Disorganized Ramblings on Rejection-Free Markov Chains for Lattice Quantum Gravity

Walter Freeman, for the Sept 27 HET Seminar

1 The problem

A configuration is made up of a collection of N associated simplices of ranks 0-4, each associated with a unique move that converts that simplex of rank d into another simplex of rank $4-d$. These are called the Pachner moves.

Each move is associated with some change in the action $\Delta\mathcal{S}_i$ and an associated Metropolis probability $P(i)$. If that move is forbidden by the local geometry, then $P(i) = 0$. Per Jack and Scott the current accept rate is $\mathcal{O}(10^{-4})$ for fine lattices, which have $N \simeq 10^5$. (Note that N includes simplices of all ranks.) The accept rate drops to $\mathcal{O}(10^{-6}$ for superfine lattices; this is too low to make any sort of reasonable progress through the Markov chain, so not much has been done in this regime. However, it is very physically interesting to enable continuum approximations.)

2 The algorithm, first try

To design an algorithm that generates the same results as the current one, we need to compute the following as quickly as possible:

- Which move i will be the first one to be accepted by Metropolis
- How many tries it took to accept that move (since per the Metropolis algorithm simulation time is measured in attempted moves, not accepted ones)

Rather than trying moves until one is accepted, I propose an algorithm that knows the probability of all possible moves ahead of time, and then uses only *one* random number to choose which is accepted. Call the probability that any given move i will be the one eventually accepted $\tilde{P}(i)$. Then

$$\tilde{P}(i) = \frac{P(i)}{\sum_j P(j)},$$

i.e. the probability of eventually accepting any move is the fraction of the total Metropolis probability that it has.

For the second point, the number of Metropolis suggestions that are rejected before one is eventually accepted is independent of which one is eventually chosen. Thus it suffices to determine the probability that any given Metropolis suggestion will result in a rejection.

The probability of accepting any move i on any particular trial is equal to the probability of choosing it times the probability of accepting it, *i.e.*

$$P_a(i) = \frac{P(i)}{N}$$

Since these probabilities are mutually exclusive, we can just add them, to determine that the probability of accepting the move that we choose is

$$P_a = \frac{\sum_i P(i)}{N}$$

i.e. the probability that the move that is chosen at random will be accepted is just the average accept probability of all the moves.

However, the Metropolis Markov chain will have a long sequence of one configuration, since every time a move is rejected, that configuration is added to the chain again. It is thus insufficient to calculate the next configuration; one must also calculate how long the system stays stuck in the current one. (This can be trivially shown to be necessary by thinking about a two-state system.)

The probability of accepting any move after rejecting n previous trials is thus

$$P(n) = P_a(1 - P_a)^n$$

and the number of previously rejected trials can be determined from a single random number r from (0,1) as

$$n_{\text{reject}} = \text{floor}(\log_{1-\text{P}_\text{a}}(\text{r})).$$

(I tested this with a quick Monte Carlo and it checks out. I think Bharath has a clever proof of this but it's tangential to our main point here.)

3 The implementation and speed gains

The key is that making any move i:

- Creates and/or destroys certain simplices, changing N and adding/removing certain moves from the pool of possibilities
- Alters the probability $P(j)$ only of moves in the local neighborhood of i (that move, its neighbors, and their neighbors, or something like that)

I propose to create an ordered binary tree (in the traditional computer science sense) of all possible moves. The index of each element on the binary tree is a simplex index i; these do not need to be continuous but they will be for us. Each node on the tree contains:

- Information about the move that it corresponds to (*i.e.* the simplex number and rank)

- The probability $P(i)$ of accepting that move if chosen, based on $\Delta\mathcal{S}(i)$ (zero if that move is impossible)
- Pointers to the left and right child nodes, and a pointer upward to the parent node
- The total probability of this node and all of its children

Note that the relation between nodes on the binary tree has nothing at all to do with the geometry of the lattice; it is only a tool for searching and indexing.

To determine which move j will be the one that the Metropolis algorithm eventually accepts:

1. Generate a random number r between 0 and the total probability $\sum P(i)$ in the whole tree (which you can read off of the root node).
2. Start at the root of the tree and traverse it, looking for the node j that will be accepted.
3. At each node, we have three options: it's either this node, or it lives somewhere along the left or the right branch.
4. Define P_{here} as the probability of the current node, P_{left} as the *total* probability along the left branch, and P_{right} as the total probability along the right branch.
5. Traverse the tree according to the following:
    - If r is less than P_{left}, then go left
    - If r is greater than P_{left} but less than $P_{left} + P_{here}$, then we are at the node j: accept it
    - If r is greater than $P_{left} + P_{here}$, then $r \leftarrow r - (P_{left} + P_{here})$ and go right

This allows us to determine the move j that will eventually be chosen in $\mathcal{O}(\log(N))$ time. (I can't think of a way to do it in $\mathcal{O}(1)$ time; this binary tree business seems harder than it necessarily needs to be, but I can't think of a better alternative.)

Then, once we've chosen the move j:

1. Actually make that move (change the geometry)
2. Create a list of other moves whose probability might be changed by that move
3. If a simplex is destroyed, we need not delete it from the tree – just zero it
4. Likewise, when a simplex is created, we "un-zero" the place in the tree corresponding to its index
5. We *could* deal with dynamic restructuring of the tree but we don't *need* to
6. We will need to update the accept probabilities $P(i)$ of the simplices which share a triangle with those simplices that were altered by the move that was just made:
    - Compute them (using the preexisting method)
    - Traverse the tree looking for them and update their probability
    - Traverse the tree back upward and update the "total probability here and below" field

I imagine that all of the $\mathcal{O}(\log N)$ operations will be extremely cheap, since they are just pointer traversal and arithmetic. The expense will come in the computation of $P(i)$ for the new simplices and those whose local neighborhood was altered by the chosen move. But this shouldn't be more than a few dozen per accepted move – far fewer than the $\mathcal{O}(10^4)$ that are required on fine lattices using standard Metropolis. *(Note: I wrote these notes before we coded it. It turns out that there may be several thousand per accepted move. But this is still much faster than the old approach, as we'll see.)*

4 Global plus local action: the algorithm, second attempt

However, this algorithm is not optimal for dynamical triangulations on large lattices.

In DT simulations there is also a global piece of the action S_{global} that affects all Pachner moves of the same order equally. This piece of the action changes whenever a Pachner move of types 1, 2, 4, or 5 is made, which would require a complete retraversal of the ponderance tree (which might contain 2×10^5 entries in current simulations and ten times that number in simulations we would like to do).

We seek a way to incorporate S_{global} into the algorithm while avoiding any step whose cost is proportional to the number of possible moves, *i.e.* we need to avoid traversing the whole tree. It would be prohibitively expensive to go through every single node in the trees and update them all just to do the probability updates corresponding to the global action.

4.1 Adding the global action

Since $\mathcal{S}_{\text{global}}$ will change frequently, we can only store $\mathcal{S}_{\text{local}}$ in the tree.

I propose that we create five "subtrees", one for each Pachner move type, and store only the probability based on $\mathcal{S}_{\text{local}}$, called P_{local} in those trees. The head of each tree knows the total probability P_{local} of the moves beneath it, as always.

Then – based on the values of $\mathcal{S}_{\text{global}}$ and the total probabilities in each Pachner-type subtree, we first make a five-way choice of which move type we will make. If changes in $\mathcal{S}_{\text{global}}$ do not affect the relative values of P_{local} within a subtree, then nothing needs to change: we can apply the effect of $\mathcal{S}_{\text{global}}$ as an overall weight factor for each subtree, generate one random number to choose which subtree to look at, and descend it searching for the right move as before. Calculating $\mathcal{S}_{\text{global}}$ for each subtree is extremely cheap since it depends only on the total number of simplices of different types.

4.2 Factorizing the probability

However, this is not the case without some modifications. Recall the formula for $P_{A\to B}$, the probability that a move from A to B will be accepted, and Google how to write a piecewise-defined function in LaTeX:

$$P_{A\to B} = \begin{cases} 1, & \text{if } \mathcal{S}_B <= \mathcal{S}_A \\ e^{S_A - S_B}, & \text{if } \mathcal{S}_B > \mathcal{S}_A \end{cases}$$

The action here, in the Metropolis prescription, is the entire action $\mathcal{S}_{\text{local}} + \mathcal{S}_{\text{global}}$. Since it is piecewise defined, we cannot easily separate global and local effects on the probability and write $P_{A\to B} = (P_{A\to B,\text{ global}})(P_{A\to B,\text{ local}})$. For instance, if a certain move greatly reduces the local action, the global action may not affect its probability at all, since it will be equal to 1 by the first case above.

We need an alternate definition for $P_{A\to B}$ that generates the same canonical ensemble as Metropolis but which meets the criterion $P_{A\to B} = (P_{A\to B,\text{ global}})(P_{A\to B,\text{ local}})$. The only thing required for detailed balance is that

$$\frac{P_{A\to B}}{P_{B\to A}} = e^{\mathcal{S}_A - \mathcal{S}_B}.$$

4.3 The square root approach

I propose a different approach here that allows us to factorize the local and global effects. The Metropolis algorithm's accept-reject step requires that these probabilities be no greater than unity, but since this algorithm is "rejection-free", we have no such constraint. These are thus no longer *probabilities* (since they can be greater than one). To allow us to use the same symbol P, and to distinguish them from Monte Carlo configuration weights (which I introduce later), I'll refer to them as the *ponderance* P – which is a generalized version of the Metropolis accept probability[1] that can be greater than unity.

Then we can also maintain detailed balance with the definition

$$P_{A\to B} = \sqrt{e^{\mathcal{S}_A - \mathcal{S}_B}},$$

which is not piecewise defined and thus factorizes neatly as we require.

$$P_{A\to B,\text{local}} = \sqrt{e^{\mathcal{S}_{A,\text{local}} - \mathcal{S}_{B,\text{local}}}} \tag{1}$$

$$P_{A\to B,\text{global}} = \sqrt{e^{\mathcal{S}_{A,\text{global}} - \mathcal{S}_{B,\text{global}}}} \tag{2}$$

(Of course, in implementing this it makes sense to code the square root as a factor of 1/2 in the exponent.)

Then we store P_{local} in the five subtrees, and first use P_{global} along with the total ponderance contained in each subtree to choose which order Pachner move we want to make before descending the chosen subtree and choosing one particular move.

4.4 Handling the reject count

However, this creates a problem with calculating n_{reject}. Recall that this is done using the relation

[1] "Ponderance" is not a common term; it is an archaic word meaning "weight". I use it here because I want to use the word "weight" in a more commonly-used sense later in the algorithm, and because it has the same first letter as "probability" so I can represent both with P.

$$n_{\text{reject}} = \text{floor}(\log_{1-\text{P}_\text{a}}(\text{r})).$$

where P_a is the average Metropolis accept probability (or ponderance) of all the moves. If we use the ponderance as calculated above, this leads to a negative logarithm when P_a is greater than one. This should not be a surprise; if we shift from thinking about accept probabilities to thinking about more abstract ponderances, it might be expected that things connected to transition probabilities might break!

Nonetheless, we need a way to account for this effect: that some configurations are likely to have a longer dwell time in the Markov chain than others.

4.5 The weighted ensemble approach

Up until now, my intent was to reproduce *exactly the same Markov chain* as the RRMH (Rosenbluth-Rosenbluth-Metropolis-Hastings) algorithm, since that algorithm is known to be correct and is widely used. However, starting now, we will need to consider another Markov chain that differs in character (perhaps) from the RRMH one, but nonetheless reproduces the canonical ensemble.

One might also imagine using this algorithm to generate a weighted ensemble rather than to reproduce exactly the Metropolis Markov chain. Thus, instead of producing something that looks like `AABCCCDDDDDAB`, it would produce something that looks like `A (2); B (1); C (3); D (5); A(1); B(1)`. Then an analysis code would simply take a weighted average over configurations, where each configuration in the sequence is weighted with its average dwell time in the MCMC sequence.

To do this, the weights w of each configuration should be proportional to the average number of Metropolis repeats that it would have in a traditional MCMC approach. I believe that this approach will have statistical and performance gains. It will eliminate a further source of stochasticity (the random nature of the dwell time of each move), although this will not be meaningful in DT simulations where analysis code typically only examines one configuration out of many because of long autocorrelation times. It will also require the generation of only one random number per step.

This calculation is much simpler. The average dwell time d_{avg} of a configuration in the sequence, which is proportional to its weight in the ensemble w, is just equal to the inverse of the average ponderance of all moves leading away from it:

$$w \propto d_{\text{avg}} = \frac{1}{P_a}.$$

This approach has no such issue: if the average ponderance is greater than one, then the weight in the final ensemble will be less than one. In the traditional Metropolis sense, this means that the probability of accepting the next suggested move is greater than one, and the dwell time of that configuration in the sequence is less than one – both things that are absurd, in an "unweighted" Metropolis sequence where each configuration is repeated an integer number of times. But this poses no problem in the weighted ensemble.

4.6 Biased sampling, weighted ensembles, and the need for this

There are two methods for doing this that seem equivalent and one, by comparison, that is **wrong**:

1. **Weighted ensemble:** Maintain a "weighted ensemble", where we save every configurations (or every N configurations) in the Markov chain, and save along with each one its weight factor. Then, when computing expectation values of observables, we must do a weighted average. This sort of thing is done in "reweighted ensembles", a different technique, and is something of a pain in the butt.
2. **Biased sampling:** Rather than saving every configuration (or every N configurations), we accumulate the total weight factor of the Markov chain rather than the number of configurations it contains, and save one configuration at predetermined intervals of accumulated weight. I call this *biased sampling*, since here the weight factors amount to a bias that any given configuration will be the one that pushes the chain over the threshold for recording a configuration.
3. **Unbiased sampling:** Save one configuration out of every N. *This is expected to be wrong.*

5 Implementation for Ising

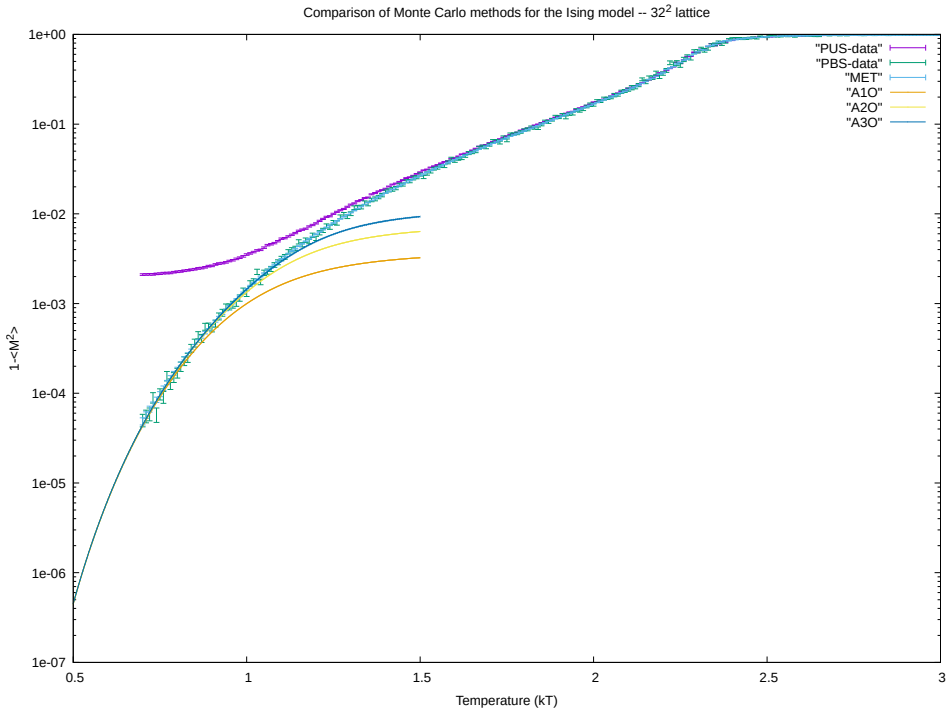
I coded this last December in a general library form that could apply to any Markov-chain Monte Carlo, and applied it to the Ising model, everyone's favorite toy. This is not expected to provide performance gains (except at very low temperature where the accept rate plummets), but is a proof of concept to see if the method is valid.

Here we can do the calculation in four ways:

1. **RRMH algorithm**, the old standby, known to be correct
2. **Ponderance with biased sampling**, which we hope to be correct
3. **Ponderance with unbiased sampling**, which we expect to be wrong at low temperature
4. **Perturbative calculation** done by explicitly writing down the partition function for configurations with up to three unaligned spins, expected to be valid in the limit of temperatures

I tested this on a 32^2 Ising lattice and confirmed that ponderance with biased sampling reproduces the RRMH results. At low temperature it matches the perturbative results, but ponderance unbiased sampling **is wrong**. (This is because it generates a Markov chain consisting of switches between configurations with 1 and 0 unaligned spins without considering the fact that the configurations with one unaligned spin have a much lower Boltzmann factor and should contribute less to the partition function. Biased sampling correctly captures this.)

This proof of concept demonstrates that this algorithm at least can produce correct results. I did this in December 2020 and was surprised at how clean the results were (MCMC never looks this good in practice!)



Comparison of three MCMC approaches to the Ising model with perturbative calculations. PUS and PBS are "ponderance unbiased sampling" and "ponderance biased sampling", and MET is the familiar RRMH algorithm. "AxO" are analytic first, second, and third order expansions of the partition function. Note that RRMH and PBS match at all temperatures, and agree with the perturbative expansion at low temperature where we expect it to be valid; PUS matches at high temperatures but is wrong at low temperatures.

This is a strong suggestion that this algorithm is valid. Can we then apply it to DT for quantum gravity?

6 Work by Jack and Mingwei

To apply this to dynamical triangulations, we need three things:

1. Code that determines the ponderance of any given Pachner move
    - This is hard and requires detailed knowledge of the local geometry
2. Code that determines which moves will have changed ponderances after a move is made
    - This is hard and require detailed knowledge of the local geometry
3. Code that manages the Markov chain and implements the rejection-free algorithm
    - This is easy but is possible to screw up (since it implements a new MCMC)

Jack modified the piece of his current RRHM code to produce #1, since it already does this calculation to determine the accept probability. He confirmed that this code functions correctly with a "precalculate all and cache" version of the RRMH algorithm – which *itself* showed performance gains!

Jack and Mingwei worked over the spring to prepare #2, which required new thinking. They concluded that, when a Pachner move is made, the only moves with a changed value of P_{local} are:

- Moves corresponding to a simplex that is directly altered by the update, one of its subsimplices, or one of its supersimplices
- Moves corresponding to a simplex that shares a subsimplex with one of the above

They verified that this is sufficient by recalculating *all* ponderances and then verifying that this procedure catches all of the ones that change.

7 Implementation of the rejection-free algorithm

I wrote a bunch of code in June that implements the new algorithm based on the work done by Jack and Mingwei above. Doing this involved a few technical details which I can talk about if there is time:

7.1 Ponderance update deduplication

7.2 Smarter ponderance update deduplication

7.3 Movetype bias calculation

7.4 Implementation of ponderance biased sampling

7.5 Ponderance floating point drift correction

8 Assessing the new algorithm

Ultimately, we are interested in two fundamental questions:

- Is the Markov chain equivalent? (It is clearly not the *same*, but does it give the same physics?)
- Does this method let us generate more accepted moves faster than RRHM? How much faster? Under what conditions?

8.1 Validating the algorithm

Recall that in our model, we have three couplings. In a crude sense:

- Two couplings (the strength of the measure term and the strength of gravity, called κ_6 and κ_2) are used to fix the lattice spacing and move "toward/away from" the phase transition
- One more coupling called κ_4 (the cosmological constant, *i.e.* "the energy cost of spacetime") is used to tune the simulation volume on the fly to the desired point.

When doing a physics analysis, some of the information we need can be gathered only from extensive calculations later. However, some is already resident in memory during the Markov chain. The two most salient ones for this purpose are:

- The auto-tuned value of κ_4, the cosmological constant, that results in the desired volume
- The ratio of the number of triangles to the number of four-simplices, giving the average curvature

We can use these quantities – of physics interest but also generated as we go – as probes.

If our Markov chain is equivalent to the RRMH one, then these ought to be equivalent.

As a test, we generated Markov chains using the new rejection-free algorithm with the same values of κ_2, κ_6, and the desired N_4 (number of four-simplices, i.e. spacetime volume), and calculated central values and uncertainties for N_2/N_4 and κ_4.

It turns out that they are *not* quite the same – there is a statistically significant overlap. However, there are some arbitrary choices made in generating these Markov chains: for instance, how often do you adjust κ_4 in the automatic tuning progress, and by how big of steps?

Since the old algorithm measures simulation time in different units (attempted RRMH moves) than the new one (accumulated weight), I didn't attempt to do κ_4-tuning in an exactly equivalent way, since it is a nonphysical, arbitrary thing anyway.

It turns out that while there *are* statistically significant differences in κ_4 between the RRMH and rejection-free Markov chains, that difference is smaller than the granularity of the κ_4 tuning steps in the RRMH chain. Jack and I decided that this difference is because of the tuning granularity and process rather than an actual difference in the physics and decided that the new algorithm passed physics validation.

8.2 Performance

Some performance numbers sent by Marc Schiffer today for the number of accepted moves per second on a quad-core computer:

	"Fine"	"Superfine"	"Ultrafine"
RRMH "parallel rejection"	230	50	11
Rejection-free	7500	16000	6000

I am out of time to type notes but I can talk more about the performance profile here, and perhaps show it in action.

9 Further developments

9.1 Parallel tempering

9.2 Wigner energy, tech debt, and code refactors