Numerical methods to solve Ordinary Differential Equations

Didier Gonze

December 3, 2009

I have come to believe that ones knowledge of any dynamical system is deficient unless one knows a valid way to numerically simulate that system on a computer.

Daniel T. Gillespie

Source: Pahle J. *Brief Bioinform* 10:53-64 (2009)

Introduction

Differential equations can describe nearly all systems undergoing change. They are widespread in physics, engineering, economics, social science, but also in biology. Many mathematicians have studied the nature of these equations and many complicated systems can be described quite precisely with compact mathematical expressions. However, many systems involving differential equations are so complex, or the systems that they describe are so large, that a purely mathematical analysis is not possible. It is in these complex systems where computer simulations and numerical approximations are useful.

The techniques for solving differential equations based on numerical approximations were developed long before programmable computers existed. It was common to see equations solved in rooms of people working on mechanical calculators. As computers have increased in speed and decreased in cost, increasingly complex systems of differential equations can be solved on a common PC. Currently, your laptop could compute the long term trajectories of about 1 million interacting molecules with relative ease, a problem that was inaccessible to the fastest supercomputers just 5 or 10 years ago.

1768	Leonhard Euler publishes his method.
1824	Augustin Louis Cauchy proves convergence of the Euler method. In this proof, Cauchy uses the implicit Euler method.
1895	Carl Runge publishes the first Runge-Kutta method.
1905	Martin Kutta describes the popular fourth-order Runge-Kutta method.
1952	C.F. Curtiss and J.O. Hirschfelder coin the term stiff equations.

Source: Wikipedia (http://en.wikipedia.org/wiki/Numerical_ordinary_differential_equations)

Principle

Most often, systems of differential equations can not be solved analytically. Algorithms based on numerical methods are therefore needed. By "*numerical integration*", we mean to compute, from an initial point x_0 (the initial condition), each successive point x_1, x_2, x_3,.. that satisfy evolution equation $\dfrac{dx}{dt} = f(t, x)$. An algorithm is thus a program that computes as precisely as possible x_{n+1} from x_n. Of course, x can be a vector and the evolution equations can be non-linear coupled differential equations.

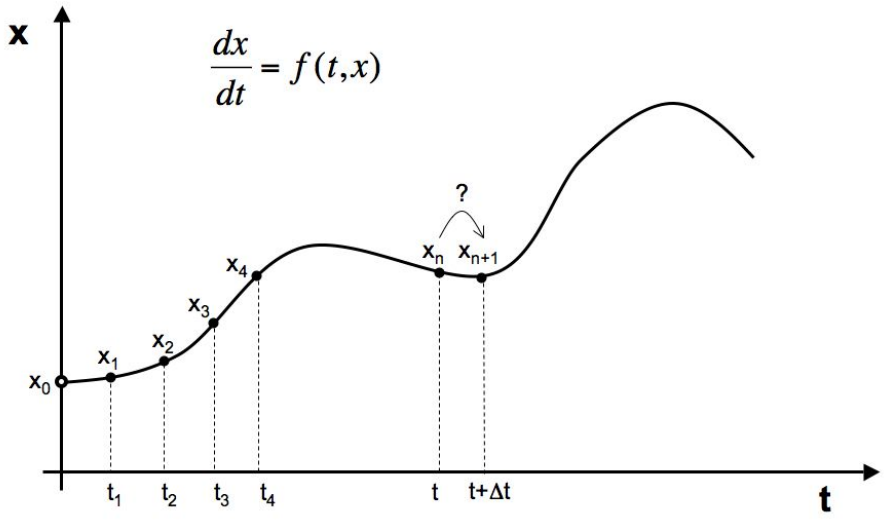

Figure 1: Principle of the numerical integration.

We present here the principle, application, and limitation of some basic numerical methods. These methods include Euler, Heun, and Runge & Kutta. More complex methods, required for stiff systems, are briefly discussed.

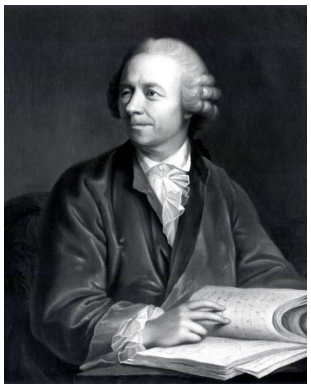

Figure 2: Leonhard Euler (1707-1783).

Euler algorithm

Principle

Let's consider the following differential equation:

$$\frac{dx}{dt} = f(t, x) \tag{1}$$

The formula for the Euler method is:

$$x_{n+1} = x_n + \Delta t.f(t_n, x_n) \tag{2}$$

Given a time step Δt (to be specified by the user), the formula thus directly calculates the point x_{n+1} from the x_n by computing the derivative at x_n. The principle is illustrated in Fig. 3.

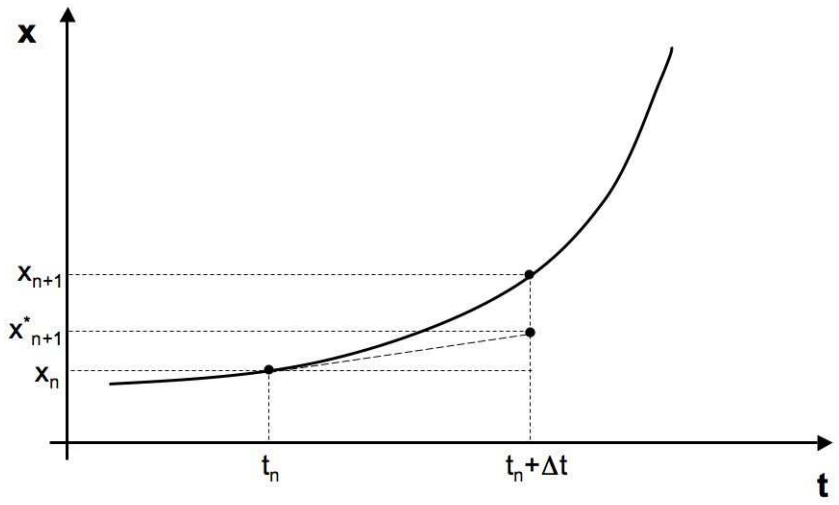

Figure 3: Principle of the Euler method.

Euler's method is not recommended for practical use because it is not very accurate compared to other methods that run at the equivalent step size. Indeed, as illustrated in Fig. 3, the imprecision comes from the fact that the derivative between t_n and $t_n + \Delta t$ changes while Euler formula relies only on the derivative at time t_n. Smaller the step size, smaller the error.

Example 1

We illustrate here some results (and errors) obtained when applying the Euler method.

Let's consider the simple differential equation:

$$\frac{dx}{dt} = x \tag{3}$$

with the initial condition $x(0) = 1$.

Figure 4 gives the time points obtained by simulating eq. (3) using the Euler formula (2) with time step $\Delta t = 0.2$. The blue curve gives the data points computed, the black curve is the exact solution ($x = e^t$). The red bar indicates the error at time $t = 2$.

The error depends on the time step Δt. Table 1 gives the computed point, the exact solution and the error calculated at time $t = 2$ for different time step.

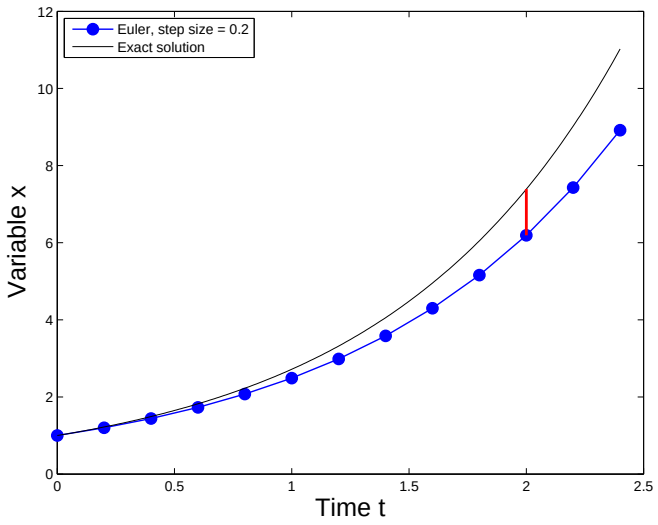

Figure 4: Euler method in application (example 1).

Time step	Computed point	Exact point	Error (%)	CPU time (s)
$\Delta t = 0.5$	5.062500	7.389056	31.48	<0.01
$\Delta t = 0.2$	6.191736	7.389056	16.20	<0.01
$\Delta t = 0.1$	6.727500	7.389056	8.953	<0.01
$\Delta t = 0.02$	7.039988	7.389056	4.724	<0.01
$\Delta t = 0.05$	7.244646	7.389056	1.954	<0.01
$\Delta t = 0.01$	7.316018	7.389056	0.988	<0.01
$\Delta t = 0.001$	7.381676	7.389056	0.100	0.15
$\Delta t = 0.0001$	7.388317	7.389056	0.010	12.2

Table 1: Error obtained with the Euler method.

Example 2

Now, consider the simple differential equation:

$$\frac{dx}{dt} = -x \tag{4}$$

with the initial condition $x(0) = 1$.

Figure 5 gives the time points obtained by simulating eq. (4) using the Euler formula (2) with time step $\Delta t = 1.5$. The blue curve gives the data points computed, the black curve is the exact solution ($x = e^{-t}$). As we can see in this example, the error is not only quantitative but also qualitative: numerical integration produces artefactual oscillations

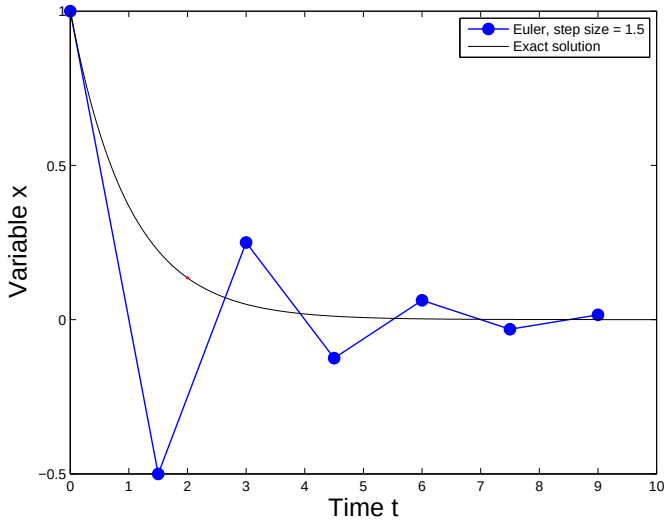

Figure 5: Euler method in application (example 2).

Estimation of the error

In order to estimate the error in the approximation to the derivative in the Euler approximation, let us consider the Taylor series approximation of the function f. If we assume that we have all the data (f and it's derivatives) at $t = 0$, then the value of the function at time $t = \Delta t$ is given by

$$f(\Delta t) = f(t=0) + \Delta t \frac{df}{dt}\Big|_{t=0} + \frac{\Delta t^2}{2}\frac{d^2 f}{dt^2}\Big|_{t=0} + \frac{\Delta t^3}{6}\frac{d^3 f}{dt^3}\Big|_{t=0} + ... \tag{5}$$

Rearranging this equation yields

$$\frac{f(\Delta t) - f(t=0)}{\Delta t} = \frac{df}{dt}\Big|_{t=0} + \frac{\Delta t}{2}\frac{d^2 f}{dt^2}\Big|_{t=0} + \frac{\Delta t^2}{6}\frac{d^3 f}{dt^3}\Big|_{t=0} + ... \tag{6}$$

Since Δt is small then the series of terms on the right hand side is dominated by the term with the smallest power of Δt, i.e.

$$\frac{f(\Delta t) - f(t=0)}{\Delta t} \approx \frac{df}{dt}\Big|_{t=0} + \frac{\Delta t}{2}\frac{d^2 f}{dt^2}\Big|_{t=0} \tag{7}$$

Therefore, the Euler approximation to the derivative is off by a factor proportional to Δt. We say that the errore is of order Δt^2 and we write $O(\Delta t^2)$. The good news is that the error goes to zero as smaller and smaller time steps are taken. The bad news is that we need to take very small time steps to get good answers (cf. Summary by Storey).

We call the error in the approximation to the derivative over one time step the *local truncation error*. This error occurs over one time step and can be estimated from the Taylor series, as we have just shown (cf. Summary by Storey).

In the next section we present alternative numerical methods with leads to a better accuracy without requiring much extra work.

Backward Euler method

Equation 2 can be written as

$$f(t_n, x_n) \approx \frac{x_{n+1} - x_n}{\Delta t} \tag{8}$$

We thus compute x_{n+1} assuming that the derivative of f at point (t_n, x_n). This is equivalent to say that the derivative at point (t_{n+1}, x_{n+1}) is the same as at point (t_n, x_n):

$$f(t_{n+1}, x_{n+1}) \approx \frac{x_{n+1} - x_n}{\Delta t} \tag{9}$$

We then find

$$x_{n+1} = x_n + \Delta t f(t_{n+1}, x_{n+1}) \tag{10}$$

We can thus find x_{n+1} from x by solving the algebraic equation 10. This is the backward Euler method. The backward Euler method is said *implicit* because we do not have an explicit expression for x_{n+1}. Of course, it costs CPU time to solve this equation; this cost must be taken into consideration when one selects the method to use. The advantage of implicit methods such as 10 is that they are usually more stable for solving a stiff equation, meaning that a larger step size Δt can be used.

Heun algorithm

Principle

Instead of computing the derivative only at the starting point x_n, a better estimation is given when the slope is computed as the average between the derivative at the starting point x_n and the derivative at the end point x_{n+1}:

$$x_{n+1} = x_n + \frac{\Delta t}{2}(f(t_{n+1}, x_{n+1}) + f(t_n + x_n)) \tag{11}$$

We do not know the end point x_{n+1} (this is the point we want to calculate!), but we can estimate it using the Euler method:

$$x^*_{n+1} = x_n + \Delta t f(t_n + x_n) \tag{12}$$

and

$$t^*_{n+1} = t_n + \Delta t \tag{13}$$

The principle of the Heun algorithm is illustrated in Fig. 6. The grey dot represent the estimate x^*_{n+1} obtained using the Euler method.

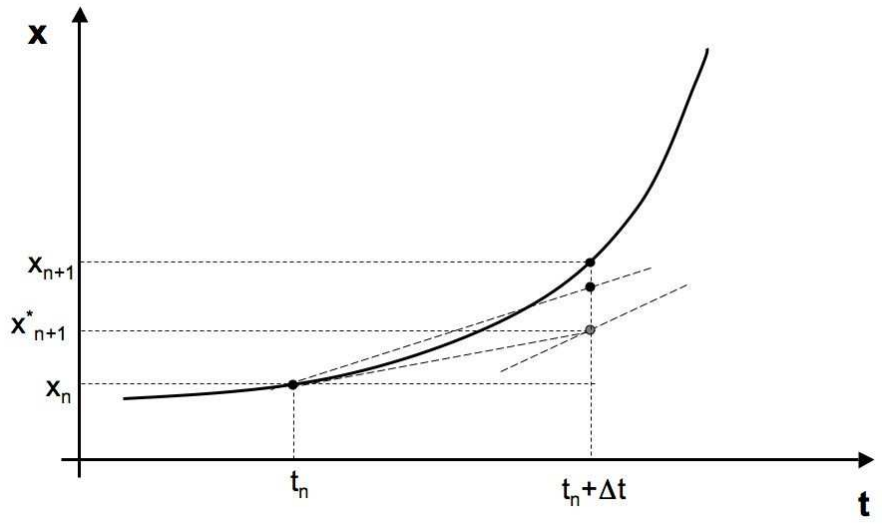

Figure 6: Principle of the Heun method.

Because this algorithm can be seen as an improvement of the Euler method, it is sometimes refered to as an *improved Euler algorithm*. This is an example of *predictor-corrector* algorithm.

Example

We illustrate here some results (and errors) obtained when applying the Heun method.

As for the Euler method, we will consider the simple differential equation:

$$\frac{dx}{dt} = x \tag{14}$$

with the initial condition $x(0) = 1$.

Figure 7 gives the time points obtained by simulating eq. (17) using the Heun formulas (11) and (12) with time step $\Delta t = 0.2$. The blue curve gives the data points computed, the black curve is the exact solution. We already see that the error is much reduced compare to the error obtained with the Euler algorithm.

Here also, the error depends on the time step Δt. Table 2 gives the computed point, the exact solution and the error calculated at time $t = 2$ for different time step:

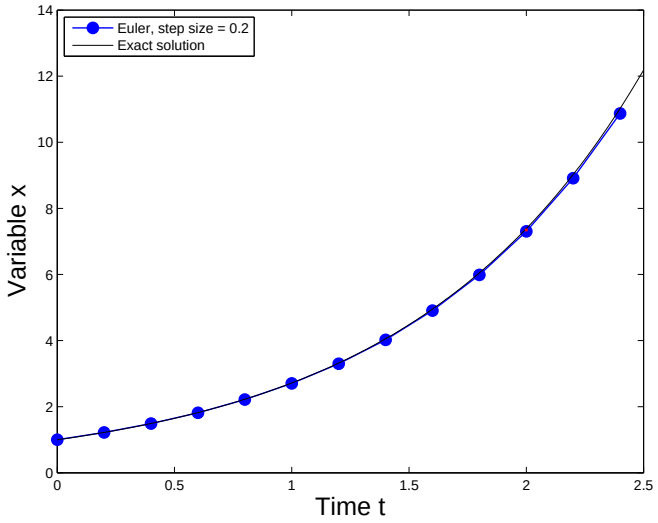

Figure 7: Heun method in application.

Time step	Computed point	Exact point	Error (%)	CPU time (s)
$\Delta t = 0.5$	6.972900	7.389056	5.63	<0.01
$\Delta t = 0.2$	7.304631	7.389056	1.14	<0.01
$\Delta t = 0.1$	7.366235	7.389056	0.309	<0.01
$\Delta t = 0.02$	7.388086	7.389056	0.131	<0.01
$\Delta t = 0.05$	7.383127	7.389056	0.0802	<0.01
$\Delta t = 0.01$	7.388812	7.389056	0.0331	<0.01
$\Delta t = 0.001$	7.389054	7.389056	0.00003	0.13
$\Delta t = 0.0001$	7.389056	7.389056	E-7	12.0

Table 2: Error obtained with the Heun method.

Runge & Kutta algorithms

Second-order Runge & Kutta

Another estimation of the slope is the derivative at the mid-point between t and $t + \Delta t$.

Consider the use of step like the one defined by the Euler formula (2) to take a "trial" step to the mid-point of the interval. Then we can use the value of both t and x at that mid-point to compute the real step across the whole interval. Fig. 8 illustrates this idea. The equations are then:

$$\begin{aligned} k_1 &= \Delta t.f(t_n, x_n) \\ k_2 &= \Delta t.f\left(t_n + \frac{\Delta t}{2}, x_n + \frac{k_1}{2}\right) \\ x_{n+1} &= x_n + k_2 + 0(\Delta t^3) \end{aligned} \tag{15}$$

This method is called the *second-order Runge & Kutta* algorithm or the *mid-point* method.

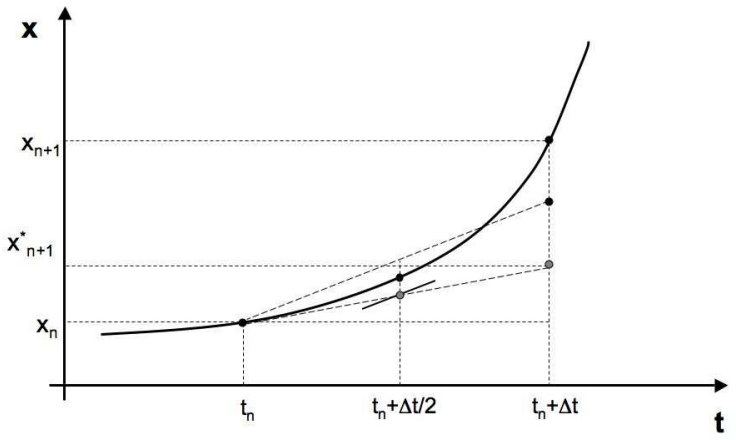

Figure 8: Principle of the second-order Runge & Kutta method

Fourth-order Runge & Kutta

More often used is the classical *four-order Runge & Kutta* algorithm:

$$\begin{aligned} k_1 &= \Delta t.f(t_n, x_n) \\ k_2 &= \Delta t.f\left(t_n + \frac{\Delta t}{2}, x_n + \frac{k_1}{2}\right) \\ k_3 &= \Delta t.f\left(t_n + \frac{\Delta t}{2}, x_n + \frac{k_2}{2}\right) \\ k_4 &= \Delta t.f\left(t_n + \Delta t, x_n + k_3\right) \\ x_{n+1} &= x_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + 0(\Delta t^3) \end{aligned} \tag{16}$$

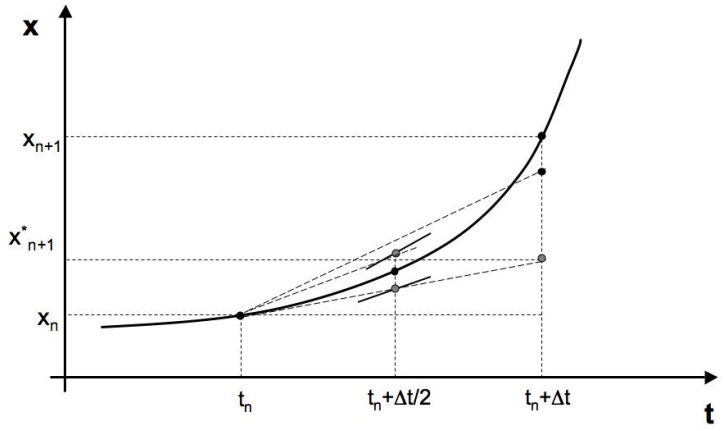

Figure 9: Principle of the fourth-order Runge & Kutta

Adaptative step size

As with the previous methods the error depends on the step size. Small step size leads to a better precision, but require larger CPU time. *Adaptative step size* methods allows to change the step size during the integration.

Example

We illustrate here some results (and errors) obtained when applying the Runge-Kutta method (order 2 vs order 4).

As for the previous methods, we will consider the simple differential equation:

$$\frac{dx}{dt} = x \tag{17}$$

with the initial condition $x(0) = 1$.

Here also, the error depends on the time step Δt. Table 3 gives the computed points (obtained using the RK2 or RK4 method), the exact solution, and the error calculated at time $t = 2$ for different time step:

Time step	Exact point	Computed point (RK2)	Error (%) (RK2)	Computed point (RK4)	Error (%) (RK4)
$\Delta t = 0.5$	7.389056	6.972900	0.056320	7.383970	6.8828E-04
$\Delta t = 0.2$	7.389056	7.304631	0.011426	7.388889	2.2581E-05
$\Delta t = 0.1$	7.389056	7.366235	0.003088	7.389045	1.5335E-06
$\Delta t = 0.05$	7.389056	7.383127	0.00080238	7.389055	9.9918E-08
$\Delta t = 0.02$	7.389056	7.388086	0.00013134	7.389056	2.6226E-09
$\Delta t = 0.01$	7.389056	7.388812	3.3083E-05	7.389056	1.6528E-10
$\Delta t = 0.001$	7.389056	7.389054	3.3308E-07	7.389056	1.5385E-14
$\Delta t = 0.0001$	7.389056	7.389056	3.3331E-09	7.389056	5.1686E-15

Table 3: Error obtained with the Runge-Kutta method.

Other methods (for "stiff" systems)

Stiff systems of ordinary differential equations represent a "special case" of the systems. There is no universally accepted definition of stiffness. Rather than proposing a precise mathematical definition of "stiffness", we will consider here that "stiff" systems are systems in which "abrupt transition" (discontinuity) are observed in the time series, as shown in Fig. 10. Such abrupt transitions often result from different time scales in the dynamics.

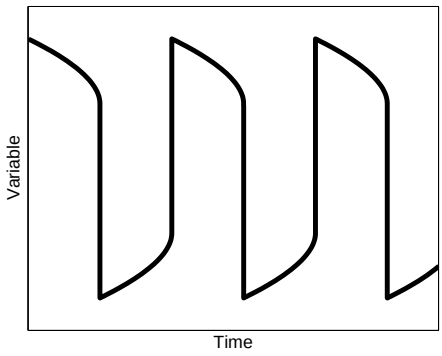

Figure 10: Example of a stiff system

To simulate precisely the abrupt transition, it is required to have a very small time steps, but reducing the time step may drastically increases the computational time. Often, even the adaptative time steps programs mentionned above are not appropriate. Therefore, algorithms specifically designed to solve such stiff systems have been proposed. GEAR is an example of such algorithm (Gear, 1969). The GEAR method is an auto-adaptative implicit algorithm which can select time step and change order automatically.

In practice...

How to know if my solution is correct?

One of the big difficulties in using numerical methods is that takes very little time to get an answer, it takes much longer to decide if it is right. Usually the first test is to check that the system is behaving physically/biologically. Usually before running simulations it is best to use physics/biological arguments to try and understand qualitatively what you think your system will do. Will it oscillate, will it grow, will it decay?

We already encountered unrealistic behavior using Euler's method (see fig. 5). These artifacts often occur when the time step is too large. One simple test of a numerical method is to change the time step Δt and see what happens. If you have implemented the method correctly (and its a good method) the answer should converge as the time step is decreased. If you know the order of your approximation then you know how fast this decrease should happen. If the method has an error proportional to Δt then you know that cutting the time step in half should cut the error in half. You should nevertheless keep in mind that it is not because the solution converges that it is correct (cf. Summary by Storey).

Softwares

There are many mathematical softwares that have built-in functions to perform numerical integration of ODE.

Matlab has several ODE solver and provides some recommendations:

Method	Problem type	Accuracy	When to use
ode45	Nonstiff	Medium	Most of the time, should be the first solver to try.
ode23	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.
ode113	Nonstiff	Low/high	For problems with stringent error tolerances or for solving computationally intensive problems.
ode15s	Stiff	Low/medium	If ode45 is slow because the problem is stiff.
ode23s	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.

XXP-AUTO, a free-ware software, specially designed to solve ODE has also several method implemented (Euler, BackEul, Heun, RK, Gear, Stiff,...). By default it uses a adaptative step size RK method.

References

Books

- Numerical recipes in C: The art of scientific computing (1992)
- Cushing (2004) Differential equations: an applied approach, Pearson Prentice Hall

On-line courses

- Storey BD, "Numerical Methods for Ordinary Differential Equations" (http://icb.olin.edu/fall_02/ec/reading/DiffEq.pdf)

Web sites

- http://www.nrbook.com/a/bookcpdf.php
- http://en.wikipedia.org/wiki/Numerical_ordinary_differential_equations
- http://en.wikipedia.org/wiki/Stiff_equation
- http://en.wikipedia.org/wiki/Runge-Kutta_method