Initial Value Problems for ODEs in Problem Solving Environments

L. F. Shampine
Mathematics Department, Southern Methodist University, Dallas, TX 75275, USA

R. M. Corless
Department of Applied Mathematics, University of Western Ontario, London, Ontario, N6A 5B7, Canada

1 Introduction

The problem solving environments (PSEs) Maple [8] and MATLAB [7] are in very wide use. Although they have much in common, they are clearly distinguished by the emphasis in Maple on algebraic computation and in MATLAB on numerical computation. We discuss here a program, IVPsolve, for solving numerically initial value problems (IVPs) for systems of first order ordinary differential equations (ODEs), $y' = f(x, y)$, in Maple. We draw upon our experience with a number of closely related solvers to illustrate the differences between solving IVPs in general scientific computation (GSC) and in these PSEs. The RKF45 code of Shampine and Watts [10,11] is based on the explicit Runge-Kutta formulas F(4,5) of Fehlberg. It has been widely used in GSC. Translations of this code have been the default solvers in both Maple and MATLAB. Neither takes much advantage of the PSE. In developing the MATLAB ODE Suite of solvers for IVPs, Shampine and Reichelt [9] exploit fully the PSE, as well as algorithmic advances. IVPsolve is the result of a similar investigation for Maple, though on a much smaller scale. It also uses the F(4,5) formulas for non-stiff problems.

In addition to general issues of solving IVPs in PSEs, we discuss specifics for the Maple PSE. Because the user is allowed to specify the precision, the floating point arithmetic of Maple is implemented in software. As the PSE has evolved, it has added facilities that allow users to work directly with the underlying hardware floating point arithmetic. IVPsolve exploits these facilities to solve IVPs faster. Using a continuous extension of the F(4,5) pair and a new design, IVPsolve handles output more efficiently and avoids numerical difficulties of the kind pointed out in [2]. The solvers of Maple look different.
to users and solve different computational problems. In contrast, it is possible to use all the solvers of the MATLAB ODE Suite in exactly the same way. IVPsolve achieves this in Maple. Methods for the solution of stiff IVPs require (approximations to) Jacobians. To make it possible to use all the solvers of the ODE Suite in the same way, Shampine and Reichelt approximate Jacobians numerically in the codes for stiff IVPs. This is accomplished in IVPsolve by using the tools of Maple to evaluate partial derivatives analytically. IVPsolve uses a Rosenbrock method for stiff IVPs, an excellent method for the PSE that is not widely used in GSC because it requires analytical partial derivatives.

**Conventions for this paper.** Because this paper discusses implementations of similarly-named solvers in different contexts (GSC, the PSE MATLAB and the PSE Maple), the following notational conventions are used to distinguish the solvers. For GSC, upper-case names such as RKF45, LSODE, and ODE/STEP, INTRP are used. For the MATLAB ODE Suite, we use the exact names `ode45`, `ode15s`, and `ode113`. For the built-in Maple routines of `dsolve[numeric]`, we use their exact names, `rkf45` and `lsode`, which are lower-case versions of the names of the corresponding GSC codes. The new routines that we have written are packaged together under the name NODES, but are referred to in this paper by their names IVPsolve, IVPval, and so on.

## 2 IVPsolve

A number of issues distinguish solving IVPs in PSEs from solving them in GSC. Obvious ones are interactive computation and graphical output. Less obvious but no less important are advanced language features and interpreted computation. The emphasis on graphical output in PSEs means that IVPs are solved to accuracies modest by the standards of GSC. This makes the convenience of default tolerances quite practical, a distinguishing characteristic of solvers in PSEs. It also means that fewer steps are taken. In GSC the \( f(x, y) \) are often so complicated that the cost of evaluating this function dominates all the remaining cost of the computation, the overhead. This is much less true of the problems solved in PSEs. Because of interactive computation, fewer steps, and relatively simple \( f \), overhead is much more important in PSEs. These factors influence greatly the choice of method, a matter we now examine in more detail.

### 2.1 Choice of non-stiff method

An explicit Runge-Kutta pair of orders 4 and 5 is very attractive for solving non-stiff IVPs in a PSE because the order is appropriate for typical accuracies
and the method has a low overhead. The original \texttt{ode45} of MATLAB is a translation of RKF45 with its F(4,5) pair, but the solver of the same name in the ODE Suite [9] that replaced it at version 5 is different in important ways. In particular, it is based on a different explicit Runge-Kutta (4,5) pair due to Dormand. The default solver of Maple, \texttt{rkf45}, is a translation of RKF45. All these codes do local extrapolation so that the integration is advanced at order 5.

Selecting a formula is a complicated matter that involves both theoretical considerations and experimentation. There have been many investigations that attempt to identify the “best” Runge-Kutta pair of moderate order, but most have been concerned with GSC. Some aspects of the formulas are much more important in the present context than in GSC. We have already mentioned the importance of overhead in a PSE. To emphasize the point, we note that the ODE Suite includes a translation, \texttt{ode113}, of the well-known Adams solver ODE/STEP/INTRP. It varies both order and step size so as to minimize the number of evaluations of \( f(x, y) \). This is very efficient in GSC, but the overhead is quite high, making it much less attractive in a PSE. That is why the MATLAB documentation recommends that \texttt{ode45} be tried before \texttt{ode113}. Furthermore, in a code with the convenient user interface and powerful capabilities of \texttt{ode45}, much of the overhead and a sometimes considerable portion of the computations are not directly associated with evaluation of the Runge-Kutta pair. Generally the number of stages is not important in GSC, so it is scaled out in comparisons. It is important in PSEs because generally there is a relatively small number of steps and the tolerances are relatively large so that failed steps are relatively common. Shortly we take up output and discuss the use of a continuous extension of the Runge-Kutta pair for output. We shall see that both \texttt{ode45} and IVPsolve evaluate their continuous extensions several times in the course of each step. Obviously the cost of the continuous extension, both in terms of evaluations of \( f(x, y) \) and overhead, is quite important in this context. We implemented several (4,5) pairs and even a (3,4) pair with attractive features and compared them experimentally. The decision was not easy, but we chose to use the F(4,5) pair with local extrapolation and the continuous extension of [3] in IVPsolve.

2.2 \textit{Choice of stiff method}

By far the most widely used method for solving stiff IVPs is a variable order implementation of the backward differentiation formulas (BDFs). Included in \texttt{dsolve[numeric]} is a translation, \texttt{lsode}, of the well-known BDF solver LSODE. The MATLAB documentation recommends that a code of this kind, \texttt{ode15s}, be tried first for stiff IVPs, this despite the high overhead due to variation of order and step size. The sparsity structure of the Jacobian is
crucial to solving large systems of stiff equations, so an important feature of ode15s is its capabilities for dealing conveniently with this. Analytical partial derivatives improve the robustness of codes for solving stiff IVPs, but they have been avoided in GSC because it may be inconvenient for users to supply them and \( f(x, y) \) may be only piecewise smooth. IVPsolve is limited to the stiff problems for which Maple can form analytical partial derivatives. Because Maple does not yet have functions for sparse linear algebra in hardware floating point, IVPsolve is also limited to relatively small systems for which Jacobians are treated as full matrices. We use the Rosenbrock (3,4) pair implemented as METH = 1 in the RODAS code [5]. This pair was constructed for local extrapolation, so the integration is advanced at order 4. Although the order is lower than that of the method for non-stiff problems, it is adequate for the PSE. The formulas are stiffly accurate, A-stable, and have small error constants. The overhead is low compared to a BDF code because the order is fixed and the formulas are linearly implicit.

2.3 Use of HF arithmetic

Speed is not one of the primary goals in a PSE. Notwithstanding this, the computation is interactive, so the faster it can be done, the better. In developing IVPsolve we aimed to accelerate the solution of IVPs in Maple by exploiting hardware floating point arithmetic (HF). This is an issue particular to Maple with its emphasis on algebraic computation because its floating point arithmetic is implemented in software (SF). The user can control the precision of computations by means of the environment variable Digits. Naturally SF is slower, but how much slower depends strongly on how the precision specified relates to HF. Invoking functions inside a call to evalhf causes them to be carried out in HF. We make heavy use of this function and in particular, use it to speed up the evaluation of the ODEs in the procedure supplied by the user. Unfortunately, some of the special functions cannot yet be evaluated in HF. This causes a considerable complication to the solver: it must begin by testing whether the ODEs can be evaluated in HF and if they cannot, it must use SF for all the computations. For this case, we set Digits := trunc(evalhf(Digits)) so that these computations are carried out in approximately the same precision as is available in HF.

The hfarray data structure was added to Maple to hold an array of numbers in HF. Converting between SF and HF representations of numbers is costly, so it is important to avoid this. At present some of the fast built-in functions cannot be applied to hfarrays, forcing either a conversion or a slower way of processing the data. Because we do not know in advance the number of steps required for an integration, we might have to adjust the sizes of the arrays that hold the output. hfarrays have fixed dimension and cannot be
extended inside a call to `evalhf`, so we have to return from the function that advances the integration to the main program in order to create a larger array for the output, copy the current output array into the new one, and then return to the function through `evalhf` to continue the integration with this larger array. Because Maple does not yet provide for solving linear systems stored as `hfarrays`, we translated some FORTRAN programs of C. B. Moler for this that we use inside a call to `evalhf` to obtain the speed of HF.

2.4 Output considerations

RKF45 provides its output in the course of the integration, either at specific points or at each internal step, by returning control to the user along with the computed solution. Exploiting the dynamic storage allocation of MATLAB, the translation of RKF45 into this PSE returns control to the user only when the integration is complete (or fails). It returns two arrays, one containing the mesh points and the other, corresponding solution values. This form of the output is perfectly suited to the plot routines of MATLAB. Because of the emphasis on graphical output, the translation did not follow RKF45 in allowing output at specific points, a deficiency remedied in the ODE Suite.

RKF45 obtains output at a specific point by stepping to the point. Reducing the step size for this reason reduces the efficiency of the integration. The natural step size of the F(4,5) pair is so large that solution values at each step may not result in a smooth graph. The development of continuous extensions made it possible to deal with these matters. The solvers of the MATLAB ODE Suite have two output modes. The user either specifies all the points where output is desired or accepts output at the points selected by the solver. In the first mode, the solvers use the largest step size possible and obtain output at the specified points by evaluating a continuous extension. In the second mode, the explicit Runge-Kutta solver `ode45` supplements the solution values at mesh points by evaluating its continuous extension at four points in the span of each step. It was found by experimentation that four additional solution values generally suffice for a smooth graph. If they do not, the user has to increase the number of additional solution values by means of an option and solve the problem again.

Output is handled very differently in Maple. `dsolve` makes available a number of methods for solving ODEs analytically. Because the numerical solution of IVPs is treated as an option, it is natural that the design of `rkf45` resembles as much as possible the analytical solution of an IVP. In particular, the solver returns a procedure for computing the solution at any specified value of the independent variable. Because the computational problem is not the same as the analytical problem, this design leads to anomalies. A fundamental difference is
that the stability of the IVP is crucial to numerical integration. This stability depends on both the direction of the integration and the length of the interval. For the sake of efficiency, \texttt{rkf45} remembers the last approximate solution and advances the integration from it to the value of the independent variable specified in the current call. This is faithful to RKF45 except for allowing the direction to change. Changing direction can have serious consequences noted by Coombes et al. [2] that are revealed by considering what happens if we start with initial value at $x = 0$, compute a solution at $x = 1$, then at $x = 2$, and again at $x = 1$. It is obviously inefficient to recompute the solution at $x = 1$. However, the more important point is that if the IVP is very stable in one direction, it is very unstable in the other, so integrating back to $x = 1$ is expensive and inaccurate. The two values computed at $x = 1$ are different and can be dramatically different. Some of the methods of \texttt{dsolve[numeric]} have an error control that depends on the history of the integration and so get different answers for an entirely different reason. Getting different answers at the same point is rather unsatisfactory. There is a more subtle effect of the design. If the output points chosen by the plot routines do not result in a smooth graph, the user can increase the number of points plotted. This causes \texttt{rkf45} to solve the problem again with shorter steps, resulting in a more accurate solution. It is rather unsatisfactory that simply asking for more plot points changes the computed solution. Generally this change is not visible in the plots, but we give an example in §3 for which the plotted solutions are qualitatively different.

\texttt{IVPsolve} requires the user to specify that the integration is to go from $a$ to $b$ so as to avoid dangers inherent in the design of \texttt{dsolve[numeric]}. \texttt{IVPsolve} integrates the IVP over the whole interval and returns the mesh, the solution at the mesh points, and some other information in a structure. We exploit the more complex data structures available in the PSEs to simplify the user interface. In our design all information about the solution is encapsulated as a structure. The user has no reason to examine the contents of this structure because auxiliary procedures are used to evaluate and plot the solution.

Some details about the form of the solution are required. Enright et al. [3] develop a continuous extension for the Fehlberg (4,5) pair that does not require any additional evaluations of $f(x, y)$. It furnishes a numerical solution $S(x) \in C^1[a, b]$ that is a quartic polynomial in the span of the step from $x_n$ to $x_n + h$. In \texttt{IVPsolve} this polynomial is evaluated at $x_n + jh/4$ for $j = 1, 2, 3$. The numerical solution is uniformly accurate, so these approximations at intermediate points are given equal treatment in the output array. A great deal of experience with the F(4,5) pair in \texttt{ode45} and a limited amount with \texttt{IVPsolve} shows that this many equally spaced solution values generally yields a smooth graph. There is another reason for choosing five values that we take up shortly. We proceed similarly with the method for stiff IVPs. The Rosenbrock (3,4) pair was derived with a continuous extension that furnishes a numerical solu-
tion \( S(x) \in C^0[a, b] \) that is a cubic polynomial in \( [x_n, x_n + h] \). The continuous extension is evaluated at \( x_n + jh/3 \) for \( j = 1, 2 \).

2.5 Auxiliary procedures

IVPval is an auxiliary procedure for evaluating the solution at any point in the interval of integration. The user interface is simple: the user supplies the solution structure computed by IVPsolve and the \( x \) at which the solution is desired. The solution structure contains the information needed to evaluate \( S(x) \) anywhere in \([a, b]\). In more detail, the number of output points in the span of each step depends on whether the F(4,5) or R(3,4) pair was used, so this number is included in the solution structure. The number of solution values we chose to output for the step is not just enough to get a smooth graph, but also enough that polynomial interpolation of the appropriate degree on the span of the step reproduces the continuous extension there. IVPsolve first locates the step containing the \( x \) input and then evaluates \( S(x) \) by interpolation. In contrast to dsolve[numeric], the answer at a given point is always exactly the same. Furthermore, the IVP is integrated only once, no matter the number and location of points where an answer is desired. An attempt to evaluate the solution outside the interval where it has been computed results in a message to this effect and a reminder of the interval \([a, b]\) corresponding to the solution structure input. The MATLAB IVP solvers return the solution on a mesh and there is no way to obtain solutions at other points except by solving the IVP again. Kierzenka and Shampine [6] have written a MATLAB code for boundary value problems that deals with output much like IVPsolve. A version of this BVP solver that exploits new capabilities of the PSE will appear in MATLAB 6. The situation with the IVP solvers is different and illustrates an important point: The user interface of the Suite is uniform, but how capabilities are realized in the various solvers necessarily depends on the underlying method and these methods are quite different in nature. Adding a new capability can be challenging because it must be provided for all the solvers. In the present instance, it would be easy enough enough to follow IVPsolve in adding a new mode of output to ode45, but not to the variable order solvers, especially ode113 with its high orders.

The traditional description of ODEs and initial conditions of dsolve[numeric] is convenient for small systems of equations, but not for large. For this reason, in IVPsolve we follow the standard in GSC of expecting the ODEs to be provided as a procedure for evaluating a system of first order equations and the initial conditions as a vector. As a convenience we provide a procedure, ODE2proc, for converting the conventional description of an IVP in Maple to the procedure and vector expected by IVPsolve. Unlike MATLAB, Maple distinguishes clearly a scalar and a vector of one component. This complicates
the user interface for a scalar ODE and the coding of the solver: We allow vectors of one component for the sake of consistency, but IVPSolve and its auxiliary procedures also accept scalar IVPs because we expect that users will find them more natural. For example, it is more natural to specify a scalar ODE like \( y' = y^2 - x \) with an operator like \( f := (x, y) \rightarrow y^2 - x \) than a procedure.

Output from dsolve[numeric] in the form of procedures for evaluating the solution is convenient for the plot routines of Maple. Unfortunately, returning the solution in arrays as in MATLAB is not as convenient because it is necessary to form a list of pairs of points for plotting. In the PSEs it is best to use the builtin functions as much as possible because they are compiled and execute much faster. Plotting proved to be unacceptably slow in Maple because the builtin functions for forming such lists do not yet accept hfarrays. Eventually we learned that we could convey hfarrays to the plotting procedures by means of the CURVE data structure. This increased the speed of all plots by as much as two orders of magnitude. We also found that a low-level plot routine handled logarithmic scales inefficiently. Avoiding it increased the speed of such plots by more than another order of magnitude.

By default, IVPplot plots all the solution values returned, just as the MATLAB programs do. Generally this results in a smooth graph, but when the user zooms in on an area using the view option or when plotting in the phase plane, it is not unusual that straight line segments are visible in the graph. The refine option is used to deal with this. Invoking IVPplot with, say, refine = 2 instructs it to double the number of plot points. The additional solution values are equally spaced in the span of each step. They are formed by interpolation using IVPval. This way of getting a smooth graph is much more efficient than solving the IVP again as would be done in dsolve[numeric] or one of the MATLAB solvers.

A number of the solvers of dsolve[numeric] are translations of FORTRAN codes written for GSC. Each of the FORTRAN codes is a good one, but their authors treated important matters in very different ways and the methods themselves require certain differences in the user interface. These differences were not resolved in dsolve[numeric], so it is fairly described as a collection of solvers. In developing the MATLAB ODE Suite, it was considered essential that it be possible to use all the solvers in exactly the same way. In the context of this PSE, it was natural that the different methods be implemented in codes with different names. Methods for the solution of stiff IVPs use the Jacobian matrix and perhaps \( \partial f/\partial x \). If it is to be possible to solve stiff problems in the same way that non-stiff problems are solved, these partial derivatives must be formed internally. In MATLAB this is done numerically, so the only difference visible to the user between solving a non-stiff IVP with ode45 and a stiff problem with, say, ode15s is that the name of the solver is different. We
have implemented an explicit Runge-Kutta method for the solution of non-
stiff IVPs and a Rosenbrock method for the solution of stiff problems. In the
context of Maple it is more natural to implement both methods in the same
solver with a keyword used to indicate whether the problem is stiff. Because
we form partial derivatives analytically, the only difference visible to the user
between solving a stiff and a non-stiff problem with IVPsolve is that the
keyword \texttt{stiff} is set to \texttt{true}.

3 Numerical Examples

In writing IVPsolve for Maple, we aimed to make it easy and efficient to solve
an IVP, evaluate the solution, and plot the solution. We were confident that
the new solver would perform much better than those of \texttt{dsolve[numeric]}
in several ways. We hoped that it might be somewhat competitive with the
ode45 and ode15s solvers of MATLAB, at least when solving problems that
do not require the additional capabilities of those solvers. In this section we
compare the solvers using standard test problems. The NODES package and
a tutorial will be made available from http://www.apmaths.uwo.ca/~rcorless.

Comparing solvers is always difficult and the present situation is no exception.
Other solvers we consider have different aims and this has affected choices
made in the codes. For example, the error control of \texttt{rkf45} and IVPsolve is
the same, but the default error tolerances in \texttt{rkf45} are much more stringent
than those of IVPsolve. They are too stringent if the aim is to plot the solution
and arguably too stringent in general for an explicit Runge-Kutta formula of
only moderate order. Nevertheless, we use default tolerances for our examples
because we believe that is what the typical user does. As discussed earlier,
the fundamentally different designs of IVPsolve and \texttt{dsolve[numeric]} make
it impossible simply to compare the costs of integrating the IVPs; we must
include the cost of a plot or evaluation of the solution. Furthermore, the designs
imply that the cost of experimentation with plot options differs greatly because
\texttt{dsolve[numeric]} causes the IVP to be integrated for every plot and IVPsolve
merely evaluates a solution already computed. Although some of our examples
do call for such experimentation, we report only the cost of an acceptable
plot. To obtain representative run times, we computed the average over five
independent runs. Only the relative run times matter, and only in a gross way
at that, but the times reported are in seconds of computing with a PC running
at 450 Mhz.

A standard test problem for codes that solve non-stiff IVPs is a restricted
three body problem—spaceship, earth, and moon. Initial conditions leading
to a periodic orbit and its period were found numerically. Because the orbit
is sensitive to perturbations as the spaceship passes close to the earth, it is
necessary to use tolerances more stringent than the default values to reproduce the qualitative behavior of the orbit. This IVP is provided as `orbitode.m` with MATLAB 5 and we follow it in using relative error tolerance $1 \times 10^{-5}$ and absolute error tolerance $1 \times 10^{-4}$. When solving the IVP with `rkf45` we had to make a number of runs with different values of `numpoints` to find a value that results in a smooth graph. A value of 800 was used for the computations of Table 1. The solver based on Adams-Moulton methods, `lsode` with `adamsfunc`, was also used. It measures error in an RMS norm, so the tolerances must be divided by the square root of the number of equations to make them comparable to the maximum norm used by the Runge-Kutta solvers. Again it was necessary to experiment with `numpoints` and again 800 seemed adequate. The default output of both `IVPsolve` and `ode45` provides a satisfactory graph.

As in the computations resulting in Fig. 9.8.6 of Boyce and DiPrima [1], we integrated the Lorenz equations with $\sigma = 10, r = 28, b = 8/3$; initial conditions $y_1(0) = 5, y_2(0) = 5, y_3(0) = 5$; and plotted $y_3(t)$ against $y_1(t)$. `IVPsolve` monitors the work expended and when a maximum number of evaluations of $f(x, y)$ is exceeded, it terminates the run. An option allows this maximum to be increased as needed. We integrated over a longer interval than [1], namely $[0, 50]$, to exercise this option. Also, the large number of steps in this integration exercises the portion of `IVPsolve` that extends storage arrays. The cost of solving with `rkf45` depended strongly on `numpoints`. The default `numpoints` produces a graph that is completely unacceptable and increasing it to 1000 results in a graph for which straight line segments are still obvious. The entry in Table 1 corresponds to a `numpoints` of 5000, which gives an acceptable graph.

<table>
<thead>
<tr>
<th></th>
<th><code>ode45</code></th>
<th><code>IVPsolve</code></th>
<th><code>rkf45</code></th>
<th><code>lsode(Adams)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>3 body</td>
<td>0.59</td>
<td>0.24</td>
<td>2.20</td>
<td>4.58</td>
</tr>
<tr>
<td>Lorenz</td>
<td>2.10</td>
<td>2.81</td>
<td>6.95</td>
<td></td>
</tr>
</tbody>
</table>

Table 1
Non-stiff examples. Times to integrate and plot averaged over 5 runs.

We turn now to some stiff IVPs. In solving them with `IVPsolve` we used the Rosenbrock method specified by `stiff = true` and in solving them with `lsode` we used the BDF method specified by `backfull`. A standard test problem is the van der Pol equation, $y'' + 1000(y^2 - 1)y' + y = 0$, with initial conditions $y(0) = 2, y'(0) = 0$ on the interval $[0, 3000]$. The solution $y(x)$ converges quickly to a relaxation oscillation with very sharp changes. When we solved this IVP with `lsode`, we found that the graph did not have sufficiently sharp corners, so we increased `numpoints` to 100. This IVP is provided as `vdpode.m` with MATLAB 5 where it is solved with analytical Jacobian. How Jacobians are handled is important when solving stiff IVPs: `IVPsolve` creates internally procedures for analytical partial derivatives; `ode15s` of MATLAB
approximates the Jacobian numerically, but we have supplied a function for the analytical Jacobian in this instance; and \texttt{lsode} approximates the Jacobian numerically. (The code LSODE that underlies \texttt{lsode} does accept analytical Jacobians, but that capability was not implemented in the Maple version.)

Shampine and Reichelt [9] use the CHM6 problem of [4] to illustrate the stiff IVP solvers of MATLAB. The log-log plot of the second solution component seen in Fig. 1 shows regions of exceedingly sharp change. Correspondingly, the step sizes used by the solvers range over many orders of magnitude. The default relative error tolerance is acceptable, but some solution components are so small that an absolute error tolerance of $1e-20$ is appropriate. The solution components have such different behavior that it is appropriate to plot them in different ways. In particular, it is appropriate to use linear scales for the first component and a log-log plot for the second component. Table 2 gives the times taken to integrate the IVP and display a log-log plot of the second component. \texttt{ode15s} is a BDF solver that solved the IVP without difficulty, but that was not the case with the BDF solver \texttt{lsode}. Indeed, we were not able to make a meaningful comparison. The differential equations of CHM6 have three constant solutions, steady states. The solution of the initial value problem tends to one of these steady states as the independent variable tends to infinity. If the integration is not sufficiently accurate, the numerical solution might tend to the wrong steady state. We found this much more likely to happen with \texttt{lsode} than with the other two solvers. Particularly disconcerting is the fact that merely increasing the number of plot points sometimes caused \texttt{lsode} to compute a solution that tended to a different steady state, a consequence of the Maple design discussed in §2.
Conclusions

We have discussed briefly a new code, IVPsolve, for solving numerically IVPs for ODEs in Maple and its auxiliary codes IVPval for evaluating the solution, IVPplot for plotting it, and ODE2proc for converting a conventional description of ODEs in Maple to a first order system. The solver provided with Maple, dsolve[numeric], is a collection of programs with different user interfaces. IVPsolve implements two methods, one for non-stiff problems and one for stiff, that are used in exactly the same way. IVPsolve is significantly faster than dsolve[numeric] because its algorithms are tailored to the PSE and it exploits hardware floating point arithmetic. The design of IVPsolve avoids certain numerical difficulties inherent in the design of dsolve[numeric].

Our discussion of IVPsolve has served as a framework for discussing the solution of ODEs in general scientific computation and problem solving environments. In this a common thread has been the solution of non-stiff problems with the F(4,5) pair of Runge-Kutta formulas. We have seen that there are important and interesting differences between solving ODEs in GSC and in PSEs and also between solving ODEs in a PSE oriented towards algebraic computation like Maple and one oriented towards numerical computation like Matlab.

References


