

Parallelizing Explicit and Implicit Extrapolation Methods for Ordinary Differential Equations

Utkarsh
Massachusetts Institute of Technology
Julia Computing
Cambridge, Massachusetts, USA
utkarsh5@mit.edu

Chris Elrod
Julia Computing
Cambridge, Massachusetts, USA
chris.elrod@juliacomputing.com

Yingbo Ma
Julia Computing
Cambridge, Massachusetts, USA
yingbo.ma@juliacomputing.com

Konstantin Althaus
Technical University of Munich
Munich, Germany
konstantin.althaus@tum.de

Christopher Rackauckas
Julia Computing
Massachusetts Institute of Technology
Pumas AI Inc.
Cambridge, Massachusetts, USA
chris.rackauckas@juliacomputing.com

Abstract—Numerically solving ordinary differential equations (ODEs) is a naturally serial process and as a result the vast majority of ODE solver software are serial. In this manuscript we developed a set of parallelized ODE solvers using extrapolation methods which exploit “parallelism within the method” so that arbitrary user ODEs can be parallelized. We describe the specific choices made in the implementation of the explicit and implicit extrapolation methods which allow for generating low overhead static schedules to then exploit with optimized multi-threaded implementations. We demonstrate that while the multi-threading gives a noticeable acceleration on both explicit and implicit problems, the explicit parallel extrapolation methods gave no significant improvement over state-of-the-art even with a multi-threading advantage against current optimized high order Runge-Kutta tableaus. However, we demonstrate that the implicit parallel extrapolation methods are able to achieve state-of-the-art performance (2x-4x) on standard multicore x86 CPUs for systems of < 200 stiff ODEs solved at low tolerance, a typical setup for a vast majority of users of high level language equation solver suites. The resulting method is distributed as the first widely available open source software for within-method parallel acceleration targeting typical modest compute architectures.

I. INTRODUCTION

The numerical approximation of ordinary differential equations (ODEs) is a naturally serial time stepping process, meaning that methods for parallelizing the solution of such ODEs requires either tricks or alternative solvers in order to exploit parallelism. The most common domain, and one of the most common applications in scientific computing, is the parallel solution of large-scale systems of ODEs (systems of millions or more equations) which arise from the semi-discretization of partial differential equations. For such cases, the size of the system allows for parallel efficiency to be achieved by parallelizing some of the most expensive computations, such as implicit parallelism of BLAS [1]/LAPACK [2], sparse linear solvers, or preconditioner computations, on compute clusters and with heterogeneous GPU+CPU compute. Commonly used open source softwares such as Sundials [3] and DifferentialE-

quations.jl [4] help users automate the parallelism in such cases. However, for this manuscript we look in the opposite direction at parallelism for small systems of ODEs (< 200).

Parallel computing on small systems of equations comes with a separate set of challenges, namely that computations can easily become dominated by overhead. For this reason, most software for accelerating the parallel solution of ODEs focus on the parallelization of generating ensembles of solutions, i.e. generating the solution to a small system of ODEs over a set of parameters or initial conditions. Examples of this include the ensembles interface in DifferentialEquations.jl [4]. However, in many cases the ensemble form may not be easy to exploit. One key example is in parameter estimation or model calibration routines which are typically done with some gradient-based optimization technique and requires one realization of the model at a given parameter before advancing. Given questions on the Julia [5] Discourse demonstrate that the vast majority of users both typically solve < 200 ODE systems and have multi-core compute available (e.g. Core i5/i7 chips with 2-16 processors), investigated whether the most standard ODE solve cases could benefit from some form of parallelism.

Prior research has developed a large amount of theory around potential solvers for such problems which fall into the categories of parallel-in-time (PIT) methods and within-step parallelism methods [6]. Parallel-in-time methods address the issue by effectively stepping the ODE at multiple time points simultaneously, similar to the ensemble approach, and impose a nonlinear system constraint to relax the initial conditions of the future points to arrive at a sufficiently smooth solution. An open source software for PIT methods, XBraid, does exist but focuses on large compute hardware [7]. Documents from the XBraid tutorials [8] estimate current PIT methods outperform classical solvers at ~ 256 cores¹, making them impractical

¹Private correspondences and tests with DifferentialEquations.jl also confirm a similar cutoff point with PIT methods

for standard compute hardware. Thus while PIT methods are a promising direction for accelerating ODE solving for the exascale computers targeted by the XBraid project [7], these methods do not suffice for the ≤ 16 core nature of standard consumer computing devices.

However, promising prior results exist within the within-step parallelism research. Nowak 1998 parallelized the implicit extrapolation methods and showed that the parallelization did improve performance [9], though its tests did not compare against multithreaded (sparse) factorizations like is seen in modern suites such as UMFPACK [10] in SuiteSparse [11], no comparison was made against optimized solver softwares such as CVODE [3] or LSODA [12], and no open source implementation exists for this method. Additionally, Ketcheson (2014) [13] demonstrated that within-step parallel extrapolation methods for non-stiff ODEs can outperform dop853, an efficient high-order Runge-Kutta method [14]. Two notable issues with this study are that (a) no optimized open source software exists for this method for further investigation, (b) the study did not investigate whether such techniques could be useful for stiff ODEs, which tend to be the most common type of ODEs for many applications from biology and chemistry to engineering models. These results highlight that it may be possible to achieve state-of-the-art (SOA) performance by exploiting parallelism, though no software is readily available.

In this manuscript we build off of the prior work in within-method parallel extrapolation to build an open source software in Julia [5] targeting < 200 ODE systems on standard compute architectures. We demonstrate that this new solver is the most efficient solver for this class of equations when high accuracy is necessary, and demonstrate this with benchmarks that include many methods from DifferentialEquations.jl [4], SUNDIALS [3] (CVODE), LSODA [12], and more. To the best of our knowledge, this is the first demonstration that a within-step parallel solver can be the most efficient method for this typical ODE and hardware combination.

II. EXTRAPOLATION METHODS

Our exposition of extrapolation methods largely follows that of Hairer I [14] and II [15], and [16]. Take the ODE:

$$u' = f(u, p, t), \quad (1)$$

with some known initial condition $u(t_0) = u_0$ over a time span $t \in (t_0, t_f)$. The extrapolation methods are variable-order and variable-step methods which generate higher precision approximations of ODE solutions computed at different time step-sizes. For the N^{th} current order of the algorithm, we generate $f(u, p, t + dt)$ at the current time-step dt for each order from 1 to N . For the approximation at $t + dt$, the algorithm chooses a subdividing sequence which discretizes into further fixed smaller steps between t and $t + dt$. Choosing a sequence of the form:

$$n_1 < n_2 < n_3 < n_4 < \dots < n_N \quad (2)$$

Generates internal step-sizes $h_1 > h_2 > h_3 > h_4 > \dots > h_N$ by $h_i = \frac{dt}{n_i}$. The subdividing sequences vary with order,

having smaller time-steps in higher orders for finer resolution. The algorithm chosen for this is suitably an efficient implicit/explicit method between t and $t + dt$ is of p^{th} order.

The tabulation of 1^{st} N^{th} calculations generate starting-stage of extrapolation computation, denoted by:

$$\begin{aligned} u_{h_i}(t + dt) &= T_{i,1} \\ i &= j, j-1, j-2, \dots, j-k+1. \end{aligned} \quad (3)$$

Extrapolation methods use the interpolating polynomial

$$p(h) = \tilde{u} - e_p h^p - e_{p+1} h^{p+1} - \dots - e_{p+k-2} h^{p+k-2} \quad (4)$$

such that $p(h_i) = T_{i,1}$ to obtain a higher order approximation by extrapolating the step size h to 0. Concretely, we define $T_{j,k} := p(0) = \tilde{u} = u(t + dt) + \mathcal{O}(dt^{p+k})$. In order to find \tilde{u} , we can solve the linear system of k variables and k equations formed by equations (3) and (4). This conveniently generates an array of approximations with different orders which allows simple estimates of local error and order variability techniques:

$$\begin{array}{cccc} T_{1,1} & & & \\ T_{2,1} & T_{2,2} & & \\ T_{3,1} & T_{3,2} & T_{3,3} & \\ \dots & \dots & \dots & \dots \end{array}$$

Aitken-Neville's algorithm uses Lagrange polynomial interpolation formulae [17], [18] to make: (4):

$$T_{j,k+1} = T_{j,k} + \frac{T_{j,k} - T_{j-1,k}}{\frac{n_j}{n_{j-k}} - 1} \quad (5)$$

Finally, the N^{th} order is $u(t + dt) = T_{N,N}$.

As shown in (2), the subdividing sequence should be positive and strictly increasing. Common choices are:

- 1) Harmonic [19]: $n = 1, 2, 3, 4, 5, 6, 7, 8 \dots$
- 2) Romberg [20]: $n = 1, 2, 4, 8, 16, 32, 64, 128, 256 \dots$
- 3) Bulirsch [20], [21]: $n = 1, 2, 3, 4, 6, 8, 12, 16 \dots$

The "Harmonic" sequence generates the most efficient load balancing and utilization of multi-threading in parallel computing which is discussed more in Section III.

A. Explicit Methods

1) Extrapolation Midpoint Deuffhard and HairerWanner:

Both the algorithms use explicit midpoint method for internal step-size computations. The representation used here is in the two-step form, which makes the algorithm symmetric and has even powers in asymptotic expansion [22]

$$\begin{aligned} u_{h_i}(t_0) &= u_0 \\ u_{h_i}(t_1) &= u_0 + h_i f(u_0, p, t_0) \\ u_{h_i}(t_n) &= u_{h_i}(t_{n-2}) + 2h_i f(u_{h_i}(t_{n-1}), p, t_{n-1}) \end{aligned} \quad (6)$$

The difference between them arises from the step-sizing controllers. The ExtrapolationMidpointDeuffhard is based on the Deuffhard's DIFEX1 [19] adaptivity behaviour and ExtrapolationMidpointHairerWanner is based on Hairer's ODEX [14] adaptivity behaviour.

The extrapolation is performed using barycentric formula which based on the lagrange barycentric interpolation [23]. The interpolation polynomial is given by:

$$\begin{aligned} w_j &= \prod_{i=1:N+1, i \neq j} \frac{1}{n_j^{-2} - n_i^{-2}} \\ \rho(h) &= \prod_{i=1:N+1} h - n_i^{-2} \\ p(h) &= \rho(h) \sum_{j=1}^{N+1} \frac{w_j}{h - n_j^{-2}} T_{j,1} \end{aligned} \quad (7)$$

Extrapolating the limit $h \rightarrow 0$, we get:

$$u(t + dt) = \rho(0) \sum_{j=1}^{N+1} \frac{w_j}{-n_j^{-2}} T_{j,1} \quad (8)$$

Where w_j are the extrapolation weights, $\rho(0)$ are the extrapolation coefficients and n_j denotes the subdividing sequence.

The choice for barycentric formula instead of Aitken-Neville is mainly due to reduced computation cost of ODE solution at each time-step [16]. The extrapolation weights w_j and coefficients $\rho(0)$ can be easily computed and stored as tableau's. The yields the computation cost to be $O(N_{max}^2)$ for the precomputation where N_{max} is the maximal order than can be achieved by the method. The extrapolation method of order N generates a method of error in order of $2(N + 1)$ [14], [16].

It can be analysed that computational cost of with Aitken-Neville for $T_{N,N}$ is $O(N^2 d)$ ($d(1 + 2 + \dots + N) = O(N^2 d)$), where N is the extrapolation order and d is the dimension of the u [16]. The computation cost of (8) is simply $O(Nd)$ (a linear combination all $T_{j,1}$ across d dimensions) [16].

Numerical Stability and Analysis: In comparison The numerical performance follows similar behaviour as that of Aitken-Neville in the subdividing sequences of Romberg and Bulirsch, where the absolute error decreases as extrapolation order q increases [24]. However, in the harmonic sequence, the numerical stability in both cases of extrapolation remains up-to as much as up-to 15 order and then it diverges [16].

B. Implicit Methods

We will consider implicit methods as basis for internal step-size calculation using subdividing sequences in extrapolation methods. They are widely used for solving stiff ODEs (maybe write more why implicit methods are suited for it).

1) *Implicit Euler Extrapolation:* The ImplicitEulerExtrapolation uses the Linearly-Implicit Euler Method for internal step-sizing. Mathematically:

$$\begin{aligned} (I - h_i J)(u_{h_i}(t_{n+1}) - u_{h_i}(t_n)) &= h f(u_{h_i}(t_n), p, t_n) \\ \therefore u_{h_i}(t_{n+1}) &= u_{h_i}(t_n) + (I - h_i J)^{-1} h f(u_{h_i}(t_{n+1}), p, t_n) \end{aligned} \quad (9)$$

where I and $J \approx \frac{\partial f(u_{h_i}, p, t)}{\partial u_{h_i}}$ is the $\mathbb{R}^{d \times d}$ identity and jacobian matrix respectively. Clearly, the method is non-symmetrical

and consequently would have non-even powers of h global error expansion. The methods are $A(\alpha)$ stable with $\alpha \approx 90^\circ$ [15]. The extrapolation scheme used is Aitken-Neville (5).

2) *Implicit Euler Barycentric Extrapolation:* We experimented with barycentric formulas [23] to replace extrapolation algorithm in ImplicitEulerExtrapolation. Since there are no even powers in global error expansion, the barycentric formula [23] needs to be changed as follows:

$$\begin{aligned} w_j &= \prod_{i=1:N+1, i \neq j} \frac{1}{n_j^{-1} - n_i^{-1}} \\ \rho(h) &= \prod_{i=1:N+1} h - n_i^{-1} \\ p(h) &= \rho(h) \sum_{j=1}^{N+1} \frac{w_j}{h - n_j^{-1}} T_{j,1} \end{aligned} \quad (10)$$

Extrapolating the limit $h \rightarrow 0$, we get:

$$u(t + dt) = \rho(0) \sum_{j=1}^{N+1} \frac{w_j}{-n_j^{-1}} T_{j,1} \quad (11)$$

Consequently, the extrapolation method of order N generates a method of error in order of $N + 1$.

3) *Implicit Hairer Wanner Extrapolation:* For implicit extrapolation, symmetric methods would be beneficial to provide higher order approximations. The naive suitable candidate is the trapezoidal rule, but the resulting method is not stiffly stable and hence undesirable for solving stiff ODEs [25], [14].

G. Bader and P. Deuffhard [26], [19] developed the linearly-implicit midpoint rule with Gragg's smoothing [27], [28] as extension of popular Gragg-Bulirsch-Stoer (GBS) [21] method. The algorithm is given by:

$$\begin{aligned} (I - h_i J)(u_{h_i}(t_1) - u_0) &= h_i f(u_0, p, t_0) \\ (u_{h_i}(t_{n+1}) - u_{h_i}(t_n)) &= (u_{h_i}(t_n) - u_{h_i}(t_{n-1})) \\ &\quad + 2(I - h_i J)^{-1} h_i f(u_{h_i}(t_n), p, t_n) \\ &\quad - 2(I - h_i J)^{-1} (u_{h_i}(t_n) - u_{h_i}(t_{n-1})) \end{aligned} \quad (12)$$

Followed by Gragg's Smoothing [27], [28]:

$$\begin{aligned} t &= t_0 + 2n h_i \\ T_{j,1} = S_{h_i}(t) &= \frac{u_{h_i}(2n + 1) + u_{h_i}(2n - 1)}{2} \end{aligned} \quad (13)$$

The Gragg's smoothing [27], [28] implementation in OrdinaryDiffEq.jl² improved the stability of algorithm resulting in lower f evaluations, time steps and linear solves. Consequently, it increased the accuracy and speed of the solvers.

The extrapolation scheme used is similar to extrapolation midpoint methods, namely the barycentric formula [23] (8). One of the trade-offs for the increased stability from Gragg's smoothing [27], [28] is the reduced order of error for the extrapolation method. As stated earlier, the barycentric formula provides error of order $2(N + 1)$ which gets reduced to $2(N + 1) - 1 = 2N + 1$ in the case of Implicit Hairer Wanner

²<https://github.com/SciML/OrdinaryDiffEq.jl/pull/1212>

Extrapolation [14], [26]. The algorithm requires to use even subdividing sequence and hence we are using multiples of 4 of the common sequences in the implementation. Furthermore, the convergence tests for these methods³ are written in the test suite of OrdinaryDiffEq.jl.

C. Adaptive time-stepping and order of algorithms

The methods follow a comprehensive step-sizing and order selection strategy. The adaptive time-stepping is mostly followed on the lines of Hairer’s ODEX [14] adaptivity behaviour. The error at k^{th} order is expressed as:

$$err_k = \|T_{k,k-1} - T_{k,k}\| \quad (14)$$

and the scaled error is represented as:

$$err_{scaled} = \frac{err_k}{abstol + \max(u, u_{prev})reltol} \quad (15)$$

Finally, the $h_{optimal}$ is calculated using the standard-controller:

$$q = \max(q_{min}, \min(\frac{err_{scaled}^{alg_order+1}}{\gamma}, q_{max})) \quad (16)$$

$$h_{optimal} = \frac{h}{q}$$

Where γ is the safety factor, q_{min} is the minimum scaling and q_{max} is the maximum scaling. For the optimal order selection, the computation relies on the “work calculation” which is stage-number A_k per step-size $h_k, \frac{A_k}{h_k}$.

The stage-number includes the no. of $f(u, p, t)$ RHS function evaluations, jacobian matrix evaluation, and forward and backward substitutions. These are pre-computed according the subdividing sequence and stored as a cache. More details about these calculations can be found here.⁴ These work calculations form the basis of order-selection. The order-selection is restricted between the window $(k-1, k+1)$ and appropriate conditions are passed to convergence monitor. Briefly describing, it checks the convergence in the window and subsequently accepts the order which has the most significant work reduction during increasing/decreasing the order and $error < 1$. Detailed description can be found at order and step-size control in [14].

III. PARALLELIZATION OF THE ALGORITHM

A. Choosing Subdividing Sequences for Static Load Balancing

The exploit of parallelism arises from the computation of $T_{j,1}$. k evaluations of $T_{j,1}$ are done to find the solution $u(t + dt)$. These evaluations are independent and are thus parallelizable. However, because of the small systems being investigated, dynamic load balancing overheads are much too high to be relied upon and thus we tailored the parallelism implementation to make use of a static load balancing scheme as follows. Each $T_{j,1}$ requires n_j calls to the function f , and

if the method is implicit then there is an additional n_j back substitutions and one LU factorization. While for sufficiently large systems the LU-factorization is the dominant cost due to its $O(n^3)$ growth for an n ODE system, in the < 200 ODE regime with our BLAS [1]/LAPACK [2] implementations⁵ we find that this cost is close enough to the cost of a back substitution that we can roughly assume $T_{j,1}$ requires n_j “work units” in both the implicit and explicit cases method cases.

If we consider the multiples of harmonic subdividing sequence is 2, 4, 6, 8, 10, 12, ... then the computation of $T_{j,1}$ needs $2j$ work. Because $T_{k,k}$ is a numerical approximation of order $p = 2k$ we can simply give each $T_{j,1}$ a processor if k processors are available. The tasks would not finish simultaneously due to different amount of work. With Multiple Instruction and Multiple Data Stream (MIMD) processors, multiple computations can be loaded to one processor to calculate say $T_{1,1}$ and $T_{k-1,1}$ on a single processor, resulting in $2 + 2(k-1) = 2k$ function evaluations. Consequently, each processor would contain $T_{j,1}$ and $T_{k-j,1}$ computations and we require $\frac{k}{2}$ processors. This is an effective load-balancing and all processes would finish simultaneously due to an approximately constant amount of work per chunk.

Julia provides compose-able task-based parallelism. To take advantage of the regularity of our work and this optimized static schedule, we avoid some of the overheads associated with this model such as task-creation and scheduling by using Polyester.jl’s [29] @batch macro, which allows us to run our computation on long-lived tasks that can remain active with a spin-lock between workloads, thereby avoiding the need for rescheduling. We note that this load-balancing is not as simple in sequences of Romberg and Bulirsch, and thus in those cases we multi-thread computations over constant p processors only. For this reason, the homonic subdividing sequence is preferred for performance in our implementation.

B. Parallelization of the LU Factorization

One of the most important ways multi-threading is commonly used within implicit solvers is within the LU factorization. The LU factorization is multi-threaded due to expensive $O(N^3)$ complexity. For Jacobian matrices smaller than 100×100 , commonly used BLAS/LAPACK implementations such as OpenBLAS, MKL, and RecursiveFactorization.jl are unable to achieve good parallel performance. However, the implicit extrapolation methods require solving linear systems with respect to $I - h_i J$ for each sub-time step h_i . Thus for the within-method parallel implicit extrapolation implementation we manually disabled the internal LU factorization multi-threading and parallelized the LU factorization step by multi-threading the computation of this ensemble of LU factorizations. If the chosen order is sufficiently then a large number of h_i would be chosen and thus parallel efficiency would be achieved even for small matrices. In the results this will be seen as the lower tolerance solutions naturally require higher

³https://github.com/SciML/OrdinaryDiffEq.jl/blob/master/test/algonvergence/ode_extrapolation_tests.jl

⁴https://github.com/utkarsh530/DiffEqBenchmarks/blob/master/Extrapolation_Methods/Extrapolation_Methods.pdf

⁵RecursiveFactorization.jl notably outperforms OpenBLAS in this regime.

order methods, and this is the regime where the implicit extrapolation methods demonstrably become SOA.

IV. BENCHMARK RESULTS

The benchmarks we use work-precision diagrams to allow simultaneous comparisons between speed and accuracy. Accuracy is computed against reference solutions at 10^{-14} tolerance. All benchmarks were computed on a AMD EPYC 7513 32-Core processor ran at 8 threads⁶.

A. Establishing Implementation Efficiency

To establish these methods as efficient versions of extrapolation methods, we benchmarked these new implementations against standard optimized extrapolation implementations. The most widely used optimized implementations of extrapolation methods come from Hairer’s FORTRAN suite [30], with ODEX [14] as a GBS [21] extrapolation method and SEULEX as a linear implicit Euler method. Figure 1 illustrates that our implementation outperforms the ODEX [14] FORTRAN method by 6x on the 100 Independent Linear ODEs problem. And similar to what was shown in Ketcheson 2014 [13], we see that the parallelized extrapolation methods were able to outperform the dop853 high order Runge-Kutta method implementation of Hairer by around 4x (they saw closer to 2x). However, when we compare the explicit extrapolation methods against newer optimized Runge-Kutta methods like Verner’s efficient 9th order method [31], we only matched the SOA⁷, which we suspect is due to the explicit methods being able to exploit less parallelism than the implicit variants. Thus for the rest of the benchmarks we focused on the implicit extrapolation methods. In Figure 2 we benchmarked the implicit extrapolation methods against the Hairer SEULEX [30] implementation on the ROBER [32] problem and demonstrated an average of 4x acceleration over the Hairer implementations, establishing the efficiency of this implementation of the algorithm.

B. State-Of-The-Art Performance for Small Stiff ODE Systems

To evaluate performance of extrapolation methods, we benchmarked the parallel implicit extrapolation implementations on the following set of standard test problems [15]:

- 1) the Robertson equation [32], 3 ODEs
- 2) Orego (Oregonator) [33], [34], 3 ODEs
- 3) Hires [35], 8 ODEs
- 4) Pollution [36], 20 ODEs
- 5) QSP [37], 109 ODEs

Figures 2-6 show the Work-Precision Diagrams with each of the respective problems with low tolerances. Implicit Euler Extrapolation outperforms other solvers with lower times at equivalent errors. However, very small systems (3-7 ODEs)

⁶While more threads were available, we did not find more threads to be beneficial to accelerating these computations.

⁷In the shown Figure 1, the methods are approximately 1.5x faster, though on this random linear ODE benchmark, different random conditions lead to different performance results, averaging around 0.8x-2x, making the speedup generally insignificant

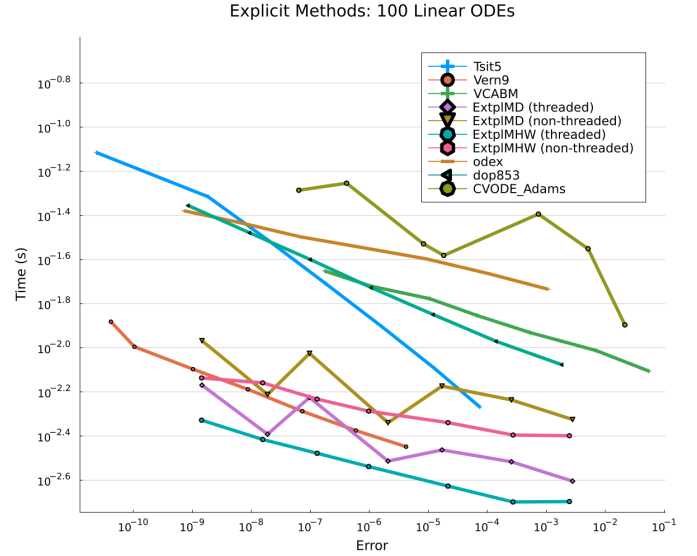


Fig. 1. Benchmark on the 100 linear ODE problem [VI-B1].

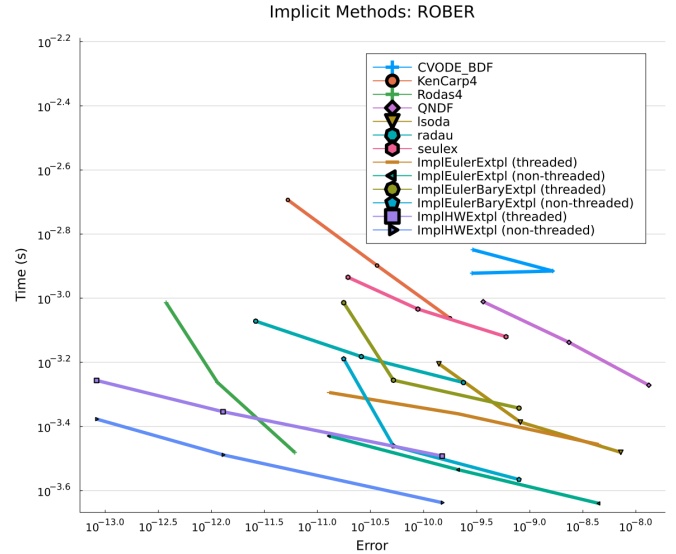


Fig. 2. Benchmark on ROBER Problem with low tolerances [VI-B2].

show a disadvantage when threading is enabled, indicating that overhead is not overcome at this size, but the non-threaded version still achieves SOA by about 2x over the next best methods (Rodas4 and radau) and matching lsoda on HIREs. By 20 ODEs, the multi-threaded form becomes more efficient and is the SOA algorithm by roughly 2.5x. But by the 109 ODEs of the QSP model, the multi-threaded form is noticeably more efficient than the non-multithreaded and the extrapolation methods outperformed lsoda BDF and CVODE’s BDF method by about 2x. Together these results show the implicit extrapolation methods as SOA for small ODE systems, with a lower bound on when multi-threading is beneficial.

V. DISCUSSION

If one follows the tutorial coding examples for the most common differential equation solver suites in high-level languages (R’s deSolve [38], Python’s SciPy [39], Julia’s DifferentialEquations.jl [4], MATLAB’s ODE Suite [40]), the vast

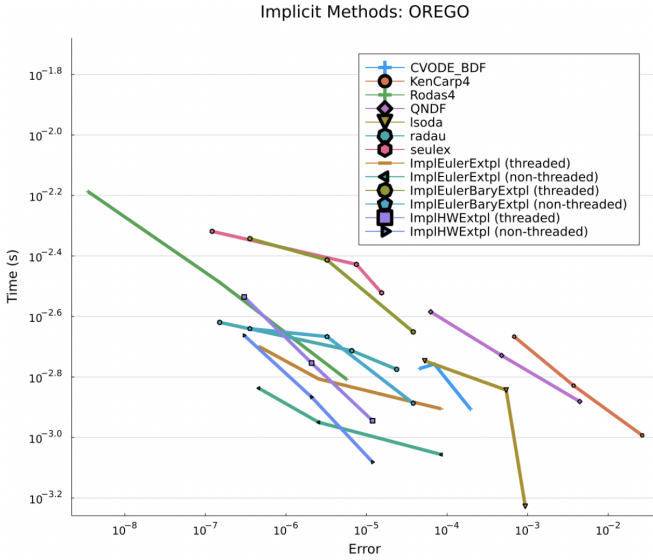


Fig. 3. Benchmark on Orego Problem with low tolerances [VI-B3].

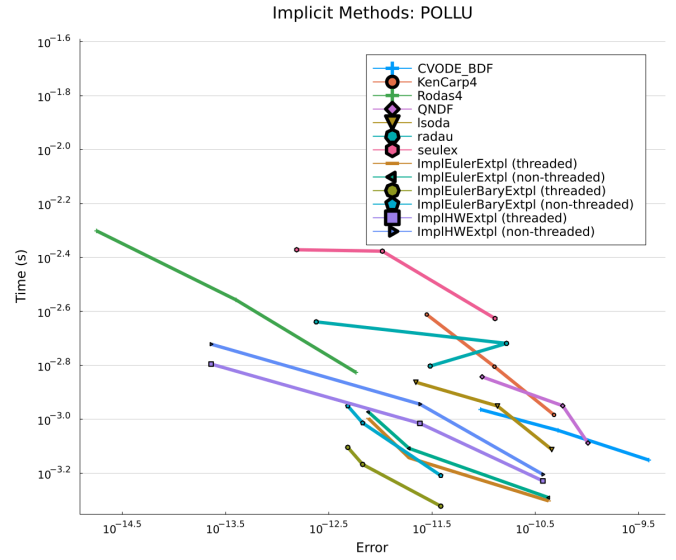


Fig. 5. Benchmark on Pollution Problem with low tolerances [VI-B5].

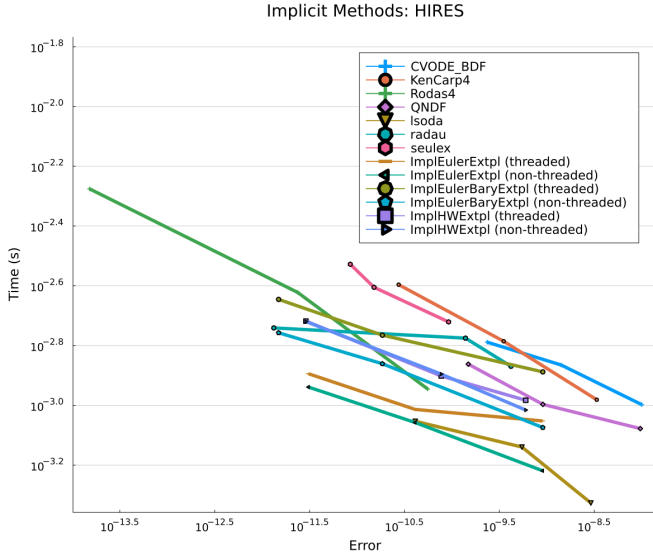


Fig. 4. Benchmark on Hires Problem with low tolerances [VI-B4].

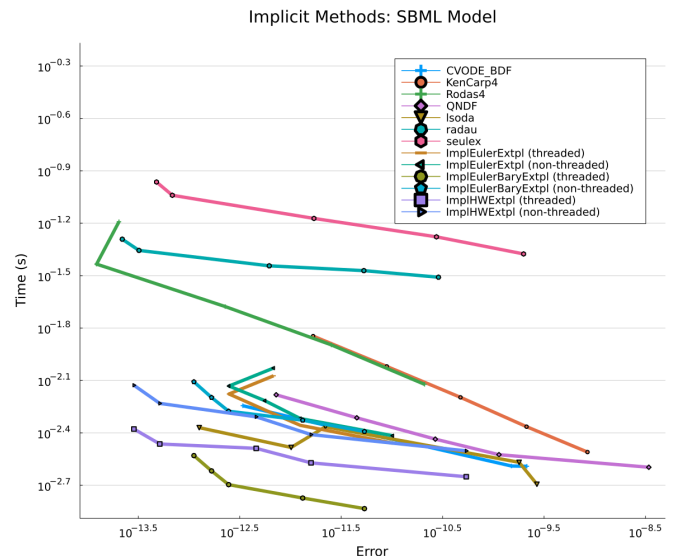


Fig. 6. Benchmark on QSP model with low tolerances [VI-B6].

majority of users in common case of < 200 ODEs will receive no parallel acceleration even though parallel hardware is readily available⁸. In this manuscript we have demonstrated how the theory of within-method parallel extrapolation methods can be used to build a method which achieves SOA for the high accuracy approximation regime on this typical ODE case. We distribute this method as open source software: existing users of DifferentialEquations.jl [4] can make use of this technique with no changes to their code beyond the characters for the algorithm choice. As such, this is the first demonstration the authors know of for automated acceleration of ODE solves on typical small-scale ODEs using parallel acceleration.⁹

Lastly, the extrapolation methods were chosen as the foundation because their arbitrary order allows the methods to

easily adapt the number of parallel compute chunks to different core counts. This implementation and manuscript statically batched the compute in a way that is independent of the specific core count, leading to generality in the resulting method. However, the downside of this approach is that extrapolation methods are generally inefficient in comparison to other methods such as Rosenbrock [41] or SDIRK [42] methods. In none of the numerous stiff ODE benchmarks by Harier II [15] or the SciML suite¹⁰ do the implicit extrapolation methods achieve SOA performance. Thus while the parallelization done here was able to achieve SOA, even better results could likely be obtained by developing parallel Rosenbrock [41] or SDIRK [42] methods (using theory described in [6]) and implementing them using our parallelization techniques. This would have the trade-off of being tied to a specific core count, but could achieve better performance through tableau optimization.

⁸Unless the f function is sufficiently expensive, like when it is a neural network evaluation

⁹The codes can be found at:
<https://github.com/utkarsh530/ParallelExBenchmarks.jl>

¹⁰<https://github.com/SciML/SciMLBenchmarks.jl>

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grant no. OAC-1835443, SII-2029670, ECCS-2029670, OAC-2103804, and PHY-2021825. We also gratefully acknowledge the U.S. Agency for International Development through Penn State for grant no. S002283-USAID. The information, data, or work presented herein was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0001211 and DE-AR0001222. We also gratefully acknowledge the U.S. Agency for International Development through Penn State for grant no. S002283-USAID. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. This material was supported by The Research Council of Norway and Equinor ASA through Research Council project "308817 - Digital wells for optimal production and drainage". Research was sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

- [1] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 1, pp. 1–17, 1990.
- [2] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney *et al.*, *LAPACK users' guide*. SIAM, 1999.
- [3] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, "SUNDIALS," *ACM Transactions on Mathematical Software*, vol. 31, no. 3, pp. 363–396, sep 2005. [Online]. Available: <https://doi.org/10.1145%2F1089014.1089020>
- [4] C. Rackauckas and Q. Nie, "DifferentialEquations.jl—a performant and feature-rich ecosystem for solving differential equations in julia," *Journal of Open Research Software*, vol. 5, no. 1, 2017.
- [5] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.
- [6] K. Burrage, *Parallel and sequential methods for ordinary differential equations*. Clarendon Press, 1995.
- [7] "XBraid: Parallel multigrid in time," <http://lnl.gov/casc/xbraid>.
- [8] J. Schroder and R. Falgout, "XBraid tutorial," in *18th Copper Mountain Conference on Multigrid Methods*. Colorado, 2017.
- [9] U. Nowak, R. Ehrig, and L. Oeverdieck, "Parallel extrapolation methods and their application in chemical engineering," in *International Conference on High-Performance Computing and Networking*. Springer, 1998, pp. 419–428.
- [10] T. A. Davis, "Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method," *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 2, pp. 196–199, 2004.
- [11] T. Davis, W. Hager, and I. Duff, "Suitesparse," *URL: faculty.cse.tamu.edu/davis/suitesparse.html*, 2014.
- [12] L. Petzold, "Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations," *SIAM journal on scientific and statistical computing*, vol. 4, no. 1, pp. 136–148, 1983.
- [13] D. Ketcheson and U. bin Waheed, "A comparison of high-order explicit runge–kutta, extrapolation, and deferred correction methods in serial and parallel," *Communications in Applied Mathematics and Computational Science*, vol. 9, no. 2, pp. 175–200, 2014.
- [14] E. Hairer, S. P. Norsett, and G. Wanner, "Solving ordinary differential equations i: Nonsti problems, volume second revised edition," 1993.
- [15] Hairer and Peters, *Solving Ordinary Differential Equations II*. Springer Berlin Heidelberg, 1991.
- [16] K. Althaus, "Theory and implementation of the adaptive explicit midpoint rule including order and stepsize control," 2018. [Online]. Available: <https://github.com/AlthausKonstantin/Extrapolation/blob/master/Bachelor%20Thesis.pdf>
- [17] A. Aitken, "On interpolation by iteration of proportional parts, without the use of differences," *Proceedings of the Edinburgh Mathematical Society*, vol. 3, no. 1, pp. 56–76, 1932.
- [18] E. H. Neville, *Iterative interpolation*. St. Joseph's IS Press, 1934.
- [19] P. Deuflhard, "Order and stepsize control in extrapolation methods," *Numerische Mathematik*, vol. 41, no. 3, pp. 399–422, 1983.
- [20] W. Rombert, "Vereinfachte numerische integration," *Norske Vid. Selsk. Forh.*, vol. 28, pp. 30–36, 1955.
- [21] R. Bulirsch and J. Stoer, "Numerical treatment of ordinary differential equations by extrapolation methods," *Numerische Mathematik*, vol. 8, no. 1, pp. 1–13, 1966.
- [22] H. J. Stetter, "Symmetric two-step algorithms for ordinary differential equations," *Computing*, vol. 5, no. 3, pp. 267–280, 1970.
- [23] J.-P. Berrut and L. N. Trefethen, "Barycentric lagrange interpolation," *SIAM review*, vol. 46, no. 3, pp. 501–517, 2004.
- [24] M. Webb, L. N. Trefethen, and P. Gonnet, "Stability of barycentric interpolation formulas for extrapolation," *SIAM Journal on Scientific Computing*, vol. 34, no. 6, pp. A3009–A3015, 2012.
- [25] G. G. Dahlquist, "A special stability problem for linear multistep methods," *BIT Numerical Mathematics*, vol. 3, no. 1, pp. 27–43, 1963.
- [26] G. Bader and P. Deuflhard, "A semi-implicit mid-point rule for stiff systems of ordinary differential equations," *Numerische Mathematik*, vol. 41, no. 3, pp. 373–398, 1983.
- [27] W. B. Gragg, "Repeated extrapolation to the limit in the numerical solution of ordinary differential equations," CALIFORNIA UNIV LOS ANGELES, Tech. Rep., 1964.
- [28] B. Lindberg, "On smoothing and extrapolation for the trapezoidal rule," *BIT*, vol. 11, no. 1, pp. 29–52, mar 1971. [Online]. Available: <https://doi.org/10.1007%2Fbfb01935326>
- [29] C. Elrod, "Polyester.jl," <https://github.com/JuliaSIMD/Polyester.jl>, 2021.
- [30] E. Hairer, "Fortran and matlab codes," 2004.
- [31] J. H. Verner, "Numerically optimal runge–kutta pairs with interpolants," *Numerical Algorithms*, vol. 53, no. 2, pp. 383–396, 2010.
- [32] H. Robertson, "Numerical integration of systems of stiff ordinary differential equations with special structure," *IMA Journal of Applied Mathematics*, vol. 18, no. 2, pp. 249–263, 1976.
- [33] A. M. Zhabotinsky, "Belousov-zhabotinsky reaction," *Scholarpedia*, vol. 2, no. 9, p. 1435, 2007.
- [34] R. J. Field and R. M. Noyes, "Oscillations in chemical systems. iv. limit cycle behavior in a model of a real chemical reaction," *The Journal of Chemical Physics*, vol. 60, no. 5, pp. 1877–1884, 1974.
- [35] E. Schäfer, "A new approach to explain the "high irradiance responses" of photomorphogenesis on the basis of phytochrome," *Journal of Mathematical Biology*, vol. 2, no. 1, pp. 41–56, 1975.
- [36] J. G. Verwer, "Gauss–seidel iteration for stiff odes from chemical kinetics," *SIAM Journal on Scientific Computing*, vol. 15, no. 5, pp. 1243–1250, 1994.
- [37] G. Bidkhor, A. Moeini, and A. Masoudi-Nejad, "Modeling of tumor progression in nsccl and intrinsic resistance to tki in loss of pten expression," *PloS one*, vol. 7, no. 10, p. e48004, 2012.
- [38] K. Soetaert, T. Petzoldt, and R. W. Setzer, "Solving differential equations in R: package desolve," *Journal of statistical software*, vol. 33, pp. 1–25, 2010.
- [39] 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*, vol. 17, no. 3, pp. 261–272, 2020.
- [40] L. F. Shampine and M. W. Reichelt, "The matlab ode suite," *SIAM journal on scientific computing*, vol. 18, no. 1, pp. 1–22, 1997.
- [41] H. Rosenbrock, "Some general implicit processes for the numerical solution of differential equations," *The Computer Journal*, vol. 5, no. 4, pp. 329–330, 1963.

- [42] R. Alexander, "Diagonally implicit runge-kutta methods for stiff ode's," *SIAM Journal on Numerical Analysis*, vol. 14, no. 6, pp. 1006–1021, 1977.
- [43] C. Li, M. Donizelli, N. Rodriguez, H. Dharuri, L. Endler, V. Chelliah, L. Li, E. He, A. Henry, M. I. Stefan *et al.*, "Biomodels database: An enhanced, curated and annotated resource for published quantitative kinetic models," *BMC systems biology*, vol. 4, no. 1, pp. 1–14, 2010.
- [44] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden *et al.*, "The systems biology markup language (sbml): a medium for representation and exchange of biochemical network models," *Bioinformatics*, vol. 19, no. 4, pp. 524–531, 2003.
- [45] Y. Ma, S. Gowda, R. Anantharaman, C. Laughman, V. Shah, and C. Rackauckas, "Modelingtoolkit: A composable graph transformation system for equation-based modeling," 2021.

VI. APPENDIX

A. Models

The first test problem is the ROBER Problem:

$$\begin{aligned}\frac{dy_1}{dt} &= -k_1 y_1 + k_3 y_2 y_3, \\ \frac{dy_2}{dt} &= k_1 y_1 + -k_2 y_2^2 - k_3 y_2 y_3, \\ \frac{dy_3}{dt} &= k_2 y_2^2.\end{aligned}\quad (17)$$

The initial conditions are $y = [1.0, 0.0, 0.0]$ and $k = (0.04, 3e7, 1e4)$. The time span for integration is $t = (0.0 \text{ s}, 1e5 \text{ s})$. [32], [15] The second test problem is OREGO:

$$\frac{dy_1}{dt} = -k_1(y_2 + y_1(1 - k_2 y_1 - y_2)), \quad (18)$$

$$\frac{dy_2}{dt} = \frac{y_3 - (1 + y_1)y_2}{k_1}, \quad (19)$$

$$\frac{dy_3}{dt} = k_3(y_1 - y_3). \quad (20)$$

The initial conditions are $y = [1.0, 2.0, 3.0]$ and $k = (77.27, 8.375 \times 10^{-6}, 0.161)$. The time span for integration is $t = (0.0 \text{ s}, 30.0 \text{ s})$. The third test problem is HIRES:

$$\frac{dy_1}{dt} = -1.71y_1 + 0.43y_2 + 8.32y_3 + 0.0007, \quad (21)$$

$$\frac{dy_2}{dt} = 1.71y_1 - 8.75y_2, \quad (22)$$

$$\frac{dy_3}{dt} = -10.03y_3 + 0.43y_4 + 0.035y_5, \quad (23)$$

$$\frac{dy_4}{dt} = 8.32y_2 + 1.71y_3 - 1.12y_4, \quad (24)$$

$$\frac{dy_5}{dt} = -1.745y_5 + 0.43y_6 + 0.43y_7, \quad (25)$$

$$\frac{dy_6}{dt} = -280.0y_6y_8 + 0.69y_4 \quad (26)$$

$$+ 1.71y_5 - 0.43y_6 + 0.69y_7, \quad (27)$$

$$\frac{dy_7}{dt} = 280.0y_6y_8 - 1.81y_7, \quad (28)$$

$$\frac{dy_8}{dt} = -280.0y_6y_8 + 1.81y_7. \quad (29)$$

The initial conditions are:

$$y = [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0057]. \quad (30)$$

The time span for integration is $t = (0.0 \text{ s}, 321.8122 \text{ s})$. The fourth test problem is POLLU:

$$\frac{dy_1}{dt} = -k_1 y_1 - k_{10} y_{11} y_1 - k_{14} y_1 y_6 - k_{23} y_1 y_4 \quad (31)$$

$$- k_{24} y_{19} y_1 + k_2 y_2 y_4 + k_3 y_5 y_2 + k_9 y_{11} y_2 \quad (32)$$

$$+ k_{11} y_{13} + k_{12} y_{10} y_2 + k_{22} y_{19} + k_{25} y_{20}, \quad (33)$$

$$\frac{dy_2}{dt} = -k_2 y_2 y_4 - k_3 y_5 y_2 - k_9 y_{11} y_2 - k_{12} y_{10} y_2 \quad (34)$$

$$+ k_1 y_1 + k_{21} y_{19}, \quad (35)$$

$$\frac{dy_3}{dt} = -k_{15} y_3 + k_1 y_1 + k_{17} y_4 + k_{19} y_{16} + k_{22} y_{19}, \quad (36)$$

$$\frac{dy_4}{dt} = -k_2 y_2 y_4 - k_{16} y_4 - k_{17} y_4 - k_{23} y_1 y_4 + k_{15} y_3, \quad (37)$$

$$\frac{dy_5}{dt} = -k_3 y_5 y_2 + 2k_4 y_7 + k_6 y_7 y_6 + k_7 u_9 \quad (38)$$

$$+ k_{13} y_{14} + k_{20} y_{17} y_6, \quad (39)$$

$$\frac{dy_6}{dt} = -k_6 y_7 y_6 - k_8 y_9 y_6 - k_{14} y_{11} y_6 - k_{20} y_{17} y_6 \quad (40)$$

$$+ k_3 y_5 y_2 + 2k_{18} u_{16}, \quad (41)$$

$$\frac{dy_7}{dt} = -k_4 y_7 - k_5 y_7 - k_6 y_7 y_6 + k_{13} y_{14}, \quad (42)$$

$$\frac{dy_8}{dt} = k_4 y_7 + k_5 y_7 + k_6 y_7 y_6 + k_7 y_9, \quad (43)$$

$$\frac{dy_9}{dt} = -k_7 y_9 - k_8 y_9 y_6, \quad (44)$$

$$\frac{dy_{10}}{dt} = -k_{12} y_{10} y_2 + k_7 y_9 + k_9 y_{11} y_2, \quad (45)$$

$$\frac{dy_{11}}{dt} = -k_9 y_{11} y_2 - k_{10} y_{11} y_1 + k_8 y_9 y_6 + k_{11} y_{13}, \quad (46)$$

$$\frac{dy_{12}}{dt} = k_9 y_{11} y_2, \quad (47)$$

$$\frac{dy_{13}}{dt} = -k_{11} y_{13} + k_{10} y_{11} y_1, \quad (48)$$

$$\frac{dy_{14}}{dt} = -k_{13} y_{14} + k_{12} y_{10} y_2, \quad (49)$$

$$\frac{dy_{15}}{dt} = k_{14} y_{11} y_6, \quad (50)$$

$$\frac{dy_{16}}{dt} = -k_{18} y_{16} - k_{19} y_{16} + k_{16} y_4. \quad (51)$$

$$\frac{dy_{17}}{dt} = -k_{20} y_{17} y_6, \quad (52)$$

$$\frac{dy_{18}}{dt} = k_{20} y_{17} y_6, \quad (53)$$

$$\frac{dy_{19}}{dt} = -k_{21} y_{19} - k_{22} y_{19} - k_{24} y_{19} y_1 + k_{23} y_{11} y_4 + k_{25} y_{20}, \quad (54)$$

$$\frac{dy_{20}}{dt} = -k_{25} y_{20} + k_{24} y_{19} y_1. \quad (55)$$

$$\begin{aligned}k &= [0.35, 26.6, 12300.0, 0.00086, 0.00082, 15000.0, \\ &\quad 0.00013, 24000.0, 16500.0, 9000.0, 0.022, 12000.0, 1.88, \\ &\quad 16300.0, 4.8e6, 0.00035, 0.0175, 1.0e8, 4.44e11, \\ &\quad 1240.0, 2.1, 5.78, 0.0474, 1780.0, 3.12]. \\ y &= [0.0, 0.2, 0.0, 0.04, 0.0, 0.0, 0.1, 0.3, 0.017, 0.0, \\ &\quad 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.007, 0.0, 0.0, 0.0].\end{aligned}$$

The time span for integration is $t = (0.0\text{ s}, 60.0\text{ s})$.

The QSP model is "BIOMD0000000452 QSP Model, Bidkhor2012 - normal EGFR signalling" [37] from BioModels Database [43]. This is a Systems Biology Markup Model (SBML) [44], parsed using SBMLToolkit.jl and Catalyst.jl [4], [45]. It notably has 109 states and 188 parameters

B. Benchmarks

1) *100 Linear ODEs*: The test tolerances for the solvers in the benchmark Fig. 1 are relative tolerances in the range of $(10^{-7}, 10^{-13})$ and corresponding absolute tolerances are $(10^{-10}, 10^{-16})$.

TABLE I
TUNED PARAMETERS FOR 100 LINEAR ODES

Extrapolation Method/Order	minimum	initial	maximum
Midpoint Deuffhard	5	10	11
Midpoint Hairer Wanner	5	10	11

2) *Robber*: The test tolerances for the solvers in the benchmark Fig. 2 are relative tolerances in the range of $(10^{-7}, 10^{-9})$ and corresponding absolute tolerances are $(10^{-10}, 10^{-12})$. The integrator used for reference tolerance at 10^{-14} is CVODE_BDF [3]. The parameters for the solvers are tuned to these settings:

TABLE II
TUNED PARAMETERS FOR ROBER

Extrapolation Method/Order	minimum	initial	maximum
Implicit Euler	3	5	12
Implicit Euler Barycentric	4	5	12
Implicit Hairer Wanner	2	5	10

3) *Orego*: The test tolerances for the solvers in the benchmark Fig. 3 are relative tolerances in the range of $(10^{-7}, 10^{-9})$ and corresponding absolute tolerances are $(10^{-10}, 10^{-12})$. The integrator used for reference tolerance at 10^{-14} is Rodas5 [15]. The parameters for the solvers are tuned to these settings:

TABLE III
TUNED PARAMETERS FOR OREGO

Extrapolation Method/Order	minimum	initial	maximum
Implicit Euler	3	4	12
Implicit Euler Barycentric	3	4	12
Implicit Hairer Wanner	2	5	10

4) *Hires*: The test tolerances for the solvers in the benchmark Fig. 4 are relative tolerances in the range of $(10^{-7}, 10^{-9})$ and corresponding absolute tolerances are $(10^{-10}, 10^{-12})$. The integrator used for reference tolerance at 10^{-14} is Rodas5 [15]. The parameters for the solvers are tuned to these settings:

5) *POLLUTION*: The test tolerances for the solvers in the benchmark Fig. 5 are relative tolerances in the range of $(10^{-8}, 10^{-10})$ and corresponding absolute tolerances are $(10^{-10}, 10^{-13})$. The integrator used for reference tolerance at 10^{-14} is CVODE_BDF [3]. The parameters for the solvers are tuned to these settings:

TABLE IV
TUNED PARAMETERS FOR HIRES

Extrapolation Method/Order	minimum	initial	maximum
Implicit Euler	4	7	12
Implicit Euler Barycentric	4	7	12
Implicit Hairer Wanner	3	6	10

TABLE V
TUNED PARAMETERS FOR POLLU

Extrapolation Method/Order	minimum	initial	maximum
Implicit Euler	5	6	12
Implicit Euler Barycentric	5	6	12
Implicit Hairer Wanner	3	6	10

6) *QSP Model*: The test tolerances for the solvers in the benchmark Fig. 6 are relative tolerances in the range of $(10^{-6}, 10^{-10})$ and corresponding absolute tolerances are $(10^{-9}, 10^{-13})$. The integrator used for reference tolerance at 10^{-14} is CVODE_BDF [3]. The parameters for the solvers are tuned to these settings:

TABLE VI
TUNED PARAMETERS FOR QSP MODEL

Extrapolation Method/Order	minimum	initial	maximum
Implicit Euler	8	9	12
Implicit Euler Barycentric	7	8	12
Implicit Hairer Wanner	2	5	10