Introduction to Hybrid Monte Carlo

Samuel Kováčik

Lecture notes

Abstract

These two lectures build up the Hybrid (Hamiltonian) Monte Carlo algorithm from the ground up. We start from the question of why one would ever integrate numerically, watch brute force fail as soon as the number of variables grows, recover from that failure by sampling randomly and then by sampling *importantly*, and arrive at the Metropolis algorithm and at HMC. The second lecture is about everything that happens afterwards: writing the code, testing it, deciding whether to trust it, and paying the very real computational bill for a phase diagram.

Contents

---

1 Lecture 1

1.1 Why compute anything numerically

The basic loop of physics can be drawn as

$$\underbrace{\text{observation}}_{\text{mathematical representation}} \xrightarrow{\text{calculate}} \underbrace{\text{prediction}}_{\text{mathematical representation}} .$$

On both ends there is a mathematical representation of something real, and in between there is a calculation: a black box that eats numbers and returns numbers. Physics is usually taught as if the middle step were unproblematic. It is not, and how we choose to perform it shapes what we end up learning.

There are three ways to build that box.

Analytic.
The "nice" option. It is worth stressing *why* it is nice: not because closed-form expressions are elegant, but because they *offer explanations*. A formula tells you which parameter controls what, where the limits are smooth and where they are singular, and what happens if you change something you have not tried yet.

Numerical.
Costly, but it does not rely on luck. An analytic solution exists only if somebody is clever enough, or lucky enough, to find it. A numerical solution exists whenever you can afford the computer time, and the class of problems it applies to is enormously larger.

Combined.
Almost always the winner in practice. Use analytics to transform the problem into something tractable – integrate out what can be integrated out, rescale, exploit symmetries – and then throw numerics at what is left.

The reason to care about explanations is that they lead to understanding, and understanding leads to new ideas: to the question "what if I change this. . . ?". Two standard examples make the point. Planck arrived at the quantum hypothesis because he was trying to explain a spectrum, not merely to reproduce it. And in the Kepler problem the analytic solution tells you at a glance that bound orbits are closed ellipses – something you could stare at in numerical output for a long time without noticing.

None of this means numerics is second-class. Numerics is just as instructive, provided you play around with it: change parameters, break the code on purpose, push it into regimes where you know the answer. That playing around is precisely what makes numerics time-consuming.

It never is just a simple simulation.

Goal of these lectures. Learn the basics of HMC, and learn them in a way that is instructive rather than merely operational.

1.2 Outline

- computing $\langle x^2 \rangle$ the hard way, with L variables,
- HMC for a simple model,
- HMC for a complex model.

1.3 Warm-up: a Gaussian done by hand

Before we compute anything numerically, let us solve a problem exactly, so that we have something to compare against. Consider the distribution $P(x) \sim e^{-x^2}$. Its normalisation is the classic polar-coordinate trick: square the integral, reinterpret the result as a two-dimensional integral, and change variables,

$$\int_{-\infty}^{\infty} e^{-x^2} \mathrm{d}x = \sqrt{\int_{-\infty}^{\infty} e^{-x^2} \mathrm{d}x \int_{-\infty}^{\infty} e^{-y^2} \mathrm{d}y} = \sqrt{\int_0^{2\pi} \mathrm{d}\varphi \int_0^{\infty} e^{-r^2}\, r\, \mathrm{d}r} = \sqrt{2\pi \int_0^{\infty} \tfrac{1}{2} e^{-t}\, \mathrm{d}t} = \sqrt{\pi}.$$

Homework. Try the same for $\displaystyle\int_{-\infty}^{\infty} e^{-x^4}\mathrm{d}x$ and see how far the trick takes you.

The normalised distribution is therefore

$$\boxed{P(x) = \frac{1}{\sqrt{\pi}}\, e^{-x^2}}$$

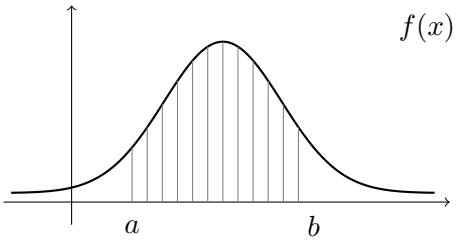
and its second moment follows from a second trick: introduce a parameter that is not there, differentiate with respect to it, and set it back to its original value,

$$\langle x^2 \rangle = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} x^2 e^{-x^2}\mathrm{d}x = \left[-\frac{\partial}{\partial a} \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} e^{-ax^2}\mathrm{d}x \right]_{a=1} = \left[-\frac{\partial}{\partial a} \sqrt{\frac{\pi}{a}}\, \frac{1}{\sqrt{\pi}} \right]_{a=1} = \frac{1}{2}.$$

Both steps worked because a trick happened to be available, and that is the point worth remembering. The availability of tricks is not a strategy but luck, and the luck runs out as soon as the integrand stops being Gaussian.

So let us now pretend we know nothing about any of this and ask the honest question: how would we compute $\langle x^2 \rangle$ numerically?

1.4 Numerical integration



We want the area under the curve between a and b, and we do not know a formula for it. The basic idea of all numerics applies: *simplify the problem down to a single elementary task, and let the computer repeat that task a large number of times.* Here the elementary task is the area of one simple geometrical shape, a rectangle,

$$\int_a^b f(x)\,\mathrm{d}x \;\longrightarrow\; \sum_{i=1}^{N} f(x_i)\,\Delta, \qquad \Delta = \frac{b-a}{N}.$$

Nothing forces the slices to be equally wide; you can be fancy and use non-uniform Δ_i, concentrating them where the function varies fastest.

It is worth having the following exchange with yourself once, explicitly.

Q: Wait – that region is not a rectangle.

A: No, but the discrepancy shrinks in the $\Delta \to 0$ limit.

Q: Except that as Δ shrinks the number of rectangles grows, and so does the total number of small errors being added up.

A: Correct, and this is exactly why the $\Delta \to 0$ (equivalently $N \to \infty$) limit has to be investigated properly rather than assumed. It has been done: for smooth integrands the error of the midpoint rule falls off faster than the number of terms grows, and the sum converges. Knowing *that* such an analysis exists, and roughly what it establishes, is part of using the method responsibly.

Check the example files: simple sum integration

Once you have such a code, interrogate it. What can go wrong? The integration range is finite while the true integral is not; the sum has to converge and you should be able to see it converge; round-off accumulates. How do we test any of this? Run checks whose answer we already know – the simplest being $\langle 1 \rangle = 1$, which tests normalisation, range and step size all at once.

To see where the real difficulty lies, keep the same question, $\langle x^2 \rangle$, but embed it in a distribution with many additional variables that do nothing at all:

$$P(x, \underbrace{y_1, y_2, \ldots, y_L}_{\text{decoys}}) = \frac{e^{-\left(x^2+y_1^2+y_2^2+\cdots+y_L^2\right)}}{\pi^{\frac{L+1}{2}}}.$$

The decoys factorise and integrate to one, so the answer is still $1/2$. The computer, however, does not know that.

Check the example files: the same integral with decoy variables, and with timing

1.5 The curse of dimensionality

If we brute-force a lattice in L dimensions with N points per direction, the number of evaluations is

$$\#\text{ steps } \sim N^L.$$

Suppose one multiplication costs of order 1 ns, and take $N \approx 10^6$, the sort of resolution you need if you care about several digits. For $L = 1$ the whole calculation finishes in about a millisecond – lightning fast. For $L = 2$ it takes a quarter of an hour, and for $L = 3$ it takes tens of years. The trouble is that L sits in the exponent, and no amount of faster hardware fixes an exponent.

This is where analytics, and your own brain, pay for themselves: every variable you can integrate out by hand removes a whole factor of N from the bill. We cannot always do that, however, and matrix models – in which the number of variables is N^2 or more – are precisely the case where we cannot.

1.6 Random sampling

So what do we do instead? Try the naive thing: *probe randomly.* Instead of visiting every point of a grid, draw points at random and average over them.

The intuition is familiar from polling. Imagine you need to check that the mean height of some population is X. You do not need to measure everyone; a few thousand people chosen without bias will tell you the answer to within a well-understood error. Crucially, the accuracy of this estimate is controlled by the number of samples and not by the size of the population – which, translated back to our problem, means it is not controlled by L.

Check the example files: random evaluation of $\langle x^2 \rangle$

How many samples are enough? Do not guess – plot the running estimate against the number of samples and watch it converge. The fluctuation of that curve is also your first, crude error estimate.

One issue nevertheless persists: **over what range do we sample?** We are still drawing points uniformly from some finite box, and if the box is too small we bias the answer, while if it is too large we waste almost all our samples in regions where the integrand is negligible. This too can be tested: run with several ranges and check whether the correct value is reachable within our computational budget.

The bookstore problem. A bookstore earns most of its money on a handful of days a year, but it has to stay open all year round in order to catch them. Uniform sampling has exactly

this structure: we add up a great many numbers of which only a few matter. In the Gaussian example, samples drawn from $|x| > 3$ contribute essentially nothing, yet in a large box they are the overwhelming majority of our effort.

The fix is to stop sampling uniformly and start sampling where the weight actually is. This is *importance sampling*, and everything that follows is a way of doing it.

1.7 The Metropolis algorithm

Write the weight in the physicist's form. For $S(x) = x^2$,

$$P(x) = Z^{-1} e^{-S(x)},$$

which is nothing but a Boltzmann factor with S playing the role of an energy. The algorithm generates a chain of configurations $x_1, x_2, \ldots$ whose distribution is P:

1. Pick an initial x_1.
2. Propose $x_{\text{new}} = x_{\text{old}} + \varepsilon$, with ε drawn from a normal distribution.
3. Compute $\Delta S = S(x_{\text{new}}) - S(x_{\text{old}})$.
4. If $\Delta S < 0$, accept.
5. Otherwise accept if $e^{-\Delta S} > r$, where r is a uniform random number, $0 \le r \le 1$.
6. If accepted, update x; if rejected, restore x_{old}.

Steps 4 and 5 can of course be compressed into the single condition $e^{-\Delta S} > r$, since $e^{-\Delta S} > 1 \ge r$ whenever $\Delta S < 0$. To check that the signs are the right way round, think of S as an energy: if a move lowers it, take it; if it raises it, take it sometimes, with a probability that falls off exponentially in how much it raises it. The "sometimes" is what prevents the chain from sliding into the nearest minimum and staying there.

One detail is easy to get wrong: a *rejected* step is still a step, and the old configuration must be entered into the chain again. Silently dropping rejections spoils the statistics.

Why it works: detailed balance

We want the chain to leave the distribution P invariant. A sufficient condition is detailed balance,

$$P(x'|x)\, P(x) = P(x|x')\, P(x').$$

The transition consists of two independent decisions – what to propose, and whether to accept – so the kernel factorises,

$$P(x'|x) = \underbrace{g(x'|x)}_{\text{propose}}\ \underbrace{A(x'|x)}_{\text{accept}}.$$

Substituting, detailed balance turns into a condition on the acceptance ratio alone,

$$\frac{A(x'|x)}{A(x|x')} = \frac{P(x')}{P(x)}\,\frac{g(x|x')}{g(x'|x)},$$

and the standard solution – the one that accepts as often as the condition permits – is

$$A = \min\left(1,\ \frac{P(x')\, g(x|x')}{P(x)\, g(x'|x)}\right).$$

For a symmetric proposal the g's cancel, P appears only as a ratio so Z cancels too, and we are left with the recipe above. That Z drops out is not a technicality; it is the reason the method is usable at all, since Z is exactly the object we cannot compute.

Note. One has to be careful with *asymmetric* proposals. For example $x_{\text{new}} = x_{\text{old}} \cdot (1+\varepsilon)$ has a non-trivial $g(x|x')/g(x'|x)$, and we must keep that factor. Forgetting it produces a chain that samples a subtly wrong distribution – and, unlike a crash, this failure is silent.

Check the example files: Metropolis

What can go wrong? Remarkably little. The method is robust, it is easy to implement, and it applies to essentially any weight you can evaluate. It is a very good method.

There is, however, one tunable aspect: how do we generate ε? Usually as a normal random number with mean $\mu = 0$ and some width σ, and the choice of σ matters:

- σ too small $\Rightarrow$ almost everything is accepted, but each step moves nowhere; the chain crawls across configuration space.
- σ too large $\Rightarrow$ proposals land far up the potential and are rejected; most of the computer time produces repeated configurations.

Either way, successive configurations in the chain are strongly correlated. This is the *autocorrelation* problem: your chain may contain a million entries but only a few thousand statistically independent ones, and it is the latter number that sets your error bars. Tuning σ trades one failure mode against the other, but it does not remove the problem.

1.8 Hybrid Monte Carlo

HMC attacks precisely this. (The name is unfortunate; read it as *Hamiltonian* Monte Carlo, although "hybrid" has a history of its own.)

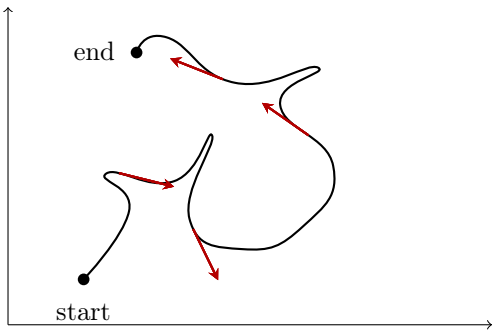
The idea is to stop proposing random jumps and start proposing *trajectories*. To each variable x we add a conjugate partner p – and the notation is suggestive: we will treat it as a momentum, and S as a potential.

1. Generate p_{old} from a normal distribution.
2. Compute $H_{\text{old}} = S(x_{\text{old}}) + \dfrac{p_{\text{old}}^2}{2}$.
3. Evolve the pair with Hamilton's equations, as if p really were a momentum:

$$x \to x + \varepsilon\, p, \qquad p \to p - \varepsilon \underbrace{\frac{\partial S}{\partial x}}_{\text{force}},$$

   iterated many times – this is the *leapfrog* integrator.
4. Compute $H_{\text{new}} = S(x_{\text{new}}) + \dfrac{p_{\text{new}}^2}{2}$.
5. Perform a Metropolis check with ΔH instead of ΔS.
6. Update or restore x; discard p and draw a fresh one next time.

Two properties make this work. First, the exact Hamiltonian flow conserves H, so if we could integrate exactly, every proposal would be accepted; the leapfrog integrator is not exact, but its energy error stays bounded rather than drifting, and the Metropolis check in step 5 corrects for whatever error remains. Second, the leapfrog is reversible and volume-preserving, which is what keeps the proposal effectively symmetric so that detailed balance still holds.



A single HMC update is an entire trajectory. The arrows are tangent vectors: at every point along the path the motion follows the local direction set by the momentum, while the force bends that direction continuously rather than kicking the configuration somewhere at random. The chain therefore travels a long way across configuration space per accepted move. The autocorrelation problem is not eliminated – successive trajectories still overlap – but it is greatly reduced, and that is the whole point of the method.

Check the example files: HMC

Ideas for further reading: ergodicity and the ways in which it fails; consistency checks; fermions; and generalisations to other areas of physics, to discrete models and to non-physical applications. Two concrete improvements of the plain algorithm, both aimed at the ergodicity problem discussed in Section 2.5, are the eigenvalue-flipping algorithm of Kováčik and Tekel, JHEP **04** (2022) 149, arXiv:2203.05422, and the eigenvalue-cluster algorithm of Kováčik and Hrmo, arXiv:2605.29077.

Important. The output of this procedure is *not* $\langle x^2 \rangle$, nor $\langle \mathcal{O} \rangle$ for any particular $\mathcal{O}$. It is a *sequence of configurations distributed with the desired statistics*. Every observable you care about – including ones you have not thought of yet – is then a cheap average over that sequence. This is why it is usually worth storing configurations, or at least a rich set of primitive quantities, rather than only the one number you happen to want today.

For discussion: why is computing Z itself difficult numerically, even though we can sample e^{-S}/Z perfectly well?

---

2 Lecture 2

2.1 Recap

- Monte Carlo works: random sampling beats grids as soon as there are many variables.
- Metropolis: take configurations x_i, define an action S, propose a change, and ask whether $\exp(-\Delta S) > r$.
- HMC is better: it uses the forces $\partial S/\partial x$ to propose long trajectories, which reduces autocorrelation.
- The result of either is a chain of configurations with the desired statistics.

A word on how the work is actually distributed. The expectation when you start is 80% coding and 20% analysis. The reality is closer to 20% coding and 80% analysis – and waiting. Plan for that: the code is the beginning of the project, not the end of it.

2.2 A concrete example

For the rest of these notes we work with the Hermitian one-matrix model

$$S = N \left(b \operatorname{Tr} \Phi^2 + g \operatorname{Tr} \Phi^4 \right),$$

where Φ is an $N \times N$ Hermitian matrix, so the number of real degrees of freedom is N^2. Why matrices? Because they are useful across many areas, and because they appear naturally in models of quantum space and quantum gravity, where the matrix entries replace the coordinates of a classical geometry. The model above is simple enough to study exhaustively and rich enough to have a non-trivial phase structure.

2.3 Practicalities

Why use C++? It is fast, and in this business a factor of ten in speed buys a factor of ten in the physics you can reach. **Why not?** It is more technical than a scripting language – although that gap has narrowed considerably, and there is no shame in using AI assistance for the boilerplate.

You will need:

- a C++ compiler,
- Armadillo for linear algebra (installing via Homebrew is easiest),
- the terminal, and a makefile so you are not retyping compiler flags,
- something for the analysis – Python or Mathematica.

Sooner or later your simulations will run on a remote server, so a little comfort with the command line pays off immediately.

You need only the most basic skills.

Check the example files: main loop; thermalisation; forces

Thermalisation

Why do we need thermalisation at all? Because the code has to be initialised somehow, and whatever we choose – zeros, a random matrix, a configuration from a different parameter point – is generally an atypical state, one that the equilibrium distribution would essentially never produce. HMC moves only so fast, so the first stretch of the chain is a transient in which the system relaxes towards typical configurations, and averages taken over it are simply wrong.

The configurations produced during thermalisation can be discarded on the fly: just do not store them. Sometimes it is better to store them anyway, look at them, and merely exclude them from the analysis – the shape of the transient tells you a good deal about whether your step size is sensible.

How long does it take to reach true ergodicity? Infinity. In practice, somewhat less: monitor an observable, and start measuring once it has stopped drifting and only fluctuates. Near a phase transition, be suspicious – relaxation slows down exactly where you most want precision.

Check the example files: a look at the thermalisation history

Observables

What should we store? There is usually a standard set for a given model – here, the traces $\operatorname{Tr} \Phi^2$ and $\operatorname{Tr} \Phi^4$, the eigenvalues, the action itself – but you are free to define new ones, and defining a good new observable is often where the physics is. A general rule: record primitive quantities and combine them afterwards, rather than storing the combination alone. If you

keep the histories of $\operatorname{Tr}\Phi^2$ and $\operatorname{Tr}\Phi^4$ separately, you can still build any function of them later – including fluctuation observables such as the specific heat, which needs $\langle S^2\rangle - \langle S\rangle^2$ and is therefore lost the moment you store only averages. The exceptions are when combining on the fly is itself instructive, or when storage genuinely becomes the bottleneck.

Check the example files: printing; how the code starts and how it ends

Parameters

It is good practice to expose everything you might want to vary. Here N, b, g and the step size ε should not be hard-coded – otherwise every new run costs a recompilation, and you lose the record of what was run with what. Two options:

- pass them as command-line parameters, e.g. `./yourcode 10 1 -0.5 0.1`,
- or read them from a separate parameter file, which has the advantage that the file can be archived next to the output.

2.4 Do you trust the output?

Suppose the code runs and the acceptance rate is comfortably away from both 0 and 1 – preferably around 0.8 ± 0.15. (There are more carefully motivated target values in the literature; I do not really trust them.) A healthy acceptance rate means the integrator and the step size are compatible. It says nothing about whether you are sampling the right distribution.

For that we need an independent check, and the cheapest one is again a quantity whose value we know: is $\langle 1\rangle = 1$? Concretely, we use the fact that the integral of a total derivative vanishes,

$$\int \partial_\Phi\big(\mathcal{O}\, e^{-S}\big)\,[\mathrm{d}\Phi] = 0,$$

which is the matrix-model version of a Schwinger–Dyson identity. Taking $\mathcal{O} = \Phi$ and using the product rule,

$$\int \partial_\Phi\Big(\Phi\, e^{-S(\Phi)}\Big)\,[\mathrm{d}\Phi] = \underbrace{(\partial_\Phi\Phi)}_{N^2}\underbrace{\int e^{-S(\Phi)}[\mathrm{d}\Phi]}_{1} - \int \Phi\,\partial_\Phi S\; e^{-S}[\mathrm{d}\Phi] = 0,$$

where N^2 is simply the number of components of Φ. With our action, $\Phi\,\partial_\Phi S = 2S_2 + 4S_4$ where $S_2 = Nb\operatorname{Tr}\Phi^2$ and $S_4 = Ng\operatorname{Tr}\Phi^4$, so we obtain the constraint

$$\boxed{\frac{2\,\langle S_2\rangle + 4\,\langle S_4\rangle}{N^2} = 1}$$

This is an exact identity, valid at every N, b and g, and it involves quantities you are measuring anyway. Measure the left-hand side and plot it against the number of steps: statistical scatter around 1 is fine and gives you a feeling for your error bars, but a *systematic* offset is a red flag – typically a wrong force, a wrong factor in the action, or an unaccounted-for asymmetry in the proposal.

2.5 Looking at the results

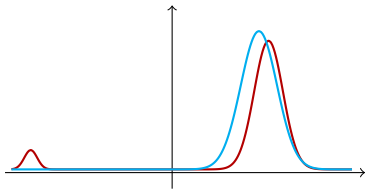
Check the example files: the output and analysis part

Things worth doing routinely: plot HMC trajectories to see how the chain moves; plot eigenvalue distributions, which are where the phase structure is visible; estimate errors properly, including autocorrelation; and assemble phase diagrams.

Is all of this easy? Yes and no.

- **Yes:** the scheme is very general. Nothing above referred to the specific action, so the same skeleton carries over to a large class of models.
- **No:** there are genuinely hard cases – fermions, which give a non-local determinant; indefinite actions, where e^{-S} is not a probability at all; and models with a complicated vacuum structure.

The last of these is worth a picture. Dirac-type models, for instance, have two possible solutions – two local minima of the free action – whose eigenvalue distributions look like this:



The two are close in free energy but separated by a barrier, so a chain started in one of them may never visit the other within any reasonable run time. The simulation will happily report whichever one it found, complete with small error bars, and nothing in the output announces that a second minimum exists.

How do we find the other one?

- Start from many different initial conditions and see how many distinct answers come back.
- Or improve the algorithm itself, so that it can propose a large, coordinated move instead of a local one. If the action has a $\Phi \to -\Phi$ symmetry, flipping the sign of a subset of eigenvalues is enough; this is the eigenvalue-flipping algorithm of Kováčik and Tekel, JHEP **04** (2022) 149, arXiv:2203.05422. Without that symmetry, flipping alone cannot connect inequivalent minima, and one instead selects a cluster of nearby eigenvalues and rescales them coherently – see Kováčik and Hrmo, arXiv:2605.29077. In either case one has to be careful that the Metropolis check accounts properly for the modified proposal.

A related and equally difficult problem is *phase identification*: deciding, in an automated way, which phase a given configuration belongs to, without inspecting every histogram by eye.

2.6 The general scheme

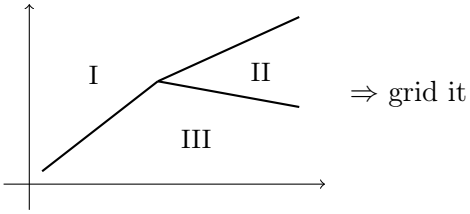
It is worth stating explicitly how little of the machinery is model-dependent. The basic scheme is universal; the physics enters only through

- the variables (e.g. the number and type of matrices),
- the action,
- the forces,
- the constraint used as a consistency check,
- and, prosaically, the printing.

Everything else – the leapfrog, the Metropolis accept/reject step, thermalisation, the analysis pipeline – is shared. In practice this means that turning one working code into another is easy, and that a well-organised code pays off across a whole career rather than a single project.

2.7 The real cost: phase diagrams

Very often the final deliverable is a phase diagram, and the honest way to produce one is to grid the parameter space.



And here the bill arrives. A 50×50 grid means 2500 runs; at one hour per run that is 2500 hours, roughly 100 days of serial computing. Running the points in parallel helps, but it does not change the scaling.

It gets worse. We are interested in the large-N limit, so every point of the grid has to be repeated for at least three values of N before an extrapolation to $N \to \infty$ is credible. Resolving a transition sharply also requires long chains, because autocorrelation times grow precisely where the transition is – whereas finishing your PhD requires a merely *reasonable* number of HMC steps. For a worked example of what this costs in practice, and of how far the extrapolation can be pushed, see the determination of the triple point of the fuzzy-sphere scalar field theory in Kováčik and O'Connor, JHEP **10** (2018) 010, arXiv:1805.08111.

Balancing these demands, defining observables that extract the most information per unit of computer time, and – when that is not enough – modifying the algorithm itself: that is what being proficient with HMC is really about.

References

[1] S. Duane, A. D. Kennedy, B. J. Pendleton and D. Roweth, *Hybrid Monte Carlo*, Phys. Lett. B **195** (1987) 216.

[2] S. Kováčik and D. O'Connor, *Triple point of a scalar field theory on a fuzzy sphere*, JHEP **10** (2018) 010, arXiv:1805.08111.

[3] S. Kováčik and J. Tekel, *Eigenvalue-flipping algorithm for matrix Monte Carlo*, JHEP **04** (2022) 149, arXiv:2203.05422.

[4] S. Kováčik and M. Hrmo, *Eigenvalue-cluster algorithm for matrix Monte Carlo*, arXiv:2605.29077.