Adaptive Probabilistic ODE Solvers Without Adaptive Memory Requirements

Nicholas Krämer *Technical University of Denmark, Kongens Lyngby, Denmark*

Abstract

Despite substantial progress in recent years, probabilistic solvers with adaptive step sizes can still not solve memory-demanding differential equations—unless we care only about a single point in time (which is far too restrictive; we want the whole time series). Counterintuitively, the culprit is the adaptivity itself: Its unpredictable memory demands easily exceed our machine's capabilities, making our simulations fail unexpectedly and without warning. Still, dropping adaptivity would abandon years of progress, which can't be the answer. In this work, we solve this conundrum. We develop an adaptive probabilistic solver with fixed memory demands building on recent developments in robust state estimation. Switching to our method (i) eliminates memory issues for long time series, (ii) accelerates simulations by orders of magnitude through unlocking just-in-time compilation, and (iii) makes adaptive probabilistic solvers compatible with scientific computing in JAX.

JAX library: `pip install probdiffeq`
Experiment code: `https://github.com/pnkraemer/code-adaptive-prob-ode-solvers`

Proceedings of the 1st International Conference on Probabilistic Numerics (ProbNum) 2025, Sophia Antipolis, France. PMLR Volume 271. Copyright 2025 with the author(s)

1 Introduction: Adaptive Probabilistic Solvers vs. Limited Memory

Probabilistic numerical methods solve problems from numerical computing by manipulating random variables (Hennig et al., 2022). For example, conditioning a stochastic process $u(t)$ on the constraints $\{u'(t_n) = u(t_n)\}_{n=1}^{N}$ and the initial condition $u(0) = u_0$ approximates the solution of $u'(t) = u(t)$, $u(0) = u_0$ (Cockayne et al., 2019). Unlike more traditional techniques, these new algorithms emphasise uncertainty quantification and the interplay between simulators and observational data. For instance, the works by Kersting et al. (2020); Tronarp et al. (2022); Schmidt et al. (2021); Beck et al. (2024); Oesterle et al. (2022); Oates et al. (2019); Lahr et al. (2024) show the utility of this new paradigm for uncertainty quantification of differential equations. Unfortunately, the probabilistic solvers' usefulness is limited by memory, as follows:

Since solving nonlinear differential equations requires time discretisation, users must provide implementations with either (i) a grid; or (ii) an error tolerance, according to which the solver selects the grid automatically and adaptively. This adaptive step-size selection has been one of the great improvements to the practicality of probabilistic solvers (Schober et al., 2019; Bosch et al., 2021). The reason is that it drastically lessens the resolution needed for solving challenging problems. For example, for Van der Pol's (1920) system,

$$u''(t) = 10^3(u'(t)(1 - u(t)^2) - u(t)), \tag{1}$$

adaptivity reduces the number of steps, thus the solver's memory footprint and general "amount of work", from almost 750,000 to under 3,000 grid points, while achieving similar accuracy (Figure 1). Furthermore, this 250× memory improvement comes with a proportional runtime reduction, but we mainly care about memory in this

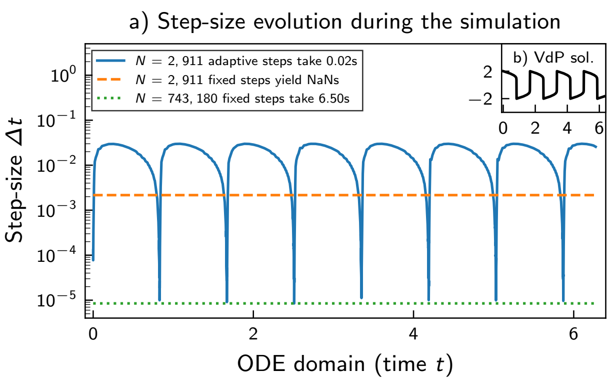

Figure 1: Adaptive step-size selection minimises the amount of work for solving (stiff) differential equations (a). Van-der-Pol system (solution in b), tolerance 0.001. One adaptive versus two fixed-grid solvers: one matches the adaptive solvers' number of steps (and fails), and one matches its accuracy (but requires over 250× more memory & runtime). Detailed setup: Appendix A.

work because many simulations, for instance, those in Section 5.2, are memory-limited and not runtime-limited. Our work shows that one can decrease memory requirements without increasing runtime. In practice, JAX-code actually becomes faster since just-in-time (JIT) compilation is now possible; more on this later.

In other words, adaptive step-size selection is critical for decreasing the solvers' memory demands without losing accuracy. *However, it doesn't reduce them enough:* The issue is that adaptive solvers represent the solution with all points used for the computation, regardless of whether or not only a tiny subset of these points is relevant to what we plan to do with the solution. For example, if we were to estimate an unknown initial condition of the van-der-Pol system (Equation 1) from a single observation in the middle of the time domain, we would still have to store all $\approx 3,000$ states used for the simulation. This excessive storage is problematic:

First, it consumes more memory than necessary, potentially too much for the simulation to be feasible. *Our work enables high-precision simulations of challenging systems on a small laptop, which exceeds prior works' memory capabilities.* Section 5.2 shows an example.

Second, not knowing the memory footprint of the simulation complicates using programming frameworks that must know the exact memory demands of a program before compiling/running it, for example, JAX (Bradbury et al., 2018). *Our work enables JIT-compiling JAX-code for adaptive probabilistic solvers.* Section 5.1 shows how this drastically improves performance.

To illustrate the second problem: until the present work, existing JAX-based implementations, for instance this[1] one or this[2] one, either use fixed steps, adaptively simulate only the terminal value (ignoring the interior of the time domain) or can't compile the solution routines. Our work doesn't suffer from any of these problems. An open-source JAX implementation of our method is available under this[3] link; install it via `pip install probdiffeq`. The lack of JIT'able adaptive solvers in JAX was the original motivation for the research presented in here.

2 Problem Statement

To summarise, the present work solves a practical problem with probabilistic differential equations solvers:

Problem statement. *We aim to solve a differential equation adaptively, to a user-specified tolerance, and with memory requirements that are fixed, independent of the time-stepping, and known* <u>*before*</u> *the simulation.*

We call this process of adaptively generating the solution at a pre-specified set of target points "adaptive target simulation," as opposed to what we call "adaptive simulation:" returning the adaptive solution at all locations used for the approximation. We could combine adaptive simulation with interpolation to achieve adaptive target simulation. Still, this combination would inherit all the memory-related problems from adaptive simulation, and our goal is to avoid them. Interpolation is thus not the correct answer; and using what often works for traditional solvers isn't either, as follows.

Adaptive target simulation for non-probabilistic solvers, like Runge–Kutta methods, usually reduces to a decision of whether or not to save the current step. Past and future increments are rarely affected. For example, passing a non-empty *t_eval* (the target locations) to SciPy's *solve_ivp* affects only very few lines of code: storing an interpolation instead of the time-step.[4] However, the case for probabilistic solvers is more complicated because they operate through a forward- and a backward-pass, where the backward-pass depends on all intermediate results of the forward pass (e.g. Krämer, 2024). Because of this dependence, all intermediate states have to be saved if we use existing formulations of adaptive solvers. For reference, the example in Figure 1 uses approximately 3,000 adaptive steps, and all of those need to be saved to parametrise the posterior distribution. However, storing even 3,000 locations may not be feasible for high-dimensional problems like discretised partial differential equations (Section 5.2).

In this work, we show that storing all intermediate steps is unnecessary. We modify the solver recursion so that the backward pass only depends on the solution's values at all target locations instead of those at compute locations. More technically, we extend recent advances in numerically robust fixed-point smoothing (Krämer, 2025) to simulating differential equations and then combine them with adaptive time-stepping. The result is a previously unknown implementation of adaptive probabilistic ODE solvers. However, existing codes need not be trashed: We can update them by tracking a single additional variable at each stage of the forward pass. The rest of this paper details how.

3 Method: Adaptive Target Simulation via Fixed-Point Smoothing

3.1 Background on Probabilistic Solvers

This work discusses solving ordinary-differential-equation-based initial value problems (ODEs). For example, the SIR model (Kermack and McKendrick, 1927) is an ODE that describes a disease outbreak, starting at initial case counts and evolving according to specific nonlinear dynamics. More specifically, let $f : \mathbb{R}^d \to \mathbb{R}^d$ be a known, potentially nonlinear vector field and $u_0 \in \mathbb{R}^d$ be a known vector. Consider the ODE

$$\frac{d}{dt}u(t) = f(u(t)), \quad t \in [0, 1], \quad u(0) = u_0, \tag{2}$$

and assume that it admits a unique, sufficiently regular solution. The precise form of Equation 2 does not matter much; second derivatives or time-varying dynamics only need minimal changes, and the general template continues to apply (Bosch et al., 2022). In other words, we use Equation 2 to explain the algorithm but it applies to general ODEs via Bosch et al. (2022). For instance, the simulation in Figure 1 involves second derivatives.

Numerical solution of ODEs requires approximation because the vector field f is nonlinear. Approximation requires time-discretisation: Introduce the point sets

$$\text{"Compute grid"}: \quad t_{0:N} \coloneqq \{t_0, ..., t_N\}, \tag{3a}$$
$$\text{"Target grid"}: \quad s_{0:M} \coloneqq \{s_0, ..., s_M\}. \tag{3b}$$

The target grid $s_{0:M}$ is specified by the user, and the compute grid $t_{0:N}$ selected by the algorithm. No technical relation exists between $t_{0:N}$ and $s_{0:M}$, but we optimise our method for $M \ll N$. Without a loss of generality, assume both grids' endpoints coincide with the domain's boundary, $t_0 = s_0 = 0$ and $t_N = s_M = 1$. If not, augment the point sets. Treating the compute- and the target-grid differently is central to this work.

[1]`https://github.com/pnkraemer/tornadox`
[2]`https://github.com/mlysy/rodeo`
[3]`https://github.com/pnkraemer/probdiffeq`
[4]`https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html`

Probabilistic numerical solvers condition a prior distribution (usually Gaussian) on satisfying the ODE on the compute-grid $t_{0:N}$. Let u be a Gaussian process that admits L or more derivatives (in the mean-square sense, like it's typical in the Gaussian process literature (Williams and Rasmussen, 2006)), and assume that L is at least as large as the highest derivative in the ODE (Equation 2). For example, ODEs involving first derivatives need $L \geq 1$ which is satisfied by, for example, a Matérn$\left(\frac{3}{2}\right)$ or a Matérn$\left(\frac{5}{2}\right)$ process. For ODEs involving second derivatives, the Matérn$\left(\frac{3}{2}\right)$ would not be regular enough. L can be thought of as the order of the solver; the higher L is (with everything else being equal), the more accurate the approximation, but also, the more expensive and ill-conditioned the solution process becomes (Krämer and Hennig, 2024).

Since differentiation is linear,

$$\mathfrak{u} := \left[u, \frac{d}{dt}u, \frac{d^2}{dt^2}u, \dots \frac{d^L}{dt^L}u\right] \tag{4}$$

is a Gaussian process. Abbreviate the ODE residual

$$\mathfrak{r}(t) := \frac{d}{dt}u(t) - f(u(t)). \tag{5}$$

This $\mathfrak{r}$ is a nonlinear function of $\mathfrak{u}$; recall that $\mathfrak{u}$ contains $\frac{d}{dt}u$. A globally small $\mathfrak{r}$ implies that a given u approximates the ODE solution accurately. Adopting Krämer's (2024) terminology, we call any approximation of

$$p(\mathfrak{u}(s_{0:M}) \mid \mathfrak{r}(t_{0:N}) = 0, u_0) \tag{6}$$

a "probabilistic ODE solution". Any algorithm that computes such a probabilistic solution shall be a "probabilistic solver". For the rest of this paper, we drop the "probabilistic" and only write "solver/solution" because all methods will be of this type. Recall how $t_{0:N}$ is selected adaptively during the forward pass, whereas the user chooses $s_{0:M}$. *Approximating the solution in Equation 6 with memory-complexity dictated by $s_{0:M}$ and not by $t_{0:N}$ has been previously unknown and is the main contribution of this paper.* More technically, we now introduce a method with $\mathcal{O}(N + M)$ runtime and $\mathcal{O}(M)$ memory, as opposed to the $\mathcal{O}(N + M)$ runtime and $\mathcal{O}(N + M)$ memory of existing approaches.

3.2 The Method

A sequential implementation of the ODE solver needs a Markovian prior; concretely, it needs Assumption 1:

Assumption 1. *Assume that $\mathfrak{u}$ has the Markov property and that the transition density is*

$$p(\mathfrak{u}(t + \Delta t) \mid \mathfrak{u}(t)) = N(\Phi(\Delta t)\mathfrak{u}(t), \Sigma(\Delta t)) \tag{7}$$

with known matrices $\Phi(\Delta t)$ and $\Sigma(\Delta t)$.

Assumption 1 is not restrictive if we compare with earlier work on selecting priors (Schober et al., 2019; Bosch et al., 2024b; Krämer et al., 2022a). All Gaussian processes that solve linear, time-invariant stochastic differential equations, for example, Matérn or integrated Wiener processes, satisfy it (Särkkä and Solin, 2019).

With Assumption 1, proceed in two steps: First, Section 3.2.1 derives an $\mathcal{O}(1)$-memory solver assuming two target points, $s_{0:M} = \{0, 1\}$, instead of arbitrarily many. The $\mathcal{O}(1)$ memory is critical, and novel in this context. The resulting procedure will serve as the foundation for the general case. Second, Section 3.2.2 extends the $\mathcal{O}(1)$-memory code for two to an $\mathcal{O}(M)$-memory version for many targets by calling the two-target code repeatedly and in a specific way. The resulting method then solves the problem stated in Section 2.

3.2.1 Two Target Points

For now, assume $s_{0:M} = \{0, 1\}$ and recall that we assume that the compute grid contains both $t_0 = 0$ and $t_N = 1$. A sequential, in-place algorithm for estimating

$$\begin{aligned} &p(\mathfrak{u}(0), \mathfrak{u}(1) \mid \mathfrak{r}(t_{0:N}), u_0) \\ &\quad = p(\mathfrak{u}(0) \mid \mathfrak{u}(1), \mathfrak{r}(t_{0:N}), u_0) p(\mathfrak{u}(1) \mid \mathfrak{r}(t_{0:N}), u_0). \end{aligned} \tag{8}$$

arises as follows. Marginalising over $\mathfrak{u}(t_{N-1})$ and using Markovianity turns the first term in Equation 8 into

$$\begin{aligned} &p(\mathfrak{u}(0) \mid \mathfrak{u}(1), \mathfrak{r}(t_{0:N}), u_0) \\ &\quad = \int p(\mathfrak{u}(0), \mathfrak{u}(t_{N-1}) \mid \mathfrak{u}(1), \mathfrak{r}(t_{0:N}), u_0) \mathrm{d}\mathfrak{u}(t_{N-1}) && \text{(9a)} \\ &\quad = \int \pi_0(\mathfrak{u}(t_{N-1})) \pi_1(\mathfrak{u}(t_{N-1})) \mathrm{d}\mathfrak{u}(t_{N-1}), && \text{(9b)} \end{aligned}$$

where Equation 9b uses the temporary variables

$$\begin{aligned} \pi_0(\mathfrak{u}(t_{N-1})) &:= p(\mathfrak{u}(0) \mid \mathfrak{u}(t_{N-1}), \mathfrak{r}(t_{0:N-1}), u_0) && \text{(10a)} \\ \pi_1(\mathfrak{u}(t_{N-1})) &:= p(\mathfrak{u}(t_{N-1}) \mid \mathfrak{u}(1), \mathfrak{r}(t_{0:N-1}), u_0) && \text{(10b)} \end{aligned}$$

to simplify the notation. By applying the same idea to $\pi_0(\mathfrak{u}(t_{N-1}))$, namely stepping through $\mathfrak{u}(t_{N-2})$ and using Markovianity, and repeating the approach for all remaining points in $t_{1:N-1}$ (backwards over the indices), we obtain the marginalised, backwards factorisation

$$\begin{aligned} &p(\mathfrak{u}(0) \mid \mathfrak{u}(1), \mathfrak{r}(t_{0:N}), u_0) && \text{(11)} \\ &= \int \left(\prod_{n=0}^{N-1} p(\mathfrak{u}(t_n) \mid \mathfrak{u}(t_{n+1}), \mathfrak{r}(t_{0:n}), u_0)\right) \mathrm{d}\mathfrak{u}(t_{1:N-1}). \end{aligned}$$

We make three critical observations:

First, the parametrisation of each conditional factor $p(\mathfrak{u}(t_n) \mid \mathfrak{u}(t_{n+1}), \mathfrak{r}(t_{0:n}), u_0)$ in Equation 11 depends on that of $p(\mathfrak{u}(t_n) \mid \mathfrak{r}(t_{0:n}), u_0)$ and the transition density of the prior. The latter is known due to Assumption 1. In other words, each factor in Equation 11 can be parametrised as a byproduct of the n-th step of the forward pass of the ODE solver. This insight is critical but not new (e.g., Krämer, 2024, Equation 3.15).

Second, we can compute Equation 11 in $\mathcal{O}(1)$ memory and $\mathcal{O}(N)$ runtime as long as we marginalise in the correct order, which is the following: for $n = 0$, initialise

$$p(\mathfrak{u}(0) \mid \mathfrak{u}(0), \mathfrak{r}(t_0), u_0) = \delta(\mathfrak{u}(0)). \tag{12}$$

Store the Dirac delta δ as a Gaussian with zero covariance to request the correct amount of storage for later steps. To simplify the upcoming notation, define the following transitions for $k = 0, ..., n-1$,

$$\rho(\mathfrak{u}(t_k), \mathfrak{u}(t_n)) := p(\mathfrak{u}(t_k) \mid \mathfrak{u}(t_n), \mathfrak{r}(t_{0:n}), u_0). \tag{13}$$

The start in Equation 12 parametrised $\rho(\mathfrak{u}(0), \mathfrak{u}(0))$. After initialising, iterate for $n = 0, ..., N-1$,

$$\begin{aligned}&\rho(\mathfrak{u}(0), \mathfrak{u}(t_{n+1})) \\ &\quad= \int \rho(\mathfrak{u}(0), \mathfrak{u}(t_n))\rho(\mathfrak{u}(t_n), \mathfrak{u}(t_{n+1}))\mathrm{d}\mathfrak{u}(t_n).\end{aligned} \tag{14}$$

Iterating Equation 14 consumes $\mathcal{O}(1)$ memory because storage for the parameters of ρ can be reused. This in-place behaviour is critical to an $\mathcal{O}(1)$ memory code and novel for probabilistic ODE solvers. Equation 14 holds generally; Assumption 1 implies that it is straightforward to evaluate (discussed next).

Third, the marginalisation in Equation 14 is possible in closed form because ρ is always an affine Gaussian transformation under Assumption 1. Appendix B discusses a numerically robust implementation, mimicking Krämer's (2025) fixed-point smoother. We revisit the connection to fixed-point smoothing below.

Algorithm 1 summarises the procedure. Note how it slightly deviates from the above derivation: It solves from $a \in [0, 1]$ to $b \in (a, 1]$ instead of from 0 to 1. It also contains an initial block that checks whether time-stepping is necessary. Both are important when we solve for many target points instead of two, which will become clear in Section 3.2.2. "Predict" and "step", including generating Δt, match Bosch et al. (2021); Krämer and Hennig (2024). Thus, we omit the details and refer to Krämer's (2024) tutorial instead. The marginalisation of the conditionals is new, and using it for constant-memory ODE solvers our contribution. Algorithm 1 distils creating $\mathcal{O}(1)$ memory ODE solvers into, essentially, a single line of code. This distillation is helpful since we can update published solvers without rewriting existing, sophisticated solution routines.

Algorithm 1 generalises Krämer's (2025) fixed-point smoother: Namely, if all compute-grid-points $t_{0:N}$ were known in advance, the ODE were affine, and if we followed Algorithm 1 up with one last marginalisation that yields $p(\mathfrak{u}(0) \mid \mathfrak{r}(t_{0:N}), u_0)$, we would recover Krämer's (2025) fixed-point smoother. However, our method is more general: our grid points need not be known in advance, the ODE is nonlinear, and we don't apply this final marginalisation step because we need the full $p(\mathfrak{u}(0), \mathfrak{u}(1) \mid \mathfrak{r}(t_{0:N}), u_0)$ to solve for many targets.

3.2.2 Many Target Points

With Algorithm 1 in place, solving for multiple target points $s_{0:M}$ reduces to calling it repeatedly, almost like we would do for non-probabilistic solvers. Let $\mathcal{I}_{[a,b]}$ be the restriction of a set to the interval $[a, b]$; for example, $\mathcal{I}_{[2,4]}(\{1, 2, 3, 4, 5\}) = \{2, 3, 4\}$. We use $\mathcal{I}$ to split the compute grid into chunks corresponding to the targets,

$$t_{0:N} = \bigcup_{n=0}^{N-1} \mathcal{I}_{[0,s_{n+1}]}(t_{0:N}). \tag{15}$$

Combining this separation with the Markov property leads to a sequential factorisation of the ODE solution over many target points instead of two, which reads

$$\begin{aligned}&p(\mathfrak{u}(s_{0:M}) \mid \mathfrak{u}(t_{0:N}), u_0) \\ &\quad= p(\mathfrak{u}(s_M) \mid \mathfrak{u}(t_{0:N}), u_0) \prod_{m=0}^{M-1} \xi_m.\end{aligned} \tag{16}$$

Recall $t_N = s_M$. The temporary variables

$$\xi_m := p(\mathfrak{u}(s_m) \mid \mathfrak{u}(s_{m+1}), \mathfrak{r}(\mathcal{I}_{[s_0,s_m]}(t_{0:N})), u_0) \tag{17}$$

serve notational simplicity and won't be needed after Equation 16. Sequential factorisation is important because it reduces a difficult problem (many target points) into a sequence of simpler problems (two target points). As a result, we solve the many-target problem with repeated calls to Algorithm 1 (recall it uses a and b):

1. Set $a = t = s_0$ and $p(\mathfrak{u}(a)) = p(\mathfrak{u}(s_0) \mid u_0, \mathfrak{r}(s_0))$. Initialise $p(\mathfrak{u}(a) \mid \mathfrak{u}(t)) = \delta(\mathfrak{u}(t))$ but implement this as a Gaussian with zero covariance to allocate the correct memory. Choose a Δt.
2. Assign $b = s_1$ and execute Algorithm 1. Save $p(\mathfrak{u}(s_0) \mid \mathfrak{u}(s_1), \mathcal{I}_{[s_0,s_1]}(t_{0:N}), u_0)$. Set $a = s_1$ and $b = s_2$; call Algorithm 1 with the other outputs.
3. Repeat until $s_{0:M}$ has been exhausted. Additionally store $p(\mathfrak{u}(s_M) \mid \mathfrak{r}(t_{0:N}), u_0)$ and discard everything not yet stored. This gives Equation 16.
4. Use Equation 16 to compute marginals or sample from the ODE solution (Krämer, 2024) in linear time or use it for parameter estimation (Tronarp et al., 2022).

Theorem 1. *The above procedure executes adaptive target simulation in $\mathcal{O}(N+M)$ runtime and $\mathcal{O}(M)$ memory. The returned values match those we would obtain through adaptive simulation and interpolation.*

Proof. The $\mathcal{O}(M)$ memory holds by construction because Algorithm 1 has $\mathcal{O}(1)$ memory, and we call it precisely M times. The $\mathcal{O}(N + M)$ runtime holds because every point in $t_{0:N}$ and $s_{0:M}$ is visited once.

The match with adaptive simulation is true because the only difference to adaptive simulation codes is the "merging" of the conditionals step at every stage of each loop in Algorithm 1. Tracking $p(\mathfrak{u}(t))$ and $p(\mathfrak{u}(a))$ separately is essential for adaptive target simulation not affecting the error control behaviour. □

4 Related Work

Our primary objective is to reduce the memory footprint of adaptive probabilistic solvers. Thus, the works by Bosch et al. (2021) and Schober et al. (2019) are most closely related because they introduce the current formulation of the adaptive algorithm. Unlike their technique, ours has constant memory requirements.

Our effort follows recent improvements to implementing Gaussian-process-based solvers (Krämer and Hennig,

Algorithm 1 Adaptive simulation in $\mathcal{O}(1)$ memory. "Predict" & "step" as usual. Marginalisation: Appendix B.

```
Require: $[p(\mathfrak{u}(a)), a]$ to start interpolation, $[p(\mathfrak{u}(t)), t]$ to start time-stepping, $p(\mathfrak{u}(a) \mid \mathfrak{u}(t))$ (which could be the identity), initial step-size $\Delta t$, right-hand side boundary $b \in (a, 1]$.
Ensure: $a \leq t$    ▷ Both $t \leq b$ and $b \leq t$ are possible.
if $b \in (a, t)$ then    ▷ Early exit in case no time-stepping is necessary
    $p(\mathfrak{u}(b)), p(\mathfrak{u}(a) \mid \mathfrak{u}(b)) = \text{predict}(p(\mathfrak{u}(a)), b - a)$    ▷ See Krämer's (2024) summary
    $\_\_, p(\mathfrak{u}(b) \mid \mathfrak{u}(t)) = \text{predict}(p(\mathfrak{u}(b)), t - b)$    ▷ Ignore one output
else    ▷ $b$ must be at least as large as $t$, so we start time-stepping
    $t_{\text{prev}} \leftarrow a$    ▷ Carry two intervals: $(t_{\text{prev}}, t]$ & $(t, t_{\text{next}}]$
    $p(\mathfrak{u}(a) \mid \mathfrak{u}(t_{\text{prev}}) \leftarrow N(\mathfrak{u}(t_{\text{prev}}), 0)$
    while $t < b$ do
        $p(\mathfrak{u}(t_{\text{next}})), p(\mathfrak{u}(t) \mid \mathfrak{u}(t_{\text{next}})), \Delta t = \text{step}(p(\mathfrak{u}(t)), \Delta t)$    ▷ See Krämer's (2024) summary
        $p(\mathfrak{u}(a) \mid \mathfrak{u}(t)) \leftarrow \int p(\mathfrak{u}(a) \mid \mathfrak{u}(t_{\text{prev}}))p(\mathfrak{u}(t_{\text{prev}}) \mid \mathfrak{u}(t))d\mathfrak{u}(t_{\text{prev}})$    ▷ new!
        $(t_{\text{prev}}, t) \leftarrow (t, t_{\text{next}})$
    $p(\mathfrak{u}(b)), p(\mathfrak{u}(t_{\text{prev}}) \mid \mathfrak{u}(b)) = \text{predict}(p(\mathfrak{u}(t_{\text{prev}})), b - t_{\text{prev}})$
    $p(\mathfrak{u}(a) \mid \mathfrak{u}(b)) \leftarrow \int p(\mathfrak{u}(a) \mid \mathfrak{u}(t_{\text{prev}}))p(\mathfrak{u}(t_{\text{prev}}) \mid \mathfrak{u}(b))d\mathfrak{u}(t_{\text{prev}})$    ▷ new!
    $\_\_, p(\mathfrak{u}(b) \mid \mathfrak{u}(t)) = \text{predict}(p(\mathfrak{u}(b)), t - b)$
return $\{p(\mathfrak{u}(a) \mid \mathfrak{u}(b)), p(\mathfrak{u}(b))\}$, $\{p(\mathfrak{u}(b) \mid \mathfrak{u}(t)), p(\mathfrak{u}(t))\}$, $\Delta t$
```

2024; Krämer et al., 2022a; Bosch et al., 2024a; Krämer and Hennig, 2021) by reformulating existing algorithms in a way that makes them more efficient or robust but without affecting their return values. We are the first to target memory efficiency and extend Krämer's (2025) fixed-point smoother to get there (Section 3.2.1).

The articles by Bosch et al. (2022) and Bosch et al. (2024b), as well as Krämer et al.'s (2022b) partial-differential-equation algorithm are related as work on (broadly) ODE methods, but different because they build new solvers instead of making existing ones more efficient as our work does. Studies like those by Tronarp et al. (2022); Kersting et al. (2020); Beck et al. (2024); Wu and Lysy (2024) also relate through the ODE-solver connection. However, those three works focus on applications, which is a different objective than ours. That said, our method can be used for all of the above.

Adaptive target simulation loosely relates to "checkpointing" for reverse-mode derivatives of ordinary differential equations (e.g. Griewank and Walther, 2000) in the sense that both approaches attempt to reduce the memory footprint of a forward-backwards pass where the backward pass depends on all intermediate values of the forward pass. However, beyond this similarity, the relation of our work to checkpointing is minimal.

5 Experiments

Theorem 1 proves that Algorithm 1 solves the problem stated in Section 2, by implementing adaptive probabilistic ODE solvers in constant memory. Next, a series of experiments demonstrates how these promises translate to better performance in memory and runtime.

Hardware & software setup All experiments run on the CPU of a consumer-level laptop with 8 GB of RAM. We use a small machine to demonstrate the memory efficiency of our tools: For example, Section 5.2 uses this 8 GB of RAM to run a simulation that would require many hundreds of GB of memory with existing methods. Everything is implemented in JAX (Bradbury et al., 2018) and open-sourced under this[5] link. The simulations use double precision instead of JAX's single precision default because tolerances of 10^{-8} and smaller are common. Runge–Kutta codes are imported from Diffrax (Kidger, 2021), not from SciPy (Virtanen et al., 2020) because prior benchmarks have demonstrated how JAX implementations of probabilistic and non-probabilistic solvers are usually faster than SciPy (Krämer, 2024), likely due to JIT-compilation. The latter plays a part in our experiments, too.

5.1 Memory (and Runtime) vs. Accuracy

5.1.1 Motivation

Section 3 proved drastic memory savings, and the introduction announced substantial runtime improvements. But do both manifest in actual simulations? The first experiment illustrates what users can expect by switching to our fixed-point-smoother-based code. More specifically, we choose a popular ODE benchmark and compare the memory and runtime of adaptive target simulation with that of the combination of adaptive simulation and interpolation. The results of such a demonstration indicate which gains to expect from our proposals.

5.1.2 Setup

We choose a simple but popular ODE problem, the rigid body problem (Hairer et al., 1993, p. 244), which describes the rotation of a rigid body in three-dimensional, principal, orthogonal coordinates. We chose it because existing ODE solver benchmarks do (Bosch et al., 2024a; Rackauckas and Nie, 2017, 2019). This first experiment measures the memory demands, runtime, and accumulated root-mean-square error over $M = 5$ uniformly spaced points for different tolerances. The choice of M does not affect the results as long as it exceeds $M = 1$

[5] `https://github.com/pnkraemer/probdiffeq`

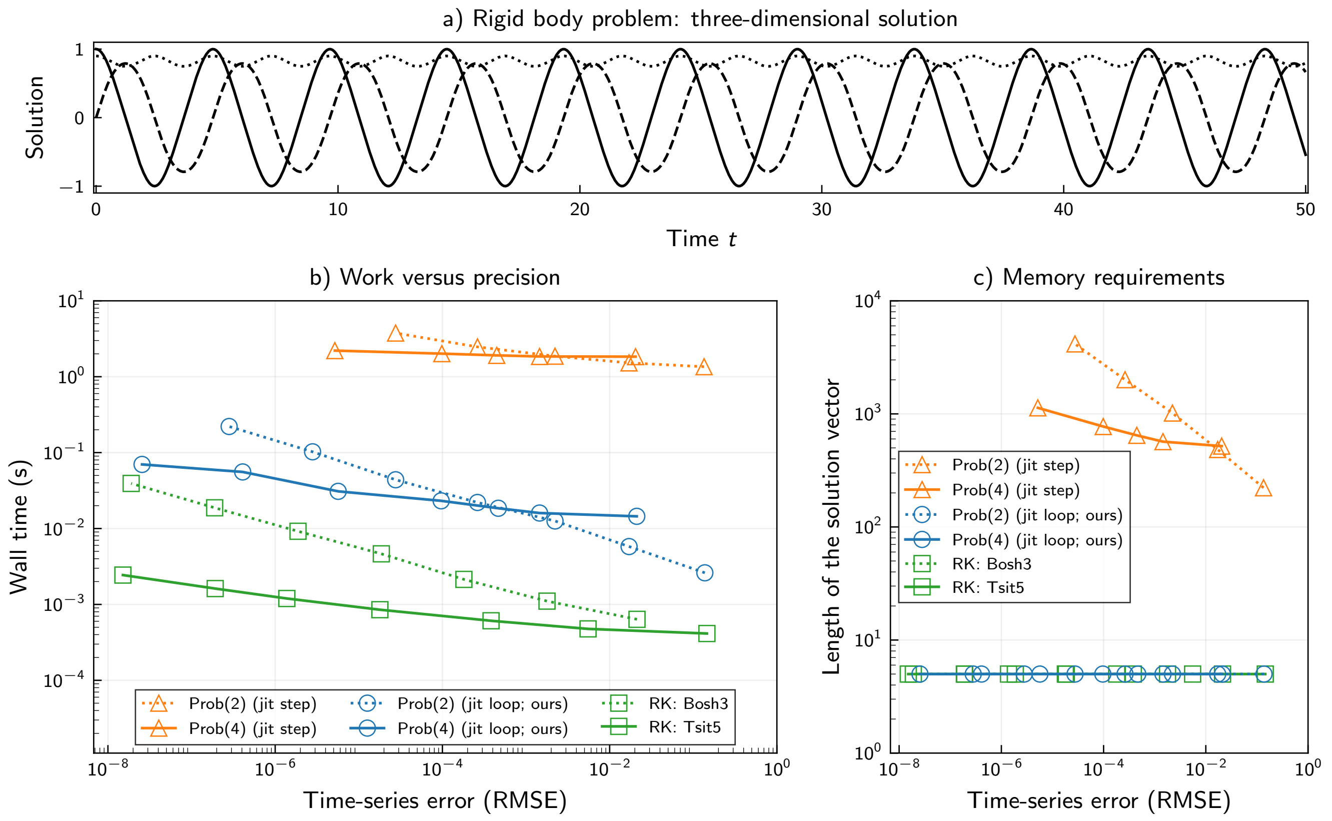

Figure 2: *Memory (and Runtime) vs. Accuracy:* Solution of the rigid-body problem (a). Wall time (b) and memory footprint (c) versus root-mean-square error for probabilistic solvers via adaptive target simulation (orange triangles) and adaptive simulation (blue circles), as well as Runge–Kutta methods (green squares). The probabilistic solvers use $L = 2$ and $L = 4$ derivatives, and Diffrax provides the Runge–Kutta methods. We choose "Bosh3" and "Tsit5" since their error decays similarly to that of the selected probabilistic solvers.

and the target points cover the time domain evenly. Three codes are evaluated:

1. Probabilistic solver: adaptive target simulation (ours; end-to-end JIT'able, constant memory)

2. Runge–Kutta methods: adaptive target simulation (end-to-end JIT'able, constant memory)

3. Probabilistic solver: adaptive simulation (can only JIT single steps due to unpredictable memory)

Appendix C lists the full setup. In this benchmark, we want fast solvers with constantly low memory requirements. The role of the Runge–Kutta methods is to provide context on the runtimes and memory requirements. We don't expect our solvers to be faster than them because we propagate (Cholesky factors of) covariance matrices, and this built-in uncertainty quantification is not free; refer to Krämer (2024); Krämer and Hennig (2024); Bosch et al. (2021, 2022, 2024b) for more. Still, we hope their performance will be similar.

5.1.3 Analysis

The results in Figure 2 express how, as expected, the memory demands of adaptive target simulation are constant. Solvers that rely on adaptive simulation instead of adaptive target simulation store thousands of grid points for high-accuracy solutions. Further, the ability to use just-in-time-compilation for adaptive target simulation substantially affects runtime: the probabilistic solvers' runtimes are closer to their Runge–Kutta relatives than to their implementations without adaptive target simulation. For example, the $L = 4$ solver via adaptive target simulation needs less than 0.1 seconds to reach accuracy $\approx 10^{-8}$, whereas not using adaptive target simulation takes multiple seconds to reach error 10^{-1}. The remaining gap between probabilistic algorithms and Runge–Kutta methods is due to the costs of built-in uncertainty quantification, as anticipated above. In summary, this experiment contrasts the efficiencies of adaptive target simulation and adaptive simulation.

5.2 Overcoming Memory Limitations

5.2.1 Motivation

The experiment in Section 5.1 shows how adaptive target simulation is more memory efficient than adaptive simulation in combination with interpolation. Next, we show that this efficiency unlocks large-scale simulations through substantial memory savings. We solve a high-dimensional differential equation with high accuracy and investigate if our new implementation pushes the boundary of feasibility to new heights. The results will provide intuition for which memory-challenged ODE problems were previously out of reach but can now be addressed probabilistically.

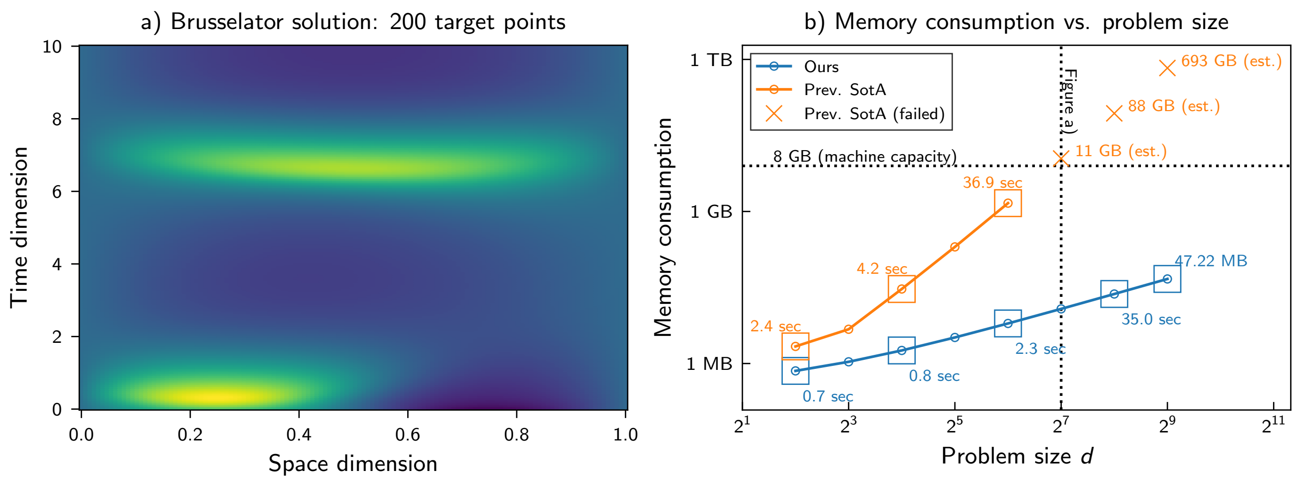

Figure 3: *Overcoming Memory Limitations:* Left: Solution of the Brusselator problem. Right: The memory demands of the previous state-of-the-art (adaptive simulation) exceed the machine capacity somewhere between $d = 2^6 = 64$ and $d = 2^7 = 128$. Adaptive target simulation on 200 points is far more memory efficient: at $d = 512$, it consumes 47 MB, in contrast to adaptive simulation's (estimated) 693 GB. Runtime is never a bottleneck.

5.2.2 Setup

Not all ODE problems are limited by memory constraints, but many high-dimensional systems over long time intervals are. As an example of such a system, we solve the Brusselator problem (Prigogine and Lefever, 1968), a time-dependent partial differential equation. We discretise spatial derivatives with centred differences on an increasing number of points d. The larger d, the higher-dimensional the ODE, and ultimately, the harder the Brusselator becomes to solve (Wanner and Hairer, 1996). We compare adaptive target simulation and adaptive simulation for increasing $d = \{2, 4, 8, ..., 512\}$, always using 200 target points, tolerance 10^{-8}, $L = 4$ derivatives, and zeroth-order linearisation (even though the problem is stiff; why? See Appendix D). We already know from Sections 3 and 5.1 that adaptive target simulation is more memory efficient than adaptive simulation; this experiment investigates to which extent previously impossible simulations are now feasible. Recall that all benchmarks use the CPU of a consumer-level laptop with 8 GB of RAM.

5.2.3 Analysis

The results in Figure 3 show two phenomena: (i) runtime is never a bottleneck, not even on the finest resolutions, but memory is; and (ii) memory constraints limit adaptive simulation to resolutions below $d = 64$ points in the space dimension, whereas adaptive target simulation can go up to 512 points and likely higher. Concretely, for resolution $d = 512$, the adaptive simulation would require 693 GB of RAM, which is enormous for a single ODE simulation. Our code takes up less than 50 MB without sacrificing accuracy or runtime. This contrast shows how a simulation that was previously impossible, even on large machines, can now be run in seconds on a small laptop. Solving the Brusselator accurately is no longer out of reach.

5.3 Further Experiments

Our paper's primary objective is to reduce the memory footprint of probabilistic solvers; its secondary objective is efficient JAX code. We are less interested in runtime differences, mainly because our contribution should not affect runtime beyond enabling JIT compilation. Still, in the absence of memory limitations, users might care more about runtime than about memory, and there exist probabilistic numerics libraries that don't use JAX (Wenger et al., 2021; Bosch, 2024). Appendix E contains more studies focusing on runtime to give insight into what happens in these two scenarios.

– Appendix E.1 repeats the experiment in Section 5.1 for a more challenging ODE, the Pleiades problem.

– Appendix E.2 discusses the runtime if we're not memory- or JIT-compilation-constrained, which suggests using our method in NumPy- or Julia-based libraries (Wenger et al., 2021; Bosch, 2024).

Since memory is our main objective, these two experiments are relegated to the appendices. Sections 5.1 and 5.2 have already shown the superior memory requirements of adaptive target simulation.

6 Discussion

6.1 Limitations

Even though our method provably advances the memory efficiency of probabilistic solvers, it should be enjoyed with two caveats. First, without memory limitations *and* outside of JAX, ODEs can (and likely should) still be solved by combining adaptive simulation and interpolation, not with our method; Appendix E.2 discusses the nuances of this recommendation. Second, even though our method technically applies to learning ODEs via the methods by Tronarp et al. (2022); Beck et al. (2024); Wu and Lysy (2024), marginal-likelihood optimisation with adaptive steps is currently unexplored. However, we now have the tools to approach this research for decently sized problems.

6.2 Conclusion

This work explained how to implement the first adaptive probabilistic solver whose memory demands are fixed and chosen by the user before the simulation. To get there, we combine Markovian priors (which are typical for probabilistic ODE solvers) with adaptive step sizes and the linear algebra of fixed-point smoothing. The experiments have demonstrated how the memory demands of a Brusselator-simulation reduce from almost 700 GB to less than 50 MB without affecting the accuracy (Section 5.2) and how JAX-based implementations gain orders of magnitude of runtime improvements (Section 5.1). For the future, this means that the utility of probabilistic ODE solvers once again continues to approach that of their non-probabilistic counterparts, thereby paving the way for probabilistic numerical methods in scientific machine learning.

Acknowledgements

This work was supported by a research grant (42062) from VILLUM FONDEN. The work was partly funded by the Novo Nordisk Foundation through the Center for Basic Machine Learning Research in Life Science (NNF20OC0062606). This project received funding from the European Research Council (ERC) under the European Union's Horizon programme (grant agreement 101125993).

References

J. Beck, N. Bosch, M. Deistler, K. L. Kadhim, J. H. Macke, P. Hennig, and P. Berens. Diffusion tempering improves parameter estimation with probabilistic integrators for ordinary differential equations. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=43HZG9zwaj.

J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

P. Bogacki and L. F. Shampine. A 3(2) pair of Runge–Kutta formulas. *Applied Mathematics Letters*, 2(4): 321–325, 1989.

N. Bosch. ProbNumDiffEq.jl: Probabilistic numerical solvers for ordinary differential equations in Julia. *Journal of Open Source Software*, 9(101):7048, 2024. doi: 10.21105/joss.07048. URL https://doi.org/10.21105/joss.07048.

N. Bosch, P. Hennig, and F. Tronarp. Calibrated adaptive probabilistic ODE solvers. In *International Conference on Artificial Intelligence and Statistics*, pages 3466–3474. PMLR, 2021.

N. Bosch, F. Tronarp, and P. Hennig. Pick-and-mix information operators for probabilistic ODE solvers. In *International Conference on Artificial Intelligence and Statistics*, pages 10015–10027. PMLR, 2022.

N. Bosch, A. Corenflos, F. Yaghoobi, F. Tronarp, P. Hennig, and S. Särkkä. Parallel-in-time probabilistic numerical ODE solvers. *Journal of Machine Learning Research*, (206):1–27, 2024a.

N. Bosch, P. Hennig, and F. Tronarp. Probabilistic exponential integrators. *Advances in Neural Information Processing Systems*, 36, 2024b.

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. URL http://github.com/jax-ml/jax.

J. Cockayne, C. J. Oates, T. J. Sullivan, and M. Girolami. Bayesian probabilistic numerical methods. *SIAM Review*, 61(4):756–789, 2019.

A. Griewank and A. Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26 (1):19–45, 2000.

K. Gustafsson, M. Lundh, and G. Söderlind. A PI stepsize control for the numerical solution of ordinary differential equations. *BIT Numerical Mathematics*, 28:270–287, 1988.

E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I, Nonstiff Problems*. Springer, 1993.

C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

P. Hennig, M. A. Osborne, and H. P. Kersting. *Probabilistic Numerics: Computation as Machine Learning*. Cambridge University Press, 2022.

W. O. Kermack and A. G. McKendrick. A contribution to the mathematical theory of epidemics. *Proceedings of the Royal Society of London. Series A*, 115(772): 700–721, 1927.

H. Kersting, N. Krämer, M. Schiegg, C. Daniel, M. Tiemann, and P. Hennig. Differentiable likelihoods for fast inversion of "likelihood-free" dynamical systems. In *International Conference on Machine Learning*, pages 5198–5208. PMLR, 2020.

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

N. Krämer. Numerically robust fixed-point smoothing without state augmentation. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=LVQ8BEL5n3.

N. Krämer and P. Hennig. Linear-time probabilistic solution of boundary value problems. *Advances in Neural Information Processing Systems*, 34:11160–11171, 2021.

N. Krämer and P. Hennig. Stable implementation of probabilistic ODE solvers. *Journal of Machine Learning Research*, 25(111):1–29, 2024.

N. Krämer, N. Bosch, J. Schmidt, and P. Hennig. Probabilistic ODE solutions in millions of dimensions. In *International Conference on Machine Learning*, pages 11634–11649. PMLR, 2022a.

N. Krämer, J. Schmidt, and P. Hennig. Probabilistic numerical method of lines for time-dependent partial differential equations. In *International Conference on Artificial Intelligence and Statistics*, pages 625–639. PMLR, 2022b.

P. N. Krämer. *Implementing Probabilistic Numerical Solvers for Differential Equations.* PhD thesis, Universität Tübingen, 2024.

A. Lahr, F. Tronarp, N. Bosch, J. Schmidt, P. Hennig, and M. N. Zeilinger. Probabilistic ODE solvers for integration error-aware numerical optimal control. *Proceedings of Machine Learning Research*, 424, 2024.

C. J. Oates, J. Cockayne, R. G. Aykroyd, and M. Girolami. Bayesian probabilistic numerical methods in time-dependent state estimation for industrial hydrocyclone equipment. *Journal of the American Statistical Association*, 2019.

J. Oesterle, N. Krämer, P. Hennig, and P. Berens. Probabilistic solvers enable a straight-forward exploration of numerical uncertainty in neuroscience models. *Journal of Computational Neuroscience*, 50(4):485–503, 2022.

L. Petzold. Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. *SIAM Journal on Scientific and Statistical Computing*, 4(1):136–148, 1983.

I. Prigogine and R. Lefever. Symmetry breaking instabilities in dissipative systems. ii. *The Journal of Chemical Physics*, 48(4):1695–1700, 1968.

P. J. Prince and J. R. Dormand. High order embedded Runge–Kutta formulae. *Journal of Computational and Applied Mathematics*, 7(1):67–75, 1981.

C. Rackauckas and Q. Nie. DifferentialEquations.jl – a performant and feature-rich ecosystem for solving differential equations in Julia. *The Journal of Open Research Software*, 5(1), 2017. doi: 10.5334/jors.151. URL https://app.dimensions.ai/details/publication/pub.1085583166andhttp://openresearchsoftware.metajnl.com/articles/10.5334/jors.151/galley/245/download/.

C. Rackauckas and Q. Nie. Confederated modular differential equation APIs for accelerated algorithm development and benchmarking. *Advances in Engineering Software*, 132:1–6, 2019.

S. Särkkä and A. Solin. *Applied Stochastic Differential Equations.* Cambridge University Press, 2019.

J. Schmidt, N. Krämer, and P. Hennig. A probabilistic state space model for joint inference from differential equations and data. *Advances in Neural Information Processing Systems*, 34:12374–12385, 2021.

M. Schober, S. Särkkä, and P. Hennig. A probabilistic model for the numerical solution of initial value problems. *Statistics and Computing*, 29(1):99–122, 2019.

L. F. Shampine. Some practical Runge-Kutta formulas. *Mathematics of Computation*, 46(173):135–150, 1986. doi: https://doi.org/10.2307/2008219.

F. Tronarp, H. Kersting, S. Särkkä, and P. Hennig. Probabilistic solutions to ordinary differential equations as nonlinear Bayesian filtering: A new perspective. *Statistics and Computing*, 29:1297–1315, 2019.

F. Tronarp, N. Bosch, and P. Hennig. Fenrir: Physics-enhanced regression for initial value problems. In *International Conference on Machine Learning*, pages 21776–21794. PMLR, 2022.

C. Tsitouras. Runge–Kutta pairs of order 5(4) satisfying only the first column simplifying assumption. *Computers & Mathematics with Applications*, 62(2): 770–775, 2011.

B. Van der Pol. Theory of the amplitude of free and forced triode vibrations. *Radio Review*, 1:701–710, 1920.

P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, et al. SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3):261–272, 2020.

G. Wanner and E. Hairer. *Solving Ordinary Differential Equations II*, volume 375. Springer Berlin Heidelberg New York, 1996.

J. Wenger, N. Krämer, M. Pförtner, J. Schmidt, N. Bosch, N. Effenberger, J. Zenn, A. Gessner, T. Karvonen, F.-X. Briol, et al. ProbNum: Probabilistic numerics in Python. *Preprint on ArXiv:2112.02100*, 2021.

C. K. Williams and C. E. Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT Press Cambridge, MA, 2006.

M. Wu and M. Lysy. Data-adaptive probabilistic likelihood approximation for ordinary differential equations. In *International Conference on Artificial Intelligence and Statistics*, pages 1018–1026. PMLR, 2024.

Overview

The following supplementary materials provide additional context for the results in the main paper. This includes additional information on experiment setups (Appendices A, C and D), implementation details for our method (Appendix B), and two more experiments (Appendices E.1 and E.2). Like in the main paper, all experiments use the CPU of a laptop with 8 GB of RAM, implement code in JAX (Bradbury et al., 2018), and use double precision because tolerances below the floating-point accuracy of single precision are typical.

A Complete Setup: Figure 1

As a differential equation, we choose Van der Pol's (1920) system, using Wanner and Hairer's (1996) parametrisation, $u''(t) = 10^3(u'(1-u^2) - u)$, $u_0 = 2, u_0' = 0$, from $t = 0$ to $t = 6.3$. This parametrisation is common for benchmarks (e.g., Bosch et al., 2021; Krämer, 2024).

To assemble a solver, we use $L = 4$ derivatives, first-order linearisation (see Tronarp et al. (2019) for zeroth- versus first-order linearisation), a time-varying output-scale (calibrated during the forward pass; Schober et al. (2019); Bosch et al. (2021)) and solve the ODE as is without transforming it into an ODE that only involves first derivatives (Bosch et al., 2022). We don't factorise covariances (Krämer et al., 2022a) because the problem is scalar-valued. Taylor-mode differentiation yields the initial state (Krämer and Hennig, 2024).

For time-stepping, we use tolerance 10^{-3} (both absolute and relative) in combination with proportional-integral-control (Gustafsson et al., 1988) and Schober et al.'s (2019) error estimate. Figure 1 shows step sizes from adaptive and fixed grids. The adaptive grid emerges from adaptive simulation, which means storing all intermediate steps of the forward pass (as opposed to adaptive target simulation, an implementation of which this paper contributes for probabilistic solvers). This adaptive grid and the corresponding ODE solution are in Figure 1. Here is how we generate the labels:

Assemble two non-adaptive grids by matching the number of steps and the smallest step size, respectively. Then, run a fixed-step solver on all three grids and measure the wall times. We use the same fixed-step solver for all grids to make the times comparable by factoring out JIT compilation as much as possible; see also Appendix E.2. The runtimes yield the labels in Figure 1.

The results are plotted and discussed in Figure 1 in the main paper.

B Numerically Robust Marginalisation of Conditionals

For two Gaussian conditionals $\rho(x, y) = N(x; Ay + a, B)$ and $\rho(y, z) = N(y; Cz + c, D)$, the marginalised conditional $\rho(x, z)$ from Equation 14 is

$$\begin{aligned}\rho(x, z) &= \int \rho(x, y)\rho(y, z)\mathrm{d}y &(18a)\\ &= N(x; ACz + Ac + a, ADA^\top + B) &(18b)\end{aligned}$$

which results from the rules of manipulating Gaussian random variables. To ensure numerical robustness, we must be careful with $ADA^\top + B$ to ensure symmetry and positive definiteness of all covariance matrices.

To this end, let $\sqrt{D}$ and $\sqrt{B}$ be the Cholesky factors of D and B. Then, we can compute the Cholesky factor of $ADA^\top + B$ only from $\sqrt{D}$ and $\sqrt{B}$, without ever assembling D or B:

$$\text{(Thin-)QR-decompose} \quad QR = \begin{pmatrix} \sqrt{D}A^\top \\ \sqrt{B}^\top \end{pmatrix}, \qquad (19a)$$

$$\text{then set} \quad \sqrt{ADA^\top + B} = R^\top. \qquad (19b)$$

To see that this is a valid assignment, observe that $R^\top$ is lower triangular, and

$$\begin{aligned}ADA^\top + B &= \begin{pmatrix} A\sqrt{D} & \sqrt{B} \end{pmatrix} \begin{pmatrix} \sqrt{D}A^\top \\ \sqrt{B}^\top \end{pmatrix} &(20a)\\ &= R^\top Q^\top Q R &(20b)\\ &= R^\top R. &(20c)\end{aligned}$$

As a result, $R^\top$ must be the Cholesky factor of $ADA^\top + B$. The same QR decomposition is also used during the forward pass of the probabilistic solver; see Krämer (2024); Krämer and Hennig (2024), and the linear algebra of Krämer's (2025) numerically robust fixed-point smoother.

C Complete Setup: Figure 2

As a benchmark ODE, we use the rigid-body problem (Hairer et al., 1993, page 244)

$$\begin{aligned}\frac{d}{dt}u_1(t) &= -2u_2(t)u_3(t), &(21a)\\ \frac{d}{dt}u_2(t) &= \frac{5}{4}u_1(t)u_3(t), &(21b)\\ \frac{d}{dt}u_3(t) &= -\frac{1}{2}u_1(t)u_2(t) &(21c)\end{aligned}$$

from $t = 0$ to $t = 50$, with the initial values $u_1(0) = 1$, $u_2(0) = 0$, and $u_3(0) = 0.9$. We compute a solution with SciPy's implementation of LSODA (Petzold, 1983) using tolerance 10^{-13} and plot it in Figure 2.

For all solvers, we choose relative tolerances

$$\{10^{-3}, 10^{-4}, ..., 10^{-9}, 10^{-10}\}, \qquad (22)$$

discarding the high-precision third of this list for all solvers that don't use adaptive target simulation (because that would take too long to execute). The absolute tolerance is always $1,000\times$ smaller than the relative one.

The target grid consists of five equispaced points in the time domain.

For probabilistic solvers, we use zeroth-order linearisation, proportional-integral control, isotropic covariance factorisation, and a time-varying output scale. We initialise with Taylor-mode differentiation. We choose $L = 2$ and $L = 4$. We contrast adaptive target simulation with the combination of adaptive simulation and interpolation. Adaptive target simulation can JIT-compile the full forward pass, but adaptive simulation can only compile a single solver step because the number of steps is unknown in advance.

For Runge–Kutta methods, we choose Bosh3 (Bogacki and Shampine, 1989) and Tsit5 (Tsitouras, 2011) as offered by Diffrax (Kidger, 2021) because they roughly match the error decay rate of the probabilistic solvers. A high-accuracy reference solution comes from Dopri8 (Prince and Dormand, 1981) with tolerance 10^{-15}.

As an error metric, we use the root-mean-square error. We always choose the best of three independent runs for the wall-time information because this choice estimates the method's efficiency without "machine background noise". Recall that all experiments run on the CPU of a small laptop. The solution vector is the number of time steps in the computed solution. By construction, it is constant for adaptive target simulation.

The results are plotted and discussed in Figure 2 in the main paper.

D Complete Setup: Figure 3

As a test problem, we solve the Brusselator problem (Prigogine and Lefever, 1968)

$$u(t,x) = 1 + u(t,x)^2 v(t,x) - 4u + \frac{1}{50}\Delta u(t,x) \tag{23a}$$

$$v(t,x) = 3u(t,x) - u(t,x)^2 v(t,x) + \frac{1}{50}\Delta v(t,x) \tag{23b}$$

from $t = 0$ to $t = 10$, with $x \in [0,1]$, and subject to the initial conditions

$$u(0,x) = 1 + \sin(2\pi x), \quad v(0,x) = 3. \tag{24}$$

We discretise the Laplacian with centred differences on d points, and vary $d \in \{2^1, 2^2, ..., 2^9\}$. The larger d, the higher-dimensional the ODE, which is why, even for adaptive target simulation, the memory demands are not constant. But also, the larger d, the stiffer the equation (Wanner and Hairer, 1996).

For all simulations, we use $L = 4$ derivatives, tolerance 10^{-8}, proportional-integral control (Gustafsson et al., 1988), and a time-varying output scale (Schober et al., 2019; Bosch et al., 2021). As always, we initialise with Taylor-mode differentiation (Krämer and Hennig, 2024).

We use zeroth-order linearisation (Tronarp et al., 2019) with an isotropic covariance factorisation (Krämer et al., 2022a) for all solvers. Using zeroth-order linearisation may be unexpected because the Brusselator is stiff, and zeroth-order linearisation is not recommended for stiff systems (first-order linearisation should be used instead; Bosch et al. (2024b)). However, the Brusselator is also high-dimensional, and in their current implementation, first-order methods cost cubically in the ODE dimension, whereas zeroth-order methods cost linearly (Krämer et al., 2022a). Put differently, we must choose between doing too many steps (by selecting a factorised zeroth-order method) and cubic-complexity linear algebra for a high-dimensional ODE. In practice, we found the former to be significantly faster and more memory-efficient, which is why we chose it.

We compare two modes of solvers: (i) adaptive target simulation using 200 equispaced points and (ii) adaptive simulation. In order to not ask the machine for more memory than it can provide, we estimate the memory demands of adaptive simulation as follows before running any code: (i) simulate the terminal value of the ODE, which costs $\mathcal{O}(1)$ memory, and track the number of steps the solver would take. Then, (ii) multiply this number of steps by the memory demands on the initialised solver state to gauge the total memory required. If this total memory is less than 4 GB (we leave some space for background processes like code editors), run the simulation with both solvers. If it is more than 4 GB, only run adaptive target simulation but store the predicted memory demands of adaptive simulation regardless. We measure the wall time of every such simulation.

The results are plotted and discussed in Figure 3 in the main paper.

E Additional Experiments

E.1 Runtime vs. Accuracy: Pleiades

E.1.1 Motivation

The rigid-body experiment in Section 5.1 demonstrates how the combination of memory improvements and JIT-compilation accelerate probabilistic ODE solvers and bring their efficiency close to their non-probabilistic counterparts. This following experiment repeats the setup from Section 5.1 but uses another differential equation. Concretely, it investigates whether the performance-similarity between probabilistic and non-probabilistic adaptive target simulation carries over to more challenging ODE problems. We consider memory questions to be answered by Sections 5.1 and 5.2 and only focus on runtime now. Like in Section 5.1, we don't expect probabilistic solvers to be faster than Runge–Kutta methods in terms of work per attained precision, but we hope their speed is similar.

E.1.2 Setup

In this experiment, we solve the Pleiades problem from celestial mechanics (Hairer et al., 1993),

$$u''(t) = f(u(t)), \quad u(0) = u_0, \tag{25}$$

with f and u_0 as on page 245 in Hairer et al.'s (1993) book. The Pleiades problem describes the motion of seven stars, formulated as a 14-dimensional differential equation involving second time-derivatives. When selecting solvers, we mimic the setup from Section 5.1 but drop adaptive simulation because we now know it will be far less efficient. Instead, we add two more solvers in adaptive-target-simulation mode. Concretely:

We solve Equation 25 from $t = 0$ to $t = 3$. Equation 25 involves second-time derivatives, and whereas probabilis-

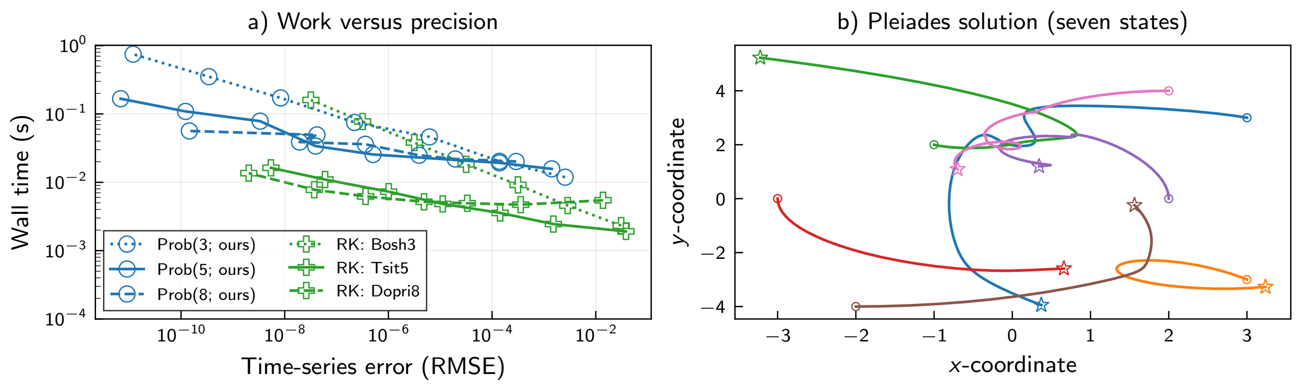

Figure 4: *Runtime vs. Accuracy: Pleiades:* Work-precision in a); ODE solution in b). The probabilistic solvers (blue) are almost as fast as their non-probabilistic counterparts (green). Context: We expect them to be slightly slower since they use covariance matrices (this built-in uncertainty quantification is not free).

tic methods can solve these types of problems natively (Bosch et al., 2022), Diffrax does not offer Runge–Kutta methods that can do the same. Therefore, the Runge–Kutta methods transform the problem into one involving only first time-derivatives before simulating.

For Runge–Kutta methods, we vary the relative tolerances from $\{10^{-3}, ..., 10^{-10}\}$. For the probabilistic solver, we vary from 10^{-2} to 10^{-9} because we found this leads to similar accuracies. For work-precision diagrams, the tolerance need not match between methods as long as the resulting work-precision ratios are comparable. The absolute tolerances are always $10^3\times$ smaller than the relative tolerances, which is typical; for example, it's the default in Scipy's ODE-solver (Virtanen et al., 2020).

We save the solution at 50 equispaced points in the time domain. We consider the wall time (in seconds, best of three runs to remove "background machine noise" as much as possible) and the root-mean-square error.

Probabilistic solvers always use zeroth-order linearisation (Tronarp et al., 2019), an isotropic covariance factorisation (Krämer et al., 2022a), time-varying output scales (Schober et al., 2019), and initialise via Taylor-mode differentiation (Krämer and Hennig, 2024).

We choose $L \in \{3, 5, 8\}$ derivatives. Runge–Kutta methods are chosen to match the probabilistic solvers' error decay rate; like before, we include Bosh3 (Bogacki and Shampine, 1989), Tsit5 (Tsitouras, 2011), and Dopri8 (Prince and Dormand, 1981) via Diffrax (Kidger, 2021).

We compute a reference solution with Dopri5 (Shampine, 1986) and tolerance 10^{-15}, to evaluate approximation errors and for visualisation.

E.1.3 Analysis

The results in Figure 4 show that, once again, the probabilistic solvers are similarly efficient to Runge–Kutta methods but slightly slower. In comparison to prior work (e.g. Krämer and Hennig, 2024), being only slightly slower is a success. Further, the results match Krämer's (2024) results for terminal-value simulation, which is reassuring. In general, this experiment underlines the value of adaptive target simulation for computing time-series benchmarks for probabilistic solvers.

E.2 Runtime on a Small Problem and Without JIT-Limitations

E.2.1 Motivation

The constant memory requirements of adaptive target simulation have been studied by Sections 5.1 and 5.2. The conclusions from these experiments do not depend on the programming environment. However, the runtime (as in, wall time) gains from Figures 2 and 3 were specific to JAX because JAX cannot compile a function whose memory demands are unknown at compilation time. This experiment eliminates JAX-specific considerations from the wall-time comparison. In other words, "what if JAX could compile adaptive simulation despite its unpredictable memory requirements?" The results of this experiment are instructive for probabilistic solver codes that are implemented in NumPy (Harris et al., 2020) or Julia (Bezanson et al., 2017); namely, those by Bosch (2024); Wenger et al. (2021). We investigate in which case they benefit from our implementation of adaptive target simulation *beyond* memory limitations, which is a scenario outside our primary objectives (memory-challenging equations or JAX).

E.2.2 Setup

This experiment investigates the solvers' performances without JAX's compilation constraints. However, we would still like to use our JAX code because we don't want to reimplement ODE solvers for this one experiment. Using JAX code for a JAX-independent benchmark necessitates cheating a little in the setup. However, we only cheat to make competing algorithms more efficient than they would be otherwise, and we leave our implementation as is. Concretely:

All methods solve the restricted three-body problem (Hairer et al., 1993, page 129), another ODE involving second derivatives (after Appendix E.1). We replicate the exact setup from Appendix E.1 with two exceptions: we don't include Runge–Kutta methods because they are irrelevant to this benchmark, and all probabilistic solvers use $L = 4$ derivatives.

Unlike the previous experiments, we don't measure work-versus-precision for ODE simulation. Instead, to empha-

Table 1: Runtime on the three-body problem: Adaptive simulation (AS; but by "cheating" JIT-compilation, see the explanation above) versus adaptive target simulation (ATS; our method). Lower is better. Winners in bold.

No. Samples	Tolerance	No. steps	Time (s): AS	Time (s): ATS (ours)
5	10^{-4}	448	**0.007**	0.015
5	10^{-7}	2,570	**0.037**	0.065
5	10^{-10}	14,469	**0.215**	0.347
50	10^{-4}	448	**0.014**	0.015
50	10^{-7}	2,570	0.098	**0.076**
50	10^{-10}	14,469	0.536	**0.374**
500	10^{-4}	448	0.127	**0.032**
500	10^{-7}	2,570	0.700	**0.080**
500	10^{-10}	14,469	3.851	**0.393**

sise using the probabilistic ODE solution for increasingly complex tasks, we compute an increasing number of samples from the ODE posterior (Krämer, 2024). Few samples imply "cheap processing" of the ODE solution, and many samples will represent "expensive processing". We expect that adaptive target simulation will be more efficient than adaptive simulation for "expensive processing" (even with cheating the JIT compilation), and this experiment investigates just how expensive the processing needs to be for this to be true. We compare the following two routines.

A. Adaptive target simulation (ours, no cheat)

1. Solve adaptively using 50 target points
2. Compute K joint samples from the posterior

B. Adaptive simulation (cheat):

1. Solve adaptively storing the full grid
2. Augment the full grid with 50 target points
3. Solve using the augmented grid as a fixed grid
4. Compute K joint samples from the posterior
5. Subselect the locations of the samples that correspond to the 50 target points.

The cheat in B is that the adaptive solve is excluded from the timing to enable JIT-compilation. Method B is not a realistic setup because it uses a JIT-compiled code for something that can't be JIT-compiled on the first run. Still, it gives insights into what would happen in environments not limited by JAX's compilation restrictions, e.g. Julia (Bezanson et al., 2017).

We are looking for fast simulation across multiple sample numbers in this experiment.

E.2.3 Analysis

The results in Table 1 show that for a low number of samples, the (JIT-cheated) combination of adaptive simulation and interpolation is faster than adaptive target simulation, with the roles reversed for larger sample counts. For small sample counts, the differences are small; for large sample counts, the differences are large. Notably, we found adaptive simulation to run into memory issues for more than 500 samples, which is why we only show up to 500 samples. This experiment suggests two conclusions: (i) in the absence of memory- and JIT-limitations and if we do cheap computations with the ODE solution, the combination of adaptive simulation and interpolation remains the state of the art; (ii) if either of those three constraints does not hold, adaptive target simulation should be used, independent of the programming environment. However, recall from Sections 5.1 and 5.2 and Appendix E.1 that in JAX, adaptive target simulation always wins.