MODE: A Modern ODE Solver for C++ and CUDA

Ali Can Demiralp[1], Marcel Krüger[1] and Tim Gerrits[1]

[1]*Visual Computing Institute, RWTH Aachen*

Abstract. Ordinary differential equations (ODE) are used to describe the evolution of one or more dependent variables using their derivatives with respect to an independent variable. They arise in various branches of natural sciences and engineering. We present a modern, efficient, performance-oriented ODE solving library built in C++20. The library implements a broad range of multi-stage and multi-step methods, which are generated at compile-time from their tableau representations, avoiding runtime overhead. The solvers can be instantiated and iterated on the CPU and the GPU using identical code. This work introduces the prominent features of the library with examples.

INTRODUCTION

An ordinary differential equation (ODE) is an equation containing the derivative of one or more dependent variables with respect to an independent variable. An nth order ODE is written in normal form as:

$$x^{(n)} = f(t, x, x', x'', ..., x^{(n-1)}) \tag{1}$$

where t is an independent variable and x is a dependent variable. Such equations appear in various branches of natural sciences and engineering where they are commonly used to model dynamic systems. As exact analytical solutions are often not possible, numerical methods play an important role in estimating solutions to ODEs. Hence a lot of recent literature [1, 2, 3, 4, 5] focuses on development of general and efficient numerical solvers.

This work presents a modern, efficient, performance-oriented ODE solver built in C++20. The solver contains a wide range of explicit as well as implicit multi-stage and linear multi-step methods, which are constructed from their tableau representations as defined in [6] at compile-time. It is fully compatible with CUDA, enabling CPU and GPU solvers with identical code. Adaptive step size iterators are supported, including implementations of standard error controllers. Moreover the library is capable of decomposing higher order ordinary differential equations to a system of first order equations, and iterate them in a coupled manner. Aside from initial value problems, boundary value problems are supported through the shooting method. We describe the prominent features of the library, and demonstrate its usage.

RELATED WORK

Various ODE solvers were presented in the last decades. The GNU Scientific Library [7] provides several multi-stage and multi-step ODE solvers, yet it is limited to low order methods and does not prioritize performance. The SUNDIALS [1] suite contains ARKODE [2] and CVODE [3], which implement adaptive Runge-Kutta and linear multi-step methods respectively. Although SUNDIALS covers a wide range of methods and is highly optimized, it hardly utilizes modern programming idioms such as RAII due to being a C interface. More recently, Boost.Odeint [4, 8] was proposed as an alternative built in C++. The library leverages language features such as function objects and template meta-programming to create solvers that are capable of operating on arbitrary states. Although Boost.Odeint has modernized solving ODEs, it remains as a C++11 library, with updates being limited to bug fixes. We build upon the concepts introduced by Boost.Odeint, incorporating C++14/17/20 language features to the context of solving ODEs. Aside from C and C++, there are libraries in several other languages. Notably, Sci.py and DiffEq.jl [5] provide ODE solvers in Python and Julia respectively. Although these languages are capable of interfacing with C++, the process often leads to verbose code and introduces performance overhead.

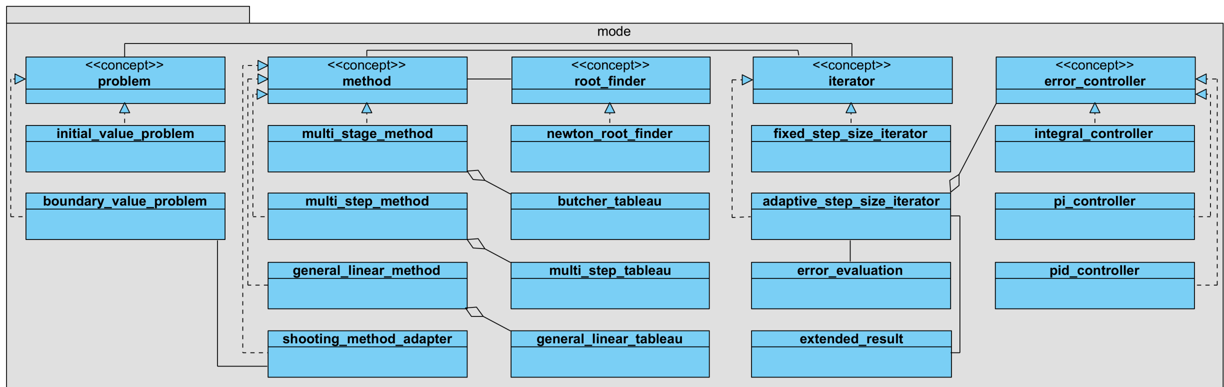

FIGURE 1. An outline of the library. The *iterators* apply *methods* to *problems* to generate solutions.

IMPLEMENTATION

An outline of the library is seen in Figure 1. We refrain from inventing new abstractions, and attempt to adapt existing mathematical concepts to code directly. The three central concepts a user interacts with are the *iterator*, the *method* and the *problem*. In an overview: The fixed- and adaptive-step *iterator*s apply explicit and implicit *method*s to initial and boundary value *problem*s to generate solutions. The library is built with emphasis on four system quality attributes:

- **Generality**: It operates on initial and boundary value problems of arbitrary order, with dependent variables consisting of arbitrary precision numbers, including complex numbers.
- **Efficiency**: It strives to match the performance of existing solvers, avoiding overhead introduced by the higher level abstractions using compile-time constant evaluations where possible.
- **Portability**: It runs on any platform and compiler that supports the C++20 standard library, and is also compatible with CUDA.
- **Usability**: It provides a uniform interface to all multi-step and multi-stage iterative methods for ordinary differential equations, using abstractions that match the theory.

The *problem* concept encapsulates the (initial) state of an ODE problem. Although problems may not have common variables or methods, this concept enables the library components to require a problem as input. For example, a *method* may require a *problem* to operate on, and further adjust its behavior based on the type of that problem. The library currently implements two problems: the *initial_value_problem* and the *boundary_value_problem*. The users provide the data, such as the initial/boundary values and the derivative functions, to the library in the form of problems. The values and the functions may be references, meaning that the library does not require a copy of the initial state. The problems are solved in-place by the *iterators* to avoid additional memory allocation. They contain the intermediate and final solutions throughout iteration. The intermediate states can be saved manually by the user as necessary.

The *method* concept describes a numerical method for solution of ODEs. It requires a single function for applying one iteration of the method. The *multi_stage_method*, *multi_step_method* and *general_linear_method* implement this concept, accepting a *tableau* and constructing the said function at compile-time. The *multi_stage_method* distinguishes between explicit and implicit methods according to the size of the tableau coefficients, which are expressed in lower triangular form for explicit methods. It furthermore detects if the tableau contains an error estimate row and applies the additional stage, modifying its return type at compile-time to contain the error accordingly. The *multi_step_method* constructs an implicit method if the tableau defines both the a and b coefficients, and an explicit method if it only defines the a coefficients.

The *tableau* concept encapsulates the coefficients that are necessary to construct a *method*. In the case of multi-stage methods, the coefficients are of an (extended) Butcher tableau. For linear multi-step methods, the coefficients are the vectors a and b appearing on the two sides of the equation. The library further contains tableau definitions for general linear methods as described by Butcher in [6]. This, in turn, enables methods which combine multiple stages with multiple steps. Aside from the coefficients, the tableau may optionally contain information about the (extended) order of the method, which is utilized by several error controllers to compute default parameters. Overall, the tableau

```cpp
#include <mode/mode.hpp>

using method_type  = mode::explicit_method<mode::runge_kutta_4_tableau<float>>;
using problem_type = mode::initial_value_problem<float, vector3f>;

int main(int argc, char** argv)
{
  const auto problem = problem_type
  {
    0.0f,                                       /* t0 */
    vector3f(1.0f, 1.0f, 1.0f),                 /* y0 */
    [ ] (const float t, const vector3f& y) /* dy/dt = f(t, y) */
    {
      const auto sigma = 10.0f;
      const auto rho   = 28.0f;
      const auto beta  = 8.0f / 3.0f;
      return vector3f(sigma*(y[1]-y[0]), y[0]*(rho-y[2])-y[1], y[0]*y[1]-beta*y[2]);
    }
  };

  auto iterator = mode::fixed_step_iterator<method_type>(problem, 0.001f /* h */);
  for (auto i = 0; i < 1000; ++i)
    ++iterator;
}
```

Listing 1. A simple example iterating the Lorenz system.

concept defines the distinct properties of a method in a concise and uniform manner. From the viewpoint of a user, the tableau is the only concept to be implemented to support custom multi-stage and multi-step methods.

The library contains a wide range of tableaux for explicit and implicit methods. Common explicit solvers such as Runge-Kutta 4, Dormand-Prince 5, Bogacki-Shampine 5 and Tsitouras 5 are included. For stiff problems, implicit solvers such as Radau IA/IIA and Rodas 4 can be used. Regarding multi-step methods, the library implements a general version of Adams-Bashforth for arbitrary number steps, as well as the backward differentiation formula for any order. For a complete list of the implemented tableaux, we refer the reader to the repository. Although the number of tableaux are a fraction of what is provided by DiffEq.jl [5], the selection is nevertheless larger and more comprehensive than most C and C++ solvers to date.

As the library implements iterative methods for ODEs, it is important to define a concept of iteration. We borrow the definition of a forward read-only iterator from the standard library, rather than creating a custom concept. The iterators accept a method and a problem, and are responsible for iterating the method on the problem. We provide three realizations of this concept: *fixed_step_size_iterator*, *adaptive_step_size_iterator* and *coupled_iterator*. The *fixed_step_size_iterator* can be used with any method, whereas the *adaptive_step_size_iterator* is reserved for embedded methods that provide an error estimate. It is capable of adjusting its step size based on the error, and may potentially perform multiple trials per iteration. The *coupled_iterator* accepts an array of first order problems obtained by decomposing a higher order problem, and iterates them simultaneously.

The adaptive step size iterators additionally require a type satisfying the *error_controller* concept. The error controllers accept the results and the error estimate of an embedded method, decide whether to accept the current step, and propose a size for the next step. The library implements several standard error controllers, including the integral [9], proportional integral (PI) [10], and proportional integral derivative (PID) [11] controllers. The error controllers include default parameters and common settings that are computed from the method order according to literature [9] [10]. It is also possible to define custom error controllers by implementing the concept and passing it to the iterators.

The implicit methods require root finding algorithms as they solve a system of nonlinear algebraic equations in each step. The *root_finder* concept encapsulates such algorithms. The library provides an implementation of the Newton method satisfying this concept, along with a modified version which evaluates the Jacobian once instead of at each iteration. With the exception of linear subspace solvers, root finders depend on and therefore require the problems to define an additional derivative. Furthermore, they rely on linear algebraic operations for computing the inverse of the Jacobian. For this purpose, we utilize Eigen3 [12] as it provides the necessary operations, is highly optimized, and supports CUDA.

An example application using the library for solving the Lorenz system is seen in Listing 1. It is possible to instantiate and iterate the problem on the GPU using identical code. The library does not explicitly provide parallelism, but instead lends itself for use by parallel C++ and CUDA applications. This is an intentional decision based on our

experience with the parallelization features of [4], which tend to limit actions such as early termination of states that fulfill a certain condition. High-performance computing applications that consume the library are expected to rely on broader parallelization schemes than one that is limited to solving ODEs. Our intent is to integrate into such schemes, rather than forcing a scheme that is potentially orthogonal to them. Users seeking a quick parallelization over problems can use OpenMP, which is provided by most compilers today.

CONCLUSION & FUTURE WORK

We have presented a library for solving ODEs in modern C++. The library provides several benefits in comparison to its predecessors: It is capable of generating multi-stage and multi-step methods from their Butcher tableaux at compile-time. It utilizes constant-evaluated contexts to move run-time overhead to compile-time where viable. It strives for full compatibility with CUDA, enabling CPU and GPU solvers with identical code. It is further capable of solving initial and boundary value problems of arbitrary order through coupling. The library is open-source software distributed under the BSD 3-Clause license, accessible at https://github.com/vrgrouprwth/mode.

In the future, we would like to benchmark the performance of the library and compare it to its alternatives. We further plan on expanding the selection of Butcher tableau with the methods presented in [5, 6, 9, 10]. Automatic computation of Jacobians via algorithmic differentiation or finite differences for use with implicit methods remains as future work. Another direction is adding support for stochastic differential equations (SDEs), random differential equations (RDEs), delay differential equations (DDEs), differential algebraic equations (DAEs), and stochastic differential algebraic equations (SDAEs).

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the German Federal Ministry of Education and Research (BMBF) and the individual state governments for supporting this work/project as part of the NHR funding.

REFERENCES

[1] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, ACM Transactions on Mathematical Software **31**, 363–396 (2005).

[2] D. R. Reynolds, D. J. Gardner, C. S. Woodward, and R. Chinomona, arXiv:2205.14077 (2022), 10.48550/ARXIV.2205.14077.

[3] S. D. Cohen, A. C. Hindmarsh, and P. F. Dubois, Computers in Physics **10**, 138–143 (1996).

[4] K. Ahnert and M. Mulansky, “Odeint – Solving Ordinary Differential Equations in C++,” in *International Conference on Numerical Analysis and Applied Mathematics 2011*, AIP Conference Proceedings 1389, edited by C. T. Theodore E. Simos, George Psihoyios and Z. Anastassi (American Institute of Physics, Halkidiki, Greece, 2011), pp. 1586–1589.

[5] C. Rackauckas and Q. Nie, Journal of Open Research Software **5** (2017), 10.5334/jors.151.

[6] J. C. Butcher, *Numerical Methods for Ordinary Differential Equations*, 3rd ed. (John Wiley & Sons, Chichester, 2003).

[7] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, F. Rossi, and R. Ulerich, *GNU Scientific Library Reference Manual*, 3rd ed. (Network Theory Limited, Godalming, 2009).

[8] K. Ahnert, D. Demidov, and M. Mulansky, “Solving Ordinary Differential Equations on GPUs,” in *Numerical Computations with GPUs*, edited by V. Kindratenko (Springer International Publishing, Cham, Germany, 2014), pp. 125–157.

[9] E. Hairer, S. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*, 2nd ed. (Springer, Berlin, 1993).

[10] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, 2nd ed. (Springer, Berlin, 1996).

[11] H. Ranocha, L. Dalcin, M. Parsani, and D. Ketcheson, Communications on Applied Mathematics and Computation (2021), 10.1007/s42967-021-00159-w.

[12] G. Guennebaud, B. Jacob, *et al.*, Eigen: A C++ Linear Algebra Library, 2010.