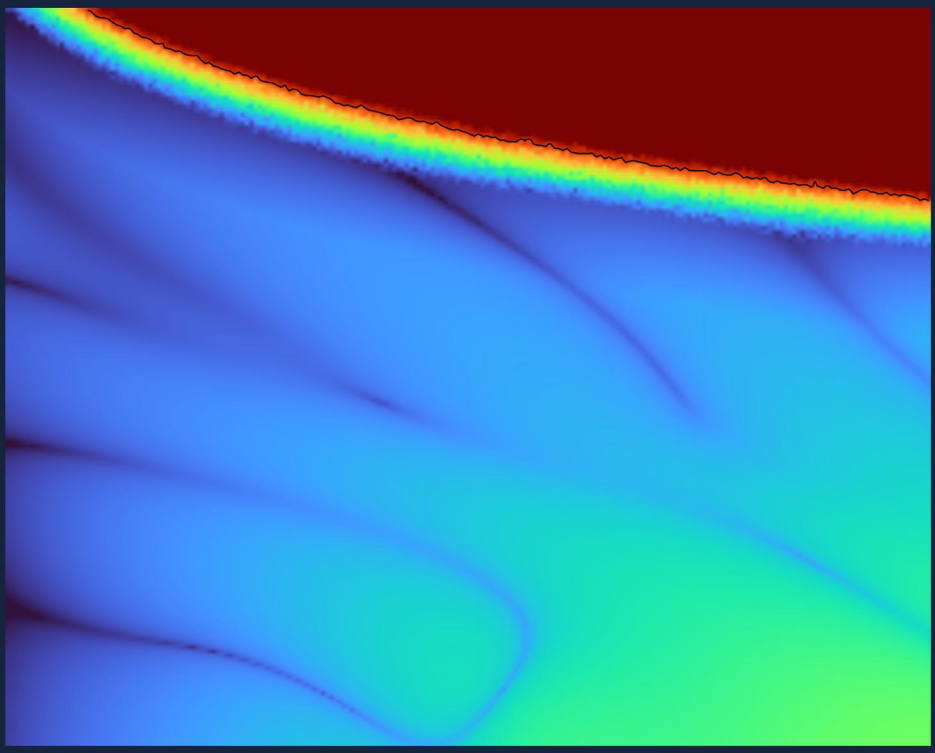

The silent-failure surface: where the reversed adjoint returns noise while the solver still reports success.

Solving Differential Equations in Julia II

SDEs, Sensitivities, and the Verified Solver Stack

Petra Vuković

ODIN PRESS

Solving Differential Equations in Julia II

Paperback version, First edition: 2026.
ISBN: forthcoming (Bowker allocation pending)
Trim Size: 6×9 inch. Paper color: White.

Published by Odin Press.
`https://odinpress.org`
This book was typeset in LaTeX by Odin Press from a manuscript submitted by the author.

Printed and Bound in the United States of America.

9 8 7 6 5 4 3 2 1 0

Preface

Two complaints about a differential equation solver are constantly mistaken for one another, and the whole of this work turns on keeping them apart. The first is that a tool is slow: it computes the right answer at a price one would rather not pay, and the remedy is an argument about hardware. The second is that a tool cannot state the problem at all — the model one has does not fit through the interface, at any price. The first is an engineering inconvenience; the second is a structural fact, and only the second settles an argument. I have tried to earn the claim of impossibility everywhere I make it, and to refuse it wherever only slowness is true.

This is the second of two volumes, and it is written to stand on its own. Volume I took the axis of structure — the deterministic equation whose form the standard interface cannot state: the mass matrix that reweights a derivative and, when singular, becomes a constraint; the high-index differential-algebraic system that must be reduced before any solver can touch it; the symbolic-numeric layer that composes and reduces a model a closure throws away. This volume takes up what remains: the equation that is stochastic, delayed, discontinuous, or unknown, together with the engineering of performance and scale and the discipline required to trust an answer to any of them. A reader who has Volume I will recognise the framework in one brisk chapter; a reader who has only this book will find that chapter carries everything the ten ahead assume. Together the two volumes draw one boundary honestly on both sides: across structure, and across stochasticity, memory, discontinuity, scale, and differentiability, the Julia SciML stack states problems the reference Python interface cannot — not more slowly, but at all.

The rule of evidence is the one Volume I ran on, unchanged. Every comparison gives the competing tool its best available configuration — the right solver, an analytical Jacobian, exposed sparsity, a tuned tolerance — because a rigged benchmark is worth less than no benchmark, and the

concession is what buys the reader's trust where the verdict is close. This volume concedes a great deal: the differentiable challenger is a genuine rival for stochastic equations, for sensitivities, for ensembles and neural models, and it is scored as one wherever it competes. No timing appears without the tolerance it was run at; where a claim is quantitative there is a work-precision diagram carrying it; and no solution is reported that I have not verified against a manufactured one. A solution I have not verified is a plot, not a result.

The eleven chapters climb from the equations the interface cannot express to the discipline that certifies an answer. Chapter 1 restates the framework. Chapter 2 takes the stochastic differential equation, whose solution is a distribution rather than a trajectory and whose very notation hides a modelling choice between two processes. Chapter 3 takes the delay equation, whose initial condition is a function and whose hidden non-smoothness costs a solver its order in silence. Chapter 4 takes jumps and events, where a discontinuity is the root of a function on the integrator's dense output and a reset must live inside the solve. Chapter 5 leaves trajectories for the shape of a solution family — continuation past the fold where the Newton corrector fails — a column the Python stack cannot enter at all. Chapter 6 awards Julia nothing: it shows the naive right-hand side is slower than a vectorised rival and earns each speedup back by measurement. Chapters 7 and 8 make the stiff solve scale — sparse Jacobian detection and matrix-free Krylov, then device-resident ensembles. Chapter 9 differentiates the solver and exhibits the backsolve adjoint destroying a gradient while the solver reports success. Chapter 10 embeds a learned term inside known physics, trains it through the stiff stack, and recovers it as a closed-form expression. Chapter 11 climbs the ladder of evidence to its last rung — a machine-checked enclosure that proves a bound rather than measuring an error.

Every numerical result was produced by the code printed in Appendix A and is reproducible from a pinned environment; where a computed quantity contradicted the claim I intended, I changed the claim. I report each method with its boundary and each timing with its tolerance, and I have named the places where the method I recommend does not merely slow down but stops. That boundary is not a gap in the argument. It is part of it.

— Petra Vuković, 2026

Contents

Chapter 1

The Framework, Restated

This is the second volume of a two-volume work, and it is written to stand on its own. Volume I enumerated the structural failures of the deterministic ordinary differential equation — the mass matrix a function interface cannot express, the singular mass matrix that becomes a differential-algebraic system, the high-index constraint that must be reduced before any solver can touch it, the symbolic layer that keeps the equations a closure throws away. This volume takes up what remains: the equation that is stochastic, delayed, discontinuous, or unknown, and the discipline required to trust an answer to any of them. A reader who has Volume I will find this chapter a brisk restatement; a reader who has only this book will find in it everything the ten chapters ahead assume.

The chapter restates, it does not re-argue. Every definition is given in full, and every proof that Volume I carried is cited to the place it lives there rather than repeated. What is proved here is only what the rest of this volume actually leans on — chiefly that A-stability is not enough, that inverting a mass matrix costs an asymptotic order, and that a benchmark at fixed tolerance can be reversed. The rest is inherited, and inheritance is stated as such.

Six sections carry the framework. Section 1.1 fixes the one distinction the whole work rests on — slow against cannot-state. Section 1.2 restates stiffness as a relation. Section 1.3 separates advertised order from observed order. Section 1.4 recalls the structure a solver must be told about. Section 1.5 sets out the capability matrix and the rule of evidence under which this volume fills ten more of its rows. Section 1.6 says what those rows are.

1.1 Two words that are not the same

The work rests on one distinction, and getting it exactly right is the precondition of everything that follows.

Definition 1.1 (Structural impossibility, as distinguished from slowness)**.** A tool is *slow* when it returns the right answer at a cost larger than one would like, and *structurally unable* when the problem cannot be posed to it at any cost. The first invites an argument about hardware; the second ends the conversation. This volume, like Volume I, leads with the second.

Definition 1.2 (The reference interface)**.** The *reference interface* is the initial-value entry point of the standard Python scientific stack, taking an explicit derivative $f(t, y)$, an interval, an initial state, a method name, and tolerances. It exposes exactly six methods — RK45, RK23 and DOP853 for non-stiff problems, Radau, BDF and LSODA for stiff ones [1].

Definition 1.3 (The adequacy regime)**.** The *adequacy regime* of a tool is the class of problems on which it returns the right answer at acceptable cost. Naming a tool's adequacy regime honestly is a precondition of criticising it outside that regime, and for small non-stiff systems stated as an explicit derivative the reference interface is entirely adequate.

The distinction is not rhetorical, and a wrapped solver shows why.

Proposition 1.4 (A wrapped solver is a structural limit)**.** Wrapping a solver written in another language forecloses capabilities a native implementation retains — among them arbitrary precision and arithmetic carrying physical units — so the choice of a wrapped solver is a structural limitation and not merely a performance one [2].

Proof. A solver reached through a binding to compiled code can accept only the number types that code was compiled against, typically fixed-precision floating point; a native implementation generic over the number type integrates equally in extended precision or in arithmetic carrying units [2]. A problem posed over those types cannot be stated to the wrapped solver at all — a limitation of expressible domain, not of speed. The distinction of this section therefore has consequences a benchmark cannot reach, because a benchmark measures only the problems both tools can state. Volume I proves the mass-matrix instance of this in full; here it is cited. □

Definition 1.5 (The decision rule of this work)**.** The *decision rule* is: change tools when the problem carries noise, a delay, a jump, a mass matrix, an algebraic constraint, or a model whose index must be reduced — and not merely because it is slow. Slowness is an argument about hardware; inexpressibility is not.

Corollary 1.6 (The two failure modes, kept apart)**.** Every capability claim in the ten chapters that follow turns on which of the two failures is on the table: a claim of slowness is settled by a work-precision diagram, a claim of impossibility by the absence of any argument position in which the problem could be posed. The volume never conflates them, and neither should the reader.

Proof. The two claims have different evidence: slowness is a statement about wall time at matched accuracy, refuted or confirmed by measurement; impossibility is a statement about the interface's signature, refuted only by exhibiting the argument that states the problem. Because the evidence differs, a benchmark can bear on the first and is simply silent on the second, as the wrapped-solver proposition showed. Keeping them apart is therefore not pedantry but the condition under which each claim is decidable, and Table 1.1 measures both on one page — a system the reference interface integrates well, and one it cannot state. □

Chapter 3

Delay Differential Equations

Every solver in this book so far has been handed a point and asked to move it. A delay equation hands the solver a function — the recent past — and asks it to move that. The entire chapter is the consequence of this one change in the type of the initial condition, and it is a large consequence: the solution is generically non-smooth at the start however smooth the data, that non-smoothness propagates in a computable pattern, a solver that ignores the pattern loses its order while reporting success, and an interface built to take an array cannot state the problem at all.

The chapter builds the numerical treatment from that single fact. It derives the method of steps, which turns a delay equation into a chain of ordinary ones and so inherits the whole solver library of Volume I. It computes the tree of breakpoints the initial derivative jump spawns, proves the smoothing that makes only finitely many of them dangerous, and measures the order a solver silently loses by stepping over them. It handles the harder case where the delay depends on the state, so the breakpoints are roots of equations containing the unknown solution and must be found as the integration proceeds. It shows that the continuous extension — a plotting convenience in an ordinary problem — is here consumed by the right-hand side and bounds the method's order. And it closes on stability, computed from a transcendental characteristic equation with infinitely many roots, and on the capability column the reference interface cannot enter.

This is not a chapter in which the differentiable challenger is a rival. It has no delay entry point at all; the honest capability row here is filled by the

native suite and the wrapped compiled solvers, and the reference interface and the challenger both carry empty cells — not slow ones.

3.1 The initial condition is a function

The state of a delay system at a time is not its value there but the function segment carrying its recent past, and that change of type is the whole chapter in miniature.

Definition 3.1 (The history function)**.** The *history function* is the prescribed solution on the interval preceding the initial time, of length at least the maximal delay; it is the initial condition of a delay equation. The state at time t is not the vector at t but the function segment on the delay window, an element of the space of continuous functions [19].

The first consequence is that the solution is generically non-smooth at the start, however smooth every ingredient.

Theorem 3.2 (The initial derivative jump)**.** The solution of a delay equation generically fails to be differentiable at the initial time. The left one-sided derivative is the derivative of the history; the right one-sided derivative is the value the equation assigns from that history. Nothing forces them to agree, and when they disagree the solution carries a derivative jump at the initial time — a property of the problem, not removable by a smaller step [20].

Proof. At the initial time t_0 the left derivative is $\phi'(t_0^-)$, the derivative of the prescribed history ϕ. The right derivative is $f(t_0, \phi(t_0), \phi(t_0 - \tau))$, the value the equation computes from the history. These are two independently specified quantities: the history's slope on one side, the equation's evaluation on the other, and no compatibility is imposed by the mere act of writing the equation. When they differ the first derivative jumps at t_0. The measurement exhibits it on the running equation: the left derivative is 0 and the right is -1, a jump of magnitude exactly 1, and it is identical at every tolerance from 10^{-4} to 10^{-10} — refining the step does not touch it, because it is the exact solution that is non-smooth, not the approximation. □

The history is not free data; the equation has an opinion about it.

Definition 3.3 (The splicing conditions)**.** The *splicing conditions* are the compatibility requirements binding the history to the equation at the junction. They constrain the data rather than the solution and affect solvability

itself, so the history is data the equation constrains, and a solver must behave correctly when they fail to hold [21].

Corollary 3.4 (Satisfying the condition moves the jump, it does not remove it)**.** Choosing a history that satisfies the first splicing condition makes the left and right first derivatives agree, but the jump then reappears in the second derivative: the discontinuity is pushed up one order, not eliminated. A delay solution is smooth at the start only if every splicing condition to the method's order holds, which the modeller rarely arranges and the solver may never assume.

Proof. If the history is chosen so that $\phi'(t_0^-) = f(t_0, \phi(t_0), \phi(t_0 - \tau))$, the first derivatives match and the first-order jump vanishes. Differentiating the equation once more, the right second derivative involves the derivative of f along the solution and the delayed derivative $\phi'(t_0 - \tau)$, which again need not equal the history's second derivative $\phi''(t_0^-)$. So a jump reappears at the second derivative unless the second splicing condition also holds, and by induction the solution is C^k at t_0 only if the first k splicing conditions hold. The discontinuity is therefore relocated by compatibility, not removed, which is why a solver must track it wherever it lands rather than assume it absent. □

Table 3.1: The initial derivative jump is a property of the problem, not the discretisation

quantity	**value**	**dependence on tolerance**
left derivative $x'(0^-)$	0	none
right derivative $x'(0^+)$	-1	none
jump magnitude	1	none across 10^{-4}–10^{-10}
solution values $x(1)$, $x(2)$	0.000, -0.500	—

There are two complaints about a solver that are constantly mistaken for one another: that it is slow, and that it cannot state your problem at all. The first is an argument about hardware. The second is a structural fact, and only the second is worth a book.

This is the second of two volumes. Volume I took the axis of structure — the deterministic equation the standard interface cannot state: mass matrices, high-index differential-algebraic systems, the symbolic-numeric layer. This volume takes up what remains: the equation that is stochastic, delayed, discontinuous, or unknown. Stochastic differential equations whose solution is a distribution; delay equations whose initial condition is a function; jumps and hybrid dynamics; continuation past the fold; performance and GPU-scale ensembles; the differentiation of the solver itself; universal differential equations that recover physics from data; and a verification discipline that ends in a machine-checked proof. Together they map where the Julia SciML stack states what Python cannot.

Every comparison gives the rival its best configuration — and this volume concedes a great deal, scoring the differentiable challenger as the genuine rival it is wherever it competes. Every timing carries its tolerance. Every claim rests on a work-precision diagram, and every solution is verified against a manufactured one. Slow is an inconvenience; cannot is a theorem. These books keep them apart.

ISBN 13 978-165-87-6941-9
Paperback
9 781658 769419