1.2 `lattices_in_bcc_fcc`

The unit cell here is the conventional 2-atom 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 lattice 1 corresponds to bcc and lattice 2 corresponds to fcc.

5.1.3 `lattices_in_bcc_hcp`

The unit cell here contains 4 atoms. The bcc unit cell is the 4-atom 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 lattice 1 corresponds to bcc and lattice 2 corresponds to hcp.

5.2 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, and are as follows.

5.2.1 `-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.

5.2.2 `-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 lattice type 1, and the third is the corresponding number of counts for lattice type 2.

5.2.3 `-extract_pos`

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.

5.2.4 `-extract_R_1`

Extract the current positions of the lattice sites for lattice type 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.

5.2.5 `-extract_R_2`

Extract the current positions of the lattice sites for lattice type 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.

5.2.6 `-extract_u`

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.

5.2.7 `-calc_rad_dist bins`

Calculate the radial distribution function, based on the current microstate, 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.

5.2.8 `-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** 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**.

Chapter 6

Worked example

We now provide a worked example to elucidate how the *monteswitch* package as a whole can be used to solve a 'real' problem. Specifically, we use *monteswitch* to calculate the free energy difference between the bcc and hcp phases of Zr, modeled using the EAM potential found in [INSERT PAPER], at 1234K and 0 pressure. In our calculation we use mobile macrostate barriers to quickly determine a reasonable weight function, and `monteswitch_mpi` to parallelise the calculations. Below we go through the entire procedure we used to obtain the free energy difference. Of course, the procedure we used will not be applicable to all problems. For this reason we anticipate problems that the user may encounter more generally, and provide advice accordingly.

The example can be found in the *Examples/EAM_Zr_bcc_hcp* directory of the *monteswitch* package. Within *Examples/EAM_Zr_bcc_hcp* there are a number of subdirectories which correspond to different simulations, all of which are necessary to obtain the free energy difference. The aforementioned directories are *bcc_preliminary*, *hcp_preliminary*, *weight_function_generation*, *weight_function_verification* and *production_simulation*. We will describe each of these simulations in turn in a moment. For now we wish to elaborate upon the contents of the subdirectories. Firstly, each subdirectory contains the input and output files for the corresponding simulation. Recall that the input files for 'new' `monteswitch` and `monteswitch_mpi` simulations (i.e., using the `-new` argument) are *params_in*, *interactions_in* and *lattices_in*, and that the output files are *state* and *data* for serial simulations (using the `monteswitch` program), and *data_0*, *state_0*, *data_1*, *state_1*, *data_2*, *state_2*, etc. and *state* for MPI simulations (using the `monteswitch_mpi` program). Regarding the input files, those for new simulations differ only in the *params_in* files: the *interactions_in* and *lattices_in* files, which define, respectively, the interatomic potential, and the phases and lattice switch, for a simulation, are the same for all simulations. Hence we have annotated the *params_in* file corresponding to each simulation to highlight the important variable choices. Recall that for resumed simulations (which use the `-resume` or `-reset` arguments) an existing *state* file is the input file. To distinguish the input input state file from the state file output by such simula-

tions, we name the input file *state_start*. To restate, in directories corresponding to resumed simulations, *state_start* is the input state file for the simulation, while *state* is the output state file. Finally, recall also that `monteswitch` and `monteswitch_mpi` output information to stdout. In each subdirectory the file 'stdout' contains a copy of the output to stdout – if there is any. Furthermore, the subdirectories may contain other files (*.dat* files) which correspond to data obtained from post-processing. These files, and how they were created, will be described at the correct time.

6.1 The `interactions_in` and `lattices_in` files

Before performing any simulations, we must first construct the input files *interactions_in* and *lattices_in* which specify, respectively, the interatomic potential, and pair of supercells we will use in all simulations. For the EAM version of monteswitch (which uses *interactions_EAM.f95* as *interactions.f95* - which is the case by default) the *interactions_in* file must be a 'setfl'-format description of the EAM potential of interest. We wish to use the aforementioned Zr potential. It can be seen by inspecting the file that the *interactions_in* in the current directory (and the *interactions_in* files in all subdirectories - they are all identical) correspond to a 'setfl'-format description of the aforementioned Zr potential - as required.

Consider now the *lattices_in* file. Recall that this contains specifications of the supercells for both phases; the atomic positions specified in *lattices_in* define the crystal lattices which define each phase. Given that we are considering the bcc and hcp phases, we can use the `lattices_in_bcc_hcp` program to generate a *lattices_in* file, as opposed to creating it from scratch. We know the density of both phases is about 0.043Å^{-3} at the temperature under consideration. Hence we use this as the first argument to `lattices_in_bcc_hcp`. We generated the file using the command.

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

Recall that the second, third and fourth arguments to *lattices_in_bcc_hcp* give how many unit cells to tile in the x-, y- and z-directions. As can be seen from inspection of the resulting *lattices_in* file, a copy of which can be found in the current directory, the above command yields a supercell of reasonable size (384 atoms), and supercell dimensions which are greater than twice the cut-off of the EAM potential (which is 7.6Å- the last value on the second line of *interactions_in*) as is required to avoid finite size effects associated with 'missing' neighbours.

6.2 Preliminary simulations: *bcc_preliminary* and *hcp_preliminary*

Now we turn to simulations. Before performing lattice-switch Monte Carlo simulations, it is necessary to first perform preliminary conventional Monte Carlo simulations for each phase, bcc and hcp. The directories *bcc_preliminary* and *hcp_preliminary* correspond to these simulations. The aim of these simulations is to determine:

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

6.2.1 `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 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 false 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 true, 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. 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 with a new informed 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) described above.

- **output_file_period** is set to 1, which means information is output to the *data* file every sweep. Normally outputing information so frequently might be considered 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_V** are set to true: the system's volume, energy and order parameter are periodically output to the *data* file. We will use the first three quantities to determine the equilibration time; we will examine how the order parameter varies with time to obtain an idea of the range of order parameters exhibited by each phase at equilibrium, which in turn will inform the values of **M_grid_min** and **M_grid_max** to be used in subsequent 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 false; otherwise the system will be locked into a certain macrostate - which we don't want.

6.2.2 Running the simulations

The simulation is run using the `-new` argument. Given that the simulation is short, it is unnecessary to use MPI; the `monteswitch` executable should be used. The following command invokes the executable (from within the *Examples/EAM_Zr_bcc_hcp/bcc_preliminary* or *Examples/EAM_Zr_hcp_preliminary* directories within the *monteswitch* package itself):

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

6.2.3 Estimating the (one-phase canonical) equilibration time

The simulation is run using the `-new` argument. Given that the simulation is short, it is unnecessary to use MPI; the `monteswitch` executable should be used. The following command invokes the executable (from within the *Examples/EAM_Zr_bcc_hcp/bcc_preliminary* or *Examples/EAM_Zr_bcc_hcphcp_preliminary* directories within the monteswitch package itself):

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

6.2.4 Extracting the appropriate order parameter range

One can create a plot of the order parameter M vs. number of sweeps similarly to above:

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

Plotting this file immediately gives an idea of the range of order parameters exhibitted at equilibrium for each phase. We see that for bcc the exhibitted range is -62 to 20, and for hcp the range is 14 to 32. Hence the appropriate range of order parameters which encompasses both phases is -62 to 32. We will use this in subsequent simulations.

[NOTE ABOUT THE RANGE OBTAINED IN THIS WAY - IT IS PERHAPS BIGGER IN REALITY] One could argue that...

6.2.5 Troubleshooting

MELTING; COULD THE PHASE HAVE 'MELTED' INTO A NEARBY SOLID PHASE

6.3 Weight function generation simulations: *weight_function_simul*

The preliminary simulations have provided us with appropriate values for the maximum particle and volume move sizes, an estimate for the equilibration time, and the range of order parameters which encompass both phases at equilibrium. With this information we can now proceed to lattice-switch Monte Carlo simulations. The first such simulation is to generate the weight function. The directory *weight_function_generation* corresponds to this simulation. We use the mobile barrier method. 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):

```
$ nohup mpiexec -n 4 ../../../monteswitch_mpi -new -explicit < /dev/null &
```

This command is, of course, specific to our system; one would have to ascertain the analogous command for other systems. Note, however, that we have used the `-explicit` argument to `monteswitch_mpi`, which is necessary for storing the *data* and *state* files pertaining to each MPI thread.

6.3.1 The `params_in` file

The salient features of the *params_in* files for this simulation is as follows:

- **init_lattice** is set to 1, which means 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.
- **M_grid_size**, **M_grid_min** and **M_grid_max** define the macrostates. We set **M_grid_min** and **M_grid_max** to -62 and 32, which corresponds to the range determined in 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 of LSMC simulations (and may not be appropriate for all systems). 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. Often it will be more efficient to use more macrostates (see below).
- **enable_multicanonical** is set to true, 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 mobile barriers to guide the system through order-parameter space. The method would work well with canonical sampling. However, using multicanonical sampling in theory should slightly help the system transition between macrostates with a high free energy difference between them.
- **enable_lattice_moves** is set to true: 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 at least 100 sweeps. Note that since we are using `monteswitch_mpi` there

is one output file for each thread; within 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 true, 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 transition matrix 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. This obviously requires the transition collection matrix to be updated during the simulation. Accordingly we set update_trans to true; 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 true since we wish to use macrostate barriers to force the system to explore order-parameter space in an artificial manner. 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, before the 'next' macrostate is opened to the system. This should be long enough that the system has enough time to equilibrate within the macrostate, but not so long that the system never proceeds to explore 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 100 sweeps, the time to equilibrate within one macrostate is expected to be far shorter. Furthermore, the time spent in each macrostate if macrostate barriers are enabled will 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.

6.3.2 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, *wf.dat* contains the weight function vs. order parameter:

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

Plotting the file *wf.dat* reveals a smooth curve with two minima separated by a high peak near 0. This is the perfect form for a weight function. If the curve were not smooth, or was constant for lare 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. Hence one possible solution is to continue the simulation with `monteswitch_mpi` using the `-resume` flag, e.g.

```
$ nohup mpiexec -n 4 ../../../monteswitch_mpi -resum -explicit < /dev/null &
```

Of course, just because the weight function has the aforementioned 'perfect form', does not guarantee that it is very good. The ideal weight function leads to the whole considered range of order-parameter space being sampled uniformly. A weight function with the 'perfect form' 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 weight function used. However the quality of the weight function affects 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 lower quality weight function is used the system will spend more time in certain regions of order-parameter space than others, and the time between lattice-switches will be less. Hence it is prudent to perform a short simulation using the weight function to verify that it is of good quality.

6.3.3 Double checking the chosen order parameter range

The ideal weight function is related to the canonical macrostate probability distribution via Eqn. [INSERT EQUATION]. Hence one can obtain a probability distribution from the weight function. Of course, this will only be the 'real' probability distribution if the weight function is ideal, which it will not be exactly

after our weight function simulation. Nevertheless we can obtain an approximate probability distribution from the weight function from our simulation. This is useful for double checking that our chosen range of order parameters to consider is appropriate. We use the following command to obtain the probability distribution from the weight function, and output 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
```

Examining *prob.dat*, we see two peaks in the probability distribution in order-parameter space, each corresponding to one phase. 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 our constraint on the order parameter range results in some states which are significantly likely at equilibrium being 'cut out' of the calculation. As can be seen from *prob.dat*, the probability distribution decays to zero within the chosen 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.

6.3.4 Obtaining an estimate of ΔF from the weight function

ith the macrostate probabilty distribution one can in fact obtain an estimate of the free energy difference between the phases. Recall that the (extensive?) free energy difference is related to the time spent in each phase via Eqn. [INSERT EQN NUMBER]. The time spent in each phase can be deduced by integrating the aforementioned peaks in the macrostate probability distribution. It turns out that the peak for negative order parameters corresponds to phase 1, and the peak in the positive region corresponds to phase 2 [JUSTIFY]. Hence the time spent in phase 1(2) is proportional to the area under the peak in the negative(positive) region. With this in mind the following command applies Eqn. [INSERT EQN. NUMBER] to obtain the free energy difference from the weight function:

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

Recall that 9.403 is the β we are considering. The analogous intensive value is obtained by dividing by the number of atoms in the system (384):

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

In the above commands we have assumed that macrostates with order parameters less than 0 'belong' to phase 1, and those with order parameters greater than zero belong to phase 2. 0 seems a sensible choice of 'cut-off' between the two peaks; note that the exact choice of cut-off doesn't really matter since the probability distribution is essentially 0 near 0 order parameter. [NOTE THAT THE ESTIMATE IS NOT ACCURATE; THERE IS NO UNCERTAINTY ASSOCIATED WITH IT HOWEVER, ONE COULD USE MANY WEIGHT FUNCTION SIMULATIONS TO 'SWEEP' THE TEMPERATURE RANGE FIRST...]

6.3.5 Coonfiguring *params_in* for other weight function generation methods

In the above example we used mobile macrostate barriers in generating the weight function. This gives a reasonable weight function very quickly. However, monteswitch supports other methods for generating weight functions. [THE BARRIER METHOD SCALES WELL WITH SYSTEM SIZE. HOWEVER IT IS NOT PERFECT. IT MIGHT BE FIDDLY? IF IT DOESN'T WORK BECAUSE THE BIN WIDTH IS TOO SMALL, THE LOCK IN TIME IS TOO LONG/SHORT - IT CAN BE FIDDLY. THE SHOOTING METHOD WITHOUT BARRIERS IS APPEALING BECAUSE IT ISN'T AS FIDDLY - JUST KEEP RUNNING IT UNTIL YOU GET A GOOD WEIGHT FUNCTION. THE WORST METHOD IS THE VISITED STATES METHOD; HOWEVER, IT CAN BE USED TO REFINE A WEIGHT FUNCTION. BELOW WE DESCRIBE HOW TO CONFIGURE THE params_in FILE TO PERFORM VARIOUS METHODS.]

HAVE A TABLE WITH THE VARIOUS FLAGS

barriers without barriers visited states

enable_multicanonical update_eta update_eta_sweeps update_trans update_eta_method enable_barriers barrier_dynamics lock_moves

TROUBLESHOOTING

If when using the barrier method the system takes a very long time to move between macrostates - far longer than the lock time, then increase the number of bins. This reduces the bin size.

6.4 Weight function verification simulations: *weight_function_verifi*

Assuming that we have a good weight function, one thing remains to know before we perform our production simulation: the correlation time for the multicanonical simulation, using this weight function (and the chosen maximum particle and volume move step sizes [PERHAPS USING DIFFERENT SIZES WOULD BE MORE APPROPRIATE?]). Recall that we obtained an estimate for the equilibration time earlier; however, this pertained only to a one-phase simulation, in a canonical ensemble. As we will see in a moment, the equilibration times in a two-phase lattice switch multicanonical simulation are far longer. We need to know this equilibration time because it determines how large the blocks should be in block analysis for determining equilibrium quantities - in particular for the free energy difference. To determine this we perform a short simulation, using the weight function. This simulation also acts to 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 as expected.

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*, with all counter

variables reset to zero, by invoking the `-reset` argument. With this in mind, we use the *state* file from our previous 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

```
$ ../../../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.

6.4.1 Creating the input `state` file

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

```
$ diff state_in ../weight_function_generation/state
```

The key changes are as follows:

- We have changed **stop_sweeps** to be 10000 - which corresponds to a short simulation
- We have set **update_eta** to false, so that the weight function is fixed throughout the simulation.
- We have set **enable_barriers** to false, 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 keep track of the progress of the simulation.

6.4.2 Results

After the simulation is complete, as before, we can use *data* to generate a plot of the order parameter vs. time in the simulation using the following command:

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

From this we see that the entire range of order-parameter space (-62 to 32) was explored within the simulation. Note that, having realised that the 10000 sweep simulation was too perhaps short, we ran the simulation for another 10000 sweeps by using the command

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

Hence the output files *state* and *data*, and *M_vs_t.dat* created by the above command, correspond to after 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*.

Plotting *M_vs_t.dat* reveals that the simulation explores the entire range of order-parameter space (-62 to 32) within 20000 sweeps. Note that this in practice means that both phases are explored within 20000 sweeps, since high order parameters correspond to phase 2 and low order parameters correspond to phase 1 (see above). Hence the correlation time - the time taken for both phases to be explored - is 10000 sweeps. The block size we use in the forthcoming production simulation must therefore be 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 can use the 'observations' in the *data* files output by the production simulation to analyse the data - once the production simulation is complete - for a variety of block sizes, though software to do this is not currently included in monteswitch.

It is instructive to plot of how often each macrostate was visited during the weight function verification simulation. `monteswitch_post` provides an easy way of extracting this information; the following command creates a file *counts.dat* which is a plot of macrostate index vs. the number of times the macrostate was visited during the entire simulation:

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

From this file we see that all macrostates are visited a reasonable number of times during the simulation. While it is far from the case that the histogram is flat - which an ideal weight function would yield - it is sufficiently flat for our purposes: for our purposes it is acceptable that all macrostates are visited uniformly to within, say, half an order of magnitude. [NOTE THAT WE WOULDN'T NECESSARILY EXPECT A FLAT HISTOGRAM EVEN WITH AN IDEAL WEIGHT FUNCTION FOR SUCH A SHORT SIMULATION - IT IS OF THE ORDER OF THE CORRELATION TIME].

6.5 Production simulations: *production_simulation*

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

```
$ nohup mpiexec -n 4 ../../../monteswitch_mpi -reset -explicit </dev/null &
```

Note that again we have used the `-reset` flag to reset the counter variables at initialisation, and that we have also used the `-explicit` flag to keep the data from all threads.

6.5.1 Creating the input `state` file

The modifications to the state file can be seen 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:

- 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 for the multicanonical simulation using our weight function.
- We have set **update_eta** to false, so that the weight function is fixed throughout the simulation.
- We have set **enable_barriers** to false, since we want 'natural' dynamics for the production simulation.
- **calc_equil_properties** is set to true. This is required for the simulation to calculate equilibrium quantities (the enthalpy, volume and mean-squared displacements of each particle) and their uncertainties using block analysis.
- **block_sweeps**, the number of sweeps which constitute a block in the block analysis is set to 155000, which is far larger than the correlation time of 10000 sweeps determined earlier. Note that, since each MPI thread performs 175000 sweeps, and since each thread will be given 20000 sweeps to equilibrate - during which time the microstates visited by the system are not used in evaluating equilibrium quantities - setting the block size to 155000 sweeps corresponds to each MPI thread performing exactly 1 block. Hence the simulation in total considers 4 blocks.

6.5.2 Results

[HOW MEANINGFUL IS THE INTENSIVE FREE ENERGY DIFFERENCE? UNCERTAINTY IN THE TRANSITION TEMPERATURE IS MORE IMPORTANT]

At the completion of the simulation the *state* file contains the equilibrium quantities, evaluated using block analysis. It is these quantities we are interested in, primarily the (Gibbs) free energy difference between the phases. The variables **equil_DeltaF** and **sigma_equil_DeltaF** in *state* contain the free energy difference and its uncertainty evaluated using block analysis. Hence the following command extracts the free energy difference and its uncertainty 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}'
```

The final value is therefore 0.00004(4)eV, where the units are derived from the convention used in the 'setfl' format used in the *interactions_in* file – the units used in monteswitch are derived entirely from the *interactions.f95* it is used in conjunction with, in this case *interactions_EAM.f95*. The convention used in *monteswitch* is that the free energy difference is the free energy of phase 1 minus that of phase 2. Hence a positive free energy difference, which is the case here, suggests that phase 2 (hcp here as opposed to bcc) is stable at the considered temperature and pressure. However, the uncertainty in our value is of the same magnitude as the value; hence the prefered phase cannot be discerned. This is a moot point though: our free energy difference of 0.04(4)meV can be considered zero for all intents and purposes; this temperature and pressure can be considered to be a coexistence point. Thus our results agree with Ref. [INSERT REFERENCE]

Using similar commands to the above, one can extract the volume and enthalpy per atom for each phase.

```
$ 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;

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

gives the enthalpy per atom for phase 2. One can also obtain the mean-squared displacements for all atoms in phase 1: the following command extracts these quantities from *state* and outputs the mean-squared displacement vs. atom number:

```
$ grep -E ' equil_umsd_1' state | awk '{for(i=2; i<=NF; i++){print i-1,$i}}'
```

Their uncertainties are similarly given by

```
$ grep -E ' sigma_equil_umsd_1' state | awk '{for(i=2; i<=NF; i++){print i-1,$i}}'.
```

Similar commands can be used to obtain the mean-squared displacements and associated uncertainties for phase 2. Of course, it is not necessary to use a lattice-switch simulation to extract all of these 'one-phase' quantities; a conventional Monte Carlo simulation will do the job more efficiently. Here the quantities are somewhat of a by-product of the lattice-switch simulation to calculate the free energy difference.

6.5.3 Consistency checks

Once the simulation is complete, it is prudent to perform some checks to ensure that it has gone as expected, and hence that the values for equilibrium quantities obtained from the simulation are meaningful. Of course, comparing the final values obtained from the simulation to 'known' values is useful. Below we describe some less obvious checks.

The free energy difference for a block is given by Eqn. [INSERT EQUATION], where [t_1] and [t_2] are the number of moves in which the system spent in phase 1 and phase 2 respectively during the block. Of course, this equation presupposes that both phases are explored during the block. In evaluating the final free energy difference, monteswitch only considers blocks during which both phases are explored. This notwithstanding, it is good to 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. Of course, this was the point of the weight function verifictaion simulation to determine the appropriate block size. However, to check this we plot the order parameter vs. simulation time for MPI thread 0:

```
$ grep 'M:' data_0 | awk '{print $2,$3}' > M_vs_t_0.dat
```

Plotting *M_vs_t_0.dat* it can be seen that the system traverses the entire range of order-parameter space (-62–32) often within the block size (175000 sweeps), as required.

Chapter 7

User-defined interatomic potentials

As mentioned in Chapter [INTRO?], the file `interactions.f95`... [DESCRIBE CUSTOMISATION PROCEDURE; MENTION THAT THE DEFAULT IS FOR LENNARD-JONES INTERACTIONS].

Bibliography

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.