A Software Package for the Numerical Integration of ODEs by Means of High-Order Taylor Methods

Àngel Jorba and Maorong Zou

CONTENTS

[To the memory of William F. Schelter]

This paper revisits the Taylor method for the numerical integration of initial value problems of Ordinary Differential Equations (ODEs). The main goal is to present a computer program that outputs a specific numerical integrator for a given set of ODEs. The generated code includes a function to compute the jet of derivatives of the solution up to a given order plus adaptive selection of order and step size at run time. 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, showing that it is very competitive in many situations. This is especially true for integrations that require extended precision arithmetic. 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.

1. INTRODUCTION

In this paper, we revisit one of the oldest numerical procedures for the numerical integration of ODEs—the Taylor method. Consider the initial value problem

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

where, for simplicity, we assume that f is analytic on its domain of definition and that $x(t)$ is defined for $t \in [a,b]$. We are interested in approximating the function $x(t)$ on $[a,b]$. The idea of the Taylor method is very simple: given the initial condition $x(t_0) = x_0$ $(t_0 = a)$, the value $x(t_0+h)$ is approximated by the Taylor series of the solution $x(t)$ at $t = t_0$,

$$\begin{aligned} x_0 &= x(t=0), \\ x_{m+1} &= x_m + x'(t_m)h + \frac{x''(t_m)}{2!}h^2 + \cdots \\ &\quad + \frac{x^{(p)}(t_m)}{p!}h^p, \quad m = 0,\dots,M-1, \end{aligned} \tag{1–2}$$

2000 AMS Subject Classification: [PRIMARY]

Keywords: [KEYWORDS]

where $t_m = a + mh$ and $h = (b - a)/M$. We refer to [Hairer et al. 00] for a discussion of the basic properties of the method. For practical implementation one needs an effective method to compute the values of the derivatives $x^{(j)}(t_m)$. One procedure to obtain them is to differentiate the first equation in (1–1) with respect to t, at the point $t = t_m$. Hence,

$$\begin{aligned} x'(t_m) &= f(t_m, x(t_m)), \\ x''(t_m) &= f_t(t_m, x(t_m)) + f_x(t_m, x(t_m))x'(t_m), \end{aligned}$$

and so on. Therefore, the first step in applying this method is, for a given f, to compute these derivatives up to a suitable order. Then, for each step of the integration (see (1–2)), we have to evaluate these expressions to obtain the coefficients of the power series of $x(t)$ at $t = t_m$. Usually these expressions are very cumbersome, so it takes a significant amount of time to evaluate them numerically. This, combined with the initial effort to compute the derivatives of f, is the main drawback of this approach to the Taylor method.

This difficulty can be overcome by means of the so-called *automatic differentiation* [Beda et al. 59, Wengert 64, Moore 66, Rall 81, Griewank and Corliss 91, Bischof et al. 92, Berz et al. 96, Griewank 00]. This is a procedure that allows for fast evaluation 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 [Steffensen 56, Steffensen 57] (see also [Broucke 71]). We give a brief account of automatic differentiation in Section 2

A drawback of this method is that f has to belong to a special class. Fortunately, this class is large enough to contain the functions that appear in many applications. We also note that the algorithm that computes these derivatives by automatic differentiation has to be coded separately for different systems. This coding can be either done by a human (see, for instance, [Broucke 71] for an example with the N-body problem) or by another program (see [Beda et al. 59, Gibbons 60, Chang and Corliss 94] for general-purpose computer programs). An alternative procedure for applying the Taylor method can be found in [Savageau and Voit 87] and [Irvine and Savageau 90].

The main goal of this paper is to present software that, given a function f (belonging to a suitable class), generates code to compute the jet of derivatives of the solution of $x' = f(t, x)$ at a given point (t_m, x_m). The order of the jet is given at run time. The software also generates code to compute, adaptively, an order and step size and to add the resulting Taylor series to predict a new point for the solution. Therefore, the output is a complete time-stepper with automatic order and step size control. The generated code is ANSI C, but we also provide a Fortran 77 wrapper for the main call to the time-stepper.

A software package that performs a similar task is ATOMFT, that can be freely downloaded from the Internet at http://www.eng.mu.edu/corlissg/FtpStuff/Atom3_11/. ATOMFT is written in Fortran 77, it reads Fortran-like statements of the system of ODEs and writes a Fortran 77 program that numerically solves the system using Taylor series.

One of the nicest characteristics of the Taylor method is the possibility of using interval arithmetic to derive bounds for the total error of the numerical integration. These ideas are used in ATOMFT to compute a step size that guarantees a prescribed accuracy, but using the standard floating point of the computer instead of interval arithmetic. We note that most of the algorithms for step size control are based on the asymptotic behavior of the error, and they do not provide true bounds for the truncation error of the method. On the other hand, the derivation of the time-step in ATOMFT is a substantial part of the computing time, while an estimation based on the asymptotic behavior of the error is usually much faster.

Here, we implement a step size control based on an asympotic estimate of the error. The main reason for this selection is that we want to compete against the "usual" numerical integrators—which use similar step size control techniques. Moreover, as we will see later, our software allows the user to plug in a user-defined step size control, so it is not difficult to implement different strategies.

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 ε. Since we have to select the value of two parameters (p and h), we can specify a second condition besides the size of the truncation error. Here, we chose to minimize the number of operations needed to advance the independent variable t in one unit [Simó 01]. We also code the algorithms to do these tasks so that the output of the program is, in fact, a complete numerical integrator—with automatic order and step size control—for the initial value problem (1–1).

We have tested this Taylor integrator against some well-known integration methods. The results show that the Taylor method to integrate is very competitive 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, [Martínez and Simó 99, Simó and Valls 01, Simó 02]). Methods whose order is not very high (less than, say, 16) can be extremely slow for computations requiring extended precision arithmetic. This is one of the strong points of the software presented here. Note that the Taylor method does not need to reduce the step size to increase accuracy; it can increase the order (see Section 3.3). As we will see, this allows a great reduction in the total number of arithmetic operations performed during numerical integration. Another advantage of our package is that it does not require expert knowledge of programming. All it needs for input is a system of ODEs in a natural mathematical form.

As with any explicit scheme, the Taylor method is not suitable for stiff equations, because, in this case, the errors can grow too fast. However, there are modifications of the Taylor method to deal with these situations [Barton 80, Jalbert and Zahar 85, Kirlinger and Corliss 92, Corliss et al. 97]. These modifications are not considered in our software.

In this paper, we present the main details of our implementation of the software. We tried to produce an efficient package, in the sense that the produced Taylor integrator be as fast as possible. Moreover, we also included support for multiple precision arithmetic. We conducted several tests 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, [Barton et al. 70, Corliss and Chang 82, Chang and Corliss 94, Hoefkens 01]). A good survey is [Nedialkov et al. 99] (see also [Corliss 95]).

The package has been released under the GNU Public License, so anybody with Internet access is free to use and to redistribute it. To obtain a copy, see [Zou and Jorba 01].

We note that our version of the package is written to run under the GNU/Linux operating system. We do not expect major problems running it under any version of Unix, but we do not plan to write ports for other operating systems.

The paper is 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 tests and comparisons.

2. A SHORT SUMMARY ON AUTOMATIC DIFFERENTIATION

Before starting with the discussion of the package, we will summarize the main rules of automatic differentiation.

Automatic differentiation is a recursive procedure that computes the value of the derivatives of certain functions at a given point (see [Moore 66, Rall 81]). The functions considered are those that can be obtained by sum, product, quotient, and composition of elementary functions (elementary functions include polynomials, trigonometric functions, real powers, exponentials, and logarithms).

2.1 Rules of Automatic Differentiation

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 nth derivative

$$a^{[n]}(t) = \frac{1}{n!}a^{(n)}(t), \tag{2–1}$$

where $a^{(n)}(t)$ denotes the nth derivative of a with respect to t. In what follows, we focus on the computation of the values $a^{[n]}(t)$.

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 nth 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

$$a^{[n]}(t) = \sum_{j=0}^{n} b^{[n-j]}(t)c^{[j]}(t).$$

(3) If $a(t) = \frac{b(t)}{c(t)}$, then

$$a^{[n]}(t) = \frac{1}{c^{[0]}(t)}\left[b^{[n]}(t) - \sum_{j=1}^{n} c^{[j]}(t)a^{[n-j]}(t)\right].$$

(4) If $a(t) = b(t)^{\alpha}$, then

$$a^{[n]}(t) = \frac{1}{nb^{[0]}(t)}\sum_{j=0}^{n-1}\left(n\alpha - j(\alpha+1)\right)b^{[n-j]}(t)a^{[j]}(t).$$

(5) If $a(t) = e^{b(t)}$, then

$$a^{[n]}(t) = \frac{1}{n}\sum_{j=0}^{n-1} (n-j)\, a^{[j]}(t) b^{[n-j]}(t).$$

(6) If $a(t) = \ln b(t)$, then $a^{[n]}(t) =$

$$\frac{1}{b^{[0]}(t)}\left[b^{[n]}(t) - \frac{1}{n}\sum_{j=1}^{n-1}(n-j) b^{[j]}(t) a^{[n-j]}(t)\right].$$

(7) If $a(t) = \cos c(t)$ and $b(t) = \sin c(t)$, then

$$a^{[n]}(t) = -\frac{1}{n}\sum_{j=1}^{n} j b^{[n-j]}(t) c^{[j]}(t),$$

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

Proof: Item (1) is obvious. Item (2) follows from Leibniz's formula:

$$\begin{aligned} a^{[n]}(t) &= \frac{1}{n!} a^{(n)}(t) \\ &= \frac{1}{n!}\sum_{j=0}^{n}\binom{n}{j} b^{(n-j)}(t) c^{(j)}(t) \\ &= \sum_{j=0}^{n} b^{[n-j]}(t) c^{[j]}(t). \end{aligned}$$

To prove Item (3), apply Item (2) to $a(t)c(t) = b(t)$. For Item (4), take logarithms and derivatives to obtain $a'(t)b(t) = \alpha a(t) b'(t)$ and then use Item (2). To prove Item (5), take logarithms and derivatives to obtain $a'(t) = a(t)b'(t)$ and use Item (2). For Item (6), take derivatives to obtain $a'(t)b(t) = b'(t)$ and use Item (2). Finally, for Item (7), take derivatives to obtain $a'(t) = -b(t)c'(t)$ and $b'(t) = a(t)c'(t)$, and then use Item (2). □

Remark 2.2. It is possible to derive similar formulas for other functions, such as inverse trigonometric functions.

Corollary 2.3. *The number of arithmetic operations needed to evaluate the normalized derivatives of a function up to order n is $O(n^2)$.*

Although these methods only allow for the derivation of a reduced subset of the set of analytic functions, we note that they cover the situations found in many applications.

2.2 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 combinations of these basic functions. As an example, we apply them to the Van der Pol equation,

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

To this end we decompose the right-hand side of these equations in a sequence of simple operations:

$$\left.\begin{aligned} u_1 &= x \\ u_2 &= y \\ u_3 &= u_1 u_1 \\ u_4 &= 1 - u_3 \\ u_5 &= u_4 u_2 \\ u_6 &= u_5 - u_1 \\ x' &= u_2 \\ y' &= u_6 \end{aligned}\right\}. \tag{2–3}$$

Then, we apply the formulas given in Proposition 2.1 (Items (1) and (2)) to each of the equations in (2–3) to derive recursive formulas for $u_j^{[n]}$, $j = 1, \ldots, 6$,

$$\begin{aligned} u_1^{[n]}(t) &= x^{[n]}(t), \\ u_2^{[n]}(t) &= y^{[n]}(t), \\ u_3^{[n]}(t) &= \sum_{i=0}^{n} u_1^{[n-i]}(t) u_1^{[i]}(t), \\ u_4^{[n]}(t) &= -u_3^{[n]}(t),\ n > 0, \\ u_5^{[n]}(t) &= \sum_{i=0}^{n} u_4^{[n-i]}(t) u_2^{[i]}(t), \\ u_6^{[n]}(t) &= u_5^{[n]}(t) - u_1^{[n]}(t), \\ x^{[n+1]}(t) &= \frac{1}{n+1} u_2^{[n]}(t), \\ y^{[n+1]}(t) &= \frac{1}{n+1} u_6^{[n]}(t). \end{aligned}$$

The factor $\frac{1}{n+1}$ in the last two formulas comes from the definition given in Equation (2–1). Then, we 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 at a given point of the ODE. Note that it is not necessary to select the value of p in advance.

Three of the tasks of the software we present are to read the system of ODEs specified as in (2–2), to decompose it into a sequence of basic operations, and to apply the formulas in Proposition 2.1 to this decomposition. This results in an ANSI C routine that, given an initial condition x_0 and a degree p, returns the jet of normalized derivatives of the solution at the pont x_0 up to degree p.

3. DEGREE AND STEP SIZE CONTROL

The power expansion of the solution $x(t)$ at $t = t_m$ will have very different radii of convergence for different t_m, and an efficient integration algorithm must take this into account. This means that, at each step (i.e., for each m), we have to compute suitable values for the order $p = p_m$ and the step size $h = h_m$.

Since two parameters (order and step size) are needed to achieve a given level of accuracy, we also try to minimize the total number of arithmetic operations to go from $t = a$ to $t = b$, so that the resulting method is as fast as possible.

3.1 On the Optimal Selections

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

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

Therefore, we want to select a sufficiently small value h_m and a sufficiently large value p_m such that the values

$$t_{m+1} \equiv t_m + h_m, \qquad x_{m+1} \equiv \sum_{j=0}^{p_m} x_m^{[j]}(t_m) h_m^j,$$

satisfy

$$\|x_m(t_{m+1}) - x_{m+1}\| \le \varepsilon.$$

On the other hand, to minimize the global number of operations of the numerical integration we want to choose a step size h_m as large as possible and an order p_m as small as possible. To determine such values, we need some assumptions about the analytical properties of the solution $x(t)$. The following result can be found in [Simó 01].

Proposition 3.1. *Assume that the function $z \mapsto x(t_m + z)$ is analytic on a disk of radius ρ_m. Let A_m be a positive constant such that*

$$|x_m^{[j]}| \le \frac{A_m}{\rho_m^j}, \qquad \forall\, j \in \mathbb{N}. \tag{3–1}$$

Then, if the required accuracy ε tends to 0, the values of h_m and p_m that give the required accuracy and minimize the global number of operations tend to

$$h_m = \frac{\rho_m}{e^2} \quad \textit{and} \quad p_m = -\frac{1}{2}\ln\left(\frac{\varepsilon}{A_m}\right) - 1. \tag{3–2}$$

Remark 3.2. It is important to note that the values in (3–2) are optimal only when the bound in (3–1) cannot be improved. If the value A_m can be reduced—or if $x(t)$ is an entire function—the previous values are not optimal, in the sense that a larger h_m and/or a smaller p_m could still deliver the required accuracy.

Remark 3.3. Note that the optimal step size does not depend on the level of accuracy. The optimal order is, in fact, the order that guarantees the required precision once the step size has been selected.

There are strategies to use step sizes that are larger than the radius of convergence of the series (see [Corliss and Chang 82]), but they only work for some singularities of $x(t)$ and require some computational effort (although this effort can pay off when the solution is close enough to one of the considered singularities). As mentioned before, we implemented a more straightforward algorithm based on Proposition 3.1.

3.2 Estimations of Order and Step Size

The main drawback of Proposition 3.1 is that it requires information that we cannot obtain easily, like the radius of convergence of the Taylor series or the value A_m. In this section we first describe, schematically, the numerical implementation and then we justify it.

Let us denote by ε_a and ε_r the absolute and relative tolerances for error. If $\varepsilon_r\|x_m\|_\infty \le \varepsilon_a$ we try to control the absolute error using ε_a; otherwise we try to control the relative error using ε_r. Note that, in any case, we are controlling the absolute error by $\max\{\varepsilon_a, \varepsilon_r\|x_m\|_\infty\}$.

First, we compute the order p_m for the Taylor method as follows: we define ε_m as

$$\varepsilon_m = \begin{cases} \varepsilon_a & \text{if} \quad \varepsilon_r\|x_m\|_\infty \le \varepsilon_a, \\ \varepsilon_r & \text{otherwise,} \end{cases} \tag{3–3}$$

and then,

$$p_m = \left\lceil -\frac{1}{2}\ln \varepsilon_m + 1 \right\rceil, \tag{3–4}$$

where $\lceil\cdot\rceil$ stands for the ceiling function. If we compare this with Proposition 3.1, we see that it is as if the value A_m is 1 and that p_m is two units larger. This is rigorously justified in Proposition 3.4.

To derive the step size, we distinguish the same two cases as before: if $\varepsilon_r\|x_m\|_\infty \le \varepsilon_a$, we define

$$\rho_m^{(j)} = \left(\frac{1}{\|x_m^{[j]}\|_\infty}\right)^{\frac{1}{j}}, \quad 1 \le j \le p, \tag{3–5}$$

and, if $\varepsilon_r\|x_m\|_\infty > \varepsilon_a$,

$$\rho_m^{(j)} = \left(\frac{\|x_m\|_\infty}{\|x_m^{[j]}\|_\infty}\right)^{\frac{1}{j}}, \quad 1 \le j \le p. \tag{3–6}$$

In both cases, we estimate that the radius of convergence is the minimum of the last two terms,

$$\rho_m = \min\left\{\rho_m^{(p-1)}, \rho_m^{(p)}\right\}, \tag{3–7}$$

and the estimated time-step is

$$h_m = \frac{\rho_m}{e^2}. \tag{3–8}$$

The next proposition bounds the truncation error corresponding to order p_m and step h_m. This is especially important in the general case $A_m \neq 1$. Using the previous notations and definitions, we prove the following proposition.

Proposition 3.4.

(1) If $\varepsilon_r\|x_m\|_\infty \leq \varepsilon_a$, we have

$$\|x_m^{[p_m-1]}h_m^{p_m-1}\|_\infty \leq \varepsilon_a, \qquad \|x_m^{[p_m]}h_m^{p_m}\|_\infty \leq \frac{\varepsilon_a}{e^2}.$$

(2) If $\varepsilon_r\|x_m\|_\infty > \varepsilon_a$, we have

$$\frac{\|x_m^{[p_m-1]}h_m^{p_m-1}\|_\infty}{\|x_m\|_\infty} \leq \varepsilon_r, \qquad \frac{\|x_m^{[p_m]}h_m^{p_m}\|_\infty}{\|x_m\|_\infty} \leq \frac{\varepsilon_r}{e^2}.$$

Proof: From Equation (3–4), it follows that $e^{2(p_m-1)} \geq \varepsilon_m^{-1}$.

(1) This is equivalent to using Equation (3–5) in Equation (3–7). Therefore,

$$\|x_m^{[p_m-1]}h_m^{p_m-1}\|_\infty \leq \frac{\|x_m^{[p_m-1]}\rho_m^{p_m-1}\|_\infty}{e^{2(p_m-1)}} \leq \varepsilon_a.$$

A similar reasoning shows the second inequality.

(2) In this case we use Equation (3–6) in Equation (3–7). So,

$$\frac{\|x_m^{[p_m-1]}h_m^{p_m-1}\|_\infty}{\|x_m\|_\infty} \leq \frac{\|x_m^{[p_m-1]}\rho_m^{p_m-1}\|_\infty}{\|x_m\|_\infty e^{2(p_m-1)}} \leq \varepsilon_r,$$

and the the second inequality follows easily.

□

Remark 3.5. Note that the term of order $p_m - 1$ in the Taylor series has a contribution of order ε_m while the term of order p_m (the last term to be considered) has a contribution of order ε_m/e^2. This shows that the proposed strategy is similar to the more straightforward method of looking for an h_m such that the last terms in the series are of the order of the error wanted.

Remark 3.6. Note that, although the formula for order uses $A_m = 1$, its real value is taken into account in Formulas (3–5) and (3–6). This is the reason why Proposition 3.4 holds when $A_m \neq 1$.

If the solution is entire—therefore, the bound (3–1) is far from optimal—then the computed values for p_m and h_m still satisfy the accuracy requirements without needing to be the ones that minimize the global number of operations (see Remark 3.2).

These results have been used to implement two step size controls.

3.2.1 First step size control.

This is equivalent to using Formulas (3–3) and (3–4) for the order and (3–5), (3–6), and (3–7) for the radius of convergence. Since these calculations are based on asymptotic estimates, we add a safety factor to Formula (3–8) to derive the step size:

$$h_m = \frac{\rho_m}{e^2}\exp\left(-\frac{0.7}{p_m - 1}\right).$$

For instance, for $p_m = 8$ the safety factor is 0.90 and for $p_m = 16$, is 0.95. These are typical safety factors used in many step size controls.

3.2.2 Second step size control.

This is a correction of the previous method that avoids overly large step sizes that could lead to cancellations when adding the Taylor series. A natural solution is to look for a step size such that the resulting series has all the terms decreasing in modulus. However, if the solution $x(t)$ has some intermediate Taylor coefficients that are very small, this technique could lead to very drastic (and unnecessary) step reductions. Therefore, we have used a weaker criterion: let $\bar{h}_m$ be the step size control obtained in Section 3.2.1 and let us define z as

$$z = \begin{cases} 1 & \text{if} \quad \varepsilon_r\|x_m\|_\infty \leq \varepsilon_a, \\ \|x_m\|_\infty & \text{otherwise.} \end{cases}$$

Let $h_m \leq \bar{h}_m$ be the largest value such that

$$\|x_m^{[j]}\|_\infty h_m^j \leq z, \qquad j = 1, \ldots, p.$$

We note that, in many cases, it is enough to take $h_m = \bar{h}_m$ to meet this condition.

3.3 High Accuracy Computations

An important property of high order Taylor integrators is their suitability for computations requiring high accuracy. For instance, assume that we are solving an IVP like (1–1) and that, at a given step, we use a step size

$h \ll 1$ and an order p to obtain a local error $\varepsilon \ll 1$. The number of operations needed to compute all the derivatives is $O(p^2)$ (see Corollary 2.3). Since the number of operations needed to sum the power series is only $O(p)$, the total operation count for a single step of the Taylor method is still $O(p^2)$. Hence, if we want to increase the accuracy to, say, ε^ℓ ($\ell \geq 2$) we can simply increase the order of the Taylor method to ℓp; so the number of operations is increased by a factor ℓ^2. 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 a step size of h^ℓ. This means that we have to use $1/h^{\ell-1}$ steps (each of size h^ℓ) to compute the orbit after h units of time; so the total number of operations is now increased by a factor of $1/h^{\ell-1}$, usually much larger than ℓ^2.

Hence, it requires much less work to increase the order than to reduce the step size. (This observation was already implicit in Proposition 3.1, where it was shown that the optimal step size is independent of the level of accuracy required.) Therefore, fixed order methods are strongly penalized for high accuracies, compared with varying order methods. For this reason, if the required accuracy is high enough, the Taylor method—with varying order—is one of the best options.

4. SOFTWARE IMPLEMENTATION

`Taylor` is a translator, it reads a system of ODEs, in its natural form, from an input file and generates a set of C routines that implements the Taylor method for the given system. `Taylor` supports a tiny language based on the following statements.

```
id = expr;

diff(v, t) = expr;
```

Here `t` is the independent variable and `v` is a state variable. We use the first statement to define either a constant, or a shorthand notation for a complex expression used in the differential equations. It is normally used to help the translator factor out common expressions, so it may generate smaller and faster codes. Variable names defined this way cannot be redefined, i.e., the symbols appearing on the left hand side must be unique. We use the second statement to define a differential equation; if the system of ODEs is autonomous, we can simply use `v’ = expr;` instead. The order that statements are input is not important.

Expressions are generated from numbers, the independent variable, the state variables, external variables, and using elementary functions: sin, cos, tan, arctan, sinh, cosh, tanh, roots ($\sqrt{\ }$), exp, and log, the four arithmetic operators, and function composition. A branching construct `if(bexpr) {expr} else {expr}` is also supported, here, `bexpr` is a boolean expression as defined in the C programming language.

4.1 The Parser

`Taylor` is written using compiler design tools *Lex* and *Yacc*. While these tools are standard in the software engineering world, their use in scientific computing is much less visible. In the following, we discuss the implementation of `Taylor` and give a brief account of these tools. A detailed exposition can be found in [Aho et al. 86].

`Taylor` works in several phases with each phase passing its output to the next phase. The first phase is the lexical phase. Here characters from the input stream are grouped into lexical units, called tokens, by a scanner (lexical anaylizer). Regular expressions are used to define patterns that will be recognized by the scanner. The scanner is implemented as a finite state automata. Our scanner is generated by *Lex*, a scanner generator developed by M. E. Lesk and E. Schmidt of AT&T Bell Labortories. *Lex* produces a C procedure `yylex()` that the parser calls repeatedly in order to fetch the next token from the input stream. The *Lex* input file contains three sections separated by `%%`. An excerpt of our input file looks like the following.

```
/* 1. definitions/declarations       */
%{
#include  "Header.h"
#include  "y.tab.h"
extern Node current_id;
%}
%%
/* 2. regular expressions and actions */
[ \t\n] {
  /* white spaces  */
  ;
}
diff {
  /* keyword       */
  return(DIFF);
}
[A-Za-z][A-Za-z0-9_]* {
  /*  identifier   */
  current_id = install_id(yytext);
  return(ID);
}
[0-9]+ {
  /*  integer const*/
  yylval.ntype = build_int(yytext);
  return(INTCON);
}
"+" {
  /* sum operator  */
  yylval.code = PLUS_EXPR;
  return('+');
}
%%
/* 3. Supporting C functions  */
```

The first section contains definitions that will be used by the actions in the second section and/or supporting routines in the third section. This section must bracketed with `%{` and `%}` markers and is copied to the *Lex* output without modification. In our example, the variable `current_id` is used in an action rule, so we declare it here. Header file `Header.h` contains type definitions and forward declarations; `y.tab.h` is generated by *Yacc*, it contains definitions of symbolic token names `DIFF`, `ID`, and `INTCON`.

The second section contains regular expressions for each token that will be recognized, and their associated actions. In our example, white spaces (blanks, tabs, and newlines) are ignored. Strings of one or more digits are recognized and installed as integer constants, and are returned to the parser as a symbolic name `INTCON`. The reserved word `diff` is returned to the parser as `DIFF`. Alphanumeric identifiers are installed in the symbol table and returned as `ID`. The literal '`+`' is returned as a sum operator `PLUS_EXPR`.

The third section is optional. It normally contains C code used in the action rules in standalone *Lex* applications. For our application, the supporting functions are implemented in separate files.

Lex input is compiled with the unix command `lex file.l`. This command generates a file `lex.yy.c` which defines the C function `yylex()`.

The next phase is syntax analysis. Here a *parser* groups tokens into syntactical units and verifies that the input is syntactically valid according to a set of context-free grammar rules. The output of the parser is the parse tree, a graphical representation of the input. Our parser is generated by *Yacc*, a parser generator developed by S. C. Johnson at AT&T Bell Laboratories in the early 1970s. The *Yacc* parser is implemented as a pushdown automata with two stacks. The *Yacc* input file consists of three sections separated by `%%` markers. The first section contains token declarations and specifies the grammar start symbol. This section may specify the precedence and associativity of operators. In the event that the stack type is a union, the entire collection of possible data types must be declared in this section using the `%union` directive; and the data type for nonterminal symbols must be declared using the `%type` directive. C code may also be included in this section bracketed by `%{` and `%}`. Here is part of the first section of our *Yacc* input file.

```
%{
#include   <stdio.h>
#include   "Header.h"
extern int yylex();
Node       current_id;
%}

%start program
%union { Node ntype; enum node_code code; }
%token    ID INTCON FLOATCON DIFF
%nonassoc IF ELSE
%left  <code>  '+' '-' '*' '/'  OR AND EQ NEQ LE GE LT GT
%right <code>  '^' UNARY
%type  <ntype> ID INTCON FLOATCON
%type  <ntype> idexpr id term expr bexpr decl_id
               declare_one declrs
```

The second section of the input file contains context-free grammar rules in Backus Naur Form, separated by ";". The left-hand side of a production is entered left-justified, followed by a ":", then followed by the right-hand side of the production. Actions associated with a rule are then entered in braces. A simplified version of our grammer looks like the following.

```
program:
            /* empty */
            | stmts ';'
            ;
stmts:
             stmt
            | stmts ';' stmt
            ;
stmt:
             derivative
            | define
            ;
derivative:
             id '\''  '='  expr
              { record_one_equation($1, NULL, $4);}
            | DIFF '(' id ',' id ')' '=' expr
              { record_one_equation($3, $5, $8);}
            ;
define:
              id  '='  expr
              { define_one_variable($1, $3);}
            ;
id:
            ID
            { $$ = current_id; }
            ;
expr:
              term
            | expr '^' expr
              { $$ = build_op(EXP_EXPR,$1,$3); }
            | expr '*' expr
              { $$ = build_op(MULT_EXPR,$1,$3); }
            | expr '/' expr
              { $$ = build_op(DIV_EXPR,$1,$3); }
            | expr '+' expr
              { $$ = build_op(PLUS_EXPR,$1,$3); }
            | expr '-' expr
              { $$ = build_op(MINUS_EXPR,$1,$3); }
            | '-' expr   %prec UNARY
              { $$ = build_op(NEGATE_EXPR, $2, NULL); }
            ;
term:
              ID
              { $$ = current_id;}
            | INTCON
```

```
        | FLOATCON
        | '(' expr ')'
          { $$ = $2; }
        | idexpr '(' expr ')'
          { $$ = build_op(CALL_EXPR, $1, $3); }
          ;
```

The third section of the *Yacc* input file contains C code. As in the case of *Lex*, this section is optional. Supporting C functions are normally defined in separate files.

The *Yacc* input file is compiled using the command `yacc -vd file.y`. This command generates two files, `y.tab.h` and `y.tab.c`. The first file contains the list of tokens included in the `file.y` that defines the scanner. The second file contains the C code for the parser `yyparse()`. This file has to be compiled and linked with `lex.yy.c` and other supporting files.

To illustrate the parsing process, let's look at a concrete example:

$$\left.\begin{array}{rcl} x' &=& x(1-x^2-y^2)+y \\ y' &=& y(1-x^2-y^2)-x \end{array}\right\}. \qquad (4–1)$$

The input file to `taylor` contains just two lines.

```
x' =  x*(1-x^2-y^2) + y;
y' =  y*(1-x^2-y^2) - x;
```

The parsing process starts with a call to `yyparse()`, the parser. The parser repeatedly calls the scanner `yylex()` to fetch tokens from the input stream and to construct the parse tree implictly. As the parser runs, it builds an internal representation of the input structure. The internal representation is based on the right-hand side of the grammer rules. When a right-hand side is recognized, it is reduced to the corresponding left-hand side. When the entire input is reduced to the start symbol of the grammer, the parsing is complete and the input is accepted.

In our example, the scanner breaks the input into the following list of tokens,

```
x ' = x * ( 1 - x ^ 2 - y ^ 2 ) + y ;
y ' = x * ( 1 - x ^ 2 - y ^ 2 ) - x ;
```

and the parser records two equations when the parser returns. A graphical representation of the parsed input is shown in Figure 1.

The next phase is optimization. At this stage, the parse tree is analyzed and modified using semantics that preserve transformations. The crucial tasks performed at this phase are:

- Identify and mark constant expressions. Constant expressions are trivial to handle when computing high order derivatives. In our example, terminal nodes marked '1' and '2' are all marked as constants. Note however, we do not fold constants because of the loss of precision when the resulting code is linked with multi-precision libraries.

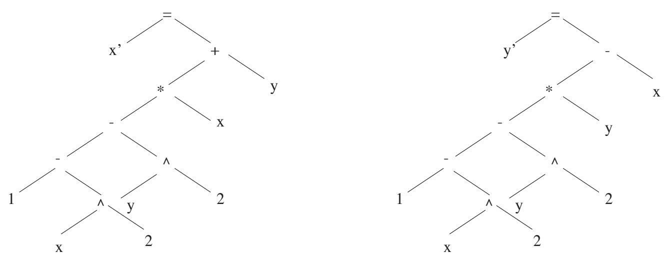

FIGURE 1. Parse tree for Equation (4–1).

- Eliminate common subexpressions. Here the task of searching for common subexpressions is done almost entirely at the "lexical level." Algebraic simplifications are not implemented except for the trivial commutative substituations $ab = ba, a+b = b+a$. For example, the expressions $5x^2+3$ and $3+5x^2$ are considered the same, while $2x^2+3$, $2x^2+2+1$, and x^2+x^2+3 are considered different. At this step, we traverse the right branch of the parse tree for each differential equation and annotate nodes by their defining subexpressions. For nonterminal nodes, temporary variables are introduced with their defining expressions recorded as their attributes. The pool of temporary variables are then compared pairwisely using their defining expressions. If two variables are found to be the same, one is eliminated. This process continues until no redundant variables are found. For our example, the following new temporary variables are introduced.

```
v01=x^2;   v02=1-v01;   v03=y^2;   v04=v02-v03;
v05=v04*x; v06=v05+y;   v07=x^2;   v08=1-v07;
v09=y^2;   v10=v08-v09; v11=v10*y; v12=v11-x;
```

It is then discovered that `v01` and `v07` are equivalent, hence `v07` is eliminated. All reference to variable `v07` is replaced by `v01`. After this replacement, the search for redundant variables starts over again. It then finds the following variables are equivalent: `v02` and `v08`, `v03` and `v09`, `v4` and `v10`, in that order. In the end, four temporary variables are eliminated.

- Introduce auxiliary variables for some elementary functions. For example, a new variable $v = \cos(x)$ is added to the symbol table if $\sin(x)$ appears on the parse tree.

- Build dependency graphs among all the variables and order the variables according to the dependency graph. In our example, the proper order of variables is: `x,y,v01,v02,v03,v04,v05,v06,v11,v12`.

The following user controlled "optimization" is also performed at this stage.

- Expand power functions as series of products. This procedure is controlled by the `-expandpower` command line switch. For example, $y = x^7$ is replaced by $u = x * x$, $v = u * u$, $w = u * v$, and $y = x * w$ if `taylor` is invoked with the option `-expandpower 7`. One reason to expand a power function using products is to avoid the singularity of Item 4 in Proposition 2.1 at the origin.

The output of this phase is an ordered list of "three variable code" like that in Equation (2–3).

The final phase is code generation. We apply formulas from Section 2 to all the "three variable code" produced by the optimizer, in order to generate procedures that compute Taylor coefficients sequentially. In our example, the list of "three variable codes" are:

```
v01=x^2;   v02=1-v01; v03=y^2;   v04=v02-v03;
v05=v04*x; v06=v05+y; v11=v04*y; v12=v11-x;
```

The original ODE system becomes `x'=v06; y'=v12;`.

4.2 Extended arithmetic

When `taylor` generates the code for the jet of derivatives and/or the step size control, it declares all the real variables with a special type called `MY_FLOAT`, and each mathematical operation is substituted by a suitable macro call (the name of these macros is independent from the arithmetic).

The definition of the type `MY_FLOAT` and the body of the macros is contained in a header file. This file is produced by invoking `taylor` with the flag `-header` plus a flag specifying the arithmetic wanted. For instance, to multiply two real numbers ($z = xy$), `taylor` outputs the code

```
   MultiplyMyFloatA(z,x,y);
```

If we call `taylor` with the `-header` flag and without specifying the desired arithmetic, it assumes we want the standard double precision and it generates a header file with the lines,

```
   typedef double MY_FLOAT;
```

to define `MY_FLOAT` as `double`. We also get the line

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

but, if we use the flag `-gmp` to ask for the GNU multiple precision arithmetic (see below), we get

```
   #define MY_FLOAT  mpf_t
```

and

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

Here, `mpf_mul` is the `gmp` function that multiplies the two numbers `a` and `b` and stores the result in `r`. Then, the C preprocessor substitutes the macros by the corresponding calls to the arithmetic library.

The package includes support for several extended precision arithmetics: `doubledouble` [Briggs 02], `dd_real`, `dq_real` [Bailey et al. 05], and `gmp` (the GNU Multiple Precision Library) [GMP 05]. Although these libraries do not contain implementations of trigonometric functions and other transcendental functions, we note that they can be defined by means of differential equations. Therefore, if an ODE includes some of these functions, we can enlarge the system of ODEs by adding the differential equation for the special function and integrating the whole system.

None of these floating point libraries is included in our package. They can be downloaded from the internet and are only needed if extended precision is required.

Note that, to use an arithmetic different from the ones provided here we only have to modify the header file. For more details, see the manual that comes with the software.

4.3 Using the Package

Here we describe briefly how to use the `taylor` program in a concrete example, the Restricted Three-Body Problem (RTBP for short). This is a well-known problem in celestial mechanics, that boils down to describeing the solutions of the differential equations

$$\begin{aligned}
\dot{x} &= p_x + y, \\
\dot{y} &= p_y - x, \\
\dot{z} &= p_z, \\
\dot{p}_x &= p_y - \frac{1-\mu}{r_{PS}^3}(x-\mu) - \frac{\mu}{r_{PJ}^3}(x-\mu+1), \\
\dot{p}_y &= -p_x - \left(\frac{1-\mu}{r_{PS}^3} + \frac{\mu}{r_{PJ}^3}\right) y, \\
\dot{p}_z &= -\left(\frac{1-\mu}{r_{PS}^3} + \frac{\mu}{r_{PJ}^3}\right) z,
\end{aligned} \qquad (4\text{–}2)$$

where μ is a mass parameter, $r_{PS}^2 = (x-\mu)^2 + y^2 + z^2$, and $r_{PJ}^2 = (x-\mu+1)^2 + y^2 + z^2$. For a more complete description of the problem see, for instance, [Szebehely 67] or [Meyer and Hall 92].

An input file for this vector field is shown in Figure 2. Although its syntax was already explained in Sections 4.1 and 4.2, we briefly describe it as an example. First of all, anything between `/*` and `*/` is ignored, so we can use them to put comments in the file. Next, we have some lines that define numerical constants, plus some with operations using the variables of the system. The variables of the equation are labeled as `x1`, `x2`, and so on, and the independent variable is labeled as `t`. Finally, the last six lines define the differential equations.

With the exception of eliminating common expressions, `taylor` does not perform any kind of optimization on the input description of the vector field. If you are concerned about the efficiency of the code generated by

t	$e(x)$	$e(y)$	$e(z)$	$e(p_x)$	$e(p_y)$	$e(p_z)$
0.2401192324190174	0.00	0.00	-1.00	1.00	-1.00	0.00
0.4952158876100076	0.00	1.00	-1.00	0.00	-2.00	-0.50
0.7653659470347371	0.00	1.00	-1.50	0.00	-1.50	1.00
1.0000000000000000	0.00	1.00	-0.50	0.00	-1.50	2.00

TABLE 1. Local relative error for an orbit of the RTBP. The first column denotes the time, and the remaining ones, the relative error for each coordinate in multiples of the machine precision. See the text for more details.

taylor, you should apply other kinds of optimizations "by hand" in your input file (for instance, to simplify algebraic expressions in order to minimize the number of operations).

A second point, upon which we want to comment, is the use of the exponent $-3.0/2$ in the expressions. There are several ways of introducing such an exponent. If we use -1.5 as the exponent, the program uses the exp and ln functions to define it (this is true for any real exponent). If we use $-3.0/2$, we can use the translator's flag -sqrt 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 shown here (such as the use of extern variables to receive parameters from the user's programs); for details check the package documentation.

To produce a numerical integrator for this vector field, assume that we have the code of Figure 2 in the file rtbp.in. Then, you can type

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

The first line outputs the file rtbp.c containing the code for the step size control and the jet of derivatives. The second line produces the header file needed by rtbp.c; 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, see the documentation for more details.

```
/* ODE specification: rtbp */
mu=0.01;
umu=1-mu;
r2=x1*x1+x2*x2+x3*x3;
rps2=r2-2*mu*x1+mu*mu;
rps3i=rps2^(-3./2);
rpj2=r2+2*(1-mu)*x1+(1-mu)*(1-mu);
rpj3i=rpj2^(-3./2);

diff(x1, t)= x4+x2;
diff(x2, t)= x5-x1;
diff(x3, t)= x6;
diff(x4, t)= x5-(x1-mu)*(umu*rps3i)-(x1+umu)*(mu*rpj3i);
diff(x5, t)=-x4-x2*(umu*rps3i+mu*rpj3i);
diff(x6, t)=-x3*(umu*rps3i+mu*rpj3i);
```

FIGURE 2. Input file for the restricted three-body problem.

5. SOME TESTS AND COMPARISONS

To show the main features of taylor, we selected three vector fields. In the first example (the RTBP) we performed a detailed study of error propagation, including comparisons with different floating point arithmetics. In Section 5.2 we show an example requiring extended precision arithmetic, and in Section 5.3 we compare the speed of the Taylor integrator with some common methods. We will also compare the speed of generation of the jet of derivatives with ADOL-C, a public domain package for automatic differentiation.

5.1 The Restricted Three-Body Problem

We started by doing some numerical integrations of the RTBP (see Section 4.3). It is well-known that the solutions of the equations in (4–2) have a preserved quantity,

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

This function is known as the Hamiltonian function of the RTBP, and it plays the role of the system's mechanical energy.

For our experiment, we used $\mu = 0.01$ and initial values x1=-0.45, x2=0.80, x3=0.00, x4=-0.80, x5=-0.45, and x6=0.58. This setup produces a stable orbit that seems to lay in a region almost filled up with quasiperiodic motions. In particular, the trajectory stays away from the singularities of the vector field.

5.1.1 Local error. First, we perform a numerical integration with the standard double precision of the computer, for 1 unit of time, using the error thresholds $\varepsilon_r = \varepsilon_a = 10^{-16}$, with the step size algorithm explained in Section 3.2.2 (in this case, the order of the Taylor expansion is 20). To check the accuracy, we performed the same integration with extended arithmetic (GMP), using the same time-step but with a higher order Taylor series (typically, two times the degree used in the double precision integration). To measure the error, we computed the relative difference between these two approximations. For instance, for the x coordinate, the operations we implemented were,

$$e(x) = 1 - \frac{\tilde{x}}{x}, \tag{5–1}$$

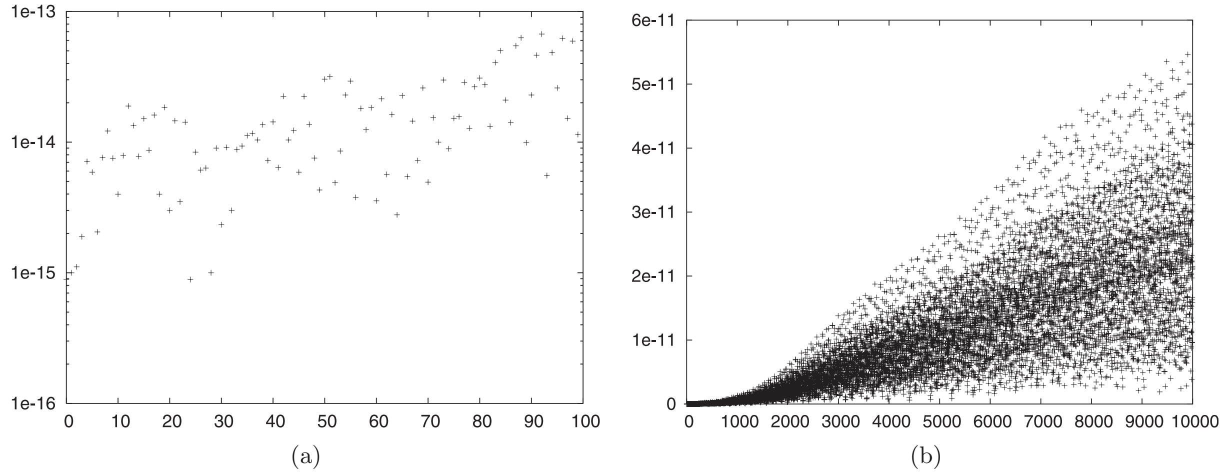

FIGURE 3. Error of a numerical integration of the RTBP. The horizontal axis denotes the number of intersections with the Poincaré section $z = 0$. (a) The first 100 intersections. (b) 10,000 intersections corresponding to a total integration time of 62837.969279 units. See the text for more details.

where x is the extended precision approximation and $\tilde{x}$ is the double precision result. All of the computations in (5–1) were done using double precision. Due to the high level of accuracy, and since we computed the relative error (in double precision), we recorded the results as multiples of the machine precision eps. In our case (an Intel-based computer), eps $= 2^{-52} \approx 2.22 \times 10^{-16}$. Moreover, when evaluating (5–1) (and only for this case) we forced the compiler to produce code such that the result of each arithmetic operation is stored in memory, this way we avoided using the extra precision available in the registers of the processor.

The results are shown in Table 1: the first column is the time and the remaining columns are the relative error for each coordinate, in multiples of eps. The "halved" factors (0.50, 1.50, etc.) are due to the fact that, due to the roundoff, the smallest (non-zero) number we can obtain from the subtraction in (5–1) is $\frac{1}{2}$eps.

A heuristic justification for this level of accuracy is the following. Let us first assume that the Taylor series has a general term that is decreasing in modulus. Then, it is well known that the sum of such a series can be calculated up to machine precision. Moreover, the propagation of the round off error in the recurrences used to derive the Taylor coefficients $x^{[j]}$ introduces an increasing relative error in them. Note that, this does not imply that the absolute error of the general term $x^{[j]}h^j$ is increasing and, in fact, if h is small enough, this absolute error is decreasing. Since we are adding a series with a general term that is decreasing in modulus, we do not need full accuracy of all the terms to achieve machine precision (roughly speaking, we only need an accuracy of a few digits for the last term of the series).

If the series is not decreasing we cannot, in principle, justify this phenomenon. However, we note that for h small enough, the series is decreasing from the first nonzero coefficient on.

5.1.2 Global error.

An interesting thing occurs in the behavior of the error in longer integrations. To examine this, we performed two tests.

The first test is based on a computation of the local error for a very long time span. We note that, in such a test, there is an extra source of error in the time parameterization of the orbit. Even if we force the same timestep in both integrations, the different level of precision introduces an extra time-shift that adds a small error to the comparison. For this reason, during the integration, we computed the sequence of intersections of the orbit with $z = 0$ (the initial condition is already given in this section). Then, for each intersection, we computed the sup norm of the difference between the double and extended arithmetic results to obtain the graphic shown in Figure 3. In this case, we followed a quasiperiodic orbit in a region that is almost completely filled by quasiperiodic orbits, each with their own frequencies. This implies that two neighboring orbits should separate at linear speed.

Since the Hamiltonian function H is constant on each orbit, a second test is simply to check for its preservation. Although the level of preservation of H does not need to be equal to the error of the integration, checking its preservation is a common test for a numerical integrator. We selected $\varepsilon_a = \varepsilon_r = 10^{-16}$, with an integration time of 10^6 units. A first version of the results is shown in Figure 4, where the horizontal axis denotes the time and the vertical axis is the difference between the actual and the initial value of H, in multiples of eps $\approx 2.22 \times 10^{-16}$.

	$\varepsilon = 10^{-14}$	$\varepsilon = 10^{-15}$	$\varepsilon = 10^{-16}$	$\varepsilon = 10^{-17}$	$\varepsilon = 10^{-18}$
-4	0	0	0	0	0
-3	45	2	7	5	6
-2	32,904	21,155	21,377	21,372	21,662
-1	772,723	745,668	760,755	768,334	777,760
0	1,970,571	2,084,758	2,134,729	2,157,287	2,174,276
1	765,519	744,438	760,183	767,596	776,776
2	32,444	21,174	21,576	21,696	21,949
3	42	6	5	3	5
4	0	0	0	0	0
τ	-6.0613	-0.9160	-0.1383	-0.0735	-0.3141

TABLE 2. Local variation of the energy for several error thresholds $\varepsilon_a = \varepsilon_r \equiv \varepsilon$, during 10^6 units of time. The first column denotes multiples of the machine precision eps and the remaining columns contain the number of integration steps for which the local variation of energy is equal to the multiple of eps in the first column. The last row is an statistical index to test for zero mean, see the text for details.

t	$e(x)$	$e(y)$	$e(z)$	$e(p_x)$	$e(p_y)$	$e(p_z)$
1.0000000000000000	0.50	-2.50	-1.00	-0.50	6.50	-5.50

TABLE 3. Local relative error (in multiples of the machine precision) for an orbit of the RTBP, after a unit of time, using `gmp` with 256 bits of mantissa. The meaning of the columns is the same as in Table 1. See the text for more comments.

Although this plot seems to indicate the presence of a bias in the values of H, we want to point out that the smallness of the drift in H compared to the length of the integration time does not allow us to consider this bias meaningful from a statistical point of view.

Let us discuss this point in detail. Let H_j be the value of H at step number j of the numerical integration and, instead of considering $H_j - H_0$, let us focus on the local variation $H_j - H_{j-1}$. In Table 2 we show a summary of the results for the same trajectory as in Figure 4, but for several local thresholds for the error. To do a standard statistical analysis, let us assume that the sequence of errors $H_j - H_{j-1}$ is given by a sequence of independent, identically distributed random variables; we are interested in knowing if its mean value is zero or not. Therefore, we apply the following test of significance of the mean. The null hypothesis assumes that the true mean is equal to zero. If k denotes a multiple of eps and ν_k the number of times that this deviation has occurred (in our case, $\nu_k = 0$ if $k > 4$), we define

$$n = \sum_{|k|\le 4} \nu_k,$$

$$m = \frac{1}{n} \sum_{|k|\le 4} k\nu_k,$$

$$s = \sqrt{\frac{1}{n^2} \sum_{|k|\le 4} (k-m)^2 \nu_k},$$

where s stands for the standard error of the sample mean m. Under the previous assumptions (independence and equidistribution of the observations), the value $\tau = \frac{m}{s}$ must behave as a $N(0,1)$ standard normal distribution. To test the null hypothesis (i.e., zero mean) with a confidence level of 95%, we check for the condition $|\tau| \le 1.96$. The last row of Table 2 shows the value of τ for the different integrations. It is clear that for $\varepsilon = 10^{-14}$ we must reject that the drift has zero mean. It is also clear that this hypothesis cannot be rejected in the other cases.

For the case $\varepsilon = 10^{-14}$ the main source of error is truncation that, from a statistical point of view, does

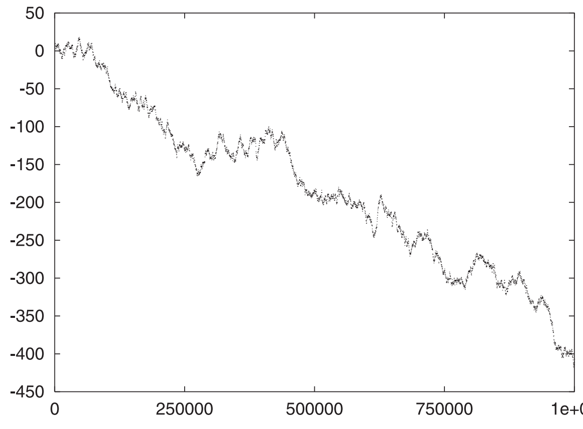

FIGURE 4. Long term behavior of the energy for local thresholds $\varepsilon_r = \varepsilon_a = 10^{-16}$. Horizontal axis: time. Vertical axis: relative variation of the value of the Hamiltonian, in multiples of the machine precision.

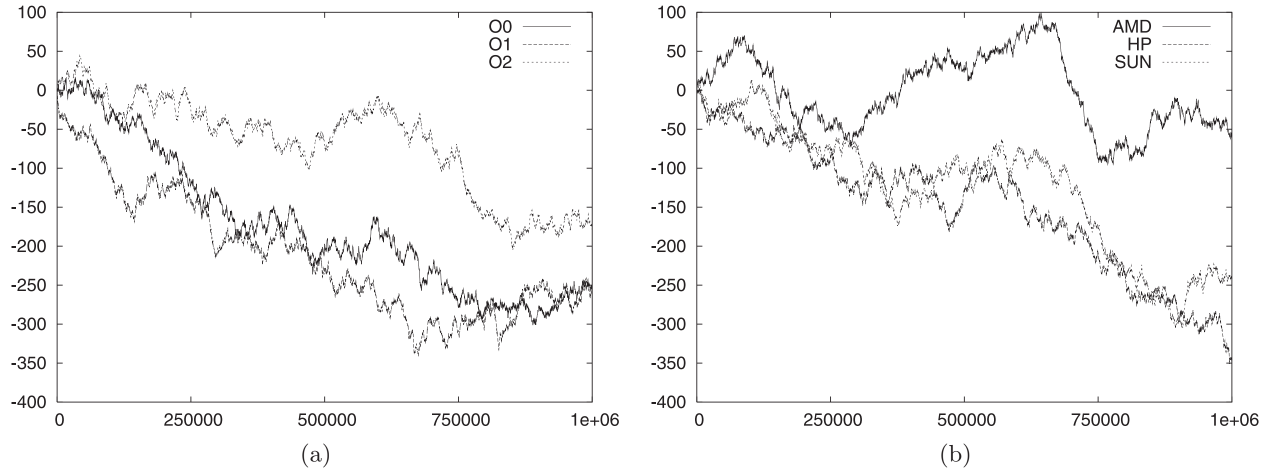

FIGURE 5. Long term behavior of the energy for local thresholds $\varepsilon_r = \varepsilon_a = 10^{-16}$. (a) Different optimization levels in an Intel processor (from top to bottom, -O2, -O0, and -O1 options). (b) Different processors (from top to bottom, AMD, Sun, and HP processors). See the text for more details.

not behave as if the drift has zero mean. When the local threshold is reduced, then the main source of error turns out to be the round off of the underlying arithmetic, which under the standard statistical tests,, looks like a zero mean random process.

A natural question is whether the Taylor method, with a sufficiently small local threshold (like 10^{-16} in the previous example), can compete with a symplectic integrator in the preservation of the geometrical structure of the phase space of a Hamiltonian system. From a local point of view, we note that the Taylor method can deliver machine precision so it is not possible to be "more symplectic." However, one has to be more careful when extending this reasoning to long term integrations, since it is possible that there exist little biases that are only visible in very long integrations. A deeper study is in progress.

5.1.3 On the influence of the underlying arithmetic.

For an example of the effect of the arithmetic, we show the different behavior of the energy. We used the same trajectory of the RTBP as before, and we computed the relative variation of the energy. The results for $\varepsilon_r = \varepsilon_a = 10^{-16}$ using standard double precision arithmetic on different hardware are shown in Figure 5. Both graphics show that the error behavior seems to be dominated by the "noise" of the floating point arithmetic. The differences between the AMD and Intel processors seem to come from hardware differences in the evaluation of the funcions ln and exp used in the step size control.

5.1.4 Extended precision calculations.

We use the same example as in the previous section. To generate code to be linked with the `gmp` library, we just need to pass the command line switch `-gmp` to Taylor.

As a first test, we computed the local error of a numerical integration of the RTBP, as done in Section 5.1.1. We selected a 256 bits mantissa (this means that the machine precision is eps $= 2^{-256} \approx 8.636168 \times 10^{-78}$) and the value 10^{-80} for both the relative and absolute error thresholds. By this method, we selected a step size near 0.2 and order 94. To obtain the exact solution, we used a mantissa of 512 bits and an error threshold of 10^{-155}. The local error of the solution after one unit of time (this required four calls to the Taylor integrator) is shown in Table 3. Comparing this with Table 1, we see that the relative error here is a little bit larger.

We also tested the variation of the value of the Hamiltonian for a long time integration, for different local thresholds. Figure 6 shows the difference between the initial value of the Hamiltonian and its value at each step of integration, for mantissas of 128 (left) and 256 (right) bits. The differences are shown in multiples of the machine precision of each arithmetic. The lower curve on these plots corresponds to the largest threshold ($\varepsilon_a = \varepsilon_r = 10^{-36}$ and $\varepsilon_a = \varepsilon_r = 10^{-75}$ for the left and right plot, respectively) where the main source of error is the truncation of the Taylor series. The remaining curves correspond to smaller thresholds for which the error mainly comes from the roundoff of the `gmp` arithmetic. We clearly see the different behavior of these two sources of error, as well as the drift introduced by the roundoff of the arithmetic.

We also tested the preservation of the Hamiltonian for a different extended arithmetic, the `qd` library. The results are shown in Figure 7. Again, we used the `dd_real` type (two doubles) for Figure 7(a) and `qd_real` type (four doubles) for (b). For this arithmetic, it does not make sense to use the machine precision as a unit

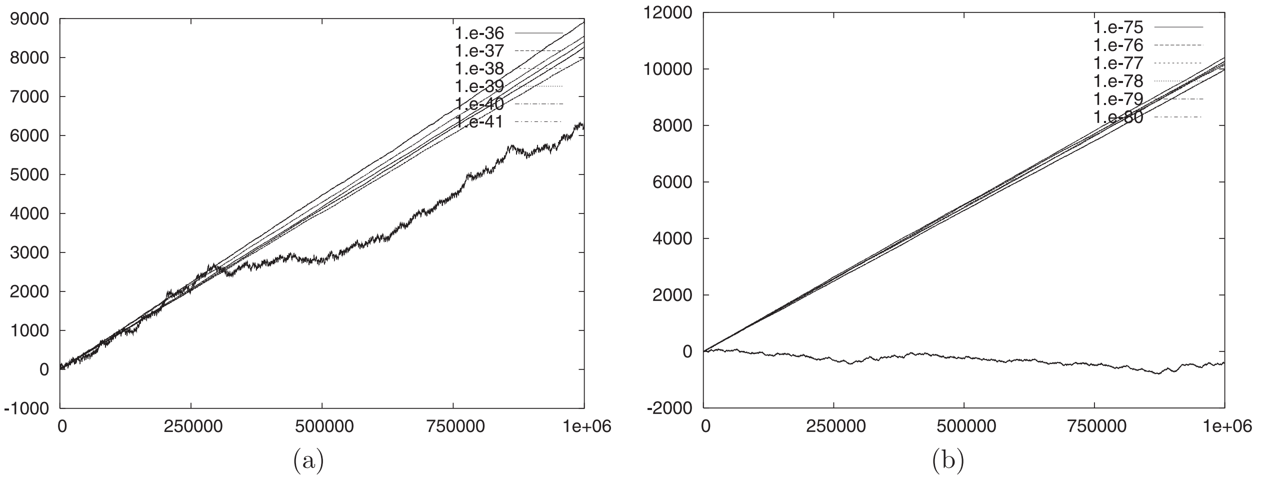

FIGURE 6. Long term behavior of the Hamiltonian for several integrations with `gmp` arithmetic. The horizontal axis displays the time and the vertical axis shows the variation of the Hamitonian with respect to its initial value. (a) Results for `gmp` arithmetic with a 128 bits mantissa, and several local thresholds $\varepsilon_a = \varepsilon_r = \varepsilon$ as shown in the graphic. (b) Results for `gmp` arithmetic with a 256 bits mantissa. See the text for more details.

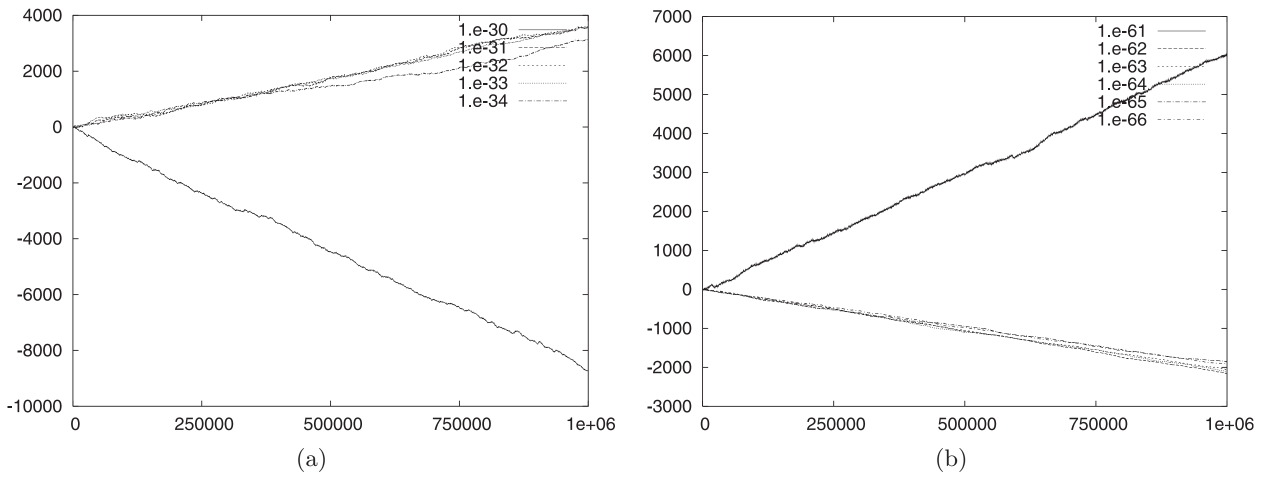

FIGURE 7. Long term behavior of the Hamiltonian for several integrations with `qd` arithmetic. The horizontal axis displays the time and the vertical axis shows the variation of the Hamitonian with respect to its initial value. (a) Results for `dd_real` arithmetic (nearly 32 decimal digits), and several local thresholds $\varepsilon_a = \varepsilon_r = \varepsilon$ as shown in the graphic. (b) Results for the `qd_real` arithmetic (nearly 64 decimal digits). See the text for more details.

for the error.[1] Hence, we simply multiplied the differences in the Hamiltonian by 10^{32} (`dd_real`) and 10^{64} (`qd_real`). In Figure 7(a), the bottom curve corresponds to $\varepsilon_a = \varepsilon_r = 10^{-30}$, the largest error threshold, where the effect of the truncation dominates the error. The remaining curves show the behavior of the round off error of the arithmetic. In Figure 7(b), the upper curve corresponds to the largest error threshold (in this case, $\varepsilon_a = \varepsilon_r = 10^{-61}$) and shows the effect of the truncation error. The remaining curves show the drift due to the round off of the arithmetic.

5.2 Computation of Small Quantities

Here we illustrate one of the uses of extended arithmetic: the computation of small quantities defined as the difference of very close numbers.

Let us consider the dynamical system

$$\ddot{x} - \sin(x) = \mu \sin\left(\frac{t}{\varepsilon}\right), \qquad (5\text{–}2)$$

where μ and ε are small parameters. When $\mu = 0$, then $x = 0$ and $x = 2\pi$ are hyperbolic points such that the

[1]A `dd_real` number is defined as the sum of two doubles. Therefore, the sum $1 + \varepsilon$ is always different from 1 as long as ε can be represented in a double.

stable and unstable manifolds of $x = 0$ coincide with the unstable and stable manifolds of $x = 2\pi$. For small $\mu > 0$, the points $x = 0$ and $x = 2\pi$ become hyperbolic periodic orbits and their invariant manifolds do not coincide but intersect transversally (for a general discussion and references see, for instance, [Delshams and Seara 97]).

It is usual to take the section $t = 0 \pmod{2\pi\varepsilon}$ so that (5–2) becomes a conservative two-dimensional map, with hyperbolic fixed points near $x = 0$ and $x = 2\pi$. Due to the symmetries of the problem, the unstable manifold of $x = 0$ transversally intersects the stable manifold of $x = 2\pi$ at $x = \pi$. Here we compute the intersection angle of these manifolds for $\mu = \varepsilon = 0.04$. The methods used here will be quite simple, since the only goal is to illustrate the capabilities provided by `taylor`. More sophisticated tools are described in [Delshams and Ramírez-Ros 99].

First, we used `taylor` to produce a time-stepper for Equation (5–2). Then, it was not difficult to write the two-dimensional map defined by the stroboscopic section $t = 0 \pmod{2\pi\varepsilon}$. The differential of this map is given by the numerical integration of the variational flow of Equation (5–2), again by means of the Taylor method. We ask `taylor` to call the `dq` library for the arithmetic, using the `qd_real` type (it provides nearly 64 decimal digits). Then, it was not difficult to code a Newton method to obtain the two hyperbolic fixed points near $x = 0$ and $x = 2\pi$, and the eigenvalues and eigenvectors of the differential of the map at these points (in fact, due to the symmetries of the problem, it is enough to perform these computations for one of them). The next step was to use the eigenvalues as a (linear) approximation to the manifolds and to grow them until they cut the line $x = \pi$. At this point we used two different procedures.

(a) We obtained a table of values of the two manifolds on a mesh of points x_j around $x = \pi$, and used numerical differentiation (with 3 steps of extrapolation) to approximate the intersection angle between the two manifolds.

(b) We computed an initial point p, at an approximated distance of 10^{-25} from the fixed point, that is mapped on the line $x = \pi$ after a certain number of iterates. Then we used the corresponding eigenvector at the fixed point as the tangent vector to the manifold at the initial point p. Then we iterated this point and the vector to obtain an approximation to the tangent vector of the manifolds at $x = \pi$.

The agreement between the two approaches allowed us to conclude that the intersection angle is

$$2.769781155284039017022 \times 10^{-17}$$

(we only write the digits common to the two approaches). We note that the computation is extremely simple provided one has an efficient procedure to integrate Equation (5–2) in extended precision.

5.3 Speed

There are plenty of numerical methods in the literature, and we do not plan to survey all of them but simply to compare our implementation of the Taylor method against a few well known methods. A shared characteristic of these methods is the free availability of an implementation program, which is the one we used. These programs are coded in FORTRAN 77, which adds an extra difficulty to the comparisons, since the observed differences may come from the different compilers. Therefore, to help the readers with these comparisons, our package includes the code for all of the examples, so that they can be run on any combination of compiler/computer for comparisons.

Our tests were done using a GNU/Linux workstation, with an Intel Pentium III processor running at 500 MHz. We used the GNU compilers `gcc` and `g77`, version 2.95.4.

The methods considered are `dop853`, an explicit Runge-Kutta code of order 8, and `odex`, an extrapolation method of varying order based on the Gragg-Bulirsh-Stoer algorithm. Both methods are documented in [Hairer et al. 00] and the code we used can be downloaded from http://www.unige.ch/math/folks/hairer/software.html. We note that the extrapolation methods are similar to the Taylor method in the sense that they can use arbitrarily high orders, so they are the natural methods against which to make comparisons.

For the tests, we used three vector fields: the RTBP, the Lorenz system, and a periodically forced pendulum. The equations for the Lorenz system are

$$\begin{aligned} \dot{x} &= 10(y - x), \\ \dot{y} &= x(28 - z) - y, \\ \dot{z} &= xy - \frac{8}{3}z, \end{aligned}$$

and the equations for the forced pendulum are

$$\begin{aligned} \dot{x} &= y, \\ \dot{y} &= -\sin(x) - 0.1y + 0.1\sin(t). \end{aligned}$$

The RTBP (see equations in (4–2)) has been coded as in Figure 2 so that, in all the cases, the vector field has the same number of operations. As before, we used the same formulas to code the vector fields for `dop853`, `odex`, and `taylor`.

One possibility for comparison, is to set the same threshold for all the methods and then compare the speeds. Note that, since the algorithms for the step size selection are completely different, one of them could be more "conservative" than the others and predict (unnecessarily) smaller step sizes so that the comparisons would

Lorenz								
dop583			odex			taylor		
ε	time	error	ε	time	error	ε	time	error
1.e-10	7.01	5.9e-03	1.e-10	8.73	6.2e-02	1.e-10	7.61	3.1e-06
1.e-11	8.91	5.0e-04	1.e-11	10.11	3.3e-03	1.e-11	7.99	4.4e-07
1.e-12	11.65	4.3e-05	1.e-12	11.54	2.0e-04	1.e-12	8.40	4.8e-08
1.e-13	15.31	3.7e-06	1.e-13	12.74	5.8e-06	1.e-13	8.80	3.3e-08
1.e-14	20.19	1.2e-06	1.e-14	15.04	6.4e-06	1.e-14	9.22	3.4e-08
1.e-15	26.76	8.9e-07	1.e-15	17.81	3.7e-06	1.e-15	9.75	9.2e-09
1.e-16	35.51	9.5e-07	1.e-16	50.47	1.9e-06	1.e-16	10.75	7.5e-09

Periodically forced pendulum								
dop583			odex			taylor		
ε	time	error	ε	time	error	ε	time	error
1.e-10	0.62	3.4e-11	1.e-10	1.49	6.9e-10	1.e-10	0.38	2.8e-13
1.e-11	0.78	3.6e-12	1.e-11	1.70	4.9e-11	1.e-11	0.42	2.1e-14
1.e-12	1.03	3.1e-13	1.e-12	1.93	1.7e-12	1.e-12	0.44	7.6e-15
1.e-13	1.38	2.7e-14	1.e-13	2.17	9.1e-14	1.e-13	0.47	1.2e-15
1.e-14	1.83	2.3e-15	1.e-14	2.36	4.4e-15	1.e-14	0.48	8.7e-16
1.e-15	2.45	2.1e-15	1.e-15	2.68	3.1e-15	1.e-15	0.52	5.8e-16
1.e-16	3.24	3.2e-15	1.e-16	3.09	1.1e-14	1.e-16	0.59	3.8e-16

RTBP								
dop583			odex			taylor		
ε	time	error	ε	time	error	ε	time	error
1.e-10	1.43	1.1e-09	1.e-10	1.74	1.8e-09	1.e-10	1.68	6.2e-12
1.e-11	1.84	9.4e-11	1.e-11	2.02	9.2e-11	1.e-11	1.86	4.6e-13
1.e-12	2.44	8.6e-12	1.e-12	2.43	2.4e-11	1.e-12	2.08	4.4e-14
1.e-13	3.24	8.0e-13	1.e-13	2.74	3.7e-13	1.e-13	2.27	7.2e-15
1.e-14	4.32	7.5e-14	1.e-14	3.14	1.5e-13	1.e-14	2.50	4.2e-15
1.e-15	5.73	9.9e-15	1.e-15	3.71	2.4e-13	1.e-15	2.82	1.7e-15
1.e-16	7.63	2.0e-15	1.e-16	4.85	1.3e-13	1.e-16	3.26	5.8e-15

TABLE 4. Speed comparison between `dopri853`, `odex`, and `taylor`. The selected threshold for the error is ε (both relative and absolute thresholds have been set to the same value), computer time is given in seconds, and the error is the absolute error at the end point of the integration. To have a measurable computer time, we repeated the same integration 1,000 times. See the text for more details.

be meaningless. For this reason we proceeded in the following way: given an initial condition, we computed the corresponding orbit during, say, 16 units of time and compared the final point with the true value to obtain the real absolute error.[2] In Table 4 we show the computer time and final error for the three methods, using different thresholds for the step size control. To have a measurable running time, the program repeated the same calculation 1,000 times.

Therefore, we ignore the column labelled ε (the error threshold used for the step size control), and we only compare the computing time needed to achieve a prescribed accuracy (this is equivalent to comparing the accuracy obtained for a fixed computing time). The results clearly show the effectiveness of the Taylor method for these examples.

[2]The true value was obtained from a Taylor method integration using the `gmp` arithmetic with mantissas of 128 and 256 bits.

5.3.1 A simple comparison with ADOL-C.

ADOL-C is a public domain package for automatic differentiation. The main differences between the automatic differentiation of our package and ADOL-C are:

(a) ADOL-C is a general purpose package, while `taylor` is specifically designed for the numerical integration of ODEs.

(b) The input of ADOL-C is a C/C++ function, with some restrictions in the grammar used, while `taylor` has its own input grammar, that is a little bit more restrictive.

(c) ADOL-C does not include code for the step size control. This means that ADOL-C can only be used to generate the Taylor coefficients and the user must supply code for the order and step size control.

For this reason, we only tested the speed of the generation of the Taylor coefficients.

	degree	Lorenz	Pendulum	RTBP
ADOL-C	40	92.82	140.57	403.22
Taylor	40	3.59	3.43	14.75
ADOL-C	20	24.44	34.82	87.99
Taylor	20	1.13	1.07	4.65
ADOL-C	10	9.13	11.58	26.20
Taylor	10	0.41	0.39	1.62

TABLE 5. Time (in seconds) to compute 100,000 times the jet of derivatives for the Lorenz system, a periodically forced pendulum, and the RTBP.

As before, the tests were done on an Intel Pentium III running at 500 MHz, using ADOL-C version 1.8.7. The examples considered were the RTBP, the Lorenz system, and a periodically forced pendulum. To measure the time, we computed the jet of derivatives 100,000 times. The results are displayed in Table 5, and clearly show the efficiency of `taylor`.

6. CONCLUSIONS

In this paper we discuss a new publicly available implementation of the classical Taylor method for the numerical solution of ODEs. This program reads the differential equations from a file which contains the differential equations in its natural mathematical form and outputs a complete Taylor integrator (including adaptive selection of degree and step size) for the given system. One of the strong points of the package is its support for extended precision arithmetic.

The package has been tested against freely available implementations of two well-known numerical integrators. We do not claim that the results from these tests can be extrapolated to any example, but simply that `taylor` can be very competitive in many situations, especially when high accuracy is needed.

ACKNOWLEDGMENTS

The authors want to thank R. Broucke, C. Simó, and, especially, R. de la Llave for their comments. This work has been supported by the Comisión Conjunta Hispano Norteamericana de Cooperación Científica y Tecnológica. The second author has also been supported by the MCyT/FEDER grant BFM2003-07521-C02-01, the Catalan CIRIT grant 2001SGR–70, and DURSI.

REFERENCES

[Aho et al. 86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley Publishing Company, 1986

[Bailey et al. 05] D. H. Bailey, Y. Hida, K. Jeyabalan, X. S. Li, and B. Thompson. "High-Precision Software Directory." Available from World Wide Web (http://www.nersc.gov/∼dhbailey/mpdist/mpdist.html), 2005.

[Barton 80] D. Barton. "On Taylor Series and Stiff Equations." *ACM Trans. Math. Software* 6:3 (1980), 280–294.

[Barton et al. 70] 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 (1970), 243–248.

[Beda et al. 59] 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).

[Berz et al. 96] M. Berz, C. Bischof, G. F. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools.* Philadelphia, Penn.: SIAM, 1996.

[Bischof et al. 92] C. H. Bischof, A. Carle, G. F. Corliss, and A. Griewank. "ADIFOR: Automatic Differentiation in a Source Translation Environment." In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, edited by Paul S. Wang, pp. 294–302. New York: ACM Press, 1992.

[Briggs 02] K. Briggs. "The Doubledouble Homepage." Available from World Wide Web (http://members.lycos.co.uk/keithmbriggs/doubledouble.html), 2002.

[Broucke 71] R. Broucke. "Solution of the N-Body Problem with Recurrent Power Series." *Celestial Mech.* 4:1 (1971), 110–115.

[Chang and Corliss 94] Y. F. Chang and G. F. Corliss. "ATOMFT: Solving ODEs and DAEs Using Taylor Series." *Computers and Mathematics with Applications* 28 (1994), 209–233.

[Corliss 95] G. F. Corliss. "Guaranteed Error Bounds for Ordinary Differential Equations." In *Theory of Numerics in Ordinary and Partial Differential Equations*, edited by M. Ainsworth, J. Levesley, W. A. Light, and M. Marletta, pp. 1–75. Oxford: Oxford University Press, 1995.

[Corliss and Chang 82] G. F. Corliss and Y. F. Chang. "Solving Ordinary Differential Equations Using Taylor Series." *ACM Trans. Math. Software* 8:2 (1982), 114–144.

[Corliss et al. 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)*, pp. 114–125. Berlin: Springer, 1997.

[Delshams and Ramírez-Ros 99] A. Delshams and R. Ramírez-Ros. "Singular Separatrix Splitting and the Melnikov Method: An Experimental Study." *Exp. Math.* 8:1 (1999), 29–48.

[Delshams and Seara 97] A. Delshams and T. M. Seara. “Splitting of Separatrices in Hamiltonian Systems with One and a Half Degrees of Freedom.” *Math. Phys. Electron. J.* 3 (1997), Paper 4, 40 pp.

[Gibbons 60] A. Gibbons. “A Program for the Automatic Integration of Differential Equations Using the Method of Taylor Series.” *Comp. J.* 3 (1960), 108–111.

[GMP 05] GNU Multiple Precision Library. “GMP Arithmetic without Limitations.” Available from World Wide Web (http://www.swox.com/gmp/), 2005.

[Griewank 00] A. Griewank. *Evaluating Derivatives.* Philadelphia, Penn.: SIAM, 2000.

[Griewank and Corliss 91] A. Griewank and G. F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.* Philadelphia, Penn.: SIAM, 1991.

[Hairer et al. 00] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I. Nonstiff Problems*, Springer Series in Computational Mathematics, 8, Second revised edition. Berlin: Springer-Verlag, 2000.

[Hoefkens 01] J. Hoefkens. *Rigorous Numerical Analysis with High Order Taylor Methods.* PhD. diss., Michigan State University, 2001.

[Irvine and Savageau 90] D. H. Irvine and M. A. Savageau. “Efficient Solution of Nonlinear Ordinary Differential Equations Expressed in S-System Canonical Form.” *SIAM J. Numer. Anal.* 27:3 (1990), 704–735.

[Jalbert and Zahar 85] 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), Congressus Numerantium* 46 (1985), 347–358.

[Kirlinger and Corliss 92] G. Kirlinger and G. F. Corliss. “On Implicit Taylor Series Methods for Stiff ODEs.” In *Computer Arithmetic and Enclosure Methods*, edited by L. Atanassova and J. Herzberger, pp. 371–379. Amsterdam: North-Holland, 1992.

[Martínez and Simó 99] R. Martínez and C. Simó. “Simultaneous Binary Collisions in the Planar Four-Body Problem.” *Nonlinearity* 12:4 (1999), 903–930.

[Meyer and Hall 92] K. R. Meyer and G. R. Hall. *Introduction to Hamiltonian Dynamical Systems and the N-Body Problem.* New York: Springer, 1992.

[Moore 66] R. E. Moore. *Interval Analysis.* Englewood Cliffs, N.J.: Prentice-Hall, 1966.

[Nedialkov et al. 99] 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 (1999), 21–68.

[Rall 81] L. B. Rall. *Automatic Differentiation: Techniques and Applications*, Lecture Notes in Computer Science, 120. Berlin: Springer Verlag, 1981.

[Savageau and Voit 87] M. A. Savageau and E. O. Voit. “Recasting Nonlinear Differential Equations as S-Systems: A Canonical Nonlinear Form.” *Math. Biosci.* 87:1 (1987), 83–115.

[Simó 01] C. Simó. “Global Dynamics and Fast Indicators.” In *Global Analysis of Dynamical Systems*, edited by H. W. Broer, B. Krauskopf, and G. Vegter, pp. 373–389. Bristol: IOP Publishing, 2001.

[Simó 02] C. Simó. “Dynamical Properties of the Figure Eight Solution of the Three-Body Problem.” In *Celestial Mechanics (Evanston, IL, 1999)*, edited by A. Chenciner, R. Cushman, C. Robinson, and Z. Xia, pp. 209–228, *Contemporary Mathematics*, 292. Providence, RI: Amer. Math. Soc., 2002.

[Simó and Valls 01] 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.” *Nonlinearity* 14:4 (2001), 1707–1760.

[Steffensen 56] J. F. Steffensen. “On the Restricted Problem of Three Bodies.” *Danske Vid. Selsk. Mat.-Fys. Medd.* 30:18 (1956), 17.

[Steffensen 57] J. F. Steffensen. “On the Problem of Three Bodies in the Plane.” *Mat.-Fys. Medd. Danske Vid. Selsk.* 31:3 (1957), 18.

[Szebehely 67] V. Szebehely. *Theory of Orbits.* New York: Academic Press, 1967.

[Wengert 64] R. E. Wengert. “A Simple Automatic Derivative Evaluation Program.” *Comm. ACM* 7:8 (1964), 463–464.

[Zou and Jorba 01] M. Zou and A. Jorba. “Taylor 1.4.” Available from World Wide Web in the United States (http://www.ma.utexas.edu/∼mzou/taylor/) and in Europe (http://www.maia.ub.es/∼angel/taylor/), 2001

Àngel Jorba, Departament de Matemàtica Aplicada i Anàlisi, Universitat de Barcelona, Gran Via 585, 08007 Barcelona, Spain (angel@maia.ub.es)

Maorong Zou, Department of Mathematics, The University of Texas at Austin, Austin, TX 78712-1082 (mzou@math.utexas.edu)

Received January 29, 2004; accepted October 18, 2004.