8.3 Stiff computation: where to go? (Cellier)

8.3.1 *Introduction*

It is the aim of this discussion to list some of the more severe shortcomings of available stiff computation codes as they have been collected through discussions with the end users of such software. These complaints, however, are formulated in terms that are understandable to the producers of integration algorithms. Some suggestions are given on how these deficiencies may be overcome in the future, and where open research fields can still be found.

8.3.2 *Discontinuity handling: where to go?*

The initial-value problem (1.1) and (1.2) may contain an $f(x, y)$ that is discontinuous in x. Typical examples of such discontinuities are (i) in mechanical engineering: friction phenomena, (ii) in electrical engineering: diodes, thyristors, any combined analog and digital circuitry, (iii) in chemical engineering: charging and discharging of batch reactors.

We can distinguish between two different types of discontinuities:

(i) Discontinuities of which we know the time when they are expected to happen. As a typical example of this type of discontinuity, consider when output voltage switches from negative to positive, or vice versa, a discontinuity takes place. The time instances when these discontinuities are expected to take place are precisely known beforehand. These discontinuities are in simulation usually referred to as "time events." Time events are scheduled events in the sense that the time instances when the discontinuity is to take place may be collected into a "calendar of events."

(ii) Discontinuities of which we do not beforehand know the time when they are expected to happen. Instead, we know the condition under which the discontinuity takes place, e.g. a state variable x_i crossing a prescribed level into positive direction. As a typical example of this type of discontinuity, we mention the output voltage of a rectifier circuit. Here we do not know when the output voltage is going to have a discontinuity (in its first time derivative). We just know that each time the input voltage crosses though zero in either direction, the output voltage has a break. These discontinuities are in simulation usually referred to as "state events." For state events, no scheduling is possible. Instead, we need a mechanism to describe the condition under which the event is going to happen. This is usually referred to as "state condition."

It makes sense to distinguish between these two types of discontinuities for two reasons:

(i) The user of the code requires two different mechanisms to describe the discontinuities to the program.

(ii) The algorithm requires two different mechanisms to handle them. Time events may be handled simply by inspection of the calendar of events for the next scheduled event time. When this moment approaches, the step size must be reduced to hit the event time precisely, or the algorithm must interpolate back to the just-passed event time, depending on the algorithm in use. State events, on the other hand, must be handled by either iteration or interpolation (depending on the algorithm).

Unfortunately, until very recently none of the existing codes for numerical integration provided for appropriate discontinuity handling. When I mentioned this problem to a prominent numerical mathematician during the Urbana ACM/SIGNUM Conference on Numerical Ordinary Differential Equations in 1979, he answered that this was a problem which should be left to the end user to be optimally adapted to his personal needs and should not be of concern to the numerical mathematician. I do not agree for several reasons:

(i) The numerical mathematician usually thinks that given a particular algorithm everything is said and clear. However, this is not the case for reasons of communication problems. The end user in general is not able to understand an integration algorithm well enough to be able to code it into a properly executing program by himself. This holds equally true for discontinuity handling. Concerning the integration algorithm, the attitude originally was also that coding of the program be basically the user's responsibility, while it has meanwhile been realized that precut standardized codes are superior in many respects, even though there are still some people around to call this the "vacuum cleaner" approach. I suggest that the handling of discontinuities is a more recent problem, and that in the long run standardized codes shall replace home-tailored software here also.

(ii) If you look into the work done by Hull and Enright at Toronto, it should become clear that one has no chance generally to compare integration algorithms. What one can do, however, is to compare the aptitude of different integration codes to solve a particular application problem. Hopefully, one will be lucky enough to be able to extend the experiences gained from entire sets of problems to a classification of applications. It should then become clearer which algorithm is best for any particular practical application without being forced to try them all. Results indicate that the step from the integration *algorithm*, once given, to the integration *code* is very large indeed. In fact, more computation time is used by the "dirty" overhead surroundings than by the integration algorithm itself. Again, this applies to the problem of discontinuity handling as well.

(iii) It is not usually economical to develop larger pieces of software for each applica- separately. On the contrary, we should concentrate on determining what large classes of problems have in common, extract this information, and provide for a program which handles all that in a standardized manner. This is time-efficient, cost-efficient, and much safer, as the amount of software left to be

coded by the end user is minimized while obviously much more care and expertise can flow into the design and production of the standardized part of the software. It is here, where simulation software comes into the game. Definitely, discontinuity handling belongs to the part of the code which should be standardized.

To conclude these considerations, let me propose some very concrete steps toward (in my view) a satisfactory solution. We need a standardized user interface for ODE solvers (Hindmarsh 1978). It is evident that I cannot know in advance which integration code will be optimally suited for my particular application under all circumstances. Thus, it should be such that I can code my application software entirely independently of the ODE solver. Ideally seen, I would like to be able to replace any ODE solver by any other ODE solver simply by replacing one subroutine name by another subroutine name without being forced to change a bit of my application program. In fact, I would want to maintain a library of integration codes to have a "remedy against all diseases." An additional advantage of this approach is that I may easily update my ODE solver library when new releases become available as there are no side effects to be expected from simply replacing the old subroutine by its modified version. However, this also means that I should not be forced to change a single line within the ODE solver for any particular application. It is now very easily shown that, when I try to graft the discontinuity handling upon the integration code (which I have done in my simulation software GASP-V—Cellier (1978, 1979)—for precisely the aforementioned reasons), I obtain inefficient code. For this reason, although I fully support Hindmarsh's idea concerning a standardized interface for ODE solvers, I do not agree to his standard proposal. I feel strongly about the need for enhancing the standard proposal by adding to it an appropriate description of conditional termination criteria (discontinuity functions). In addition, the standard also should be expanded in another sense. If I want to maintain a library of ODE solvers, I shall most certainly need a linear system solver (LSS) in the majority of them. If this (lower end) interface is not standardized, too, I have to maintain any number of basically identical LSS in my ODE library. For this reason, it would make sense to standardize this lower end of the ODE code, also. Different LSS may then reside in a possibly separate library (e.g. general LSS and sparse LSS) out of which I may select whatever seems appropriate to go with whatever ODE solver I want to use. It ought to be mentioned that Hindmarsh has introduced a root solver into one of his programs, LSODAR, and uses sparse matrix techniques for linear systems solving in another program (LSODES) (see §4.3.4). However, there currently exists no version which combines root finding with sparse linear system solving. I would strongly suggest a version which combines all these features at the expense of a somewhat reduced efficiency.

A basic problem here is that the producers of the integration algorithms are not necessarily equally well trained in computer science. However, the above-mentioned considerations are basically those of a computer scientist, and it is sometimes hard to convince algorithm developers of the fruitfulness of such considerations.

When I asked Wanner after his presentation of a newly developed A-contractive

algorithm (that is, in terms of Dahlquist, a G-stable algorithm, if I understood Wanner correctly), during the Rutishauser Symposium in Zurich (Wanner 1980c), whether he had already produced any executable code for his algorithm, he looked at me rather puzzled and answered that his work was purely theoretical, and that the job of coding would definitely have to be someone else's task. (For reasons of fairness it should be mentioned that Wanner has personally produced several integration codes for some other algorithms.)

I can assure you that the end user is certainly *not* willing to try his hands at such an adventure because he is (1) not able to do it, and (2) not able to judge beforehand whether this new algorithm will get him anywhere or not. It has already too often happened (also to myself) that a good-looking new algorithm turned out to be a complete failure after it had been coded into a program. Once I tried to code a DIRK algorithm: the effort spent on coding this algorithm was tremendous, it never worked satisfactorily, and it never became clear afterwards whether the problem was really with the algorithm or just with the code!

Even Henrici (who is known for his sympathy for practical solutions to real-world problems as opposed to ivory-tower solutions to green-grass problems), when I asked him a couple of years ago whether he would agree to join the consenting committee for my PhD thesis which I was to write on "combined continuous/discrete system simulation" (Cellier 1979), answered that he did not know anything about simulation but that he was certainly willing to learn something about it. I then told him that this was beside the point in that I was sure that he knew a lot about simulation. The problem was simply that he did not know that he knew something about it. This indeed is a severe problem in that it indicates that the average mathematician, even being a specialist in numerical integration, does not scan the literature for articles on simulation, even though these articles could be as important to his work as any contribution on numerical integration. I was enchanted to learn during the Park City meeting that several of the participating mathematicians had very much adopted the engineering viewpoint in contrast to the view of a pure theoretician (see §2.2 and §4.2).

Coming back to my former issue on discontinuity handling: What has happened since the Urbana meeting? I was glad to realize that there were at least a few mathematicians who took my comments seriously enough to think of some remedies:

(i) For nonstiff problems, the discontinuity-handling problem had already been solved prior to the Urbana meeting by people like Pritsker (1974) and myself (Cellier 1979) in a fairly general way. Applying an R-K algorithm, we just have to make sure that the discontinuity takes place at the end of an integration step. Time events are handled simply by reducing the step size if the next event is shortly ahead. State events are handled by iterating back to the unknown event time by any available method. Pritsker uses bisection, whereas I resorted to inverse Hermite interpolation. For obvious numerical reasons, it is important *not* to code the discontinuity itself in the ODE set but only the state conditon (e.g. by means of conditional termination criteria). The discontinuity itself is then expressed by context switching during the execution of the event (after execution of the event, another set of ODEs becomes active). A somewhat more mathematical view of this procedure can be found from Mannshardt (1978).

(ii) For stiff problems, one usually wants to apply multistep integration, for which the step-size adjustment at least creates a certain amount of overhead. The now common approach is to maintain two independent clocks, the *external* simulation clock and the *internal* integration clock. The integration proceeds with optimized step size and order, whereas the synchronization with the simulation clock is done by interpolating back using the Nordsieck vector. This methodology also may be applied to discontinuity handling if the state conditions are formulated as an adjoint set of discontinuity functions with the meaning that the simulation run terminates at either the final time or when the first of these functions crosses through zero in either direction, whatever comes first. According to my knowledge, we owe this formulation to (Carver 1977).

(iii) Meanwhile, two of the available GEAR codes, one by Carver, Stewart, Blair, and Selander (1978) and the other by Kahaner (1979), have been upgraded to contain a discontinuity handling mechanism. The Kahaner implementation differs from the Hindmarsh implementation in that the former DIFSUB subroutine has been modularized into about twenty smaller subroutines which are now much more understandable and which avoid all those "dirty" GOTO statements pointing backward in code.

(iv) I might want to suggest the introduction of an additional flag to determine whether all crossings are to be detected or whether only positive or only negative crossings are important. This is not really essential, but it is useful in many applications to formulate hysteresis effects, e.g. a heating system is switched on when the temperature falls below 18 degree centigrade, while it is switched off only after the temperature has reached 21 degrees centigrade as too frequent switching may damage the switching mechanism. By use of the above mentioned flag, a lot of unnecessary context switching can be avoided which makes the user programs execute more efficiently.

(v) Unfortunately, the aforementioned mechanism does not yet solve all the problems we would like it to solve. This is because after a discontinuity has taken place, the integration needs to be restarted. In the case of the GEAR codes, this means that the algorithm has to start again at an order of one. If discontinuities occur at frequent rates, the integration algorithm needs to be restarted again and again. One easily ends up having an extremely inefficient implementation of the trapezoidal rule, as higher orders get no chance to build up.

(vi) It is quite frequent in engineering applications that the accuracy requirements are not very severe. Then, an efficient low-order algorithm may do a better job on the problem.

(vii) Point (vi) was realized by Deuflhard who developed a new low-order code for stiff integration which looks very promising (Deuflhard, Bader, and Nowak 1980). For such an algorithm, the discontinuity handling also becomes easier.

(viii) If higher accuracy requirements are important, a higher-order algorithm has its advantages. A typical engineering application for this would be the simulation of a combined analog and digital circuitry containing memory elements

(flip flops). In such applications, it is extremely important to know whether spikes are around, and what the maximum trasient voltages in some parts of the analog circuit are. The answer to these questions is very sensitive to parameter variations and also to event timing. For this reason, the system usually must be simulated with a relative accuracy of from 10^{-6} to 10^{-8}.

(ix) After I mentioned this problem to Gear, he realized that his algorithm could be substantially improved if the warming-up period could be made more efficient, that is, if one could avoid having to restart at one one after discontinuities. He developed in the meantime R-K starters for GEAR codes (Gear 1980d). Although I have not yet found the time to implement such an algorithm personally, I am fully convinced that this amendment will make the code substantially faster when frequent discontinuities occur.

The question of discontinuity handling is by no means settled. To illustrate, let me discuss the following application problem which I offered some time ago as a new bench mark problem for simulation software (Cellier 1979).

Given: a set of domino stones from a domino game (usually 55, but any number will do). Place these stones in a distance d from each other. If the first of these stones is pushed, all stones will fall. The question to be answered is, at which distance d between two consecutive stones is the chain velocity maximized.

This problem is of a sufficiently "green-grass" nature to refresh the heart of even a mathematician. However, the problem is by no means academic as precisely the same simulation problem arises in many practical applications, e.g. the heating of steel ingots in a steel soaking pit and slabbing mill, or chemical batch reactions with charging and discharging of batch reactors, or the traffic flow though an intersection where each car may be modelled by a set of discontinuous ODEs whereby new cars may enter the considered area at any time while old cars disappear from the region after they have stayed in the system for some time.

What is common to all these applications? Obviously, whenever a discontinuity arises, the entire structure of the problem may change, and even the number of ODEs varies with time. We call these problems "variable structure problems." Each domino stone in my benchmark problem has to obey Newton's law, being represented by a second-order system or by a set of two first-order ODEs. Taking the 55 stones of the game altogether, we obtain a 110th-order model. However, it makes little sense to program the model in this way as only a very small number of stones are moving simultaneously. Some stones may have fallen while others remain untouched. Moreover, the physical law governing the motion of any stone in the system is the same. It therefore makes much more sense to code an entity to represent any "model" stone and allow new stones of the prescribed type to be generated at event times while others may be destroyed at event times. It is evident that there will exist an interaction (possibly even continuous) between falling stones which has to be taken into account. Any ODE solver as they are currently marketed is theoretically able to handle this situation as long as the code provides for appropriate discontinuity handling mechanisms in the previously discussed sense. However, the portion of the program which remains to be user-coded will still be substantial. We feel that, again, the user should be relieved of that part of the coding which is common to all the aforementioned examples.

Software for this type of application is currently under development at our group. This software will consist of a FORTRAN coded subroutine package GASP-VI (Rimvall and Cellier 1982), an extension to the existing package GASP-V (Cellier and Blitz 1976), together with a PSCAL coded preprocessor (front end) COSY (Cellier and Bongulielmi 1979; Cellier and Rimvall 1981) to make the user interface a little more convenient and less error prone.

8.3.3 *Automated partitioning*

It is quite common to many applications (e.g. in chemical engineering) that some portions of the model are considerably faster than others (e.g. a chemically reacting system consisting of fast and slow reactions). It seem intriguing to try to reduce the overhead involved in the numerical integration of the slow subsystem by splitting the system into fast and slow portions and by using different step sizes and possibly different algorithms for the two of them (Palusinski and Wait, 1977). This works quite well for some applicatons, e.g. for self-tuning regulators with a fast inner loop and a considerably slower outer adaptation loop. We tried this out by use of the SIMNON software (Elmquist 1976, 1977).

However, this partitioning scheme is not always easily done. It requires expertise to master such software. Moreover, it is not guaranteed that such a partitioning scheme even exists for a particular stiff system. In fact, if the system is nonlinear, the eigenvalues of the Jacobian may move around freely with time, and it may well happen that some modes of the system are fast during some period of time and slow during other periods. Carver came up with a brilliant idea (because extremely simple) for automated partitioning, which he presented during the International Conference on Simulation at Interlaken (Carver and MacEwen 1980). His method requires one single additional parameter to be user-tuned, a parameter which even has some physical meaning assigned to it and which makes the adjustment reasonably easy. Except for this parameter, the partitioning is fully automated and even adaptive in that it may vary with time.