# Developing Tailored Software for Specific Problems

Àngel Jorba(1) and Maorong Zou(2)

(1) Institut de Matemàtica de la Universitat de Barcelona – IMUB
(2) University of Texas at Austin

June 22nd, 2009

In this talk we will discuss the development of numerical integrations for Ordinary Differential Equations (ODEs).

In this talk we will discuss the development of numerical integrations for Ordinary Differential Equations (ODEs).

Our approach is to write a (public domain) package that, given our set of differential equations, writes (in C) the numerical integration

In this talk we will discuss the development of numerical integrations for Ordinary Differential Equations (ODEs).

Our approach is to write a (public domain) package that, given our set of differential equations, writes (in C) the numerical integration

*"Programs making programs, what a obscenity"*, C3PO (Star Wars)

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 a closed formula for the value of the derivative at any point $x$.

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

Assume that $a$ is a (smooth) 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 Taylor series of $a$ at $t$.

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:

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

□

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

Example: Van der Pol equation.

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

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

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

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

**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

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

**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. Note that this includes ODEs involving special functions.

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

- the error is smaller than $\varepsilon$.
- the total number of operations is minimal.

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

- the error is smaller than $\varepsilon$.
- 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 $\rho_m$. 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 $\varepsilon$ 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 $\varepsilon$ 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 $\varepsilon$: inserting this value of $p$ in the formula for $h$ we have

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

□

## 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.

To use effectively the Taylor method, it has to be coded for each vectorfield.

Coding can be done

- by a human...

To use effectively the Taylor method, it has to be coded for each vectorfield.

Coding can be done

- by a human...
- ... or by a computer program.

To use effectively the Taylor method, it has to be coded for each vectorfield.

Coding can be done

- by a human...
- ... or by a computer program.

In this last case, the program must produce efficient code.

We will present a concrete implementation of the Taylor method, written by Maorong Zou (U. Texas) and A.J.

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, $\sqrt{\phantom{x}}$, 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.

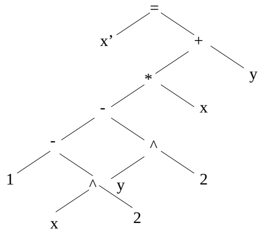
To illustrate the parsing process, let's look at this example:

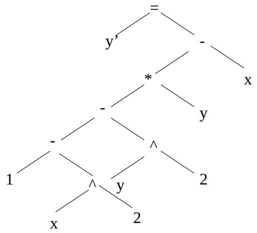
$$\begin{aligned} x' &= x(1-x^2-y^2)+y, \\ y' &= y(1-x^2-y^2)-x. \end{aligned}$$

The scanner breaks the input the following list of tokens:

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

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 $n$th power of a square root (instead of using log and exp).

## Step size control

We use and absolute ($\varepsilon_a$) and a relative ($\varepsilon_r$) 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* $\varepsilon_r \|x_m\|_\infty \le \varepsilon_a$, *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*

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

**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.

Remark

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

*In this case, 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

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

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.

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

It installs in a GNU/Linux system.

It requires the packages `flex` and `bison`.

Now we will see how it works.

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

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

To produce a numerical integrator for this vector field, assume that the previous code is in the file `rtbp.in`
Then, you can type

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

to produce two files:

- `taylor_rtbp.c`: The time stepper
- `taylor.h`: Header to define the arithmetic used

```
Usage: ./taylor
 [-name ODE_NAME]
 [-o outfile]
 [-doubledouble | -qd_real | -dd_real | -gmp -gmp_precision PRECISION]
 [-main | -header | -jet | -main_only]
 [-step STEP_CONTROL_METHOD]
 [-u | -userdefined] STEP_SIZE_FUNCTION_NAME ORDER_FUNCTION_NAME
 [-f77]
 [-sqrt]
 [-headername HEADER_FILE_NAME]
 [-debug] [-help] [-v]  file
```

Main C call:

```
int taylor_step_ODE_NAME(MY_FLOAT *time,
                         MY_FLOAT *xvars,
                         int      direction,
                         int      step_ctrl_method,
                         double   log10abserr,
                         double   log10relerr,
                         MY_FLOAT *endtime,
                         MY_FLOAT *stepused,
                         int      *order)
```

Main Fortran 77 call:

```
void taylor_f77_ODE_NAME__(MY_FLOAT *time,
                           MY_FLOAT *xvars,
                           int      *direction,
                           int      *step_ctrl_method,
                           double   *log10abserr,
                           double   *log10relerr,
                           MY_FLOAT *endtime,
                           MY_FLOAT *stepused,
                           int      *order,
                           int      *flag)
```

Let us see an example of numerical integration by selecting the initial condition `x1=-0.45`, `x2=0.80`, `x3=0.00`, `x4=-0.80`, `x5=-0.45` and `x6=0.58`.

We will perform a numerical integration with the standard double precision of the computer, for 1 unit of time.

As a first test, we will check the preservation of the Hamiltonian.

We have coded a small main program that uses this initial condition to call the Taylor integrator till the time has advanced in one unit.

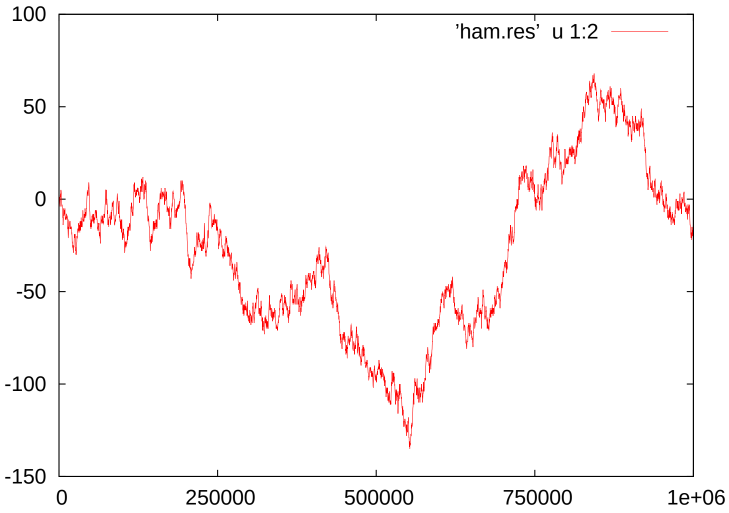
Next run takes $\varepsilon_a = \varepsilon_r = 10^{-16}$

```
value of H at the initial condition:  -1.3362071584596453
numerical integration starts...
    0.24011923241902   20  -1.00000
    0.49521588761001   20   0.00000
    0.76536594703474   20   0.00000
    1.00000000000000   20  -1.00000
```

Next run takes $\varepsilon_a = \varepsilon_r = 10^{-16}$

```
value of H at the initial condition:  -1.3362071584596453
numerical integration starts...
    0.24011923241902   20  -1.00000
    0.49521588761001   20   0.00000
    0.76536594703474   20   0.00000
    1.00000000000000   20  -1.00000
```

Is there a drift in the energy?


'ham.res' u 1:2
100
50
0
-50
-100
-150
0
250000
500000
750000
1e+06

It is possible to check (statistically) that the variation of the energy behaves like a random walk.

Let $H_j$ be the value of $H$ at the step number $j$ of the numerical integration and, instead of consider $H_j - H_0$, let us focus on the local variation $H_j - H_{j-1}$.

| | $\varepsilon = 10^{-14}$ | $\varepsilon = 10^{-15}$ | $\varepsilon = 10^{-16}$ | $\varepsilon = 10^{-17}$ | $\varepsilon = 10^{-18}$ |
|---|---|---|---|---|---|
| -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 |

Local variation of the energy during $10^6$ units of time. The first column denotes multiples of the machine precision 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.

To do an 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, and we are interested in knowing if its mean value is zero or not.

Therefore, we will 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 $\nu_k$ the number of times that this deviation has occurred (in our case, $\nu_k = 0$ if $k > 4$), we define

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

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 have to check for the condition $|\tau| \leq 1.96$.

| | $\varepsilon = 10^{-14}$ | $\varepsilon = 10^{-15}$ | $\varepsilon = 10^{-16}$ | $\varepsilon = 10^{-17}$ | $\varepsilon = 10^{-18}$ |
|---|---|---|---|---|---|
| -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 |
| $\tau$ | -6.0613 | -0.9160 | -0.1383 | -0.0735 | -0.3141 |

The last row shows the value of $\tau$ for the different integrations. It is clear that for $\varepsilon = 10^{-14}$ we must reject that the drift has zero mean, and it is also clear that this hypothesis cannot be rejected in the other cases.

## A comparison with a Runge-Kutta-Fehlberg 7-8

We ask the rk78 for an accuracy of $10^{-16}$.

The error in $H$ after $10^6$ units of time, in number of multiples of the machine epsilon, is $-13412$ for the rk78, and $-217$ for the Taylor method.

The time taken for the rk78 was of 9m and 34s, while the Taylor method needed 4m and 4s.

If we ask the rk78 for an accuracy of $10^{-14}$ then the time taken goes down to 4m and 58s, but the final error is 649368 times the epsilon of the machine (that is, $1.44 \times 10^{-10}$).

This is a second benchmark using the standard quadruple precision of a HP 9000/712 computer, with a 100 MHz PA-RISC 1.1 processor.

We have used the vector field of the Restricted Three-Body Problem, with the same initial condition and mass parameter as before.

The integration time has been restricted to 10 units, to avoid long testing times. We have asked for a local error of $10^{-32}$ for the rk78, and of the $10^{-33}$ for the Taylor method.

The total cpu time for the rk78 is of 3m 48s, while the Taylor method only takes 4.1 seconds.

Next, we will discuss the capabilities of `taylor` to use different arithmetics.

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 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 will assume we want the standard double precision and it will generate a header file with the lines,

```
typedef double MY_FLOAT;
```

to define `MY_FLOAT` as `double`. We will also find the line

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

If we use the flag -gmp to ask for the GNU multiple precision arithmetic (see below), we will 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 will substitute the macros by the corresponding calls to the arithmetic library.

The package includes support for several extended precision arithmetics: DOUBLEDOUBLE, DD_REAL, DQ_REAL, GMP (the GNU Multiple Precision Library) and MPFR.

If a library does not contain implementation of trigonometric functions and/or 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 to integrate 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...).

Next, we are going to use extended precision (more concretely, the `gmp` library) to compute the error of the double precision version.

To measure the error, we have computed the relative difference between these two approximations. For instance, for the $x$ coordinate, the operations we have implemented are,

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

where $x$ is the extended precision approximation and $\tilde{x}$ is the double precision result. All the computations have been done in double precision. The result is written in multiples of the machine precision.

```
value of H at the initial condition: -0.1336207158e1
numerical integration starts...
1  0.183827140545086 174 -0.82041748043202e-153
2  0.374344795509428 174 -0.59666725849601e-153
3  0.574965855180706 174 -0.67125066580801e-153
4  0.789299521404807 174 -0.14916681462400e-153
5  1.000000000000000 174 -0.22375022193600e-153
iterates: 5  final time: 1.000000e+00
```

Numerical integration using gmp with a 512 bits mantissa and asking for a relative error of $10^{-150}$.

```
value of H at the initial condition: -0.1336207158e1
numerical integration starts...
1  0.180071007388544 346 0.124236998889749e-303
2  0.366492191198110 346 0.678258138919457e-303
3  0.562476214016376 346 0.878726168201663e-303
4  0.771527181403796 346 0.921848099579533e-303
5  0.997166681972274 346 0.106043126217200e-302
6  1.000000000000000 346 0.106043682485665e-302
iterates: 6  final time: 1.000000e+00
```

As before, but using a 1024 bits mantissa and asking for a relative error of $10^{-300}$.

## Variational flow

$$\dot{x} = f(x), \quad x(0) = x_0$$

For each initial data $x_0$, we call $F(x_0)$ the value of the solution at a given time $T$, $F(x_0) = x(T)$.

We are interested in the derivatives of $F$, up to a given order.

We have two options to compute these derivatives:

- A numerical integration of the variational equations
- To use automatic differentiation on the evaluation of $F$.

We focus on the second option.

Or, in other words:

For the initial value problem

$$\begin{aligned} \mathbf{x}'(t) &= F(t, \mathbf{x}(t)) \qquad (1) \\ \mathbf{x}(t_0) &= \mathbf{x}_0 \qquad (2) \end{aligned}$$

we have to find a simple method to integrate the variational equations w. r. t. $\mathbf{x_0}$ along $\mathbf{x}(t)$. The variational equations may be of an arbitrarily prescribed order.

The standard procedure to compute the first variational equations is to linearize (1),

$$
\begin{aligned}
\delta_x' &= D_x F(t, x(t))\delta_x && (3)\\
\delta_x(t_0) &= I && (4)
\end{aligned}
$$

and integrate (1) and (3) simultaneously. Computation of $D_x F$ by hand is laborious and error prone when $F$ is complex. This is especially true when we need to integrate higher order variational equations.

An option is to use a computer algebra system to obtain $D_x F$.

Main idea: Automatic Differentiation

We use series arithmetic libraries to extend the floating point arithmetic.

It will allow us to integrate the variational equations automatically along with the IVP.

## Some Applications

- Computation of Lyapunov exponents (1st order variational equations)
- Computation and continuation of periodic orbits (1st order variational equations)
- Analysis of bifurcations of periodic orbits (at least, 2nd order variationals)
- Propagation of a small region of initial conditions, i.e., the uncertainty of an asteroid (high order variational equations)

## Multivariate series

Suppose $x = x(s_1, s_2, \cdots, s_n) = x(\mathbf{s})$ is a $C^m$ function near $\mathbf{0}$, then $x$ can be expanded as a Taylor polynomial,

$$x = x(\mathbf{0}) + \sum_i \alpha_i s_i + \sum_{i+j=2} \alpha_{ij} s_i s_j + \cdots + \sum_{i_1+i_2+\cdots+i_k=k} \alpha_{i_1 i_2 \cdots i_k} s_1^{i_1} s_2^{i_2} \cdots s_k^{i_k} + \cdots + O( \tag{5}$$

where

$$\alpha_{i_1 i_2 \cdots i_k} = \frac{1}{k!} \frac{\partial^k x}{\partial^{i_1} s_1 \partial^{i_2} s_2 \cdots \partial^{i_k} s_k} \tag{6}$$

are the normalized partial derivatives.

If $x$ and $y$ are $C^m$ functions, then the Taylor polynomial of $xy$ is the product of the Taylor polynomials of $x$ and $y$, truncated to degree $m$. Hence the partial derivatives of $xy$ can be obtained algebraically through the arithmetic operation on two polynomials.

This is true for $x \pm y$, $x/y$ and elementary functions on $x$.

In other words, if we code $x$, $y$ as

$$x = [x_0, \alpha_i, \alpha_{ij}, \cdots] \tag{7}$$

$$y = [y_0, \beta_i, \beta_{ij}, \cdots] \tag{8}$$

then we can compute the partial derivatives of $x \circ y$ and $f(x)$ automatically through series arithmetic. This is essentially an implementation of automatic differentiation (AD).

Example: Consider the function

$$f(s,t) = \big(st + \sin(s) + 4\big)\big(3t^2 + 6\big)$$

$\nabla f$ can be obtained automatically in the following manner when evaluate $f(s_0, t_0)$. We just need to promote $s$ and $t$ to series,

$$\begin{aligned} s &= [s_0, 1, 0] = [s_0, \frac{\partial s}{\partial s}, \frac{\partial s}{\partial t}] \\ t &= [t_0, 0, 1] = [s_0, \frac{\partial t}{\partial s}, \frac{\partial t}{\partial t}] \end{aligned}$$

and carry out the evaluation of $f(s_0, t_0)$ using series arithmetic,

$$\begin{aligned} s\ t &= [s_0 t_0, t_0, s_0] \\ \sin(s) &= [\sin(s_0), \cos(s_0), 0] \\ t^2 &= [t_0^2, 0, 2t_0] \end{aligned}$$

$$\begin{aligned}
st+\sin(s)+4 &= [s_0t_0+\sin(s_0)+4, \quad t_0+\cos(s_0), \quad s_0] \\
3t^2+6 &= [3t_0^2+6, \quad 0, \quad 6t_0] \\
f(s_0,t_0) &= [(s_0t_0+\sin(s_0)+4)(3t_0^2+6), (3t_0^2+6)(t_0+\cos(s_0)), \\
&\qquad 6t_0(s_0t_0+\sin(s_0)+4)+s_0(3t_0^2+6)]
\end{aligned}$$

and

$$\nabla f = [(3t_0^2+6)(t_0+\cos(s_0)), 6t_0(s_0t_0+\sin(s_0)+4)+s_0(3t_0^2+6)]$$

Higher order derivatives can be obtained the same way.

Example: Consider the initial value problem

$$\begin{aligned} x' &= f(t,x,y), \qquad & y' &= g(t,x,y) \\ x(0) &= x_0, \qquad & y(0) &= y_0 \end{aligned}$$

If we promote the state variable to series, like

$$x(t) = [x; \frac{\partial x}{\partial x_0}, \frac{\partial x}{\partial y_0}; \dots], \qquad y(t) = [y; \frac{\partial y}{\partial x_0}, \frac{\partial y}{\partial y_0}; \dots]$$

with initial values

$$x = [x_0; 1, 0; 0, \dots, 0], \qquad y = [y_0; 0, 1; 0, \dots, 0]$$

Then for any explicit integration algorithm,

$$(t_{k+1}, x_{k+1}, y_{k+1}) = \Phi(t_k, x_k, y_k)$$

we can overload the arithmetic in $\Phi$ using series arithmetic, to obtain the jet of derivatives $(t_1, x_1, y_1)$ using the initial value $(t_0 = 0, x_0, y_0)$, and to iterate the procedure to obtain the jet of $\Phi^n$,

$$(t_n, x_n, y_n) = \Phi(t_{n-1}, x_{n-1}, y_{n-1}) = \Phi^n(t_0, x_0, y_0).$$

In this sense, we are integrating the variational equations up to the order of the jets used for the state variables $(x, y)$.

In general, for the initial value problem

$$\begin{aligned} \mathbf{x}'(t) &= F(t, \mathbf{x}(t)) \quad (9) \\ \mathbf{x}(t_0) &= \mathbf{x}_0 \quad (10) \end{aligned}$$

if $F$ is composed from elementary functions, then we can integrate the variational equations (up to an arbitrarily specified order) automatically using series arithmetic via an explicit integrator.

The Taylor integrator is an explicit method, so we can overload it with series arithmetic. However, series operations are much more expensive than floating point operations, so we don't want to promote all variables to series unless we have to. For example, in the problem of studing the orbit of an asteroid, the vector field looks like

$$\begin{aligned} \mathbf{x}' &= f(\mathbf{x}) \\ \mathbf{y}' &= g(\mathbf{x}, \mathbf{y}) \end{aligned}$$

where the first equation describe the motion of 9 planets, the second equation describe the motion of the asteroid. Obviously we are only interested in the variations of $\mathbf{y}$.

We add one new declaration statement

**jet** VARSLIST **variables** COUNT **degree** DEGREE ;

to declare series variables. For example

**jet** y1,y2,y3 **variables** 5 **degree** 7 ;

specifies $y1, y2, y3$ are series of 5 variables, and are of degree 7.

The input for lorenz equations may look like

```
x' = 10.0 * (y - x);
y' = 28.0 * x - x * z - y;
z' = x * y -8.0 * z / 3.0;

jet x, y, z variables 3 degree 1;
```

With this input, the code generated by Taylor will integrate the first order variation equations along the reference trajectory.

For efficiency reasons, we are implementing several libraries for the operations with series.

We note that the key operation (regarding to efficiency) is the product of series.

- Computation of maximal Lyapunov exponents.
  In this case, we only need a series up to order 1, with one variable ($a^{(0)} + a^{(1)}s$). The arithmetic of series is straightforward and the code can be inlined.

- Computation of (1st order) variational flow.
  It has many applications, like computation of all Lyapunov exponents, linear stability of periodic orbits, differential of Poincaré maps, etc.
  In this case, the series are expressions of the form $a_0 + \sum a_j^{(1)} s_j$.
  Again, the formula for the product of 1st order series (up to 1st order) is straightforward, and it has been coded using simple loops.
- Computation of second order variational flow.
  This is also a quite common case. We have coded a specific algorithm for the product of series.

- Computation of variationals of order $N$, with two variables. This is a special case. The series is stored as a sequence of homogeneous polynomials. Each of them can be seen as a polynomial of 1 variable,
$$\sum_{j=0}^{n} a_{j,n-j}^{(n)} s_1^j s_2^{n-j}.$$
Then, the product of two homogeneous polynomials is coded with a single loop (like a product of polynomials of 1 variable).

- Computation of variationals of order $N$, with $M$ variables. This is the general case. The series is stored as a sequence of homogeneous polynomials.

  Each homogeneous polynomial is stored as an array. For each homogeneous polynomial, we use two hash functions to relate its position inside the array to its multiindex. Using these two functions the product is straightforward.

## Application: motion of Apophis

This is joint work (still in progress) with E.M. Alessi, A. Farrés, C. Simó and A. Vieiro (all from U. of Barcelona). It started as an Ariadna Contract with the European Space Agency.

Apophis is a near-Earth asteroid (NEO) that caused a brief period of concern in December 2004 because initial observations indicated a small probability (up to 2.7%) that it would strike the Earth in 2029.

Additional observations provided improved predictions that eliminated the possibility of an impact on Earth or the Moon in 2029.

However, Apophis will have succesive approaches to the Earth and, for the moment being, we cannot rule out an impact for the next approach in 2036, although it is very unlikely...

Here we want to transport the uncertainty region for Apophis (coming from the errors in the measurements of its actual postion).

The model is a restricted $N-$body model. The asteroid moves driven by the gravitational force exerted by the Sun, the nine planets and the Moon ($N = 11$).

Other effects will be added in the future.

We will use a Taylor method for the integration of the ODE and the jet transport methodology to transport a box of data along time.

- At a fixed time $t > 0$ the solution is represented as a truncated Taylor series in $\xi = (\xi_1, ..., \xi_6)$.
- The coordinates can be adapted according to the shape of the initial uncertainty set.
- The effect of the uncertainty in parameters can be included.
- It can be modified to take into account different distributions for the uncertainty.
- By replacing the usual arithmetic by interval arithmetic and rigorous estimates on the errors, the computations could be made absolutely rigorous.

- In 2029 Apophis will get at about 37000 km w.r.t. the centre of the Earth.
- The most significant changes that the orbit of Apophis will suffer are in the inclination w.r.t. the ecliptic plane, in the semi-major axis and thus in the period, which will change from 320 days to 425 days in average. The speed of the asteroid will slow down of about 3 km/s in average.
- Between 2036 and 2037, Apophis will undergo three Earth close approaches, at least of approximately $40 \times 10^6$ km.
- A good estimate of these approaches strongly depends on a very precise determination of the changes on the asteroid's trajectory in the 2029 approach.

- We have checked that at least 13th or 14th order variational equations are required to have a good description of the 2036-2037 passages with the data available at present.
- If, with the actual data, we assume standard errors 4 times smaller, then 7th order variational equations would be enough to depict the behaviour of the image box accurately and in that case we can guarantee that Apophis will not collide with the Earth in these approaches.