Proceedings of ALGORITMY 2000
Conference on Scientific Computing, pp. 217–226

ANATOMY OF AN AUTOMATIC SOLUTION GENERATOR FOR DIFFERENTIAL EQUATIONS

DANA PETCU*

Abstract. Designing a new automatic numeric solver for ordinary differential equations is motivated only if it answers some critical user needs. Such needs and their solutions are discussed in this paper. We describe also the capabilities of a prototyped problem solving environment.

Key words. Software for scientific computations, parallel methods, scientific visualization

AMS subject classifications. 65L05, 65L12, 65M20, 65Y05, 68T20

1. Introduction. A vast collection of mathematical software, representing a significant source of mathematical expertise, is available for use by scientists and engineers in their efforts for modeling the solution of real systems. Designing a new mathematical software is motivated only if some user requests are not fulfilled by the actual software. In this paper we underline some of such requests (Section 2) and we shortly describe the facilities of our prototype of a solving environment (Section 3) for initial value problems (IVPs) for ordinary differential equations (ODEs) of the following form:

$$y'(t) = f(t, y(t)),\ t \geq t_0,\ y(t_0) = y_0,\ f : [t_0, t_0 + T] \times R^n \to R^n \tag{1}$$

Our prototype's attempt to simulate the reasoning of an expert in writing a program to solve and plot the ODE solution qualifies it as an expert system. Numerical tests demonstrate the prototype capabilities to solve large IVPs using efficiently the parallelism across method strategy. Note that only few experimental expert systems have been recently developed to address the needs in IVP problem area. We mention here Plod [1], Odexpert [5] and Godess [6] (all without capabilities to solve very large systems of ODEs with sequential, parallel or distributed methods). Solving ODEs is also an actual issue for computer algebra systems developers [10].

2. Software requirements. Non-scientific software for small computers raises the standard by which scientific software is judged. Users expect a similar level in technical applications.

2.1. Human versus automated expertise. In addition to knowledge of the software options, the user of a mathematical software must have considerable mathematical expertise in order to determine the category into a problem falls. The general problem solving tools, like Mathematica or Maple, can solve very easy a common problem, but special problems require special methods which must be written in the tool language or must be selected from a list (the user must have some knowledge in the field to do that).

Using automatic analysis (including symbolic where necessary) is more costly than solving the problem directly. It may itself involve trying to solve the problem with a possibly inappropriate code solely to provide information for the purposes of the analysis.

*Computer Science Department, Faculty of Mathematics, Western University of Timişoara, B-dul Vasile Pârvan 4, 1900 Timişoara, Romania (petcu@info.uvr.ro).

2.2. Expert systems. The purpose of an expert system is to determine the optimal solving method that may be applied to solve an input problem submitted by the user. An algorithm to determine the best method must evaluate all possibilities, which is practically impossible. The expert can only recommend a set of appropriate solving methods, selected by their suitability for the particular problem and the particular user, i.e. a partially ordered subset of solving methods. The ordering of this subset is based on heuristic estimate [2] of the relative suitability of each recommended method for the given problem. A method suitability is a quantitative measure that may be roughly defined as the degree of compatibility between the attributes of the module with the properties of the input problem and the desires of the user. Modules with a higher suitability value are determined to be better suited to solve the problem.

The algorithm that drives the recommendation must proceed in three stages: conflict detection, knowledge acquisition and suitable analysis [2]. In the first stage incompatibilities are sought between the method and either the properties of the input or the constraints given by the user (for example, a memory inefficient code). In the second stage information about the problem are obtained (relevant attributes affecting the performance of the module, but not essential in order for the module to be used; for example a preference for an easy-to-use-code). In the third stage computational methods and heuristic guidelines for estimating the suitability function are under investigation.

2.3. User interface and scientific visualization. Actual computer users are not tolerant of poorly designed interfaces, nor are they highly motivated to spend a large amount of time learning to use software. Instead they want interfaces that let them get started immediately, preferably without a manual. Interfaces that frustrate the user by demanding an extremely high threshold of knowledge can ruin the effectiveness of the software [1].

Critical ingredients of any software for exploring solutions of IVPs are flexibility, interaction, and graphics. These must work together and be flexible enough to allow a user to develop his own route to a solution.

An intelligent interface for ODEs supposes that their data may be entered in a relatively natural mathematical language. Parsers must be used particularly for error handling. A simple parser must be used to determine whether a user-input represents a valid number, and another to parse the differential equation model. A model parser must also identify independent variables, dependent variables, and supplementary variables. The values of the primed variable can be obtained re-evaluating the right hand side of the ODE system. Separating parsing from evaluation is appropriate if the same expression is to be evaluated at many different times.

Numerical procedures for solving (1) require the translation of the equations and initial conditions from their familiar mathematical form to a potentially complex language needed by the program. Mathematical entities, such as Jacobians, are nonessential from the users point of view; nevertheless they must be calculated for many implicit methods. Once the problem is solved, software is needed to display the data in a variety of ways. General plotting software is needed for use with almost every type of scientific problem solving software. Unfortunately, plotting is hardware dependent.

2.4. Implementation issues. The implemented software varies significantly in the way the trade-off among efficiency, space, accuracy and convenience of use are handled. Developing an expert system for mathematical problems poses some technical problems. For example, a special component must be designed, respectively that which allows a learning process. This component must provide at least two facilities: do not repeat the same mistake (i.e. an unsuccessful solving route for similar problems) and to learn about new solving methods.

Method authors usually present only few properties of the proposed method, so that the problem solvers (human or machine) often cannot decide if a method is good or bad for a problem solving process. Therefore an automatic solver can not be relied only on the theoretical results concerning particular methods and must provide a mechanism for detecting the method properties.

Solving ordinary differential equations, another problem is the impact of structure when using implicit methods. The cost of using different structures (full or sparse) depends on the size of the problem, on the overhead of using sparse solvers and on machine dependent factors such as availability of sufficient memory for the model used. The problem size makes frequently symbolic analysis (and possibly even numeric analysis) computationally infeasible. In such cases the user is very unlikely to be able to provide the missing information (like classifying stiffness – see below – on a graded scale), but the software must give reasonable recommendations.

2.4.1. Problem attributes. Problem attributes are the determining factors in recommending numerical solving methods. A particular attribute like the problem stiffness refers to a mathematical property of problems. Other attributes correspond more closely to user concerns as the efficiency of a numerical method for high accuracy computations or the memory requirements of the code. These attributes are relevant if the user has some preferences regarding time and accuracy [2].

An IVP is said to be stiff if its solution has components with widely different time scales and the solution is dominated by the slowly varying components [3]. A measure of this property which is often recommended is the stiffness ratio, i.e. the ratio of the real parts of the maximum and minimum eigenvalues of the Jacobian of f in (1). The difficulty here is that this ratio does not inform us whether the components of interest are the slowly varying ones, nor take account of the fact that the accuracy required of the solution will play a large part in determining the integration code. Codes for stiff problems generally use implicit methods and so will cost more to use them on the same integration interval. A problem may be only mildly stiff, but sufficiently so to make stiff software competitive, yet fall bellow a stiffness threshold where one would be certain that it is correct to classify the problem as stiff.

2.4.2. Difference methods. The volume of difference methods for ordinary differential equations is tremendous. Each method author can find some IVPs for which his method works better than others (with different meanings: convergence ratio to the exact solution, time spent in solving process, solution accuracy, optimal implementation etc.). This is a consequence of the fact that there are no perfect numerical recipes for solving ODEs. The most used methods can easy lead to unexpected crashes into the solving processes if the IVPs have particular forms (like the default solving method from Maple, which in case of a stiff problem, cannot be applied with a reasonable step-size without causing overflows). Therefore an ODE solving tool cannot have a database of methods for solving all possible IVPs. The possibility to extend the database must be taken into account in designing the tool. The language in which a new method is described must be similar to the mathematical language in which the numerical methods are described in books.

Almost all current ODE solving tools implement only general linear methods. In order to establish a pattern for any difference method, including also nonlinear methods (efficient methods since they can be adapted to the problem), we propose a modification of the least known A-method form for an iterative method:

$$Y^{(n+1)} = AY^{(n)} + h_n\Phi(t_n, Y^{(n)}, Y^{(n+1)}, h_n),$$
$$e^{(n)} = Y^{(n)} - Z(t_n, h_n),\ n \geq 0,\ Y^{(0)} = \Psi(h) \tag{2}$$

where Ψ is a starting procedure, A is a matrix independent from the problem, the first equation of (2) is an advance formula, $e^{(n)}$ is a global error estimation for the approximate solution, $Z(t_n, Y)$ is a procedure for approximate solution validation, Φ is a problem-dependent function, $h_n = t_{n+1} - t_n$ is the step-size used to the current iteration step. Unfortunately, the studies concerning the mathematical conditions in which particular properties are fulfilled do not work with the general form (2), but with more particular cases, and generalizations are hard to be obtained.

A software for interpreting method of (2)-type must have at least the following components: one for describing in mathematical language any pairs (A,Φ) (Φ must refer to f from (1)), one for advancing to the next step, one for establishing the method starting values (if the method is a multi-step one), one for solving implicit equations (if the method is an implicit one, i.e. when $D_{Y^{(n+1)}}\ \Phi \neq 0$), and one for approximate solution validation and error control.

A starting procedure, Ψ, for multi-step methods is based on an advancing formula similar with that from (2). The main difference is that it is applied once and $D_{Y^{(n+1)}}\Phi = 0$, i.e. the method must be explicit. Other restrictions are related to some method properties like order and stability.

The selection of the implicit equation solver strongly influences the method properties. For example, using the simple iteration method combined with an implicit method, the property of A-stability of the difference method is surely lost (see Table 2). This fact is not underlined in the literature and can raise a lot of problems for the software user. Different implicit equation solvers must be provided in an ODE solving tool in order to preserve the properties of the ODE solving method.

The particular class in which a given method must be included can be found using some patterns. Our prototype analyzes the dependency graph between estimated values (components of $Y^{(n+1)}$) and the preceding estimated values (components of $Y^{(n)}$) in order to establish if the method is an explicit or implicit one, one-step or multi-step one, one-derivative or multi-derivative one etc.

The most important property of a difference method is the convergence to the exact solution if the step-size goes to zero. This supposes a method order greater than one and method stability.

Many formulae have been developed to mathematically describe the order of a formula (see for example [3]), but these formulae are method-class dependent. A general tool for evaluating difference method must identify each case and apply the known order formulae or can interpret the definition of the method order, i.e. to find the value of p from the following equality:

$$l_n(h_n) := y(t_{n+1}) - z_{n+1} = c^* h_n^{p+1} + \mathcal{O}(h_n^{p+2}) \tag{3}$$

where z_{n+1} is the component of $Y^{(n+1)}$ which approximates $y(t_{n+1})$ (using step-size h_n) when the values of $Y^{(n)}$ are the exact solution values. For our prototype we have select the second variant in the idea to allow any difference method to be described and analyzed although if it does not follows any classic pattern. The second reason for do so is that the starting procedure has a great influence on the scheme order. A big gap between the starting procedure order and the advance formula order can lead to a smaller order than the theoretical reported one for the solving method (studies in which the starting procedure has been neglected appear often).

Equation $y' = \lambda y,\ \lambda \in \mathbb{R}$ can be used as problem with known solution and l_n defined in (3) can be approximated:

$$l_n(h_n) \approx c_0 + c_1 h_n + c_2 h_n^2 + \cdots + c_m h_n^m,\ m \geq 15. \tag{4}$$

Our solution is to vary h_n so that a system in c_i is formed and can be solved. Note that $c_p \approx c^*$, the error constant.

The error control mechanism is the most difficult component to be designed since a general error formula for any kind of difference method cannot be established. There are some formulae for general linear methods [3] which estimate the method error at each integration step and rules how this will be propagated. In a particular implementation, the method error is only one component of the global error. In order to control the global error some heuristic methods have been considered in our prototype, based especially on experiments and numerical tests.

An solving method is stable for any stable ODE if there are K, n_0, h_0 so that:

$$\|Y^{(n)} - \overline{Y}^{(n)}\| \leq K\|Y^{(0)} - \overline{Y}^{(0)}\|,\ \forall h \in (0, h_0), n \geq n_0 \tag{5}$$

where $Y^{(n)}$ is produced starting from $Y^{(0)}$ and $\overline{Y}^{(n)}$ starting from $\overline{Y}^{(0)}$. There are several possibilities to describe stability: by absolute stability region, by stiff stability region, by stability function, by mathematical rules for being zero-stable, etc. Unfortunately the mathematical rules for being stable in some particular senses are restricted to different classes of methods and cannot be very easy generalized to any method written in (2) form. Moreover, in implementations, the stability properties of an implicit method are strongly influenced by the implicit equation solver. In this case theoretical results concerning only the ODE solving method can not be of great help for the stability of the entire solving scheme.

Our proposal is to establish the extent of the set $S = \{\lambda \in \mathrm{R} \mid \lambda < 0,\ (5) \text{ is true for } y' = \lambda y\}$. If $S \neq 0$ we have a stable method at least for some linear problems. By numerical tests and looking to the absolute stability regions of the most known methods we state that the half ellipse, constructed in the negative complex half-plane $h\lambda$ with 0 as origin, S as big (real) half-axis and $(0, i \max S/2)$ as smallest (non-real) axis, is included in the (linear) absolute stability region of the method.

2.4.3. Parallelism. The means of archiving parallelism in IVP solvers can be classified into the following categories [4]: parallelism across the system (problem or space), parallelism across the method, and parallelism across the steps (time). Parallelism across space is the possibility of partitioning the system of ODEs by assigning one single equation or block of them to a processor for concurrent integration. Parallelism across the method is the possibility of distributing the computational effort of each single integration step among distinct processors. It has the benefit of not requiring special properties of the given ODE, but it has also the disadvantage of giving a limited speedup. Parallelism across steps is the possibility of concurrently executing the integration over a certain number of successive time steps, yielding numerical approximations in many points of the t-axis in parallel.

In this paper we are concerned with the potential for parallelism across method, which seems to have the greatest potential for successful on a general-purpose ODE solving environment using the distributed computing power of a small number of processors connected into a local network. The degree of parallelism of a particular method can be establish from a convenient representation of the oriented graph of dependencies between the $Y^{(n+1)}$ and $Y^{(n)}$ components. More details about this technique can be found in [8]. Note that the generation of code for parallel computers involves several nontrivial operations which have no analog in code generation for one-processor architecture.

2.4.4. Solving process. The problem solving process for a given problem and a given method can be successfully only if the problem and method properties are correlated. The following correlations which must be done before the solving process will start. First the method step-size h_0 must be selected so that a given global error level will be respected by the approximate solution at each integration step. Moreover the method step-size h_0 must be selected so that $h_0\lambda_i(t_0)$ are inside the method stability region, where $\lambda_i(t_0)$ are the eigenvalues of the matrix $J_0 = f_y(t_0, y(t_0))$. The total solution computation time must be reasonable and, finally, ineffective method for particular problems must be excluded. If the problem to be solved is nonlinear or it is used a variable step-size or variable-method scheme, different re-correlations must be take into account when the solving process runs.

3. Prototype presentation. Prototype's name, EpODE, first presented in [7], is an acronym for ExPert system for Ordinary Differential Equations.

3.1. Goals. One prototype's goal is to automate many decisions associated with computer solutions of ODE systems. It is interactive, it requires little programming effort and it satisfies the usual requirements for an expert system: eliminates routine decisions by inexperienced users, and incorporate knowledge in the form of program flow and access to a high quality integration engine. We wrote also a small set of screen plotting subroutines to be used in the visualization of the approximate solution. Code for functions, Jacobians that are required by the appropriate numerical method is generated automatically. EpODE solves the model equations by parsing the input equations, classifying them, and choosing an appropriate numerical solution method and adequate solving parameters.

EpODE has been designed the respect the following criteria: to be easy to use, to solve large ODEs (on one-processor or multi-processor systems), to use current numerical methods (one-step or multistep methods, explicit or implicit ones), to be in the public domain (available at http://www.info.uvt.ro/˜ petcu/epode), to be a stand-alone software (program independence), to correlate automatic detected problem properties, method properties and user options.

One of the main objectives of this system is also the creation of a uniform programming environment in which sequential and parallel numeric ODE solvers can be easily implemented and executed.

3.2. Components and user interface. EpODE has a built-in, easy-to-use screen editor for creating and editing the ODE model. When parser detects a problem a panel will be displayed explaining the nature of the problem. After the integration the user can change initial conditions, equations, integration interval, integration parameters like iteration methods, error tolerance etc. Although far from complete, our symbolic interface and adaptive procedures offer several advantages to scientist and engineers. Using EpODE, equations are entered in a relatively natural language and code for appropriate functions (and Jacobian) is automatically generated.

Figure 1 presents some parts of the tool panels and Table 1 shortly describes the system capabilities. More details about EpODE design principles can be found in [9].

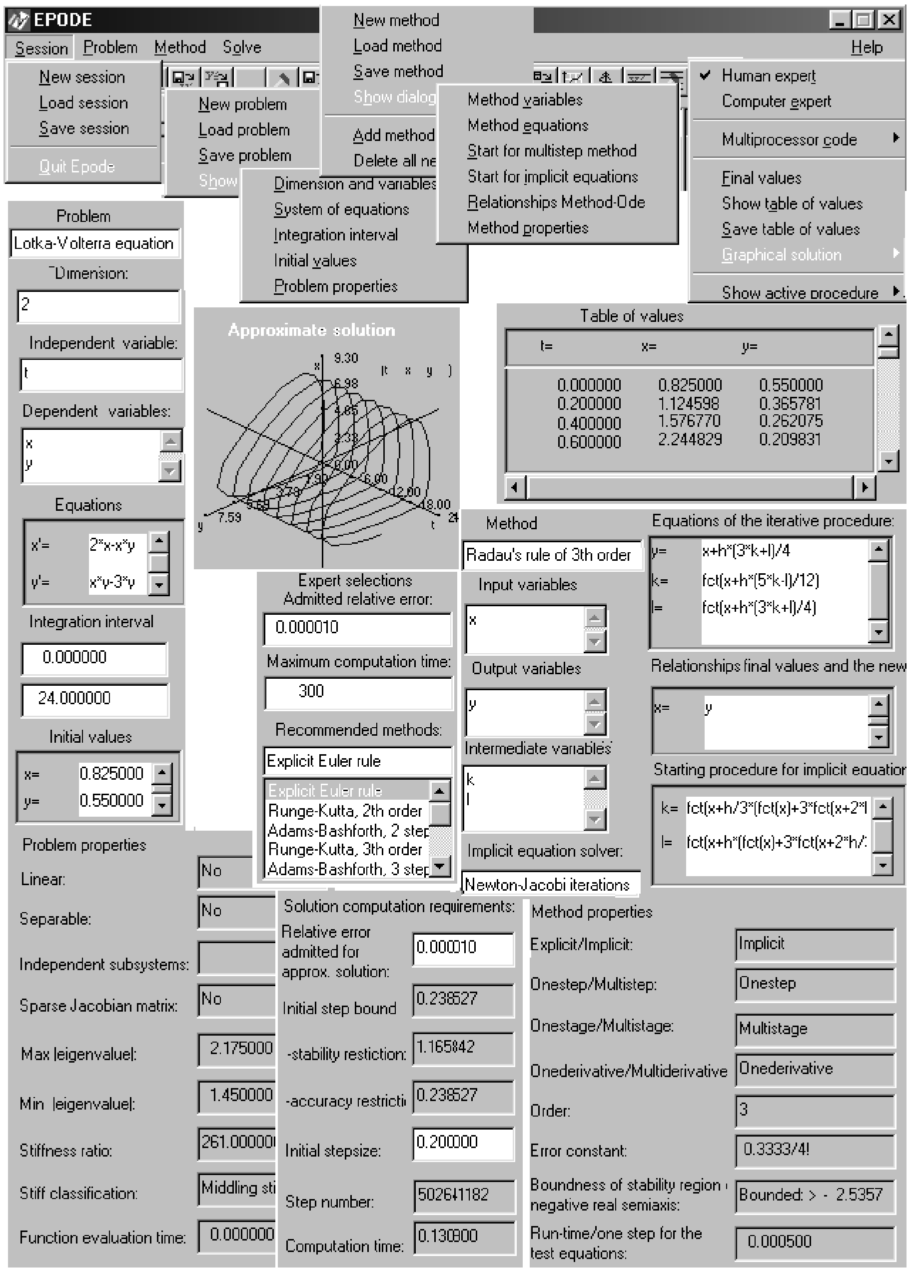

FIG. 1. *Images from EpODE user inteface*

3.3. Numerical tests. The results of the numerical experiments presented here refer to the functionality of the method property mechanism, the computer requirements when solving large IVPs and the distributed solution computations.

Table 2 presents three simple methods for which the method property mechanism was applied. In the second method case, the difference between the theoretical reported stability and the practical one is due to the used implicit equation solver.

TABLE 1
Prototype's components and capabilities

Component	*Facilities*
Interface	problem parser, difference method parser, solution visualization, menu-driven application, on-line help
Expert	problem and method properties detection, method and parameters selection, distributed task scheduler, efficiency measurements
Inputs	model equations, integration interval, initial values, max. computation time, solution accuracy level, options for visualization and distributed computations
Problem attribute detector	linearity, separability, sparsity, Jacobian matrix, Jacobian eigenvalues, stiff ratio, function evaluation time
Method attribute detector	explicit or implicit, one or multi-step, one or multi-stage, one or multi-derivative, order, error constant, absolute stability, parallelism degree, time per one-step for the test equation

TABLE 2
Method properties: theoretical established or reported by EpODE

	Exact	*Reported*	*Exact*	*Reported*	*Exact*	*Reported*
Method	Euler implicit rule		Obrechkoff implicit rule		Runge-Kutta standard	
Eqs.	$y_{n+1} = y_n + h_n f(y_{n+1})$		$y_{n+1} = y_n + \frac{h_n}{2}[f(y_n)+$ $+f(y_{n+1})] + \frac{h_n^2}{12}\cdot$ $\cdot[D_y f(y_n) - D_y f(y_{n+1})]$		$y_{n+1} = y_n + \frac{h_n}{6}\cdot$ $\cdot(k_1 + k_2 + k_3 + k_4)$, $k_1 = f(y_n)$, $k_2 = f(y_n + \frac{h_n}{2}k_1)$, $k_3 = f(y_n + \frac{h_n}{2}k_2)$, $k_4 = f(x + h_n k_3)$	
Impl.eqs.	Newton iterations		simple iterations		–	
Start.it.	$y_n + h_n f(y_n)$		$y_n + h_n f(y_n) + \frac{h_n^2}{2} D_y f(y_n)$		–	
Order	1	1	3	3	4	4
Er.const.	$\frac{1}{2}$	$\frac{1.0001}{2!}$	$\frac{1}{12}$	$\frac{2.0001}{4!}$	$\frac{1}{120}$	$\frac{1.0000}{5!}$
Stab.$\big\vert\mathbb{R}_-$	$(-\infty, 0)$	$(-\infty, 0)$	$(-\infty, 0)$	$(-2, 0)$	$(-2.78, 0)$	$(-2.78, 0)$
Type	Implicit	Implicit	Implicit	Implicit	Explicit	Explicit
Derivat.	One	One	Multi	Multi	One	One
Steps	One	One	One	One	One	One
Stages	One	One	One	One	Multi	Multi

By semi-discretization of one or more time-dependent nonlinear partial differential equations (PDEs) of the following form:

$$(6)\quad u_t = g(x, t, u, u_x, u_{xx}, \ldots),\ u(x, t_0) = h(x),\ b(x_b, t, u(x_b, t), u_x(x_b, t), \ldots) = 0$$

is obtained a particular class of nonlinear ordinary differential equation systems. In the method of lines (MOL), t is treated as a continuous variable and the partial derivatives u_x, $u_{xx}, \ldots$ are replaced with algebraic approximation evaluated at different space-points. This procedure lead to an ODE system with independent variable t. A lot of real time-dependent phenomena are modeled in terms of PDEs like (6). We mention here the models of chemical reaction-diffusion processes, penetrations of radio-labeled antibodies into tumorous tissue, turbulences etc. The movement of a rectangular plate under the load of a car passing across can be modeled by

$$u_{tt} + \omega u_t + \sigma \Delta\Delta u = f(x, y, t),\ u\big|_{\partial\Omega} = 0,\ \Delta u\big|_{\partial\Omega} = 0,\ t \in [0, T],\ u_t(x, y, 0) = 0,$$
$$f(x, y, t) = \begin{cases} 200(e^{-5(t-x-2)^2} + e^{-5(t-x-5)^2}) & \text{if } y = 4/9 \text{ or } y = 8/9, \\ 0 & \text{otherwise.} \end{cases}$$

where $\Omega = [0,2]\times[0,4/3]$, $\omega = 1000$, $\sigma = 100$, $t \in [0,7]$. The number of ODEs depends on the number of space grid points, which follows the accuracy requirements in the PDE solution. In the above described example, if we take the step size $\Delta x = \Delta y = 1/9$ we have 17×11 grid points and 374 stiff ODEs. As the accuracy requirement increases the spatial grid needs to be refined and this leads to a larger ODE system. Table 3 shows the different time scales in plate problem solving process.

TABLE 3
Time and storing requests in computing $u(3,x,y)$ on a Pentium III 450 MHz

Method	*Grid points*	Δt	*Iterations*	*Total time*	*Results*
Euler explicit rule	8×5	0.00001	300000	1.431 h	251 Mb
Euler implicit rule	8×5	0.00050	6000	0.116 h	5.2 Mb
Euler implicit rule	17×11	0.00010	30000	45.23 h	126 Mb

Using a computer expert system for ODEs, like EpODE, we can select the correct time-step in order to obtain the solution with the desired accuracy. The resulted ODE system is classified as a stiff one, i.e. the Euler explicit rule will suffer from step size restrictions due to the stability requirements.

If we want an approximate solution on a finer grid than those above used, we must think to appeal to a more powerful computational system. A natural approach for giving a positive answer to the need of faster ODE solvers is the use of parallel computers or distributed systems. The following results have been obtained using a network of 4 SUN Sparc stations connected into a virtual parallel machine.

Table 4 explains how the time is spent in one iteration step. The big gap between the time spent by an explicit method and an implicit one is due to the implicit equation solver. The third column indicates the time necessary to send the data between two computers of an network (three tests at different moments of the day). The different scales of time can explain why explicit methods are not efficient in a distributed computation environment.

TABLE 4
Details about one integration step applying the first method from the above table

No. eqs.	*Sequential times (s)*	*One function evaluation (s)*	*One Newton iteration (s)*	*Send-receive times (s)*		
				Test 1	*Test 2*	*Test 3*
16	0.87	0.44×10^{-5}	0.37×10^{-3}	0.06	0.15	0.15
32	2.82	0.90×10^{-5}	0.16×10^{-2}	0.16	0.08	0.18
80	11.6	0.18×10^{-4}	0.93×10^{-2}	0.09	0.15	0.09

Table 5 presents the time reduction in a solution computation process when two different parallel methods were applied. We see that the implicit method can significantly reduce the solution computation time. More results and comments concerning other parallel methods for IVPs are presented in [8].

4. Conclusions. We have described the design principles of a software tool for numerical solving ordinary differential equations which recommends a set of appropriate solving methods, selected by their suitability for the particular problem and the particular user. Its distributed computing facilities using parallelism across method can improve the time costs of the problem solving process. Large initial value problems for ordinary differential equations can be solved not only on parallel computers but also using network of workstations.

TABLE 5

Time per one step, speedups and efficiency results (plate model and two parallel methods)

Method	Diagonally implicit Runge-Kutta, A-stable, fourth-order, a 5-value, 3-level, 2-processor method: $k_1 = f(t_n + \frac{1}{3}h, y_n + \frac{h}{3}k_1)$, $k_2 = f(t_n + \frac{2}{3}h, y_n + \frac{h}{3}(k_1 + k_2))$, $k_3 = f(t_n + \frac{21+\sqrt{57}}{48}h, y_n + \frac{21+\sqrt{57}}{48}hk_3)$, $k_4 = f(t_n + \frac{27-\sqrt{57}}{48}h, y_n + h(\frac{6-2\sqrt{57}}{48}k_3 + \frac{21+\sqrt{57}}{48}k_4))$, $y_{n+1} = y_n + h[\frac{9+3\sqrt{57}}{16}(k_1 + k_2) - \frac{1+3\sqrt{57}}{16}(k_3 + k_4)]$				Block predictor-corrector method $\begin{pmatrix} y_{n+3/2} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} y_{n+1/2} \\ y_n \end{pmatrix} + \frac{h}{18}\Big[\begin{pmatrix} 7 & 0 \\ 0 & 3 \end{pmatrix} \cdot \begin{pmatrix} f^P_{n+3/2} \\ f^P_{n+1} \end{pmatrix} + \begin{pmatrix} 15 & -4 \\ 12 & 3 \end{pmatrix}\begin{pmatrix} f_{n+1/2} \\ f_n \end{pmatrix}\Big]$, $\begin{pmatrix} y^P_{n+3/2} \\ y^P_{n+1} \end{pmatrix} = \frac{1}{7}\begin{pmatrix} 3 & 4 \\ -28 & 35 \end{pmatrix}\begin{pmatrix} y_{n+1/2} \\ y_n \end{pmatrix} + \frac{h}{7}\begin{pmatrix} 15 & -6 \\ 14 & 7 \end{pmatrix}\begin{pmatrix} f_{n+1/2} \\ f_n \end{pmatrix}$			
No. eqs.	*Seq. times* T_s (s)	*Distr. times* T_d (s)	*Speedup* T_s/T_d	*Method efficiency* $T_s/(pT_d)$	*Seq. times* T_s (s)	*Distr. times* T_d (s)	*Speedup* T_s/T_d	*Method efficiency* $T_s/(pT_d)$
16	0.88	0.63	1.40	70%	0.01	0.61	0.02	0.8%
32	2.82	1.81	1.55	78%	0.02	0.63	0.03	1.7%
80	11.7	6.92	1.69	84%	0.05	0.65	0.08	4.1%
176	117	64.4	1.82	91%	0.25	0.78	0.32	16%
374	702	379	1.84	92%	0.85	1.02	0.84	42%

REFERENCES

[1] D. BARNETT, AND D. KAHANER, , *Experiences with an expert system for ODEs*, in Intelligent Mathematical Software Systems, E.N. Houstis, J.R. Rice and R. Vichnevetsky, eds. North-Holland, Amsterdam, 1990, pp. 5–13.

[2] I. GLADWEEL, AND M. LUCKS, *An automated consultation system for the selection of mathematical software*, in Intelligent Mathematical Software Systems, E.N. Houstis, J.R. Rice and R. Vichnevetsky, eds., North-Holland, Amsterdam, 1990, pp. 179-186.

[3] E. HAIRER, AND G. WANNER, *Solving ordinary differential equations II. Stiff and differential-algebraic problems*, Springer, Berlin, 1991.

[4] P.J. VAN HOUWEN, *Parallel step-by-step methods*, Applied Numerical Mathematics, 11 (1993), pp. 69–81.

[5] M.S. KAMEL, AND K.S. MA, *An expert system to select numerical solvers for initial value ODE systems*, ACM Transactions on Mathematical Software, Vol. 19, No. 1 (1993), pp. 44-61.

[6] H. OLSSON, *Object-oriented solvers for initial value problems*, in Modern Software Tools for Scientific Computing, E. Arge, A.M. Bruaset and H.P. Langtangen, eds., Birkhäuser, 1997, pp. 270-301.

[7] D. PETCU, *Implementation of some multiprocessor algorithms for ODEs using PVM*, in Recent Advances in Parallel Virtual Machine and Message Passing Interface, M. Bubak, J. Dongarra and J. Waśniewski, eds., Lectures Notes in Computer Science, Vol. 1332 Springer Verlag, Berlin, 1997, pp. 375–383.

[8] D. PETCU, *Solving Initial Value Problems with a Multiprocessor Code*, in Parallel Computing Technologies, V. Malyshkin, ed., Lectures Notes in Computer Science, Vol. 1662, Springer, Berlin, 1999, pp. 452–466.

[9] D. PETCU, AND M. DRĂGAN, *Designing an ODE solving environment*, in Advances in Software Tools for Scientific Computing, H.P. Langtangen, A.M. Bruaset and E. Quak, eds., Lecture Notes in Computational Science and Engineering, Vol. 10, Springer, 2000, pp. 319–338.

[10] L.F. SHAMPINE, AND M.W. REICHELT, *The Matlab ODE Suite*, SIAM Journal on Scientific Computing, Vol. 18, No. 1 (1997), pp. 1-22.