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

Teddy Groves
The Novo Nordisk Center for Biosustainability, DTU, Denmark
tedgro@biosustain.dtu.dk

Nicholas Luke Cowie
The Novo Nordisk Center for Biosustainability, DTU, Denmark
nicow@biosustain.dtu.dk

Lars Keld Nielsen
The Novo Nordisk Center for Biosustainability, DTU, Denmark;
Australian Institute for Bioengineering and Nanotechnology, The University of Queensland, Australia
lakeni@biosustain.dtu.dk

Abstract

Thanks to scientific machine learning, it is possible to fit Bayesian statistical models whose parameters satisfy analytically intractable algebraic conditions like steady-state 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 demonstrate that dynamically varying the starting guess within Hamiltonian trajectorie 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.

1 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 experiment results reported in this paper are available at `https://github.com/dtu-qmcm/grapevine` and from the Python Package Index.

2 Related work

Bayesian statistical modelling with embedded numerical root-finding are reported in a wide range of scientific contexts, including ignition chemistry [1], cell biology [2, 3] and optimal control [4].

Hamiltonian Monte Carlo [5, 6] and related algorithms such as the No-U-Turn sampler [7] (below we include such variants under "HMC") support sampling for this kind of model: see [8, 9]. HMC is often preferable to alternative inference algorithms because of its good performance [10], asymptotic exactness and the existence of well-maintained implementations, e.g. [11, 12].

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 [13] 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 guide [14] 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-obtimal elsewhere.

3 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 [15, 16] 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.

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 1 for details of how *guess-implicit* is defined and implemented.

3.1 Experiments

We compared the performance of dynamic vs static guessing by fitting a range of models with embedded root-finding problems using a version of the No-U-Turn sampler augmented with dynamic guessing, which we called GrapeNUTS. For each model, we randomly generated 6 parameter sets, each of which we used to randomly simulate one fake 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.

We quantified performance by dividing the total number of Newton steps taken in each run by the effective sample size. See [14] for definition of effective sample size and [17] 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.

All of our results were computed on a MacBook Pro 2024 with Apple M4 Pro processor and 48GB RAM, running macOS 15.3.1.

4 Results

Figure 1 illustrates the grapevine method; Figure 2 shows the results of our experiments.

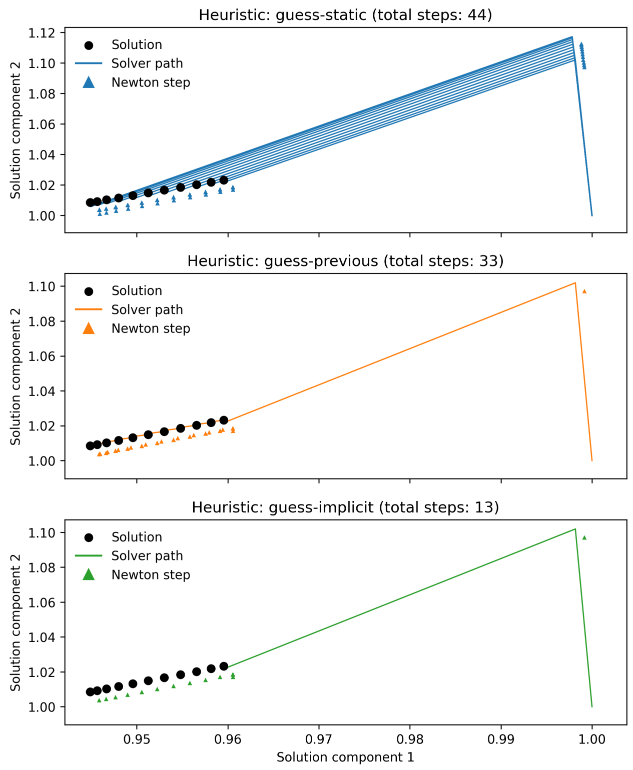

Figure 1: An example illustrating the benefit of dynamic guessing by comparing behaviour of dynamic and static heuristics along a single Hamiltonian trajectory. Black dots show the solutions of a root-finding problem (finding the minimum of a parametrised 2-dimensional Rosenbrock function) at 11 steps along a simulated Hamiltonian trajectory through parameter space. Coloured triangles show offset intermediate solutions from a Newton solver. Lines show the path from step to step. To find the first solution all heuristics use the default guess at coordinate (1, 1). The dynamic heuristics produce better guesses in this case, resulting in fewer Newton steps.

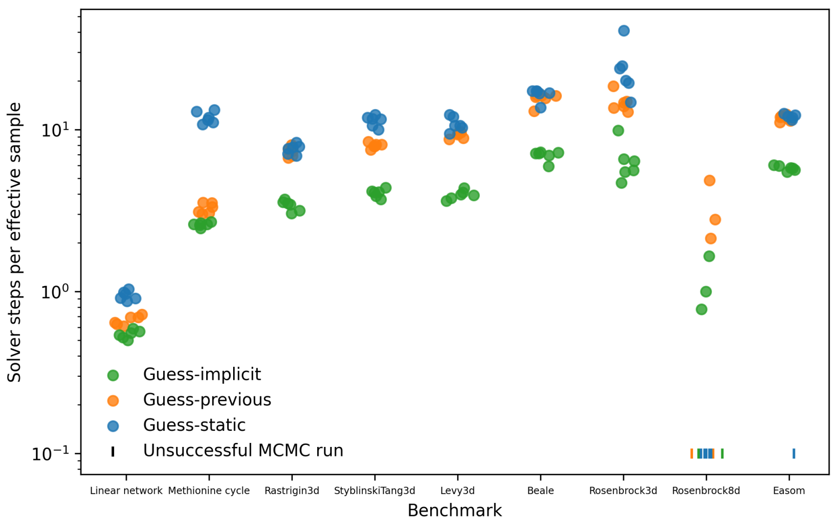

Figure 2: Performance comparison for three dynamic guessing heuristics over nine statistical models with embedded root-finding problems. For each point, a true parameter set was randomly selected, then a simulated dataset was generated consistently with the target model and used to generate posterior samples. Sampler performance is quantified by the number of solver steps divided by the number 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.

5 Discussion and conclusions

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. This is likely because the dynamic algorithms failed 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 these results, we expect that replacing a static guessing algorithm with a grapevine-augmented algorithm like grapeNUTS will typically improve sampling performance for similar MCMC tasks, 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 parameters. 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.

6 Acknowledgements

This work was funded by the Novo Nordisk Foundation (grant numbers NNF20CC0035580 NNF14OC0009473) and by the European Commission under the Horizon-Widera-2022 program (Project number: 101060066 Project name: Centre for Digitalisation of Biology Towards the Next-Generation of Biosustainable Products; in short DIGIBIO).

References

[1] 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*, vol. 80, no. 6-7, pp. 789–814, 2009.

[2] T. Groves, N. L. Cowie, and L. K. Nielsen, "Bayesian Regression Facilitates Quantitative Modeling of Cell Metabolism," *ACS Synthetic Biology*, Apr. 2024.

[3] N. Linden-Santangeli, J. Zhang, B. Kramer, and P. Rangamani, "Increasing certainty in systems biology models using Bayesian multimodel inference," June 2024.

[4] D. Leeftink, Ç. Yıldız, S. Ridderbusch, M. Hinne, and M. van Gerven, "Probabilistic Pontryagin's Maximum Principle for Continuous-Time Model-Based Reinforcement Learning," Apr. 2025.

[5] R. Neal, "MCMC Using Hamiltonian Dynamics," in *Handbook of Markov Chain Monte Carlo* (S. Brooks, A. Gelman, G. Jones, and X.-L. Meng, eds.), vol. 20116022, Chapman and Hall/CRC, May 2011.

[6] M. Betancourt, "A Conceptual Introduction to Hamiltonian Monte Carlo," *arXiv:1701.02434 [stat]*, July 2018.

[7] M. D. Hoffman and A. Gelman, "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo," *Journal of Machine Learning Research*, vol. 15, pp. 1593–1623, 2014.

[8] J. Timonen, N. Siccha, B. Bales, H. Lähdesm äki, and A. Vehtari, "An importance sampling approach for reliable and efficient inference in Bayesian ordinary differential equation models," May 2022.

[9] C. C. Margossian, "A Review of automatic differentiation and its efficient implementation," *WIREs Data Mining and Knowledge Discovery*, vol. 9, July 2019.

[10] O. Mangoubi, N. S. Pillai, and A. Smith, "Does Hamiltonian Monte Carlo mix faster than a random walk on multimodal densities?," Sept. 2018.

[11] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell, "Stan: A Probabilistic Programming Language," *Journal of Statistical Software*, vol. 76, pp. 1–32, Jan. 2017.

[12] O. Abril-Pla, V. Andreani, C. Carroll, L. Dong, C. J. Fonnesbeck, M. Kochurov, R. Kumar, J. Lao, C. C. Luhmann, O. A. Martin, M. Osthege, R. Vieira, T. Wiecki, and R. Zinkov, "PyMC: A modern, and comprehensive probabilistic programming framework in Python," *PeerJ Computer Science*, vol. 9, p. e1516, Sept. 2023.

[13] F. Casella and B. Bachmann, "On the choice of initial guesses for the Newton-Raphson algorithm," *Applied Mathematics and Computation*, vol. 398, p. 125991, June 2021.

[14] Stan Development Team, "Stan Modeling Language Users Guide and Reference Manual, version 2.36," 2025.

[15] W. C. Swope, H. C. Andersen, P. H. Berens, and K. 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*, vol. 76, pp. 637–649, Jan. 1982.

[16] S. Blanes, F. Casas, and J. M. Sanz-Serna, "Numerical integrators for the Hybrid Monte Carlo method," May 2014.

[17] R. Kumar, C. Carroll, A. Hartikainen, and O. Martin, "ArviZ a unified library for exploratory analysis of Bayesian models in Python," *Journal of Open Source Software*, vol. 4, p. 1143, Jan. 2019.

[18] O. R. B. de Oliveira, "The Implicit and the Inverse Function theorems: Easy proofs," *Real Analysis Exchange*, vol. 39, no. 1, p. 207, 2014.

[19] A. Ben-Tal and A. Nemirovski, "CONVEX ANALYSIS NONLINEAR PROGRAMMING THEORY NONLINEAR PROGRAMMING ALGORITHMS."

[20] A. Cabezas, A. Corenflos, J. Lao, and R. Louf, "BlackJAX: Composable Bayesian inference in JAX," 2024.

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

[22] Bayeux developers, "Bayeux: State of the art inference for your bayesian models," 2025.

[23] J. Rader, T. Lyons, and P. Kidger, "Optimistix: Modular optimisation in JAX and equinox," *arXiv:2402.09983*, 2024.

[24] P. Kidger, *On Neural Differential Equations*. PhD thesis, University of Oxford, 2021.

[25] S. Surjanovic and D. Bingham, "Virtual library of simulation experiments: Test functions and datasets."

[26] M. R. A. Matos, P. A. Saa, N. Cowie, S. Volkova, M. de Leeuw, and L. K. Nielsen, "GRASP: A computational platform for building kinetic models of cellular metabolism," *Bioinformatics Advances*, vol. 2, p. vbac066, Jan. 2022.

[27] A. Fiedler, S. Raeth, F. J. Theis, A. Hausser, and J. Hasenauer, "Tailored parameter optimization methods for ordinary differential equation models with steady-state constraints," *BMC Systems Biology*, vol. 10, p. 80, Aug. 2016.

[28] P. Lakrisenko, P. Stapor, S. Grein, Ł. Paszkowski, D. Pathirana, F. Fröhlich, G. T. Lines, D. Weindl, and J. Hasenauer, "Efficient computation of adjoint sensitivities at steady-state in ODE models of biochemical reaction networks," *PLOS Computational Biology*, vol. 19, p. e1010783, Jan. 2023.

7 Appendix 1: implementation of *guess-implicit*

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

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

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

To obtain $\frac{dx}{d\theta}$, we use the following consequence of the implicit function theorem[18]:

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

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

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

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

```
import jax

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 [19].

8 Appendix 2: software details

Using Blackjax [20], 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 [21] scientific computing ecosystem, allowing users to straightforwardly define statistical models and adapt existing models to work with grapeNUTS. Similarly to Bayeux [22], 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 [23] or diffrax [24].

9 Appendix 3: benchmark models

9.1 Optimisation test functions

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

$$\begin{aligned} \theta &\sim Normal(0, \sigma_\theta) \\ \hat{y} &= root_y(f(y+\theta)) \\ y &\sim Normal(\hat{y}, \sigma_y) \end{aligned}$$

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 [25]:

- Easom function (2 dimensions)
- 3-dimension Levy function
- Beale function (2 dimensions)
- 3-dimension Rastrigin function
- 3-dimension and 8-dimension 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.

9.2 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 [26, 27, 2].

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 A^{int} and B^{int} according to the following graph:

$$\begin{tikzcd}
A^{ext} \arrow[r, leftrightarrow, "v_1"] & A^{int} \arrow[r, leftrightarrow, "v_2"] & B^{int} \arrow[r, leftrightarrow, "v_3"] & B^{ext}
\end{tikzcd}$$

The rates v_1, v_2 and v_3 are calculated as follows, given internal concentrations $x^{int} = x_A^{int}, x_B^{int}$ and parameters $\theta = k_A^m, k_B^m, v^{max}, k_1^{eq}, k_2^{eq}, k_3^{eq}, k_1^f, k_3^f, x_A^{ext}, x_B^{ext}$:

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

According to these equations, rates v_1 and $v3$ 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 [2], 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 [27] and [28, Introduction] for further discussion.

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