Dynamic guessing for Hamiltonian Monte Carlo with embedded numerical root-finding

A new algorithm and Python package for a class of difficult Bayesian statistical models

AUTHORS
Anonymous

AFFILIATIONS
Anonymous

SUBMITTED
Not submitted yet.

Contents

Abstract

Introduction

Hamiltonian Monte Carlo

Embedded root-finding

Related work

Methods

Illustrative example

Performance benchmarks

Discussion and conclusions

Appendix 1: Table comparisons of our performance benchmarks

Appendix 2: pseudocode description of dynamic guessing

Appendix 3: implementation of guess-implicit

Appendix 4: software details

Appendix 5: benchmark models

Abstract

It is possible to fit Bayesian statistical models whose parameters satisfy analytically intractable algebraic conditions by embedding a differentiable numerical root-finder inside a gradient-based sampling algorithm like Hamiltonian Monte Carlo. This technique has enabled important scientific breakthroughs, but is limited by the high computational cost of computing and differentiating large numbers of numerical solutions. We show that dynamically varying the starting guess within a Hamiltonian trajectory can improve performance. To choose a good guess we propose two heuristics: *guess-previous* reuses the previous solution as the guess and *guess-implicit* extrapolates the previous solution using implicit differentiation. We benchmark these heuristics on a range of representative models. We also present a JAX-based Python package providing easy access to a performant sampler augmented with dynamic guessing.

Introduction

If a modeller knows that some quantities jointly satisfy algebraic constraints, they may want to embed a root-finding problem inside a Bayesian statistical model. For example, biochemical reaction networks are governed by partially-known kinetic parameters and often satisfy steady-state constraints. Typically the root-finding problem has no analytical solution, so that its solution must be approximated using numerical methods.

Statistical inference for this kind of model is possible using gradient-based Markov Chain Monte Carlo algorithms like Hamiltonian Monte Carlo and its variants, as the parameter gradients of root-finding problems can usually be found. Unfortunately, solving and differentiating root-finding problems in the course of gradient-based MCMC imposes a substantial computational overhead, limiting the range of models that can practically be fit.

In this paper we propose to address this problem by dynamically updating the root-finding algorithm's starting guess as the sampler moves along a simulated Hamiltonian trajectory. We propose heuristics for updating the guess and test these on a range of statistical models, showing that dynamic guessing improves performance compared with the state of the art. We also present a Python package `grapevine` containing our implementation of HMC with dynamic guessing, as well as benchmarks and convenience functions that allow users to easily fit their own statistical models using our algorithm.

Our Python package and the code used to perform the experiments reported in this paper are available at [ANONYMISED] and from the Python Package Index.

Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC) and its variants sample N parameter vectors $\theta_1, \ldots, \theta_N \in \mathbb{R}^k$ according to a target probability density function $\pi : \mathbb{R}^k \to [0, 1]$. If the algorithm works, then the statistical properties of the sample will approximately agree with the target distribution, i.e. $\sum_i \frac{1}{N} f(\theta_i) \approx E_\pi f(\theta)$ for any function f.

See [1] for a full introduction to HMC. The core strategy, shared with the older Metropolis-Hastings-Rosenberg algorithm (MHRMC), is to randomly perturb an initial parameter vector $\theta^\dagger$ to generate a proposal vector $\theta^\star$, then accept or reject the proposal randomly, with probability depending on the ratio $\frac{\pi(\theta^\star)}{\pi(\theta^\dagger)}$. Whereas MHRMC generates proposals according to a Gaussian random walk, so that $\theta^\star \sim N(\theta^\dagger, \Sigma)$, HMC constructs an auxiliary dynamical system obeying Hamiltonian dynamics, then generates a proposal by numerically simulating the trajectory resulting from randomly perturbing this system.

In more detail, the auxiliary dynamical system maps the parameter vector $\theta \in \mathbb{R}^k$ to a particle in k-dimensional space with potential energy $V(\theta) = -\ln \pi\theta$ and kinetic energy $K(\theta, \kappa) = -\ln \pi(\kappa \mid \theta)$ for auxiliary momentum vector $\kappa \in \mathbb{R}^k$. Perturbations are modelled by choosing an initial κ at random, then finding the trajectory where the Hamiltonian $H(\theta, \kappa) = K(\theta, \kappa) + V(\theta)$ is constant.

A Hamiltonian trajectory can be simulated numerically using a symplectic integrator such as velocity Verlet [2] [3]. These integrators linearise the trajectory into successively computed segments of given "step size" ϵ. To compute each segment the local potential energy $V(\theta)$ and its gradient $\nabla_\theta = \frac{\partial V}{\partial \theta}$ must be calculated at every step. Below when we refer to a "simulated Hamiltonian trajectory", we mean the series $\theta^\dagger, \theta^1, \ldots, \theta^n$ resulting from this process.

With carefully chosen hyperparameters, HMC can reliably generate a proposal vector $\theta^\star$ that is far away in parameter space from the initial parameter vector $\theta^\dagger$, yet has a non-negligible acceptance probability, leading to excellent performance [4]. This feature, together with the asymptotic exactness of Markov Chain Monte Carlo and the existence of well-maintained implementations, e.g. [5] [6] [7], make HMC a popular choice for applications requiring sampling of probability functions, such as Bayesian statistical modelling.

Embedded root-finding

We consider sampling by Hamiltonian Monte Carlo of a probability density function that embeds a root-finding problem, so that $\pi(\theta) = f(\theta, x)$ where x is a vector such that $g(x, \theta) = \bar{0}$. We assume that solving g for x requires a numerical solver.

To generate a proposal vector $\theta^\star$, HMC simulates a trajectory for the adjoint Hamiltonian system using a symplectic integrator, starting at the current parameter vector $\theta^\dagger$. At each step along the simulated trajectory, the numerical solution x is required to calculate the potential energy $V(\theta)$, and the parameter gradient $\frac{\partial x}{\partial \theta}$ is required to calculate the potential energy's gradient ∇_θ. Thus a differentiable numerical root-finding algorithm is needed.

Automatically differentiable numerical root-finders are available that interface with popular HMC implementations [8] [9], or are fully integrated [10]. These have enabled application of Bayesian statistical modelling with embedded numerical root-finding appears in a wide range of scientific contexts, including ignition chemistry [11], cell biology [12] [13] and optimal control [14].

See [15] [16] for discussion of previous implementations of HMC with embedded root-finding.

While many numerical root-finding algorithms exist, below we focus on the Newton-Raphson algorithm [17]. To find x satifying $g(x, \theta) = \bar{0}$ given θ, this algorithm iteratively updates a starting guess x_0 according to the rule

$$x_{i+1} = x_i - J_{x_i}^{-1} g(x_i, \theta)$$

where $J_{x_i} = \frac{\partial g(x_i, \theta)}{\partial x_i}$ is the jacobian of the target function with respect to x_i.

Once a solution x is found that satisfies $g(x, \theta) = 0$ to within a desired tolerance, its gradients with respect to θ can be found using the implicit function theorem:

$$\frac{\partial x}{\partial \theta} = -J_x^{-1} J_\theta$$

Assuming that calculating and inverting the jacobian term $J_{x_i}^{-1}$ is approximately equally costly throughout, the computational cost of using the Newton-Raphson algorithm to solve an embedded root-finding problem for a single simulated segment of a Hamiltonian trajectory is approximately proportional to the total number of Newton steps required. Like other numerical root-finding algorithms, this cost is highly sensitive to the starting guess x_0: if x_0 is in the neighbourhood of the solution then the algorithm converges quadratically, whereas a poor starting guess can prevent convergence altogether. See [18] for a general discussion of the effect of the choice of starting guess on numerical root-finding performance.

A natural way to speed up HMC with embedded root-finding is to find the best possible guess for each problem. The main recommendation of the Stan user's guide [10] is to hard-code a guess that is reasonable, given the likely values of the parameters. However, previous implementations of HMC with embedded root-finding require the starting guess to be the same for all problems on the same simulated Hamiltonian trajectory. Since the sampling performance of HMC is tied to its ability to propose a parameter vector $\theta^\star$ that is distant in parameter space from the current vector $\theta^\dagger$, this limitation is problematic. The simulated trajectory from $\theta^\dagger$ to a distant $\theta^\star$ will include intermediate parameter vectors θ^a and θ^b that are also distant. The solution to problem a, i.e. $g(x, \theta^a) = \bar{0}$, will typically be different from that of problem b, $g(x, \theta^b) = \bar{0}$. Therefore a starting guess that is optimal for problem a will be sub-optimal for problem b.

Related work

Our approach draws on previous work on numerical continuation and warm starting. Continuation refers to methods that use one numerical solution to generate an initial guess for a perturbed problem, which can then be solved more precisely using another method. In particular, our *guess-implicit* heuristic is a case of the Euler-

Newton predictor-corrector method investigated by Allgower and Georg [19]. Chapter 5 of this work proves that a Euler-Newton predictor-corrector method converges to the correct solution path of a continuously varying root-finding problem under sufficiently small linear perturbations.

In model predictive control it is often useful to "warm start" a numerical solver using a solution from a previous time step [20]. These ideas have recently been combined in the context of bilevel optimisation of neural network hyperparameters [21]. Sambharya et al [22] propose using neural networks to learn the relationship between parameter values and optimal warm starts of numerical fixed-point optimisers and compare this learned warm start approach with a "nearest-neighbour" warm start that is similar to our method. The learned warm start performs better for problems where the problem to be solved is far away from its nearest neighbour, so that the neighbour's parameter and root combination and their gradients are uninformative. In contrast, embedded root-finding problems on adjacent steps of a simulated Hamiltonian trajectory are usually close, and very many warm starts are required for a full MCMC run, motivating our consideration of cheaper linear warm starting procedures.

Methods

Consider the simulated Hamiltonian trajectory $\theta^1, \ldots, \theta^n$, where each θ^i embeds a root-finding problem $g(x^i, \theta^i) = \bar{0}$. The state of the art is to use the same starting guess x_0^{default} to numerically solve each embedded root-finding problem, we propose choosing the starting guess dynamically. Instead, we propose using a default value for the first starting guess x_0^1, and choosing subsequent starting guesses x_0^i for $i > 1$ using a heuristic function h, i.e. $x_0^i = h(\text{info}^{i-1})$ where info^{i-1} is some information available to the sampler at trajectory step $i - 1$. This strategy includes the state of the art approach as a trivial heuristic:

$$\text{guess-static}(\emptyset) = x_0^{\text{default}}$$

We call our approach the "grapevine method" after the expression "I heard it through the grapevine" and the visual resemblance of a simulated Hamiltonian trajectory to a vine, with the numerical solution at each integrator step representing grapes.

We implemented this idea by augmenting the velocity Verlet integrator with a dynamic state variable containing information that will be used to calculate an initial guess for the root-finding algorithm when the integrator updates its position. This variable is initialised at a default value, which is used to solve the numerical problem at the first step of any trajectory, and then modified at each position update. See appendix 2 below for a pseudocode description of our algorithm.

Our method does not affect the theoretical validity of MCMC sampling, as it does not change the target probability density function. If all embedded root-finding problems have single roots, the log probability density at any point in parameter space will be the same for a sampler using dynamic guessing as for a static one: the guessing heuristic does not affect what solution the root-finder will find, but only how quickly it converges. If an embedded problem has multiple roots, then the root that the solver finds may depend on the guess, so that the target density is not constant. Users must therefore verify that all embedded root-finding problems have single roots.

We tested two non-trivial guessing heuristics: *guess-previous* and *guess-implicit*. The information for the heuristic *guess-previous* is the solution x^{i-1} of the previous root-finding problem. The heuristic is simply to use the previous solution as the next guess:

$$\text{guess-previous}(x^{i-1}) = x^{i-1}$$

The information for the heuristic *guess-implicit* is the solution x^{i-1} of the previous root-finding problem, the current parameter vector θ^i and the previous parameter vector θ^{i-1}. The heuristic is to use implicit differentiation to find the local derivative of the previous solution with respect to the previous parameters, then perturb the previous solution by the product of this derivative and the change in parameter values:

$$\text{guess-implicit}(x^{i-1}, \theta^i, \theta^{i-1}) = x^{i-1} - \frac{\partial x^{i-1}}{\partial \theta^{i-1}}(\theta^i - \theta^{i-1})$$

See Appendix 3 for details of how we implemented *guess-implicit*.

A limitation of *guess-previous* and *guess-implicit* is that they rely on a smooth relationship between the parameter vector θ and the embedded root-finding solution vector x, so that the solution of the root-finding problem $g(x^i, \theta^i) = \bar{0}$, is informative as to the solution of the problem at the next leapfrog step, i.e. $g(x^{i+1}, \theta^{i+1}) = 0$.

In the context of Hamiltonian Monte Carlo we hypothesise that this kind of smoothness is likely to obtain. Stable HMC sampling requires the target density to be sufficiently smooth for a leapfrog integrator's discretisation error to remain bounded [23]. If the integrator's step size is too large, or the target density is locally non-smooth, the leapfrog trajectory diverges from the true Hamiltonian flow. Such divergences are a key diagnostic tool: see [1]. Most implementations of HMC detect divergences and tune the integrator specifically to avoid them, for example by adjusting the step size, applying geometric transformations or raising diagnostic errors. This process naturally restricts the sampler to regimes where $\pi(\theta)$ behaves smoothly.

In problems with embedded root-finding, $\pi(\theta)$ is invariably coupled with the root x of the embedded root-finding problem $g(x, \theta) = \bar{0}$. If $\pi(\theta)$ were insensitive to x, there would be little reason to calculate x at each leapfrog step. Consequently, we hypothesise, the required smoothness in $\pi(\theta)$ will lead to a correspondingly smooth relationship between the parameter vector θ and the roots x. This argument is difficult to make rigorous due to the complexity of non-linear root-finding so we relied on empirical tests to see whether it tends to hold in practice.

Illustrative example

To understand the motivation behind our method, consider the illustrative example shown below in figure 1. The example shows the solutions to a series of root-finding problems embedded on a Hamiltonian trajectory extracted from a representative model run, as well as the path through solution space from starting guess to solution taken by Newton solvers using different guessing heuristics. In this example solving the 11 embedded problems with static guessing required 45 total Newton steps, whereas with *guess-implicit* only 14 Newton steps were needed.

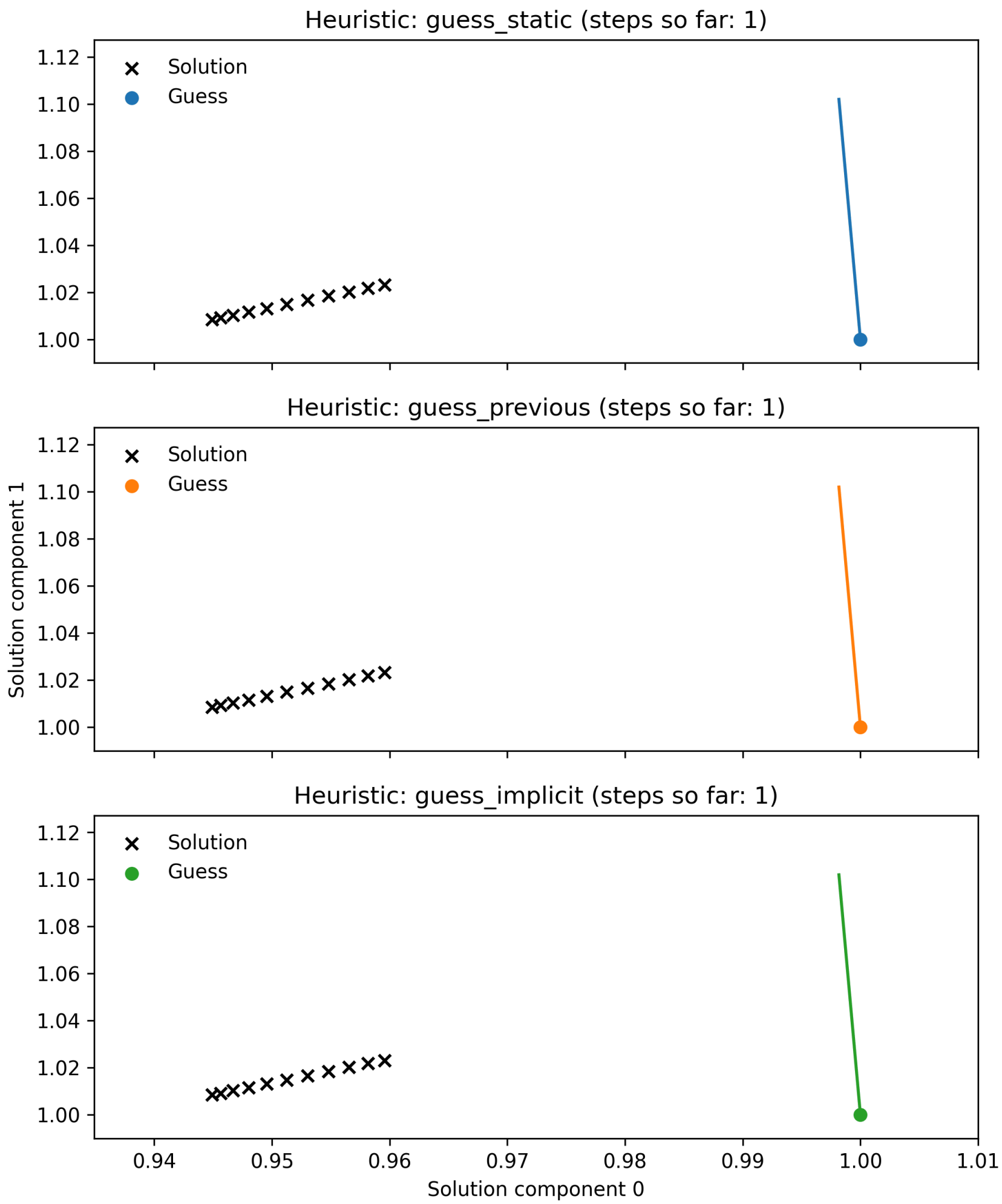


Figure 1: Illustrative example This figure compares the behaviour of dynamic and static heuristics along a single Hamiltonian trajectory. Black crosses show the solutions of 11 root-finding problems (finding the minimum of a parametrised 2-dimensional Rosenbrock function) corresponding to points along a simulated Hamiltonian trajectory through parameter space. Coloured lines show the paths through solution space taken by a Newton solver to approximately solve each problem. Note that some of these paths are very small. Coloured dots indicate the guess used for the current problem. For the first problem in the trajectory all heuristics use the default guess at coordinate (1, 1); for subsequent problems the *guess-static* continues to use the default guess whereas the dynamic heuristics use better guesses that lie closer to the target, thus saving Newton steps. The heuristic *guess-implicit* saves additional steps by exploiting gradient information.

Performance benchmarks

Our illustrative example is encouraging, as it is roughly representative of embedded root-finding in practice, but it is not sufficient to demonstrate our method's efficacy. The example only shows a single trajectory, whereas a typical MCMC run may require thousands of trajectories, with root-finding performance varying from one trajectory to another. The example also only considers one root-finding problem, whereas a good heuristic should perform well across a range of problems. For a more

comprehensive performance comparison, we compared the performance of dynamic and static guessing for a selection of eleven statistical models with different embedded root-finding problems. These models, described in detail in Appendix 5, fall into three categories.
First, there were seven simple statistical models embedding standard problems used to test optimisation algorithms, reformulated as parametrised root-finding problems: "Easom", "Beale", "Rastrigin (3d)", "Rosenbrock (3d)", "Rosenbrock (8d)", "Styblinski Tang (3d)" and "Levy (3d)". These models aimed to test the hypothesis that dynamic guessing would solver performance and robustness on especially difficult embedded problems. We chose the specific embedded problems because they have a range of different difficult features for numerical solvers, vary in dimensions, have global minima so that the associated root-finding problems are well-posed and are straightforward to implement.

Second, we tested two steady state metabolic network models, one relatively small ("Linear network") and one large ("Methionine cycle"). This kind of model is common in many fields, especially biochemistry: see for example [24] [25] [12]. These models therefore served to test whether dynamic guessing would yield practically significant benefits in real applications

Third, to test our hypothesis that HMC adaptation would tend to induce a smooth relationship between the solution vector x and the parameter vector θ, we tested two models embedding the root-finding problem $g(x, \theta) = x^3 - x \odot \sin(k\theta) \odot \cos(k\theta)$, with $\odot$ representing element-wise multiplication and scalar hyper-parameter k set to the very high value $1e8$. The solution to this problem depends non-smoothly on θ due to the oscillations induced by the trigonometric functions. In one model, "Adversarial Dependent", the target density $\pi(x, \theta)$ depends on the solution x via a Gaussian likelihood function, as would be seen in a typical application. The other model "Adversarial Independent", was exactly the same, but with π made independent of x by setting the likelihood to be constant. We expected that, if our hypothesis were correct, then dynamic guessing would outperform static guessing on the dependent model but not on the independent model.

For each model, we randomly generated 20 parametrisations based on the prior distribution. For each parametrisation we randomly simulated a measurement set consistently with the model's likelihood function, resulting in random draws from the models' prior predictive distributions. We then generated samples from each measurement set's posterior distribution using the No-U-Turn sampler, using the Stan adaptation algorithm provided by blackjax to tune the sampler. We compared the performance of the trivial heuristic *guess-static* with our proposed heuristics *guess-previous* and *guess-implicit*. Sampler configurations and solver tolerances were set per model, and so were the same for all heuristics. In addition, for each measurement set, we initialised each heuristic's sampler at the same point and with the same random seed, so that they explored the same path through parameter space. Appendix 4 below describes our software implementation.

We quantified performance using the total wall time required to complete the whole MCMC run, including warmup and sampling phases, and by counting the total number of post-warmup Newton steps taken in each run. The wall time metric gives a practical indication of the benefit of our approach, though at the cost of generalisability beyond our software implementation and hardware setup. The total number of post-warmup Newton steps provides an implementation-independent performance metric. Note that, as noted above, the number of Newton steps only approximately quantifies the computational cost of solving a root finding problem as the cost per step can vary depending on the parameter vector θ and the previous state x_{i-1}.

We diagnosed sampling by verifying that the effective sample size [5] was not small compared with the total number of MCMC samples, that there were no post-warmup divergent transitions and that the embedded solver never failed to converge.

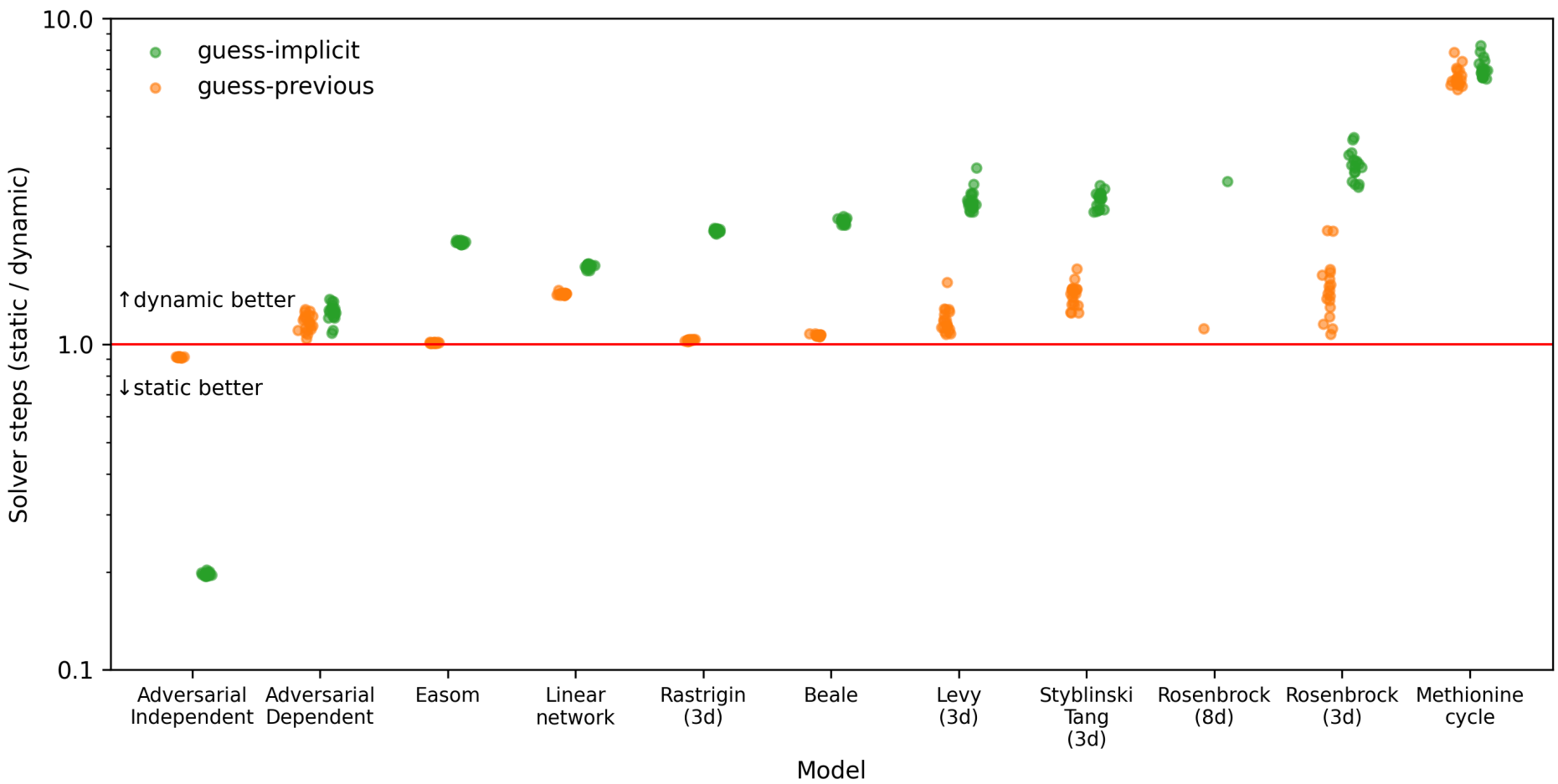


Figure 2: Newton step performance comparison The relative performance of the dynamic guessing methods compared with *guess-static*. Each dot shows the total number of Newton steps taken by an MCMC sampler to generate 500 post-warmup samples with the *guess-static* heuristic, divided by the number of steps taken to generate the same post-warmup samples using dynamic guessing. 20 random parametrisations of each model were tested.

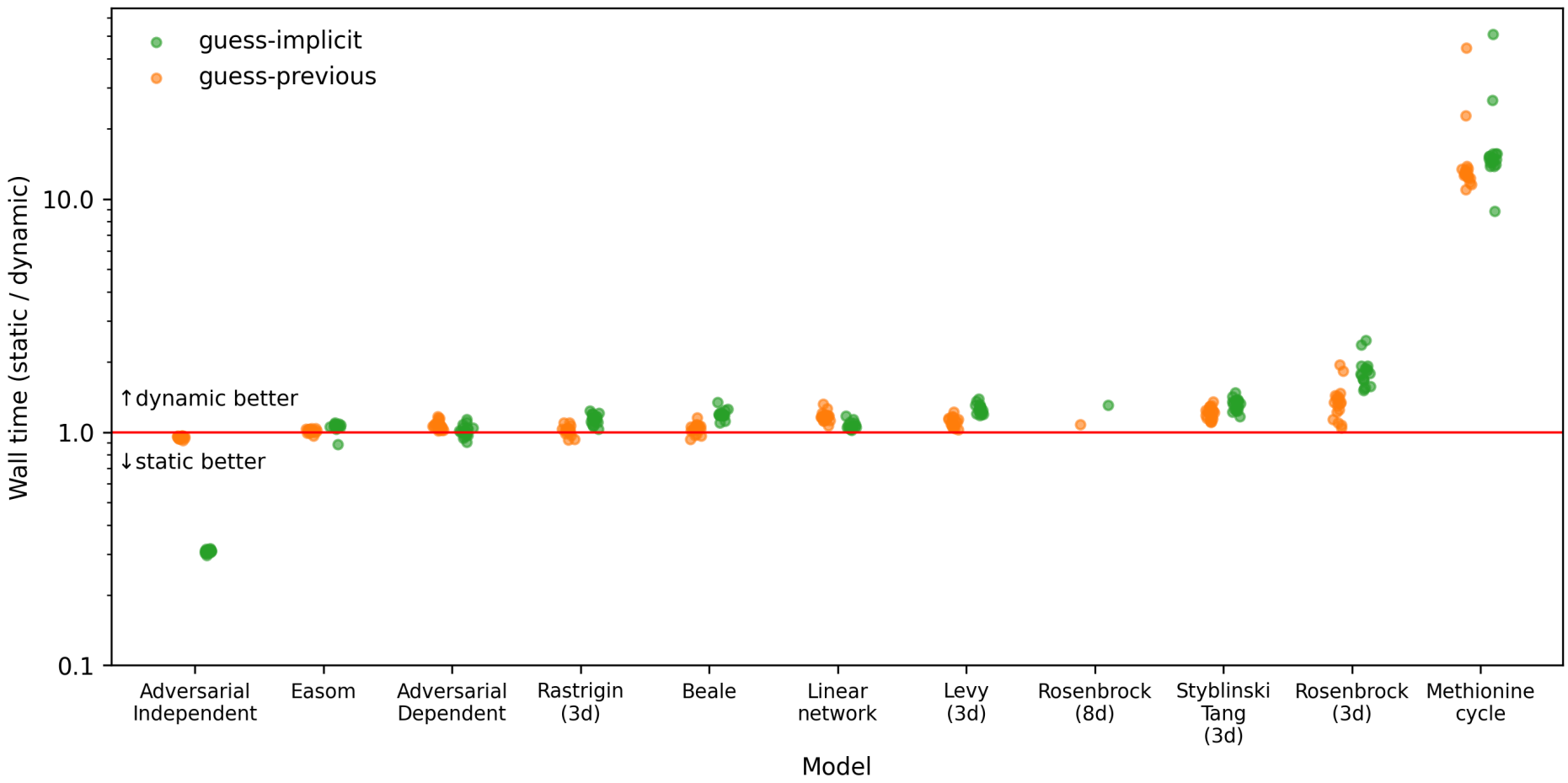


Figure 3: Wall time performance comparison The relative wall-time performance of the dynamic guessing methods compared with *guess-static*. Each dot shows the time required to generate 500 samples by a sampler using *guess-static*, divided by the time taken to generate the same post-warmup samples using dynamic guessing. 20 random parametrisations of each model were tested. Each run was performed in duplicate without recomilation of code, with only the time to complete the second run recorded, so that the recorded results exclude compilation times.

The results of our benchmarks are tabulated below in supplementary tables S1 and S2.

The number of failed MCMC runs for each heuristic and model was as follows:

model	*guess-static*	*guess-previous*	*guess-implicit*
Rosenbrock (8d)	19	2	2
Rosenbrock (3d)	2	1	1
Methionine cycle	1	0	0
Linear network	0	0	1
Easom	1	0	0
Levy (3d)	0	0	0
Beale	0	0	0
Adversarial Dependent	0	0	0
Rastrigin (3d)	0	0	0
Adversarial Independent	0	0	0
Styblinski Tang (3d)	0	0	0

Table 1: Failure counts The number of failed MCMC runs out of 20 for each heuristic and model. A run was counted as failed if there was a numerical solver failure during warmup or sampling.

Dynamic guessing reduced the number of Newton steps required to generate samples in all models except “Adversarial Independent” (Figure 2). The use of gradient information via the heuristic *guess-implicit* showed a greater reduction than *guess-previous* for every model except “Adversarial Independent”but performed worse than *guess-previous* for “Adversarial Independent”. The benefit of dynamic guessing was especially marked for the complex metabolic network model “Methionine cycle”, with performance improved by a factor of approximately 6 (*guess-previous*) to 7 (*guess-implicit*). The computational overhead from using the implicit gradient method meant that the wall time advantage was only meaningful for the “Methionine cycle” model (Figure 3). The dynamic methods also reduced the number of failed MCMC runs (Table 1). Interestingly, *guess-implicit* performed worse in wall time than *guess-previous* on the “Linear network” and “Adversarial Dependent” models, despite using fewer Newton steps.

All of our experiments were performed on a MacBook Pro 2024 with Apple M4 Pro processor and 48GB RAM, running macOS 15.7.2. Instructions for reproducing our benchmarks can be found in our code repository in the file `readme.md`.

Discussion and conclusions

The dynamic algorithms’ lower failure rate is likely because, for these algorithms, the sampler diverged less often at the start of the warmup phase of sampling, when the sampler must simulate trajectories that traverse low-probability regions of parameter space. Such trajectories are especially unfavourable for static guessing because they have root-finding problem solutions that are far away from any reasonable global guess.

Based on our results, we expect that replacing a static guessing algorithm with dynamic guessing will typically improve sampling performance for MCMC with embedded root-finding, making it possible to fit previously infeasible statistical models. The performance improvements on models embedding optimisation test functions show that dynamic guessing can improve MCMC performance even when the embedded problem is challenging for the solver. The improvements on the steady state metabolic network models show that this performance benefit extends to realistic cases, and can lead to a practically significant improvement in wall time.

As we expected, dynamic guessing was unhelpful in the "Adversarial Independent" model, which embedded a root-finding problem whose solution varied non-smoothly with the parameter vector, and where there was no coupling of the root-finding problem with the target probability density. In this case *guess-previous* showed similar performance as *guess-static*, whereas *guess-implicit* performed worse due to the local parameter gradient of the solution tending to guide the solver away from the next solution. However, when we coupled the root-finding problem with the target probability density in the "Adversarial Dependent" model, we again saw improved performance from dynamic compared with static guessing. We conclude that HMC adaptation algorithms are able to adjust appropriately for even highly non-smooth parameter-root relationships, provided that the roots are coupled with the target density function via a likelihood function.

In our experiments *guess-implicit* consistently saved Newton steps compared with *guess-previous*, except in the non-representative "Adversarial Independent" case. This improvement in theoretical performance mostly translated to improved wall-time on our hardware and with our software implementation, except in the "Linear network" and "Adversarial Dependent cases", where the *guess-previous* sampler used slightly less time despite performing additional Newton steps. This is likely due to the additional computational overhead imposed by calculating and caching $\frac{\partial x^i}{\partial \theta^i}$ at each leapfrog step. While our implementation calculates this quantity efficiently (see appendices 3 and 4 below for details), re-calculating it at all wastes work as $\frac{\partial x^i}{\partial \theta^i}$ is already needed to obtain the gradient of the potential energy (see section on embedded root-finding above). It is therefore likely possible to reduce the cost of *guess-implicit* by more closely integrating it with the leapfrog integrator. In the meantime, we recommend using *guess-implicit* in preference to *guess-previous* because it will tend to perform better in the worst-case scenario where the cost per Newton step is very high.

Our approach is not strictly limited to Hamiltonian Monte Carlo, and would also work for MCMC algorithms such as Metropolis-Hastings-Rosenberg or Langevin Monte Carlo [26] that do not generate proposals by simulating continuous trajectories of an adjoint dynamical system. However, we expect that our approach would only yield limited benefits for these algorithms. First, algorithms that do not require leapfrog integration, and where the previous proposal tends to be close to the current proposal, can implement warm starting far more simply. Second, whereas leapfrog integration within HMC yields a series of smoothly varying target densities and, we argue above, a smooth parameter-root relationship, this is not the case for other algorithms. As a result, non-linear warm starting procedures such as the one proposed in [22] may work better for non-HMC algorithms than our heuristics.

An opportunity for further performance improvement would be to use a different method to solve the first root-finding problem in a trajectory than for later problems. Plausibly, a slow but robust solver could be preferable for the first problem, which uses a default guess, whereas a faster but more fragile solver might be preferable for later problems where a potentially better guess is available.

There are several ways in which our approach can fail, beyond those explored in our experiments. For embedded problems with multiple roots, the previous solution may be a bad guess for the next solution. If the jacobian J_x is singular or near-singular then our conjugate gradient based method for finding $\frac{\partial x}{\partial \theta}$ will fail. If the target probability distribution has varying characteristic length scale, so that a too-large leapfrog integrator step size is chosen, then *guess-implicit* may suggest a suboptimal starting guess. These failure modes are shared with non-linear root-finding and Hamiltonian Monte Carlo in general, but dynamic guessing may exacerbate them.

Additionally, our software implementation must be used with care, as it requires a posterior log density function where the guess variable is only used by the numerical solver, and does not otherwise affect the output. We do not provide an automatic safeguard preventing the user from breaking this requirement, even though doing so risks producing invalid MCMC inference. It may therefore be beneficial to implement dynamic guessing using a stricter interface that makes inappropriate use of the guess variable impossible.

Appendix 1: Table comparisons of our performance benchmarks

The tables below show the mean, minimum and maximum values of both performance metrics in our experiments, supplementing the visual comparisons in figures 2 and 3 above.

model	*guess-static*	*guess-previous*	*guess-implicit*
Linear network	9718 (↓9111, ↑ 9992)	6802 (↓6441, ↑ 6976)	**5589** (↓5417, ↑ 5749)
Adversarial Dependent	10935 (↓9525, ↑ 12365)	9386 (↓7895, ↑ 10862)	**8741** (↓7381, ↑ 9846)
Rosenbrock (3d)	24605 (↓15352, ↑ 45383)	16336 (↓14063, ↑ 21025)	**6993** (↓5056, ↑ 10865)
Rastrigin (3d)	8393 (↓7854, ↑ 8920)	8162 (↓7660, ↑ 8661)	**3756** (↓3590, ↑ 3921)
Easom	6527 (↓6216, ↑ 6875)	6439 (↓6113, ↑ 6810)	**3170** (↓3057, ↑ 3291)
Adversarial Independent	**8765** (↓7895, ↑ 9515)	9594 (↓8608, ↑ 10442)	44509 (↓38980, ↑ 49076)
Methionine cycle	2826011 (↓2502717, ↑ 3416417)	427899 (↓409979, ↑ 464347)	**402295** (↓377295, ↑ 429837)
Beale	8875 (↓8225, ↑ 9670)	8329 (↓7755, ↑ 9053)	**3717** (↓3541, ↑ 3919)
Styblinski Tang (3d)	12742 (↓10557, ↑ 15834)	8959 (↓8038, ↑ 9517)	**4556** (↓4097, ↑ 5147)
Levy (3d)	12450 (↓10387, ↑ 17051)	10460 (↓9557, ↑ 12248)	**4479** (↓4072, ↑ 5066)
Rosenbrock (8d)	18202 (↓18202, ↑ 18202)	15868 (↓15210, ↑ 16975)	**5691** (↓5504, ↑ 5949)

Table S1: Newton step performance comparisonAverage, minimum (↓) and maximum (↑) Newton step counts for each heuristic and model. Failed runs were not included.

model	*guess-static*	*guess-previous*	*guess-implicit*
Rosenbrock (3d)	0.55 (↓0.41, ↑0.87)	0.40 (↓ 0.37, ↑ 0.45)	**0.30** (↓ 0.27, ↑ 0.36)
Easom	0.25 (↓0.23, ↑0.26)	0.24 (↓ 0.23, ↑ 0.28)	**0.23** (↓ 0.22, ↑ 0.26)
Beale	0.29 (↓0.27, ↑0.31)	0.28 (↓ 0.26, ↑ 0.31)	**0.24** (↓ 0.23, ↑ 0.26)

model	*guess-static*	*guess-previous*	*guess-implicit*
Styblinski Tang (3d)	0.36 (↓0.31, ↑0.42)	0.29 (↓ 0.28, ↑ 0.32)	**0.27** (↓ 0.25, ↑ 0.30)
Adversarial Independent	**0.04** (↓0.04, ↑0.05)	0.04 (↓ 0.04, ↑ 0.05)	0.14 (↓ 0.13, ↑ 0.15)
Methionine cycle	1218.53 (↓931.40, ↑3424.77)	82.88 (↓ 74.37, ↑ 124.10)	**74.16** (↓ 63.87, ↑ 159.66)
Linear network	0.37 (↓0.35, ↑0.40)	**0.31** (↓ 0.30, ↑ 0.34)	0.34 (↓ 0.33, ↑ 0.36)
Rosenbrock (8d)	0.57 (↓0.57, ↑0.57)	0.54 (↓ 0.52, ↑ 0.60)	**0.44** (↓ 0.41, ↑ 0.50)
Adversarial Dependent	0.07 (↓0.05, ↑0.07)	**0.06** (↓ 0.05, ↑ 0.07)	0.07 (↓ 0.05, ↑ 0.08)
Levy (3d)	0.39 (↓0.36, ↑0.46)	0.35 (↓ 0.34, ↑ 0.43)	**0.31** (↓ 0.29, ↑ 0.37)
Rastrigin (3d)	0.31 (↓0.30, ↑0.33)	0.30 (↓ 0.29, ↑ 0.33)	**0.27** (↓ 0.26, ↑ 0.29)

Table S2: Wall time performance comparison Average, minimum (↓) and maximum (↑) time in seconds to generate 500 post-warmup samples, for each heuristic and model. Failed runs were not included. The time includes both warmup and sampling phases. Each run was performed in duplicate without recomilation of code, with the time of the second run recorded, so that the recorded results exclude compilation times.

Appendix 2: pseudocode description of dynamic guessing

We assume we have:

- a function Solve that takes in a guess and returns an approximate solution to the target root-finding problem
- a scalar-valued function $\mathrm{LogProb}$ that takes in a set of parameter values and a root-finding solution
- a function $\mathrm{GetInfo}$ that takes in a set of parameter values and a root-finding solution, returning information for a heuristic.
- a function $\mathrm{Heuristic}$ that takes in the output of $\mathrm{GetInfo}$ and returns a guess
- initial parameters θ_0
- initial momentum κ_0
- default information $\mathrm{info}_{default}$
- step size ϵ

The aim is to initialise and then simulate a Hamiltonian trajectory using Leapfrog integration, while passing root-finding information appropriately. For this we use the functions $\mathrm{Heuristic}$, $\mathrm{LogDensityAndInfo}$, $\mathrm{PotentialGradientAndInfo}$, $\mathrm{InitialiseTrajectory}$ and $\mathrm{LeapfrogStep}$:

Function 1: Generate a new guess ($\mathrm{Heuristic}$)

- **Input:**
  - Parameters θ
  - Information from previous step, info
- **Output:** A guess for the root-finding problem, x_0

For example, for the heuristic *guess-previous*, info is the solution from previous step, if available, or a dummy value indicating that this is the first problem in the trajectory. The output is the previous solution, if available, or else a default value.

Algorithm 2: Update log density and information ($\mathrm{LogDensityAndInfo}$)

- **Input:**
  - Parameters θ
  - Information from previous step, info

1. $x_0 \leftarrow \mathrm{Heuristic}(\theta, \mathrm{info})$
2. $x \leftarrow \mathrm{Solve}(x_0)$
3. $\mathrm{lp} \leftarrow \mathrm{LogProb}(\theta, x)$
4. $\mathrm{info}_+ \leftarrow \mathrm{GetInfo}(\theta, x)$
5. Return $\mathrm{lp}, \mathrm{info}_+$

Algorithm 3: Evaluate potential energy and gradient ($\mathrm{PotentialGradientAndInfo}$)

- **Input:** Parameters θ, information from previous step, info

1. $lp, \mathrm{info}_+ \leftarrow \mathrm{LogDensityAndInfo}(\theta, \mathrm{info})$
2. $V \leftarrow -lp$ (Potential energy is the negative log density)
3. $\nabla_\theta V \leftarrow -\nabla_\theta lp$ (Compute the gradient of the potential energy)
4. Return $V, \nabla_\theta V$ and info_+

Algorithm 4: Initialise Trajectory ($\mathrm{InitialiseTrajectory}$)

1. $V_0, \nabla_\theta V_0, \text{info} \leftarrow \text{PotentialGradientAndInfo}(\theta_0, \text{info}_{default})$

2. Return $\theta_0, \kappa_0, V_0, \nabla_\theta V_0, \text{info}$

Algorithm 5: Leapfrog Integration Step (LeapfrogStep)

- **Input:**
  - Current state $\theta, \kappa, V, \nabla_\theta V, \text{info}$
  - step size ϵ

1. $\kappa_{\text{mo}} \leftarrow \kappa - \frac{\epsilon}{2}\nabla_\theta V$ (Update momentum, first half-step)

2. $\theta_+ \leftarrow \theta + \epsilon\kappa_{\text{mo}}$ (Update parameters)

3. $V_{+}, \nabla_{\theta} V_{+}, \text{info}{+} \leftarrow \text{PotentialGradientAndInfo}(\theta{+}, \text{info})$

4. $\kappa_+ \leftarrow \kappa_{\text{mo}} - \frac{\epsilon}{2}\nabla_\theta V_+$ (Update momentum, second half-step)

5. Return $\theta_+, \kappa_+, V_+, \nabla_\theta V_+, \text{info}_+$

The functions $\text{InitialiseTrajectory}$ and LeapfrogStep are drop-in substitutes for their counterparts in standard Hamiltonian Monte Carlo.

Appendix 3: implementation of *guess-implicit*

The *guess-implicit* heuristic is defined as follows, given previous solution x^{i-1}, previous parameters θ^{i-1} and current parameters θ^i:

$$\text{guess-implicit}(x^{i-1}, \theta^{i-1}, \theta^i) = x^{i-1} + \frac{dx}{d\theta}\Delta\theta$$

where $\Delta\theta = (\theta^i - \theta^{i-1})$.

To obtain $\frac{dx}{d\theta}$, we use the following consequence of the implicit function theorem [27]:

$$\frac{\partial x}{\partial \theta} = -(\text{jac}_x\, g(x^{i-1}, \theta^{i-1}))^{-1}\, \text{jac}_\theta\, g(x^{i-1}, \theta^{i-1})$$

In this expression the term

$$\text{jac}_x\, g(x^{i-1}, \theta^{i-1})$$

abbreviated below to J_x, indicates the jacobian with respect to x of $g(x^{i-1}, \theta^{i-1})$. Similarly

$$\text{jac}_\theta\, g(x^{i-1}, \theta^{i-1}) = J_\theta$$

is the jacobian with respect to θ of $g(x^{i-1}, \theta^{i-1})$.

Substituting terms we then have

$$\text{guess-implicit}(x^{i-1}, \theta^{i-1}, \theta^i) = x^{i-1} - J_x^{-1} J_\theta \Delta\theta$$

The *guess-implicit* heuristic can be implemented using the following Python function:

```
import jax
from jax import numpy as jnp

def guess_implicit(guess_info, params, f):
    "Guess the next solution using the implicit function theorem."
    old_x, old_p, *_ = guess_info
    delta_p = jax.tree.map(lambda o, n: n - o, old_p, params)
    _, jvpp = jax.jvp(lambda p: f(old_x, p), (old_p,), (delta_p,))
    jacx = jax.jacfwd(f, argnums=0)(old_x, old_p)
    u = -(jnp.linalg.inv(jacx))
    return old_x + u @ jvpp
```

Note that this function avoids materialising the parameter jacobian J_θ, instead finding the jacobian vector product $J_\theta \nabla_\theta$ using the function `jax.jvp`. It is possible to avoid materialising the matrix J_x using a similar strategy, as demonstrated by the function `guess_implicit_cg` below.

```
import jax

def guess_implicit_cg(guess_info, params, f):
    "Guess the next solution using the implicit function theorem."
    old_x, old_p, *_ = guess_info
    delta_p = jax.tree.map(lambda o, n: n - o, old_p, params)
    _, jvpp = jax.jvp(lambda p: f(old_x, p), (old_p,), (delta_p,))

    def matvec(v):
        "Compute Jx @ v"
        return jax.jvp(lambda x: f(x, old_p), (old_x,), (v,))[1]

    dx = -jax.scipy.sparse.linalg.cg(matvec, jvpp)[0]
    return old_x + dx
```

Which implementation of *guess-implicit* is preferable depends on the relative cost and reliability of directly calculating the matrix inverse J_x^{-1} as in the function `guess_implicit`, compared with numerically solving $J_x J_p \nabla_p = 0$ as in `guess_implicit_cg`. In general, this depends on the performance of the characteristics of the numerical solver relative to direct matrix inversion as implemented by the function `jax.numpy.linalg.inv`. For example, if J_x is sparse but positive semi-definite, `guess_implicit_cg` will likely perform better as the conjugate gradient method can exploit sparsity [28].

Appendix 4: software details

Using Blackjax [7], we implemented a version of a No-U-Turn sampler with dynamic guessing, which we call "grapeNUTS". For convenience we provide a Python package `grapevine` containing our implementation, including a utility function `run_grapenuts` with which users can easily test the GrapeNUTS sampler.

Our implementation builds on the popular JAX [29] scientific computing ecosystem, allowing users to straightforwardly define statistical models and adapt existing models to work with grapeNUTS. Similarly to Bayeux [30], grapevine requires a model in the form of a function that returns a scalar log probability density given a JAX PyTree of parameters; additionally, in grapevine such a function must also accept and return a PyTree containing information for guessing the answers to embedded root-finding problems. Users can specify root-finding problems using arbitrary JAX-compatible libraries, for example optimistix [8] or diffrax [31].

Appendix 5: benchmark models

Optimisation test functions

We compared our four heuristics on a series of variations of the following model:

$$\theta \sim Normal(0, \sigma_\theta)$$
$$x \sim Normal(\hat{x}, \sigma_x)$$

where $\hat{x}$ is the root such that $f(\hat{x} + \theta) = 0$.

In this equation f is the gradient of a textbook optimisation test function, sol is the textbook solution and θ is a vector with the same size as the input to f. We tested the following functions from the virtual library of simulation experiments [32]:

- Easom function (2 dimensions)
- 3-dimensional Levy function
- Beale function (2 dimensions)
- 3-dimensional Rastrigin function
- 3-dimensional and 8-dimensional Rosenbrock functions
- Styblinski-Tang function

Adversarial example

To test the effect of a non-smooth relationship between the solution vector x and the parameter vector θ, we constructed two adversarial models: "Adversarial-Dependent" and "Adversarial-Independent". Both of these models embed the non-smooth root-finding problem

$$g(x, \theta) = x^3 - x \odot \sin(k\theta) \odot \cos(k\theta)$$

where $\odot$ represents element-wise multiplication and scalar $k = 1E8$.

The model "Adversarial-Dependent" was the same as the test function models, i.e.

$$\theta \sim Normal(0, \sigma_\theta)$$
$$x \sim Normal(\hat{x}, \sigma_x)$$

In this model the solution x is coupled with the total log probability density $\pi(\theta)$ by the likelihood .

The model "Adversarial Independent" was the same, but without the likelihood, i.e.

$$\theta \sim Normal(0, \sigma_\theta)$$

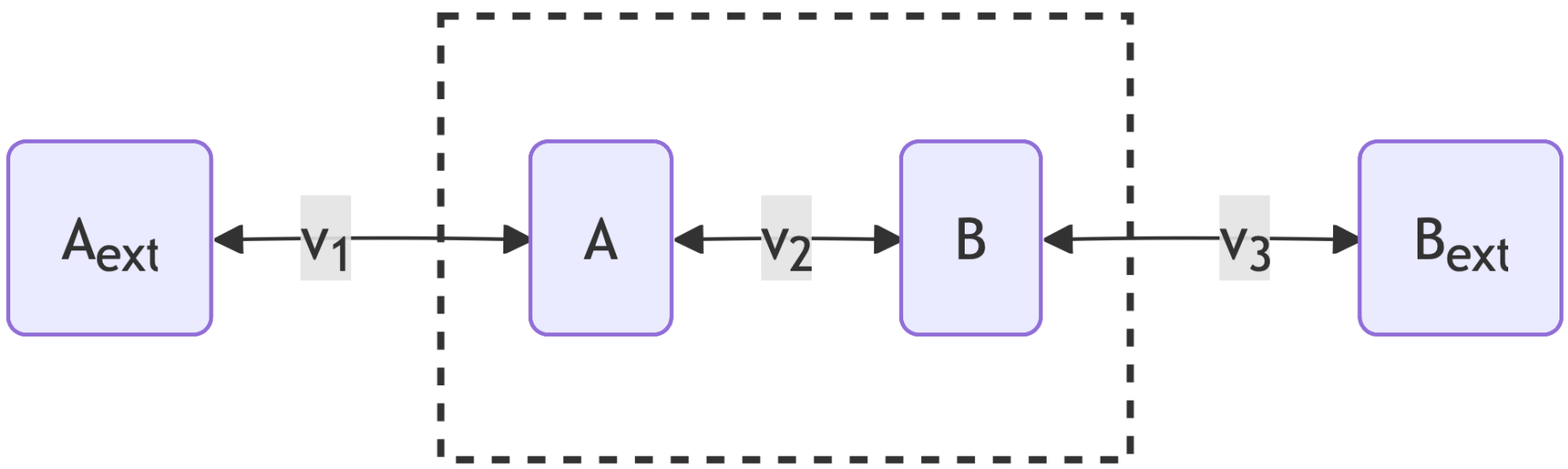
The code implementing "Adversarial Independent" nonetheless evaluated x at every leapfrog step.

Steady-state reaction networks

To illustrate our algorithm's practical relevance we constructed two statistical models where evaluating the likelihood $p(y \mid \theta)$ requires solving a steady-state problem, i.e. finding a vector x such that $\frac{dx}{dt} = S \cdot v(x, \theta) = \bar{0}$ for known real-valued matrix S and function v. In the context of chemical reaction networks, $S_{ij} \in \mathbb{R}$ can be interpreted as representing the amount of compound i consumed or produced by reaction j, x as the abundance of each compound and $v(x, \theta)$ as the rate of each reaction. The condition $\frac{dx}{dt} = \bar{0}$then represents the assumption that the compounds' abundances are constant.

We tested two similar models with this broad structure, one embedding a small biologically-inspired steady state problem and one a relatively large and well-studied realistic steady-state problem.

The smaller modelled network is a toy model of a linear pathway with three reversible reactions with rates v_1, v_2 and v_3. These reactions affect the internal concentrations x_A and x_B according to the following graph:



The rates v_1, v_2 and v_3, given internal concentrations

$$x = x_A, x_B$$

and parameters

$$\theta = k_A^m, k_B^m, v^{max}, k_1^{eq}, k_2^{eq}, k_3^{eq}, k_1^f, k_3^f, x_A^{ext}, x_B^{ext}$$

are calculated as follows:

$$
\begin{aligned}
v_1(x,\theta) &= k_1^f(x_A^{ext} - x_A/k_1^{eq}) \\
v_2(x,\theta) &= \frac{\frac{v^{max}}{k_A^m}(x_A - x_B/k_2^{eq})}{1 + x_A/k_A^m + x_B/k_B^m} \\
v_3(x,\theta) &= k_3^f(x_B^{ext} - x_B/k_3^{eq})
\end{aligned}
$$

According to these equations, rates v_1 and v_3 are described by mass-action rate laws: transport reactions are often modelled in this way. Rate v_2 is described by the Michaelis-Menten equation that is a popular choice for modelling the rates of enzyme-catalysed reactions.

The larger network models the mammalian methionine cycle, using equations taken from [12], including highly non-linear regulatory interactions. We selected this model because it describes a real biological system and has a convenient scale, being large and complex enough to test the grapevine method's scalability, but small enough for benchmarking purposes.

For the small linear network, we solved the embedded steady-state problem using the optimistix Newton solver. For the larger model of the methionine cycle we simulated the evolution of internal concentrations as an initial value problem until a steady-state event occurred, using the steady-state event handler and Kvaerno5 ODE solver provided by diffrax. In this case a guess is still needed in order to provide an initial value. Solving a steady-state problem in this way is often more robust than directly solving the system of algebraic equations; see [25] and [33] for further discussion.

Code used for these two experiments is in the code repository files `benchmarks/methionine.py` and `benchmarks/linear.py`.

References

1. **A Conceptual Introduction to Hamiltonian Monte Carlo** [PDF]
   Betancourt, M., 2018. arXiv:1701.02434 [stat]. DOI: https://doi.org/10.48550/arXiv.1701.02434
2. **A Computer Simulation Method for the Calculation of Equilibrium Constants for the Formation of Physical Clusters of Molecules: Application to Small Water Clusters** [link]
   Swope, W.C., Andersen, H.C., Berens, P.H. and Wilson, K.R., 1982. The Journal of Chemical Physics, Vol 76(1), pp. 637--649. DOI: https://doi.org/10.1063/1.442716
3. **Numerical Integrators for the Hybrid Monte Carlo Method** [PDF]
   Blanes, S., Casas, F. and Sanz-Serna, J.M., 2014. arXiv. DOI: https://doi.org/10.48550/arXiv.1405.3153
4. **Does Hamiltonian Monte Carlo Mix Faster than a Random Walk on Multimodal Densities?** [PDF]
   Mangoubi, O., Pillai, N.S. and Smith, A., 2018. arXiv. DOI: https://doi.org/10.48550/arXiv.1808.03230
5. **Stan Reference Manual, 2.37** [link]
   Team, S.D., 2026.
6. **PyMC: A Modern, and Comprehensive Probabilistic Programming Framework in Python** [link]
   Abril-Pla, O., Andreani, V., Carroll, C., Dong, L., Fonnesbeck, C.J., Kochurov, M., Kumar, R., Lao, J., Luhmann, C.C., Martin, O.A., Osthege, M., Vieira, R., Wiecki, T. and Zinkov, R., 2023. PeerJ Computer Science, Vol 9, pp. e1516. DOI: https://doi.org/10.7717/peerj-cs.1516
7. **BlackJAX: Composable Bayesian Inference in JAX**
   Cabezas, A., Corenflos, A., Lao, J. and Louf, R., 2024. DOI: https://doi.org/10.48550/arXiv.2402.10797
8. **Optimistix: Modular Optimisation in JAX and Equinox** [link]
   Rader, J., Lyons, T. and Kidger, P., 2024. arXiv:2402.09983. DOI: https://doi.org/10.48550/arXiv.2402.09983
9. **Roots.jl: Root Finding Functions for Julia** [link]
   Verzani, J., 2020.
10. **Stan User's Guide, 2.37** [link]
    Team, S.D., 2026.
11. **Uncertainty Quantification in Chemical Systems** [link]
    Najm, H.N., Debusschere, B.J., Marzouk, Y.M., Widmer, S. and Le Maitre, O.P., 2009. International Journal for Numerical Methods in Engineering, Vol 80(6-7), pp. 789--814. DOI: https://doi.org/10.1002/nme.2551
12. **Bayesian Regression Facilitates Quantitative Modeling of Cell Metabolism** [link]
    Groves, T., Cowie, N.L. and Nielsen, L.K., 2024. ACS Synthetic Biology. American Chemical Society. DOI: https://doi.org/10.1021/acssynbio.3c00662
13. **Increasing Certainty in Systems Biology Models Using Bayesian Multimodel Inference** [PDF]
    Linden-Santangeli, N., Zhang, J., Kramer, B. and Rangamani, P., 2024. arXiv. DOI: 10.48550/arXiv.2406.11178
14. **Probabilistic Pontryagin's Maximum Principle for Continuous-Time Model-Based Reinforcement Learning** [PDF]
    Leeftink, D., Yildiz, C., Ridderbusch, S., Hinne, M. and van Gerven, M., 2025. arXiv. DOI: https://doi.org/10.48550/arXiv.2504.02543
15. **An Importance Sampling Approach for Reliable and Efficient Inference in Bayesian Ordinary Differential Equation Models** [PDF]
    Timonen, J., Siccha, N., Bales, B., Lähdesmäki, H. and Vehtari, A., 2022. arXiv. DOI: https://doi.org/10.48550/arXiv.2205.09059
16. **A Review of Automatic Differentiation and Its Efficient Implementation** [PDF]
    Margossian, C.C., 2019. WIREs Data Mining and Knowledge Discovery, Vol 9(4). DOI: https://doi.org/10.1002/WIDM.1305
17. **The Solution of Nonlinear Systems of Equations by A-Stable Integration Techniques** [link]
    Boggs, P.T., 1971. SIAM Journal on Numerical Analysis, Vol 8(4), pp. 767--785. DOI: 10.1137/0708071
18. **On the Choice of Initial Guesses for the Newton-Raphson Algorithm** [link]
    Casella, F. and Bachmann, B., 2021. Applied Mathematics and Computation, Vol 398, pp. 125991. DOI: https://doi.org/10.1016/j.amc.2021.125991

19. **Introduction to Numerical Continuation Methods** [link]
Allgower, E.L. and Georg, K., 2003. Society for Industrial and Applied Mathematics. DOI: https://doi.org/10.1137/1.9780898719154

20. **Efficient Numerical Methods for Nonlinear MPC and Moving Horizon Estimation** [link]
Diehl, M., Ferreau, H.J. and Haverbeke, N., 2009. Nonlinear Model Predictive Control: Towards New Challenging Applications, pp. 391--417. Springer. DOI: https://doi.org/10.1007/978-3-642-01094-1_32

21. **Optimizing Millions of Hyperparameters by Implicit Differentiation** [HTML]
Lorraine, J., Vicol, P. and Duvenaud, D., 2020. Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics, Vol 108, pp. 1540--1552. PMLR.

22. **Learning to Warm-Start Fixed-Point Optimization Algorithms** [link]
Sambharya, R., Hall, G., Amos, B. and Stellato, B., 2024. J. Mach. Learn. Res., Vol 25(1), pp. 166:7854--166:7899.

23. **Geometric Integrators and the Hamiltonian Monte Carlo Method** [link]
Bou-Rabee, N. and Sanz-Serna, J.M., 2018. Acta Numerica, Vol 27, pp. 113--206. DOI: 10.1017/S0962492917000101

24. **GRASP: A Computational Platform for Building Kinetic Models of Cellular Metabolism** [link]
Matos, M.R.A., Saa, P.A., Cowie, N., Volkova, S., de Leeuw, M. and Nielsen, L.K., 2022. Bioinformatics Advances, Vol 2(1), pp. vbac066. DOI: https://doi.org/10.1093/bioadv/vbac066

25. **Tailored Parameter Optimization Methods for Ordinary Differential Equation Models with Steady-State Constraints** [link]
Fiedler, A., Raeth, S., Theis, F.J., Hausser, A. and Hasenauer, J., 2016. BMC Systems Biology, Vol 10(1), pp. 80. DOI: https://doi.org/10.1186/s12918-016-0319-7

26. **Fluctuation without Dissipation: Microcanonical Langevin Monte Carlo** [HTML]
Robnik, J. and Seljak, U., 2024. Proceedings of the 6th Symposium on Advances in Approximate Bayesian Inference, pp. 111--126. PMLR.

27. **The Implicit and the Inverse Function Theorems: Easy Proofs** [PDF]
de Oliveira, O.R.B., 2014. Real Analysis Exchange, Vol 39(1), pp. 207. DOI: https://doi.org/10.14321/realanalexch.39.1.0207

28. **Lecture Notes: Optimization III** [PDF]
Ben-Tal, A. and Nemirovski, A., 2023.

29. **JAX: Composable Transformations of Python+NumPy Programs** [link]
Bradbury, J., Frostig, R., Hawkins, P., Johnson, M.J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S. and Zhang, Q., 2018.

30. **Bayeux: State of the Art Inference for Your Bayesian Models** [link]
developers, B., 2025.

31. **On Neural Differential Equations** [PDF]
Kidger, P., 2021. DOI: https://doi.org/10.48550/arXiv.2202.02435

32. **Virtual Library of Simulation Experiments: Test Functions and Datasets** [link]
Surjanovic, S. and Bingham, D..

33. **Efficient Computation of Adjoint Sensitivities at Steady-State in ODE Models of Biochemical Reaction Networks** [link]
Lakrisenko, P., Stapor, P., Grein, S., Paszkowski, L., Pathirana, D., Frohlich, F., Lines, G.T., Weindl, D. and Hasenauer, J., 2023. PLOS Computational Biology, Vol 19(1), pp. e1010783. Public Library of Science. DOI: https://doi.org/10.1371/journal.pcbi.1010783

For attribution in academic contexts, please cite this work as

```
Anonymous, "Dynamic guessing for Hamiltonian Monte Carlo with embedded numerical root-finding", TMLR Beyond PDF, 2025.
```

BibTeX citation

```
@inproceedings{anonymous2025dynamicguessingfor,
  author = {Anonymous, },
  title = {Dynamic guessing for Hamiltonian Monte Carlo with embedded numerical root-finding},
  abstract = {A new algorithm and Python package for a class of difficult Bayesian statistical models},
  booktitle = {TMLR Beyond PDF},
  year = {2025},
  date = {April 22, 2026},
  note = {http://127.0.0.1:8080/tmlr-beyond-pdf/under_review/submission/},
  url  = {http://127.0.0.1:8080/tmlr-beyond-pdf/under_review/submission/}
}
```