C++ Playground for Numerical Integration Method Developers

Stepan Orlov

Computer Technologies in Engineering dept.
Peter the Great St. Petersburg Polytechnic University
St. Petersburg, Russian Federation
majorsteve@mail.ru

Abstract. A C++ framework for investigating numerical integration methods for ordinary differential equations (ODE) is presented. The paper discusses the design of the software, rather than the numerical methods. The framework consists of header files defining a set of template classes. Those classes represent key abstractions to be used for constructing an ODE solver and to monitor its behavior. Several solvers are implemented and work out-of-the-box. The framework is to be used as a playground for those who need to design an appropriate numerical integration method for the problem at hand. An example of usage is provided. The source code of the framework is available on GitHub under the GNU GPL license.

Keywords: C++ · Extensible framework · Object-oriented programming · Numerical integration · Differential equations

1 Introduction

In this paper we consider the numerical solution of the initial value problem for a system of ordinary differential equation in the normal form:

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}), \quad \mathbf{x}\big|_{t=t_0} = \mathbf{x}_0, \tag{1}$$

where $\mathbf{x} = [x_1, \ldots, x_n]^T$ is the vector of n state variables of the system, t is the time, dot denotes the time derivative, and $\mathbf{f}$ is the ODE right hand side vector. Sometimes we also consider a more general case of ODE, namely

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}, \phi), \quad \mathbf{x}\big|_{t=t_0} = \mathbf{x}_0, \quad \phi\big|_{t=t_0} = \phi_0, \tag{2}$$

where $\phi = [\phi_1, \ldots, \phi_m]^T$ is the vector of m discrete state variables. Each of the variables ϕ_k may only change at discrete time instants identified by the relation

$$e_k(t, \mathbf{x}) = 0, \quad k = 1, \ldots, m; \tag{3}$$

functions e_k are called *event indicators*, and time instants satisfying (3) are called *events*. How the variables ϕ change is determined by a state machine that has

to be defined along with the ODE system and event functions. See [1] for more information.

Of course, there is software, including open source software, implementing numerical solvers of ODE initial value problem. For example, the SUNDIALS software suite [2] is capable of solving problems (1) and (2), and also differential-algebraic equations. The `odeint` library [3] from the `boost` project [4] provides many solvers for problems (1). There are many more, written in different programming languages. Nevertheless, we chose to create another piece of software for solving ODE initial value problem, named `ode_num_int`[1]. While existing software is focused on obtaining the numerical solution, the idea behind our framework is different. Our goal is to provide user with flexible components to create new solvers and to investigate how these solvers behave.

The `ode_num_int` framework currently covers single-step numerical integration schemes, including Runge–Kutta schemes [5]. It also provides a template class for building solvers based on Richardson extrapolation [5, ch. II.9]. It contains code to solve linear and nonlinear algebraic equations, so explicit and implicit schemes are easily constructed. The framework is designed for systems of medium scale, with n up to several thousands.

While the implementation of explicit ODE solvers is typically straightforward and requires no special tuning to get them work, the implementation of implicit solvers may require a lot of effort from a developer, especially if the goal is to have an efficient solver in terms of CPU time consumed. The latter applies to fully implicit schemes such as SDIRK [5, ch. II.7] and to linearly implicit methods of Rosenbrock type [6], such as W-methods [7]. Notice that both classes of methods require the Jacobian of ODE right hand side, $\mathbf{J} = D\mathbf{f}/D\mathbf{x}$, or an approximation to it, which we denote as $\mathbf{A}$. When a researcher starts applying such a method to the problem at hand, he or she may face a number of difficulties listed below.

- The quality of solution obtained with a linearly implicit W-method may depend on how close $\mathbf{A}$ is to $\mathbf{J}$. It is therefore natural for a developer to experiment with different strategies to update the $\mathbf{A}$ matrix. It should also be noticed that keeping the same $\mathbf{A}$ for as many time steps as possible is what can make a W-method outperform any other implicit method. Each time $\mathbf{A}$ changes, and each time the step size h changes, the matrix $\mathbf{W} = \mathbf{I} - hd\mathbf{A}$ (where $\mathbf{I}$ is the identity matrix and d is a parameter) has to be factorized, which consumes CPU time.
- The ways of calculation of the Jacobian matrix $\mathbf{J}$ may be different. For some ODE systems, it is easy to provide an explicit formula for $\mathbf{J}(t, \mathbf{x})$; for more complicated systems, it may still be possible to compute $\mathbf{J}$ analytically with an automatic differentiation tool like ADOL-C [8]; for complex systems, one has to compute the Jacobian numerically using finite differences.
- The Jacobian may happen to be a sparse matrix, and taking its sparsity structure into account during its numerical calculation and the solution of linear system becomes crucial for overall performance of a solver, as soon as n

[1] The source code of the framework is available at https://github.com/deadmorous/ode_num_int

is not too small. In addition, sparse structure of the Jacobian imposes certain constraints on the choice of algorithms to update its approximation $\mathbf{A}$.
– The convergence and performance of Newton-type method [9] used to solve nonlinear algebraic system at a time step may depend on several factors. It might require too many iterations to converge if $\mathbf{A}$ is not updated frequently enough; on the other hand, it may take too long time if we enforce $\mathbf{A} = \mathbf{J}$; it also may fail to converge unless a specific regularization strategy is applied. Last but not least, the number of iterations may strongly depend on the initial guess to the solution.

The `ode_num_int` framework has been designed with the idea to equip developers with useful abstractions helping to build problem-specific solvers that best fit the ODE system at hand. The construction of such a solver is often an investigation and requires from a developer to try various combinations of components and algorithms and to observe how the resulting solver works.

2 Software Design

The framework is written in C++11 and is a set of template classes. There are also a few translation units that support the dynamic creation of instances and implement some timing utilities. Other functionality is implemented in header files. Subsections below outline framework components and actually explain how an extensible system can be designed in C++.

2.1 Common Infrastructure

In this subsection, we describe how some general design patterns are employed in the `ode_num_int` framework and how they help to build consistent easy-to-use software.

Observers. In C++, functions are not really first-class citizens, like, e.g., in JavaScript. However, C++ allows classes (so called functors) pretending to be functions; moreover, C++11 allows to easily create functors using lambda expressions. The `Observers` template class represents an array of such functors with certain signature, which is the template parameter pack. User can add to or remove from this array using corresponding methods. Besides, `Observers` itself is a functor. When it is invoked, it in turn invokes all functors from the array. This pattern exists in other programming languages and is similar to `Boost.Signals2`, but our implementation is more lightweight.

Interfaces make use of the `Observers` template class by declaring public fields where interface users can add arbitrary callbacks. Interface method implementations invoke the callbacks by “calling” those fields (they are functors).

Property Holders. To hold a member variable, a C++ class can simply declare it. However, it is a good practice to keep the field private and use getter and setter methods to access the field; it is also sometimes desirable to notify any interested party about member modification when the setter is called. Another desirable thing is being able to hold similar members in different classes. An elegant solution to this is to put member declaration, as well as getter and setter code, into a separate class, and to inherit that class. This way we also follow the single responsibility principle. Since there might be many different things to be stored like this, the type of the member and the names of getter and setter should be different in each certain case, we have come to the solution using a preprocessor macro to declare such classes. We call such classes *property holders*.

Factories. Factory is a well-known design pattern used to provide a way to create instances of classes that implement certain interface [10]. Implementations do not need to be known when the factory is designed and even when their instances are created: the exact type of the instance is identified, e.g., by a number or a string.

In our framework, there are two template classes to support the pattern. We have at most one factory per interface, therefore each interface for a dynamically creatable entity inherits the `Factory` template class and gives it itself as the template parameter. On the other hand, each creatable implementation of the interface inherits the `FactoryMixin` template class instantiated with two template parameters, class type and interface type. Finally, each creatable type has to be registered in the factory, which can be done with a macro declaring a static registrator variable, or in a number of different ways.

Optional Parameters. It is often necessary to provide parameter values for object instances. Parameter types could be numbers (e.g., a tolerance), strings (e.g., a file name), or typed objects (e.g., an ODE solver). When parameters are specified directly from C++ code, there is no problem. However, it might be necessary to read all parameters from a file and set all of them to appropriate objects, and also to create objects by type identifiers found in the file. To support this, two classes have been designed, `OptionalParameters` and `Value`. The former one is an interface declaring methods to read parameters from the object and to set parameters. The latter one is for storing a single value of arbitrary type; it is similar to to the `QVariant` type from the Qt library [11], although it has a feature providing interoperability with factories: if a value is a pointer to an interface with a factory, assigning a string to it leads to the creation of appropriate instance, followed by assigning the created instance to the value. The string in this case is treated as a type identifier. As long as any type can be stored in a `Value`, it is easy to have a tree of parameters and to organize its transformation to any suitable format, e.g., XML, JSON, or plain text.

Timing Utilities. In software engineering, the standard tool used to identify performance bottlenecks is a profiler. However, since the performance of an ODE

solver and its parts is always an important issue for a developer, we included a few lightweight tools to measure time intervals. The `TickCounter` class has methods to count number of CPU cycles between method invocations. Its design is similar to that of `QTimer` from Qt, but the measurement is much more precise due to the use of the `rdtsc` instruction. On multi-core AMD CPUs, however, this approach requires pinning threads to CPU cores. To convert CPU cycle counts into milliseconds, the `TimerCalibrator` class can be used.

The `TimingStats` class is convenient to manage timing statistics for multiple invocation of the same code. It counts the total time, the invocation count, and can compute average time per one invocation.

The `TickCounter` and `TimingStats` classes can be used in combination with `Observers` to easily measure the time spent in any part of solver code.

2.2 Linear Algebra

There are several implementations of linear algebra code (e.g., Intel MKL or AMD ACML libraries). In order to simplify building from source code, the `ode_num_int` framework does not depend on any of them and implements a minimal set of linear algebra operations internally. At the same time the design of template class for a vector allows interoperability with such implementations.

The `VectorTemplate` template class provides an interface to a column vector. It defines linear operations, such as addition and multiplication by a number, and provides their reference implementations. There is one template parameter, `VectorData`, that determines how to access actual data in a vector, element type, and how vector data is copied. Different instantiations of vector template can be seamlessly assigned to each other, which improves code flexibility. The `ode_num_int` framework provides an implementation of `VectorData` that stores the data in `std::vector`. Other implementations are possible. Such design helps to avoid data copying in some cases, e.g., when it is necessary to represent a part of a vector (for example, its upper half) as another vector. To achieve this, we use the `VectorProxy::Block` as an implementation of `VectorData`, thus employing the *proxy* pattern [10]. Another similar example is when we need a scaled vector. To avoid immediate copying and multiplication, the `VectorProxy::Scale` class is used in place of `VectorData`. Importantly, `VectorData` is a template parameter for most of the template classes of the framework.

The `SparseMatrixTemplate` template class provides an interface to a sparse matrix, as follows from its name. Its design is also split into the interface class with the above name and its template parameter, `MatrixData`. Rectangular blocks of sparse matrices can be represented as separate matrices without copying by using proxies similar to those for vectors. To actually store matrix data, there are two implementations of `MatrixData`. One of them (call it `D` to be short) allows dynamically changing the sparsity pattern and stores matrix elements in `std::map`. Matrix operations, like matrix-vector multiplication, require iteration over an associative array, and are not really fast in this case. For fixed matrix sparsity pattern, there is a faster solution (call it `F`), with matrix elements stored in `std::vector`. Like for vectors, different instantiations of sparse

matrices can be seamlessly assigned to each other, which allows the following usage pattern, e.g., to compute the Jacobian. If sparsity pattern is not known, use `SparseMatrixTemplate<D>` to compute it, but don't store the resulting matrix; instead, store an instance of type `SparseMatrixTemplate<F>` and copy the first matrix into it. Then perform operations on the second matrix. When the Jacobian is computed next time and its sparsity pattern remains the same, take it into account to compute the Jacobian (probably much faster [12]); do not use `D` at all until the sparsity pattern changes.

The `LUFactorizer` template class is capable of solving sparse linear systems using the LU factorization [13]. It has a method to specify a sparse matrix, another method to update matrix with the same sparsity pattern (faster than the first one because doesn't require memory allocation and the calculation of layout of matrices $\mathbf{L}$ and $\mathbf{U}$), and a method to solve linear system with the specified right hand side. The factorization is done when necessary. The class automatically manages timing statistics for setting the matrix, decomposition, and backward substitution using `TimingStats` fields.

2.3 Nonlinear Algebraic Newton-type Solver

This section presents abstractions specific to the solution of systems of nonlinear equations at a time step of an implicit ODE solver, as well as several implementations. The functionality is split into interchangeable components in order to provide great flexibility in combining different algorithms. The components naturally follow the single responsibility principle. Importantly, interfaces described in this and next subsections provide relevant observers (not described here due to size limitation) that can be exploited to pull all necessary information from components and use it for understanding how they perform.

An iteration of a Newton-type solver for the equation $\mathbf{f}(\mathbf{x}) = 0$ with the initial guess $\mathbf{x}^{(k)}$ can be written as follows [9]:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{d}^{(k)}, \quad \mathbf{A}\mathbf{d}^{(k)} = -\mathbf{f}(\mathbf{x}^{(k)}), \tag{4}$$

where $\mathbf{A}$ is an approximation to the Jacobian $\mathbf{J} = D\mathbf{f}/D\mathbf{x}$, $\mathbf{d}^{(k)}$ is the *search direction*, and $\alpha^{(k)}$ is a number determined by a line search algorithm (see below).

Vector Mapping. To formulate the problem for a nonlinear solver, we use the `VectorMapping` template class that defines an interface for a vector function of a vector argument, $\mathbf{f}(\mathbf{x})$. Implementations provide actual mappings. A vector mapping can be held in a property holder (see above).

Error Estimator. The interface declares methods to compute the norms of absolute and relative error vectors of numerical solution obtained at an iteration of a Newton-type method. It also declares iteration status codes and a method returning such a code to instruct iteration performer (see below) what to do next. The default implementation of error estimator has absolute and relative tolerances and thresholds used to detect the divergence. Those are available through the `OptionalParameters` interface.

Jacobian Provider. The interface declares methods to compute the Jacobian matrix $\mathbf{J}$ and to retrieve it as a sparse matrix of type `SparseMatrixTemplate<F>` (see above), and also a method to inform the instance about possible change in the sparsity pattern after an event. It is up to the implementation how $\mathbf{J}$ is computed. Currently we have an implementation that computes $\mathbf{J}$ numerically, taking its sparsity pattern into account. The latter means that for mappings $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ the number of $\mathbf{f}$ evaluations could be much less than $n + 1$, which is the case for dense $\mathbf{J}$. See [12] for more information.

Jacobian Trimmer. One way to speed up the solution of the linear system in (4) is to reduce the number of nonzero elements in $\mathbf{A}$. This can be done by applying a trimmer transforming $\mathbf{J}$ into a matrix with a smaller number of nonzero elements by throwing away certain elements (for example, by limiting the bandwidth of the matrix). On the other hand, the trimming might severely impact the convergence of Newton iterations, hence much care must be taken with the trimming.

Descent Direction. The interface defines a method to compute the search direction $\mathbf{d}^{(k)}$ by solving the second of equations (4). This procedure may also involve an update of $\mathbf{A}$ according to certain strategy. Currently, a number of implementations are available: (i) $\mathbf{A} = \mathbf{J}$; (ii) $\mathbf{A}$ is subject to Broyden's update of rank 1 [14]; (iii) $\mathbf{A}$ is subject to "fake Broyden's update", such that the sparsity pattern of $\mathbf{A}$ is preserved by ignoring all elements of the update matrix that do not have corresponding nonzero elements in $\mathbf{A}$; (iv) Hart's update directly to the LU decomposition of $\mathbf{A}$ [15]; (v) constant $\mathbf{A}$; some more. Importantly, an implementation of the search direction interface may have an internal state changing between iterations. Therefore, there is a method telling it to reset the internal state and recompute the Jacobian next time.

Line Search. The interface provides a method to perform the search along the direction $\mathbf{d}^{(k)}$ and ensure $|\mathbf{f}(\mathbf{x}^{(k+1)})| < |\mathbf{f}(\mathbf{x}^{(k)})|$ by picking a suitable value of $\alpha^{(k)}$. Currently we have one simple implementation that starts with $\alpha^{(k)} = 1$ and divides it by two till the above condition is satisfied; if during this process $\alpha^{(k)}$ reaches a minimum threshold, the Newton iteration is considered as divergent.

Iteration Performer. The interface provides methods to specify initial guess to the solution, to perform a single iteration and to reset components. Most important, the iteration performer is a placeholder for components implementing the vector mapping specifying $\mathbf{f}(\mathbf{x})$, the error estimator, the descent direction algorithm, and the line search algorithm.

Regularization Tools. Unfortunately, Newton iterations may diverge. This problem can be addressed by introducing some kind of regularization. For example, instead of solving $\mathbf{f}(\mathbf{x}) = 0$, one could consider equations $\mathbf{g}(\mathbf{x}, \gamma) = 0$. The

regularization parameter γ could vary from 0 to 1, and $\mathbf{g}$ is such that, on the one hand, $\mathbf{g}(\mathbf{x}, 1) = \mathbf{f}(\mathbf{x})$, and, on the other hand, Newton iterations converge better as γ decreases. Then we could consider iterations with γ changing gradually from 0 to 1. There is a hope that this process converges because the leading iterations give better initial guess to the problem with $\gamma = 1$. Notice that in the case of ODE numerical integration, γ could be proportional to the step size, but other choices are possible, too. The iteration performer described above can hold the regularized mapping $\mathbf{g}$ and a *regularization strategy* object that decides how γ should vary depending on iteration status.

Newton-type Solver. The solver interface declares methods to specify the initial guess and to run Newton iterations. It is a placeholder for iteration performer, so the logics of a single iteration is out of its responsibility. The implementation of the solver just runs a loop in which it performs an iteration, and if there is no convergence so far, suggests regularization strategy to do something. When iteration count reaches certain limit (which is a parameter), the solver makes one more try with the Jacobian computed from scratch (for this, the solver instructs the iteration performer to reset any internal state of its components).

2.4 ODE Solvers

All ODE solvers implemented in our framework are considered to be single-step methods (though, the support for multistep methods is planned). Besides, we do not currently support methods of variable order, e.g., based on the extrapolation. The `OdeSolver` interface for a solver declares methods returning its order, specifying initial state, specifying initial step size, retrieving current state, and performing a single step (the latter is called `doStep`).

ODE Right Hand Side is specified by inheriting the `OdeRhs` interface. It is similar to `VectorMapping`, but is better suited for evaluating $\mathbf{f}(t, \mathbf{x})$ rather than $\mathbf{f}(\mathbf{x})$. In addition, keeping mechanical systems in mind, there is some support for second-order equations. Namely, the vector of state variables is considered to consist of coordinates $\mathbf{u}$, speeds $\mathbf{v}$ (those are vectors of size n_2), and first order variables $\mathbf{z}$ (a vector of size n_1): $\mathbf{x} = [\mathbf{u}^T, \mathbf{v}^T, \mathbf{z}^T]^T$, so the ODE right hand side is then $\mathbf{f}(t, \mathbf{x}) = [\mathbf{v}^T, \mathbf{f}_u^T, \mathbf{f}_v^T]^T$. Therefore, the interface is augmented with a method returning the number of coordinates; the total number of state variables is $n = 2n_2 + n_1$. This kind of representation allows to reduce the size of linear and nonlinear problems to be solved by an implicit solver at a step to $n_2 + n_1$, because $\mathbf{u}$ is easily excluded.

Explicit Solvers implemented in our framework can be formulated as Runge-Kutta schemes (ERK), see [5, ch. II]. There are two general implementations taking a Butcher tableau as the constructor parameter. One implementation considers schemes without automatic step size control (our framework implements

Euler and RK4). The other implementation considers embedded ERK schemes with automatic step size control (currently the framework implements the DOPRI45, DOPRI56, and DOPRI78 schemes). To implement an ERK solver, one has to inherit the appropriate implementation and provide the Butcher tableau.

The Gragg's explicit solver is attractive as the reference solver for extrapolation due to its symmetry property [5, ch. II.8]. Originally, it is a two-step method. Although it can be reformulated as an ERK scheme, there is no need for that. Our implementation treats the solver as a single-step one, but internally it takes two steps of half size in its `doStep` method implementation. The solver has an additional parameter instructing it to return smoothed current state, which is the average of two last states. This allows to build the extrapolation Gragg–Bulirsch–Stoer scheme with smoothing, as described in [5]. Our experience shows that in certain cases this solver gives better results than other explicit schemes.

Extrapolation-based Solver takes another ODE solver as the *reference solver*. Each step boils down to splitting the step interval of size h into smaller steps of sizes h/n_k and making smaller steps with the reference solver. The sequence of whole numbers n_k may vary and is specified as extrapolator parameter (we implemented the Romberg's, the Bulirsch's, and the harmonic sequences). The index k runs from 1 to certain number of stages, N, which we currently consider fixed. Finally, the Aitken–Neville's algorithm is applied to find the extrapolated solution. The details of the method can be found in [5, ch. II.9]. The extrapolation-based solver possesses great flexibility because an arbitrary reference solver and step sequence can be specified as its parameters. Another parameter is a flag telling the extrapolator if the reference solver is symmetric. In the same time, our implementation is not the most general one because the order of the method does not adjust automatically at run time.

Step Size Controller interface declares a method that suggests the value of step size to be used at the next step. The method requires current step size, scheme order, and error norm as parameters. Therefore, the interface can be used by embedded ERK schemes and by any other schemes that are capable of computing the error norm at a step. Simpler schemes would have to estimate error norm using an extrapolation-based approach. The implementation of the interface has a number of parameters, which are the acceptable tolerance and some more. Additional parameters of the step size controller may instruct it to keep the step size constant if possible. This may be very important, e.g., for W-methods, since a change in the step size also changes the matrix of linear system to be solved at step.

Event Controller interface declares two methods, one to be called in the beginning of the step, and the other one to be called at the end. Those methods check for events occurring within a step, find first of them according to (3), and

change discrete state variables ϕ in (2) according to the event occurred. Finally, the second method reports the actual step size, probably truncated, and provides some information about the event. The interpolation of ODE system state inside the step is beyond the event controller responsibility; therefore, an interpolator has to be specified as a parameter for the second method. To interpolate the state vector, one can use the linear interpolator; however, if solver supports dense output it can provide a better quality interpolator.

Linearly Implicit W-methods. The idea behind W-methods is to replace the system Jacobian $\mathbf{J}$ in Rosenbrock methods with its arbitrary approximation $\mathbf{A}$ and still to have method order conditions satisfied. When $\mathbf{A}$ is close enough to $\mathbf{J}$, the scheme is expected to have certain stability properties. Refer to [7] for more information. In our framework, two W-methods are currently implemented, the SW2(4) method [7] and the W1 method (actually used as the reference solver for extrapolator). Implementations of higher order W-methods are planned. The implementations inherit a helper class that is capable of solving the linear system at a step and to compute the $\mathbf{A}$ matrix. The same helper class can be used as a base for new implementations.

SDIRK Solvers. Implicit solvers require the solution of a nonlinear system of algebraic equations at a step. Therefore they use an instance of Newton-type solver internally. The solver and its components can be set up through the `OptionalParameters` interface. Currently the framework implements the following method:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h[(1-\alpha)\mathbf{f}(t_k, \mathbf{x}_k) + \alpha\mathbf{f}(t_k + h, \mathbf{x}_{k+1})], \tag{5}$$

where the index k denotes the step number, h is the step size, and α is a parameter of the scheme, such that it is the explicit Euler scheme at $\alpha = 0$, the implicit Euler scheme at $\alpha = 1$, and the trapezoidal rule at $\alpha = 0.5$. The implementation handles the case $n_2 > 0$ (variables of second order) such that the nonlinear system solved at a step has size $n_2 + n_1$, and the coordinates $\mathbf{u}$ are excluded. The initial guess can be specified using a predictor, which is another solver specified as a parameter (e.g., explicit Euler). Other implementations are planned.

ODE Solver Output. For convenience, there are a number of components producing some text output as an ODE solver proceeds. The `StatisticsOutput` object reports time and count statistics for various parts of code (right hand side evaluations, LU decompositions and backward substitutions, Jacobian calculations, and more, depending on solver). The `GeneralOutput` object reports current information during the solution (time, step size, error norm at step, event information, error at a Newton iteration, when applicable). The `SolutionOutput` object outputs solution vector at each step or after user-specified time intervals.

ODE Solver Configuration. The object is merely a placeholder for parameters that specify ODE solver, ODE right hand side, initial state, time interval, and output options. It is convenient for supplying the entire description of a numerical experiment from a text file.

3 Example of Application

The `ode_num_int` framework has emerged due to an attempt to design a numerical solver for the problem of continuously variable transmission (CVT) dynamics [16]. The models of CVT components developed so far are implemented in a C++ software package. System state vector contains about 3600 variables that are generalized coordinates, generalized speeds, and a few first order variables. CVT components are modeled as deformable elastic bodies. There are many contacts with friction between the bodies; in particular, torque transmission is possible only due to the friction forces. For many years, the numerical solution of the initial value problem has been done using the explicit RK4 scheme. Although the scheme works well, it requires quite small step size due to stability requirements, such that the step size is about 10^{-7}. As a result, numerical simulations take too long time. Estimations and direct calculation show that the Jacobian of the ODE right hand side has quite large eigenvalues, and they arise due to the friction characteristic. The ODE system appears to be stiff. It is known that for stiff systems implicit methods are preferable. For some of them, Rosenbrock or W-methods work good; for others, including the model of CVT, they don't.

As already said, the framework has been used for applying various solvers to the ODE system of CVT dynamics. Since the ODE system is very complex, the performance of solver should be as good as possible, which has suggested the choice of C++ as the programming language for the framework.

The design of the `ode_num_int` framework has allowed to apply many different numerical integration methods to a real-world application in quite a short period of time. There is no final result so far because it is a work in progress, but still we have found, e.g., that the trapezoidal rule at step size 10^{-5} is as good as RK4 at step size 10^{-8}. Knowing it is important as a motivation for the development of an optimized implementation of the scheme for CVT (in particular, it can only outperform RK4 if the Jacobian is computed faster, which is possible but requires tedious programming).

4 Conclusions and Future Work

The `ode_num_int` framework has been created to help developers find and tune the appropriate solver for certain ODE system. It is very flexible in combining various components together to build an ODE solver, so it can be used as a playground. Much effort has been applied to decouple functionality in different components, so each of them is responsible for one thing. That should make it quite easy for a developer to implement any missing component rather quickly, e.g., to check if some idea will work. When a suitable solver is found, it probably

could be used as is; however, in many cases an optimized implementation will give better performance.

Future plans for framework development include the implementation of more components, like solvers, line search algorithms, and more. On the other hand, it is planned to provide a number of tools working out-of-the-box, like stability region generator, multi-parametric study organizer, and others.

References

1. Blochwitz, T. et al.: The Functional Mockup Interface for Tool independent Exchange of Simulation Models. Proceedings of the 8th International Modelica Conference (2011).
2. Hindmarsh, A.C. et al.: SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers. ACM Trans. Math. Software, pp. 363-396 (2005).
3. Ahnert, K., Mulansky, M.: Odeint — Solving ordinary differential equations in C++. IP Conf. Proc. vol. 1389, pp. 1586–1589 (2011).
4. Schling, B.: The Boost C++ Libraries. XML Press (2011).
5. Hairer, E., Nørsett, S.P., Wanner, G.: Solving Ordinary Differential Equations I (2Nd Revised. Ed.): Nonstiff Problems. Springer-Verlag New York, Inc. (1993).
6. Rosenbrock, H.H.: Some general implicit processes for the numerical solution of differential equations. Comput J 5, pp. 329–330 (1963).
7. Steihaug, T., Wolfbrandt, A.: An attempt to avoid exact Jacobian and nonlinear equations in the numerical solution of stiff differential equations. Math. Comp., 33 pp. 521–534 (1979).
8. Griewank, A., Juedes, D., Utke, J.: Algorithm 755: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++. ACM Trans. Math. Softw., vol. 22, issue 2, pp. 131–167 (1996).
9. Brown, J., Brune, P.: Low-rank quasi-Newton updates for robust Jacobian lagging in Newton-type methods. International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering, pp. 2554–2565 (2013).
10. Gamma, E.: Design Patterns. Addison-Wesley (1994).
11. Lazar, G., Penea, R.: Mastering Qt 5. Packt Publishing, Ltd (2017).
12. Ypma, T.J.: Efficient estimation of sparse Jacobian matrices by differences. Journal of Computational and Applied Mathematics, vol. 18, issue 1, pp. 17–28 (1987).
13. Golub, G.H., Van Loan, Ch.F.: Matrix computations. Johns Hopkins University Press, Baltimore, MD, USA (1996).
14. Broyden, C.: A class of methods for solving nonlinear simultaneous equations. Mathematics of Computation, vol. 19, no. 92, pp. 577–593 (1965).
15. Hart, W.E., Soesianto, F.: On the solution of highly structured nonlinear equations. J. Comput. Appl. Math. 40 (3), pp. 285–296 (1992).
16. Shabrov, N., Ispolov, Yu., Orlov, S.: Simulations of continuously variable transmission dynamics. ZAMM 94 (11), pp. 917–922. WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim (2014).