Semi-analytical computation of Normal Forms, Centre Manifolds and First Integrals of Hamiltonian systems (I)

Àngel Jorba
angel@maia.ub.es

University of Barcelona

Advanced School on Specific Algebraic Manipulators

Outline

Introduction

In these talks we will focus on the approximation of invariant structures of the phase space of a Hamiltonian system.

We will show how to effectively manipulate the Hamiltonian function to derive semilocal information around fixed points of the system.

In the next slides we will discuss practical techniques to implement these calculations, in an efficient language such as C/C++ (or Fortran).

One of the main problems faced when considering these kind of computations is how "to store" the object in the computer.

The easiest case is the computation of a single trajectory, that can be stored as a sequence of points in the phase space.

Note that, when the invariant object has bigger dimension, it can be very difficult (usually, it is impossible) to store it by simply storing a net of points. The approach taken here is to use some kind of series expansion to represent the object.

- The advantage is that in many cases only "a few" terms of these series are needed to get a good accuracy and that they can be handled very easily.
- As disadvantages we note that sometimes they have convergence problems making impossible to represent the object in this way.

Sometimes, when only a qualitative description of the dynamics is needed, it is enough to use a low order computation (this is the typical situation encountered, for instance, in the analysis of a bifurcation).

This is not the case considered here. The methodology presented in this paper is directed to produce high order computations, with a high degree of accuracy.

Hence, the first point addressed is how to build an efficient algebraic manipulator (in an efficient language such as C or C++) to manipulate these expansions fast, and using as little memory as possible.

As an example, we will show how to use these techniques to describe the (nonlinear) dynamics near the collinear points of the RTBP.

We will also address related topics such as error analysis (including the use of interval arithmetic), efficiency (both from the memory and speed points of view) and some possible extensions (more variables, time dependence, etc.).

The source code for several of the algorithms explained here can be retrieved from my web page,

`http://www.maia.ub.es/~angel/soft.html`

Let us consider a 3DOF Hamiltonian system with an equilibrium point at the origin, of the type centre×centre×saddle.

We are interested in finding a description of the dynamics in a neighbourhood (as big as possible) of the origin.

One possibility is to perform the so-called reduction to the centre manifold. That is, to perform changes of variables in order to uncouple (up to some finite order) the hyperbolic behaviour from the centre one (one can look at this as a partial normal form).

Hence, the restriction of the Hamiltonian to this (approximate) centre manifold will be a 2DOF Hamiltonian system. So, selecting an energy level $H = h$ and doing a suitable Poincaré section we can produce a collection of 2-D plots that can give a good description of the dynamics.

Let us assume that we are interested in the dynamics near an elliptic equilibrium point (that, for simplicity, we will locate at the origin) of a three degrees of freedom Hamiltonian system.

Assume we are able to rewrite the initial Hamiltonian H as

$$H = H_0 + H_1,$$

where H_0 is integrable and H_1 is non integrable.

Then, if H_1 is small enough near the point, the trajectories corresponding to H_0 are close to the trajectories of H (at least for moderate time spans).

Hence, from the integrable character of H_0 it is not difficult to obtain approximations for the invariant tori of H.

Let us assume that we are also interested in estimates of the diffusion time near the origin.

Note that the computational effort needed to do this by single numerical integration is too big that it can not be considered a feasible option.

An alternative procedure can be the following: lassume that we are able to rewrite the initial Hamiltonian as $H = H_0 + H_1$.

As H_0 is integrable, the diffusion present in H must come from H_1. Hence, one can easily derive bounds for the diffusion time in terms of the size of H_1. Of course, in order to produce realistic diffusion times one needs to have H_1 as small as it can be.

A standard way of producing the splitting $H = H_0 + H_1$ is by means of a normal form calculation: H_0 is the normal form and H_1 the corresponding remainder.

There are alternative ways of estimating the diffusion time near elliptic equilibrium points.

For instance, one can construct approximate first integrals near the point and estimate the "drift" of these integrals. Of course, although one can use as many first integrals as degrees of freedom, it is enough to use a single positive-definite integral (near the point, its level surfaces split the phase space in two connected components so they act as a barrier to the diffusion).

We want to note that although from the theoretical point of view both approaches are equivalent (the first integrals we compute are in fact the action variables of the normal form), from the computational point of view they behave differently.

In this course we will present several methodologies to deal with those computations, based on the use of algebraic manipulators.

There are several possible schemes, depending on the kind of calculation we are interested in. For instance, if the procedure only needs to substitute trigonometric series in the nonlinear terms of the equations (like in the Lindstedt-Poincaré method), one of the best choices is to look for a recurrent expression of those nonlinear terms (the substitution is simply done by inserting the series into the recurrence).

In this paper, we will apply schemes that work with the power expansion of the Hamiltonian (when the system is not Hamiltonian, one must work with the differential equations –or with the equations of the map if the system is discrete– but, of course, this increases the computational effort).

A general scheme for the problems considered here is:

1. Power expansion of the Hamiltonian around the origin.
2. Complexification of the Hamiltonian. This is not a necessary step but, as we will see, it allows to simplify further computations.
3. Changes of variables (usually by means of Poisson brackets), up to some finite order.
4. Realification of the final Hamiltonian. Again, this is not a necessary step. It is done only to reduce the size of the resulting series.
5. Computation of the change of variables that goes from the initial Hamiltonian to the final one.

So, one needs computer routines for all these steps.

A natural way of handling the power expansions is as a sequence of homogeneous polynomials:

$$H = \sum_{k \geq 2} H_k,$$

where H_k is an homogeneous polynomial of degree k.

As we will see, the bottleneck (with respect to speed) of the methods exposed here is the handling of homogeneous polynomials.

Basic Tools

Here we will discuss the basic algorithms to handle homogeneous polynomials.

For the moment, we will not specify the kind of coefficients of the polynomials.

Let us assume that we want to store an homogeneous polynomial P_n of degree n, with 6 variables $(x_0, \ldots, x_5)$,

$$P_n = \sum_{\substack{k \in \mathbb{N}^6 \\ |k|=n}} p_k x^k,$$

where we use the notation $x^k \equiv x_0^{k_0} \ldots x_5^{k_5}$ and $|k| = k_0 + \cdots + k_5$.

For the moment we assume that all the coefficients p_k are different from zero. Let us define

$$\psi_6(n) = \#\{k \in \mathbb{N}^6 \text{ such that } |k| = n\}$$

(that is, $\psi_6(n)$ denotes the number of monomials of P_n).

To store the polynomial,

- we use an array of $\psi_6(n)$ components (the kind of array depends on the kind of coefficients of the polynomial),
- we use the position (index) of a coefficient inside the vector to know the monomial it corresponds to.

To store the polynomial,

- we use an array of $\psi_6(n)$ components (the kind of array depends on the kind of coefficients of the polynomial),
- we use the position (index) of a coefficient inside the vector to know the monomial it corresponds to.

To this end we construct two functions:

- `llex6`: Given a place inside the array (that is, an integer between 0 and $\psi_6(n) - 1$) it returns the multiindex that corresponds to this coefficient.
- `exll6`: Given a mulltiindex, it returns its position inside the array.

Before going into the details of these functions, we want to stress that, from the point of view of efficiency, they are the most important ones: if they are efficient, the package will be efficient.

Let us see a first example of the use of these functions, to compute the product of two (homogeneous) polynomials.

```
/* code for p3 = p3 + p1*p2 */
  nt1=ntph6(g1); /* function psi */
  nt2=ntph6(g2);
  for (i=0; i<nt1; i++)
  {
    llex6(i,k1,g1);
    for (j=0; j<nt2; j++)
    {
      llex6(j,k2,g2);
      for (l=0; l<6; l++) k3[l]=k1[l]+k2[l];
      lloc=exll6(k3,g3);
      p3[lloc] += p1[i]*p2[j];
    }
  }
```

To have a fast implementation, we use an integer array (we assume here that every integer is four bytes long) to store some information to be used by function `llex6`.

This array has $\psi_6(n)$ components and each one contains (encoded) the multiindex of the corresponding coefficient.

We use this array in the obvious way: each time we need to know the exponent of the monomial whose coefficient is stored in the place j of the homogeneous polynomial, we get it from the component j of this array.

The way of encoding the multiindex k is the following: as we know the degree we are working with, one of the exponents (say k_0) is redundant, so we only need to store $k_1, \ldots, k_5$.

This has to be stored inside a 32 bits number, so we can use 6 bits for each index, leaving 2 unused.

This introduces the restriction $k_j < 64$. As we want to handle homogeneous polynomials the maximum degree allowed is 63, enough for the applications considered here.

The (most) basic functions

Their source code is stored in the file `mp6.c`. As these routines are the most important ones, we will discuss them more carefully.

For portability reasons, in the heading of several files we redefine the standard type `int` as `integer`.

Before continuing, let us define the function $\psi_i(n)$ as

$$\psi_i(n) = \#\{k \in \mathbb{N}^i \text{ such that } |k| = n\},$$

Before continuing, let us define the function $\psi_i(n)$ as

$$\psi_i(n) = \#\{k \in \mathbb{N}^i \text{ such that } |k| = n\},$$

Exercise

Prove that $\psi_i(n)$ can be evaluated by means of

$$\psi_i(n) = \sum_{j=0}^{n} \psi_{i-1}(j) = \binom{n+i-1}{i-1}.$$

The first step is to allocate space to store the values $\psi_i(j)$.

At this moment we only need to know ψ_6 but we will also compute $\psi_2, \ldots, \psi_5$ (they will be needed later on).

To this end we allocate a rectangular matrix `psi` with the first index ranging from 2 to 6 and the second one from 0 to `nor`.

The values $\psi_i(j)$ are computed (using the previous recurrence) and stored in the position (i, j) of the matrix `psi`.

Next step is to allocate space for the table `clmo`.

The first dimension of this table ranges from 0 to `nor`, and it refers to the degree of the homogeneous polynomials.

If the first index is i, the second index ranges from 0 to $\psi_6(i) - 1 \equiv$ `psi[6][i]` $- 1$.

The position (i, j) of this array is the encoded version of the multiindex of the monomial number j of a polynomial of degree i.

Once this table has been allocated, we have to fill it with the information about the multiindices.

Next, We need an order inside the set of multiindices of a given degree: Let k be a multiindex of degree n and let us define $\overline{k}$ as the integer number (in base $n+1$) $k_5 k_4 k_3 k_2 k_1 k_0$ (for instance, if $k = (1,2,3,4,5,6)$ then $\overline{k} = 654321$).

Then, the order is given by

$$k^{(1)} < k^{(2)} \Longleftrightarrow \overline{k^{(1)}} < \overline{k^{(2)}}.$$

This is usually called reverse lexicographic order.

Now, for a given degree `i`, we compute all the multiindices according to this order and we store them in the table `clmo`: the first one for degree `i` is `(i,0,0,0,0,0)`, and all the others are generated by routine `prxk6` (see next slide).

```
void prxk6(natural k[])
/*
   given a multiindex k, this routine computes the next one
   according to the lexicographic order.

   parameters:
   k: array of 6 components containing the multiindex. it is
       overwritten on exit (input and output).
*/
{
   if (k[0] != 0) {k[0]--; k[1]++; return;}
   if (k[1] != 0) {k[0]=k[1]-1; k[1]=0; k[2]++; return;}
   if (k[2] != 0) {k[0]=k[2]-1; k[2]=0; k[3]++; return;}
   if (k[3] != 0) {k[0]=k[3]-1; k[3]=0; k[4]++; return;}
   if (k[4] != 0) {k[0]=k[4]-1; k[4]=0; k[5]++; return;}
   puts("prxk6 error 1."); exit(1);
}
```

We store the components of each multiindex in the corresponding place of `clmo`, using 6 bits for each component: this means that the coded version of the multiindex is (note that we do not code k_0 because, as we know the degree, it is redundant)

$$k_1 + k_2 \times 2^6 + k_3 \times 2^{12} + k_4 \times 2^{18} + k_5 \times 2^{24}.$$

This is the value we will store in `clmo[i][j]`, where we have assumed that `j` stands for the place of the multiindex (and the monomial) inside this order.

Routine `llex6`

Given a location in the array of coefficients, `lloc`, and a degree, `no`, it computes the multiindex corresponding to them.

The way it works is very straightforward because the multiindex is contained (encoded) in `clmo[no][lloc]`, and to decode it we only need to invert the formula used to compute it using the modulus function.

```
void llex6(integer lloc, integer k[], integer no)
{
   natural n;
   integer m;
   if (lloc >= psi[6][no]) {puts("llex6 error."); exit(1);}
   n=clmo[no][lloc];
   k[1]=n%64; m=k[1];  n/=64;
   k[2]=n%64; m+=k[2]; n/=64;
   k[3]=n%64; m+=k[3]; n/=64;
   k[4]=n%64; m+=k[4];
   k[5]=n/64; m+=k[5];
   k[0]=no-m;
   return;
}
```

Routine `exll6`

Given a multiindex `k` of degree `no` (this is redundant information but it is very useful to avoid calling these routines in a wrong way), it returns the corresponding place.

The implementation of this routine can be done in many ways. Let us see one.

Let us denote by $k = (k_0, \ldots, k_5)$ the multiindex and let n be $k_0 + \cdots + k_5$. Define $k^{(5)}$ as $(k_0, \ldots, k_4)$ and let $n_5 = n - k_5$ be the degree of $k^{(5)}$. Then, if we are able to compute

1. the number of multiindices $(\ell_0, \ldots, \ell_5)$ of 6 variables with degree n such that $0 \leq \ell_5 < k_5$,
2. the place it corresponds to $k^{(5)}$ among the multiindices of 5 variables of degree n_5,

then, the sum of these two numbers is the place we are looking for.

Let us denote by $k = (k_0, \ldots, k_5)$ the multiindex and let n be $k_0 + \cdots + k_5$. Define $k^{(5)}$ as $(k_0, \ldots, k_4)$ and let $n_5 = n - k_5$ be the degree of $k^{(5)}$. Then, if we are able to compute

1. the number of multiindices $(\ell_0, \ldots, \ell_5)$ of 6 variables with degree n such that $0 \leq \ell_5 < k_5$,
2. the place it corresponds to $k^{(5)}$ among the multiindices of 5 variables of degree n_5,

then, the sum of these two numbers is the place we are looking for.

1. The first of these numbers is $\psi_5(n_5 + 1) + \cdots + \psi_5(n)$.
2. The second one is the same problem we want to solve, but with one dimension less, so we can apply again the same procedure until we reach dimension 2 (polynomials of two variables), where the solution of the problem is trivial.

There are some more functions in the file `mp6.c`

Routine `ntph6`: This routine returns the number of monomials of a given degree (this information is contained in the array `psi`).

Routine `prxk6`: It is used to produce all the multiindices of a given order, according to the order we are using. For more details, we refer to the source code.

Routine `imp6`: This routine allocates and initializes some internal arrays to store the encoded multiindices. The only parameter of this routine is an integer (`nr`) that contains the maximum degree we want to use. This value is stored in the variable `nor`.

Routine `amp6`: It frees the memory allocated by `imp6`. Of course, once it has been called the manipulator can not be used until a new call to `imp6` has been done.

It is quite common in physical examples to have some kind of symmetry in the Hamiltonian. For instance, in the examples used in this paper we have a symmetry with respect to the variable z.

This implies that not all the possible monomials of the power expansion of the Hamiltonian are really present.

In the examples used here we have that, if i is the exponent of z and j the exponent of p_z, the only monomials that appear in the expansion are the ones in which $i + j$ is even.

Hence, taking this into account it is possible to reduce the amount of memory used and the computing time by a factor of approximately two.

In order to exploit the symmetry we have developped special versions of the routines in file `mp6.c`

File `mp6s.c` contains the same routines as `mp6.c` (but with an "s" at the end of the name, to be able to use them in the same program if necessary), but assuming that the only monomials present are the ones that satisfy that $k_4 + k_5$ is even.

As they work in a very similar way, we only mention the main differences.

`imp6s` Function $\psi_6(n)$ is not longer valid to compute the number of monomials. The number of monomials for a given degree n is

$$\sum_{j=0}^{[\frac{n}{2}]}(2j+1)\psi_4(n-2j),$$

where $[\frac{n}{2}]$ denotes the integer part of $n/2$.

`exll6s` To have a simple formula for the position for a given index, we have changed the order used for the monomials: we use the reverse lexicografic order for the exponents (k_4, k_5) and the reverse lexicographic order for the exponents (k_0, k_1, k_2, k_3) (this is usally called product reverse lexicographic order). It allows to derive a closed formula for the position (see the source code).

`prxk6s` It is changed in order to produce the exponents in the product reverse lexicographic order defined above.

File `mp6p.c` contains the same routines as `mp6s.c`, but with a different symmetry: here it is assumed that all the monomials that are present satisfy that $k_4 + k_5$ is odd (this kind of symmetry will appear in some computations).

The implementation is almost identical to `mp6s.c`, so we do not add further remarks.

File `mp6p.c` contains the same routines as `mp6s.c`, but with a different symmetry: here it is assumed that all the monomials that are present satisfy that $k_4 + k_5$ is odd (this kind of symmetry will appear in some computations).

The implementation is almost identical to `mp6s.c`, so we do not add further remarks.

In fact, as the examples considered in this paper have the above mentioned simmetry, we do not make use of the routines in `mp6.c`. I have included them for the sake of completeness, and because they are the most natural ones to start describing how these kind of routines work.

Finally, let us note that if the symmetries are "too complex" to derive closed formulas for the routines `exll`, one can always perform a binary search on the array `clmo`.

In this case, it is very convenient to use an order such that the integer values stored in `clmo` are sorted as integer numbers.

Although this is not as efficient as a closed formula, it can be easily applied in all the cases.

As the examples in this paper are three degrees of freedom Hamiltonian systems, the basic routines explained here handle polynomials with six variables.

If one is interested in a different number of variables, it is not difficult to build the corresponding basic routines.

For instance, later on we need to handle the normal form of a 3DOF Hamiltonian system, that depends on 3 variables. They are constructed using the same algorithms as for six variables.

We have put those routines in file `mp3.c`, Note that this file is, essentially, a minor modification of file `mp6.c`.

In a similar way we have the file `mp4s.c` and `mp4p.c`, that are needed during the reduction to the centre manifold.

Homogeneous Polynomials

The routines of this section are contained in the files `basop6s.cc` and `basop6sp.cc`.

Note that we have several versions of some of them, in order to deal with polynomials with different symmetries.

I recommend to give a look at the source code, since it will clarify (we hope!) our explanations.

In what follows, assume that `p1` and `p2` are two arrays containing (the coefficients of) homogeneous polynomials of degrees `g1` and `g2`.

Assume that both have the same degree, and that we want to add them, storing the result in an array called `p3`.

If we call `nm` the number of monomials of one of these polynomials (this is the value returned by a routine like `ntph6`), then the sum is easily computed:

```
for (i=0; i<nm; i++) p3[i]=p1[i]+p2[i];
```

Here we have assumed that we have defined the operation + for the type of the coefficients of the polynomial

Let us see the product of homogeneous polynomials (now we are not assuming that p1 and p2 have the same degree).

The algorithm is very straightforward and uses the previous rutines.

Let us call n_1 and n_2 the number of monomials of each polynomial p1 and p2.

Then, to multiply the monomial number i of p1 with the monomial number j of p2 we only have to compute the corresponding multiindices $k^{(i)}$ and $k^{(j)}$, to ask for the position where the coefficient of the monomial $k^{(i)} + k^{(j)}$ must be stored, and to add there the product of the coefficients.

Doing this for all the possible values of i and j we obtain the desired product.

The Poisson bracket of two homogeneous polynomials can be implemented using the same ideas as the product.

The algorithm we have used is based on the following identity:

$$\left\{ \sum_{k,\ell} p_{k,\ell} x^k y^\ell, \sum_{k',\ell'} q_{k',\ell'} x^{k'} y^{\ell'} \right\} =$$

$$\sum_{k,\ell,k',\ell'} p_{k,\ell} q_{k',\ell'} \left(\sum_{j=1}^{3} (k_j \ell'_j - k'_j \ell_j) \frac{x^{k+k'} y^{\ell+\ell'}}{x_j y_j} \right),$$

where, of course, k, ℓ, k' and ℓ' belong to $\mathbb{N}^3$.

Thus, for any term of this sum, we proceed as in the product of homogeneous polynomials: we look for the exponents of the monomials, we compute the exponents of the result and, in the corresponding position, we add the coefficients.

Poisson bracket

```
nt1=ntph6s(g1); nt2=ntph6s(g2); nt3=ntph6s(g3);
for (i=0; i<nt1; i++) {
  llex6s(i,k1,g1);
  for (j=0; j<nt2; j++) {
    llex6s(j,k2,g2);
    for (l=0; l<6; l++) k3[l]=k1[l]+k2[l];
    w=p1[i]*p2[j];
    m=k1[0]*k2[1]-k2[0]*k1[1];
    if (m != 0) {
        k3[0]--; k3[1]--; lloc=exll6s(k3,g3);
        p3[lloc] += m*w; k3[0]++; k3[1]++;
      }
    m=k1[2]*k2[3]-k2[2]*k1[3];
    if (m != 0) {
        k3[2]--; k3[3]--; lloc=exll6s(k3,g3);
        p3[lloc] += m*w; k3[2]++; k3[3]++;
      }
    m=k1[4]*k2[5]-k2[4]*k1[5];
    if (m != 0) {
        k3[4]--; k3[5]--; lloc=exll6s(k3,g3);
        p3[lloc] += m*w;  k3[4]++; k3[5]++;
      }
  }
}
```

We have coded several routines in order to read and write power expansions and homogeneous polynomials (both in ASCII and binary format).

It is important to add redundant information to the files (specially in binary files), like some integer codes to indicate the kind of symmetry (if any), etc.