If Only My Posterior Were Normal: Introducing Fisher HMC

Adrian Seyboldt
adrian.seyboldt@gmail.com
PyMC-Labs

1. November 2024

Abstract

Hamiltonian Monte Carlo (HMC) is a powerful tool for Bayesian inference, as it can explore complex and high-dimensional parameter spaces. But HMC's performance is highly sensitive to the geometry of the posterior distribution, which is often poorly approximated by traditional mass matrix adaptations, especially in cases of non-normal or correlated posteriors. We propose Fisher HMC, an adaptive framework that uses the Fisher divergence to guide transformations of the parameter space. It generalizes mass matrix adaptation from affine functions to arbitrary diffeomorphisms. By aligning the score function of the transformed posterior with those of a standard normal distribution, our method identifies transformations that adapt to the posterior's scale and shape. We develop theoretical foundations efficient implementation strategies, and demonstrate significant sampling improvements. Our implementation, nutpie, integrates with PyMC and Stan and delivers better efficiency compared to existing samplers.

Keywords Bayesian Inference · Hamiltonian Monte Carlo · Mass Matrix Adaptation · Normalizing Flows · Fisher Divergence

1 Introduction

Hamiltonian Monte Carlo (HMC) is a powerful Markov Chain Monte Carlo (MCMC) method widely used in Bayesian inference for exploring complex posterior distributions. HMC can explore high-dimensional parameter spaces more efficiently than traditional MCMC techniques, which makes it popular in probabilistic programming libraries like Stan and PyMC. However, the performance of HMC depends critically on the parameterization of the posterior space. Modern samplers automate a part of those reparametrizations by adapting a "mass matrix" in the warmup phase of sampling. Tuning the mass matrix is crucial for efficient sampling, as tuning takes a large part of the sampling time, and has a big impact on the efficiency of the sampler after tuning. But even a well-tuned mass matrix can not do much to help us sampling from more challenging posterior distributions, such as those that contain strong correlations in high dimensional models, or that have a funnel-like structure.

In most cases these problems can be overcome by careful, manual reparametrization of models, but this requires a lot of expertise and time. In many cases, good reparametrizations are also data-dependent, which makes it difficult to write a model once, and apply it to a wide range of individual datasets.

A common approach in HMC is to estimate a mass matrix based on the inverse of the posterior covariance, typically in a diagonalized form, to adjust for differences in scale across dimensions. We can think of this as a reparametrization that simply rescales the parameters so that they have a posterior variance of one. It is not obvious however, that this is the best rescaling we could do. And at the same time it is also quite limiting, as it can not fix correlations or funnels in the posterior.

To address these limitations, we propose an adaptive HMC framework that extends beyond the traditional concept of a mass matrix: Instead of just rescaling variables, we allow arbitrary diffeomorphisms to dynamically transform the parameter space. We use the Fisher divergence as a criterion to choose between different transformations, which allows us to adapt the geometry of the posterior space in a way that optimizes HMC's efficiency. By aligning the scores (derivatives of the log-density) of the transformed posterior with those of a standard normal distribution, we approximate an idealized parameterization that promotes efficient sampling.

2 Fisher HMC: Motivation and Theory

2.1 Motivation: Example with independent gaussian posterior

When we run HMC, we compute the derivatives of the posterior log density (the scores). They contain a lot of information about the posterior, but when we compute the mass matrix using traditional methods, we ignore this information.

To illustrate how useful the scores can be, let's start very simple, and assume our posterior is a one dimensional normal distribution $N(\mu, \sigma^2)$ with density p. Let's assume we have two posterior draws x_1 and x_2, together with the covector of scores $\alpha_i = \frac{\partial}{\partial x_i} \log p(x_i) = \sigma^{-2}(\mu - x_i)$. Based on this information we can directly compute μ and σ, and identify the exact posterior, by solving for μ and σ. We get $\mu = \bar{x} + \sigma^2 \bar{\alpha}$ and $\sigma^2 = \mathrm{Var}[x_i]^{\frac{1}{2}} \mathrm{Var}[\alpha_i]^{-\frac{1}{2}}$, where $\bar{x}$ and $\bar{\alpha}$ are the sample means of x_i and α_i respectively. If we take advantage of the scores, we can compute the exact posterior and an ideal mass matrix with no sample variance based on just two points!

This generalizes directly to multivariate normal posteriors $N(\mu, \Sigma)$. Let's assume we have $N + 1$ linearly independent draws x_i, and the scores $\alpha_i = \Sigma^{-1}(x_i - \mu)$. The mean of these equations gives us $\mu = \bar{x} - \Sigma\bar{\alpha}$. It follows that $S = \Sigma^{-1} X$, where the i-th column of S is $\alpha_i - \bar{\alpha}$, and the i-th column of X is $x_i - \bar{x}$. And finally we get $SS^T = \mathrm{Cov}[\alpha_i] = \Sigma^{-1} X X^T \Sigma^{-1} = \Sigma^{-1} \mathrm{Cov}[x_i] \Sigma^{-1}$. We can recognize Σ as the geometric mean of the positive symmetric matrices $\mathrm{Cov}[x_i]$ and $\mathrm{Cov}[s_i]^{-1}$, so

$$\Sigma = \mathrm{Cov}[x_i]^{-\frac{1}{2}} \left(\mathrm{Cov}[x_i]^{\frac{1}{2}} \mathrm{Cov}[\alpha_i] \mathrm{Cov}[x_i]^{\frac{1}{2}} \right)^{\frac{1}{2}} \mathrm{Cov}[x_i]^{-\frac{1}{2}} \tag{1}$$

Given the scores, we can compute the parameters of a normal distribution exactly.

Most posterior distributions are not multivariate normal distributions, or we would not have to run MCMC in the first place. But it is quite common that they approximate normal distributions reasonably well, so this indicates that the scores contain useful information we are currently neglecting.

2.2 Transformed HMC

When we manually reparameterize a model to make HMC more efficient, we try to find a transformation (a diffeomorphism) that changes our posterior distribution so that HMC works better, for the most part so that it resembles a standard normal distribtuion. Formally, if our posterior μ is defined on a space M, we try to find a diffeomorphism $f : N \to M$ (note the order here, to stay consistent with Normalizing Flow literature I define the transformation as a function *to* our posterior) such that the transformed posterior $f^*\mu$ is nice in some way. $f^*\mu$ refers to the pullback of the posterior (which we interpret as a volume form), ie we *pull it back* to the space N along the transformation f. We will show later how to do this in practice.

We can explicitly incorporate this transformation into HMC (or variations of it like NUTS):

```python
def hmc_proposal(rng, x, mu_density, F, step_size, n_leapfrog):
    # Compute the log density and score (differential of the log density)
    # at the initial point.
    logp_x, score_x = score_and_value(mu_density)(x)
    # Compute the log density and score in the transformed space.
    # We will see later how to do this.
    y, score_y, logp_y = f_pullback_score(x, score_x, logp_x)

    # Sample a new velocity for our trajectory.
    # In the transformed space we assume an identity mass matrix,
    # so we don't have to distinguish between momentum and velocity.
    velocity = rng.normal(size=len(x))

    # Several leapfrog steps
    for i in range(n_leapfrog):
        # velocity half-step
        velocity += step_size / 2 * score_y
        # position step
        y += step_size * velocity

        # Transform back and evaluate density
        x = f_transform(y)
        logp_x, score_x = score_and_value(mu_density)(x)
        y, score_y, logp_y = f_pullback_score(x, score_x, logp_x)

        # second velocity half-step
        velocity += step_size / 2 * score_y

    return x
```

If f is an affine transformation, this simplifies to the usual mass-matrix based HMC. For instance, if we choose $f(x) = \sigma \odot x$, we get the mass matrix $\operatorname{diag}(\sigma^{-2})$. This is explicitly spelled out in the Neal paper (Neal 2012).

HMC efficiency is famously dependent on the parametrization, so we know that this is much more or less efficient for some choices of f than for others. It is however not obvious what criterion we could use to decide how good a particular choice of f is, so that this choice can be automated. We need a loss function that maps the diffeomorphism to a measure of difficulty for HMC.

This hard to quantify in general, but we can notice that the efficiency of HMC largely depends on the trajectory, and this trajectory does not depend on the density directly, but only the scores. We also know that HMC is efficient if the posterior is a standard normal distribution. So a reasonable loss function can ask how different the scores on the transformed space are from the scores of a standard normal distribution. If they match well, we will use the same trajectory we would use for a standard normal distribution ω, which we know to be a good trajectory. And because the standard normal distribution is defined in terms of an inner product, we already have a well-defined norm on the scores that we can use to evaluate their difference. This directly motivates the following definition of the Fisher divergence.

2.3 Fisher divergence

Let (N, g) be a Riemannian manifold with probability volume forms ω_1 and ω_2. We can define a scalar function z on N by $\omega_2 = z\omega_1$, or equivalently we also write this as $z = \frac{\omega_2}{\omega_1}$.

We define the fisher divergence of ω_1 and ω_2 as

$$\mathcal{F}_g(\omega_1, \omega_2) = \int \|\nabla \log(z)\|_g^2 d\omega_1. \tag{2}$$

Note that $\mathcal{F}$ requires more structure on N than the KL-divergence $\int \log(z) d\omega_1$, as the norm depends on the metric tensor.

Given a second (non-Riemannian) manifold M with a probability volume form μ, and a diffeomorphism $f : N \to M$, we can define the divergence between μ and ω_1 by pulling back μ to X, ie $D_g(f^* \mu, \omega_1)$.

We can also compute this fisher divergence directly on X, by pulling back the metric tensor to X:

$$\mathcal{F}_g(F^* \mu, \omega_1) = \mathcal{F}_{(f^{-1})^* g}\left(\mu, \left(f^{-1}\right)^* \omega_1\right) \tag{3}$$

In our case, M is the space we originaly defined the posterior on, μ is the posterior, and ω_1 is a standard normal distribution.

2.4 Sobolev divergence

The fisher divergence uses the information in the scores of our posterior. It still ignores some additional information we have however: For each pair of points where we evaluate the posterior, we know the ratio of the densities at those points. If our transformed posterior has the density $p(x)$ and the standard normal distribution has density $q(x)$, then we want $\frac{p(x)}{p(y)} \approx \frac{q(x)}{q(y)}$. We look only at pairwise ratio densities, because we don't know the normalization factor of our posterior, and can't compute the ratios $\frac{p(x)}{q(x)}$ directly.

This leads us to the additional loss term

$$\begin{aligned}\int\int ((\log(p(x)) - \log(p(y))) - (\log(q(x)) - \log(q(y)))^2)p(x)p(y)dxdy \\ = \mathrm{Var}_{p(x)}[\log(p(x)) - \log(q(x))]\end{aligned} \tag{4}$$

We extend the Fisher divergence to a (as far as I know new) divergence, the Sobolev divergence by adding this second term:

$$\mathcal{S}_{g(\omega_1,\omega_2)} = \mathcal{F}_{g(\omega_1,\omega_2)} + \int \log(z)^2\omega_1 - \left(\int \log(z)\omega_1\right)^2 = \mathcal{F}_{g(\omega_1,\omega_2)} + \mathrm{Var}_{\omega_1}[\log(z)] \tag{5}$$

The name is due to the similarity to the Sobolev norm $\|f\|^2_{H^1} = \int |f|^2 + |\nabla f|^2$.

From a little bit of experimentation it seems that the Sobolev divergence can improve over just using the Fisher divergence, but I need to experiment more with this. The following will just use the Fisher divergence, and hopefully I can extend that to the Sobolev divergence soon.

2.5 Transformations of scores

In practice, we don't work with the measure μ directly, but with the densities relative to Lebesgue measures λ_N and λ_M, so $\mu = p\lambda_M$ and $\omega = q\lambda_N$. The change-of-variable tells us that $f^*\lambda_M = |\det(df)|\lambda_N$. This allows us to compute the transformed scores $d\log\left(f^*\frac{p}{\lambda_N}\right)$:

$$d\log\left(f^*\frac{\mu}{\lambda_N}\right) = d\log\left(f^*\frac{p}{\lambda_M}\frac{f^*\lambda_M}{\lambda_N}\right) = f^*d\log(p) + d\log|\det(df)| = \hat{f}^*(d\log(p), 1) \tag{6}$$

where $\hat{f} : N \to M \times \mathbb{R}, x \mapsto (f(x), \log|\det(f)|)$

This allows us to implement the score pullback function in autodiff systems, for instance in Jax:

```
def f_and_logdet(y):
    """Compute the transformation F and its log determininant jacobian."""
    ...

def f_inv(x):
    """Compute the inverse of F."""
    ...

def f_pullback_score(x, s_x, logp):
    """Compute the transformed position, score and logp."""
    y = F_inv(x)
    (_, log_det), pullback_fn = jax.vjp(f_and_logdet, y)
    s_y = pullback_fn(s_x, jnp.ones(()))
    return y, s_y, logp + log_det
```

We can further simplify the Fisher divergence if we set ω to a standard normal distribution, and use the standard euclidean inner product as the metric tensor:

$$\begin{aligned}\mathcal{F}(f^*\mu, \omega) &= \int \left\|\nabla \log \frac{f^*\mu}{\omega}\right\|^2_g f^*\mu \\ &= \int \left\|\hat{f}^*(d\log p, 1) - d\log(q)\right\|^2_{g^{-1}} f^*\mu \\ &= \int \left\|\hat{f}^*(d\log p_x, 1) + x\right\|^2 f^*\mu(x)\end{aligned} \tag{7}$$

Given posterior draws x_i and corresponding scores α_i in the original posterior space X we can approximate this expectation as

$$\hat{\mathcal{F}} = \frac{1}{N}\sum_i \left\| \hat{f}^*(s_i, 1) + f^{-1}(x_i) \right\|^2 \tag{8}$$

Or in code:

```
def log_loss(f_pullback_scores, draw_data):
    draws, scores, logp_vals = draw_data
    pullback = vectorize(F_pullback_scores)
    draws_y, scores_y, _ = pullback(draws, scores, logp_vals)
    return log((draws_y + scores).sum(0).mean())
```

Note: Some previous literature (todo ref) proposed to minimize $\mathbb{E}[\alpha_x^T \alpha_x]$, which is similar, but does not solve the issue of choosing a well-defined inner product. But finding a good inner product is the whole point of mass matrix adaptation. If we pull pack the inner product of the standard normal distribution to X and use the corresponding inner product on the dual space of 1-forms, we end up with an equivalent definition for the loss function defined above.

2.6 Specific choices for the diffeomorphism F

For particular families of diffeomorphisms F, we can get more specific results.

2.6.1 Diagonal mass matrix

If we choose $F_{\sigma,\mu} : Y \to X$ as $x \mapsto y \odot \sigma + \mu$, we get the same effect as diagonal mass matrix estimation.

In this case, the fisher divergence reduces to

$$\hat{D}_{\sigma,\mu} = \frac{1}{N}\sum_i \left\| \sigma \odot \alpha_i + \sigma^{-1} \odot (x_i - \mu) \right\|^2 \tag{9}$$

This is a special case of the affine transformation in Appendix A and minimal if $\sigma^2 = \mathrm{Var}[x_i]^{\frac{1}{2}} \mathrm{Var}[\alpha_i]^{-\frac{1}{2}}$ and $\mu = \bar{x}_i + \sigma^2 \bar{s}_i$. So we recover the same result we got in Section 2.1. It is very easy to compute (also using an online algorithm for the variance to avoid having to store the x_i and α_i values), and therefore the default in nutpie. In a Section 5 we will show some benchmarks to compare its performance.

Some theoretical results for normal posteriors

If the posterior is $N(\mu, \Sigma)$, then the minimizers $\hat{\mu}$ and $\hat{\sigma}$ of $\hat{\mathcal{F}}$ converge to μ and $\exp(\frac{1}{2} \log \mathrm{diag}(\Sigma) - \frac{1}{2} \log \mathrm{diag}(\Sigma^{-1})$ respectively. This is a direct consequence of $\mathrm{Cov}[x_i] \to \Sigma$ and $\mathrm{Cov}[\alpha_i] \to \Sigma^{-1}$.

$\hat{\mathcal{F}}$ converges to $\sum_i \lambda_i + \lambda_i^{-1}$, where λ_i are the generalized eigenvalues of Σ with respect to $\mathrm{diag}(\hat{\sigma}^2)$, so large and small eigenvalues are penalized. When we choose $\mathrm{diag}(\Sigma)$ as mass matrix, we effectively minimize $\sum_i \lambda_i$, and only penalize large eigenvalues. If we choose $\mathrm{diag}(\mathbb{E}(\alpha\alpha^T))$ (as proposed for instance in Tran and Kleppe (2024)) we effectively minimize $\sum \lambda_i^{-1}$ and only penalize small eigenvalues. But based on theoretical results for multivarite normal posteriors in Langmore et al. (2020), we know that both large and small eigenvalues make HMC less efficient.

We can use the result in (todo ref) to evaluate the different diagonal mass matrix choices on various gaussian posteriors, with different numbers of observations. Figure todo shows the resulting condition numbers of the posterior as seen by the sampler in the transformed space.

2.6.2 Full mass matrix

We choose $f_{A,\mu}(y) = Ay + \mu$. This corresponds to a mass matrix $M = \left(AA^T\right)^{-1}$. Because as we will see $\hat{\mathcal{F}}$ only depends on AA^T and μ, we can restrict A to be symmetric positive definite.

We get

$$\hat{\mathcal{F}}[f] = \frac{1}{N} \sum \left\| A^T s_i + A^{-1}(x_i - \mu) \right\|^2 \tag{10}$$

which is minimal if $AA^T \operatorname{Cov}[x_i] AA^T = \operatorname{Cov}[\alpha_i]$ (for a proof, see Appendix A), and as such corresponds again to our earlier derivation in Section 2.1. If the two covariance matrices are full rank, we get a unique minimum at the geometric mean of $\operatorname{Cov}[x_i]$ and $\operatorname{Cov}[s_i]$.

2.6.3 Diagonal plus low-rank

If the number of dimensions is larger than the number of draws, we can add regularization terms. And to avoid $O(n^3)$ computation costs, we can project draws and scores into the span of x_i and α_i, compute the regularized mean in this subspace and use the mass matrix .... If we only store the components, we can avoid $O(n^2)$ storage, and still do all operations we need for HMC quickly. To further reduce computational cost, we can ignore eigenvalues that are close to one.

todo

This is implemented in nutpie with

```
nutpie.sample(model, low_rank_modified_mass_matrix=True)
```

2.6.4 Model specific diffeomorphisms

Sometimes we can suggest useful families of diffeomorphism based on the model itself. A common reparametrization for instance is the non-centered parametrization, where we change a model from the centered parametrization

$$x \sim N(0, \sigma^2) \tag{11}$$

into the non-centered parametrization

$$\begin{aligned} z &\sim N(0, 1) \\ x &= z \odot \sigma \end{aligned} \tag{12}$$

We can generalize this for any choice of $0 \leq t \leq 1$ to (todo add ref, ask Aki)

$$\begin{aligned} z &\sim N(0, \sigma^{2t}) \\ x &= z \odot \sigma^{1-t} \end{aligned} \tag{13}$$

This suggests $f_{t(\sigma,z)} = (\sigma, \sigma^{1-t} z)$...

2.6.5 Normalizing flows

Normalizing flows provide a large family of diffeomorphisms that we can use to transform our posterior. We have rather strong requirements for the flows however: We need forward and inverse transformations, and we need to be able to compute the log determinant of the jacobian of the transformation efficiently. A well studied family of normalizing flows that provides all of those is RealNVP (todo ref). For our experiments we used the library flowjax that implements RealNVP and other normalizing flows in jax. I slightly changed the adaptation schema from the usual nutpie algorithm (described in more detail in the next section), so that for the first couple of windows only diagonal mass matrix adaptation is used, and only after 150 draws do we start to fit a normalizing flow. We then repeatedly run an adam optimizer on a window of draws, and use the updated normalizing flow to sample.

Especially for larger models the size of our training data set seems to be very important, so I included the full trajectory of the HMC sampler as training data, not just the draws themselves. I think this might be useful to do even if the amount of training data is not a limiting factor, as we would like that the transformed posterior matches the normal distribution everywhere the HMC sampler evaluates it, not just at those points it accepts as draws. So far I have not tested this systematically, but my experiments suggest that this can help us to sample a wide range of posterior distributions a lot more efficiently, or also to sample many distributions that were previously not possible to sample without extensive manual reparametrizations. It also comes with a large computational cost however, as the optimization itself can take a long time. Luckily, it seems to run relatively efficiently on GPUs, so that even if the logp function itself does not lend itself to evaluation on GPUs, we can still spend most of the computational time running scalable code. Often, the number of gradient evaluations that are necessary to sample even complicated posterior distributions decrease a lot.

With the current implementation I did not have too much luck with problems however, that have significantly more than around 1000 parameters. I am running into memory issues (I think those are probably an implementation issue and not a fundamental issue with the algorithm however) and the effectiveness of the normalizing flows, or our ability to fit them properly with the limited number of evaluation points seems to decrease.

I hope that this issue can be overcome by choosing normalizing flow layers in a way that takes model structure into account, or by adding layers one after the other, based on which parameters seem to not fit the normal distribution well yet.

code: https://github.com/pymc-devs/nutpie/pull/154

3 Adaptation schema to learn the diffeomorphism

Whether we adapt a mass matrix using the posterior variance as Stan does, or if we use any of the bijections based on the fisher divergence defined above, we always have the same problem: In order to generate posterior draws we need the mass matrix (or the bijection), but to estimate the mass-matrix/bijection we need posterior draws.

There is a well known way out of this conundrum: We start sampling with same initial transformation, and collect a number of draws. Based on those draws, we estimate a better transformation, and repeat. This adaptation-window approach has long been used in the major implementations of HMC, and remained largely unchanged for a number of years. PyMC, numpyro and blackjax all mostly use the same details as Stan, with at most minor modifications.

There are howerver I think a couple of minor changes that seem to improve the effiencey of this schma significantly. Except for the last section, where I discuss adaptation schemas for normalizing flows, I will assume that we adapt a mass matrix (or equivalently an affine bijection).

3.1 Choice of initial position

Stan draws initial points independently for each chain from the interval $(-1, 1)$ on the unconstrained space. I don't have any clear data to suggest so, but for some time PyMC has initialized using draws from the prior instead, and it seems to me that this tends to be more robust. This is of course only possible for model definition languages that allow prior sampling, and would for instance be difficult to implement in Stan.

3.2 Choice of initial diffeomorphism

Stan starts the first adaptation with an identity mass matrix. The very first HMC trajectories seem to be overall much more reasonable if we use $M = \mathrm{diag}(\alpha_0^T \alpha_0)$ instead. This also makes the initialization independent of variable scaling.

3.3 Accelerated window based adaptation

Stan and other sampler do not run vanilla HMC, but often NUTS, where the length of the trajectory is choosen automatically. This can make it very costly to generate draws if the mass matrix is not adapted well, because in those cases we often use a large number of HMC steps for each draw (in the 100s or typically up to 1000). Very early on during sampling we have a large intcentive to use available information obout the posterior as quickly as possible, to avoid these long trajectories. By default Stan starts adaptation with a step-size adaptation window, (50 draws todo check), where we do not change the mass matrix at all. It is then followed by a mass matrix adaptation window (100 draws? todo), where we generate draws for the next mass matrix update, but still use the initial mass matrix for sampling.

It is not uncommon that trajectories are very long during these first 150(todo) draws, and drop significantly after the first update. And because the trajectory lengths can easily vary by a factor of 10 or 100 between these phases, the draws before the first mass matrix change can take a sizable percentage of the total sampling time.

```python
class MassMatrixEstimator:
    def update(self, position, score):
        ...
    def current(self) -> MassMatrix:
        ...
    def num_points(self) -> int:
        ...

class StepSizeAdapt:
    def reset(self):
        ...
    def update(self, accept_statistic):
        ...
    def current_warmup(self) -> float:
        ...
    def current_best(self) -> float:
        ...
```

```python
def warmup(num_warmup, num_early, num_late, early_switch_freq,
late_switch_freq):
    position, score = draw_from_prior()
    foreground_window = MassMatrixEstimator()
    foreground_window.update(position, score)
    background_window = MassMatrixEstimator()
    step_size_estimator = StepSizeAdapt(position, score)
    first_mass_matrix = True

    for draw in range(num_warmup):
        is_early = draw < num_early
        is_late = num_warmup - draw < num_late

        mass_matrix = foreground_window.current()
        step_size = step_size_estimator.current_warmup()
        (
          accept_stat, accept_stat_sym, position, score,
          diverging, steps_from_init
        ) = hmc_step(mass_matrix, step_size, position, score)

        # Early on we ignore diverging draws that did not move
        # several steps. They probably just used a terrible step size
        ok = (not is_early) or (not diverging) or (steps_from_init > 4)
        if ok:
            foreground_window.update(position, score)
            background_window.update(position, score)

        if is_late:
            step_size_estimator.update(accept_stat_sym)
            continue

        step_size_estimator.update(accept_stat)

        switch_freq = early_switch_freq if is_early else late_switch_freq
        remaining = num_warmup - draw - num_late
        if (remaining > late_switch_freq
            and background_window.num_points() > switch_freq
        ):
            foreground_window = background_window
            background_window = MassMatrixEstimator()
            if first_mass_matrix:
                step_size_estimator.reset()
            first_mass_matrix = False
```

Constant step size adaptation with dual averaging. Overlapping windows, so that we can switch to a better estimate quickly.

Intuition: We want to update quickly, and not use an old mass matrix estimate at a point when we have more information and could already compute a better estimate. We could just use a tailing window and update in each step with the previous k draws. This is computationally inefficient (unless the logp function is very expensive), and can not easily be implemented as a streaming estimator (see below for more details). But if we use several overlapping estimation windows, we can compromise between optimal information usage and computational cost, and still use streaming mass matrix estimators.

Three phases:

- Initial phase with small window size to find the typical set. Discard duplicate draws.
- Main phase with longer windows
- Final phase with constant mass matrix: Only step size is adapted. Use a symmetric estimate of the acceptance statistic (todo ref stan discourse).

This scheme can be used with arbitrary mass matrix estimators. If the estimator allows a streaming (ref) implementation, we do not need to store the draws within each window.

4 Implementation in nutpie

todo

5 Experimental evaluation of nutpie

We run nutpie and cmdstan on posteriordb to compare performance in terms of effective sample size per gradient evaluation and in terms of effective sample size per time…

Code is here:

todo

Bibliography

[1] R. M. Neal, "MCMC using Hamiltonian dynamics." Accessed: Nov. 26, 2024. [Online]. Available: http://arxiv.org/abs/1206.1901

[2] J. H. Tran and T. S. Kleppe, "Tuning diagonal scale matrices for HMC." Accessed: Nov. 26, 2024. [Online]. Available: http://arxiv.org/abs/2403.07495

[3] I. Langmore, M. Dikovsky, S. Geraedts, P. Norgaard, and R. Von Behren, "A Condition Number for Hamiltonian Monte Carlo." Accessed: Oct. 16, 2022. [Online]. Available: http://arxiv.org/abs/1905.09813

A Minimize Fisher divergence for affine transformations

$D[F]$ for $F(y) = Ay + \mu$ is minimal if $\Sigma \operatorname{Cov}[\alpha]\Sigma = \operatorname{Cov}[x]$ and $\mu = \bar{x} + \Sigma\bar{\alpha}$, where $\Sigma = AA^T$:

We collect all α_i in the columns of G, and all x_i in the columns of X. Let e we the vector containing only ones. And let $\Sigma = AA^T$.

$$D = \frac{1}{N}\left\|A^T G + A^{-1}(X - \mu e^T)\right\|_F^2 \tag{14}$$

A-A Find $\hat{\mu}$

Take the differential with respect to only μ:

$$\begin{aligned} dD &= -\frac{2}{N}\operatorname{tr}\left[\left(A^TG + A^{-1}(X-\mu e^T)\right)^T A^{-1} d\mu e^T\right] \\ &= -\frac{2}{N}\operatorname{tr}\left[e^T\left(A^TG + A^{-1}(X-\mu e^T)\right)^T A^{-1} d\mu\right] \end{aligned} \tag{15}$$

At a minimum of D this has to be zero for all $d\mu$, which is the case iff

$$e^T\left(A^TG + A^{-1}(X-\mu e^T)\right)^T A^{-1} = 0 \tag{16}$$

It follows that

$$\mu = \bar{x} + \Sigma\bar{\alpha} \tag{17}$$

A-B Find $\hat{A}$

Take the differential of D with respect to A

$$dD = \frac{2}{N}\operatorname{tr}\left[\left(A^TG + A^{-1}(X-\mu e^T)\right)^T \left(dA^TG - A^{-1}dAA^{-1}(X-\mu e^T)\right)\right] \tag{18}$$

Using the result for μ we get $X - \mu e^T = \tilde{X} - \Sigma\bar{\alpha}e^T$. This, and using cyclic and transpose properties of the trace gives us

$$\begin{aligned} dD &= \frac{2}{N}\operatorname{tr}\left[\left(A^TG + A^{-1}(\tilde{X} - \Sigma\bar{\alpha}e^T)\right)G^T dA\right] \\ &+\frac{2}{N}\operatorname{tr}\left[A^{-1}(\Sigma\bar{\alpha}e^T - \tilde{X})\left(A^TG + A^{-1}(\tilde{X} - \Sigma\bar{\alpha}e^T)\right)^T A^{-1}dA\right] \end{aligned} \tag{19}$$

This is zero for all dA iff

$$\begin{aligned} 0 &= \left(A^TG + A^{-1}(\tilde{X} - \Sigma\bar{\alpha}e^T)\right)G^T + A^{-1}(\Sigma\bar{\alpha}e^T - \tilde{X})\left(A^TG + A^{-1}(\tilde{X} - \Sigma\bar{\alpha}e^T)\right)^T A^{-1} \\ &= A^T\tilde{G}G^T + A^{-1}\tilde{X}G^T + \left(A^T\bar{\alpha}e^T - A^{-1}\tilde{X}\right)\left(\tilde{G}^T + \tilde{X}^T\Sigma^{-1}\right), \end{aligned} \tag{20}$$

where $\tilde{X} = X - \bar{x}e^T$, the matrix with centered x_i in the columns, and $\tilde{G} = G - \bar{\alpha}e^T$. Because $e^T\tilde{X}^T = e^T\tilde{G}^T = 0$ we get

$$A^T\tilde{G}G^T + A^{-1}\tilde{X}G^T - A^{-1}\tilde{X}\tilde{G}^T - A^{-1}\tilde{X}\tilde{X}^T\Sigma^{-1} = 0 \tag{21}$$

Or

$$\begin{aligned} &\left(\tilde{G} + \Sigma^{-1}\tilde{X}\right)G^T - \Sigma^{-1}\tilde{X}\tilde{G}^T - \Sigma^{-1}\tilde{X}\tilde{X}^T\Sigma^{-1} \\ &= \tilde{G}G^T + \Sigma^{-1}\tilde{X}e\bar{\alpha}^T - \Sigma^{-1}\tilde{X}\tilde{X}^T\Sigma^{-1} \\ &= \tilde{G}\tilde{G}^T - \Sigma^{-1}\tilde{X}\tilde{X}^T\Sigma^{-1} \end{aligned} \tag{22}$$