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

Contents

Abstract . . . 1
Introduction . . . 2
Related work . . . 2
Methods . . . 3
Experiments . . . 4
Results . . . 6
Discussion and conclusions . . . 7

Appendix 1: pseudocode description of dynamic guessing **8**

Appendix 2: implementation of *guess-implicit* **9**

Appendix 3: software details **11**

Appendix 4: benchmark models **12**
Optimisation test functions . . . 12
Steady-state reaction networks . . . 12

Abstract

Thanks to scientific machine learning, it is possible to fit Bayesian statistical models whose parameters satisfy analytically intractable algebraic conditions like steady-state constraints. This is often done by embedding a differentiable numerical root-finder inside a gradient-based sampling algorithm like Hamiltonian Monte Carlo. However, computing and differentiating large numbers of numerical solutions comes at a high computational cost. 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 partially-known 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 known to satisfy steady-state constraints. Often the root-finding problem's solution must be found 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.

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 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, 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.

Related work

Bayesian statistical modelling with embedded numerical root-finding appears in a wide range of scientific contexts, including ignition chemistry [Najm et al., 2009], cell biology [Groves et al. [2024]][**?**] and optimal control [Leeftink et al., 2025].

We are not aware of any previous work that developed specialised inference methods for Bayesian models with embedded root finding. Instead, previous work has embedded numerical root-finding within generic inference algorithms such as Hamiltonian Monte Carlo [Neal [2011]][Betancourt, 2018] or a related algorithm such as the No-U-Turn sampler [Hoffman and Gelman, 2014] (below we include such variants under "HMC"). See [Timonen et al. [2022]][Margossian, 2019] for discussion of this approach. HMC is often preferable to alternative inference algorithms because of its good performance [Mangoubi et al., 2018], asymptotic exactness and the existence of well-maintained implementations, e.g. [Team [2026a]][Abril-Pla et al., 2023].

Hamiltonian Monte Carlo couples the target probability distribution with a dynamical system representing a particle lying on a surface with one dimension per parameter of the target distribution and a potential energy derived from

the target density. To generate a sample, a symplectic integrator simulates the particle's trajectory along the surface after a perturbation. Ideally, the trajectory takes the particle far from its starting point, leading to efficient sampling. The symplectic integrator proceeds by linearising the trajectory in small segments, evaluating the target density and its parameter gradients on log scale at the end of each segment. For targets with embedded numerical root-finding, the roots and their local parameter gradients must also be found at every step.

Many numerical root-finding algorithms are iterative, generating a series of numbers that start with an initial guess and converge towards the true solution. These algorithms tend to perform better, the closer the initial guess is to the true solution: see [Casella and Bachmann, 2021] for discussion of this topic. Thus a natural way to speed up HMC with embedded root-finding is to find the best possible guess for each problem; indeed, the main recommendation of the Stan user's guide [Team, 2026b] is to choose a guess that is reasonable, given the likely values of the parameters. However, previous implementations of HMC with embedded root-finding have required the starting guess to be the same for all problems that lie on the same simulated Hamiltonian trajectory. Since HMC trajectories aim to traverse a large distance in parameter space, and embedded root-finding problems will typically have different solutions depending on the parameters, the solution is likely to vary for different points on the trajectory. As a result, a guess that is optimal at one point on the trajectory will be sub-optimal elsewhere.

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. For example, an Euler predictor, which perturbs the previous solution using local gradient information, is known to work well with Newton solvers[Allgower and Georg, 2003]. Similarly, in model predictive control it is often useful to "warm start" a numerical solver using a solution from a previous time step [Diehl et al., 2009]. These ideas have recently been combined in the context of bilevel optimisation of neural network hyperparameters[Lorraine et al., 2020].

Methods

Instead of using the same starting guess for every root-finding problem on one simulated Hamiltonian trajectory, we propose choosing the guess dynamically, based on the previous integrator state. We call this 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 [Swope et al. [1982]][Blanes et al., 2014] with a dynamic state variable containing information that will be used to calculate an initial guess for the root-finding

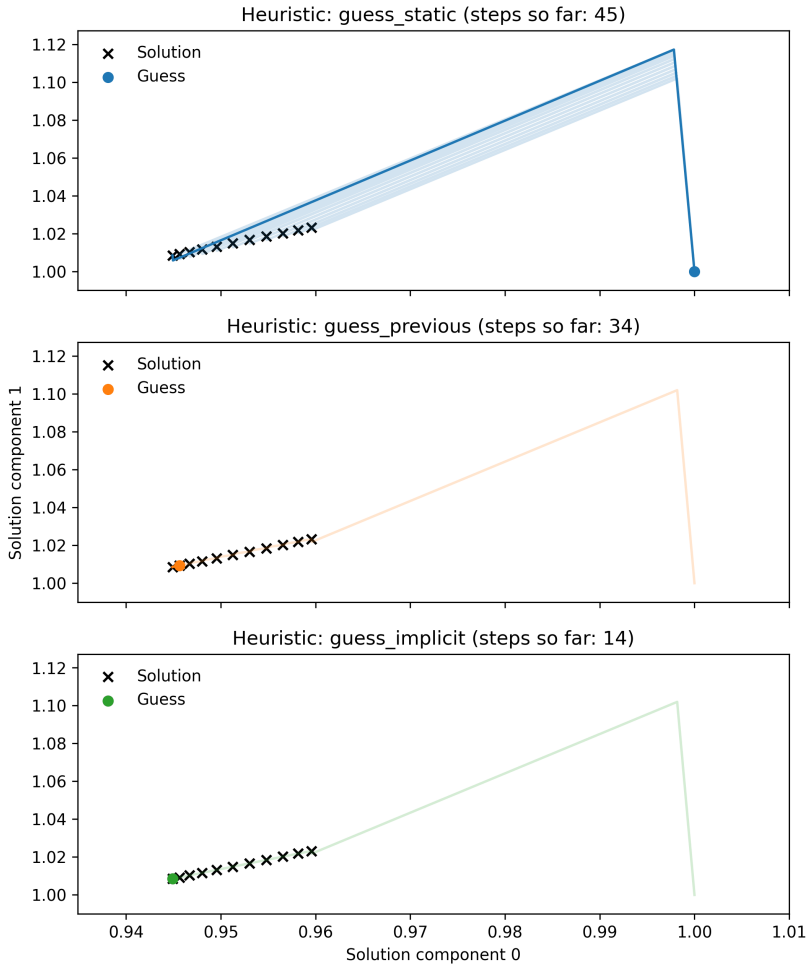
algorithm when the integrator updates its position. This variable is initialised at a default value, which is used to solve the numerical problem at the first step of any trajectory, and then modified at each position update. In this way all steps except the first have access to non-default guessing information. See appendix 1 below for a pseudocode description of the grapevine algorithm.

For HMC sampling to be valid, it is important that the target probability density function is constant. If all embedded root-finding problems have single roots, this will be the case up to numerical error for the grapevine algorithm, as the guessing heuristic does not affect what solution the root-finder will find, but only how quickly it converges. Therefore in this case there is no difference in MCMC validity between a grapevine sampler and an equivalent sampler with static guessing. 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 heuristics for generating an initial guess: *guess-previous* and *guess-implicit*. The information for the heuristic *guess-previous* is the solution of the previous root-finding problem. The heuristic is simply to use the previous solution as the next guess. The information for the heuristic *guess-implicit* is the solution of the previous root-finding problem, and also the parameter values at the previous step. The heuristic is to use implicit differentiation to find the local derivative of the previous solution with respect to the previous parameters, then obtain a guess by perturbing the previous solution by the product of this derivative and the change in parameter values. See Appendix 2 for details of how *guess-implicit* is defined and implemented.

Experiments

First we prepared an example illustrating the benefit of dynamic guessing by comparing 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), whereas the dynamic heuristics subsequently use better guesses that lie much closer to the target. As a result they take fewer Newton steps in total. Additionally, the *guess-implicit* heuristic is able to save more steps by exploiting gradient information.



This 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.

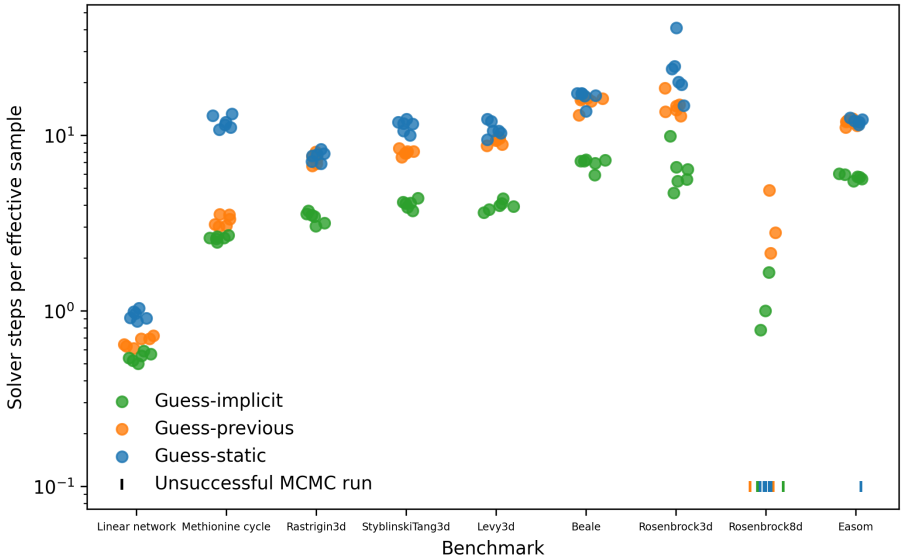
For a more comprehensive performance comparison, we therefore compared the performance of dynamic vs static guessing for a selection of models, each

embedding a different root-finding problem. To test the dynamic heuristics we augmented the No-U-Turn sampler with dynamic guessing, calling the resulting sampler "grapeNUTS". For each model, we randomly generated 6 parameter sets, each of which we used to randomly simulate an observation set. We then sampled from the resulting posterior distribution using unaugmented NUTS (*guess-static*), and grapeNUTS with our heuristics *guess-previous* and *guess-implicit*. The sampler configurations were the same for all heuristics and solver tolerances were set per problem. Appendix 3 below describes our software implementation and appendix 4 details the models.

We quantified performance by dividing the total number of Newton steps taken in each run by the effective sample size. See [Team, 2026a] for definition of effective sample size and [Kumar et al., 2019] for the software we used to calculate it. We diagnosed sampling by verifying that the effective sample size was not small compared with the total number of MCMC samples and that there were no post-warmup divergent transitions. We also recorded MCMC runs containing at least one solver convergence failure: in these cases we counted the whole run as a failure.

Results

The figure below shows performance comparisons for three dynamic guessing heuristics over nine statistical models with embedded root-finding problems. For each point, we randomly selected a true parameter set, then generated a simulated dataset consistently with the parametrised model. We then used the simulated data as measurements and performed posterior sampling. We quantified sampler performance as the number of solver steps divided by the number of effective samples generated, as plotted on logarithmic scale on the y axis. Lower values indicate better performance. Vertical lines denote MCMC runs that were unsuccessful because of a numerical solver failure.



Dynamic guessing tended to improve MCMC performance compared with static guessing for all the statistical models that we tested. The heuristic *guess-previous* showed similar or better performance compared with *guess-static*, whereas *guess-implicit* performed substantially better than *guess-static* on every benchmark. It is also notable that the dynamic algorithms failed less frequently than *guess-static* on the difficult `Rosenbrock8d` and `Easom` benchmarks.

All of our results 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 when traversing low-probability trajectories at the start of the adaptation phase. 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 a grapevine-augmented algorithm like grapeNUTS will typically improve sampling performance for MCMC with embedded root-finding, making it possible to fit previously infeasible statistical models.

While dynamic guessing generally outperformed static guessing, the relative performance of *guess-previous* compared with *guess-implicit* varied between benchmarks. We expect that this variation was caused by differences in how smoothly the solution of the embedded problem changes with changes in pa-

rameters. The smoother this relationship, the more likely that the embedded problems at adjacent points in an HMC trajectory will have similar solutions, leading to better relative performance of the *guess-previous* heuristic.

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.

While our implementation of the grapevine method is performant and flexible, it 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. In our implementation there is no 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 a stricter grapevine interface that makes inappropriate use of the guess variable impossible.

Appendix 1: 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 LogProb that takes in a set of parameter values and a root-finding solution
- a function GetInfo that takes in a set of parameter values and a root-finding solution, returning information for a heuristic.
- a function Heuristic that takes in the output of GetInfo and returns a guess
- initial parameters θ_0
- initial momentum p_0
- default information $\text{info}_{default}$
- 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 Heuristic, LogDensityAndInfo, PotentialGradientAndInfo, InitialiseTrajectory and LeapfrogStep:

Function 1: Generate a new guess (Heuristic)

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

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 (LogDensityAndInfo)

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

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

Algorithm 3: Evaluate potential energy and gradient (PotentialGradientAndInfo)

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

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

Algorithm 4: Initialise Trajectory (InitialiseTrajectory)

1. $U_0, \nabla_\theta U_0, \text{info} \leftarrow \text{PotentialGradientAndInfo}(\theta_0, \text{info}_{default})$
2. Return $\theta_0, p_0, U_0, \nabla_\theta U_0, \text{info}$

Algorithm 5: Leapfrog Integration Step (LeapfrogStep)

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

1. $p_{\text{mo}} \leftarrow p - \frac{\epsilon}{2}\nabla_\theta U$ (Update momentum, first half-step)
2. $\theta_+ \leftarrow \theta + \epsilon p_{\text{mo}}$ (Update parameters)
3. $U_+, \nabla_\theta U_+, \text{info}_+ \leftarrow \text{PotentialGradientAndInfo}(\theta_+, \text{info})$
4. $p_+ \leftarrow p_{\text{mo}} - \frac{\epsilon}{2}\nabla_\theta U_+$ (Update momentum, second half-step)
5. Return $\theta_+, p_+, U_+, \nabla_\theta U_+, \text{info}_+$

The functions InitialiseTrajectory and LeapfrogStep are drop-in substitutes for their counterparts in standard Hamiltonian Monte Carlo.

Appendix 2: implementation of *guess-implicit*

The *guess-implicit* heuristic is defined as follows, given previous solution x_{prev}, previous parameters θ_{prev} and current parameters θ_{next}:

$$\text{guess-implicit}(x_{prev}, \theta_{prev}, \theta_{next}) = x_{prev} + \frac{dx}{d\theta}\Delta\theta$$

where $\Delta\theta = (\theta_{next} - \theta_{prev})$.

To obtain $\frac{dx}{d\theta}$, we use the following consequence of the implicit function theorem[de Oliveira, 2014]:

$$\frac{\partial x}{\partial \theta} = -(\text{jac}_x\, f(x_{prev}, \theta_{prev}))^{-1}\, \text{jac}_\theta\, f(x_{prev}, \theta_{prev})$$

In this expression the term

$$\text{jac}_x\, f(x_{prev}, \theta_{prev})$$

abbreviated below to J_x, indicates the jacobian with respect to x of $f(x_{prev}, \theta_{prev})$. Similarly

$$\text{jac}_\theta\, f(x_{prev}, \theta_{prev}) = J_\theta$$

is the jacobian with respect to θ of $f(x_{prev}, \theta_{prev})$.

Substituting terms we then have

$$\text{guess-implicit}(x_{prev}, \theta_{prev}, \theta_{next}) = x_{prev} - J_x^{-1} J_\theta \Delta\theta$$

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.

```python
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 [Ben-Tal and Nemirovski, 2023].

Appendix 3: software details

Using Blackjax [Cabezas et al., 2024], 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 [Bradbury et al., 2018] scientific computing ecosystem, allowing users to straightforwardly define statistical models and adapt existing models to work with grapeNUTS. Similarly to Bayeux [developers, 2025], 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 [Rader et al., 2024] or diffrax [Kidger, 2021].

Appendix 4: 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 [Surjanovic and Bingham]:

- 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

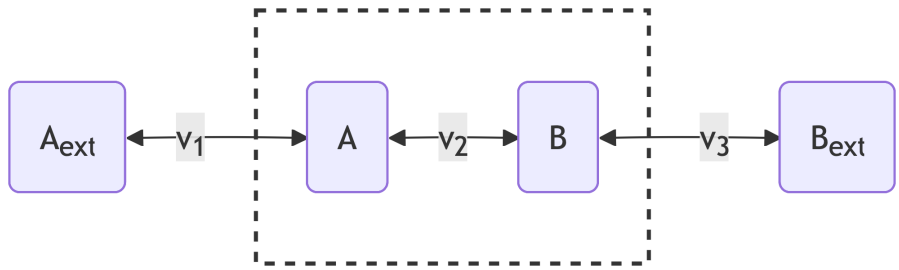
We chose these functions because they have a range of different difficult features for numerical solvers, vary in dimensions, have global minima so that the associated root-finding problems are well-posed and are straightforward to implement.

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. This kind of model is common in many fields, especially biochemistry: see for example [Matos et al. [2022]][Fiedler et al., 2016][Groves et al., 2024].

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 [Groves et al., 2024], 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 [Fiedler et al., 2016] and [Lakrisenko et al., 2023] for further discussion.

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

References

Oriol Abril-Pla, Virgile Andreani, Colin Carroll, Larry Dong, Christopher J. Fonnesbeck, Maxim Kochurov, Ravin Kumar, Junpeng Lao, Christian C. Luhmann, Osvaldo A. Martin, Michael Osthege, Ricardo Vieira, Thomas Wiecki, and Robert Zinkov. Pymc: A modern, and comprehensive probabilistic programming framework in python. *PeerJ Computer Science*, 9:e1516, September 2023. ISSN 2376-5992. doi: https://doi.org/10.7717/peerj-cs.1516. URL https://peerj.com/articles/cs-1516.

Eugene L. Allgower and Kurt Georg. *Introduction to Numerical Continuation Methods*. Society for Industrial and Applied Mathematics, January 2003. ISBN 978-0-89871-544-6 978-0-89871-915-4. doi: https://doi.org/10.1137/1.9780898719154. URL http://epubs.siam.org/doi/book/10.1137/1.9780898719154.

Aharon Ben-Tal and Arkadi Nemirovski. Lecture notes: Optimization iii, 2023. URL https://www2.isye.gatech.edu/~nemirovs/OPTIIILN2023Spring.pdf.

Michael Betancourt. A conceptual introduction to hamiltonian monte carlo. *arXiv:1701.02434 [stat]*, Jul 2018. doi: https://doi.org/10.48550/arXiv.1701.02434. URL http://arxiv.org/abs/1701.02434.

Sergio Blanes, Fernando Casas, and J. M. Sanz-Serna. Numerical integrators for the hybrid monte carlo method, May 2014. URL http://arxiv.org/abs/1405.3153.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. Jax: Composable transformations of python+numpy programs, 2018. URL http://github.com/jax-ml/jax.

Alberto Cabezas, Adrien Corenflos, Junpeng Lao, and Rémi Louf. Blackjax: Composable bayesian inference in jax, 2024.

Francesco Casella and Bernhard Bachmann. On the choice of initial guesses for the newton-raphson algorithm. *Applied Mathematics and Computation*, 398:125991, June 2021. ISSN 0096-3003. doi: https://doi.org/10.1016/j.amc.2021.125991. URL https://www.sciencedirect.com/science/article/pii/S0096300321000394.

Oswaldo Rio Branco de Oliveira. The implicit and the inverse function theorems: Easy proofs. *Real Analysis Exchange*, 39(1):207, 2014. ISSN 01471937. doi: https://doi.org/10.14321/realanalexch.39.1.0207. URL http://arxiv.org/abs/1212.2066.

Bayeux developers. Bayeux: State of the art inference for your bayesian models, 2025. URL https://github.com/jax-ml/bayeux.

Moritz Diehl, Hans Joachim Ferreau, and Niels Haverbeke. Efficient numerical methods for nonlinear mpc and moving horizon estimation. In Lalo Magni, Davide Martino Raimondo, and Frank Allgöwer, editors, *Nonlinear Model Predictive Control: Towards New Challenging Applications*, pages 391–417. Springer, Berlin, Heidelberg, 2009. ISBN 978-3-642-01094-1. doi: https://doi.org/10.1007/978-3-642-01094-1_32. URL https://doi.org/10.1007/978-3-642-01094-1_32.

Anna Fiedler, Sebastian Raeth, Fabian J. Theis, Angelika Hausser, and Jan Hasenauer. Tailored parameter optimization methods for ordinary differential equation models with steady-state constraints. *BMC Systems Biology*, 10(1):80, August 2016. ISSN 1752-0509. doi: https://doi.org/10.1186/s12918-016-0319-7. URL https://doi.org/10.1186/s12918-016-0319-7.

Teddy Groves, Nicholas Luke Cowie, and Lars Keld Nielsen. Bayesian regression facilitates quantitative modeling of cell metabolism. *ACS Synthetic Biology*, April 2024. doi: https://doi.org/10.1021/acssynbio.3c00662. URL https://doi.org/10.1021/acssynbio.3c00662.

Matthew D Hoffman and Andrew Gelman. The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo. *Journal of Machine Learning Research*, 15:1593–1623, 2014.

Patrick Kidger. *On Neural Differential Equations*. PhD thesis, University of Oxford, 2021. URL https://arxiv.org/abs/2202.02435.

Ravin Kumar, Colin Carroll, Ari Hartikainen, and Osvaldo Martin. Arviz a unified library for exploratory analysis of bayesian models in python. *Journal of Open Source Software*, 4(33):1143, January 2019. ISSN 2475-9066. doi: https://doi.org/10.21105/joss.01143. URL http://joss.theoj.org/papers/10.21105/joss.01143.

Polina Lakrisenko, Paul Stapor, Stephan Grein, Łukasz Paszkowski, Dilan Pathirana, Fabian Fröhlich, Glenn Terje Lines, Daniel Weindl, and Jan Hasenauer. Efficient computation of adjoint sensitivities at steady-state in ode models of biochemical reaction networks. *PLOS Computational Biology*, 19(1):e1010783, January 2023. ISSN 1553-7358. doi: https:

//doi.org/10.1371/journal.pcbi.1010783. URL https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1010783.

David Leeftink, Çağatay Yıldız, Steffen Ridderbusch, Max Hinne, and Marcel van Gerven. Probabilistic pontryagin's maximum principle for continuous-time model-based reinforcement learning, April 2025. URL http://arxiv.org/abs/2504.02543.

Jonathan Lorraine, Paul Vicol, and David Duvenaud. Optimizing millions of hyperparameters by implicit differentiation. In Silvia Chiappa and Roberto Calandra, editors, *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pages 1540–1552. PMLR, August 2020. URL https://proceedings.mlr.press/v108/lorraine20a.html.

Oren Mangoubi, Natesh S. Pillai, and Aaron Smith. Does hamiltonian monte carlo mix faster than a random walk on multimodal densities?, September 2018. URL http://arxiv.org/abs/1808.03230.

Charles C. Margossian. A review of automatic differentiation and its efficient implementation. *WIREs Data Mining and Knowledge Discovery*, 9(4), July 2019. ISSN 1942-4787, 1942-4795. doi: https://doi.org/10.1002/WIDM.1305. URL http://arxiv.org/abs/1811.05031.

Marta R A Matos, Pedro A Saa, Nicholas Cowie, Svetlana Volkova, Marina de Leeuw, and Lars K Nielsen. Grasp: A computational platform for building kinetic models of cellular metabolism. *Bioinformatics Advances*, 2(1):vbac066, January 2022. ISSN 2635-0041. doi: https://doi.org/10.1093/bioadv/vbac066. URL https://doi.org/10.1093/bioadv/vbac066.

H. N. Najm, B. J. Debusschere, Y. M. Marzouk, S. Widmer, and O. P. Le Maître. Uncertainty quantification in chemical systems. *International Journal for Numerical Methods in Engineering*, 80(6-7):789–814, 2009. ISSN 1097-0207. doi: https://doi.org/10.1002/nme.2551. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.2551.

Radford Neal. Mcmc using hamiltonian dynamics. In Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng, editors, *Handbook of Markov Chain Monte Carlo*, volume 20116022. Chapman and Hall/CRC, may 2011. ISBN 978-1-4200-7941-8 978-1-4200-7942-5. doi: https://doi.org/10.1201/b10905-6. URL http://www.crcnetbase.com/doi/abs/10.1201/b10905-6.

Jason Rader, Terry Lyons, and Patrick Kidger. Optimistix: Modular optimisation in jax and equinox. *arXiv:2402.09983*, 2024. doi: https://doi.org/10.48550/arXiv.2402.09983. URL https://doi.org/10.48550/arXiv.2402.09983.

S. Surjanovic and D. Bingham. Virtual library of simulation experiments: Test functions and datasets. URL http://www.sfu.ca/{~}ssurjano.

William C. Swope, Hans C. Andersen, Peter H. Berens, and Kent R. Wilson. A computer simulation method for the calculation of equilibrium constants

for the formation of physical clusters of molecules: Application to small water clusters. *The Journal of Chemical Physics*, 76(1):637–649, January 1982. ISSN 0021-9606. doi: https://doi.org/10.1063/1.442716. URL https://doi.org/10.1063/1.442716.

Stan Development Team. Stan reference manual, 2.37, 2026a. URL https://mc-stan.org.

Stan Development Team. Stan user's guide, 2.37, 2026b. URL https://mc-stan.org.

Juho Timonen, Nikolas Siccha, Ben Bales, Harri Lähdesmäki, and Aki Vehtari. An importance sampling approach for reliable and efficient inference in bayesian ordinary differential equation models, May 2022. URL http://arxiv.org/abs/2205.09059.