CWI Quarterly Volume 11 (1) 1998, pp. 1 – 5

Survey of Relevant Concepts

Jacques J.B. de Swart
CWI, P.O. Box 94079,
1090 GB Amsterdam, The Netherlands
and
Paragon Decision Technology B.V.
P.O.Box 3277
2001 DG Haarlem, The Netherlands

Implicit Differential Equations (IDEs) can model many processes in industry. For example, to test the design of a computer chip, its behavior is modeled by a set of IDEs. If the solution of these equations satisfies the requirements, then the chip is manufactured; if not, then the design is to be adjusted. Thus the production process becomes much cheaper than in the case where all designs—including the wrong ones—are first manufactured and tested afterwards. Other examples that can be modeled by differential equations are the behavior of a train on a rail track, the steering of robots and chemical reactions.

The time required to solve IDEs can be reduced by designing algorithms that can be implemented on modern computer architectures with more than one processor, so-called *parallel* computers. To increase the speed of the fastest state-of-the-art processor becomes more difficult and costly, whereas the price of simpler, but still reasonably fast processors has dropped considerably over the years. This development inspired many computer companies to start the production of parallel computers.

The differential equations arising from the modeling process may have different forms. The most simple formulation of interest here is that of the Initial Value Problem (IVP) for Ordinary Differential Equations (ODEs), which reads: Given a function $f : \mathbb{R}^d \to \mathbb{R}^d$, find the function $y : \mathbb{R} \to \mathbb{R}^d$ that fulfills

$$y'(t) = f(y(t)), \qquad y(t_0) = y_0, \qquad t_0 \le t \le t_{\rm end}. \tag{1}$$

Of course in practice one often encounters more complex classes of differential equations, but for simplicity of notation, we restrict ourselves mostly to the class defined by (1).

Almost every method to solve (1) numerically is a step-by-step method; one divides the interval $[t_0, t_{\rm end}]$ in subintervals $[t_0, t_1]$, $[t_1, t_2]$, $\ldots$, $[t_{N-1}, t_N]$,

where $t_N = t_{\text{end}}$, and computes approximations y_1, y_2, ... , y_N to the solution at the end of each subinterval. The accuracy of the method will depend on the length of the subintervals, which we call the *stepsize* and denote by h, which may depend on the specific subinterval. If $y_N - y(t_{\text{end}}) = \mathcal{O}(h^p)$, then the *order* of the method is p.

The computation of y_n in a conventional step-by-step method depends on approximations in time points prior to t_n; to proceed in time, information from the past has to be available. This means that the numerical solution process is to a large extent sequential by its nature and offers little scope for parallelization. Nevertheless, several attempts have been made to exploit parallel computer architectures, which has been classified by GEAR [?] as follows:

1. parallelism across the problem,
2. parallelism across time,
3. parallelism across the method.

The first class consists of rather obvious ways to distribute the various components of the system of ODEs amongst the available processors and will not be discussed here. The strategy for methods in class 2 is to compute approximations to the solution at different time points concurrently. Solution techniques belonging to the third category employ parallelism inherently available within a method. For example, the method may be such that the computation of y_n requires several evaluations of f that can be done in parallel. Notice that this form of parallelism may even be effective for scalar problems (i.e. $d = 1$ in (1)), whereas approach 1 requires high d-values. Due to the limited size of this special issue of CWI Quarterly, we will confine ourselves to methods belonging to class 3. For methods based on parallelism across time, we refer to [?, ?, ?, ?].

Since most approaches of type 3 are based on some variant of a Runge–Kutta (RK) method, we briefly resume some terminology of RK methods.

A Runge–Kutta method has the following form:

$$Y_n = \mathbb{1} \otimes y_{n-1} + h(A \otimes I)F(Y_n)\,, \tag{2}$$

$$y_n = y_{n-1} + h(b^{\mathrm{T}} \otimes I)F(Y_n)\,. \tag{3}$$

Here, Y_n is the so-called *stage vector*, which contains s approximations $Y_{n,i}$, $i = 1, 2, \ldots, s$, to the solution in the time points $t_{n-1} + c_i h$, i.e.,

$$Y_n = (Y_{n,1}^{\mathrm{T}}, Y_{n,2}^{\mathrm{T}}, \ldots, Y_{n,s}^{\mathrm{T}})^{\mathrm{T}},$$

where $Y_{n,i} \approx y(t_{n-1} + c_i h)$. The scalars c_i determine where the solution is approximated and are called the *abscissae*. The length of the subinterval $[t_{n-1}, t_n]$ is the *stepsize* and is denoted by h. The symbol $\otimes$ denotes the direct product, which is defined by

$$\begin{bmatrix} v_{11} & \cdots & v_{1l} \\ \vdots & & \vdots \\ v_{k1} & \cdots & v_{kl} \end{bmatrix} \otimes W = \begin{bmatrix} v_{11}W & \cdots & v_{1l}W \\ \vdots & & \vdots \\ v_{k1}W & \cdots & v_{kl}W \end{bmatrix},$$

where $V = (v_{ij})$ and W are matrices of arbitrary dimensions. Furthermore, $\mathbb{1}$ stands for the s-dimensional vector $(1, \ldots, 1)^{\mathrm{T}}$, I is the identity matrix of the problem with dimension d, and $F(Y_n)$ means the componentwise f-evaluation, i.e.

$$F(Y_n) = \begin{pmatrix} f(Y_{n,1}) \\ \vdots \\ f(Y_{n,s}) \end{pmatrix},$$

so that $F(Y_n)$ is of dimension sd. The $s \times s$ matrix A and the s-dimensional vector b contain the parameters of the Runge–Kutta method. If the matrix A is full, then we call (2)–(3) an Implicit Runge–Kutta method (IRK). In most cases the function f in (1) is non-linear, which implies that for an IRK the sd-dimensional system (2) is non-linear. Once Y_n is solved from this system, we can compute the approximation to the time point t_n by formula (3).

To select the type of RK method and the strategy to solve Y_n from the non-linear system, the notion of stiffness is important. If the time scales of the various solution components vary greatly and if the rapidly changing components are physically unrelevant, then we call a problem *stiff*. For example, if both high and low frequency signals are present in an electrical circuit but the highly frequent signals are small in magnitude, then the modeling of such a circuit gives rise to a stiff system of differential equations. Such a problem imposes severe stability demands on the numerical method.

For non-stiff problems we may use so-called *fixed-point iteration* to find Y_n. Given some initial guess $Y_n^{(0)}$, we define a sequence of iterates by

$$Y_n^{(j)} = \mathbb{1} \otimes y_{n-1} + h(A \otimes I) F(Y_n^{(j-1)}), \tag{4}$$

and accept $Y_n^{(m)}$ as approximation for Y_n if it fulfills (2) accurately enough. The method for determining $Y_n^{(0)}$ is called the *predictor* and (4) the formula for the *corrector*. The paper by Sommeijer and part of the paper by Van der Houwen and Sommeijer develop predictor–corrector methods, in which several stages can be computed in parallel.

Since many applications yield stiff systems of differential equations, most methods presented in this volume deal with numerical solution techniques capable of handling stiffness. The fixed-point iteration may cause severe stepsize restrictions to converge for stiff problems and can not be used anymore. A well-known alternative is the modified Newton process, which takes the form

$$(I - A \otimes hJ)(Y_n^{(j)} - Y_n^{(j-1)}) = -R(Y_n^{(j-1)}), \tag{5}$$

where, for any $X \in \mathbb{R}^{sd}$, $R(X) = X - \mathbb{1} \otimes y_{n-1} - h(A \otimes I)F(X)$, and J stands for an approximation to the Jacobian of f evaluated in $y(t_{n-1})$, i.e.,

$$J \approx \frac{\partial f}{\partial y}(y(t_{n-1})).$$

Again $Y_n^{(0)}$ is produced by a predictor formula and (5) is applied as many times as needed to make $Y_n^{(j)}$ sufficiently close to the true solution of (2).

The dimension of the linear system (5) is sd, which makes IRKs relatively expensive to implement. For this reason they are not frequently used in practice. Most industrial codes for stiff problems are based an Backward Differentiation Formulas (BDFs), which require the solution of linear systems only of dimension d in every Newton iteration (see, e.g., [?]). On the other hand, BDFs do not allow for parallelism across the method and from the famous Dahlquist order barrier it follows that having high order and being unconditionally stable are two properties that cannot be combined by BDFs. Another disadvantage is that the BDF of order k is a k-step method; it bases the approximation y_n on information collected in the previous k subintervals. Evidently, this complicates the change of stepsizes. Moreover, if for some reason the method has to be restarted frequently, e.g. due to discontinuities in the function f, then in every restart one has to apply the one-step BDF of first order and 'to build up' the order in the subsequent steps.

For IRKs the situation is opposite. From (2)–(3) it is clear that these are one-step methods and, although expensive to implement on a sequential computer (a computer with only one processor), IRKs can benefit from parallelism across the method and are unconditionally stable. Although most parallel methods in the forthcoming papers are not based on pure IRKs but on some variant, the main idea behind these is to use some sort of Newton process, such that it only requires the solution of linear systems of dimension d and that the additional processors can solve several of these systems in parallel.

Summarizing, the parallel IRK based methods can still cherish the low effective costs of a BDF method, but overcome the shortcomings of BDF. As we will see, the way in which this goal is achieved in the paper by Van der Houwen and Sommeijer and those by Chartier and Voss (on DIMSIMs and MIRKs, respectively) differs considerably.

The final goal of developing numerical methods is to be able to solve real-life problems. Therefore it is not sufficient to construct numerical methods, but one also needs to develop a piece of software that incorporates these methods. It is a long road from method to software. First of all, one has to decide for which problem class the code should be written. Many applications require a much broader formulation than (1). For example, to model the behavior of a train on a rail track, one has to add algebraic (meaning not containing differentials) side conditions, which state that the train and rail track can not intersect. The resulting system is called a set of Differential–Algebraic Equations (DAEs). An even broader class is that of Implicit Differential Equations (IDEs), which are of the form

$$\begin{array}{lll} g(t,y,y') = 0\,, & g : \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d\,, & y : \mathbb{R} \to \mathbb{R}^d\,, \\ t_0 \le t \le t_{\rm end}\,, & y(t_0) = y_0\,, & y'(t_0) = y_0'\,, \end{array} \tag{6}$$

where some components of g may contain differentials and some not.

A complication of IDEs, which does not apply to ODEs, is that some components of the IDE solution may be more sensitive to perturbations. These components are said to be of *higher index*. In the paper by De Swart & Lioen in this issue, we will see an instance of such a situation.

Many other questions have to be answered when implementing these techniques, e.g., how to form a prediction for the Newton process, when to evaluate the Jacobian, how many Newton iterations should be done, is the error conducted in one time step small enough, how to vary the stepsize? Søderlind contributed a paper that explains how one can answer these questions using control theory.

To get insight in the performance of a solver compared to other solvers, when applied to different problems, it is important to have a well-defined test protocol and representative test problems. Proper testing of software is a whole field by itself and will be discussed in the paper by De Swart and Lioen.