Software for the numerical integration of ODE by means of high-order Taylor methods (I)

Àngel Jorba
angel@maia.ub.es

University of Barcelona

Advanced Course on Long Term Integrations

Outline

1. Introduction
2. A software implementation
   - Step size control

Problem: find a function $x : [a, b] \to \mathbb{R}^m$ such that

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

Taylor method:

$$\begin{array}{rcl} x_0 & = & x(a), \\ x_{m+1} & = & x_m + x'(t_m)h + \cdots + \dfrac{x^{(p)}(t_m)}{p!}h^p, \end{array}$$

for $m = 0, \ldots, N - 1$.

A first approach is to compute the derivatives by means of the direct application of the chain rule,

$$x'(t_m) = f(t_m, x(t_m)),$$

$$x''(t_m) = f_t(t_m, x(t_m)) + f_x(t_m, x(t_m))x'(t_m),$$

and so on.

These expressions have to be obtained explicitly for each equation we want to integrate.

Example: Van der Pol equation.

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

The *n*th order Taylor method for the initial value problem is

$$\begin{array}{rcl} x_{m+1} & = & x_m + x'_m h + \frac{1}{2!}x''_m h^2 + \cdots + \frac{1}{n!}x_m^{(n)} h^n, \\ y_{m+1} & = & y_m + y'_m h + \frac{1}{2!}y''_m h^2 + \cdots + \frac{1}{n!}y_m^{(n)} h^n. \end{array}$$

There are several ways of obtaining the derivatives of the solution w.r.t. time.

A standard way is to take derivatives on the differential equation,

$$\begin{aligned}
x'' &= (1-x^2)y - x,\\
y'' &= -2xy^2 + [(1-x^2)^2 - 1]y - x(1-x^2),\\
x''' &= -2xy^2 + [(1-x^2)^2 - 1]y - x(1-x^2),\\
y''' &= 2y^3 - 8x(1-x^2)y^2 + [4x^2 - 2 + (1-x^2)^3]y\\
&\quad +x[1-(1-x^2)^2],\\
&\vdots
\end{aligned}$$

Note how the expressions become increasingly complicated.

These closed formulas allow for the evaluation of the derivatives at any point, so they have to be computed only once (for each vector field).

These closed formulas allow for the evaluation of the derivatives at any point, so they have to be computed only once (for each vector field).

For a long time integration,

- the effort needed to produce these formulas is not relevant,
- the effort to *evaluate them is very relevant*

There is an alternative procedure to compute derivatives:
Automatic differentiation

Automatic differentiation is a recursive algorithm to evaluate the derivatives of a closed expression on a given point.

Automatic differentiation does not produce closed formulas for the derivatives.

A good reference book is:
A. Griewank: *Evaluating derivatives*, SIAM (2000).
ISBN: `0-89871-284-X`

Assume that a is a real function of a real variable.

Definition

The normalized j-th derivative of a at the point t is

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

Normalized derivatives are the coefficients of the power expansion of the solution.

Lemma

If $a(t) = b(t)c(t)$, *then* $a^{[n]}(t) = \sum_{i=0}^{n} b^{[n-i]}(t)c^{[i]}(t)$.

Proof.

It follows from Leibniz formula:

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

□

Lemma

If $a(t) = b(t)c(t)$, *then* $a^{[n]}(t) = \sum_{i=0}^{n} b^{[n-i]}(t)c^{[i]}(t)$.

Proof.

It follows from Leibniz formula:

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

$\square$

If we know the (normalized) derivatives of b and c at t, up to order n, we can compute the nth derivative of a at t.

Example: Van der Pol equation.

$$\left.\begin{array}{rcl} x' &=& y, \\ y' &=& (1-x^2)y - x. \end{array}\right\} \qquad \left.\begin{array}{rcl} u_1 &=& x, \\ u_2 &=& y, \\ u_3 &=& u_1 u_1, \\ u_4 &=& 1 - u_3, \\ u_5 &=& u_4 u_2, \\ u_6 &=& u_5 - u_1, \\ x' &=& u_2, \\ y' &=& u_6. \end{array}\right\}$$

$$\left.\begin{array}{rcl} u_1 &=& x, \\ u_2 &=& y, \\ u_3 &=& u_1 u_1, \\ u_4 &=& 1 - u_3, \\ u_5 &=& u_4 u_2, \\ u_6 &=& u_5 - u_1, \\ x' &=& u_2, \\ y' &=& u_6. \end{array}\right\} \qquad \left.\begin{array}{rcl} u_1^{[n]} &=& x^{[n]}, \\ u_2^{[n]} &=& y^{[n]}, \\ u_3^{[n]} &=& \displaystyle\sum_{i=0}^{n} u_1^{[n-i]} u_1^{[i]}, \\ u_4^{[n]} &=& -u_3^{[n]} \quad (\text{if } n > 0), \\ u_5^{[n]} &=& \displaystyle\sum_{i=0}^{n} u_4^{[n-i]} u_2^{[i]}, \\ u_6^{[n]} &=& u_5^{[n]} - u_1^{[n]}, \\ x^{[n+1]} &=& \dfrac{1}{n+1} u_2^{[n]}, \\ y^{[n+1]} &=& \dfrac{1}{n+1} u_6^{[n]}. \end{array}\right\}$$

The recurrence can be applied up to a suitable order p.

It is not necessary to select the value p in advance.

If the functions b and c are of class C^n, we have

1. If $a(t) = b(t) \pm c(t)$, then $a^{[n]}(t) = \sum_{i=0}^{n} b^{[i]}(t) \pm c^{[i]}(t)$.
2. If $a(t) = b(t)c(t)$, then $a^{[n]}(t) = \sum_{i=0}^{n} b^{[n-i]}(t)c^{[i]}(t)$.
3. If $a(t) = \dfrac{b(t)}{c(t)}$, then

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

4. If $\alpha \in \mathbb{R} \setminus \{0\}$ and $a(t) = b(t)^{\alpha}$, then

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

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

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

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

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

Many ODE can be “decomposed” as a sequence of binary operations, so it is possible to produce the jet of derivatives of the solution at a given point, in a recursive way.

This includes ODEs involving special functions. We will see some examples later on.

Next step is to find an order p and a step size h such that

- the error is smaller than ε.
- the total number of operations is minimal.

Lemma (C. Simó, 2001)

Assume that the function $h \mapsto x(t_n + h)$ is analytic on a disk of radius ρ_m. Let A_m be a positive constant such that

$$|x_m^{[j]}| \leq \frac{A_m}{\rho_m^j}, \qquad \forall j \in \mathbb{N}.$$

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

$$h_m = \frac{\rho_m}{e^2}, \qquad p_m = -\frac{1}{2}\ln\left(\frac{\varepsilon}{A_m}\right) - 1$$

Proof.

The error is of the order of the first neglected term $E \approx A \left(\frac{h}{\rho}\right)^{p+1}$.

To obtain an error of order ε we select $h \approx \rho \left(\frac{\varepsilon}{A}\right)^{\frac{1}{p+1}}$.

The operations to obtain the jet of is $O(p^2) \approx c(p+1)^2$.

So, the number of floating point operations per unit of time is given, in order of magnitude, by $\phi(p) = \frac{c(p+1)^2}{\rho\left(\frac{\varepsilon}{A}\right)^{\frac{1}{p+1}}}$.

Solving $\phi'(p) = 0$, we obtain $p = -\frac{1}{2}\ln\left(\frac{\varepsilon}{A}\right) - 1$.

We use this order with the largest step size that keeps the local error below ε: inserting this value of p in the formula for h we have

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

$\square$

Dangerous step sizes

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

We are interested in computing $x(10) = \exp(-10) \approx 0.0000454$.

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.

From a numerical point of view, this is a disaster!

High accuracy and varying order

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.

We will present a concrete implementation of the Taylor method.

The software has been released under the GNU Public License, so everybody is free to use it, to modify it and to redistribute it.

It has been written to run under the GNU/Linux operating system.

The software can be retrieved from
`http://www.maia.ub.es/~angel/taylor/`

For more information:
A. Jorba, M.Zou, *A software package for the Numerical Integration of ODEs by means of High-Order Taylor Methods*, Experimental Mathematics 14:1 pp. 99–117 (2005).

The package reads a system of ODEs in a quite natural form, and it can output several ANSI C routines:

- a routine that computes the jet of derivatives of the solution (up to an order given at runtime),
- routines to estimate an order and, from the jet of derivatives, a suitable step size (for a local error below given thresholds),
- a routine to use the previous data to advance the solution in one time step.

It supports different arithmetics (i.e., extended precision), including user-defined types.

In the next slides we will explain the algorithms used by `taylor`.

Taylor supports a tiny language using three kind of statements:

- `extern MY_FLOAT var;`
- `id = expr;`
- `diff(v, t) = expr;`

where t is the independent variable and v is a state variable.

We use the first statement to declare external variables. These declarations re copied to the output routine without modification. External variables are treated as constants.

We use the second statement to define a constant, or a shorthand notation for a complex expression used in the differential equations. It is normally used to help the translater to factor out common expressions, which in turn, may generate smaller and faster codes.

Expressions are made from numbers, the time variable, the state variables, external variables, elementary functions sin, cos, tan, arctan, sinh, cosh, tanh, $\sqrt{\ }$, exp, and log, using the four arithmetic operators, $(.)^{(\cdot)}$ 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.

The translation process consists of several phases each of which passes its output to the next phase.

The first phase is the lexical phase. Here characters from the input stream is grouped into lexical units called tokens by a scanner (lexical anaylizer). Regular expressions are used to define tokens recognized by the scanner. The scanner is implemented as a finite state automata. The actual code for the scanner is generated by *Lex*. The input to Lex is a file containing definitions of tokens using regular expressions. The output is a C procedure `yylex()` that is called repeatedly by the parser to fetch the next token from the input stream.

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 prescribed set of grammatical rules. The output of the parser the parse tree, a graphical representation of the input. Our parser is generated by *Yacc*. The input to Yacc is a file containting a set of grammar rules. The output of Yacc is a procedure `yyparse()` which is used to generate the parse tree.

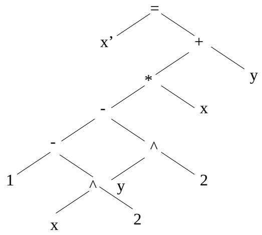
To illustrate the parsing process, let’s look at this example:

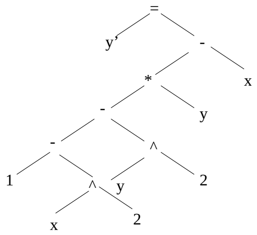
$$\begin{aligned} x' &= x(1-x^2-y^2)+y, \\ y' &= y(1-x^2-y^2)-x. \end{aligned}$$

The scanner breaks the input the following list of tokens:

```
x ’ = x * ( 1 - x ˆ 2 - y ˆ 2 ) + y ; y ’ =  x * ( 1 - x ˆ 2 - y ˆ 2 )
```

A graphical representation of the parsed input could be:





The next phase is optimization. The crucial tasks are:

- Identify and mark constant expressions (constant expressions are easy to handle when computing derivatives...)
- Eliminate common subexpressions. Algebraic simplifications is 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 all different.
- 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.

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

- Expand power function as a series of products. This procedure is controlled by the `-expandpower` command line switch. For example, $y = x^7$ will be replaced by $u = x * x, v = u * u, w = u * v, y = x * w$ if taylor is invoked with the option `-expandpower 7`. One reason to expand a power function using products is to avoid singularities (at the origin).
- The flag `-sqrt` forces the parser to treat exponents like $n/2$ as the nth power of a square root (instead of using log and exp).

Step size control

We use and absolute (ε_a) and a relative (ε_r) tolerance.

We define

$$\varepsilon_m = \begin{cases} \varepsilon_a & \text{if} \quad \varepsilon_r \|x_m\|_\infty \le \varepsilon_a, \\ \varepsilon_r & \text{otherwise}, \end{cases} \qquad p_m = \left\lceil -\frac{1}{2} \ln \varepsilon_m + 1 \right\rceil .$$

where $\lceil . \rceil$ stands for the ceiling function.

To derive the step size, we will also distinguish the same two cases as before.

If $\varepsilon_r \|x_m\|_\infty \le \varepsilon_a$, we define

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

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

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

In any case, we estimate the radius of convergence as the minimum of the last two terms,

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

and the estimated time step is

$$h_m = \frac{\rho_m}{e^2}.$$

Lemma

With the previous notations and definitions:

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*

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

Lemma

With the previous notations and definitions:

1. *If* $\varepsilon_r \|x_m\|_\infty \le \varepsilon_a$, *we have*

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

2. *If* $\varepsilon_r \|x_m\|_\infty > \varepsilon_a$, *we have*

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

Hence, 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.

Fact

If the solution is entire, the Cauchy bounds are far from optimal.

Then, the computed values for p_m and h_m still satisfy the accuracy requirements but they do not need to be the ones that minimise the global number of operations.

The previous formulas have been used for the first order and time step control, but with a safety factor: Instead of using

$$h_m = \frac{\rho_m}{e^2}$$

we use

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

For instance, for $p_m = 8$ the safety factor is 0.90 and for $p_m = 16$ is 0.95. Those are typical safety factors found in the literature.

The code provides a second step size control, which is a minor correction of the previous one.

The idea is to avoid too large step sizes that could lead to cancellations when adding the Taylor series.

A natural solution is to look for an 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 a very drastic (and unnecessary) step reductions.

Therefore, we have used a weaker criterion.

Let $\bar{h}_m$ be the step size control obtained previously. 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.

The generated code allows for used-defined order and step size controls.