Traps and Pitfalls in Simulation

Kjell Gustafsson

Ericsson Mobile Communications AB, S-223 70 LUND, Sweden,
Email: kgn@ldecs.ericsson.se

Abstract Numerical integration of ordinary differential equations is a central part of any software environment for modeling and simulation of continuos-time systems. We give a short introduction to numerical solution of ordinary differential equations, and describe some common problems when using this type of software in practice. The accuracy and efficiency of the simulation can often be improved by choosing an appropriate numerical integration method and setting its parameters correctly. Expressing the mathematical model on a form that suites the integration method is also important. We discuss these issues using simple and illustrative examples.

1. Introduction

Simulation is a powerful alternative to practical experiments. The properties of many physical systems and/or phenomena can be assessed by combining mathematical models with a numerical simulation program. Ordinary differential equations (ODE) are basic building blocks in such models.

Some programs, e.g. SYSTEM-BUILD [10], SIMULINK [12], PSpice [13], etc, provide complete environments for both the modeling as well as the actual simulation, while others, e.g. DASSL [1], RADAU5 [8], LSODE [9], STRIDE [2], specialize on the numerical solution of the ODE.

Many users regard the numerical integration of the ODE as a black box routine. Given a set of ODEs

$$\dot{y} = f(t, y), \quad t \in [t_0, t_f], \quad y(t_0) = y_0 \in R^N, \quad (1)$$

the user expects the software to find an accurate solution $y(t)$ almost independently of the character of the function f. The ODE could be nonlinear, include discontinuities or stochastic variables, and the integration should still be performed accurately and efficiently. Modern integration methods perform very well, but they do not yet arrive at this generality. In particular:

- The integration routine approximates the differential equation with a difference equation. This difference equation cannot retain all the properties of the ODE, and, consequently, there is no guarantee on the global accuracy of the generated solution.
- There are many different types of integration methods, and which one to use depends on the properties of the ODE. Choosing an inappropriate method may lead to inefficient simulation, or, possibly, erroneous results.
- The integration method relies on the solution $y(t)$ being smooth (f is assumed to be many times differentiable, i.e. $f \in C^p$). An ODE that includes discontinuities or stochastic variables therefore needs special attention.

When a simulation delivers results with poor quality, or maybe no results at all, the user can often improve the situation by e.g. rephrasing the ODE, choosing a different integration method or tuning its parameters. This, however, requires an educated user.

In this paper we will give a brief introduction to numerical integration of ordinary differential equations (for more complete presentations we refer to standard textbooks such as [3, 5, 7, 8]). The aim is to demonstrate what means a user has to affect the efficiency and accuracy of the simulation. Some classes of ODE's are difficult to solve numerically. We will, using a few simple problems, demonstrate how the situation can be improved by formulating the ODE in an appropriate way and/or by running the integration method correctly.

2. Selecting Integration Method

An integration method computes a solution to the initial value problem (1) by approximating it with

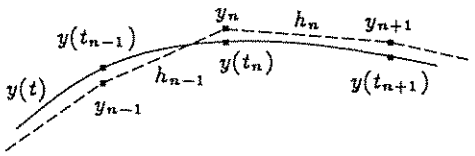

Figure 1. Repeated evaluation of the difference equation (2) produces solution points that approximates the true solution of the differential equation (1).

	y_{n+1}	y_n	$y_k,\ k<n$
explicit onestep		•	
explicit multistep		•	•
implicit onestep	•	•	
implicit multistep	•	•	•

Table 1. Different types of integration methods use different information when forming $\bar{y}_n$ and $\bar{f}_n$.

the difference equation

$$y_{n+1} = \bar{y}_n + h_n \bar{f}_n, \quad n = 0, 1, 2, \ldots \tag{2}$$

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

The discretization (2) can be defined in many different ways. The aim is, of course, that (2) should mimic (1) as closely as possible. Whether this is achieved or not depends on how $\bar{y}_n$ and $\bar{f}_n$ are constructed, as well as on the properties of the underlying ODE.

One way to categorize integration methods is to consider what information is used when forming $\bar{y}_n$ and $\bar{f}_n$, cf. Table 1. An integration method is a *onestep* method if $\bar{y}_n$ and/or $\bar{f}_n$ do not depend on solution points prior to y_n. If any $y_k, k < n$ is involved in the calculation, the method is said to be *multistep*. Complicated onestep methods normally involve iterated evaluations of f when forming $\bar{f}_n$. Such methods are known as Runge-Kutta methods. An *explicit* method does not include y_{n+1} in the definition of $\bar{y}_n$ and/or $\bar{f}_n$, and the new solution point can be calculated explicitly. An *implicit* method includes y_{n+1} in the construction of $\bar{y}_n$ and/or $\bar{f}_n$, and a nonlinear equation has to be solved in order to get the next solution point.

The *order* of an integration method relates to how the solution to the difference equation (2) approaches the solution of the differential equation (1) as the stepsize h_n tends to zero. If the solution is smooth, then $y(t_{n+1})$ can be expressed in terms

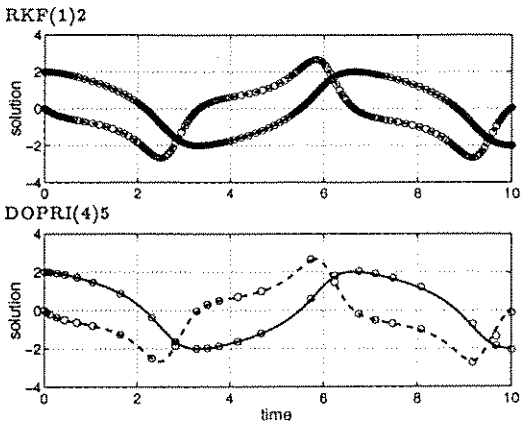

Figure 2. At the same accuracy requirement the second order method RKF(1)2 needs many more integration steps (each step is marked with a ring) to traverse the simulation interval compared to the fifth order method DOPRI(4)5. Each step is, however, computationally less expensive.

of $y(t_n)$ using a Taylor expansion, i.e.

$$y(t_{n+1}) = y(t_n) + h_n \frac{dy}{dt}(t_n) + \frac{h^2}{2}\frac{d^2y}{dt^2}(t_n) + \ldots \tag{3}$$

The discretization (2) is constructed to match as many of the terms in the Taylor expansion as possible. An order p method matches all the p first terms, and, assuming that $y_k = y(t_k), k \leq n$, the discretization error e_{n+1} behaves as

$$e_{n+1} = y_{n+1} - y(t_{n+1}) = \mathcal{O}(h^{p+1}) \tag{4}$$

Why Have Methods of Different Order?

It may seem beneficial to have integration methods with as high order as possible. The higher the order, the larger we could choose the stepsize and still obtain a numerical solution with sufficient accuracy. Each individual integration step is, however, computationally more expensive in a high order method, and therefore high order does not necessarily lead to less computation for a given accuracy. Which order that is the most efficient depends on the ODE, on the type of integration method, as well as on the required accuracy. It is, normally, most efficient to use high order methods for high accuracy and low order methods for low accuracy.

As an example, consider the van der Pol oscillator [7, pp. 107, 236]

$$\begin{aligned} \dot{y}_1 &= y_2, & y_1(0) &= 2 \\ \dot{y}_2 &= \sigma\left(1 - y_1^2\right) y_2 - y_1, & y_2(0) &= 0 \end{aligned} \tag{5}$$

with $\sigma = 1$. This problem was simulated using three different explicit Runge-Kutta methods (RKF(1)2 [7, pp. 174–175], DOPRI(4)5 [7, p. 171], and DOPRI(7)8 [7, p. 193]) of orders 2, 5, and 8,

tol	10^{-2}	10^{-4}	10^{-6}	10^{-8}
RKF(1)2	3.0	20.0	182.8	—
DOPRI(4)5	1.0	1.9	4.2	10.0
DOPRI(7)8	1.3	1.9	3.2	5.2

Table 2. The total work (measured as number of evaluations of the function f) needed to integrate the van der Pol problem (5) at different accuracy requirements using different explicit Runge-Kutta methods. The figures are normalized with respect to the work needed for DOPRI(4)5 at $tol = 10^{-2}$.

respectively. Figure 2 depicts how the low order method needs many more integration steps (each step is marked with a ring) to traverse the interval compared to a method of higher order. Table 2 summerizes the amount of work (measured as evaluations of the function f) needed to perform the integration at different error tolerances. The figures are normalized with respect to the work required for DOPRI(4)5 at $tol = 10^{-2}$. The required work grows as the accuracy requirement is tightened. The growth is fastest for the low order method, and slowest for the high order method. It is evident that, compared to a low order method, a high order method is much more efficient when asking for a very accurate solution. For low accuracies, however, it does not pay off to use a complicated high order method. In practice it is advantageous to have methods of different orders available.

Why Have Different Types of Methods?

Two methods with the same order may behave very differently on the same problem. The reason is that specific discretizations perform better on some classes of problems than other. No discretization is able to handle all types of problems equally well. Consequently, many different integration methods have been suggested over the years, each one with its own advantages and disadvantages. As with method order, it is important to have a spectrum of different methods available.

To exemplify how important the choice of method type may be, we will consider the Robertson problem [4, Problem D2]

$$\begin{aligned} \dot{y}_1 &= -0.04y_1 + 0.01y_2y_3 & y_1(0) &= 1 \\ \dot{y}_2 &= 400y_1 - 100y_2y_3 - 3000y_2^2 & y_2(0) &= 0 \\ \dot{y}_3 &= 30y_2^2 & y_3(0) &= 0 \end{aligned} \quad (6)$$

which originates from chemical reaction kinetics. After a fast initial transient, two of the states settle at a more or less constant value, while the third states increases almost linearly, cf. Figure 3. The problem was simulated with both an explicit Runge-Kutta method (DOPRI(4)5) and an

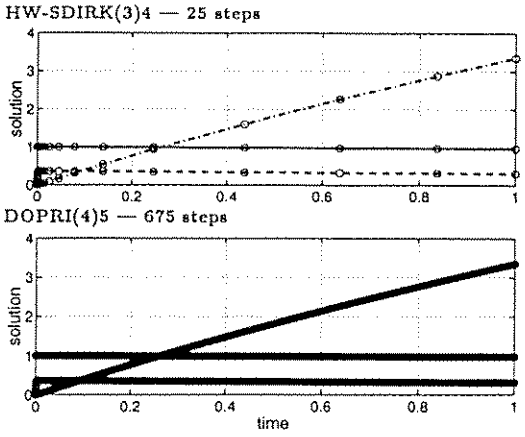

Figure 3. The Robertson problem (6) contains modes with very different time constants. The implicit integration method is able to handle such problems much more efficiently than the explicit method.

implicit Runge-Kutta method (HW-SDIRK(3)4 [8, p. 107]). The implicit method has one step lower order, but still executes the simulation with far less integration steps than the explicit method (25 steps and 675 steps, respectively). The explicit method uses so many steps that the rings indicating the individual steps form a thick line in the lower plot of Figure 3.

The Robertson problem contains modes with very different time constants. This is often referred to as a *stiff* problem. The fast mode settles quickly during the initial transient, and then the slow modes govern the solution. The implicit method exploit this by increasing its stepsize dramatically after the initial transient. The explicit method would become unstable if the stepsize was increased this much, and it is therefore forced to restrict the stepsize to a small value during the whole simulation. The implicit method needs to solve a nonlinear equation in order to obtain y_{n+1}. Each integration step is therefore more expensive compared to the explicit method. The number of integration steps are, however, reduced substantially, and the implicit method leads to much less total work than the explicit method.

Preserving the Properties of the ODE

One way of viewing the discretization is as a rational approximation of the exponential operator. It is not possible to get a good fit everywhere in the complex plane, and somewhere the approximation errors will be large. The stepsize acts as a scaling factor, and the accuracy of the approximation can be improved by reducing it. There will, however, always be a discrepancy, and this discrepancy may create a numerical solution with different properties than the solution of the ODE.

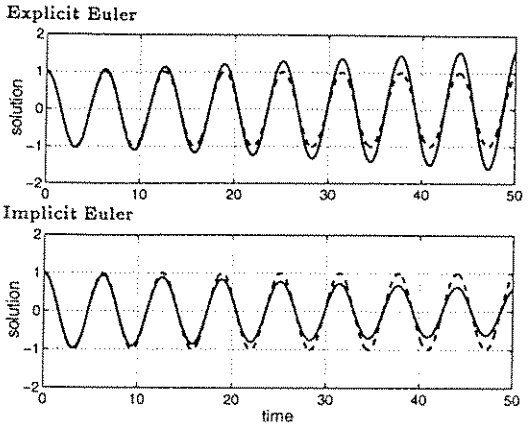

Figure 4. The discretizations resulting from using explicit and implicit Euler fail to preserve the stability of the underlying ODE. The explicit method shows an oscillation (y_1) with growing amplitude, while the oscillation created by the implicit method decays. The true solution (dashed line) has constant amplitude.

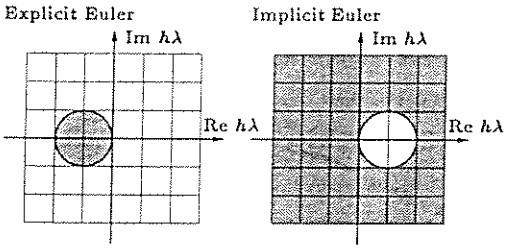

Figure 5. The gray areas depict the areas where the explicit and implicit Euler discretizations of $\dot{y} = \lambda y$ are stable.

Consider the linear oscillator

$$\dot{y} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} y, \qquad y(0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{7}$$

Figure 4 depicts y_1 resulting from simulating this problem with the explicit and implicit Euler methods, respectively (cf. [7, pp. 32, 199]). The integration stepsize was constant and equal to 0.02. This makes the error in each step very small; the problem is that they accumulate. As a result, explicit Euler portrays the system as being unstable (slowly growing oscillation amplitude), while implicit Euler creates a slowly decaying oscillation, indicating a stable system.

The two Euler methods discretize the problem $\dot{y} = Ay$ as

$$\begin{aligned} y_{n+1} &= (I + hA)y_n \quad \text{(explicit Euler)} \\ y_{n+1} &= (I - hA)^{-1}y_n \quad \text{(implicit Euler)} \end{aligned} \tag{8}$$

i.e. e^{hA} is approximated with $I + hA$ and $(I - hA)^{-1}$, respectively. The difference equations in (8) are stable when $\lambda h, \lambda \in \text{eig}(A)$ is within the gray area depicted in Figure 5. It is evident that it is easy to construct a stable ODE equation that

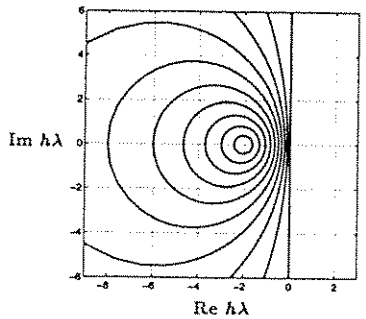

Figure 6. Level curves for the trapezoidal discretization (9) of the linear equation $\dot{y} = \lambda y$. The discretization equals zero at $h\lambda = -2$. The circles around this point represents the values $0.1, 0.2, \ldots, 1.0$.

explicit Euler will portray as unstable, or an unstable ODE that implicit Euler portrays as stable. Reducing the stepsize will make the problem less pronounced, but cannot remove it.

It is of course possible to construct integration methods that have the same stability region as the original linear ODE. One such example is the trapezoidal rule. It discretizes $\dot{y} = \lambda y$ as

$$y_{n+1} = \frac{2 + h\lambda}{2 - h\lambda} y_n. \tag{9}$$

This method preserves the stability of the linear problem, but instead misrepresents other properties. A very large negative λ-value corresponds to a fast stable mode. The trapezoidal rule will represent such a mode as a slowly decaying oscillating mode. This is evident when studying the level curves

$$\left| \frac{2 + h\lambda}{2 - h\lambda} \right| = \gamma, \quad \gamma = 0.1, 0.2, \ldots, 1.0, \tag{10}$$

which are plotted in Figure 6. When following the negative real axis from the origin towards $-\infty$, γ first decays from 1 to 0 ($h\lambda = -2$). It then increases again, and approaches 1 as $h\lambda \to -\infty$.

The bottom line is that a rational function cannot accurately approximate the exponential function $e^{h\lambda}$ everywhere in the complex plane. An integration method will correctly represent some properties of the underlying ODE, while misrepresenting others. A simulation program needs to implement a spectrum of different integration methods, and the user should experiment to see which method works best for his class of problems.

What Integration Method to Choose?

There is no "best" method! Different integration methods perform well on different classes of problems. A simulation should be repeated with different types of methods to check that the result is

indeed correct and not an artifact due to a bad combination of integration method and ODE.

In general,

- Use high order methods when asking for high accuracy, and low order methods for low accuracy.
- Use implicit methods when simulating problems with both fast and slow dynamics (stiff problems). This is especially important when trying to resolve what happens in steady state after fast transients have died out.
- Use onestep methods if the problem includes many discontinuities. These methods are, compared to multistep methods, less expensive to restart, and will therefore be more efficient when the simulation has to pass many discontinuities.
- Use low order explicit onestep methods when simulating combined discrete-time (sampled) and continuous-time systems. The sampling leads to frequent restarts making a multistep method inefficient. Often the sampling interval of the discrete-time system prevents the use of stepsize that would make an high order method or implicit method efficient.

3. Solution Accuracy

The accuracy of the numerical solution can be affected by varying the stepsize h_n. A large stepsize results in large errors, while a small stepsize leads to small errors. It is difficult for the user to relate a given stepsize to a specific accuracy and the choice is normally left to the error control algorithm within the integration method [6]. The user specifies an error tolerance *tol*, which the integration method tries to fulfill by varying the stepsize in relation to the behavior of the produced solution.

When having specified a value on *tol*, one may believe that the integration method will keep the global error of the numerical solution below this limit. What the software actually does is something very different. Suppose that the integration starts with correct initial values. The first solution point y_1 will deviate slightly from the true solution $y(t_1)$, cf. Figure 7. The new solution point lies on a different solution trajectory $y_1(t)$, which may behave differently than $y(t)$. Each integration step leads to local truncations errors e_n perturbing the numerical solution from $y(t)$. The local truncation errors propagate through the differential equation and accumulate to form a global error $y_n - y(t_n)$.

The global error is the fundamental measure of the quality of the numerical solution. This quantity is not computable (it requires that the true solution

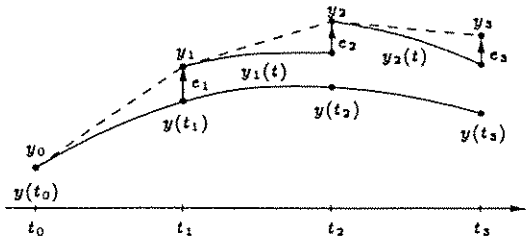

Figure 7. The error propagation in a numerical integration method.

is known), and in general, it is even difficult to estimate [15]. Most integration methods resort to estimating the local truncation error and keeping the norm $\|\hat{e}_n\|$ of this estimate below *tol*, in the hope that this will lead to a numerical solution with acceptable global error.

Some integration methods considers the estimate $\|\hat{e}_n\|/h_{n+1}$. This latter quantity is called error-per-unit-step (EPUS), while the former is called error-per-step (EPS). The heuristic motive to use EPUS is to get the same "accumulated global error" for a fixed integration time regardless of the number of steps used. Most integration methods will use EPS, because when controlling EPUS the simulation sometimes have a tendency to get stuck at discontinuities. Very few simulation programs will tell you what they actually have implemented.

The Choice of Error Norm

There a many ways to measure the size of the vector error estimate $\hat{e}$. For robustness, a mixed absolute-relative "norm" is often used. A common choice is

$$\|\hat{e}\| = \sqrt{\sum_i \left(\frac{\hat{e}_i}{\tilde{y} + \eta_i}\right)^2}, \qquad (11)$$

where $\tilde{y}_i$ is a (possibly smoothed) absolute value of y_i, and η_i is a scaling factor for that component of y. The idea with this type of norm is to relate the error in each solution component to the value of that specific component. In that way errors in different components will be equally important independently of the magnitude of the solution. The factor η_i prevents division by zero in the norm.

The parameter η_i is important. It tells when the i:th solution component should be regarded as small. A too small value will force ridiculously small integration steps when the solution passes zero, while a too large value removes the "relative" properties of the norm. Unfortunately, very few commercial simulation packages allow the user to affect η. Often it is set to an (undocumented) predefined value, being equal for all components.

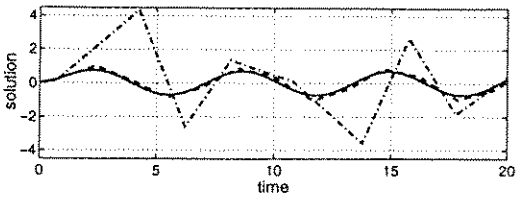

Figure 8. The accuracy of the numerical solution of y_2 in (13) is destroyed due to a poor choice of error norm. When y_1 is large it shadows the error contribution from y_2, and y_2 is not taken into account when choosing stepsize. The full/dashed/dash-dotted curves correspond to $y_1 = 0$, $y_1 = 10^3$, and $y_1 = 10^4$, respectively.

Some implementations use the norm

$$\|\hat{e}\| = \sqrt{\frac{\sum_i \hat{e}_i^2}{\sum_i \tilde{y}_i^2 + \eta}} \tag{12}$$

This is a poor choice of relative norm. If one solution component grows large, it completely blocks the influence from errors in smaller components. Consider the problem

$$\begin{aligned} \dot{y}_1 &= 0, & y_1(0) &= \beta \\ \dot{y}_2 &= -y_2 + \sin t, & y_2(0) &= 0 \end{aligned} \tag{13}$$

The two components are completely uncoupled, but by setting the initial value β for y_1 the accuracy in y_2 can be completely destroyed. Figure 8 depicts y_2 resulting from solving (13) with a fifth order Runge-Kutta method at $tol = 10^{-3}$. The numerical solution for y_2 is completely wrong when $\beta > 10^3$. One example, where this problem may be very difficult to diagnose, is when simulating a system with a component that increases steadily, e.g. the angle of a rotating motor, the position of a moving object, etc. The accuracy of the simulation slowly deteriorates as the solution value grows.

Tolerance Proportionality

Most error control schemes concentrate on keeping an estimate of the local truncation error bounded. One would hope that such a scheme leads to a global error that decreases when the error tolerance *tol* is decreased (so-called *tolerance proportionality*), but this is not always true. It requires special attention in the implementation of the integration routine to obtain this property, and far from all commercial implementations have managed.

Even a code that delivers tolerance proportionality will not have a perfectly linear relation between the global error and the choice of *tol*. To demonstrate this we return to the van der Pol oscillator (5). This time σ is set to 5 to create a problem with faster solution variations. Figure 9 depicts the solution of the problem as well as the stepsize

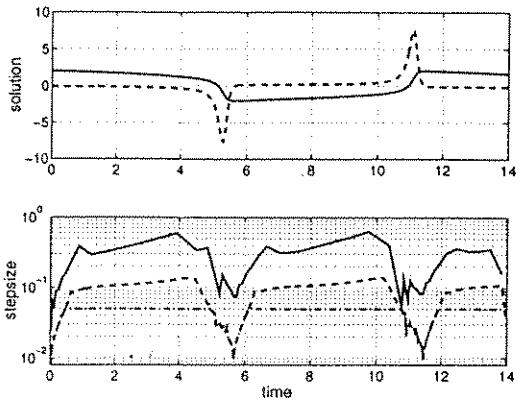

Figure 9. The upper plot shows the solution to the van der Pol oscillator (5) for $\sigma = 5$. The lower plot depicts the stepsize sequence needed to obtain the solution when $tol = 10^{-2}$ (full curve) and $tol = 10^{-6}$ (dashed curve). A stricter value on *tol* requires smaller stepsizes. The dash-dotted curve corresponds to $tol = 10^{-6}$, but with a restriction that the stepsize must be kept below 0.05.

sequences needed to obtain a solution at different accuracies. When the solution varies quickly the integration routine uses small steps, and at the flat parts the stepsize is large. Reducing *tol* scales down the stepsize over the whole simulation interval.

The van der Pol problem was simulated using two different Runge-Kutta methods (DOPRI(4)5 and DOPRI(7)8) of orders 5 and 8, respectively. Figure 10 shows how the global error (measured as deviation from the true solution at the end point $t = 14$) relates to the choice of *tol*. The global error decreases with decreasing *tol*. The decrease is, however, not monotonic. Sometimes small changes in *tol* cause large variations in the global error. In this case DOPRI(4)5 (the line with crosses) happens to have an almost one to one correspondence between global error and *tol*. DOPRI(7)8 (the curve with rings), on the other hand, happens to produce a solution with a global error that is much smaller than *tol*. In general, there is (at best) only a proportionality between the global error and *tol*, not an absolute relation.

Stepsize Restrictions

Most integration methods allow the user to specify restrictions, h_{min} and h_{max}, on the stepsize. Specifying a minimum and maximum stepsize may be helpful in problems that contain discontinuities. We will return to this issue in Section 4. The two most common reason to specify a maximum stepsize are

- the user wants a more accurate solution and tries to obtain that by restricting the stepsize, and
- the solution points are to sparsely spaced to

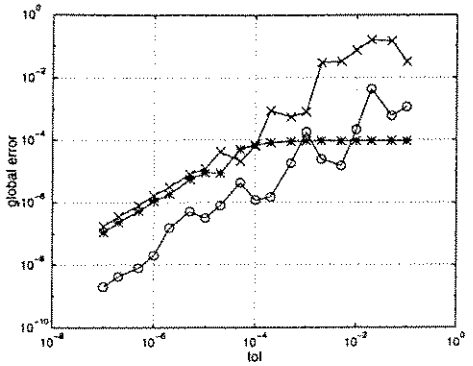

Figure 10. The global error at the end point as function of *tol*, when simulating the van der Pol oscillator (5) with $\sigma = 5$. Two different methods, DOPRI(4)5 (crosses) and DOPRI(7)8 (rings), were used. The global error decreases, however not monotonically, with decreasing *tol*. The curve with stars corresponds to DOPRI(4)5, but with the restriction that the stepsize must be kept below 0.05. This results in much extra computation without substantial gain in accuracy (cf. the region where $tol < 10^{-4}$). In the region $tol > 10^{-4}$, the stepsize restriction do lead to an accuracy gain. The reason is that h_{max} reduces the stepsize everywhere for these values of *tol*, cf. Figure 10. The computational cost is, however, high, cf. Figure 11.

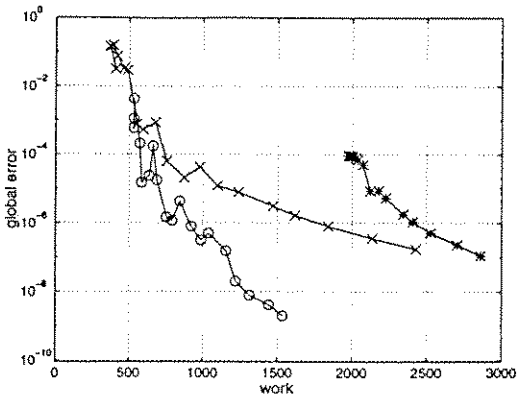

Figure 11. The global error as function of total work (number of evaluations of f) when simulating the van der Pol oscillator ($\sigma = 5$) from $t = 0$ to $t = 14$. DOPRI(7)8 (the curve with rings) achieves high accuracy with less work than DOPRI(4)5 (the curve with crosses). For high accuracies DOPRI(7)8 results in less work than DOPRI(4)5. Restricting the maximum stepsize to 0.05 forces DOPRI(4)5 to work harder for a specific accuracy (the curve with stars).

produce nice plots and more solutions points are forced by specifying a maximum stepsize.

Neither of these strategies can be recommended. In order to improve the accuracy the stepsize has to be reduced *everywhere*. Specifying a maximum stepsize will only reduce the stepsize in certain areas (cf. the dash-dotted curve in Figure 9). The price is a lot of extra computation without much gain in accuracy. This is well demonstrated by the results in Figures 10 and 11. Reduce *tol* instead of specifying h_{max} when aiming for higher accuracy.

Modern integration methods are constructed with embedded interpolators. This means that once a new solution point y_{n+1} has been obtained, one can fairly cheaply obtain an interpolated value for any point between t_n and t_{n+1}. The accuracy of the interpolation is similar to the one of y_{n+1}. If the simulation produces too few points for nice plots the extra points should be obtained through the interpolation and not by restricting the stepsize. Restricting the stepsize is computationally much more expensive.

All restrictions of the stepsize interfere with the error control within the integration method. This may destroy the proportionality between *tol* and the global error (cf. the curve with stars in Figure 10), making assessments of the quality of the produced numerical solution more difficult.

Error Control

The tolerance parameters in an integration method should be regarded as tuning knobs. Reducing *tol* will normally result in a more accurate solution. The relation is not monotonic, and a specific value on *tol* does not result in a specific global error. The simulation should therefore be redone at different error tolerances to check that the observed solution properties do not depend on an unfortunate choice of error control.

In general,

- Scale the problem such that solution components are about 1 in size, or set η in (11) componentwise. This makes the simulation less sensitive to a poor choice of error norm.
- Improve accuracy by decreasing *tol* instead of restricting the maximum stepsize h_{max}.
- Use a method with embedded interpolator if you want output points closer than the integration steps.
- Do not count on tolerance proportionality. In a professional implementation the global error is related to *tol*, but reducing *tol* a factor of (say) 10 does not necessarily result in a similar reduction in global error.

4. ODE's with Discontinuities

The construction of the discretization (2) is based on a Taylor expansion and assumes that the true solution is smooth. If the function f is not sufficiently differentiable (an order p method assumes

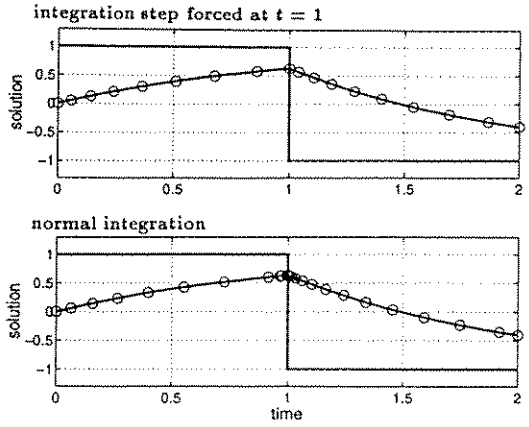

Figure 12. Two simulations of the ODE (14) using DOPRI(4)5. The integration steps are marked with rings. The upper plot depicts a simulation where the integration method was forced to take a step exactly when the input signal $u(t)$ changes from 1 to -1. The lower plot demonstrates what happens when the integration method is not informed about the discontinuity. A lot of unnecessary integration steps are wasted trying to pass $t = 1$, cf. Figure 13.

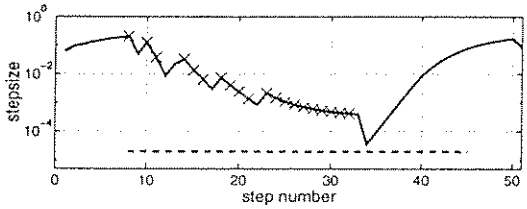

Figure 13. The stepsize sequence used in the simulation depicted in the lower plot of Figure 12. Steps that had to be rejected due to too large an error, are marked with a cross. Many integration steps are spent trying to pass the discontinuity at $t = 1$. The region marked with the dashed line corresponds to the integration steps taken between $t = 0.8$ and $t = 1.2$. Some 35 steps were spent on a time interval that normally would require 4 to 5 steps (cf. the upper plot in Figure 12).

f to be at least p times differentiable), this underlying assumption is no longer true. One should, consequently, not expect an integration method to handle an ODE with discontinuities in an accurate and efficient way.

To demonstrate the problem with efficiency, we will consider the following ODE

$$\dot{y} = -y + u, \quad t \in [0, 2], \quad y(0) = 0$$
$$u(t) = \begin{cases} 1, & 0 \le t \le 1 \\ -1, & t > 1 \end{cases} \qquad (14)$$

Figure 12 shows the result of two simulations of (14) using DOPRI(4)5. In the first simulation (the upper plot) the integration routine was informed about the step in u at $t = 1$. The integration

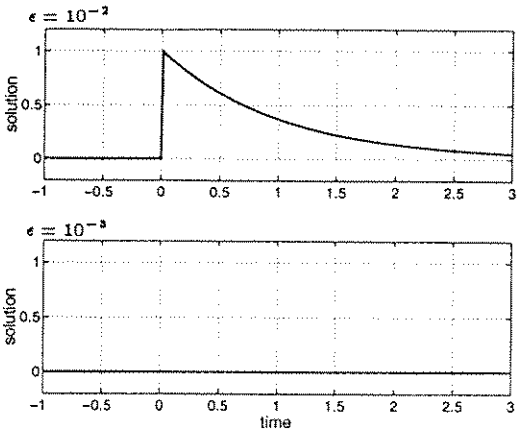

Figure 14. A simulation of the impulse response of a linear system (15). When the duration ϵ of the impulse becomes short, the integration method misses the event and produces an erroneous solution.

proceeded up till $t = 1$, the value of u was changed, and the integration continued. By "restarting" the integration at $t = 1$, thus avoiding to take a step *over* the discontinuity in u, the method is able to simulate the problem efficiently.

The lower plot in Figure 12 depicts what happens when the integration method tries to take a step over the discontinuity. Many steps are rejected due to too large an error. The stepsize is reduced four orders of magnitude, and is eventually small enough that the discontinuity can be passed with acceptable integration error. More than two thirds of the total computation is spent trying to pass $t = 1$, cf. Figure 13. If the integration method had used EPUS for error control the situation would have been even worse.

Some simulation programs recommend the user to specify a minimum stepsize h_{min}. This will prevent the integration method to reduce the stepsize too much when trying to pass a discontinuity. The efficiency of the integration is improved, but to the price of *switching off error control* when passing the discontinuity. This may lead to an erroneous solution.

Specifying a minimum stepsize is, normally, the wrong way to handle discontinuities. Often the locations of the discontinuities are known in advance (e.g. input signals in the form of steps or square waves, combinations of continuous-time and sampled systems, etc.), and the integration method can be forced to restart at these points, thus almost completely avoiding the problem. A well implemented simulation environment scans the description of the ODE and handles the problem automatically.

Another problem with discontinuous ODEs are "missed events." The integration method evaluates the function f at discrete points, and if an event has short duration, the integration method

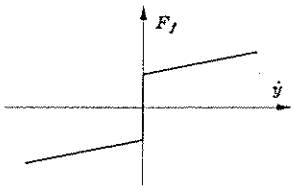

Figure 15. The friction force on a linearly moving object as function of the velocity $\dot{y}$. The friction force includes a static component that causes a discontinuity at $\dot{y} = 0$.

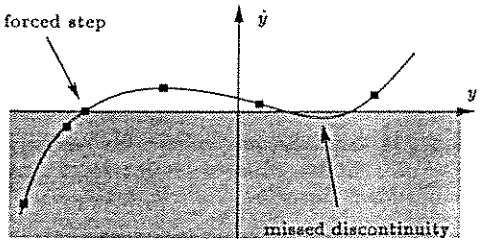

Figure 16. The integration proceeds step by step. The discontinuity in the friction force F_f is handled by looking for solutions of the indicator function $\dot{y} = 0$. An integration step is forced exactly at the point where $\dot{y} = 0$. Depending on where the indicator function is evaluated, some discontinuities may be missed.

may step over it without noticing it. As an example consider

$$\dot{y} = -y + \delta_\epsilon, \quad t \in [-1, 3], \quad y(-1) = 0$$
$$\delta_\epsilon(t) = \begin{cases} 1/\epsilon, & 0 \le t \le \epsilon \\ 0, & \text{otherwise} \end{cases} \tag{15}$$

Figure 14 depicts two simulations at different values of ϵ. The upper plot is the result of having $\epsilon = 10^{-2}$, and correctly portrays an approximation of the impulse response of the system. In the lower plot, ϵ was reduced to 10^{-3}. The integration routine does not evaluate f in the short interval where $\delta_\epsilon(t)$ is nonzero, and the result is an erroneous simulation result.

An inefficient solution to the problem of missed events is to specify a small value on h_{max}. This prevents the integration method from taking large steps and missing the event. A much better solution to the problem is, as before, to inform the integration routine about the location of the discontinuities. This will prevent the integration method from missing the event, and will also lead to an efficient integration when trying to pass the locations where $\delta_\epsilon(t)$ changes value. Many simulation programs allow for both continuous-time and discrete-time (sampled) models. By putting the generation of discontinuous signals inside a discrete-time model, the program can automatically schedule the time points where the integration routine should be restarted.

There are ODEs with discontinuities that depend on the value of the produced solution, which hence will occur at locations that cannot be predicted beforehand, e.g. a mechanical system with friction and/or backlash, electronic circuitry including switching components. This type of problems are very difficult to simulate accurately and efficiently. Too many simulation programs approach them by having the user specify h_{min} and h_{max}, and then hope for the best. An advanced integration method instead uses indicator functions in combination with a root solver [14]. We will describe the technique using a simple example.

Consider the ODE

$$m\ddot{y} = F - F_f$$
$$F_f = \alpha_1 \text{sign}(\dot{y}) + \alpha_2 \dot{y} \tag{16}$$

It describes the linear movements of a body, with mass m, affected by an external force F and a friction force F_f. The friction force depends on the velocity $\dot{y}$ as depicted in Figure 15. When the movement changes direction there is a discontinuity in the friction force. Simulating this ODE directly would lead to the same type of problems as depicted in Figures 12 and 13.

A better way is to separate the description of F_f into two models, one valid for positive $\dot{y}$ and one valid for negative $\dot{y}$. Neither of these models are discontinuous, and they are only used one at a time. A special indicator function, telling when to switch model, is also constructed. In our case it is simply $\dot{y}$.

The integration will proceed using one of the models for F_f. After each integration step the method checks whether there has been a sign change in the indicator function. If so, a root finder is called to find the exact location where the indicator function equals zero, i.e. for which t is $\dot{y} = 0$. The integration method provides an embedded interpolation function so the indicator function can, relatively cheaply, be evaluated anywhere between t_n and t_{n+1}. Once the location of the discontinuity has been found, a step is taken up to this time point, the model for F_f is changed, and the integration proceeds. The simulation program handles these steps automatically. The technique, however, puts a larger modeling burden on the user. Figure 16 exemplifies how the integration may proceed, and also indicates that some events may be missed depending on where the indicator function happens to be evaluated.

How to Handle Discontinuities

Simulating an ODE that includes discontinuities is a difficult task. All integration methods are constructed assuming a smooth solution, and are more or less unable to handle a discontinuous ODE

accurately and efficiently. The situation can be improved by expressing the ODE on a form that the integration method can exploit. Specifically,

- Force integration steps at known discontinuities, either by using a simulation program that is able to handle a combination of continuous-time and discrete-time models or by manually dividing the simulation interval into subintervals.
- Use an integration method that includes a root solver and which can handle indicator functions. Write your models to support this functionality.
- Remember that the integration method assumes $f \in C^p$, and discontinuous derivatives (e.g. a triangle wave as input signal) may be as harmful as discontinuous signal.
- Use h_{max} and h_{min} as a last resort to make the integration method handle discontinuities. The result is often poor accuracy or inefficient simulation, compared to the approaches described above.
- Multistep methods are expensive to restart and are therefore inefficient for ODEs with many discontinuities.

5. Noise

Stochastic differential equations (the function f includes stochastic elements) are very difficult to solve numerically [11]. The combination of solution points and function values that are used to form $\bar{y}_n$ and $\bar{f}_n$, almost always fail to correctly represent the contribution from the stochastic variable over the integration step. To obtain reasonable result requires both special numerical software and a very educated user.

Often the stochastic component in f serves only to investigate how the system reacts to a noiselike input signal, e.g. how does measurement noise in a sensor affect the attitude control in an airplane. This type of simulations are easier than the general stochastic differential equation case, but still requires special care.

To exemplify, we will describe an *erroneous* approach, that, unfortunately, is not uncommon even in commercial programs. Suppose that we want to investigate how the system $\dot{y} = -y + u$ reacts to a "white noise" input signal u. The input signal is generated by substituting u with a random number every time the function $f = -y + u$ is evaluated by the integration method. The random number is drawn from a normal distribution with zero mean and unit variance. Figure 17 depicts the resulting y when performing the simulation using DOPRI(4)5 at two different error tolerances.

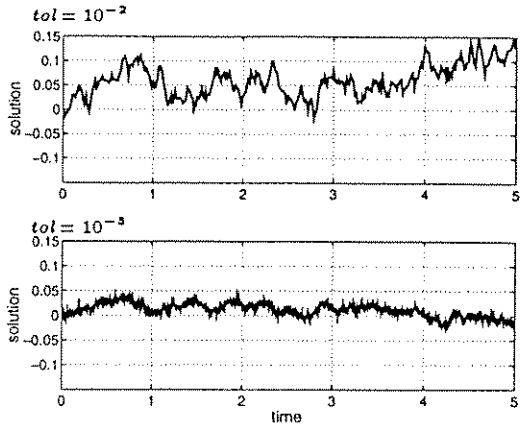

Figure 17. The plot depicts the result of two erroneous simulations of a linear first order system $\dot{y} = -y + u$ with "white noise" input u. The noise signal is represented with a random number each time the function $f = -y + u$ is evaluated. The simulation result depends strongly on the choice of error tolerance *tol* and the choice of integration method.

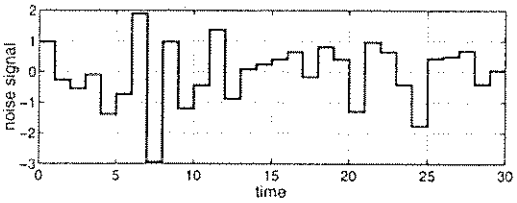

Figure 18. One way to approximate a continuous-time "white" noise signal is to use a piecewise constant signal, where each step has random amplitude. By choosing the sampling time sufficiently short, this signal will have an almost flat spectrum within the frequency region of interest. The variance of the signal is set by choosing the variance of the random numbers that decides the amplitude of the steps.

The result is very different for the two simulations. Both are wrong! DOPRI(4)5 samples the function f at a few places when forming $\bar{f}$. This will effectively form an average of the random u values, creating an input signal that is far from white. Depending on the stepsize the integration method uses, there will be different number of samples of u per unit time. This changes the effective variance of the signal u. When asking for a more accurate solution, DOPRI(4)5 is forced to take shorter integration steps, which results in a completely different result. Changing integration method (different way to form $\bar{y}_n$ and $\bar{f}_n$) also changes the result drastically.

To obtain reproducible and accurate results in this type of simulations, the "white noise" signal u has to be approximated with a signal that the integration method is able to handle. One way is to use a piecewise constant signal (cf. Figure 18) with random amplitude steps. The spectrum of this signal

can be made almost flat over the frequency region of interest by choosing the sampling period sufficiently short. It is important to use an integration method that restarts at each new step of the input signal. The sampling period is normally short and a low order explicit onestep method gives the most efficient simulation.

6. Final Remarks

Solving ordinary differential equations is probably one of the most complicated numerical tasks that people expect to solve with a "black box" code. There has been dramatical improvement in both integration methods and simulation environments during the last decades, but the user still needs to be aware of potential problems. By formulating the simulation problem in an inappropriate way, choosing an integration method poorly suited for the problem class, or setting the method parameters unwisely, the user may end up with an inefficient simulation and/or an erroneous result. Numerical simulation is a powerful tool, but requires an educated user.

From the comments above it may seem as if successful numerical simulation is an impossible task for general ODEs. On the contrary! Many of today's commercial tools will perform perfectly on large classes of problems. Sometimes, however, they will fail. We have tried to describe some common reasons for such failures, and also suggested what the user may do to improve the situation. Often the changes needed are within the capability of the simulation environment that one uses. It just takes some thinking to formulate the problem on the right form, or some reading in the manual to find out how to change the appropriate parameters.

Finally,

- Never trust numerical simulation blindly. Does the result agree with what can be expected? Repeat the simulation with different integration methods and different error tolerances to check that you get qualitatively similar results.
- Be careful in the modeling of the system that is to be simulated. Try to express the model on a form that suites the integration method, e.g. scale the problem, try to express discontinuities so that the integration method can handle them.
- The research field in numerical integration has seen strong progress the last two decades. Many commercial simulation packages, however, still choose to use almost historical integration methods. Push vendors to implement and use state of the art algorithms.

7. References

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

[2] K. Burrage, J. C. Butcher, and F. H. Chipman. "An implementation of singly-implicit Runge-Kutta methods." *BIT (Nordisk Tidskrift för Informationsbehandling)*, **20**, pp. 326–340, 1980.

[3] J. C. Butcher. *The Numerical Analysis of Ordinary Differential Equations.* John Wiley & Sons, 1987.

[4] W. H. Enright, T. E. Hull, and B. Lindberg. "Comparing numerical methods for stiff systems of ODE:s." *BIT (Nordisk Tidskrift för Informationsbehandling)*, **15**, pp. 10–48, 1975.

[5] C. W. Gear. *Numerical Initial Value Problems in Ordinary Differential Equations.* Prentice-Hall, Englewood Cliffs, New Jersey, 1971.

[6] K. Gustafsson. *Control of Error and Convergence in ODE Solvers.* PhD thesis, Department of Automatic Control, Lund Institute of Technology, 1992.

[7] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I – Nonstiff Problems*, volume 8 of *Springer Series in Computational Mathematics.* Springer-Verlag, 1987.

[8] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II – Stiff and Differential-Algebraic Problems*, volume 14 of *Springer Series in Computational Mathematics.* Springer-Verlag, 1991.

[9] A. C. Hindmarsh. "ODEPACK, a systematized collection of ODE solvers." In Stepleman, Ed., *Scientific Computing*, pp. 55–64. North-Holland, Amsterdam, 1983.

[10] Integrated Systems, Inc., Santa Clara, California. *SYSTEM-BUILD User's Guide*, 1985.

[11] P. E. Kloeden and E. Platen. *The Numerical Solution of Stochastic Differential Equations.* Springer, 1992.

[12] The MathWorks Inc., Cochituate Place, 24 Prime Park Way, Natick, MA 01760, USA. *SIMULINK – A Program for Simulating Dynamic Systems*, 1992.

[13] MicroSim Corp, Campbell, California. *PSpice User's Manual*, 1987.

[14] L. F. Shampine and I. G. R. W. Brankin. "Reliable solution of special event location problems for odes." *ACM Transactions on Mathematical Software*, **17:1**, pp. 11–25, March 1991.

[15] R. D. Skeel. "Thirteen ways to estimate global error." *Numerische Mathematik*, 48, pp. 1–20, 1986.