Object-Oriented Implementation of Software for Solving Ordinary Differential Equations

KJELL GUSTAFSSON

Department of Automatic Control, Lund Institute of Technology, S-22100 Lund, Sweden

ABSTRACT

We describe an object-oriented implementation of numerical integration methods for solving ordinary differential equations. Software components that are common to many different integration methods have been identified and implemented in such a way that they can be reused. This facilitates the design of a uniform user interface and makes the task of implementing a new integration method fairly modest. The sharing of code in this type of implementation also allows for less subjective comparisons of the result from different integration methods. © 1994 John Wiley & Sons, Inc.

1 INTRODUCTION

The traditional way of writing software for solving ordinary differential equations (ODE) is to provide one subroutine to the user. The user calls this subroutine specifying the function that should be integrated, the integration interval, and the accuracy requirements. In some cases the user can also affect internal algorithmic decisions, for example, how to handle the Jacobian, but mostly the integration routine acts as a black-box module. This way of coding has resulted in several very successful implementations, for example, DDASSL [1], RADAU5 [2], LSODE [3], and STRIDE [4].

When solving ODEs there is, unfortunately, no "optimal" integration method. Different problems require different discretization methods in order to be solved accurately and efficiently. As a consequence, software environments for modeling and simulation of ODEs need to implement several different types of integration methods. The way of implementing described above is then no longer preferable. In particular:

1. Many of the internal tasks an integration routine has to perform are independent of the implemented discretization method. In the current style of coding the code that performs these tasks is spread out and sometimes hard to identify. A more modular coding style would allow for reuse, and the task of implementing a new integration method would be easier.
2. If integration methods were to share code for common internal tasks they would be easier to maintain and comparisons of results from different methods would be less subjective.
3. Most of the implementations available today have slightly different user interfaces. In part this is due to them implementing different discretization methods, but often they differ more than necessary. A common interface would facilitate the use of integration

Received April 1993
Revised June 1993

This work was supported in part by the Hans Werthén fund and the Claes Adelsköld fund.

Scientific Programming, Vol. 2, pp. 217–225 (1993)
CCC 1058-9244/94/040217-09

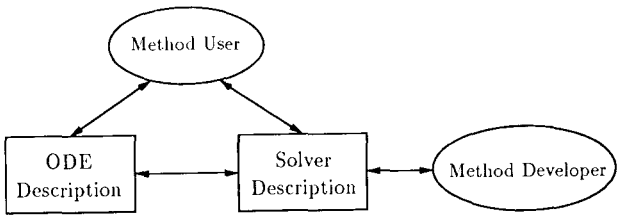

FIGURE 1 ODE-solving software structure. There are two main building blocks: one describing the ODE and one describing the solver. There are also two different types of users: one mainly interested in using existing methods to solve different ODEs and the other trying to implement new integration methods or improving on old ones. It is important that the software structure supports both of these users.

routines as building blocks in other software.

4. ODEs often arise as models of physical systems and/or phenomena. In a modeling and simulation environment [5], one is normally interested in doing more than just integrating the equations, for example, find stationary points, linearize, check parameter sensitivity, etc. The paradigm where the integration method is the central part is then less useful. Rather one would like to have a structure where different aspects of the ODE are described, and the integration routine is just one of many possible operations that may be applied to this data structure.

In the following we will outline an experimental software project that addresses the points made above. It is an object-oriented C++ implementation of a family of different integration methods. For simplicity the current implementation is limited to general explicit Runge-Kutta (ERK) methods and singly diagonally implicit Runge-Kutta (SDIRK) methods. The structure of the software is, however, such that it can fairly straightforwardly be extended to other types of integration methods, for example, multistep methods, extrapolation methods, etc. The software was used to produce the results in [6].

Our view of ODE solving software is outlined in Figure 1. We divide the software into two main building blocks: one describing the solver and one describing the ODE. We also envision two types of users: the first type wants to use the software to solve ODEs, whereas the second type is developing new integration methods. For the first type of user it is important that the solver provides a clear and consistent interface that is virtually independent of the implemented integration method. The user should be able to switch solvers without major changes to the application software. For the second type of user the internal structure of the solver description is of vital importance. It should be written to encourage reuse of basic building blocks and also supply structures that facilitate the implementation of a new method.

After a short review of some aspects of numerical integration we will continue with the description of the software.

2 NUMERICAL INTEGRATION

Consider the initial value problem

$$\dot{y} = f(t, y),\ y(t_0) = y_0,\ t \in [t_0, t_{\text{end}}], \qquad (1)$$

with the exact solution $y(t)$. A time-stepping method approximates (1) with the difference equation

$$y_{n+1} = \bar{y}_n + h_n \bar{f}_n,\ n = 0,1,2, \ldots \qquad (2)$$

Here $\bar{y}_n$ is formed as a combination of solution points and $\bar{f}_n$ is formed as a combination of function values evaluated at the solution points or in their neighborhood. Using (2) we compute $y_0, y_1, y_2, \ldots$ as approximations to $y(t_0), y(t_1), y(t_2), \ldots$. The stepsize h_n between consecutive solution points is defined as $t_{n+1} = t_n + h_n$.

The appropriate stepsize h_n varies along the solution of the differential equation. The choice is a matter of both accuracy and efficiency. Small steps make the solution accurate, but require more computation due to the increase in the number of steps needed. Therefore, the strategy is to choose the stepsize as large as possible within the accuracy requirements, because that gives an acceptable solution with the least amount of computation. It is hard for the user to relate a given stepsize to a specific accuracy and therefore the choice is normally left to the error control algorithm within the integration method.

In an implicit integration method the formulation of $\bar{y}_n$ and/or $\bar{f}_n$ involves the value of y_{n+1}, and a set of nonlinear equations has to be solved to get the new solution point y_{n+1}. The integration method has to implement a numerical scheme to solve for y_{n+1}. This is usually done either using a fixed-point iteration or some modified version of

Newton iteration. The way the iterative equation solver is operated is crucial for the efficiency of the integration method.

3 INTEGRATION METHOD DESCRIPTION

An important step to address the points raised in the introduction is to create a software structure where it is possible to have different integration methods share code. We have achieved this by using the inheritance concept in C++ (Fig. 2). The code that is common to all methods has been collected in the base class `ODESolver`, whereas code that only applies to a specific method or a subfamily of methods has been put in classes further down the inheritance chain.

3.1 The ODESolver Class

The base class `ODESolver` serves two main purposes:

1. It provides the interface to the user who wants to integrate an ODE
2. It defines a skeleton code for the implementation of an integration method

`ODESolver` provides a set of interface routines to the user. The user calls them to specify the conditions of the integration, what ODE to integrate, relative/absolute accuracy requirements, when and what solution data to store, minimum/maximum stepsize, etc. Once this is done the integration routine itself can be called. The integration can either be done step by step (the user regains control after each integration step) or a whole interval at once. After the integration is completed the produced solution can be accessed in different ways, for example, final value, accepted solution points, or interpolated values at specific time instances. Statistical information, for example, number of failed/succeeded steps, number of function evaluations, etc., is also collected during the integration and can be accessed through the interface routines.

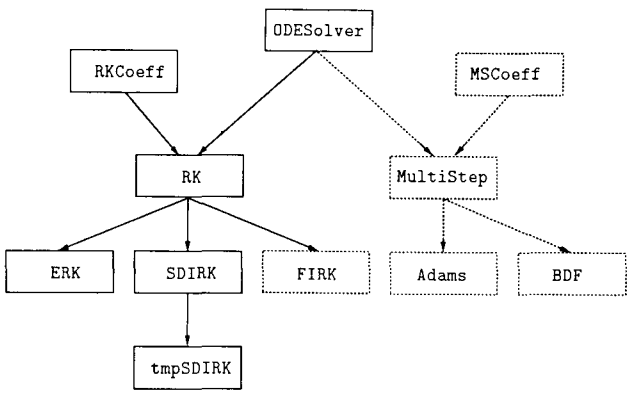

FIGURE 2 Inheritance graph for the classes used when defining an integration routine. The dashed boxes have not yet been implemented but serve to indicate how the structure could be extended.

Having a whole set of interface routines may seem a complicated alternative to the "single" subroutine call provided by standard implementations. It is, however, our belief that it leads to greater flexibility. When instantiating a specific integration method all internal variables concerning the integration are given reasonable default values. A nonsophisticated user may stick to these values whereas a sophisticated user has full freedom to alter any of them through the interface routines, only having to call those related to the properties that he/she wants to change. A typical code segment performing the integration looks like

```
// Create explicit RK method using
// the DOPRI45 coefficient set.
ERK method ("dopri45");
// Install problem in solver.
// 'prob' points to an instance of
// the ODE class.
method.SetODE( prob );
// Set relative error tolerance.
// Same value for all solution
// components.
method.SetTol( 0.001 );
// Integrate to t = 10.
method.IntegrateInterval( 10 );
// Print solution.
cout << method.GetSolution() >> endl;
```

The integration method is of type ERK. All the routines called belong, however, to the general interface defined by `ODESolver`.

The second important part of `ODESolver` is the skeleton code and the support routines it defines. The task of taking one integration step can be separated into several subtasks. The order of these subtasks does not depend on how a specific integration method defines $\bar{y}_n$ and $\bar{f}_n$, and some subtasks may even be identical from method to method. On a coarse level we could claim that taking an integration step involves:

1. Deciding on new stepsize,
2. Calculating new solution point
3. Estimating error
4. Checking if error fulfills requested accuracy
5. If error is acceptable then updating solution.

Some of these tasks are clearly method dependent, for example, calculating the new solution point, whereas others are independent of the implemented method, for example, checking if the error is sufficiently small.

`ODESolver` defines the routine `IntegrateOneStep`. This routine is called every time an integration is to be performed. An integration of a complete interval (as in Example 3.1) is internally implemented as several calls to `IntegrateOneStep`. `IntegrateOneStep` breaks down the task of taking one integration step to several subtasks (to a finer granularity than indicated above) and defines in which order they should be performed. Every subtask is implemented as a subroutine call. The subroutines corresponding to subtasks independent of the method are defined in `ODESolver`, whereas the others are declared as virtual functions. These virtual functions will be defined by deriving classes, for example, `RK` or `ERK` in Figure 2.

The benefit of defining a skeleton code is that once it has been implemented and debugged it need not be redone. The actual implementation of a specific integration method now "only" involves defining specific routines in a predefined structure, and much of the logic intricacies have already been handled.

3.2 RKCoeff and RK Classes

The classes `RKCoeff` and `RK` are used to extend `ODESolver` with data structures and basic routines needed to implement a general Runge-Kutta method.

An s-stage Runge-Kutta method [7, p. 200] defines $\bar{y}_n$ and $\bar{f}_n$ as

$$\bar{y}_n = y_n \tag{3}$$

$$\bar{f}_n = \sum_{j=1}^{s} b_j f(t_n + c_j h_n, Y_j) \tag{4}$$

$$Y_i = y_n + h_n \sum_{j=1}^{s} a_{ij} f(t_n + c_j h_n, Y_j) \tag{5}$$

where Y_i, $i = 1 \ldots s$, are the stage values. The coefficients a_{ij}, b_j, and c_j are chosen such that the Taylor expansion of the numerical solution y_n matches as many terms as possible of the Taylor expansion of the true solution $y(t_n)$. The exponent of the matched term with the highest order is referred to as the method order. The method typically supports two formulas of orders $k - 1$ and k, respectively. They are represented by the two coefficient sets b_j and $\hat{b}_j$. One coefficient set is used to advance the integration ($\bar{f}_n$ above) and the other is used for the error estimate $\hat{e}$

$$\hat{e}_{n+1} = h_n \sum_{j=1}^{m} (b_j - \hat{b}_j) f(t_n + c_j h_n, Y_j). \tag{6}$$

In addition, many Runge-Kutta methods define an interpolant

$$q(t_n + v h_n) = y_n + h_n \sum_{j=1}^{s} b_j(v) f(Y_j), \tag{7}$$

to provide solution values between y_n and y_{n+1}, i.e., $0 \le v \le 1$. Each $b_j(v)$ is a polynomial in v.

A Runge-Kutta method is defined by the coefficients a_{ij}, b_j, and c_j. The class `RKCoeff` implements a data structure that stores these coefficients as well as the set of polynomials $b_j(v)$. The coefficients are read from file when the class is instantiated. Which file to use is passed as an argument to the class constructor. This provides a general implementation capable of representing any type of Runge-Kutta method.

Depending on the structure of the coefficient matrix a_{ij} Runge-Kutta methods can be divided into different classes, for example, a method that has $a_{ij} = 0$, $j \ge i$ is called explicit because the stage values Y_i can be calculated explicitly, one after another. In an implicit method, Equation 5 defines a set of nonlinear equations that have to be solved in order to obtain the stage values.

The class `RK` implements data structures to store Y_i. The actual calculation of Y_i depends a lot on the structure of a_{ij} and is left to be defined in deriving classes specializing the behavior of `RK` to a specific family of Runge-Kutta methods (Fig. 2). Once Y_i has been calculated it is straightforward to obtain the new solution point (compare Equations 2, 3, and 4), and estimate the error vector (Equation 6). This type of operation is defined in `RK`.

3.3 ERK and SDIRK Classes

With the support from base classes as `ODESolver`, `RKCoeff`, and `RK`, it is fairly straightforward to define a specific type of Runge-Kutta

method. This is done in deriving classes such as ERK and SDIRK. One of the routines that is called from the skeleton code in IntegrateOneStep is the virtual function CalculateNewSolutionPoint. In the case of a Runge-Kutta method this task amounts to obtaining the stage values Y_i and combining them to form y_{n+1}. This routine is defined in ERK and SDIRK. It uses several of the support routines in RK to achieve its goal.

In the case of an explicit method (ERK) the calculation of Y_i is done by a few function evaluations, whereas in an implicit method, for example, SDIRK, a set of nonlinear functions have to be solved. A numerical equation solver is called to do this. To be efficient the solver should exploit the structure of the coefficient set a_{ij} as well as the ODE, and consequently it is not sufficient to provide only one type of equation solver. We will return to this matter in Section 5.

Another routine called from the IntegrateOneStep skeleton code and implemented in ERK and SDIRK is ErrorAndConvergenceControl. After each integration step this routine decides what stepsize to use in the next step and, in the case of an implicit method, how to update the iteration matrix in the equation solver. These decisions depend on whether the current integration step fulfilled the accuracy requirement or not and how well the equation solver iterations converged. The strategy underlying these decisions is vital for the efficiency and accuracy of the integration method, and ErrorAndConvergenceControl should be tailored to each specific type of integration method [6]. Collecting the complete control algorithm in one place helps bring global considerations into the design of the control strategy.

3.4 Extensions

The amount of code in ERK and SDIRK is fairly modest. It amounts to providing implementations of specific routines in a predefined structure. This makes the task of implementing a new type of Runge-Kutta method, for example, fully implicit Runge-Kutta (FIRK in Fig. 2), less formidable. Implementing a different type of integration method, for example, multistep methods, is a larger task. As Figure 2 indicates this involves the design of data structures and support routines corresponding to the ones in RKCoeff and RK. The administrative routines and the skeleton code in ODESolver, however, make the effort much less tedious than doing the implementation from scratch.

Finally, a structure such as the one we have described makes it very easy to experiment with modifications to existing methods. Suppose we want to investigate a change in strategy for determining stepsize in an SDIRK method. This can be achieved by deriving a new class SDIRK, for example, tmpSDIRK in Figure 2. This new class will behave exactly as an SDIRK, but we are free to supply new implementations for any of the original routines. In our case we would just provide a new ErrorAndConvergenceControl with the part calculating the new stepsize rewritten.

4 ODE DESCRIPTION

The idea behind the class hierarchy describing an ODE is similar to the one underlying the description of the solver. We have defined a base class ODE providing basic data structures and operations. A specific ODE is then defined by deriving from the ODE class and providing implementations of the virtual functions defined in the base class. The structure is fairly simple because, so far, we only support ODEs on the explicit form (1). We have chosen this form for simplicity. The software can, however, quite easily be extended to cover also other formulations, for example, ODEs on implicit form $f(t, y, \dot{y}) = 0$ or differential-algebraic equations, etc.

An important difference between our structure and traditional implementations is that the solver does not keep its own copies of the ODE state, i.e., t and y, but rather works directly on the data stored in ODE. The ODE class makes this possible by providing access routines to set and read the state values. Storing the ODE state in ODE is a natural consequence of regarding the ODE as the main object, and the solver as one of many operations one wants to apply to it.

In addition to the routines related to the state of the ODE the base class ODE also defines interface routines to access the derivative $f(t, y)$ and the Jacobian $\partial f/\partial y$. These functions are of course virtual and their implementation will be provided by the deriving class that implements a specific ODE. Having the definitions of Derivative and Jacobian in ODE provides for a well-defined interface between the ODE description and the solver irrespective of what specific ODE the user eventually ends up defining and integrating.

Many integration method implementations will allow the Jacobian to be defined either through numerical approximation or analytically in the form of a supplied subroutine. Which option to

choose is normally decided with a parameter in the subroutine call to the integration routine. Our implementation differs on this point. We regard analytic or numerical Jacobian as a problem property and hence this distinction is included in the ODE description. The solver will always call the routine `Jacobian`. How this entity is calculated is then decided internally in the ODE representation. The base class `ODE` provides internal routines that will construct a numerical approximation of the Jacobian, and it is up to the deriving class to either use this code or provide its own method (numerical or analytic) to calculate the Jacobian.

5 THE EQUATION SOLVER BUILDING BLOCKS

At every integration step an implicit integration method needs to solve a set of nonlinear equations. Most numerical equation solvers lead to an iteration scheme on the form

$$z_{m+1} = z_m + M^{-1}R(z_m) \tag{8}$$

where z is the vector of variables solved for, R a residual function, and M the iteration matrix. The structure of Equation 8 depends on both the ODE and the type of integration method. To get an efficient implementation it is essential to exploit this structure.

In the case of Runge-Kutta methods z represents the stage values. Depending on the structure of the coefficient matrix a_{ij}, z may involve all, a subset, or only one of the stage values Y_i. The iteration matrix M depends on the type of iterations performed. In the case of fixed point iteration it equals the identity matrix, whereas for Newton iteration it involves both the Jacobian $\partial f/\partial y$ of the ODE as well as the stepsize h and coefficients from the integration method.

Due to the intricate structural effect the type of ODE and integration method have on the iteration (Equation 8), it is not possible to design one equation solver that works efficiently for any combination of ODE and method. Rather, the equation solver has to be put together from several building blocks, among which some are intrinsic whereas others depend on the ODE and/or the method. We have identified and implemented the building blocks indicated in Figure 3. These blocks make the work of adapting the equation solver to a new integration method or a new formulation of the ODE fairly modest. Some blocks can be reused directly whereas others require some adjustments. These adjustments are easily implemented in the form of a deriving class. This is indicated by the dashed blocks in Figure 3.

To implement a complete equation solver the integration method will contain one instance of `IterationController`, `StageUpdate`, and `IterationMatrix`. The `IterationController` supervises the iteration process, `StageUpdate` handles one iteration of (Equation 8), and `IterationMatrix` captures the structure of M. Both `StageUpdate` and `IterationMatrix` have to be adapted to the specific type if integration method used as well as to the way the ODE is formulated. This is done by deriving from the respective base classes. The changes that need to be done are normally fairly modest. Information about the current iteration is kept in an object of type `IterationStatus`.

5.1 IterationController and IterationStatus Classes

The iteration (Equation 8) can be rewritten as

$$M\delta_m = \rho_m. \tag{9}$$

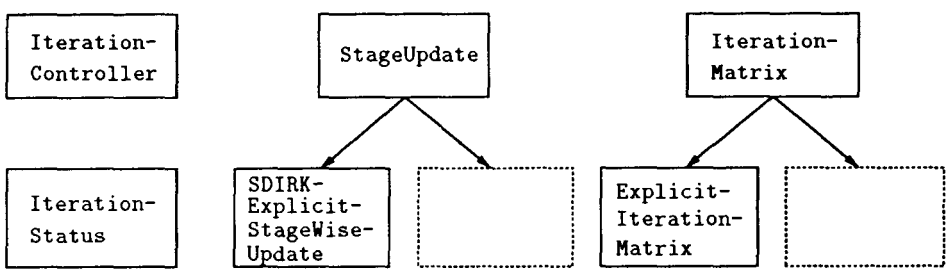

FIGURE 3 The equation solver building blocks. The two dashed blocks indicate where new classes have to be written when implementing a new type of integration method or allowing a different ODE formulation.

Given a residual ρ_m we solve for the displacement δ_m, and update $z_{m+1} = z_m + \delta_m$. The iteration should continue until the solution z is sufficiently accurate, typically measured by the size of $\|\delta_m\|$ or $\|\rho_m\|$. During the iteration the rate of convergence, i.e., $\|\delta_m\|/\|\delta_{m-1}\|$, is monitored and at sign of divergence the iteration is terminated. For robustness, there is also normally a limit on the maximum number of iterations allowed.

The described iteration control has been implemented in the `IterationController`. This class contains interface routines that can be used to set up the terms of the iteration, e.g. max/min number of iterations, stopping criteria in terms of iteration error, and rate of convergence, etc. Once this has been defined the `IterationController` is asked to start the iteration. The `IterationController` expects to have available a routine `Update`, which it will call at every iteration. The `Update` routine performs one iteration in Equation 8 and returns the current value of $\|\rho_m\|$ and $\|\delta_m\|$. Based on these values the `IterationController` decides whether to continue or to terminate the iteration. The implementation of `IterationController` is completely independent of how ρ_m is formed, how Equation 9 is solved, and how z is updated.

Once the iteration has been terminated the `IterationController` returns an object of type `IterationStatus`. This object contains information about the iteration, for example, number of iterations, rate of convergence, final iteration error, etc. This information is used by the `ErrorAndConvergenceControl` routine described above to decide on how to update the iteration matrix M for the next iteration.

5.2 The StageUpdate Class Hierarchy

In the case of Runge-Kutta methods the `Update` routine called from the `IterationController` is implemented in the `StageUpdate` class hierarchy. This routine performs one iteration by calculating the residual ρ, solving for δ and updating z.

The base class `StageUpdate` defines the interface to `Update`, but defers its implementation to deriving classes. The reason is that what goes into the variable z depends on the coefficient matrix a_{ij} of the Runge-Kutta method. In addition, the calculation of ρ depends on the way the ODE is formulated, for example, explicit or implicit.

The class `SDIRKExplicitStageWiseUpdate` in Figure 3 corresponds to a situation where the `Update` routine has been specialized to an SDIRK solving an ODE on explicit form and iterating for the stage values Y_i one after another. Implementing a new type of Runge-Kutta method normally requires writing a new derived class in the `StageUpdate` hierarchy.

5.3 IterationMatrix Class Hierarchy

The `Update` routine in `StageUpdate` uses an object of type `IterationMatrix` when solving for δ in Equation 9. This object encapsulates the description of the iteration matrix M. M involves both the stepsize h and the Jacobian of the ODE, and should be reevaluated every time any of these changes. For efficiency reasons this is not acceptable. The success of a particular implementation of an implicit integration method depends strongly on finding a good strategy for updating M.

The base class `Iterationmatrix` provides the interface routines to update M. These routines are used by the control algorithm implemented in `ErrorAndConvergenceControl`. Calling `NewJacobian` will evaluate a new Jacobian using the access routines provided by `ODE`, while `Factorize` forms a LU decomposition of M based on the current stepsize and the latest available Jacobian.

The structure of M depends on the way the ODE is formulated, and the implementation of some of the operations we are interested in cannot be defined in the base class. Again we use derivation to specialize to different situations. The class `ExplicitIterationMatrix` in Figure 3 corresponds to an ODE on explicit form. New types of integration methods and/or a new way of formulating the ODE will require a new derived class in the `IterationMatrix` hierarchy.

6 OTHER GENERAL BUILDING BLOCKS

In our implementation we have used several other building blocks than the ones mentioned in Sections 3–5. Some of them are worth mentioning:

6.1 Matrix and Vector Operations

Many of the operations in an integration method involve the manipulation of matrices and vectors. In our implementation we have used the package `newmat` [Davis, personal communication, 1992]. This package provides routines that make it possible to write matrix expressions in much the same way as expressions involving standard floating point variables.

For many ODEs the Jacobian has special structure, e.g., banded. In standard implementations [1–4], this information can be included in the pa-

rameter list when calling the integrator. In a typical implementation the handling of a Jacobian with structure adds complexity to the program. Every time the Jacobian is accessed there will be tests in the code making sure that the right representation and routines are used. Using C++ and a package such as `newmat` we get this functionality almost for free. `newmat` distinguishes between different types of matrices. If the `Jacobian` routine in `ODE` returns the result as a banded matrix this property will propagate. Whenever the matrix is accessed through the `newmat` primitives its format will be checked and the appropriate routines automatically chosen. Hence, in the case of banded Jacobian algorithms for banded matrices will be used automatically both when forming M and when calculating its LU decomposition. This is achieved without changes or additions to the equation solver code.

6.2 Norms

Most integration methods use a mixed absolute-relative "norm" to measure the size of the error estimate (Equation 6). For consistency the same norm should be used when measuring the size of the iteration errors in the equation solver, i.e., δ and ρ.

To handle the error measure consistently we have implemented a class `RelAbsNorm`. Given a vector x this class will provide methods and data structures to calculate its weighted norm. The weight factor w_i used for component i in x is formed as

$$w_i = \frac{1}{tol_i(\eta_i + |y_i|)} \tag{10}$$

The relative part of the "norm" comes from relating x_i to the magnitude of the same component in the solution vector y. The factor η_i acts as a scale factor telling when y_i is regarded as small. Effectively it results in an absolute error measure when the magnitude of y_i is less than η_i. The class can handle $\eta = 0$ (pure relative norm) and $\eta =$ "∞" (pure absolute norm).

The class `RelAbsNorm` implements all the interface routines needed to set the values of *tol* and η, either componentwise or the same value for all. Once this has been done the norm object can be passed around and anyone wanting to evaluate the norm of a vector x weighted according to Equation 10 just calls the `Evaluate` function in `RelAbsNorm` passing x as a parameter. The class handles 1-, 2-, and max norms.

6.3 Data Storage

One of the services the base class `ODESolver` provides for the user is storage of the solution. Instead of integrating step by step the solver can be asked to integrate a whole interval and store the produced solution internally. The user can decide what solution components to store, when to store them, etc. After the integration the stored solution can be accessed through interface routines defined by `ODESolver`.

A support class `DataStore` was written to facilitate this type of data storage. Instances of this class are used whenever the programmer needs to store some data. The implementation of `DataStore` is done such that memory is allocated in an efficient way. In addition, the data are stored in a manner allowing fast access.

7 CONCLUSIONS

We have described an object-oriented implementation of a family of different numerical integration methods. This implementation differs from standard implementations in several important ways. For a user who wants to solve an ODE:

1. It provides a uniform interface independent of the implemented integration method
2. It works on a simple data structure describing the ODE, which the user easily can incorporate in a larger software framework for analyzing ODEs
3. It makes different integration methods use the same implementation of basic building blocks, e.g., norms, iteration criterias, etc., resulting in less subjective comparisons of results produced with different methods.

For the user who develops and implements new integration methods:

1. It provides a software structure and basic building blocks on which the implementation of new methods can be based
2. It facilitates debugging because many of the basic routines and structures have already been used and tested in other methods
3. It collects the code that describes the error and convergence control in an integration method in one place, making it possible to introduce global considerations in the design of the control strategy
4. It makes the equation solver of an implicit method easier to write by providing basic

building blocks and automatic support for Jacobians with structure.

Turning a new integration method into a working efficient implementation is often a long and difficult process. We strongly believe that it can be sped up and simplified using a software structure such as the one described here.

ACKNOWLEDGMENTS

This article was written during a research stay at Stanford University. The author would like to thank Gene Golub for providing the opportunity to visit Stanford.

REFERENCES

[1] K. E. Brenan, S. L. Campbell, and L. R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations.* North-Holland: Elsevier, 1989.

[2] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II—Stiff and Differential-Algebraic Problems* (Springer Series in Computational Mathematics, vol. 14). Springer-Verlag, 1991.

[3] A. C. Hindmarsh, *ODEPACK, A Systematized Collection of ODE Solvers* (Scientific Computing). Amsterdam: IMACS/North-Holland Publishing, 1983, pp. 55–64.

[4] K. Burrage, J. C. Butcher, and F. H. Chipman, "An implementation of singly-implicit Runge-Kutta methods," BIT, vol. 20, pp. 326–340, 1980.

[5] S. Erik Mattsson, M. Andersson, and K. J. Åström, *Object-Oriented Modelling and Simulation.* New York: Marcel Dekker, Inc., 1992.

[6] K. Gustafsson, "Control of error and convergence in ODE solvers" Ph.D. thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1992.

[7] E. Hairer, S. Paul Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I—Nonstiff Problems* (Springer Series in Computational Mathematics, vol. 8).