Chapter 9

# Algorithms for single MCMC runs

Chapters 5, 7, and 8 describe the swap-only, swap-and-reverse, band-update, and worm algorithms. The content there focuses on algorithm steps and algorithm correctness, without reference to a specific programming language or a specific software implementation. This chapter, by contrast, focuses on particular software-design choices which were made by the author.

The software program (`mcrcm`) which does a single MCMC run is written in the C language. Execution of multiple MCMC runs, including parallel processing, is done using a scripting language such as Bash or Python; this is described in the next chapter. Throughout this chapter and the next, names of data structures and subroutines from the program code are written in `typewriter font`.

The methods here are applicable for any of the algorithms of chapters 5 and 7 — swap-only, swap-and-reverse, and worm — or any other to-be-invented algorithm which satisfies the hypotheses listed in section 4.5 on recipes for MCMC algorithms.

## 9.1 Data structures

There is one main data structure, of C type `points_t`, containing *points* and a *cycle list* (see figure 9.1):

- `dims` contains the lattice dimensions $L$, $L$, $L$. (For 1-dimensional or 2-dimensional use, not described in this thesis, these would be $L, 1, 1$ or $L, L, 1$, respectively.)
- `N` is the product of the three dimensions. It is used so frequently in the program code that it is worth computing this product once at the start of the program and keeping it here.
- `lattice`: $L \times L \times L$ array of points (data type `point_t`).
- `wormhole`: an $(N+1)$st point (data type `point_t`) called the wormhole point (chapter 7). (For the swap-only and swap-and-reverse algorithms, all permutations send the wormhole point to itself.)
- `pcycinfo_list_head` and `pcycinfo_list_tail` are the locations of the first and last cycle-information structure in the doubly linked list of cycle-information structures.

Each point (`point_t` data type) `x` contains:

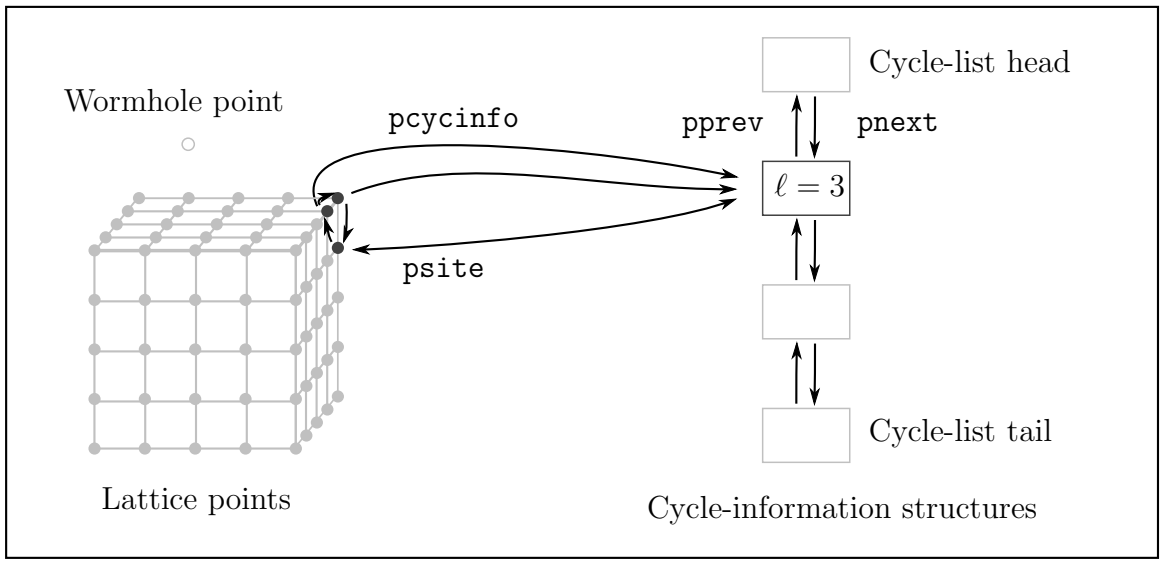


FIGURE 9.1. Lattice sites and the wormhole point are shown in grey; three lattice sites participating in a 3-cycle are shown in black. Every point (including the grey ones for which these arrows are not shown) contains the location (`pcycinfo`) of a cycle-information structure which caches the length of the cycle (`cyclen`); each cycle-information structure contains the location (`psite`) of one point in the cycle. Cycle-information structures are stored in a doubly linked list (`pprev` and `pnext`).

- `selfi`, `selfj`, `selfk`: lattice coordinates of each point $\mathbf{x}$, with $x_i, x_j, x_k$ from 0 to $L-1$. For the wormhole point, these coordinates are set to an undefined value.
- `pfwd`: location of permutation image $\pi(\mathbf{x})$.
- `bfwd`: location of permutation preimage $\pi^{-1}(\mathbf{x})$. This is used for the reason described in section 5.1. Namely, in a swap, one
  - selects a site $\mathbf{x}$,
  - follows the forward permutation pointer to find $\pi(\mathbf{x})$,
  - uses the lattice structure to find a site $\pi(\mathbf{y})$ which is a nearest neighbor of $\pi(\mathbf{x})$, and finally
  - follows the backward permutation pointer to find $\mathbf{y} = \pi^{-1}(\pi(\mathbf{y}))$.
- `fwd_d` and `fwd_dsq`: distance $\|\pi(\mathbf{x}) - \mathbf{x}\|_\Lambda$ and squared distance $\|\pi(\mathbf{x}) - \mathbf{x}\|_\Lambda^2$. These are cached for performance reasons. It is found empirically that approximately 10-20% of proposed Metropolis changes are accepted. Thus, without

caching of these distances, most of the time the forward-distance values would be computed redundantly.

- `pcycinfo`: location of cycle-information structure (see below).
- `mark`: this is a single integer stored at each lattice site. When the cycle-information list is set up (section 9.3) or sanity-checked (section 9.16), it is necessary to sweep through all lattice sites, following permutation cycles, remembering which sites have already been visited. One could allocate a list of marks (one for each lattice site) at the beginning of these routines, and free that list at the end. Instead, the marks are stored in the lattice structure so that this scratch space is available when needed.

Cycle information is stored in a doubly linked list of cycle-information structures (data type `cycinfo_t`). Each contains:

- `psite`: Location of one site in the cycle.
- `cyclen`: Length of the cycle.
- `pprev` and `pnext`: Location of the previous and next cycle-information structure in the doubly linked list.

## 9.2 Overview

A true outline of the program (routine `main` in file `mcrcm.c`) embodies the outline given in section 4.5. Namely:

- Determine the input parameters $L$, $T$, interaction type (non-interacting, $r_2$, $r_\ell$) and parameter $\alpha$, algorithm type (swap-only, swap-and-reverse, worm), and `number_of_sweeps`. (These are passed into the program via the command line.)
- Print all control parameters (see section 9.18).
- Initialize (see section 9.3).
- Thermalization phase:

  Loop until thermalized:
  Do one swap-only, swap-and-reverse, or worm sweep (section 9.4).
  See if thermalization is complete (section 9.6).
  Optionally sanity-check $H$ and cached cycle information (section 9.16).
  Optionally display random-variable instances (section 9.18).

- Accumulation phase:

  For sweep number from 0 to `number_of_sweeps`−1:
  Do one swap-only, swap-and-reverse, or worm sweep (section 9.4).
  Optionally write $\pi$ to disk (section 9.17).
  Optionally sanity-check $H$ and cached cycle information (section 9.16).
  Optionally print realizations of user-specified random variables (section 9.18).
  Remember random-variable instances, for statistical use.
  Compute statistics of random variables (theorem 4.2.9 and section 4.3).
  Display statistics of random variables (section 9.18).

## 9.3 Initialization

Software initialization consists of four main steps:

- Allocate memory for the lattice points: subroutine `get_cubic_lattice_points`. Set the initial permutation to one of the following:
  - The identity permutation. (This is the default, set up by the subroutine `get_cubic_lattice_points`.)
  - A uniform-random permutation on $\mathcal{S}_N$, with the wormhole point sent to itself: subroutine `set_unif_rand_pmt`.
- Allocate time-series arrays for each random variable: `allocate_rvs`.
- Initialize the cycle-information list: subroutine `set_up_cycinfo_list`.
- Find the initial system energy $H$, separated into $D$ and $V$ terms: subroutine `get_H_of_pi`. This is a straightforward implementation of equation (2.1.3).

## 9.4 Metropolis sweeps

The swap-only algorithm uses a swap-only sweep; the swap-and-reverse algorithm uses a swap-only sweep followed by a cycle-reverse sweep. A swap-only sweep (subroutine `SO_sweep`) is as follows:

- Loop through lattice sites $\mathbf{x} = (x, y, z)$ lexically, i.e. $x$ from 0 to $L-1$, $y$ from 0 to $L-1$, $z$ from 0 to $L-1$.
- For the site $\mathbf{x}$, follow the forward permutation pointer to find $\pi(\mathbf{x})$.
- Use the lattice structure to select $\pi(\mathbf{y})$ which is one of the six (i.e. $2d$) nearest-neighbor sites to $\pi(\mathbf{x})$.
- Follow the backward permutation pointer to find the point $\mathbf{y}$.

- In subroutine `try_SO_swap`, propose and perhaps accept a modification of the permutation. This is a Metropolis step, described in section 9.5.

(A slight modification of this algorithm would choose $\mathbf{x}$ at uniform-random location on the lattice, $N$ times, rather than looping sequentially through all $N$ lattice sites.) After a swap-only sweep has been performed, the swap-and-reverse algorithm then does a reverse sweep (subroutine `reverse_sweep`):

- For each cycle in the permutation (i.e. one follows the doubly-linked list of cycle-information structures), with probability 1/2 reverse all the arrows in that cycle. This is done for the reason described in section 5.4, namely, it permits non-zero (but only even) winding numbers.

- At each point, the forward and backward permutation pointers must be updated, and the cached forward distance and forward squared distance must be copied from one point to another. The cycle-information structure is not affected. Also, the system energy is not affected.

A worm sweep (subroutine `worm_sweep`) is done as in section 7.4:

- One attempts to open the permutation at a uniform-random lattice site $\mathbf{x}$.

- If the open is not accepted, the sweep is complete.

- Otherwise, some number of head swaps and/or tail swaps are proposed and perhaps accepted. Eventually, a close is proposed and accepted. The sweep is then complete.

Notice that the swap-only and reverse sweeps are of deterministic length: the former processes all $N$ lattice sites; the latter processes all cycles. (Of course, it takes more CPU time for an accepted proposal then a rejected proposal. Nonetheless, a fixed number of proposals is always made.) For the worm sweep, though, the stopping time is random, depending on the time for the worm head and tail to approach one another on the lattice. (See section 7.8 for more details.)

## 9.5 Metropolis steps

See chapter 8, and in particular section 8.3 ($\Delta r_\ell$), for background information on $\Delta H$ and $\Delta r_\ell$. A *Metropolis proposal*, for a swap as described in section 5.1, is as follows:

- To find $\Delta D$ as described in section 8.1: compute $\|\mathbf{x}-\pi(\mathbf{y})\|_\Lambda^2$ and $\|\mathbf{y}-\pi(\mathbf{x})\|_\Lambda^2$. These will be $D$ terms for the proposed new permutation $\pi'$, if it is accepted. The corresponding $D$ terms for the current permutation $\pi$, namely, $\|\mathbf{x}-\pi(\mathbf{x})\|_\Lambda^2$ and $\|\mathbf{y}-\pi(\mathbf{y})\|_\Lambda^2$, are already cached at the points $\mathbf{x}$ and $\mathbf{y}$. Then

$$\Delta D = \frac{T}{4}\left(\|\mathbf{x}-\pi(\mathbf{y})\|_\Lambda|^2 + \|\mathbf{y}-\pi(\mathbf{x})\|_\Lambda|^2 - \|\mathbf{x}-\pi(\mathbf{x})\|_\Lambda|^2 - \|\mathbf{y}-\pi(\mathbf{y})\|_\Lambda|^2\right).$$

- The subroutine `get_Delta_V_SO` performs the $\Delta V$ computations described in chapter 8. It also computes $\mathbf{x} \circ\!\!-\!\!\circ \mathbf{y}$ (namely, whether the points $\mathbf{x}$ and $\mathbf{y}$ are in the same cycle or not), along with $\ell_{\mathbf{x},\mathbf{y}}(\pi)$ and $\ell_{\mathbf{y},\mathbf{x}}(\pi)$. (If $\mathbf{x}$ and $\mathbf{y}$ are in the same cycle, then the cycle lengths are equal; if they are in different cycles, $\ell_{\mathbf{x},\mathbf{y}}(\pi)$ and $\ell_{\mathbf{y},\mathbf{x}}(\pi)$ are undefined.)

- The total change in system energy for the proposed change is $\Delta H = \Delta D + \Delta V$.

The Metropolis proposal is accepted with probability $\min\{1, \exp(-\Delta H)\}$. That is, if a pseudorandom number (section 9.19) uniformly distributed between 0 and 1 is less than $\exp(-\Delta H)$, then $\pi$ is replaced by $\pi'$.

A *Metropolis update* consists of the following:

- The new system energy $H'$ is set to $H + \Delta H$.

- The forward and backward permutation pointers `pfwd` and `pbwd` at points $\mathbf{x}$ and $\mathbf{y}$ are updated so that $\pi'(\mathbf{x}) = \pi(\mathbf{y})$ and $\pi'(\mathbf{x}) = \pi(\mathbf{y})$.

- The cached forward squared distances $\|\mathbf{x} - \pi'(\mathbf{x})\|_\Lambda|^2$ and $\|\mathbf{y} - \pi'(\mathbf{y})\|_\Lambda|^2$ are stored in the `fwd_dsq` slots of the `point_t` data structures for points $\mathbf{x}$ and $\mathbf{y}$. Respective square roots are stored in the `fwd_d` slots.

- Cycle-information structures are updated by the subroutine `update_cycinfo`, which is discussed next.

**Without cycle-length caching:** Recall from section 8.3 that computing $\Delta r_\ell$ requires the following steps:

- See if $\mathbf{x}$ and $\mathbf{y}$ are in the same cycle.

- If so, find $\ell_{\mathbf{x},\mathbf{y}}(\pi)$ and $\ell_{\mathbf{x},\mathbf{y}}(\pi)$; if not, find $\ell_{\mathbf{x}}(\pi)$ and $\ell_{\mathbf{y}}(\pi)$.

To find these values, one may start at site $\mathbf{x}$, moving forward one permutation jump at a time. If one reaches $\mathbf{y}$ before returning to $\mathbf{x}$, then $\mathbf{x}$ and $\mathbf{y}$ are in the same cycle, and $\ell_{\mathbf{x},\mathbf{y}}(\pi)$ has already been found. Continuing to count hops back to $\mathbf{x}$ yields $\ell_{\mathbf{y},\mathbf{x}}(\pi)$. If, on the other hand, one returns to $\mathbf{x}$ without having encountered $\mathbf{y}$, then $\mathbf{x}$ and $\mathbf{y}$ are in different cycles, and $\ell_{\mathbf{x}}(\pi)$ has already been computed. Counting hops from $\mathbf{y}$ back to itself yields $\ell_{\mathbf{y}}(\pi)$.

Notice that each of these cycle-following steps requires $O(\ell)$ machine operations where $\ell$ is the mean cycle length. For subcritical temperatures $T$ where cycles become long (of length at most $N$), these cycle-following steps become unacceptably time-consuming. Caching of cycle lengths has been found to reduce simulation time, for $L = 40$ lattices, by a factor of 15. The improvement is even more pronounced as $L$ is increased.

**With cycle-length caching:** Keeping a list of cycles means that the following Metropolis-proposal steps take $O(1)$ machine operations:

- To see if $\mathbf{x}$ and $\mathbf{y}$ are in the same cycle, check the two points' `pcycinfo` slots and see whether they are equal or not.

- The cycle lengths $\ell_{\mathbf{x}}(\pi)$ and $\ell_{\mathbf{y}}(\pi)$ are immediately found by consulting the `cyclen` slots of the `pcycinfo` data structures.

Now, if $\mathbf{x}$ and $\mathbf{y}$ are in the same cycle, one must additionally find either $\ell_{\mathbf{x},\mathbf{y}}(\pi)$ or $\ell_{\mathbf{y},\mathbf{x}}(\pi)$. (Note that $\ell_{\mathbf{x},\mathbf{y}}(\pi) + \ell_{\mathbf{y},\mathbf{x}}(\pi) = \ell_{\mathbf{x}}(\pi) = \ell_{\mathbf{y}}(\pi)$ so it suffices to find either one or the other.) This is, *a priori*, is of complexity $O(\ell)$. However, it has been found empirically that in the MCMC simulations described by this thesis, one of the two of $\ell_{\mathbf{x},\mathbf{y}}(\pi)$ or $\ell_{\mathbf{y},\mathbf{x}}(\pi)$ is almost always small. That is, a split of a large cycle usually pinches off a small cycle; rarely is a large cycle split evenly. It is likewise found empirically that on merges of disjoint cycles, usually one of the two cycles is small. Thus, if $\mathbf{x}$ and $\mathbf{y}$ are in the same cycle, it suffices to start at $\mathbf{x}$, searching forward and backward one jump at a time. If one encounters $\mathbf{y}$ on forward jumps, $\ell_{\mathbf{x},\mathbf{y}}(\pi)$ has been found; if one encounters $\mathbf{x}$ on backward jumps, $\ell_{\mathbf{y},\mathbf{x}}(\pi)$ has been found. Thus the complexity is of order

$$O(\min\{\ell_{\mathbf{x}}, \ell_{\mathbf{y}}\}).$$

A result of this is that all other computations done in our MCMC simulations are $O(N)$. This bit, however, is necessarily $O(N^2)$. Yet, it is $O(N^2)$ with a low constant of proportionality, since one of $\ell_{\mathbf{x}}$ and $\ell_{\mathbf{y}}$ is almost always small. Plots of CPU time as a function of $N$ are shown in section 9.21.

It has just been demonstrated that keeping cached cycle lengths makes $\Delta r_\ell$ computations for a Metropolis proposal quicker. Of course, one pays for this by needing to maintain cached cycle lengths after Metropolis updates. The following steps are performed in the subroutine `update_cycinfo`.

If the swap has split a cycle into two (figure 9.2):

- To minimize the number of computations, as described above for Metropolis proposals, find the shorter new cycle. Suppose $\mathbf{x}$'s new cycle is longer than $\mathbf{y}$'s. (If not, swap local variables in the subroutine to make this so.)

- All points in $\mathbf{y}$'s new cycle must now point to a new cycle-information structure. The cycle has been split, so follow from the new $\pi'(\mathbf{y})$ (which was $\pi(\mathbf{x})$ before the merge) around to and including $\mathbf{y}$. As above, the number of sites that must be visited is $\min\{\ell_{\mathbf{x}}(\pi'), \ell_{\mathbf{y}}(\pi')\}$.

- The cycle-information structures for both split cycles need to have their cycle lengths updated: $\ell_{\mathbf{x}}(\pi') = \ell_{\mathbf{y},\mathbf{x}}(\pi)$ and $\ell_{\mathbf{y}}(\pi') = \ell_{\mathbf{x},\mathbf{y}}(\pi)$.

- The cycle-information structures for the two cycles need to have their site pointers point to $\mathbf{x}$ and $\mathbf{y}$, respectively.

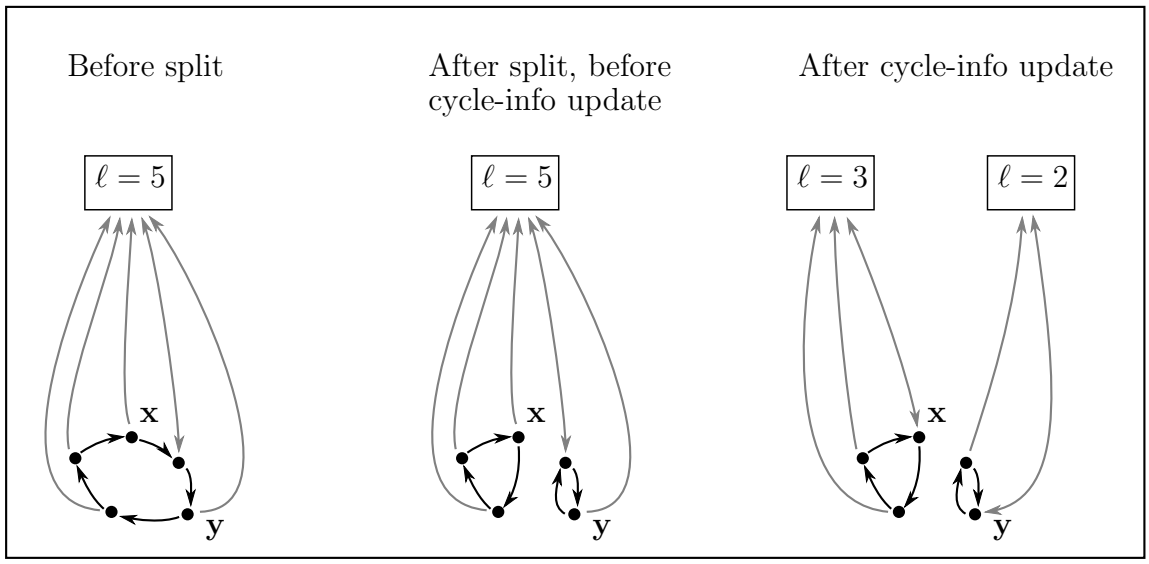


FIGURE 9.2. Update of permutation and cycle information on a split swap. Grey arrows represent pointers between sites and their cycle-information structures; black arrows represent forward permutation pointers.

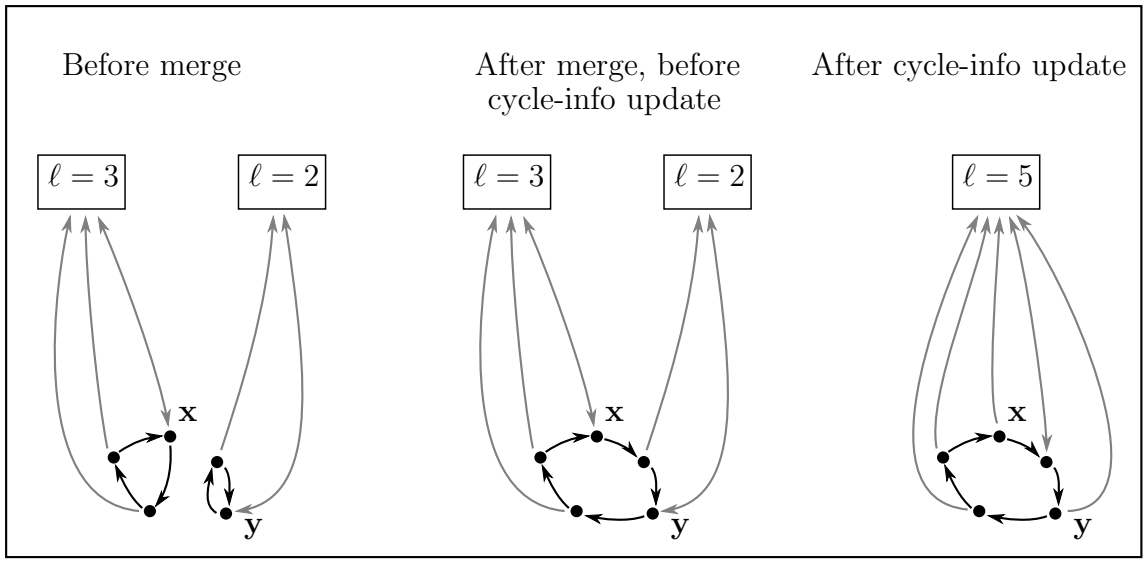


FIGURE 9.3. Update of permutation and cycle information on a merge swap. Grey arrows represent pointers between sites and their cycle-information structures; black arrows represent forward permutation pointers.

- The new cycle-information structure for the $\mathbf{y}$ cycle must be added to the doubly linked list of cycle-information structures.

If the swap has merged two cycles into one (figure 9.3):

- To minimize the number of computations, find the shorter old cycle. Suppose $\mathbf{x}$'s old cycle is longer than $\mathbf{y}$'s. (If not, swap local variables in the subroutine to make this so.)
- The cycle-information structure for the merged cycle needs to have its cycle length updated: $\ell_{\mathbf{x}}(\pi') = \ell_{\mathbf{x}}(\pi) + \ell_{\mathbf{y}}(\pi)$.
- All points in $\mathbf{y}$'s old cycle must have their cycle-information structures now point to $\mathbf{x}$'s cycle-information structure. The two cycles have been merged, though, so follow from the new $\pi'(\mathbf{x})$ (which was $\pi(\mathbf{y})$ before the merge) around to and including $\mathbf{y}$. As above, the number of sites that must be visited is $\min\{\ell_{\mathbf{x}}(\pi), \ell_{\mathbf{y}}(\pi)\}$.
- The cycle-information structure for the old $\mathbf{y}$ cycle must be removed from the cycle-information list and freed.

## 9.6 Thermalization detection

As noted in section 9.2, the initial permutation selected in an MCMC sequence is atypical: the identity permutation has zero energy, lower than the mean energy for the stationary distribution, whereas a uniform-random permutation almost always has energy higher than the mean due to its long jump lengths. (See equation (2.1.3)and figure 9.4.)

As always in MCMC simulations [Berg, LB], one must run through some number of Metropolis steps until the system has thermalized, i.e. when the stationary distribution has been reached. In the MCMC discipline, practitioners use various techniques. The thermalization-detection algorithm chosen for the work described by this dissertation counts turning points of smoothed system energy. This algorithm may be justified using conditional expectation of $\Delta H$, as well as visually.

Recall from chapters 2 and 5.2 that permutations have energies $H(\pi)$ and $H(\pi')$, probabilities $P_{\text{Gibbs}}(\pi) = e^{-H(\pi)}/Z$ and $P_{\text{Gibbs}}(\pi') = e^{-H(\pi')}/Z$, and Metropolis transition probabilities

$$A(\pi, \pi') = C(1 \wedge e^{-(H(\pi')-H(\pi))}).$$

The premise is that a permutation $\pi$ is taken from the stationary distribution if the subsequent permutation $\pi'$ is equally likely to have higher or lower energy. Stated probabilistically, we say that the expected value of $H(\pi')$, conditioned on transitioning

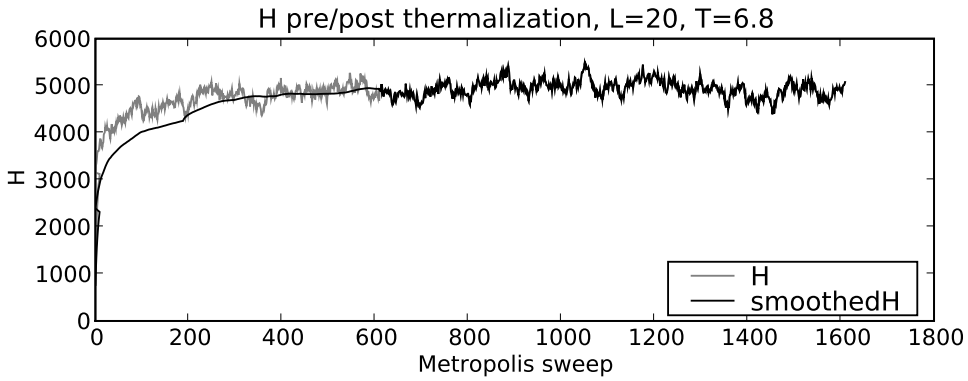


FIGURE 9.4. Plot of system energy versus Metropolis sweep number. The transition from grey to black indicates thermalization was detected, via 40 turning points of energy smoothed over a sliding window of 200 sweeps.

from $\pi$, should be zero. Recall that for a random variable $X$ and an event $A$, with outcomes $x$, we have

$$\mathbb{E}[f(X) \mid X \in A] = \frac{\sum_{x \in A} P(x) f(x)}{\sum_{x \in A} P(x)} = \frac{\sum_{x \in A} P(x) f(x)}{P(A)}.$$

Here, this translates to

$$\mathbb{E}[H(\pi') \mid \pi] = \frac{\sum_{\pi' \circ\!\!-\!\!\circ \pi} P_{\text{Gibbs}}(\pi') H(\pi')}{\sum_{\pi' \circ\!\!-\!\!\circ \pi} P_{\text{Gibbs}}(\pi')} = \frac{\sum_{\pi' \circ\!\!-\!\!\circ \pi} P_{\text{Gibbs}}(\pi) A(\pi, \pi') H(\pi')}{\sum_{\pi' \circ\!\!-\!\!\circ \pi} P_{\text{Gibbs}}(\pi) A(\pi, \pi')}$$

$$= \frac{\sum_{\pi' \circ\!\!-\!\!\circ \pi} P_{\text{Gibbs}}(\pi) A(\pi, \pi') (H(\pi) + \Delta H)}{\sum_{\pi' \circ\!\!-\!\!\circ \pi} P_{\text{Gibbs}}(\pi) A(\pi, \pi')}$$

$$= \frac{\sum_{\pi' \circ\!\!-\!\!\circ \pi} P_{\text{Gibbs}}(\pi) A(\pi, \pi') H(\pi) + \sum_{\pi' \circ\!\!-\!\!\circ \pi} P_{\text{Gibbs}}(\pi) A(\pi, \pi') \Delta H}{\sum_{\pi' \circ\!\!-\!\!\circ \pi} P_{\text{Gibbs}}(\pi) A(\pi, \pi')}$$

$$= H(\pi) + \frac{\sum_{\pi' \circ\!\!-\!\!\circ \pi} P_{\text{Gibbs}}(\pi) A(\pi, \pi') \Delta H}{\sum_{\pi' \circ\!\!-\!\!\circ \pi} P_{\text{Gibbs}}(\pi) A(\pi, \pi')}.$$

(Recall from definition 5.2.4 that the sum over $\pi' \circ\!\!-\!\!\circ \pi$ includes all permutations $\pi'$, including $\pi$ itself, reachable from an accepted or rejected swap.) For pre-thermalization $\pi$, this expectation should be highly positive (when the initial $\pi_1$ is the identity), or highly negative (when the initial $\pi_1$ is uniform-random). For post-thermalization $\pi$, this expectation should be approximately zero. Alternatively, for various post-thermalization $\pi$, this expectation should be equally likely positive or negative. As discussed in section 5.2, given $\pi$, there are $3N + 1$ choices of $\pi'$ (including $\pi$ itself) which could be reached on a swap. Yet, in the MCMC sequence, only one $\pi'$ is chosen. Thus, this conditional expectation is CPU-intensive to compute, and moreover does not take advantage of the MCMC sequence itself.

We estimate the above conditional expectation by first defining

$$H_S(t) = \frac{1}{S} \sum_{i=0}^{S-1} H(\pi_{t-i}),$$

where $S$ is the *smoothing window size* and $t$ denotes a Metropolis sweep counter with $t \geq S$. That is, the system energy at Metropolis sweep $t$ is averaged over the last $S$ sweeps. The system is deemed to be thermalized when $H_S(t)$ has changed sign sufficiently many times, i.e. when $H_S(t)$ has had more than a threshold number of turning points.

As is typically the case in such matters, rigorously determining the spectral gap in the Markov transition matrix is computationally intractable. One relies instead on heuristics which are justified by the practitioner's experience and all available evidence. Visually examining the plot in figure 9.4, one may decide that thermalization has occurred by, say, Metropolis sweep 300. Examining a plot for different $L$, $T$, and interaction strength, one might choose a different Metropolis sweep count. I have examined such plots over a broad range of parameter values; I have chosen $S = 200$ and threshold number of turning points equal to 40, ensuring that the automated thermalization detection (when grey turns to black in figure 9.4) agrees with my visual judgment. Thermalization takes one percent or less of total CPU time, as shown in the caption of figure 9.5.

## 9.7 Computation of random variables

All the random variables described in chapter 3 are computed in `mcrcm`. Details of each are described in the sections following this one. The software architecture of `mcrcm` is such that it is easy to add a newly invented random variable to the code.

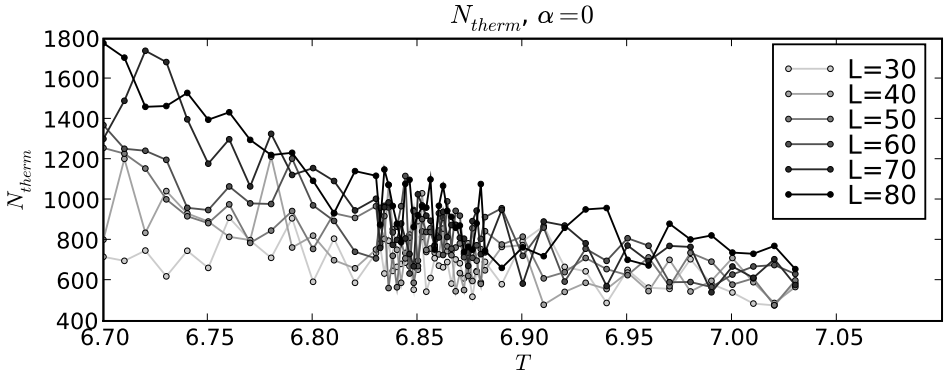


FIGURE 9.5. Thermalization time as a function of $L$ and $T$, for $\alpha = 0$. Plots for other $\alpha$ are similar. Data accumulation takes $10^5$ or $10^6$ sweeps; thermalization is here seen to take on the order of $10^3$ sweeps, i.e. a fraction of a percent of CPU time.

## 9.8 Computation of system energy

The system energy $H$, with non-interacting terms $D$ and interacting terms $V$ (as defined in chapter 2), is tracked; one easily scales to obtain the energy per site (or energy density) $h = H/N$, $d = D/N$, and $v = V/N$.

As described in section 9.3, system energy $H$ is computed for the initial permutation. As described in section 9.4, $\Delta H$ (split out into $\Delta D$ and $\Delta V$) is computed for a proposed new permutation; if the proposal is accepted, $H$ is replaced by $H + \Delta H$. As described in section 9.16, optional automated checks verify that the accumulated $\Delta H$ computations (chapter 8) correctly track the true system energy.

A plot of $H$ as a function of Metropolis sweep, within a single invocation of `mcrcm`, is shown in figure 9.4 on page 95. Dependence of $H$ on $L$, $T$, and $\alpha$ is shown in figure 9.6. Note that the system energy, as centrally important as it is, does not function as an order parameter (section 3.7): it exhibits no sharp transition near the critical temperature.

## 9.9 Computation of $r_\ell(\pi)$

Given the cycle-information list as described in sections 9.1 and 9.5, it is trivial to compute the cycle-length occupation numbers $r_\ell(\pi)$ for $\ell = 1$ to $N$. Namely: Set $r_1, \ldots, r_N = 0$. For each cycle in the cycle list, increment $r_\ell$ by 1 where $\ell$ is the length of the current cycle. Dependence of the sample mean of $r_2$ as a function of $L$, $T$, and $\alpha$ is shown in figure 9.7.

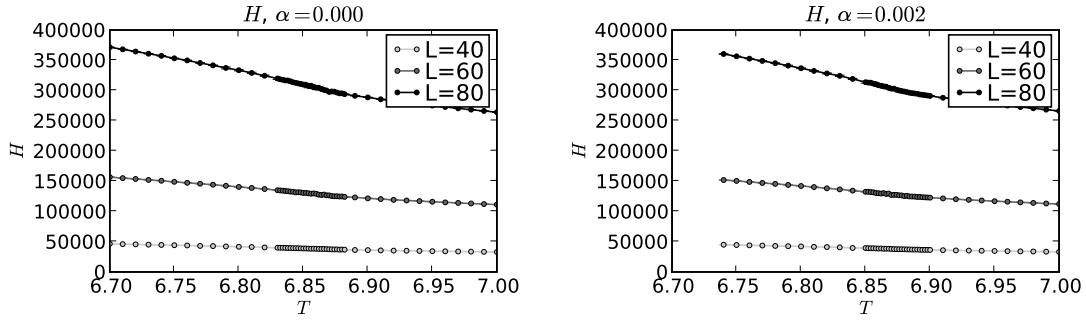


FIGURE 9.6. Behavior of system energy $H$ as function of $L$ and $T$, for $\alpha = 0$, 0.002.

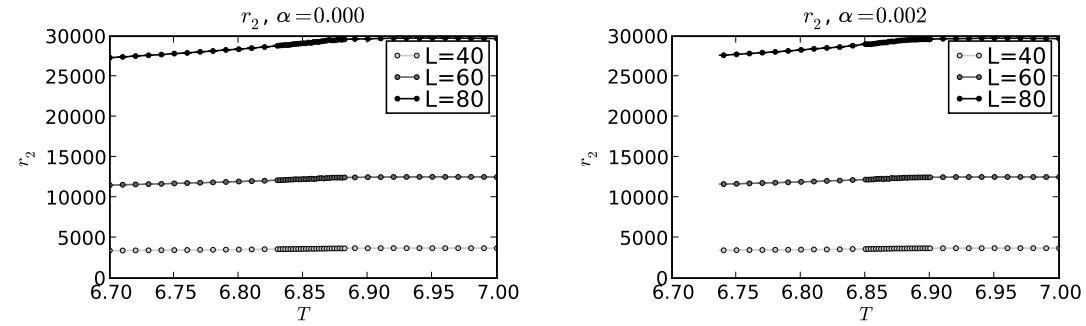


FIGURE 9.7. Behavior of $r_2$ as function of $L$ and $T$, for $\alpha = 0$, 0.002.

## 9.10 Computation of cycle lengths and correlation length

Given the lattice data structure as described in section 9.1, one easily computes the the spatial cycle lengths $s_{\mathbf{x}}(\pi)$ defined in section 3.2. For each site $\mathbf{x}$ in the lattice $\Lambda$, one examines the cached `fwd_d` attribute, which is precisely $s_{\mathbf{x}}(\pi)$. As described in section 3.2, the correlation length $\xi$ is the spatial cycle length averaged over all $N$ lattice points. Plots and analysis of the order parameter $1/\xi$ may be found in chapter 11.

## 9.11 Computation of mean and maximum jump length

Mean jump length is as defined in section 3.3. The maximum jump length is the largest jump length encountered at any of the $N$ lattice sites, for any of the $M$ permutations in the MCMC sequence. This confirms the hypothesis of short jump lengths as mentioned in section 3.6. See figure 9.8. As discussed in section 3.2, the jump length is averaged not only over all permutations $\pi$ generated in the MCMC

sequence, but moreover is averaged over all $N$ lattice points for each $\pi$.

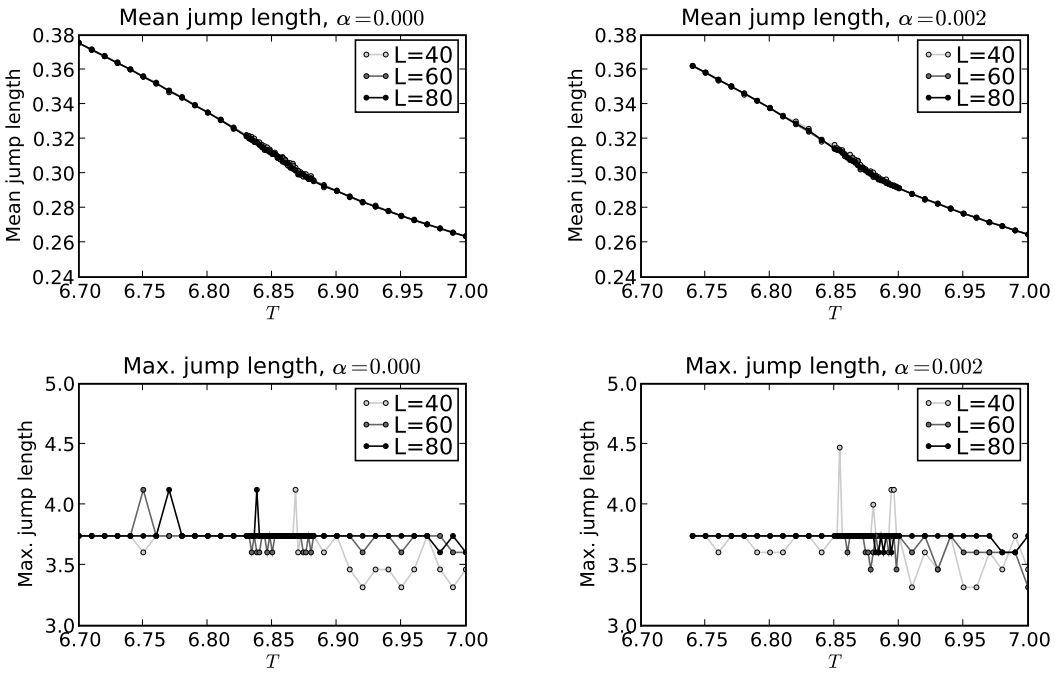


FIGURE 9.8. Mean and maximum jump length as function of $L$ and $T$, for $\alpha = 0$, 0.002.

## 9.12 Computation of $f_I$

As discussed in section 3.4, $f_I$ is computed using the vertical intercept of a tangent-line approximation to $\mathbb{E}[f_{1,k}]$ as a function of $k/N$.

Computation of $\mathbb{E}[f_{1,k}]$ as a function of $k$ is straightforward: we maintain an array indexed from 1 to $N$ of counters, all initially set to zero. For each permutation $\pi$, we count the number of sites participating in cycles of length $k$. This is directly obtained from the cycle-information structure (see section 9.1). Scaling this array by $1/N$ yields a finite-sample approximation to $\mathbb{E}[f_{k,k}]$. Cumulatively summing the array gives an estimator of $\mathbb{E}[f_{1,k}]$. See figure 9.9.

Given that, the tangent line is found, in the presence of statistical variability, as follows: subtract off the diagonal line $(k/N, k/N)$ which runs from the lower left corner to the upper right corner. The peak of the difference (the dashed line in figure 9.9) shows the outermost point of the original $\mathbb{E}[f_{1,k}]$ estimator. Then the tangent line is taken to have slope 1 running through this point. As the number $M$ of permutations increases, this outermost point adheres closely to the visible diagonal; its location is

defined by its peers. Thus, we are not computing $f_I$ based on a single, noisy data point, but rather using much of the available $\mathbb{E}[f_{1,k}]$ data.

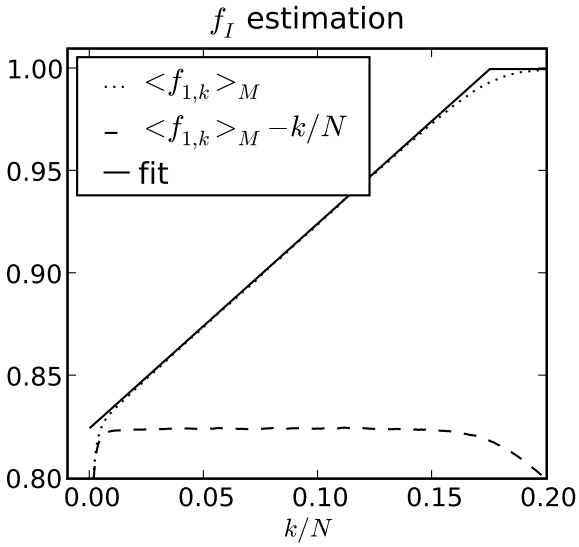


FIGURE 9.9. Estimation of $f_I$. The dotted line shows raw $\langle f_{1,k} \rangle_M$ data obtained by cumulatively summing an array of sample averages of $f_{k,k}(\pi)$. The dashed line is the dotted line minus the diagonal $(k/N, k/N)$. The abscissa of the peak of this difference is the abscissa of the outermost point of the dotted line. The tangent line is drawn through there, with slope 1. Then $f_I$ is one minus the vertical intercept of that tangent line: in this case, $f_I = 1 - 0.825 = 0.175$. The simulation used an MCMC run of $10^4$ permutations on $L = 20$ at $T = 6.5$.

## 9.13 Computation of $\ell_{\max}$, $f_{\max}$, and macroscopic-cycle quotient

The quantities $\ell_{\max}$, $f_{\max}$, $f_I$, and macroscopic-cycle quotient $f_{\max}/f_I$ were defined in sections 3.4 and 3.5. It is easy to compute $\ell_{\max}$: find the longest cycle in the cycle-information list (section 9.1). Computation of $f_I$ is found as described in section 9.12.

Figure 9.10 makes clear the difference between subcritical and supercritical behavior which was alluded to in section 2.3. Below $T_c$, $r_1$, $r_2$, etc. are smaller than above $T_c$, since there is occasionally a long cycle: $\ell_{\max}$ is markedly higher below $T_c$. Plots and analysis of the order parameter $f_{\max}$ may be found in chapter 11. The macroscopic-cycle quotient is analyzed in section 11.8.

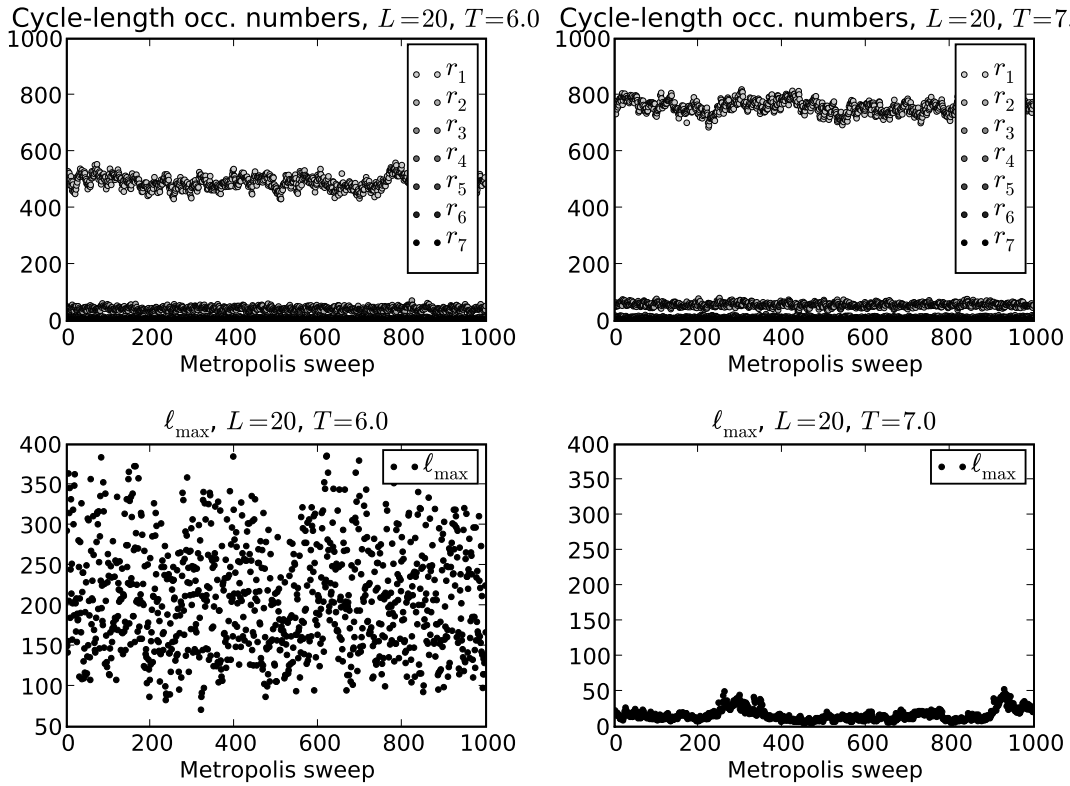


FIGURE 9.10. Per-realization values of $r_\ell$ and $\ell_{\max}$ with $L = 20$, $T = 6.0, 7.0$.

## 9.14 Computation of winding numbers, $f_S$, and $f_W$

The winding-number triple $\mathbf{W} = (W_x, W_y, W_z)$ is as defined in section 3.6. Straightforward application of equations (3.6.2) and (3.6.3), for $\mathbf{W}$ and $\mathbf{W}^2$, respectively, are sufficient as long as the difference vectors $\mathbf{d}_\Lambda(\mathbf{x}, \mathbf{y})$ are obtained. For these (see also section 3.1) one first computes the difference $\mathbf{x} - \mathbf{y}$. Then, for each of the three coordinate slots, one adds or subtracts multiples of $L$ until the result is between $-L/2$ and $L/2$. In figure 9.11 one observes the even parity of winding-number components, as discussed in section 5.4. As well, the temperature dependence in the histograms shows the subcritical transition to winding cycles. Plots and analyses of the order parameters $f_S$ and $f_W$ may be found in chapter 11.

## 9.15 Computation of integrated autocorrelation times

Integrated autocorrelation times are estimated precisely as described in sections B.7 through B.10 of appendix B. Error bars in order-parameter plots in this dissertation are the $\tau_{\text{int}}$-corrected sample standard deviations of the sample means.

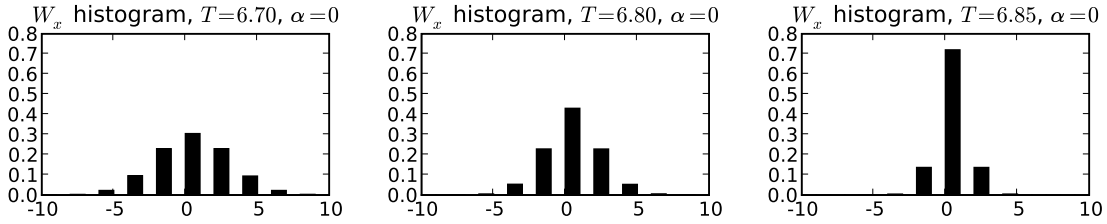


FIGURE 9.11. Histograms of $W_x$ for $\alpha = 0$ and $T = 6.70, 6.80, 6.85$.

Some estimated integrated autocorrelation times are shown in figure3 9.12. The key point is that uncertainty increases in the critical region. For this reason, simulations in the critical region were run with $10^6$ Metropolis sweeps for $L = 30, 40, 50$, and with $10^5$ sweeps otherwise. Similarly, figure 9.13 shows estimates of the correlation factors $\eta$ , in the context of appendix B. Namely, $\eta = 0$ yields an IID sequence; $\eta$ for MCMC simulations done in this dissertation are typically 0.99 and above, indicating highly autocorrelated MCMC sequences.

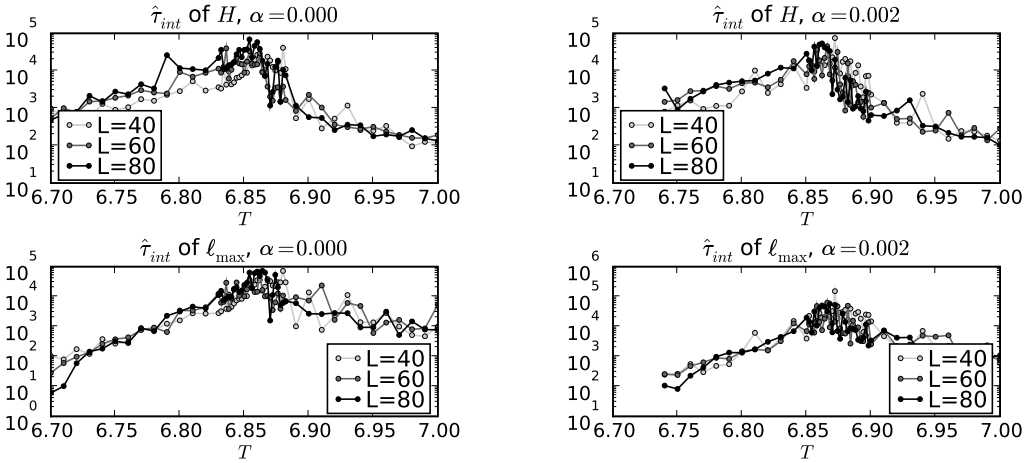


FIGURE 9.12. Estimated integrated autocorrelation times of $H$ and $\ell_{\max}$ as functions of $L$ and $T$, for $\alpha = 0$ and 0.002.

## 9.16 Consistency checks

The following checks are run for every MCMC simulation described in this dissertation:

- Interaction type is one of none, two-cycle interactions, $r_\ell$ (Ewens) interactions. This protects against uninitialized variables.

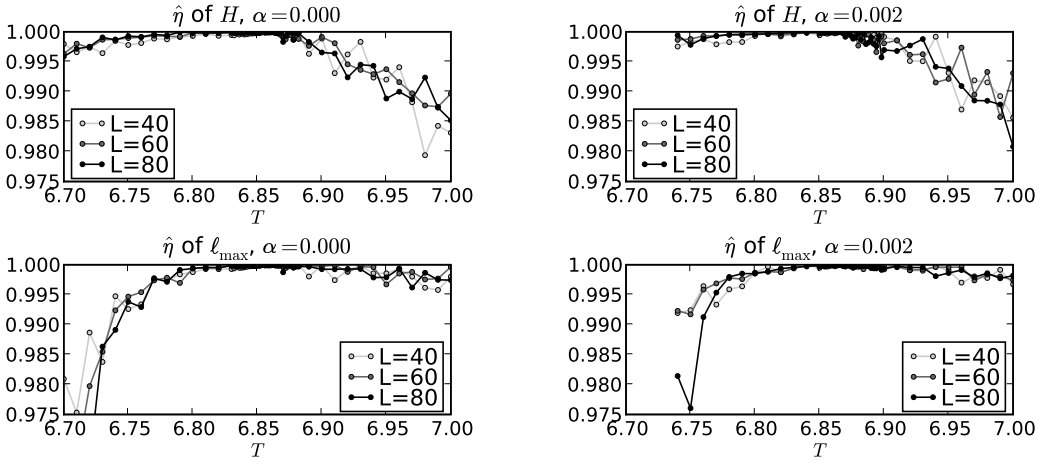


FIGURE 9.13. Estimated autocorrelation factor of $H$ and $\ell_{\max}$ as functions of $L$ and $T$, for $\alpha = 0$ and 0.002.

- Worm move is one of open, close, head swap, or tail swap. Again, this protects against uninitialized variables.
- $L \geq 1$; $d = 1, 2, 3$.
- The cycle-information list is never empty when a cycle-information structure is removed (e.g. on a merge).

The following checks are omitted from production runs due to their CPU-time expense, but were run during code development and testing:

- `check_H`: As described in sections 9.3 and 9.5, the system energy is initially computed by `get_H_of_pi`, then updated by $\Delta H$ as computed in Metropolis proposals/updates. If $H$-checking is enabled (in the header file `checks.h`), then after every Metropolis sweep, the system energy is computed by brute force, and is verified to be within roundoff error of the system energy which has been tracked by $\Delta H$ computations.
- `sanity_check_cycinfo_list`: This is also enabled in the header file `checks.h`. It consists of three checks, which are run after every Metropolis sweep: (1) The cycle list partitions the lattice sites. All sites are marked unvisited; all cycles in the cycle-information list are followed, with visits to each site counted; each site is checked to have been visited no more and no less than once. (2) The cached cycle lengths are correct: These are verified by following permutation pointers around each cycle. (3) For each cycle, all points in the cycle are verified to point to the same cycle-information structure.

The following visual checks were done for selected single runs:

- $H$ plots, such as figure 9.4 of section 9.6 should show system energy increasing from 0 up to equilibrium for initial identity; decreasing down to equilibrium for initial uniform-random.

Visual checks for single runs:

- Output from `mcrcm`, as shown in section 9.18, looks sane; numbers appear to be in their normal ranges.
- $\langle f_{1,k} \rangle_M$ plots, as shown in section 9.12, are as usual.
- $f_W$, $f_{\max}$, and $f_I$ are between 0 and 1.
- Sample mean of $\sum_{\ell=1}^{N} \ell r_\ell$ is equal to $N$. (This is `k*counts sum` in the MCMC output, as described in section 9.18.)
- Sample mean of energy density $h$ is of order 1.
- Mean jump length is of order 1; maximum jump length is approximately 3-4.
- Winding-number histogram peaks at $W_x = 0$, with population out to perhaps $\pm 6$ depending on system temperature.
- Metropolis acceptance rate is 10 to 20 percent.

Visual checks for runs over parameter values (chapter 10):

- A 30-page file containing all plots of the form shown in this chapter — not just a representative selection — is used to compare random-variable output to results from before a software change.

Automated software checking:

- All source code is compiled with `gcc -Wall -Werror`. This enables all warnings (e.g. unused variables, reading variables before they are initialized, missing arguments to `printf`, and so on), and treats warnings as fatal compilation errors.
- The open-source `valgrind` tool finds, at run time, many (but certainly not all) common programming errors. These include reading or writing off the end of arrays, continuing to use dynamically allocated memory after it is freed, dynamically allocated memory which is not freed, and reading of uninitialized variables.
- The open-source `gprof` tool tabulates in which subroutines CPU time is being spent. This helps to identify inefficient programming. See also the `gprof` output in section 9.20.

## 9.17 Saved realizations

The computer program described up to this point is called `mcrcm`, since it does Markov chain Monte Carlo simulations for the random-cycle model. As described in section 9.2, it uses MCMC methods to generate a sequence of permutations, accumulating sums of various random variables along the way, in order to display statistics of those random variables at the end.

A key software-optimization insight was offered by Volker Betz. Namely, the MCMC steps are time-consuming, so one might optionally wish to have `mcrcm` write the sequence of realized permutations to a disk file. Then, another program can simply read those permutations — quicker than they were generated using MCMC methods — and compute random variables. This is particularly advantageous in a shoulder-to-shoulder collaboration environment: when one invents a new random variable to compute, one need not re-realize another sequence of permutations.

The trade-off is that a single realization file can be quite large. Specifically, $M$ permutations are stored. For each permtuation, 3 bytes are stored for each of $N$ lattice sites $\mathbf{x}_i$: these are the $x$, $y$, and $z$ coordinates of $\pi(\mathbf{x}_i)$. The realization file contains a header and a footer of negligible size (a few bytes each). Thus, it totals $3ML^3$ in size — for example, 240 MB for $L = 20$ and $M = 10^4$, and one would likely want several such files for different values of $L$, $T$, and/or $\alpha$. One might, of course, develop more clever storage representations which reduce the necessary file size.

This second program is called `rvrcm`, since it computes random variables for the random-cycle model. By default, `mcrcm` does not store realization files: they occupy several megabytes for each run, totalling several gigabytes for a run through a set of parameters (chapter 10). If desired, however, one invokes `mcrcm` with an extra command-line argument `rzn=`*myfile*`.rzn`. Later, one invokes simply `rvrcm rzn=`*myfile*`.rzn`. The `.rzn` file contains a header with all control parameters which were initially supplied to `mcrcm`. The `rvrcm` program prints the same information as described in section 9.18.

For example:

```
mcrcm L=20 T=6.4 rell alpha0=0.2 rzn=experiment.rzn
rvrcm rzn=experiment.rzn
```

A `.rzn` file consists of three parts: The header contains all control parameters, e.g. $L$, $T$, interaction type, etc. The footer contains elapsed time spent in `mcrcm` as well as Metropolis statistics, i.e. acceptance rate for Metropolis proposals. In between is a sequence of permutation realizations.

## 9.18 MCMC output

Outputs from `mcrcm` fall into two categories: (1) sample statistics of random variables, which are always printed; (2) per-realization values of specified random variables,

which are optionally printed.

Each invocation of `mcrcm` prints information such as the following. For interactive use, one views these data on-screen, or may send them to a printer. When running through a set of parameters, typically this output is redirected to a self-descriptive filename, e.g. `L_80_T_6.7_alpha_0.000.txt`, as described in chapter 10.

```
# RNG = Mersenne twister
# L = 80 d = 3 N = 512000
# Boundary conditions: periodic.
# T =   6.7000000 beta =   0.1492537 alpha0 =   0.0000000
# gamma =   0.0500000
# Interactions:  constant r_ell.
# Initial permutation: identity.
# Site selection for Metropolis sweeps: sequential.
# Thermalization is detected by 40 turning points of system energy
# smoothed over a 200-point window.
# Terminate after 100000 accumulations:
# * 1 SAR  sweep per accumulation.
# * 0 worm sweeps per accumulation.
#
# Initial HDV =   0.0000000   0.0000000   0.0000000
# Sweep 0 HDV = 54809.3500000 54809.3500000   0.0000000
# Thermalization complete:  sweep   1778 H = 359840.2499997
# Ntherm =   1778
# Accumulation complete:  acc 100000 H = 373669.0499998
#
# rho_infty max dev =   0.9140418 at k = 21976
# fI                            =    0.0859582
#
# k=  1 <counts> 339457.4268600
# k=  2 <counts> 27347.3173400
# k=  3 <counts> 5740.3265700
# k=  4 <counts> 2631.4549600
# k=  5 <counts> 1342.3165500
# k=  6 <counts> 804.4376700
# k=  7 <counts> 512.2292200
# k=  8 <counts> 351.5231700
# k=  9 <counts> 252.2729800
# k= 10 <counts> 188.1364500
# k*counts sum                  = 512000.0000000
#
# fM                            =    0.0537150
```

```
# fM/fI                         =    0.6248961
#
# mean_H                        = 371597.3002873
# stddev_H                      = 1701.6752760
# tauint_H                      =  481.1931762
# eta_H                         =    0.9958523
# cssm_H                        =  118.0419619
#
# mean_D                        = 371597.3002873
# mean_V                        =    0.0000000
# mean_h                        =    0.7257760
# mean_d                        =    0.7257760
# mean_v                        =    0.0000000
#
# mean_r2                       = 27347.3173400
# stddev_r2                     =  172.6096035
# tauint_r2                     =  163.4730368
# eta_r2                        =    0.9878400
# cssm_r2                       =    6.9789168
#
# mean_lmax                     = 27502.0656900
# stddev_lmax                   = 8577.6258591
# tauint_lmax                   =    6.0556505
# eta_lmax                      =    0.7165392
# cssm_lmax                     =   66.7494207
#
# mean_jumplenbar               =    0.3754422
# stddev_jumplenbar             =    0.0015945
# maxjumplen                    =    3.7416574
#
# mean_ellbar                   = 1893.6137332
# stddev_ellbar                 =  821.5275072
# tauint_ellbar                 =   12.1998395
# eta_ellbar                    =    0.8484830
# cssm_ellbar                   =    9.0740082
#
# mean_recipmeanspatlen         =    0.0005406
# stddev_recipmeanspatlen       =    0.0002356
# tauint_recipmeanspatlen       =   38.0712869
# eta_recipmeanspatlen          =    0.9488115
# cssm_recipmeanspatlen         =    0.0000046
```

```
#
# mean_wno                    =   20.3278800
# stddev_wno                  =   16.5433065
#
# mean_fS                     =    0.5674866
# stddev_fS                   =    0.4618340
# tauint_fS                   =    3.5586073
# eta_fS                      =    0.5612695
# cssm_fS                     =    0.0027550
# recip_fS                    =    1.7621560
#
# mean_fW                     =    0.0810994
# stddev_fW                   =    0.0082444
# tauint_fW                   =  155.0150517
# eta_fW                      =    0.9871807
# cssm_fW                     =    0.0003246
# recip_fW                    =   12.3305479
#
# Wx < -10:         1 /  100000 = 0.00001000
# Wx = -10:        14 /  100000 = 0.00014000
# Wx =  -9:         0 /  100000 = 0.00000000
# Wx =  -8:       233 /  100000 = 0.00233000
# Wx =  -7:         0 /  100000 = 0.00000000
# Wx =  -6:      2195 /  100000 = 0.02195000
# Wx =  -5:         0 /  100000 = 0.00000000
# Wx =  -4:      9496 /  100000 = 0.09496000
# Wx =  -3:         0 /  100000 = 0.00000000
# Wx =  -2:     22900 /  100000 = 0.22900000
# Wx =  -1:         0 /  100000 = 0.00000000
# Wx =   0:     30423 /  100000 = 0.30423000
# Wx =   1:         0 /  100000 = 0.00000000
# Wx =   2:     22919 /  100000 = 0.22919000
# Wx =   3:         0 /  100000 = 0.00000000
# Wx =   4:      9341 /  100000 = 0.09341000
# Wx =   5:         0 /  100000 = 0.00000000
# Wx =   6:      2196 /  100000 = 0.02196000
# Wx =   7:         0 /  100000 = 0.00000000
# Wx =   8:       261 /  100000 = 0.00261000
# Wx =   9:         0 /  100000 = 0.00000000
# Wx =  10:        19 /  100000 = 0.00019000
# Wx >  10:         2 /  100000 = 0.00002000
```

```
# Wx_min    =     -12
# Wx_max    =      12
#
# OPENS:
#   None.
# CLOSES:
#   None.
# HS:
#   None.
# TS:
#   None.
# GK:
#   Metropolis keeps:   44596819174 / 51200000000 ( 87.103%)
#   Metropolis changes: 6603180826 / 51200000000 ( 12.897%)
# Elapsed thermalization seconds: 302.841613
# Elapsed accumulation   seconds: 136966.313018
# Elapsed total seconds:          137269.154631
```

In addition to the above sample statistics, per-realization values of specified random variables may also be printed. A full list of options may be found by invoking `mcrcm --help`. For example, `mcrcm hv=1` (i.e. $H$ verbosity is set to 1, rather than its default value of 0) results in the following output:

```
# RNG = Mersenne twister
# L = 10 d = 3 N = 1000
... (header information is similar to the previous example)
# Initial HDV =   0.00000   0.00000   0.00000
# Sweep 0 HDV = 132.60000 132.60000   0.00000
    0 132.60000 132.60000   0.00000 132.60000   0.00000 # 0 H D V
    1 190.40000 190.40000   0.00000 161.50000   0.00000 # 0 H D V
   ... (many per-realization values omitted for brevity)
  597 629.00000 629.00000   0.00000 603.73065   0.00000 # 0 H D V
  598 632.40000 632.40000   0.00000 603.73065   0.00000 # 0 H D V
  599 642.60000 642.60000   0.00000 603.73065   0.00000 # 0 H D V
# Thermalization complete:  sweep    600 H = 642.6000000
  600 598.40000 598.40000   0.00000 598.40000 598.40000 # 1 H
  601 591.60000 591.60000   0.00000 591.60000 591.60000 # 1 H
  602 581.40000 581.40000   0.00000 581.40000 581.40000 # 1 H
   ... (many per-realization values omitted for brevity)
 1597 581.40000 581.40000   0.00000 581.40000 581.40000 # 1 H
 1598 578.00000 578.00000   0.00000 578.00000 578.00000 # 1 H
 1599 595.00000 595.00000   0.00000 595.00000 595.00000 # 1 H
```

```
# Accumulation complete:  acc 1000 H = 372745.0766667
#
# rho_infty max dev =   0.9471790 at k = 27
# fI                            =    0.0528210
... (statistical output as in the previous example)
# Elapsed thermalization seconds: 0.241671
# Elapsed accumulation   seconds: 0.477683
# Elapsed total seconds:          0.719354
```

Notice that all the sample statistics are printed as before, on lines which start with a pound sign. In addition, since $H$ verbosity was requested, non-pound-sign lines show Metropolis sweep number, $H$, $D$, $V$, smoothed $H$ (see section 9.6), and thermalization flag times smoothed $H$. With pound-sign lines stripped out or ignored, a graphing utility can then be used to produce a plot such as that shown in figure 9.4 on page 95.

## 9.19 Pseudorandom numbers

The default pseudorandom-number generator ("RNG") is the Mersenne Twister [MN]. One may instead select, at compile time via `rcmrand.h`, `rand48` (of lower quality than Mersenne twister), Linux `/dev/urandom` (slower than Mersenne twister), or `psdes` from Numerical Recipes [NR].

## 9.20 Tools

- Linux environment, although: in principle everything other than use of `/dev/urandom` should be portable to other operating systems.
- Optimizing compiler with full warnings enabled: `gcc -O3 -Wall -Werror`.
- Build tool: `make` and automatic makefile generation.
- Performance analyzer: `gprof`. This shows where a program is spending most of its time.
- Error detector: `valgrind`. Finds many (but not all) common errors, e.g. `malloc` without `free`, or double `free`.
- Code navigation: `ctags`. Allows a smart text editor (e.g. `vim`, `emacs`) to jump directly to a subroutine body.
- Graphing utility, used for all plots in this dissertation: `pgr`, which is the author's Python script wrapped around `pylab.plot()`.

Sample `gprof` output:

```
%     cumul.self          self    total
time  secs. secs.  calls us/call us/call name
32.52 1.20  1.20     1500 0.80   1.41  SO_sweep
25.47 2.14  0.94 21597444 0.00   0.00  get_mtrand_double
 9.49 2.49  0.35     1500 0.23   0.51  U_sweep
 7.32 2.76  0.27     1000 0.27   0.27  get_pmt_winding_numbers
 5.15 2.95  0.19     1000 0.19   0.19  get_mean_cycle_length
 4.61 3.12  0.17 12000000 0.00   0.00  get_Delta_V_SO
 2.71 3.22  0.10 12000000 0.00   0.00  get_mtrand_int32
 2.71 3.32  0.10     1000 0.10   0.10  get_mean_jump_length
 2.44 3.41  0.09 12000000 0.00   0.00  find_dxy_dyx_xoy
 2.17 3.49  0.08     1000 0.08   0.14  get_rho_L_pi
 1.63 3.55  0.06     1000 0.06   0.06  pmt_get_cycle_counts_and_lmax
 ...
 ...
 0.00 3.69  0.00        1 0.00   0.00  report_metro_stats
 0.00 3.69  0.00        1 0.00   0.00  set_default_mcmc_params
 0.00 3.69  0.00        1 0.00   0.12  set_up_cycinfo_list
 0.00 3.69  0.00        1 0.00   0.00  therm_ctl_free
 0.00 3.69  0.00        1 0.00   0.00  therm_ctl_init
```

What is being seen here:

- The `name` column shows the names of all subroutines invoked during the execution of the program.
- The `% time` column shows the percent of total CPU time spent in a given subroutine. The output is sorted by decreasing order of CPU time.
- The self-seconds column shows the total wall time spent in the given subroutine; the cumulative-seconds column shows total wall time spent in the given subroutine and all those listed above it.
- The `calls` column counts the number of invocations of the subroutine. This helps the programmer distinguish between a routine which is time-consuming on each call, and a routine which is quick but perhaps overused.
- The `self us/call` and `total us/call` columns displays the mean and total number of microseconds spent in invocations to the subroutine.

## 9.21 Performance

Memory requirements with $N = L^3$ points and $M = 10^5$ sweeps, taken from the `VSZ` field of a Linux `ps aux` listing, are shown in table 9.1. A few hundred bytes are needed for each lattice point; this scales linearly with $N$. As well, for each random variable, the full time series over all $M$ permutations in the MCMC sequence is saved. This scales linearly with $M$.

Figure 9.14 shows CPU times as a function of $L$, $T$, and $\alpha$. For $T$ near $T_c$ and $L = 30, 40, 50$, in order to reduce variance in the critical-slowdown regime, $10^6$ sweeps were performed. The sawtooth effect is due to the fact that simulations for a few initial values of $T$, e.g. $T = 6.74, 6.76$, were performed in a different computing environment than later simulations for more values of $T$ including $T = 6.75$, 6.834, etc.

| $L$ | 10 | 20 | 30 | 40 | 50 | 80 | 100 | 200 |
|---|---|---|---|---|---|---|---|---|
| Megabytes | 25 | 26 | 29 | 34 | 44 | 103 | 177 | 1243 |

TABLE 9.1. Memory requirements for `mcrcm` with $M = 10^5$ and varying $L$.

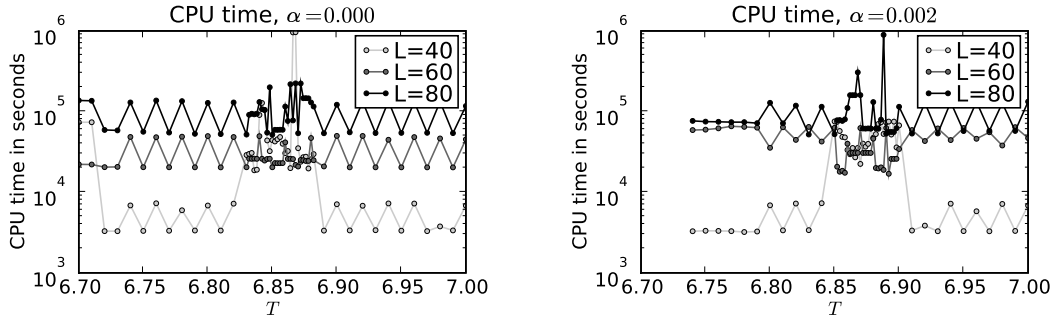


FIGURE 9.14. CPU time in seconds as function of $L$ and $T$, for $\alpha = 0$, 0.002.

As was discussed in section 9.5, most computations in our simulations are $O(N)$, with an $O(N^2)$ component that has a small constant of proportionality. See figure 9.15 which substantiates this claim. See also section 7.8 for a comparison of the SAR algorithm with the worm algorithm.

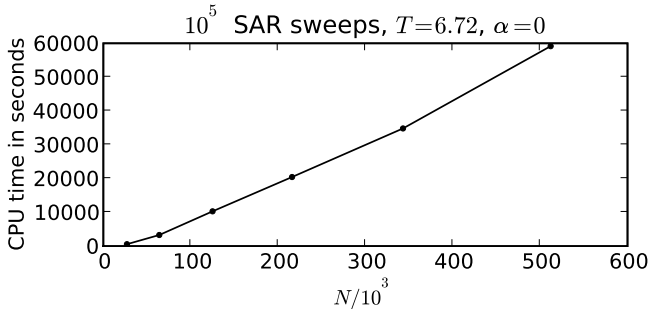


FIGURE 9.15. Scalability of the SAR algorithm. CPU times for $10^5$ sweeps are shown as a function of $N = L^3$ for $L = 30$, 40, 50, 60, 70, and 80. SAR time is nearly linear in $N$.