STUDIA UNIV. BABEŞ-BOLYAI, MATHEMATICA, XL, 3, 1995

PERTURBATION METHODS WITH LIE SERIES

Alin BLAGA*

Received: 3.09.1995
AMS subject classification: 68Q25, 68Q40, 65Y25

REZUMAT. - Metode de perturbaţie cu serii Lie. Puterea programării pe obiecte, facilităţile de paralelism ale calculatoarelor moderne cât şi cele ale produselor program de calcul simbolic sunt instrumentele perfecte pentru algoritmii folosiţi în teoria hamiltoniană a perturbaţiilor, pentru prezicerea traiectoriilor sateliţilor atrificiali. Este vorba, aici, despre metodele cu serii Lie, datorate lui Hori, Deprit şi Kamel. A se nota că formalismul Lie a fost introdus de Gröbner şi poate fi găsit în Nayfeh, sau Giacaglia.

1. Introduction. In celestian mechanics the methods used to predict the artificial satellites trajectories with a big accuracy were changed from time to time, become more eficient, more precise. Thus, we find important names in this field such Poincaré.

In Hamiltonian Theory of Perturbations the ice was broken by introducing Lie series and transforms, according to Hori (1966), Garrido (1968) and Deprit (1969). The Lie formalism can be found in Nayfeh (1973) and Giacaglia (1972). The basic idea is to find the solution of perturbation problems using power series. The problem is of the type:

$$\frac{dy_i}{dt} = x_i(t, y_j) + \varepsilon f_i(t, y_j).$$

This is a differential equations system with unknown functions $y_i(t)$ and with perturbed terms $\varepsilon f_i(t, y_j)$, where ε is a small parameter showing that the perturbing terms are small with respect to the principal terms of x_i. When ε is zero the system is reduced to the unperturbed system:

$$\frac{dy_i}{dt} = x_i(t, y_j).$$

In real problems the perturbing functions are more sophisticated, so it is too difficult

* *"Babeş-Bolyai" University, Faculty of Mathematics and Computer Science, 3400 Cluj-Napoca, Romania*

to solve them by classical Runge-Kutta methods, the error propagation is too high in such a problem. But we find particular problems solved in this way, using also refined numerical methods (see [16]). Steifel introduces the Fourier expansion as a good reason to approximate the solution.

Transformation of variables expanded in power of a small parameter plays an important role in the theory of perturbation. Hori (1966) and Deprit (1967) have proposed two methods to build canonical transformations, depending on a small parameter and based on the consideration of Lie series and transforms.

This paper tries to use the Lie transforms technology developed by Kamel (1969), combined with approximation methods of orthogonal Chebyshev polynomials. A big advantage of this technology is that it allows parallelism.

This method has been implemented in Maple V, but for high solution accuracy it was too hard to handle it, because of many symbolical differeniation and ordering, and impossibility of parallelization; and out of run-time and out of memory space appeared. It has been also implemented in MACSYMA and SPASM (see [2]), but there were the same problems of run-time and exhaustive memory needs. First, we have:

$$T_n(x, y) = H_n(x, y) + \sum_{j=1}^{n-1} \left(C_{n-1}^{j-1} [H_{n-j}(x, y), S_j(x, y)] + C_{n-1}^{j} K_{j, n-j}(x, y) \right), \text{ where}$$

$$S_n = i(Tr),$$

$$K_n = p(Tr),$$

$$K_{j,i} = [K_i, S_j] - \sum_{m=1}^{j-1} C_{j-1}^{m-1} [K_{j-m,i}, S_i].$$

$[f,g]$ is Poisson bracket operator, defined as:

$$[f, g](P, Q) = \sum_{i=1}^{n} \left(\frac{\partial f}{\partial Q_i} \frac{\partial g}{\partial P_i} - \frac{\partial f}{\partial P_i} \frac{\partial g}{\partial Q_i} \right).$$

We also have here:

$$i(f) = \int_0^y f(p, q)\, dq_1$$

$$p(f) = \int [f - i(f)]\, dq_1, \text{ where the Hamiltonian:}$$

$$F(P, Q, \varepsilon) = \sum_{i=0}^{\infty} \frac{\varepsilon^i}{i!} K_i(P, Q).$$

To avoid Poisson brackets, Kamel developed an algorithm based on Lie transforms, where

$$K_{j,i} = K_{j,i-1} + \sum_{m=0}^{i-j} C_i^m{}_{-j} \, L_{m+1} K_{j-m-1,\, i-1} \text{ and}$$

$$L_m K(x) = H_m(x) \frac{\partial K}{\partial x}(x).$$

This short table shows the run-time for the sequential algorithm (C++, PAC++) and for the parallel (Athapascan) algorithm. The default values are $n = 100$ for matrix dimension and m for matrices:

m - sequential/ processors - parallel	**PAC++** sequential double floating point (sec)	**PAC++** sequential Rational (sec)	**Athapascan** parallel double floating point (sec)
4	31.66	44.16	28.44
5	42.53	88.33	36.23
30	492.12	4057.3	402.32
100	2328.8	51021	2331.9

In this paper we do not try to increase the Hamiltonian's order, but increase the accuracy with a given precision (≈ 30) according to speed. With such accuracy we can find a more powerful approximation of Lie transform, we have maximum run-time speed, by its parallelism and no more exhausted workspace. It is amazing what we can do using power orthogonal series, then aliate with symbolical calculus to make almost perfect the ideea that the trajectory of an artificial satellite is more precise, more perfect even after hundred revolutions.

Most of technologies of perturbation analysis can be introduced by a short study of a sample algebraic equation. So, let us consider the following quadratic:

$$x^2 + \varepsilon x - 1 = 0 \tag{1}$$

Here ε plays a perturbing role for the solution. This quadratic has exact solutions:

$$x_{1,2} = \frac{-\varepsilon \pm \sqrt{4 + \varepsilon^2}}{2}.$$

If we expand them using Taylor series, for a small ε, we find:

$$x_1 = 1 - \frac{1}{2}\varepsilon + \frac{1}{8}\varepsilon^2 - \frac{1}{128}\varepsilon^4 + \dots$$
$$x_2 = -1 - \frac{1}{2}\varepsilon - \frac{1}{8}\varepsilon^2 + \frac{1}{128}\varepsilon^4 + \dots$$

The most important thing is that we have a very good approximation, even for a few given terms. Let see for $\varepsilon = 0.1$ what is going on:

$$x_1 \approx 1$$
$$\approx 0.95$$
$$\approx 0.95125$$
$$\approx 0.95124921$$
$$x_1 = 0.951249219 \ldots$$

Let us have the new quadratic:

$$\varepsilon x^2 + x - 1 = 0. \tag{2}$$

Here ε plays a perturbing role for the solution. For $\varepsilon = 0$ we have only one root, $x = 1$ and for $\varepsilon \neq 0$ we find two roots. This is the most easies example of that we call *singular* perturbation problem. The problems that are not singular are *regular*. In reality, most of the problems are singular and those are the most interesting.

Performing a Taylor series expansion for those two roots, for a small ε, we have:

$$x_1(\varepsilon) = 1 - \varepsilon + 2\varepsilon^2 - 5\varepsilon^3 + \dots \tag{3}$$
$$x_2(\varepsilon) = -\frac{1}{\varepsilon} - 1 + \varepsilon - 2\varepsilon^2 + 5\varepsilon^3 + \dots \tag{4}$$

But the power series for x_2 starts with an ε^0 instead of the usual ε^{-1}, so a very good ideea is to make a coordinate transformation, thus the singular equation becomes a regular one. That is the mechanism of Lie transforms, to regularise the singular problem and more, to reduce the finding of solution to a standard power series expansion form problem.

For example let us rescale:

$$x = \frac{y}{\varepsilon}. \tag{5}$$

We now have the regular equation:

$$y^2 + y - \varepsilon = 0.$$

2. General theory of the algorithm

DEFINITION. Let $G \subseteq C$ be an open set and $\rho_i : G^n \to C$, $i = \overline{1, n}$ holomorphic functions. If $z = (z_1, z_2, \ldots, z_n) \in G^n$, then

$$D = \rho_1(z) \frac{\partial}{\partial z_1} + \rho_2(z) \frac{\partial}{\partial z_2} + \ldots + \rho_n(z) \frac{\partial}{\partial z_n} \tag{6}$$

is the Lie differential operator.

Notation. Let be the open set $G \subseteq C$ and f, $\rho_i : G^n \to C$, $i = \overline{1, n}$, holomorphics. Then

$$Df = \rho_1(z) \frac{\partial f}{\partial z_1} + \rho_2(z) \frac{\partial f}{\partial z_2} + \ldots + \rho_n(z) \frac{\partial f}{\partial z_n}, \tag{7}$$

$$D^0 f = f, \tag{8}$$

$$D^{n+1} f = D(D^n f). \tag{9}$$

DEFINITION. A power series of type

$$\sum_{n=0}^{\infty} \frac{t^n}{n!} D^n f(z) = f(z) + \frac{t}{1!} Df(z) + \frac{t^2}{2!} D^2 f(z) + \ldots$$

where $f : G^n \to C$ is an holomorphic function, $G \subseteq C$ is an open set, $t > 0$, is called Lie power series. Formally

$$e^{tD} f(z) = \sum_{n=0}^{\infty} \frac{t^n}{n!} D^n f(z). \tag{10}$$

THEOREM. *$G \subseteq C$ is an open set. Any x-dependent, holomorphic and indefinitely differentiable vector $F(x,\varepsilon)$ of the form*

$$F(x, \varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} F_n(x),$$

where ε has a perturbing role, can be expressed in the form of another holomorphic and indefinitely differentiable vector of power series, using $x = y + \sum_{n=1}^{\infty} \frac{\varepsilon^n}{n!} F_n(x)$ transform, where f_n: $G \to C$.

Proof. Let be the indefinitely differentiable vector $F(x,\varepsilon)$ developed in power series of the form

$$F(x, \varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} F_n(x), \quad \text{with} \tag{11}$$

$$F_n(x) = \left[\frac{\partial^n}{\partial \varepsilon^n} F(x, \varepsilon) \right]_{\varepsilon=0}. \tag{12}$$

If $x = x(y,\varepsilon)$, then the vector has the power series form such as

$$F(x, \varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} F_{<n>}(x), \text{ where} \tag{13}$$

$$F_{<n>}(x) = \left[\frac{\partial^n}{\partial \varepsilon^n} F(x, \varepsilon) \right]_{\varepsilon=0,\ x=y}. \tag{14}$$

The relation between $\frac{\partial F(x, \varepsilon)}{\partial \varepsilon}$ and $\frac{dF(x, \varepsilon)}{d\varepsilon}$ was established using obsolete differentiation formula:

$$\frac{dF}{d\varepsilon} = \frac{\partial F}{\partial \varepsilon} + \frac{\partial F}{\partial x} \frac{dx}{d\varepsilon}. \tag{15}$$

It is obvious that if x is y independent, F_n is expressed using partialy differentiation, else totaly differentiations required, such as in (14).

At the next step we need to make the transform

$$x = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} f_n(y), \text{ where} \tag{16}$$

$f_0(y) = y$, according to Lie transform by the known generating functions (12). Thus, for the $x \to y$ and known F_n functions we can express $F_{<n>}$. This powerful mechanism can reduce a singular system of differential equations to a regular one.

We may now differentiate equation (16), and we find

$$\frac{dx}{d\varepsilon} = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} f_{n+1}(y),$$

which has no solitary y. Let be $W(x, \varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} f_{n+1}(y)$ and $f \equiv W$ we have the simple formula

$$W(x, \varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} W_{n+1}(y). \tag{17}$$

(17) is called a Lie Transform. This is just a power series, but it has the most important role to perform the final Kamel Transform.

Combining (15) with (17) we obtain

$$\frac{dF(x, \varepsilon)}{d\varepsilon} = \frac{\partial F(x, \varepsilon)}{\partial \varepsilon} + W(x, \varepsilon) \frac{\partial F(x, \varepsilon)}{\partial x}. \tag{18}$$

Let us make the notation $L_W F(x, \varepsilon) \equiv W(x, \varepsilon) \frac{\partial F(x, \varepsilon)}{\partial \varepsilon}$ which is the Lie derivative. Applied to (18) it will have a more compact form

$$\frac{dF(x,\varepsilon)}{d\varepsilon} = \frac{\partial F(x,\varepsilon)}{\partial \varepsilon} + L_W F(x,\varepsilon). \tag{19}$$

Let substitute F in (19) by (11) and W also in (19) but for this time in (17)

$$\frac{dF(x,\varepsilon)}{d\varepsilon}\sum_{n=0}^{\infty}\frac{\varepsilon^n}{n!}F_{n+1}(x) + \\ + \sum_{n=0}^{\infty}\frac{\varepsilon^n}{n!}\frac{\partial F_n}{\partial x}(x)\sum_{n=0}^{\infty}\frac{\varepsilon^n}{n!}W_{n+1}(x). \tag{20}$$

Sorting terms of (19) we have for ε^n

$$\frac{1}{n!}\frac{1}{0!}\frac{\partial F_0(x)}{\partial x}W_{n+1}(x),$$

$$\frac{1}{(n-1)!}\frac{1}{1!}\frac{\partial F_1(x)}{\partial x}W_n(x),$$

$$\vdots$$

$$\frac{1}{0!}\frac{1}{n!}\frac{\partial F_n(x)}{\partial x}W_1(x).$$

Thus, one obtains

$$\frac{dF(x,\varepsilon)}{d\varepsilon} = \sum_{n=0}^{\infty}F_{n+1}(x) + \sum_{n=0}^{\infty}\varepsilon^n\sum_{k=0}^{n}\frac{1}{k!(n-k)!}\frac{\partial F_{n-k}(x)}{\partial x}W_{k+1}(x), \text{ or}$$

$$\frac{dF(x,\varepsilon)}{d\varepsilon} = \sum_{n=0}^{\infty}\frac{\varepsilon^n}{n!}\left[F_{n+1}(x) + \sum_{k=0}^{n}C_n^k\frac{\partial F_{n-k}(x)}{\partial x}W_{k+1}(x)\right]. \tag{21}$$

Making the notation $F_{n,1}(x) = F_{n+1}(x) + \sum_{k=0}^{n}C_n^k L_{k+1}F_{n-k}(x)$, where

$$L_k F(x) = \frac{\partial F(x)}{\partial x}W_k(x),$$

will have for (21)

$$\frac{dF(x,\varepsilon)}{d\varepsilon} = \sum_{n=0}^{\infty}\frac{\varepsilon^n}{n!}F_{n,1}(x). \tag{22}$$

Performing formal notation of F_n by $F_{n,1}$ and by induction, yields to

$$\frac{d^m F(x)}{d\varepsilon^m} = \sum_{n=0}^{\infty}\frac{\varepsilon^n}{n!}F_{n,m}(x), \quad m\geq 1, \text{ where} \tag{23}$$

$$F_{n,m}(x) = F_{n+1,m-1}(x) + \sum_{k=0}^{n}C_n^k L_{k+1}F_{n-k,m-1}(x), \quad m\geq 1, \quad n\geq 0. \tag{24}$$

Here, $F_{n,0} \equiv F_n$ and $F_{0,n} \equiv F_{<n>}$. QED, only if we show that $F_{<n>}$ is holomorphic. But we know that

$$F_{n,1}(x) = F_{n+1}(x) + \sum_{k=0}^{n} C_n^k L_{k+1} F_{n-k}(x),$$

so it is obvious, according to F_i, $i \geq 0$, that $F_{n,1}$ is an holomorphic function. Thus, using (24), $F_{0,n}$, $n \geq 0$ are also holomorphic functions.

Remark. The recursive relation of (24) can be visualized in the forward triangle

$$\begin{array}{ccccccc}
F_0 & = & F_{<0>} & & & & \\
\downarrow & & & & & & \\
F_1 & \rightarrow & F_{0,1} & = & F_{<1>} & & \\
\downarrow & & \downarrow & & & & \\
F_2 & \rightarrow & F_{1,1} & \rightarrow & F_{0,2} & = & F_{<2>} \\
\downarrow & & \downarrow & & \downarrow & & \\
F_3 & \rightarrow & F_{2,1} & \rightarrow & F_{1,2} & \rightarrow & F_{0,3} = F_{<3>} \\
\downarrow & & \downarrow & & \downarrow & & \downarrow
\end{array}$$

Remark. The recursive relation of (24) is the same one found by Deprit, except to L_k operators, substituting Poisson brackets, which is more eficient.

We introduce some theoretical fundaments of Chebyshev polynomials, now.

DEFINITION. We call Chebyshev polynomial of degree n, the function T_n: [-1,1] → [-1,1],

$$T_n(x) = \cos n \arccos x. \tag{25}$$

LEMMA. *If T_n is the n degree Chebyshev polynomial, then*

$$T_{n+1}(x) = 2x T_n(x) - T_{n-1}(x), \quad n \geq 1. \tag{26}$$

LEMMA. *The n degree Chebyshev polynomial, $T_n(x)$, has n zeros on* [-1,1]

$$x_k = \cos\left(\frac{2k+1}{2n}\pi\right), \quad k = \overline{0, n-1} \tag{27}$$

and n+1 extremal points

$$x_k' = \cos \frac{k\pi}{n}, \quad k = \overline{0, n}. \tag{28}$$

THEOREM (of Orthogonality). 1°. *Continuous case. Let be the scalar product*

$$(f, g) = \int_{-1}^{1} \frac{f(x)\, g(x)}{\sqrt{1-x^2}}\, dx. \quad \textit{Then}$$

$$(T_i, T_j) = \begin{cases} 0, & \text{if } i \neq j \\ \pi/2, & \text{if } i = j \neq 0 \\ \pi, & \text{if } i = j = 0. \end{cases}$$

2°. Discrete case. Let be the scalar product

$$(f, g) = \sum_{k=0}^{m} f(x_k)\, g(x_k),$$

where x_j, $j = \overline{0, m}$ *are the zeros of Chebyshev polynomials of* $m+1$ *degree. Then*

$$(T_i, T_j) = \begin{cases} 0, & \text{if } i \neq j \\ \dfrac{m+1}{2}, & \text{if } i = j \neq 0 \\ m+1, & \text{if } i = j = 0. \end{cases}$$

Remark. Fourier coefficients c_i^* for an orthogonal system $\{\varphi_i\}_{i=0}^{m}$ and continuous function f are picked-up from the interpolating problem

$$\sum_{j=0}^{m} c_j^* \varphi_j(x_i) = f(x_i), \quad i = \overline{0, m} \text{ where} \tag{29}$$

$$c_i^* = \frac{(f, \varphi_i)}{\|\varphi_i\|^2}, \quad i = \overline{0, m} \tag{30}$$

Remark. Fourier coefficients for Chebyshev polynomials are

$$c_0 = \frac{\sum_{k=0}^{m} f(x_k)}{m+1}, \tag{31}$$

$$c_i = 2\,\frac{\sum_{k=0}^{m} f(x_k)\, T_i(x_k)}{m+1}, \quad i = \overline{1, m} \tag{32}$$

LEMMA. *Let be* c_i *the Chebyshev coefficients of a continuous function* f: $[-1,1] \to R$ *and* c_i' *the Chebyshev coefficients of its derivative. Then*

$$c_{i-1}' = c_{i+1}' + 2(i-1)\, c_{i-1}, \quad i = \overline{m-1, 1}, \tag{33}$$

$$c_m' = c_{m-1}' = 0.$$

3. Sequential computing of Lie matrix. The computing scheme is obtained by equations presented in last section

$$F_{n, m}(x) = F_{n+1, m-1}(x) + \sum_{k=0}^{n} C_n^k L_{k+1} F_{n-k, m-1}(x), \quad m \geq 1, \quad n \geq 0, \tag{34}$$

$$L_k F(x) = \frac{\partial F(x)}{\partial x}\, W_k(x), \quad k \geq 1,$$

$$F_{n,0} \equiv F_n, \quad F_{0,n} \equiv F_{<n>}$$

We find useful some complexity studies. We start here with the background algorithm:

```
INPUT.      Matrix dimension N=n+1, where n is fixed;
            Generating functions Fi(x), i=0,n;
            Transform functions Wj(x), j=0,n;
ALGORITHM.
Step1.            F(i,0) := F(i), i=0,n;
            {filling the first colum - initialisation step}
Step2.            F(n,n-i) := F'(i), i=0,n-1;
            {initialise the last row with derivatives
            of the first column}
Step3.      for i=0,n do
              for j=1,i+1 do
                Sum := 0;
                for k=0,i-j-1 do
                  Sum := Sum + C(i-j-1,k) *
                              W(k+1) * F(n-j,n-i+j+k+1);
                           {C(n,k) are binomial coefficients}
                endf;
              endf;
              {complete upper triangle:}
              F(i-j-1,j+1) := F(i-j,j) + Sum;
              {keeping the result before making modifications :}
              if n-j-1 = 0 then O(j+1) := F(i-j-1,j+1);
              endif;
              {complete lower triangle with derivatives :}
              F(n-j-1,n-i+j+1) := F'(i-j-1,j+1);
              {The main idea, for a best comprehensive algorithm
              is that on generate indices pairs of the form,
              keeping this order :
                        (0,1)
                        (1,1), (0,2)
                        (2,1), (1,2), (0,3)
                        (3,1), (2,2), (1,3), (0,4)
                Corresponding indices are F(i,j) → F(i-j-1,j+1)
                                          F'(i,j) → F(n-j,n-i).
              }
OUTPUT.     O(i),i=0,n;
```

We must note that the triangular matrix become a dense matrix by keeping also the derivatives. Thus, the algorithm is more complex, but is more faster, according to the complexity study at this stage. The computation scheme look like, at this moment:

$$
\begin{array}{ccccccc}
F_0 = F_{<0>} & \rightarrow & F_{-1,1} & \rightarrow & F_{-2,2} & \rightarrow & F_{-3,3} \rightarrow \\
\downarrow & & \uparrow & & \uparrow & & \uparrow \\
F_1 & \rightarrow & F_{0,1}=F_{<1>} & \rightarrow & F_{-1,2} & \rightarrow & F_{-2,3} \rightarrow \\
\downarrow & & \downarrow & & \uparrow & & \uparrow \\
F_2 & \rightarrow & F_{1,1} & \rightarrow & F_{0,2}=F_{<2>} & \rightarrow & F_{-1,3} \rightarrow \\
\downarrow & & \downarrow & & \downarrow & & \uparrow \\
F_3 & \rightarrow & F_{2,1} & \rightarrow & F_{1,2} & \rightarrow & F_{0,3} = F_{<3>} \rightarrow \\
\downarrow & & \downarrow & & \downarrow & & \downarrow
\end{array}
$$

This is just the formal algorithm which is the basic form of MapleV used algorithm, but for *C++* or Athapascan languages it is not possible to make formal derivatives. These languages, first for a sequential algorithm second for a parallel one, can increase the speed of the algorithm and it is implemented by choosing Chebyshev approximation methods. This idea make a faster algorithm and a more accurate computations according to Plauger studies and implementation on C Standard Library. The algorithm is changing now:

```
INPUT.      n for matrix dimensions;
            m is the number of Chebyshev coefficients;
            Generating functions Fi(x), i=0,n;
            Transform functions Wj(x), j=0,n;

ALGORITHM.
Step1.      for i=0,m-1 do
               FRes_i[j,0] := c_i[F_j]; j=0,n;
            endfor;

Step2.      for i=0,m-1 do
               Fres_i[n,n-i] := diff(c_i[F_j]); j=0,n;
            endfor;

Step3.      for i=1,n do
              for j=1,i do
               for l=0,m-1 do
```

```
            compute c_1 of FRes[i,j] function;
          endfor;
         Setting FRes[i,j];
        {keeping the result :}
        if i=j then O(i) := FRes(i,i);
        endif;
        {complete lower triangle with derivatives :}
        FRes(j,i) := FRes(i,j);

OUTPUT.     O_1(i), 1=0,m-1; i=0,n;
```

One Lie matrix is filled with Chebyshev coefficients and thus, the function F_{ij} is represented by its $c_k(i, j)$, $k = \overline{0, m}$ Chebyshev coefficients on each matrix. We may see that it is an exhaustive memory consumption, but we win on run-time.

In the first step we must fill the first column functions of the forward triangle by computing all their coefficients. And the same thing for the first row of derivatives.

In Step1 we compute all the (m-1) coefficients for each F_j, $j = \overline{0, n}$ continuous functions, using the common formula:

$$c_k = \frac{2}{m+1} \sum_{k=0}^{m} f(x_k)\, T_i(x_k), \quad i = \overline{1, m},$$

where x_k are the Chebyshev roots of the T_i polynomial. Step2 was developed in purpose of computing all derivatives of continuous functions F_j, $j = \overline{0, n}$, according to Chebyshev coefficients computed at the Step1 and to formula:

$$c'_{k-1} = c'_{k+1} + 2(k-1)\, c_{k-1}, \quad k = \overline{m-1, 1},$$

$$c'_m = c'_{m-1} = 0,$$

where c'_k are the corresponding Chebyshev coefficients to derivative of the function represented by c_k, $k = \overline{0, m-1}$.

The main loop needs to compute all $c(i,j)$ coefficients only by using $(j\text{-}1)^{th}$ column and j^{th} row of each matrix. For one $c_l(i,j)$ element we have (see also the (25) formula):

$$c_l(i, j) = CE(c(i, j-1)) + \sum_{k=0}^{i-j} C_{i-j}^{k}\, CE(c(j-1, i-k-1)) \;\forall\; i = \overline{1, n},\ j = \overline{1, i}, \quad (35)$$

where *CE* is a function that evaluate one function by its Chebyshev coefficients.

The internal Chebyshev computations are done by using of trigonometric high accuracy mathematical functions, that are more exact that the Standard C Library Math trigonometric functions. Also, for the binomial coefficients we made a vector to keep all values for all 1 to n degree. This method increase the algorithm run-time to very high speed and the vector reach only $\frac{(n+1)(n+2)}{2}$ size.

3.1 Complexity by algorithm. Some changes in the computation triangle are required, from this point of view.

If all values are recorded using triangle model shown in the 7th page of this paper, the algorithm complexity depends most of all in accessing and unaccessing direct matrix transform and some high precision evaluations. So, let consider that the other subalgorithms, wich are not shown in the background algorithm, have zero-complexity. Everyone can see that these subalgorithms are low-level functions that performs that performs some computations we do not need to know how, right now.

Dereferencing the classical forward triangle it leads to the upper triangle

$$
\begin{array}{ccccccc}
F_{0,0} & \rightarrow & F_{0,1} & \rightarrow & F_{0,2} & \rightarrow & F_{0,3} \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
F_{1,0} & \rightarrow & F_{1,1} & \rightarrow & F_{1,2} & & \\
\downarrow & & \downarrow & & \downarrow & & \\
F_{2,0} & \rightarrow & F_{2,1} & & & & \\
\downarrow & & \downarrow & & & & \\
F_{3,0} & & & & & & \\
\downarrow & & & & & &
\end{array}
$$

But let follow the algorithm and find its complexity. It is obvious that we may consider all matrix accesings and the first study is done only by this point of view.

At the Step 1 on access only the first column to store generating known functions F_n. If we denote by α_1 the algorithm to this level, will have

$$\text{cplx}\ (\alpha_1) = (n+1)\left[1 + \frac{n}{2}(n+1)\right]. \tag{36}$$

Next, on Step 2, must initialize the last row of lower triangle by first column derivatives. So, denoting by α_2 this algorithm, on leads to

$$\text{cplx}\ (\alpha_2) = \frac{n}{2}(n^2+7n+4) + 1. \tag{37}$$

The recursive equation access intensively the matrix, in Step 3, so we must split its complexity in three parts. Anyway, the final result obtained here is

$$\text{cplx}\ (\alpha) = \frac{n^5}{8} + 2n^4 + \frac{43n^3}{8} + 12n^2 + \frac{21n}{2} + 7.$$

We don't need to worry about, because nothing will stop here. We must try another method to store and compute triangle computations. On see that we begin with storing not in the lower triangle, but in the upper side, wich is the natural ideea. But we access more frequent derivatives stored in the lower triangle. So, the complexity increase at very high order. Thus we must inverse storing. This leads to others formulae. First, the recursive formula is changing.

Thus, if we denote by β the new algorithm complexity, we have

$$\text{cplx}\ (\beta) = \frac{11n^4}{12} + \frac{5n^3}{2} + \frac{13n^2}{12} + \frac{n}{2} + 2.$$

This was obtained by the algorithm steps Step1, Step2 and Step3. It is obvious that cplx(β) << cplx(α). For example, if $n = 32$ on have

$$\text{cplx}(\alpha) = 6{,}476{,}781 \text{ and}$$

$$\text{cplx}(\beta) = 1{,}044{,}242.$$

Thus

$$\frac{\text{cplx}\ (\alpha)}{\text{cplx}\ (\beta)} = 6,202,$$

which means:

run-time α	run-time β
(sec)	(sec)
3600	580.27
1800	290.13
300	48.22
60	9.40
1.0	0.09

We can see its density and only one look make you think of its complexity. Even in this case, the method leads to a more efficient algorithm. Until this reached point we have made studies on the formal algorithm.

We discuss the β-version of the algorithm, because of its already studied performance. So, let be $\pi(t)$ the time need to access one vector element, $t(^{*})$, $t(^{*})$ the basic operations time needings for a user or predefined specified operands and $C(t)$ the time for Chebyshev computations.

Remark. If $C(i,j;t)$ is the complexity to compute $c(i,j)$ coefficients by using (35) then the main loop complexity is

$$\text{cplx}\ (t) = \frac{n(n+1)}{2} C(t) + (m+1) \sum_{i=1}^{n} \sum_{j=1}^{i} C(i, j; t).$$

On evaluating cplx(t) let crossing $C(i,j;t)$ and find that

$$C(i,j;t) = (m+1)(i-j+2)\pi(t) + 3(i-j+1)t(^{*}) + t(^{*}) + (i-j+2)C(t) =$$

$$= (i-j+2)[(m+1)\pi(t) + C(t) + t(^{*})] + t(^{*}) - 3t(^{*}). \qquad (38)$$

Thus the order of complexity will be of the form:

$$C(t) = \frac{n^3}{6} C(t) + \frac{n^3 m}{6} t(^{*}) + \frac{n^2 m}{2} t(^{*}) + \frac{n^3 m^2}{6} \pi(t). \qquad (39)$$

Remark. The complexity of one trigonometric function has the order

$$\tau(t) = 12[t(^{*}) + t(^{*}) + \pi(t)].$$

4. Parallelization of the algorithm. An elementary task is defined as an indivisible work unit, specified in terms of its external environment such as I/O, execution time and so on. The parallel complexity study consists in splitting a part of the algorithm in elementary tasks. Thus, to compute $F_{i,j}$ function we need m Chebyshev coefficients. Thus the m tasks are:

$$\text{Task } P_t = \sum_{k=0}^{m} c_{i,j-1}^{(k)} T_i(c_{i,j}^{(t)}) - \frac{c_{i,j-1}^{(0)}}{2} + \\ + \sum_{k=0}^{m} c_{i,j}^{(k)} \left(\sum_{l=0}^{m} c_{j-1,\, i-k-1}^{(l)} T_l(c_{i,j}^{(t)}) - \right. \\ \left. - \frac{c_{j-1,\, i-k-1}^{(0)}}{2} \right) W(c_{i,j}^{(t)}), \quad t = \overline{0,\, m} \tag{40}$$

This leads to the well known precedence task graph, called *PARBEGIN - PAREND* graph, introduced by Dijkstra. If we denote by $t(T_i)$, the time to execute the task T_i, $i = \overline{0,\, m}$, then the T_1 time need to execute the algorithm from point $A(T_A)$ to point $B(T_A)$ will be:

$$T_1 = \max \{T_i,\ i = \overline{0,\, m}\},$$

instead of sequential time

$$T = \sum_{i=0}^{m} T_i .$$

We can write, using Dijkstra *PARBEGIN - PAREND* form, that

$$T_1 :\ T_A;\ \textit{PARBEGIN}\ T_0;\ |\ T_1;\ |\ \ldots\ |\ T_m;\ \textit{PAREND};\ T_B.$$

In (35) *CE* can be use unless all $c_{i,\cdot}^{(l)}$, $l = \overline{0,\, m}$ are computed. We must note that one computes for one function the all its coefficients into one FOR loop and these computations are independent. Also one coefficient needs a lot of time for the computations since this is dependent of mn^2 other coefficients. So this requires a small code for paralellization. One may see that we describe the parallel algorithm here.

```
INPUT. n for matrix dimensions;
       m is the number of Chebyshev coefficients;
       Generating functions Fi(x), i=0,n;
       Transform functions Wj(x), j=0,n;
```

```
ALGORITHM.
Step1.  for i=0,m-1 do
          FRes_i[j,0] := c_i[F_j]; j=0,n;
        endfor;
Step2. for i=0,m-1 do
          FRes_i[n,n-i] := diff(c_i[F_j]); j=0,n;
        endfor;
Step3. for i=1,n do
          for j=1,i do
            BEGIN PARALLEL LOOP from 0 to m-1
             compute c_1 of FRes [i,j] function;
            END PARALLEL LOOP;
          Setting FRes[i,j];
        {keeping the result :}
        if i=j then O(i) := FRes(i,i);
        endif;
        {complete lower triangle with derivatives :}
        FRes(j,i) := Fres(i,j);
OUTPUT. O_1(i), 1=0,m-1; i=0,n;
```

The $c(i,j)$ element is computed by spawning all $c_l(i,j)$, from l=0 to m-1. This means, to this level, that on run with m parallel processes and at the end of all of them will have the $FRes(i,j)$ function expressed in terms of Chebyshev polynomials.

The slave, in parallel algorithm, compute one coefficient for the specified function, so, if we want to obtain the maximum speed it will be preferable to set the numbers of slaves to the number of Chebyshev coefficients. The result is collected when all of the slaves done their work.

Parallelizing the algorithm leads to maximize relation (38) for $i = \overline{1, n}$ and $j = \overline{1, I}$. It is obvious that the order of $cplx_1$ is from now:

$$Q_1(t) = \frac{n^3}{2} C(t) + \frac{n^3}{2} t(*) + \frac{n^2}{2} t(*) + \frac{n^2 m}{2} \pi(t). \tag{41}$$

So, getting the results from (39) and (40) on have

$$Q(t) - Q_1(t) = \frac{n^2 m}{2}\left(\frac{nm}{3} - 1\right)\pi(t) + \\ + \frac{n^2(m-1)}{2} t(*) + \frac{n^3(m-1)}{6} t(*) - \frac{n^3}{3} C(t). \tag{42}$$

On 3D graphical representation by array dimensions and number of coefficients we can get the evaluating continuous time of all operations described here. We can see how time-smooth is the parallel computing behind the sequential computations. The smoothness can be also visualized according to (40).

REFERENCES

1. Blaga A., *Lie series and transforms for applications in celestial mechanics*, Institut IMAG, LMC Grenoble, France, 1995.
2. Char B., McNamara B., *Adiabatic invariants of simple Hamiltonian systems via the Lie transforms*, MACSYMA Users Conference, Washington DC, 1979.
3. Christaller M., *Athapascan-0a sur PVM 3, définition et mode d'emploi*, IMA Grenoble, 1994.
4. Dahlquist G., Björck A., *Numerical Methods*, Prentice Hall, New Jersey, 1974.
5. Deprit A., *Canonical transformations depending on a small parameter*, Celestial Mechanics, volume 1, 1969.
6. Gautier T., Roch J.L., Villard G., *PAC++ v2.0, User and Developer Guide*, IMAG, Grenoble, 1994.
7. Gröbner W., *Die Lie-Reihen und ihre Anwerdungen*, Springer Verlag, Berlin, 1960.
8. Hamming R.W., *Numerical methods for scientists and engineers*, McGraw-Hill Book Company, New York, 1962.
9. Henrard J., *On a perturbation theory using Lie Transforms*, Celestial Mechanics, volume 2, 1970.
10. Kamel A.A., *Expansion formulae in canonical transformations depending on a small parameter*, Celestial Mechanics, Volume 1, 1969.
11. Kamel A.A., *Lie Transforms and the Hamiltonization of non Hamiltonian systems*, Celestian Mechanics, volume 4, 1971.
12. Kamel A.A., *Perturbation method in the theory of nonlinear oscillations*, Celestial Mechanics, volume 3, 1970.
13. Kinoshita H., *Third-order solution of an artificial-satellite theory*, Smithsonian Institution, Astrophisical Observatory, Cambridge, Massachusets, 1977.
14. Mersman W.A., *A new algorithm for the Lie Transformation*, Celestial Mechanics, volume 3, 1970.
15. Plauger P.J., *The Standard C Library*, Prentice Hall, New Jersey, 1992.
16. Stiefel P.J., Scheifele G., *Linear an Regular Celestial Mechanics*, Springer Verlag, New York, 1971.