A software package for the numerical integration of ODE by means of high-order Taylor methods

Àngel Jorba(1) and Maorong Zou(2)

14th November 2001

(1) Departament de Matemàtica Aplicada i Anàlisi, Universitat de Barcelona,
Gran Via 585, 08007 Barcelona, Spain. E-mail: `angel@maia.ub.es`

(2) Department of Mathematics, The University of Texas at Austin,
Austin, TX 78712-1082, USA. E-mail: `mzou@math.utexas.edu`

To the memory of William F. Schelter

Abstract

This paper revisits the Taylor method for the numerical integration of initial value problems of Ordinary Differential Equations (ODEs). The main issue is to present a computer program that, given a set of ODEs, produces the corresponding Taylor numerical integrator. The step size control adaptively selects both order and step size to achieve a prescribed error, and trying to minimize the global number of operations. The package provides support for several extended precision arithmetics, including user-defined types.

The paper discusses the performance of the resulting integrator in some examples. As it can select the order of the approximation used, it has a very good behaviour for high accuracy computations. In fact, if we are interested in a very accurate computation in extended precision arithmetic, it becomes the best choice by far. The main drawback is that the Taylor method is an explicit method, so it has all the limitations of these kind of schemes. For instance, it is not suitable for stiff systems.

Contents

1 Introduction

Let us consider the following initial value problem: find a smooth function $x : [a, b] \to \mathbb{R}^m$ such that

$$\left\{ \begin{array}{rcl} x'(t) & = & f(t, x(t)), \\ x(a) & = & x_0, \end{array} \right. \tag{1}$$

where $f : [a, b] \times \Omega \subset \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$ is a smooth function, $\Omega = \overset{\circ}{\Omega}$ and $m \geq 1$. There is a classical result of the theory of ODE that ensures the existence and uniqueness of a function $x(t)$ satisfying (1). However, the effective computation of such a function $x(t)$ is a much more difficult question.

The search of good numerical methods for (1) is one of the classical problems in numerical analysis. The usual procedures for solving numerically this problem are based in approximating the values $x(t)$, for a suitable mesh of values of t. For the moment, and to simplify the presentation, we will use an equispaced mesh: if $N \in \mathbb{N}$, then we define

$$h = \frac{b - a}{N}, \quad t_n = a + nh, \quad 0 \leq n \leq N.$$

So, the problem is to find approximations to the values $x(t_n)$. In what follows, we will denote these approximated values by x_n.

In this paper we will revisit one of the oldest numerical procedures for this problem: the Taylor method. For simplicity, we will assume that the function f is analytic for $(t, x) \in [a, b] \times \Omega$. The idea of the method is very simple: given the initial condition $x(t_n) = x_n$, the value $x(t_{n+1})$ is approximated from the Taylor series of $x(t)$ at $t = t_n$. The algorithm is then,

$$\begin{array}{rcl} x_0 & = & x(a), \\ x_{n+1} & = & x_n + x'(t_n)h + \dfrac{x''(t_n)}{2!}h^2 + \cdots + \dfrac{x^{(p)}(t_n)}{p!}h^p, \quad n = 0, \dots, N-1. \end{array} \tag{2}$$

We refer to [But87] for a discussion of the basic properties of the method. One of the key points for a practical implementation is the effective computation of the values of the derivatives $x^{(j)}(t_n)$. A first procedure to obtain them is to differentiate the equation in (1) w.r.t. t, at the point $t = t_n$. Hence,

$$x'(t_n) = f(t_n, x(t_n)), \quad x''(t_n) = f_t(t_n, x(t_n)) + f_x(t_n, x(t_n))x'(t_n),$$

and so on. Hence, the first step to apply this method is, for a given f, to compute these derivatives up to a suitable order. Then, for each step of the integration (see (2)), we have to evaluate these expressions to obtain the coefficients of the power series of $x(t)$ at $t = t_n$. Usually, these expressions will be very cumbersome, so it will take a significant amount of time to evaluate them numerically. This, jointly with the initial effort to compute those derivatives, is the main drawback of the Taylor method. As an example to illustrate these ideas, let us consider the Van der Pol equation:

$$\left. \begin{array}{rcl} x' & = & y \\ y' & = & (1 - x^2)y - x \end{array} \right\}. \tag{3}$$

The second and third order derivatives of x and y with respect to time are

$$\begin{aligned}
x'' &= (1-x^2)y - x\\
y'' &= x^3 - x - 2xy^2 + (x^4-2x^2)y\\
x''' &= x^3 - x - 2xy^2 + (x^4-2x^2)y\\
y''' &= 2x^3 - x^5 + (-1+5x^2+3x^4-x^6)y + (-8x+4x^3)y^2 - 2y^3
\end{aligned} \tag{4}$$

Hence a third order Taylor method for the initial value problem (3) is

$$\begin{aligned}
\begin{pmatrix} x_{n+1}\\ y_{n+1}\end{pmatrix} &= \begin{pmatrix} x_n\\ y_n\end{pmatrix} + \begin{pmatrix} y_n\\ (1-x_n^2)y_n - x_n\end{pmatrix} h\\
&+\frac{1}{2!}\begin{pmatrix} (1-x_n^2)y_n - x_n\\ x_n^3 - x_n - 2x_ny_n^2 + (x_n^4-2x_n^2)y_n\end{pmatrix} h^2\\
&+\frac{1}{3!}\begin{pmatrix} x_n^3 - x_n - 2x_ny_n^2 + (x_n^4-2x_n^2)y_n\\ 2x_n^3 - x_n^5 + (-1+5x_n^2+3x_n^4-x_n^6)y_n + (-8x_n+4x_n^3)y_n^2 - 2y_n^3\end{pmatrix} h^3
\end{aligned}$$

As one can see from these equations, expressions for higher order derivatives are quite complicated, and the complexity increases dramatically as the order increases.

This difficulty can be overcomed by using the so-called *automatic differentiation* ([BKSF59], [Wen64], [Moo66], [Ral81], [GC91], [BCCG92], [BBCG96], [Gri00]). This is a procedure that allows for a fast computation of the derivatives of a given function, up to arbitrarily high orders. As far as we know, these ideas were first used in Celestial Mechanics problems ([Ste56], [Ste57]; see also [Bro71]).

A first inconvenience of this method is that the function f has to belong to a special class; fortunately, this class is large enough to contain the functions that appear in most applications. A second difficulty is that the program to compute these derivatives has to be specifically coded for each function f.

The goal of this work is to present a software that, given a function f (belonging to a suitable class), generates the code that computes the jet of derivatives needed by the Taylor method up to any order.

For an efficient numerical integration, we need some knowledge of the order $p \in \mathbb{N}$ up to which the derivatives have to be computed, and an estimate of the step size h, in order to have a truncation error below a given threshold value ε. We note that, as we have to select the value of two parameters (p and h), we can ask for a second condition besides the size of the truncation error. Here we have chosen to minimize the number of operations needed to advance the independent variable t in one unit. We have also coded the algorithms to do these tasks so that the output of the program is, in fact, a numerical integrator for the initial value problem (1).

We have tested this Taylor integrator against some well-known integration methods. The results show that Taylor method is a very competitive method to integrate with the standard double precision arithmetic of the computer. However, the main motivation for writing this software is to address the need of highly accurate computations in some problems of Dynamical Systems and Mechanics (see, for instance, [MS99], [SV01] and [Sim]). The standard methods of the numerical integration of ODEs become extremely slow for more accurate computations that require extended precision arithmetic. Here we show that, if we need extended precision for our calculations, Taylor method becomes the best choice by far. The reason for this advantage is, basically, that Taylor method does not need to reduce the step size to increase accuracy; it can simply increase the order (see Section 3.4). As we will see, this allows to greatly reduce the number of arithmetic operations during the numerical integration.

As any explicit scheme, the Taylor method is not suitable for stiff equations because, in this case, the errors grow too fast. However, there are modifications of the Taylor method to deal with these situations ([Bar80], [JZ85], [KC92] and [CGH$^+$97]). These modifications have not been considered in our software.

In the paper we present the main details of our implementation. We have tried to produce an efficient package, in the sense that the produced Taylor integrator be as fast as possible. Moreover, we have also included support for quadruple precision and multiple precision arithmetic. We have done several test to compare the efficiency and accuracy of the generated Taylor routine against other numerical integrators.

There are several papers that focus on computer implementations of the Taylor method in different contexts; see, for instance, [BWZ70], [CC82], [CC94]. A good survey is [NJC99] (see also [Cor95]).

This package has been released under the GNU Public License, so anybody with Internet access is free to get it and to redistribute it. To obtain a copy, visit the URLs `http://www.ma.utexas.edu/~mzou/taylor/` or `http://www.maia.ub.es/~angel/`. We note that the actual version of the package is written to run under the GNU/Linux operating system. We do not expect major problems to run it under any version of Unix, but we do not plan to write ports for other operating systems.

The paper has been split as follows: Section 2 contains a survey about automatic differentiation, Section 3 is devoted to the selection of step size and truncation degree, Section 4 gives some details about the software and Section 5 provides some examples of the use of the package.

2 A short summary on automatic differentiation

Automatic differentiation is a recursive procedure to compute the value of the derivatives of certain functions at a given point (see [Moo66, Ral81]). The considered functions are those that can be obtained by sum, product, quotient, and composition of elementary functions.[1] We will see this in detail.

To simplify the discussion let us introduce the following notation: if $a : t \in I \subset \mathbb{R} \mapsto \mathbb{R}$ denotes a smooth function, we call its normalized j-th derivative to the value

$$a^{[j]}(t) = \frac{1}{j!} a^{(j)}(t). \tag{5}$$

where $a^{(j)}(t_0)$ denotes the j derivative w.r.t. t. In what follows, we will focus on the computation of the values $a^{[j]}(t)$.

---

[1] Elementary functions include polynomials, trigonometric functions, real powers, exponentials and logarithms.

Assume now that $a(t) = F(b(t), c(t))$ and that we know the values $b^{[j]}(t)$ and $c^{[j]}(t)$, $j = 0, \dots, n$, for a given t. The next proposition gives the n-th derivative of a at t for some functions F.

Proposition 2.1 *If the functions b and c are of class C^n, and $\alpha \in \mathbb{R} \setminus \{0\}$, we have*

1. *If $a(t) = b(t) \pm c(t)$, then $a^{[n]}(t) = b^{[n]}(t) \pm c^{[n]}(t)$.*
2. *If $a(t) = b(t)c(t)$, then* $a^{[n]}(t) = \displaystyle\sum_{i=0}^{n} b^{[n-i]}(t)c^{[i]}(t)$.
3. *If* $a(t) = \dfrac{b(t)}{c(t)}$, *then* $a^{[n]}(t) = \dfrac{1}{c^{[0]}(t)}\left[b^{[n]}(t) - \displaystyle\sum_{i=1}^{n} c^{[i]}(t)a^{[n-i]}(t)\right]$.
4. *If $a(t) = b(t)^\alpha$, then* $a^{[n]}(t) = \dfrac{1}{nb^{[0]}(t)}\displaystyle\sum_{i=0}^{n-1} (n\alpha - i(\alpha+1))\, b^{[n-i]}(t)a^{[i]}(t)$.
5. *If $a(t) = e^{b(t)}$, then* $a^{[n]}(t) = \dfrac{1}{n}\displaystyle\sum_{i=0}^{n-1} (n-i)\, a^{[i]}(t)b^{[n-i]}(t)$.
6. *If $a(t) = \ln b(t)$, then* $a^{[n]}(t) = \dfrac{1}{b^{[0]}(t)}\left[b^{[n]}(t) - \dfrac{1}{n}\displaystyle\sum_{i=1}^{n-1}(n-i)b^{[i]}(t)a^{[n-i]}(t)\right]$.
7. *If $a(t) = \cos c(t)$ and $b(t) = \sin c(t)$, then*

$$\begin{aligned} a^{[n]}(t) &= -\frac{1}{n}\sum_{i=1}^{n} i b^{[n-i]}(t)c^{[i]}(t) \\ b^{[n]}(t) &= \frac{1}{n}\sum_{i=1}^{n} i a^{[n-i]}(t)c^{[i]}(t) \end{aligned}$$

Proof: These proofs can be found in the literature, so we only give some hints about them.

1. Obvious.
2. It follows from Leibniz's formula:

$$a^{[n]}(t) = \frac{1}{n!}a^{(n)}(t) = \frac{1}{n!}\sum_{i=0}^{n}\binom{n}{i}b^{(n-i)}(t)c^{(i)}(t) = \sum_{i=0}^{n} b^{[n-i]}(t)c^{[i]}(t).$$

3. Apply item 2 to $a(t)c(t) = b(t)$.
4. Take logarithms and derivatives to obtain $a'(t)b(t) = \alpha a(t)b'(t)$. Use item 2 and (5).
5. Take logarithms and derivatives to obtain $a'(t) = a(t)b'(t)$. Use item 2 and (5).
6. Take derivatives to obtain $a'(t)b(t) = b'(t)$. Use item 2 and (5).
7. Take derivatives to obtain $a'(t) = -b(t)c'(t)$ and $b'(t) = a(t)c'(t)$. Use item 2 and (5).

■

Remark 2.1 *It is possible to derive similar formulas for other functions, like inverse trigonometric functions.*

Corollary 2.1 *The number of arithmetic operations to compute the derivatives up to order* n *of a vector field is of* $O(n^2)$.

Proof: The number of operations to obtain the j-th derivative once we know the previous ones is $O(j)$ (see the previous formulas). Hence, the total number of operations is $\sum_{j=1}^{n} O(j) = O(n^2)$. ■

Although these methods only allow for the derivation of a reduced set of functions, we note that the functions that can be coded in a computer belong to this set.

2.1 An example: The Van der Pol equation

These rules can be applied recursively so that we can obtain recursive formulas for the derivatives of a function described by combination of these basic functions. As an example, we can apply them to the Van der Pol equation (3). To this end we decompose the right-hand side of these equations in a sequence of simple operations:

$$\left.\begin{array}{rcl} u_1 &=& x \\ u_2 &=& y \\ u_3 &=& u_1 u_1 \\ u_4 &=& 1 - u_3 \\ u_5 &=& u_4 u_2 \\ u_6 &=& u_5 - u_1 \\ x' &=& u_2 \\ y' &=& u_6 \end{array}\right\} \tag{6}$$

Then, we can apply the formulas given in Proposition 2.1 (items 1 and 2) to each of the equations in (6) to derive recursive formulas for $u_j^{[n]}$, $j = 1, \ldots, 6$. Then, from the last two equations of (6) we have

$$\begin{array}{rcl} x^{[n+1]} &=& \dfrac{1}{n+1} u_2^{[n]}, \\ y^{[n+1]} &=& \dfrac{1}{n+1} u_6^{[n]}. \end{array}$$

The factor $\frac{1}{n+1}$ comes from the definition given in equation (5). Then, we can apply recursively these formulas for $n = 0, 1, \ldots$, up to a suitable degree p, to obtain the jet of normalized derivatives for the solution of the ODE. Note that is not necessary to select the value p in advance. Compare these formulas with the result of using formal derivatives on the right hand side of (3), as it has been done in the Introduction.

The main task of the software we present is to read read the system of ODEs, to decompose it in a sequence of basic operations, and to apply the formulas in Proposition 2.1 to this decomposition. The result is an ANSI C routine that, given an initial condition and a final degree, computes normalized derivatives of the solution up to that degree.

3 Degree and step size control

Let us now focus on a single step of the integration procedure for equation (1). In what follows, to simplify the discussion we will assume that the step size is always positive (a similar discussion also holds for negative step sizes).

Assume that, for a given time t_n, the solution is at the point x_n, and that we want to compute the position of the trajectory for a new time $t_{n+1} = t_n + h_n$ within a given error ε. We will not assume that t_{n+1} is fixed in advance, so we have to determine not only the degree of the Taylor expansion to be used but also the value h_n.

So, let us denote by $\{x_n^{[j]}(t_n)\}_j$ the jet of normalized derivatives of the solution x_n of (1) that satisfies $x_n(t_n) = x_n$. Then, if $h = t - t_n$ is small enough, we have

$$x_n(t) = \sum_{j=0}^{\infty} x_n^{[j]}(t_n) h^j.$$

The idea is to select a sufficiently small value h_n and a sufficiently large value p, such that the values

$$t_{n+1} \equiv t_n + h_n, \qquad x_{n+1} \equiv \sum_{j=0}^{p} x_n^{[j]}(t_n) h_n^j,$$

satisfy

$$\|x_n(t_{n+1}) - x_{n+1}\| \leq \varepsilon.$$

3.1 On the optimal step size selection

We are interested in values p and h such that a) the error is below a given value ε; and b) the total number of operations of the numerical integration is as small as possible. To determine such values, we need some assumptions on the analyticity properties of the solution $x(t)$. The following result can be found in [Sim01].

Proposition 3.1 *Assume that the function $h \mapsto x(t_n + h)$ is analytic on a ball of radius ρ_n, and that there exists positive constants M_1 and M_2 such that*

$$\frac{M_1}{\rho_n^j} \leq |x_n^{[j]}| \leq \frac{M_2}{\rho_n^j}, \qquad \forall\, n \in \mathbb{N}. \tag{7}$$

Then, if the required accuracy ε tends to 0, the optimal value of h that minimizes the number of operations tends to

$$h_n = \frac{\rho_n}{e^2}.$$

Remark 3.1 *Note that the optimal step size does not depend on the level of accuracy. The optimal order p is, in fact, the order that guarantees the required precision once the step size has been selected. If ε tends to 0, this order p behaves like*

$$p_n = -\frac{1}{2} \ln \varepsilon.$$

Remark 3.2 *The upper bound in (7) is the usual Cauchy bound on the Taylor coefficients of an analytic function. Note that this upper bound is directly related to the computational effort, because when the normalized derivatives are larger, we need more terms in the series to achieve a given accuracy. On the other hand, the lower bound in (7) is needed for the optimality of* h *w.r.t. the (total) number of operations: if the normalized derivatives could be very small, then the step size obtained using the upper bound would be far from optimal.*

Proof: Although the proof can be found in [Sim01], we will include it here for the convenience of the reader.

The error introduced when cutting a Taylor series to a given degree p is of the order of the first neglected term,

$$E \approx M \left(\frac{h}{\rho}\right)^{p+1}.$$

Hence, to obtain an error of order ε we have to select

$$h \approx \rho \left(\frac{\varepsilon}{M}\right)^{\frac{1}{p+1}}, \tag{8}$$

On the other hand, the computational effort to obtain the jet of normalized derivatives up to order p is $O(p^2) \approx c(p+1)^2$ (see Corollary 2.1). So, the (instantaneous) number of floating point operations per unit of time is given, in order of magnitude, by

$$\phi(p) = \frac{c(p+1)^2}{\rho \left(\frac{\varepsilon}{M}\right)^{\frac{1}{p+1}}},$$

and, solving $\phi'(p) = 0$, we obtain

$$p = -\frac{1}{2} \ln \left(\frac{\varepsilon}{M}\right) - 1.$$

So, we will use this order with the largest step size that keeps the local error below ε: inserting this value of p in (8) we have

$$h = \frac{\rho}{e^2}.$$

Finally, note that the optimal order p behaves like $-\frac{1}{2}\ln\varepsilon$ when ε goes to zero. ■

There are strategies to use step sizes that are larger than the radius of convergence of the series, but we will not discuss them here. See [CC82] for more details.

3.2 Dangerous step sizes

An important point to take into account is that, after the step size and the order have been selected, we have to sum the Taylor series to obtain the final result. Although adding a series seems a trivial numerical operation, there are cases in which this is a very difficult problem. For instance, consider the initial value problem

$$\dot{x} = -x, \qquad x(0) = 1, \tag{9}$$

and assume we are interested in computing $x(10)$ by numerical integration of the ODE (as this problem can be easily solved by hand, we already know that the solution is given by $x(t) = \exp(-t)$, so $x(10) = \exp(-10) \approx 0.0000454$). Using (9), the Taylor series at $x(0)$ is very simple to obtain:

$$x(h) = 1 + \sum_{n\geq 1} (-1)^n \frac{h^n}{n!}.$$

Due to the entire character of this function, the optimal step size is $h = 10$, and the degree is selected to have a truncation error smaller than a given precision. However, using such a large h introduces strong cancellations in the sum of the series, that lead to the loss of many significant digits when the sum is carried out with floating point arithmetic.

The solution for this problem is to use smaller values of h, to avoid these cancellations. The key point is to use a step size such that the series is decreasing in modulus. For instance, in this case we should start with $h = 1$. For this reason, once the step size has been selected, it is advisable to check the size of the intermediate terms of the series, and to reduce h when necessary. This will be discussed again in Section 3.3.2.

3.3 A more practical approach

The main drawback of the previous approach is that it requires information that we cannot obtain easily, like the radius of convergence of the Taylor series. Hence, in this section we will discuss some modifications to simplify the numerical implementation.

The first issue we have to address is whether we need to control the absolute or the relative error. This is a question we cannot answer in a general way, so we have provided code for both situations, and we let the user to choose the kind of control he wants. This is accomplished by asking the user for both tolerances, and the code looks for an step size that tries to satisfy both of them.

When controlling the relative error, there is a dangerous situation that we should avoid. Note that the usual definition of relative error ε_r,

$$\varepsilon_r = \frac{\varepsilon_a}{|x_n|},$$

assumes that the solution x_n is different from zero. Hence, if some of the points x_n are zero (or very small), we should avoid the use of the relative error for these points, since this could result in an undefined (or ridiculously small) step size. Therefore, we have used the following estimate for the relative error,

$$\varepsilon_r = \frac{\varepsilon_a}{\max\{|x_n|, |\dot{x}_n|\}}, \tag{10}$$

where $|.|$ denotes the sup norm of a vector, and $\dot{x}_n \equiv f(t_n, x_n)$. There is still the possibility that the origin be an equilibrium point and that our solution is approaching it. We do not deal with this case, since there is no a unique answer for all the possible situations like that. The user should use the absolute and relative controls to obtain the kind of error control that he wants (see also Section 4.2).

Finally let us note that, as all the step size controls presented are local, the values used to control the absolute and relative errors used in one part of the phase space can be meaningless in a different region. As before, the user is responsible for setting these values properly, and to change them when needed.

3.3.1 First step size control

The simplest step step size control we have implemented is based on a very simple estimation of the radius of convergence of the Taylor series from the size of the last two terms (a more sophisticated approach can be found in [CC82]). Then, the order and step size are estimated using the formulas of Proposition 3.1. To explain the details, let us define (t_n, x_n) as the actual point of the orbit, and let us call ε_a and ε_r the absolute and relative required accuracies. Moreover, we define

$$\varepsilon_n = \min\left\{ \frac{\varepsilon_a}{\max\{|x_n|, |\dot{x}_n|\}}, \varepsilon_r \right\}, \qquad p_n = -\frac{1}{2}\ln\varepsilon_n + 1. \tag{11}$$

Then, if we denote

$$\bar{\rho}_j = \left(|x_n^{[j]}|\right)^{-\frac{1}{j}}, \quad \hat{\rho}_j = \left(\frac{|x_n^{[j]}|}{\max\{|x_n|, |\dot{x}_n|\}} \right)^{-\frac{1}{j}}, \quad j = 1, \ldots, p_n,$$

we estimate the radius of convergence of the series as

$$\rho_n = \min\{\bar{\rho}_{p_n-1}, \, \bar{\rho}_{p_n} \, \hat{\rho}_{p_n-1}, \, \hat{\rho}_{p_n}\},$$

so the computed step size is

$$h_n = \pm\frac{\rho_n}{e^2}.$$

The sign "$-$" is for a backwards integration.

As these values are approximations to the asymptotics given in Proposition 3.1, we are not sure if they guarantee the required precision. For this reason, we want to note that, for $j = p_n - 1$ and $j = p_n$,

$$\begin{aligned} |x_n^{[j]} h_n^j| &\le \frac{|x_n^{[j]}|\rho_n^j}{e^{2j}} \le \frac{1}{e^{2j}}, \\ \frac{|x_n^{[j]} h_n^j|}{\max\{|x_n|, |\dot{x}_n|\}} &\le \frac{|x_n^{[j]}|\rho_n^j}{|x_n| e^{2j}} \le \frac{1}{e^{2j}}, \end{aligned}$$

and, if $j = p_n - 1$,

$$\frac{1}{e^{2j}} = \frac{1}{e^{2(p_n-1)}} \approx \frac{1}{e^{-\ln\varepsilon_n}} = \varepsilon_n.$$

This means that the terms of order $p_n - 1$ in the Taylor series have a contribution of order ε_n (either in absolute value or relative to the size of the initial condition) while the terms of order p_n (the last terms to be considered) have a contribution of order ε_n/e^2. Hence, this shows that the proposed strategy is similar to the more straightforward method of looking for an h_n such that the last terms in the series are of the order of the error wanted.

3.3.2 Second step size control

This is based in correcting the previous method to avoid too large step sizes, as it has been explained in Section 3.2. There are several possibilities to implement this approach. First, we want to note that it can be very dangerous to ask for a strictly decreasing series, since this could lead to a very drastic (and unnecessary) step reductions. A typical example of this situation occurs when the solution $x(t)$ has all the odd (or even) Taylor terms equal to zero. Hence, to avoid these situations we have used a weaker criterion: let h_0 be the step size control obtained using the methods of Section 3.3.1, and let h be the largest value such that $h \leq h_0$ and that

$$|x^{[0]}| + h_0|x^{[1]}| \geq |x^{[j]}|h^j, \qquad j = 2, \dots, p. \tag{12}$$

Note that, in many cases, it is enough to take $h = h_0$ to meet this condition. On the other hand, when dealing with entire functions, this avoids selecting too large step sizes that could lead to a dramatic loss of precision (we recall the example in Section 3.2).

3.3.3 User defined step size control

All the implemented step sizes are based on rough estimations of the error and, for this reason, they cannot be taken as “rigorous”.

One of the main uses of Taylor methods is the so-called verified integration, or integration with guaranteed error bounds. This means that it is possible to produce rigorous bounds on the truncation error so that, working with interval arithmetic, the trajectory is computed with true error bounds (see, for instance, [CC82], [CC94], [Cor95] and [Hoe01]). This is not the approach we have taken here. The proposed step size controls are more in the lines of the classical numerical analysis, in which the error is estimated by means of asymptotic formulas ([But87]). The main reason for that is to compete, both in speed and accuracy, with the classical numerical integrators that one can find in numerical libraries.

However, our implementation can also be used for different purposes. In our opinion, the main point of this package is that, given a vector field, produces an efficient C routine that computes the jet of normalized derivatives at a given initial condition (this is usually the most tedious part of building a Taylor series integrator). So, it is not difficult to use this routine with an interval arithmetic, plus a suitable step size control to integrate a given ODE with guaranteed error bounds. We also offer to the user the possibility of using his own error control algorithm. This can be very useful in some specific cases where some special properties of the solution are known (for instance, when the solution is known to be a polynomial). See the documentation that comes with the software for more details about this.

3.4 High accuracy computations

An important property of high order Taylor integrators is their suitability for high accuracy computations.

Assume that we are solving an IVP like (1) and that, at a given step, we are using an step size $h << 1$ and an order p to obtain a local error $\varepsilon << 1$. The number of operations needed to compute all the derivatives is of $O(p^2)$ (see Corollary 2.1). As the number of operations to sum the power series is only $O(p)$, the total operation count for a single step of the Taylor method is of $O(p^2)$. Hence, if we want to increase the accuracy to, say, ε^2 we can simply increase the order of the Taylor method to $2p$ so the number of operations is increased by a factor 4. Note that, if we want to achieve the same level of accuracy not by increasing the order but by reducing the step size h, we have to use an step size of h^2. This means that we will have to use h^{-1} steps (of size h^2) to compute the orbit after h units of time so the total number of operations is, roughly speaking, increased by a factor of h^{-1}. For instance, assume that the truncation error is exactly h^p. If $\varepsilon = 10^{-16}$ and $p = 8$, then the step size has to be $h = 0.01$. Note that, if p is fixed, to achieve an accuracy of 10^{-32} we have to use $h = 10^{-4}$, that forces to use 100 times more steps (hence, 100 times more operations) than for the $\varepsilon = 10^{-16}$ case. Changing the value of p from 8 to 16 allows to keep the same step size $h = 0.01$ while the computational effort required to obtain the derivatives is only increased by a factor 4. If the required precision were higher, these differences would be even more dramatic.

Hence, it requires much less work to increase the order rather than to reduce the step size (this observation was already implicit in Proposition 3.1, where it was shown that the optimal step size was independent from the level of accuracy required).

If we are using a numerical integrator of fixed order (for instance, a Runge-Kutta or Adams-Bashford method), the only possibility to increase accuracy is to reduce the step size. Hence, fixed order methods are strongly penalized for high accuracies, compared with varying order methods. For this reason, if the required accuracy is high enough, Taylor methods –with varying order– are the best option by far. This will be illustrated in Sections 5.2 and 5.3.

4 Software implementation

The basic operations of the `taylor` translator are first to parse the differential equations to reduce them to a sequence of binary operations and calls to mathematical functions, and then to apply the rules of automatic differentiation as it has been explained in Section 2.

4.1 Extended arithmetic

A point we think is worth noting is the way we deal with the basic arithmetic. When the Taylor translator writes the code for the jet of derivatives and/or the step size control, it declares all the real variables as `MY_FLOAT` (we will explain this type later on), and substitutes each mathematical operation by a suitable macro; the name of these macros is independent from the arithmetic. In other words, if we use the `taylor` program to only generate the jet of derivatives and/or the step size control, we do not need to specify the kind of arithmetic we want, since the output does not depend on that.

The definition of the type `MY_FLOAT` and the body of the macros is contained in the header file. This file is produced by the flag `-header` plus a flag specifying the arithmetic wanted. For instance, the macro for the product of two `gmp` numbers is

```
/* multiplication r=a*b */
#define   MultiplyMyFloatA(r,a,b)   mpf_mul(r,(a), (b))
```

Here, `mpf_mul` is a `gmp` function that multiplies the two numbers `a` and `b` and stores the result in `r`. Then, the C preprocessor will substitute the macros by the corresponding calls to the arithmetic library. Hence, to use an arithmetic different than the ones provided here we *only* have to modify the header file. For more details, see the manual that comes with the software.

The package includes support for several extended precision arithmetics, namely

doubledouble This is a C++ library that defines a extended float type, in which each number is stored as the sum of two `double` numbers. The accuracy is then of nearly 30 decimal digits. The standard way of using this library is by means of overloading. See `http://www.btexact.com/people/briggsk2/doubledouble.html`

dd_real, qd_real This is also a C++ library, similar to `doubledouble`, that defines the types `dd_real` (2 doubles) and `qd_real` (4 doubles), providing accuracies of nearly 32 and 64 decimal digits, respectively. For more details, visit the URL `http://www.nersc.gov/~dhbailey/mpdist/mpdist.html`

GNU Multiple Precision Library (gmp) This is the standard GNU library for extended precision. This library allows to define arbitrary long integer, rational and real types, and to operate on them by means of function calls (more details on the library can be found in `http://www.swox.com/gmp/`). Unfortunately, this library does not provide transcendental functions so, in principle, we are restricted to vector fields that can be written with the basic arithmetic functions plus square root (those are the only functions provided for floating point types). For more sophisticated vector fields, the user has to code the corresponding functions.

None of these floating point libraries is included in our package.

4.2 Step size control

We have included two implementations of the previously mentioned step size methods. Moreover, the user can also provide his own step size control.

The first step control we provide is the direct coding of the method explained in Section 3.3.1. The Taylor series is generated up to the order p_n defined in (11), and the step size is computed as explained in Section 3.3.1. Finally, the series is added using the Horner method. The second step size control is based on a modification of the previous control, as explained in Section 3.3.2. The main difference is a loop that runs over all the coefficients of the Taylor series and checks for the condition (12). When it is not satisfied, h is reduced accordingly.

We have also included a user defined step size control. In this case, the user must code its own step control. More details can be found in the documentation that comes with the software.

```
/* ODE specification: rtbp */
mu=0.01;
umu=1-mu;
r2=x1*x1+x2*x2+x3*x3;
rpe2=r2-2*mu*x1+mu*mu;
rpe3i=rpe2^(-3./2);
rpm2=r2+2*(1-mu)*x1+(1-mu)*(1-mu);
rpm3i=rpm2^(-3./2);

diff(x1, t)= x4+x2;
diff(x2, t)= x5-x1;
diff(x3, t)= x6;
diff(x4, t)= x5-(x1-mu)*(umu*rpe3i)-(x1+umu)*(mu*rpm3i);
diff(x5, t)=-x4-x2*(umu*rpe3i+mu*rpm3i);
diff(x6, t)=-x3*(umu*rpe3i+mu*rpm3i);
```

Figure 1: Input file for the restricted three-body problem.

5 Examples

In this section we have included a few examples to show how Taylor method works. We have assumed that the `taylor` translator is already installed in your computer. For details about how to do this, check the documentation that comes with the package.

5.1 The three-body problem

This is, probably, the best known problem in Celestial Mechanics. Here we will use a simplified version of it, the so-called restricted three-body problem (for details about this problem see, for instance, [Sze67]). The problem boils down to describe the dynamics of the Hamiltonian system

$$H_{RTBP} = \frac{1}{2}(p_x^2 + p_y^2 + p_z^2) + yp_x - xp_y - \frac{1-\mu}{r_{PS}} - \frac{\mu}{r_{PJ}},$$

being μ a mass parameter, $r_{PS}^2 = (x-\mu)^2 + y^2 + z^2$ and $r_{PJ}^2 = (x-\mu+1)^2 + y^2 + z^2$. An input file for the corresponding vector field is shown in Figure 1.

There are several kind of instruction in this file. First of all, anything between `/* */` is ignored by the program, so we can use them to put comments in the file. Next, we have some lines to define numerical constants, plus some operations with the variables of the system. The variables of the equation are labeled `x1`, `x2` and so on, and the independent variable is labeled `t`. Finally, the last 6 lines are the definition of the differential equations.

Although we think that the notation used is clear and self-explicative, we want to make some comments about it. First, the `taylor` translator does not make any kind of optimization on the input description of the vector field, with the exception of common expression eliminations. If one of your main concerns is the efficiency of the code generated by `taylor`, you should apply other kind of optimizations "by hand" in your input file (for instance, to simplify algebraic expressions to minimize the number of operations).

A second point we want to comment on is the use of the exponent "$-3.0/2$" in the expressions. There are several ways of introducing such an exponent. If we use the expression "-1.5", the program will use the exp and ln functions to define it (this is true for any real exponent). If we use "$-3.0/2$", then we can use the flag "`-sqrt`" of the translator to force the program to use the square root function instead of the exp and ln functions. Without this flag, the value "$-3.0/2$" is treated as "-1.5".

The input file supports more features than the ones showed here (like the use of `extern` variables to receive parameters from the user's programs); for details check the manual that comes with the package.

To produce a numerical integrator for this vector field, assume that we have the code of Figure 1 in the file `rtbp.in`. Then, you can type

```
taylor -name rtbp -o taylor_rtbp.c -step -jet -sqrt rtbp.in
taylor -name rtbp -o taylor.h -header
```

(we have assumed that the `taylor` binary is in a directory contained in your path; otherwise you should specify its location). The first line outputs the file `taylor_rtbp.c` with the code for the step size control and the jet of derivatives. The second line produces the header file; it is needed for the file `taylor_rtbp.c`, and the user may also want to include it in the calling routine, since it contains the prototype for the call to the integrator. There are more options to control the output of `taylor`, look at the documentation for more details.

Fortran 77 users can use a single instruction:

```
taylor -name rtbp -o taylor_rtbp.c -step -jet -f77 -header -sqrt rtbp.in
```

This will put everything in the file `taylor_rtbp.c`, so you can simply compile and link it with your (Fortran) calling routine. For details about how to call the Taylor integration routine, look at the documentation.

Then, you can call the routine `taylor_step_rtbp` (with suitable parameters), to perform a numerical integration of the previous vector field. As it is usual in one step explicit methods, each call advances the independent variable in some amount that depends on the level of accuracy required. The details about this call (parameters, etc.) can be found in the documentation.

For instance, let us show an example of numerical integration with this routine by selecting the initial condition `x1=-0.45`, `x2=0.80`, `x3=0.00`, `x4=-0.80`, `x5=-0.45` and `x6=0.58`. We will perform first a numerical integration with the standard double precision of the computer, for 1 unit of time. As the Hamiltonian function is a first integral of the system, we will check the preservation of its value as an indicator of the accuracy of the integration. So, we have coded a small main program that uses this initial condition to call the Taylor integrator till the time has advanced in one unit, asking for a relative error less than 10^{-17}. The results are shown in Figure 2. The three columns that appear in the middle of the table are the following: the first column is a counter that is increased

```
value of H at the initial condition:  -1.3362071584596451
numerical integration starts...
1        0.240976814735   21      -1.0
2        0.494777364783   21      -2.0
3        0.764527205506   21      -2.0
4        1.000000000000   21      -2.0

total number of iterates: 4  final time: 1.000000e+00
```

Figure 2: Integration, for one unit of time, of the Restricted Three-Body Problem. See the text for details.

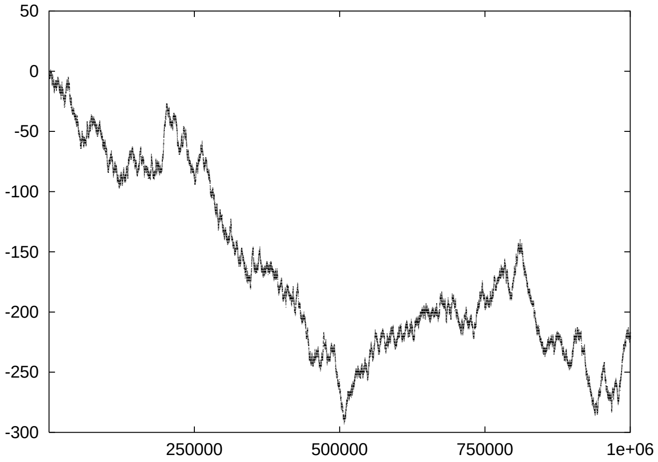

Figure 3: Long term behaviour of the energy. Horizontal axis: time. Vertical axis: variation of the value of the Hamiltonian, in multiples of the epsilon of the machine.

each time the main program calls the `taylor` integrator; the second column is the value of the time variable, the third column is the order used in the Taylor method, and the fourth value is the difference between the actual value of H and the corresponding value for the initial condition, *in multiples of the epsilon of the machine.*[2] We stress that, after the first step of the integrator, the energy has changed *in only one bit.* In the next three calls, the energy changes in an amount of two times the machine epsilon, which gives an idea of the extremely good behaviour of the rounding errors. The program stops exactly at time 1.

An interesting experiment is to perform a longer integration, to see the long term behaviour of the error. This is shown in Figure 3 for 10^6 units of time. Although the total amount of deviation is very small, we note that it does not behave as a random walk, since the plot shows a clear drift on the variation of the Hamiltonian. This is a weakness that should be taken into account in very long integrations.

---

[2] The epsilon of the machine is the smallest number we can add to 1 such that the result is different from 1. It is commonly used as a measure of the level of accuracy of a given arithmetic.

```
value of H at the initial condition: -0.13362071584596453978e1
numerical integration starts...
1    0.183827140545086 174 -0.82041748043202274176e-153
2    0.374344795509428 174 -0.59666725849601653946e-153
3    0.574965855180706 174 -0.6712506658080186069e-153
4    0.789299521404807 174 -0.14916681462400413487e-153
5    1.000000000000000 174 -0.2237502219360062023e-153

iterates: 5  final time: 1.000000e+00
```

Figure 4: Numerical integration of the vector field in Figure 1, using `gmp` with a 512 bits mantissa and asking for a relative error of 10^{-150}. See the text for more details.

5.2 Extended precision calculations

We will use the same example as in the previous section. The main difference when generating the code for the `taylor` integrator with extended precision is that we have to tell to the taylor translator that we want to use an extended arithmetic. For instance, to generate code using `gmp`, we can do

```
taylor -name rtbp -o taylor_rtbp.c -step -jet -sqrt rtbp.in
taylor -name rtbp -o taylor.h -gmp -header
```

(note that we only need to specify the kind of arithmetic when generating the header file). As a first test, we can repeat the run shown in Figure 2 using a 512 bits mantissa and asking for a relative error of 10^{-150}. The results are displayed in Figure 4 where, in the fourth column, we have replaced the machine epsilon by the difference in the value of H with respect to its initial value.

The output for a relative error of 10^{-300} (and using a mantissa of 1024 bits) is also shown in Figure 5. We note that the step sizes are very similar – the differences come from different estimates of the radius of convergence of the Taylor series – and the most relevant change is the order of the method.

5.3 Some comparisons

Let us start by comparing the implementation of the Taylor method presented here with the well-known Runge-Kutta-Fehlberg of orders 7 and 8 (from now on, rk78), using the standard double precision of the computer. First, let us repeat the calculation showed in Figure 3. We have asked the rk78 for an accuracy of 10^{-16}. Note that the step sizes controls of rk78 and Taylor method are totally different; from our numerical experiments, it seems that the best results (i.e., the best accuracy) for the rk78 in this problem are obtained by asking for a local error 10^{-16}. The results, both for the rk78 and the Taylor method, can be summarized as follows: the error in the value of H after 10^6 units of time, in number of multiples of the machine epsilon, is -13412 for the rk78, and -217 for the Taylor method. The time taken for the rk78 was of 9m and 34s, while the Taylor

```
value of H at the initial condition: -0.13362071584596453978e1
numerical integration starts...
1    0.180071007388544 346 0.12423699888974958922e-303
2    0.366492191198110 346 0.6782581389194576616e-303
3    0.562476214016376 346 0.87872616820166397021e-303
4    0.771527181403796 346 0.92184809957953353301e-303
5    0.997166681972274 346 0.10604312621720083032e-302
6    1.000000000000000 346 0.10604368248566545712e-302

iterates: 6  final time: 1.000000e+00
```

Figure 5: As Figure 4, but using a 1024 bits mantissa and asking for a relative error of 10^{-300}.

method needed 4m and 4s (these runs were done in a GNU/Linux environment, running on a Pentium II processor at 500 MHz). If we ask the rk78 for an accuracy of 10^{-14} then the time taken goes down to 4m and 58s, but the final error is 649368 times the epsilon of the machine (that is, 1.44×10^{-10}).

Next we describe a second benchmark using the standard quadruple precision (the type `long double` in C) of a HP 9000/712 computer, with a 100 MHz PA-RISC 1.1 processor. In this case, the `long double` type is 16 bytes long, holding nearly 33 decimal digits – we note that this is not the usual situation for many C compilers, in which the type `long double` is only 10 bytes long, holding nearly 19 decimal digits. We have used the vector field of the Restricted Three-Body Problem, with the same initial condition and mass parameter as before. The integration time has been restricted to 10 units, to avoid long testing times. We have asked for a local error of 10^{-32} for the rk78, and of the 10^{-33} for the Taylor method. The total cpu time taken for the rk78 is of 3m and 48s, while the Taylor method only needed 4.1 seconds.

Acknowledgements

The authors want to thank R. Broucke, R. de la Llave and C. Simó for their comments. This work has been supported by the Comisión Conjunta Hispano Norteamericana de Cooperación Científica y Tecnológica. A.J. has also been supported by the Spanish CICYT grant BFM2000–0623, the Catalan CIRIT grant 2000SGR–00027 and DURSI.

References

[Bar80] D. Barton. On Taylor series and stiff equations. *ACM Trans. Math. Software*, 6(3):280–294, 1980.

[BBCG96] M. Berz, C. Bischof, G.F. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools.* SIAM, Philadelphia, Penn., 1996.

[BCCG92] C.H. Bischof, A. Carle, G.F. Corliss, and A. Griewank. ADIFOR: Automatic differentiation in a source translation environment. In Paul S. Wang, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 294–302, New York, 1992. ACM Press.

[BKSF59] L.M. Beda, L.N. Korolev, N.V. Sukkikh, and T.S. Frolova. Programs for automatic differentiation for the machine BESM. Technical Report, Institute for Precise Mechanics and Computation Techniques, Academy of Science, Moscow, USSR, 1959. (In Russian).

[Bro71] R. Broucke. Solution of the N-Body Problem with recurrent power series. *Celestial Mech.*, 4(1):110–115, 1971.

[But87] J.C. Butcher. *The Numerical Analysis of Ordinary Differential Equations.* Wiley, 1987.

[BWZ70] D. Barton, I.M. Willers, and R.V.M. Zahar. The automatic solution of ordinary differential equations by the method of Taylor series. *Computer J.*, 14(3):243–248, 1970.

[CC82] G.F. Corliss and Y.F. Chang. Solving ordinary differential equations using Taylor series. *ACM Trans. Math. Software*, 8(2):114–144, 1982.

[CC94] Y.F. Chang and G.F. Corliss. ATOMFT: Solving ODEs and DAEs using Taylor series. *Computers and Mathematics with Applications*, 28:209–233, 1994.

[CGH$^+$97] G.F. Corliss, A. Griewank, P. Henneberger, G. Kirlinger, F.A. Potra, and H.J. Stetter. High-order stiff ODE solvers via automatic differentiation and rational prediction. In *Numerical analysis and its applications (Rousse, 1996)*, pages 114–125. Springer, Berlin, 1997.

[Cor95] G.F. Corliss. Guaranteed error bounds for ordinary differential equations. In M. Ainsworth, J. Levesley, W. A. Light, and M. Marletta, editors, *Theory of Numerics in Ordinary and Partial Differential Equations*, pages 1–75. Oxford University Press, Oxford, 1995. Lecture notes for a sequence of five lectures at the VI-th SERC Numerical Analysis Summer School, Leicester University, 25 - 29 July, 1994.

[GC91] A. Griewank and G.F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.* SIAM, Philadelphia, Penn., 1991.

[Gri00] A. Griewank. *Evaluating Derivatives.* SIAM, Philadelphia, Penn., 2000.

[Hoe01] J. Hoefkens. *Rigorous numerical analysis with high order Taylor methods.* PhD thesis, Michigan State University, 2001.

[JZ85] F. Jalbert and R.V.M. Zahar. A highly precise Taylor series method for stiff ODEs. In *Proceedings of the fourteenth Manitoba conference on numerical mathematics and computing (Winnipeg, Man., 1984)*, volume 46, pages 347–358, 1985.

[KC92] G. Kirlinger and G.F. Corliss. On implicit Taylor series methods for stiff ODEs. In *Computer arithmetic and enclosure methods (Oldenburg, 1991)*, pages 371–379. North-Holland, Amsterdam, 1992.

[Moo66] R.E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1966.

[MS99] R. Martínez and C. Simó. Simultaneous binary collisions in the planar four-body problem. *Nonlinearity*, 12(4):903–930, 1999.

[NJC99] N.S. Nedialkov, K.R. Jackson, and G.F. Corliss. Validated solutions of initial value problems for ordinary differential equations. *Appl. Math. Comput.*, 105(1):21–68, 1999.

[Ral81] L.B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, 1981.

[Sim] C. Simó. Dynamical properties of the figure eight solution of the three–body problem. To appear in A. Chenciner, R. Cushman, C. Robinson and Z. Xia, editors, *Proceedings of the Chicago Conference dedicated to Don Saari.*

[Sim01] C. Simó. Global dynamics and fast indicators. In H.W. Broer, B. Krauskopf, and G. Vegter, editors, *Global analysis of dynamical systems*, pages 373–389, Bristol, 2001. IOP Publishing.

[Ste56] J.F. Steffensen. On the restricted problem of three bodies. *Danske Vid. Selsk. Mat.-Fys. Medd.*, 30(18):17, 1956.

[Ste57] J.F. Steffensen. On the problem of three bodies in the plane. *Mat.-Fys. Medd. Danske Vid. Selsk.*, 31(3):18, 1957.

[SV01] C. Simó and C. Valls. A formal approximation of the splitting of separatrices in the classical Arnold's example of diffusion with two equal parameters. To appear in Nonlinearity, 2001.

[Sze67] V. Szebehely. *Theory of Orbits.* Academic Press, 1967.

[Wen64] R. E. Wengert. A simple automatic derivative evaluation program. *Comm. ACM*, 7(8):463–464, 1964.