Background and Documentation of Software for Computing Hamiltonian Normal Forms

Contributions by Peter Collins, Andrew Burbanks, Stephen Wiggins,
Holger Waalkens, and Roman Schubert
School of Mathematics, University of Bristol, University Walk, Bristol BS8 1TW

September 5, 2008

Abstract

This document describes software for computing the normal form in the neighborhood of equilibria of n-degree-of-freedom Hamiltonian systems.

Contents

Chapter 1

Introduction

This is a description of the theoretical background, structure and usage of the Normal Form software developed at the School of Mathematics at the University of Bristol over the past few years.

1.1 History

This software grew out of a research programme in theoretical chemistry concerned with the development of a *phase space reaction theory* for high dimensional Hamiltonian systems. Much of this research is reported in (22; 23; 28; 24; 25; 26; 19; 27)

The major part of the software was written in 2004/5 by Dr. Andrew David Burbanks who left the University of Bristol in 2005. In late 2005 Peter Collins took over the normal form software.

The code was originally developed in Mathematica, and our Mathematica code borrowed heavily from the code of Professor Jesus Palacian and Professor Patricia Yanguas. Subsequent work was focussed on using Python and C++ code to process the output from the Mathematica code (which did all the calculation of Normal Forms) and calculate coordinate changes and manifolds such as the normally hyperbolic invariant manifold (NHIM), it's stable and unstable manifolds, and the dividing surface. Code to calculate Normal Forms was then developed in Python, and for the more computationally intensive calculation, it was recoded in C++. The calculation of Semi-classical Normal Forms was added. The code has been developed to the stage where it will produce Normal Forms and coordinate transformations in the same format as the Mathematica code. Much of the information in this document is deduced from the code and some will be subject to revision over time.

Throughout the work one of the main drivers has been the desire for increases in the complexity of the systems which can be investigated from the dynamical systems point of view, the number of degrees of freedom, and the order to which the Normal Forms can be calculated.

Because the calculations are performed in terms of multivariate polynomials of increasing degree, the potential number of terms in each polynomial can become very large. Often, because of the nonlinear transformations involved, most of the possible terms can be present with significant coefficients.

There are three particular aspects if the manipulation of large polynomials that present problems:

- The speed of computation and time taken to perform a computation (e.g. loading transformations into memory can take a significant amount of time).
- The accuracy required to prevent error accumulation.
- The size of the polynomials themselves and memory requirements.

There is not a single solution that will solve all of these problems at once since the polynomial representations require a tradeoffs between them and each system examined may have a different requirement. As the problem complexity increases in terms of:

- The degree of freedom and number of variables.
- The degree to which the normal form is calculated.
- The accuracy of the calculations.

the potential number of terms in each polynomial increases, as does the memory footprint. For example, each monomial term in a polynomial can require hundreds of bytes and many polynomials may be in memory during a Lie triangle calculation. Assuming a few megabytes of memory we start to have potential problems at about a thousand terms. Since, for each polynomial multiplication we are now dealing with up to a million multiply-add operations, this can significantly reduce the accuracy of the result.

1.2 Environment

The software runs on a linux PC It has been developed under Suse linux version 9.2 and earlier. There is no known reason why it should not run under any standard Linux system. It may be possible to run it under Windows using Python etc but this is not expected to be trivial. There are a number of requirements in terms of software on the host machine, many of which will be present in most Linux distributions but some will not. The list starts with: Python C++ etc. but includes the Gnu Multi Precision library and its Python bindings. Not all of them are needed in all situations, a simple problem may only need Python not C++ or GMP. The linux version used here is now somewhat outdated but the use of newer software versions is not expected to raise significant problems. The software and some of these packages needs the environment set up. This should be done automatically by the script setenv.sh located in the bin directory. This can be run in the users .bashrc

1.3 Standard

The code is research standard software. It is still under development. It is neither fully optimised nor fully robust. It works on the problems we have applied it to under the conditions and parameters we have used. We have done our best to make it general purpose but expect that if you apply it to a new problem you may need to do anything from resetting tolerance levels to recoding.

1.4 Structure

There are three packages of software which have been developed: the Normal form code itself, software to do calculations using the coordinate transforms (either from Mathematica or from the Normal form code) and visualisation software to display some of the surfaces calculated. If you have received this package you probably have only the Normal form code, and this is described in this document. If you are at Bristol University you probably have access to all the software.

The Normal Form software is written in two languages Python and C++. Python was used for initial development and ease of use. The compute intensive parts of the code were translated to C++. The Python code retains all the functionality and may be used alone to produce results. The C++ code will speed up the Lie triangle and coordinate transform calculations.

At Bristol there are two main software areas, lib/src and projects/nf–unified–0.5/src. projects/nf–unified–0.5/src. The latter contains the software to calculate normal forms. The other software in lib/src calculates phase space objects and displays a projection of them in a visualisation. Most of the visualisation software is in the individual project areas. Many precalculated normal forms are stored in projects/normal-form These may all have been produced using the older Mathematica code. The visualisation requires MayaVi and VTK which can be tricky to set up.

There are two sets of software to calculate normal forms.

nf–unified–0.5/src/py – python code.

Example runs are in NormalFormTest.py The python code takes a Hamiltonian and an equilibrium point. It calculates a diagonalised form at the equilibrium point. It uses the Hamiltonian as a Taylor series expansions. It calculates the normal form using the Deprit triangle. The latter suffers from processing time and accuracy problems. Although it has been set up to test multi-precision reals it does not now seem to use them. I suspect that processing time and memory usage would make it impractical.

nf–unified–0.5/src/cpp – C++ code.

This starts from series expansions output from the python code and duplicates some of the functionalityof the Python code. The C++ code takes the Taylor series expansions and the diagonalised form and calculates the normal form using the Deprit triangle. It uses multprecision reals for accuracy and gives improved speed. It suffers from memory usage problems. There is much potential for reduction of memory usage but the best solution depends on the problem under investigation.

1.5 License

GPL Insert standard GPL text

Chapter 2

The Normal Form Software

In this chapter we will describe the structure of the normal form code. For completeness, in Chapter 4 we will provide some background for the theory for classical normal forms that is implemented in our code. Our code also has the capability for computing semiclassical normal forms. We will comment on that as appropriate. However, we emphasize that many of the notions we discuss in this Chapter (e.g. integrals of the motion, coordinate transformations) only have a meaning (that we are aware of) for classical Hamiltonian systems. A parallel development of the algorithms to compute classical and semiclassical normal forms in a way that emphasizes the similarities and differences is given in (27).

2.1 General Comments

First, it is useful to recall what we expect for the "output" from the normal form algorithm.

- An expression for the "normalised" Hamiltonian in terms of the "normal form (NF) coordinates".
- Explicit expressions for integrals of the motion in the NF coordinates.
- An explicit expression for the normalised Hamiltonian as a function of the integrals of the motion.
- Explicit expressions for the coordinate transformations between the NF coordinates and the original coordinates.

Before subjecting the Hamiltonian to the normalisation algorithm, a number of preliminary steps are required.

- Find a suitable (and *relevant*) equilibrium point.
- Compute the Taylor expansion of the Hamiltonian about this equilibrium point to some "appropriately high order". In general, the order depends on the accuracy required and one tends to know this only after subjecting a normal form, truncated (or computed) to a given order, to a battery of "accuracy checks".

- Determine the normal form "style" to use. The word "style" refers to the quadratic part of the Hamiltonian, which determines the form of the higher order terms. The "style" that we are most concerned with here is the quadratic part of the Hamiltonian associated with an equilibrium point of saddle-centre-...-centre stability type where the pure imaginary eigenvalues are non-resonant. Other normal form "styles" can be considered, e.g. resonance relations amongst the pure imaginary eigenvalues. See (5) for a discussion of the "normal forms" of quadratic Hamiltonians.

- As a first step the Normal Form algoriths will "Diagonalise" the system (via a symplectic change) so that the quadratic part, H_2, is in (real) Jordan canonical form. In this was we say that H_2 is in normal form.

For systems with many degrees-of-freedom, few, if any, of these steps are trivial!

For a large system, with a high dimensional phase space, and thus large numbers of variables, the order to which the calculations may be performed will probably hit other limits before accuracy becomes a problem. The first limits may be memory size, resulting in a memory allocation failure, which may be properly reported by your system or the software may just "crash". There may be simply an unacceptable run time, it takes too long to run for that system calculating results to that degree. The options are to try it and see how well it does and then consider various approaches to the problem:

- Initially consider a smaller problem and then rework the full problem.

- Do not use GMP initially – to get a faster smaller but less accurate solution.

- Get a bigger computer, faster processor more memory - but it is hard to get a significant gain.

- In theory a 64 bit processor should not have the same memory limitations - but this is not fully tested and still requires physical memory.

- Look at parallel processing to solve both problems

2.2 Simplified Outline of a Data-Flow Diagram Associated with the Normal Form Algorithm

Figure 2.1 shows a schematic outline of the structure of the normal form computation. The main Hamiltonian's involved are represented by the filled green ellipses and the main flow of computation involving these Hamiltonian's is indicated by the chain of bold arrows beginning at the top of the diagram. Code modules are represented by the filled gold boxes. Mappings of various kinds are shown as blue boxes. The arrows indicate the direction of the flow of data. The inner normalization module is contained within the box indicated.

The following "steps", and resulting "data" appear in Figure 2.1.

Steps (Orange):-

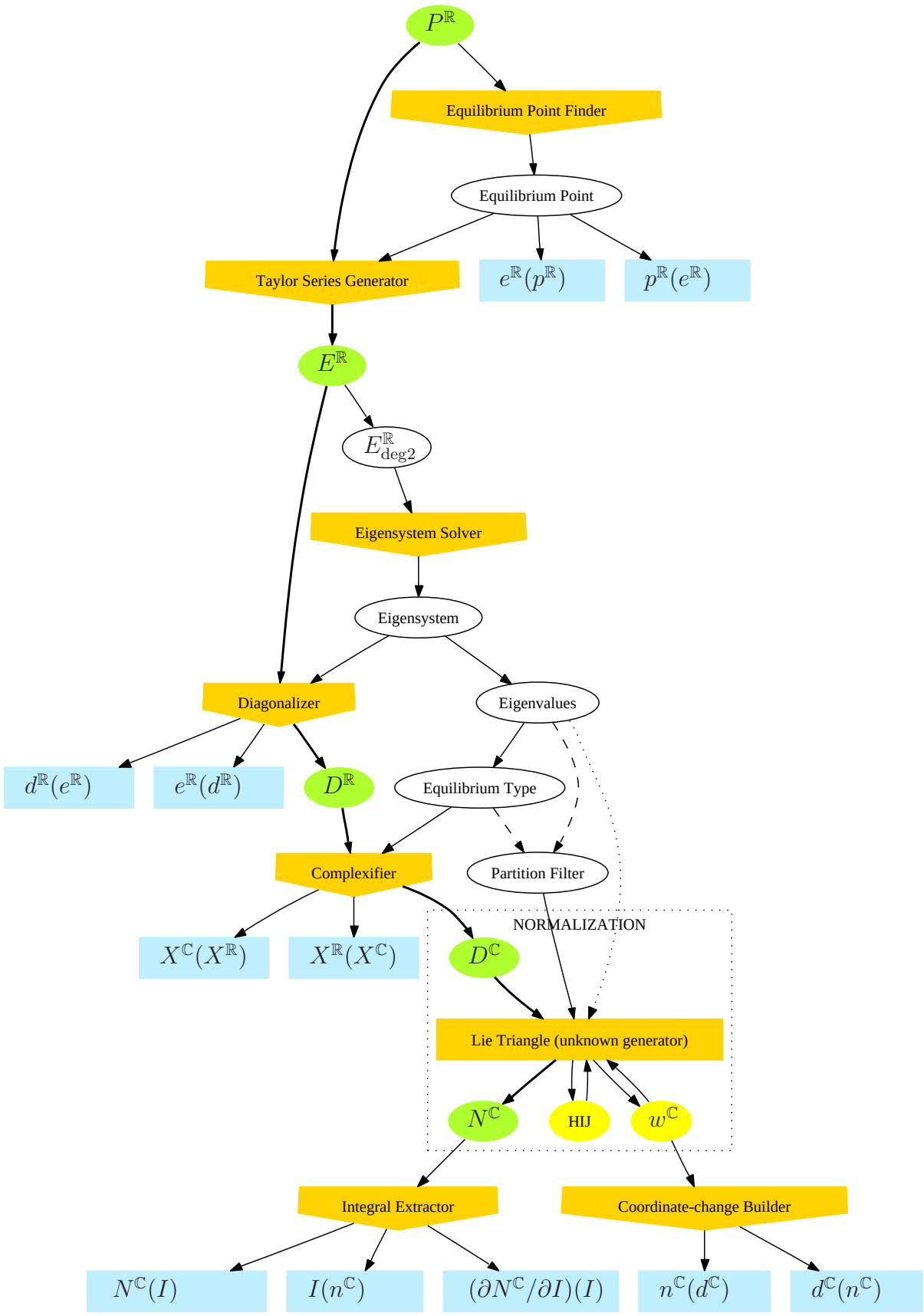

Figure 2.1: Structure of the normal form computation.

Offline calculations

- Find equilibrium
- Taylor expand
- Determine style

Normal Form calculations:-

- "Diagonalise" H_2
- Normalise
- Extract integrals
- Coordinate transforms

Data:-

- Hamiltonian terms (green)
- Data determining style (white)
- Intermediate quantities (yellow)
- Integral and coordinate maps (blue)

The following expressions appear in Figure 2.1.

$P^{\mathbb{R}}$ -the original Hamiltonian in real "physical" coordinates, $p^{\mathbb{R}}$.

$E^{\mathbb{R}}$ -the Hamiltonian expressed in real "equilibrium" coordinates, $e^{\mathbb{R}}$, centered on the equilibrium point of interest.

$E^{\mathbb{R}}_{\text{deg2}}$ -the quadratic part of $E^{\mathbb{R}}$.

$D^{\mathbb{R}}$ -the real diagonalised Hamiltonian (that is, $E^{\mathbb{R}}$ under a linear change of coordinates that puts the quadratic part into real normal form) expressed in "real diagonal coordinates", $d^{\mathbb{R}}$.

$D^{\mathbb{C}}$ -the complex diagonalised Hamiltonian (in which the quadratic part is expressed as a complex (in fact, pure real and pure imaginary)-weighted sum over products of coordinates and their corresponding conjugate momenta), expressed in complex coordinates, $d^{\mathbb{C}}$.

$N^{\mathbb{C}}$ -the normalized complex Hamiltonian, expressed in complex normal form coordinates, $n^{\mathbb{C}}$.

$w^{\mathbb{C}}$ -the complex generating function which, being invariant under the transformation has the same expression in both $n^{\mathbb{C}}$ and $d^{\mathbb{C}}$.

$I(n^{\mathbb{C}})$ -the mapping from the complex normal form coordinates, $n^{\mathbb{C}}$, to the integrals of motion for the normalized Hamiltonian.

$N^{\mathbb{C}}(I)$ -the mapping from the integrals of motion for the normalized Hamiltonian to the normalized Hamiltonian itself.

$(\partial N^{\mathbb{C}}/\partial I)(I)$ -the mapping from the integrals of motion for the normalized Hamiltonian to the frequencies of motion for the modes of the normalized Hamiltonian.

The following coordinate-change mappings appear in the figure:-

$e^{\mathbb{R}}(p^{\mathbb{R}})$ -the coordinate change from $p^{\mathbb{R}}$ to $e^{\mathbb{R}}$.

$p^{\mathbb{R}}(e^{\mathbb{R}})$ -the inverse coordinate change, from $e^{\mathbb{R}}$ to $p^{\mathbb{R}}$.

$d^{\mathbb{R}}(e^{\mathbb{R}})$ -the coordinate change from $e^{\mathbb{R}}$ to $d^{\mathbb{R}}$.

$e^{\mathbb{R}}(d^{\mathbb{R}})$ -the inverse coordinate change, from $d^{\mathbb{R}}$ to $e^{\mathbb{R}}$.

$X^{\mathbb{C}}(X^{\mathbb{R}})$ -the general change from real to complex coordinates for any coordinate system $X^{\mathbb{R}}$.

$X^{\mathbb{R}}(X^{\mathbb{C}})$ -the general inverse coordinate change, from $X^{\mathbb{C}}$ to $X^{\mathbb{R}}$.

$n^{\mathbb{C}}(d^{\mathbb{C}})$ -the coordinate change from $d^{\mathbb{C}}$ to $n^{\mathbb{C}}$.

$d^{\mathbb{C}}(n^{\mathbb{C}})$ -the inverse coordinate change, from $n^{\mathbb{C}}$ to $d^{\mathbb{C}}$.

2.3 Implementation & Methodology

These are some of the main features of the Implementation.

- Main programming in C++; fine control over aspects.

  The main part of the sofware running time is the solution of the Lie Triangle This is programmed in C++ for speed and can be run as a process from the Python code. This gives us acceptable processing times, so far, with the flexibility of Python for the less time constrained parts of the process and the overall control. The Python code still contains a Lie Triangle solver for testing and use in small, non time critical problems.

- OOP & template techniques for flexibility & speed.

  General Object Oriented design is used for better code reuse

- Efficient monomial & polynomial representations.

  A variety of polynomial representations are available, with different advantages and disadvantages

- Fast multi-precision arithmetic; any number of bits.

  The Gnu Multi-Precision (GMP) code can be used where required for accuracy.

- Fast prototyping in Python; flexible design.

  Python is used for non time critical code to allow rapid prototyping for new problems and flexible design changes

The devellopment Methodology adopted was a test-driven development using the open source Unit Test techniques and software

Test-driven development has been employed, at least for the low level routines and structures, with Unit Testing using: PyUnit & CPPUnit. Around 800 unit tests are available to help to ensure integrity.

The advantages of the testing are:

- unit tests are write-once, use many times.

- ensures that the code produces the correct answers.

- allows us to quickly test out new methods and representations.

- they can be written once and run many times across different versions of the same routine or different data structures. As we go to larger and larger systems, the requirements of the data structures change (e.g. using arrays for powers is fast and compact for small systems, but sparse strutures can be better for larger ones, depending on the number of terms in the polynomials)

- they can be used many times across different algorithms designed to perform the same task more efficiently on different sized systems

- we use some external libraries; tests ensure that when we install new versions of the these libraries, they still produce correct results and obey the assumptions that our code places upon them.

- for large systems, we will likely need to run our code on a cluster, but it is being developed on different machines, for example on a laptop and a desktop computer; tests ensure that things behave correctly on the different architectures.

- we also use different versions of linux and, in particular, the kernel and glibc, on these different machines; again, the tests ensure that the code works the same way on all of them.

Implementation note oncerning the “generator” in Deprit’s method:-

The following observations note IMPORTANT differences between the method of Lie series exemplified by Deprit’s method, and other methods (for example, Hori’s method, the method of Dragt-Finn, etc.) See reference [1], below, for more details.

[c(i)] In Deprit’s method, the ”generator” w appearing in the Deprit triangle is NOT the generator of the flow between the two coordinate systems in the usual sense.

[c(ii)] In particular, the ”generator” w is NOT in general invariant under the coordinate transformation defined by the Deprit triangle! It is very important to understand this point. [d(i)] In Deprit’s method, the negative (-w) of the ”generator” (w) is NOT the generator of the inverse (transform/flow). [d(ii)] Instead, one must apply the Deprit triangle recurrence ”in reverse” using the original (direct) ”generator” in order to get the inverse transformation. This is the reason for the methods forwardTriangle and reverseTriangle in what follows.

[c(iii)] Instead, the proper test for correctness of implementation is one that tests both the normalisation Deprit triangle and the direct coordinate change triangle together. In other

words, let the normalisation triangle applied to h yield h' with "generator" w, and the direct coordinate change triangle applied to x yield x' under the same "generator"; then the test for correctness is h'(x') == h(x). Suppose that the inverse triangle (using the same "direct" generator) maps y' to y, then we test this via y(x') == x.

2.4 Software Issues

2.4.1 Space / Time / Accuracy Trade-offs

A substantial effort has been made to address the multiple Trade-offs required in the software between Space / Time / Accuracy / Degree / Flexibility.

To state the problem simply: a large system converted to normal form using a high degree for the power series expansion of the Hamiltonian involves a lot of processing.

In addition the large number of possible terms in a high order multivariate polynomial takes up a substantial ammount of computer memory. Polynomials of a million terms are perfectly possible and at about that stage even a high end computer starts to run out of memory. The grace with which this is handled depends on the operating system but in any case the program will crash and/or stop.

Thus a large problem will both require lots of memory and take a long time. The limits on this will depend upon the polynomial representation which in itself trades off Space / Time. In addition calulations with large numbers of large polinomials require higher accuracy which may require multiprecision numerical calulations increasing requirements for both computer memory and processing time.

Although many options have been tried, and the facility to use them is generally still in the code, little concrete advice is possible since the optimal choices depend on the particulars of the problem being considered. At the upper limits, disk based storage of polynomials (or even parts of polynomials) may be the final option and should, in theory, allow the processing to work on any size system but the processing time will be large, though a large RAID array may make it acceptable.

2.4.2 Numerical Precision

Because the calculations are performed in terms of multivariate polynomials of increasing degree the potential number of terms in each polynomial can become very large. Because of the non linear transformations involved, often the coefficients of most of these terms can be significant. Even a relatively simple hamiltonian with few terms can have significant values for most possible polyinomial terms after diagonalisation. The nonlinear cordinate transforms can often have an almost maximum number of terms. The calculation accuracy required to prevent significant error accumulation during the addition and multiplication of many large polynomials can be surprisingly high. During the development it was found that in some cases normal double precision real numbers was insufficient and there was a need for the use of the GNU multiprecision library, or similar, to give higher accuracy. This was impimented in the C++ which can perform the polynomial calculations to the required accuracy. GMP numbers, by their nature, require more memory and processing time. The

python diagonalistion calculations has not been converted to the use of GMP since this gives a significant reduction of speed, GMP versions of the eigenvector routines are not readily available and standard double precision seems to be sufficiently accurate at present.

2.4.3 "Efficient" Monomial and Polynomial Representations

During devellopment a number of different representations have been used for products of powers (monomials) and for polynomials. The choice has several different implications, not just for storage size in memory or on disk, but also in the need to access & compute with them and the implications in processing time, memory footprint and import resources. Unfortunately the trade offs are complex and optimum solutions depend on the system being studied. In general the options chosen allow an initial look at a problem, say only looking at low order or low dimensions. Some examples are given below:-

At present a polynomial is dictionary of coefficients indexed by the indices of the variables. The elements of the dictionary, the coefficients, may be real or complex, double precision or GMP. The indices, powers of the variables, are generally integers (whatever that means in the compiler or python version). They have in the past been smaller, even nibbles (4 bits, 0-15 values), but the space saved was not worth the extra processing involved, this would involve very careful coding to make it useful. It may be most useful for offline storage or for temporary disk based storage of intermediate results.

There are a number of ways of representing the "variable" part of a monomial element of a polynomial. Since the variables are in a predetermined order what is needed is the list of the indices of all the variables. This can be done in several ways.

- "dense" (full basis), Simply store a tuple (list) of the indices. For an N dimentional phase space this is N integers.

- "sparse", Only store non zero indices. However this requires that with each non zero index there is an integer indicating which variable that index applies to. Thus if all variables have non zero indices the storage space is doubled. The code to achieve this is quite neat and adds little additional complexity. However it is often the case that very few monomials have non zero indices especially for high degree polynomials and so for the general case there may be an increase rather than decrease in storage space. For a large dimensional phase space with low degree the trade off may be more advantageous. In general this is probably a loss unless the hamiltonian and diagonalisation are simple.

- "total ordering indexed", Since the maximum number of terms in a polynomial (of a certain degree in N variables) may be calculated and also listed, it is possible to simply list the coefficients and the integer index into that list implicitly gives the powers of the variables. Clearly this is a massive space saving. It is a matter of choice as to whether all coefficients are stored or simply the non zero coefficients. This has been impimented and tested however polynomial multiplication then required conversion between this single integer representation and into a list of indices, which is computationally intensive and increased processing time by an order of magnitude. More computationally efficient polynomial multiplication could be implimented but this has not yet been done.

Plus distributed variants of the above.

If we progress to a large dimensional phase space with high degree polynomials it rapidly reaches the stage where the polynomials will not fit in computer memory and parallel distributed processing is required to start a number of processes which can each fit all or part of a polynomial in memory to process it. It may be that such polynomials will use different representations depending on their properties.

2.4.4 Some object-oriented hierarchies

Specific classes were develloped for polynomial ring, graded lie algebra base, classical and semi-classical algebras. The polynomial class can be implimented in a number of ways to optimise performance for a class of problems.

The various polynomial (and powers) representations can be implemented in many different ways. For example:- dense, sparse, distributed methods, and also unordered list, ordered list, ordered hash, binary tree for sparse implementation methods.

2.4.5 Parallelisation

When larger systems are considered, code run times will become longer and the use of parallel code running on numerous CPU's or Cores may be required. At least the following parts of the code are parallelisable:-

Coordinate changes	COARSE	HIGH	by component
Poisson bracket	FINE	HIGH	by summands
Lie triangle row	COARSE	MEDIUM	by Poisson brackets
Polynomials	VERY FINE	MEDIUM	by +,-,*

Coordinate changes For an n-DoF system, there are $2n$ Lie triangles to compute for the complex coordinate changes. Input to each is the generation function and the index to indicate which component should be computed. Each component can therefore be computed by a different process.

Poisson bracket We can parallelise over the sum; each process computing 4 derivatives and a difference.

Lie triangle row We can view each intermediate quantity in the Lie triangle as a bunch of Poisson brackets and a pre-multiplier. The brackets can be computed, *partitioned*, and the linear components of the Homological equation solved, separately. Since, for large systems, we are unlikely to be able to go to especially high degree (and thus will not have too many brackets to compute), this is a low priority parallelisation.

Polynomials We can parallelise over the processing of each Polynomial. It may be that as the memory requirements for each polynomial become large there may be a need to hold intermediate polynomials on disk. Each polynomial add/subtract/multiply could then be performed in a separate process. This would give very fine grained parallel processing but the disk overheads would make this acceptable only for large systems with no other choice.

[remove this section, maybe move a few bits or rewrite chunks]

2.5 Some features of the code

Some aspects worthy of note in the software: In the following [ma], [py], and [c++] indicate in which languages these features have been implemented.

STRUCTURES:

the following structures have been implemented:

- implement pure-powers objects and polynomials [py, c++].
- allowing all ring operations and also special operations such as efficient subsitutions [py, c++].
- efficient implementation of powers operator, using repeated squaring [py, c++].
- over arbitrary coefficient types [c++].
- to arbitrary accuracy [c++].
- generation of compatible arbitrary precision complex type from real type [c++].
- in both dense and sparse formats [py, c++].
- investigated using various formats for efficiency [py, c++]:
  – dense representation for polynomials and powers [py, c++],
  – packed representation for powers, packed into single longs [py, c++].
  – efficient hash-bucket representation for both powers and polynomials [py].
  – disk-based representation via hash-bucket files [py].
- polynomial rings [py, c++].
- lie algebras, both classical with poisson bracket and semi-classical with moyal bracket [py, c++].

ALGORITHMS:

the normal form code has implemented the following modules for arbitrary degrees of freedom:

- import/export the various coordinates, polynomials, and transformations, in a file format suitable for all modules [ma, py, c++].
- compute the eigensystem of the quadratic part of the hamiltonian [py]. fixed accuracy [py].
- perform the real diagonalisation [py].
- perform the complexification [py].
- perform the lie triangle, computing both the transformed hamiltonian and the generating functions [py].
- handle partial normalisation in the lie triangle [py].
- allow the lie triangle computation to be restarted for higher degree [py].
- extract the simple integrals from the normalised hamiltonian [py].
- extract also the non-simple integrals in the case of partial normalisations [py].
- express the normalised hamiltonian as a polynomial over all the integrals [py].
- compute all the coordinate changes via lie transform [py].
- re-implemented all existing normal form systems, e.g., Hill's equations in 3-dof, and verified that the new code produces the same results [py].
- implemented system bath in both original and mass-weighted forms [py] with random frequency distributions.

1. Handles any Degree Of Freedom (DOF).

2. Handles any number of saddles and centres, i.e., (saddle^m $\times$ centre^n).

3. Objects can be in memory or on the disk using the same code. In fact, any list-like or dict-like objects can be used to represent the homogeneous parts of the various polynomials and the intermediate quantities in the Lie triangles.

4. Flexible polynomial representation in terms of the underlying basis monomials, ability to change which representation we use (for example, a compressed version runs more slowly but consumes less memory).

5. A polynomial can now be evaluated over a vector of polynomials from a different polynomial ring, which means that one can perform substitutions (e.g., evaluating an expression for the normalised Hamiltonian in terms of simple and non-simple integrals at a vector of polynomials that represent the integrals themselves results in an expression for the Hamiltonian in terms of the normal form coordinates).

6. Fast arbitrary precision floating point capability is done and tested. this now needs to be put into the diagonalizer, which means using a dedicated eigensystem solver (see also below). i also added the capability to "pickle" the resulting numbers, for use with on-disk storage.

7. Can compute higher degree terms without re-computing lower degree ones.

8. Ability to load files from our old Mathematica format, so that we can compare with familiar systems.

9. Computes the eigensystem and diagonalizes the input Hamiltonian Taylor series.

10. Performs the real-complex transformation.

11. Performs the normalization (results agree so far with 3-DOF model systems).

12. Can perform partial normalization according to criteria that we specify (e.g., for "near-resonance").

13. Automatically extracts all simple and non-simple integrals and expresses normalized Hamiltonian as a polynomial over them via a new module, even in the case of partial normalization. this is done efficiently by directly analysing the normalized Hamiltonian and makes no assumptions on the system.

14. The coordinate transform is a new module, which has been designed to be used in parallel (if we are, e.g., normalizing a 200-DOF system then computing the coordinate transformation requires computig 400 maps, one for each phase space coordinate. Each of those maps is computed by a Lie triangle which is of pretty much the same complexity as the computation of the normalized Hamiltonian itself). the design will allow different processes/processors to compute different components of the coordinate change.

15. Much of the system is self-testing for robustness (most methods check their arguments and return values).

16. I have written over 300 unit tests and functional tests. when portions of code are rewritten in C++ and wrapped, there is no need to re-write the tests; we can simply use the same ones and thus ensure that the new module acts as the old one did. when modules are exchanged for new, more efficient ones, we simply re-run the tests and can see instantly if anything has been broken by the change.

2.6 Software Description

2.6.1 Overview

The definitive description of the code is the source itself but we give here an outline guide:

- brief Overview description of the software
- Fuller description of the software
- Environment and directory structure
- brief description of the Python source code files
- brief description of the C++ source code files
- brief description of the configuration and run software
- brief description of the configuration options

2.6.2 source code description

This is a brief description of the source code in the distribution:

The core of the configuration and run software is in the Python code file RunConfig.py in the directory nf–unified–0.5/src/config-run. This has a default set of configuration options and if run on its own will perform an example run. It can be preloaded in various ways with alternative configuration options to perform calculations or load different Hamiltonians. It can be called by an external Python file which sets up a Hamiltonian first. Examples of such files are given in the directory.

The input to the Normal Form software itself is a Hamiltonian polynomial which is a taylor series at the equilibrium point.

RunConfig.py, under the control of the configuration options, will call Python routines to diagonalise the Hessian matrix and produce a diagonalisation transform and a complexification transform. The steps are as follows:-

- confirmation that we have a valid equilibrium point,
- diagonalization of the equilibrium Hamiltonian into real normal form,
- complexification of the real block diagonal system into the complex one,

The Python code can print out a Hamiltonian as a set of Taylor series homogeneous polynomials, together with files of polynomial vectors to do the coordinate conversions from equilibrium to diagonal and from real to complex. These can be used by the C++ code.

Again under the control of configuration options, either Python or C++ code (or both) will be called to use Lie triangle calulations to obtain the normal form Hamiltonian and the corresponding coordinate transforms. The steps are as follows:-

- normalization to complex normal form,

- extraction of the complex integrals,
- realification of the normal form,
- realification of the integrals,
- confirmation that the generating function is invariant under the coordinate changes,
- confirmation that the coordinate changes really do transform the diagonal Hamiltonians into their normalized counterparts and vice versa.

2.6.3 Setup

In this description the standard Python notation of e.g. "RunConfig.compute_diagonalisation" will normally refer to an object or routine compute_diagonalisation in the file RunConfig.py but equally may refer to an object's method etc.

RunConfig.py after the initial setup calls two other routines within the file. RunConfig.compute_diagonalisation() and RunConfig.compute_normal_form() which then run code from the src/py directory.

The setup code in RunConfig.py starts profiling and logging. In the setup a NormalForm object (see src/py/NormalForm.py) called nf is initialised with the hamiltonian h_er in real equilibrium coordinates.

The input is a hamiltonian polynomial at the equilibrium point Give confirmation that we have a valid equilibrium point by checking 1st order terms are zero and setting the constant to zero The hamiltonian is truncated to order 2 for diagonalisation RunConfig.compute_diagonalisation() after these checks will run the nf.diagonalize() method

2.6.4 main code areas

A brief description of the main areas of the code is given here:

diagonalize

The diagonalization code is called from NormalForm.diagonalize() after initialising self.dia as a Diagonal.Diagonalizer object, it runs:

dia.compute_eigen_system(self.h_er, self.tolerance)

dia.compute_diagonal_change()

mat = self.dia.get_matrix_diag_to_equi()

assert self.dia.matrix_is_symplectic(mat)

er_in_terms_of_dr =

dia.matrix_as_vector_of_row_polynomials(mat)

- compute_eigen_system calculates the linear_matrix from the hessian_matrix_of_quadratic_part using:

  [eigenvalue_eigenvector_pairs]

  [self.linear_matrix]

  [skew_symmetric_matrix] this does the setup of the matrix J to be the multidimensional form of:

  ```
  { 0 1}
  {-1 0}
  ```

  [hessian_matrix_of_quadratic_part] This takes partial differential twice of the truncated Hamiltonian to get the Hessian matrix and multiply J by the hessian

- compute_eigen_system then calculates the eigensystem using the eig routine from MLab.py.

  [eig(linear_matrix)] Get the eigensystem as e-value e-vector pairs [calls MLab.eig -¿ LinearAlgebra.eigenvectors -¿ lapack_lite.[zd]geev] This uses a lite version of LinAlg.py module which contains a high-level Python interface to the LAPACK library. The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real. (from zgeev)

  [check_eigen_value_vector_pairs] Check that they work (JHess .x = l.x) and print error

  [purify_eigenvalue] zero the smaller of real or imag part of each eigenvalue since each eigenvalue pair should be either real or imaginary

  It then reorders them as eigenvalue/eigenvector pairs using routines from EigenSystem.py.

- compute_diagonal_change() uses analyse_eigen_system() from EigenSystem.py and then runs _compute_symplectic_matrix_diag_to_equi to build a symplectic_matrix from the eigenvalue/eigenvector pairs.

  [analyse_eigen_system() from EigenSystem.py] does:

  self._truncate_eigen_values_and_vectors()

  self._sort_eigensystem_reals_first()

  self._collect_plus_minus_pairs()

  self._determine_equilibrium_type

  [_truncate_eigen_values_and_vectors()] Round the eigenvalues and vectors to a tolerance, normally 1.0e-15

  [_sort_eigensystem_reals_first()]

  [sort_by_ranking_function] Sort eigenvalues, real before imaginary, decreasing real magnitude, decreasing imaginary magnitude, so as to pair up equal-magnitude but opposite-sign pairs with the positive partner

[_collect_plus_minus_pairs()]

[positive_member_first] Given two EigenValueVectorPairs, left and right, return the pairs ordered so that the positive (or positive imaginary) eigenvalue pair comes first.

[_determine_equilibrium_type()] For each pair of eigenvalues a real e-v is a saddle an imaginary e-v is a centre

[_compute_symplectic_matrix_diag_to_equi] Choose the vectors a, b in each plane to go into the coordinate transform matrix. saddle - use the real e-vecs from the pos and neg e-values centre - use the real and imaginary parts of the e-vec for the pos e-val error for zero e-values

– _compute_symplectic_matrix_diag_to_equi does: For each plane calculate the difference of the coordinate-momentum product: a[0]*b[1] - a[1]*b[0] sum this across the degrees of freedom for the a,b in a vector pair to give c. if c is negative swap a and b (c is now positive) (This is what gives the sign difference for a saddle point my suggestion is for a saddle point just negate b) scale a and b by 1/sqrt(c) put these vectors into columns of a matrix and transpose it

[matrix_is_symplectic] Check to a tolerance of 1.0e-12 that: The determinant is 1

```
Calculate and check,  ( ' =transpose)
m.J.m' = J
Partition the matrix into coordinate-momentum parts as
{ a b }
{ c d }
(i.e. convert from x0p0x1p1.. to the conventional x0x1..p0p1.. )
Calculate and check that,
ad' - bc' = I
ab' - ba' = 0
cd' - dc' = 0
Note that
{ a b } . { 0 1} . { a' c' } =  {ab'-ba'  ad'-bc'}
{ c d }   {-1 0}   { b' d' }    {cb'-da'  cd'-dc'}
and cb'-da' = -(ad'-bc')'
```

so the last three checks are equivalent to m.J.m' = J store as the diagonal to equilibrium coordinate transform The matrix which maps a vector in the real diagonal coordinate system into the corresponding vector in the real equilibrium coordinate system.

[dia.matrix_is_symplectic] Check that the matrix is symplectic again

[matrix_as_vector_of_row_polynomials] Express the matrix as a list of linear polynomials, each representing one row

[self.h_er.substitute(er_in_terms_of_dr)] Apply (substitute) this to the hamiltonian

Check that the imaginary part of the hamiltonian is small. Remove it to make the hamiltonian real Remove terms with small coefficients from the hamiltonian

- get_matrix_diag_to_equi() returns the matrix which maps a vetor in the real diagonal coordinate system into the corresponding vector in the real equilibrium coordinate system.
- matrix_as_vector_of_row_polynomials(mat) converts the matrix to a vector of polynomials which achieve the same result.

complexify

[NormalForm.complexify] [

com = Complexifier(lie_algebra, eq_type) instantiate com from class Complexifier

r_from_c = com.calc_sub_complex_into_real()

c_from_r = com.calc_sub_real_into_complex() Note this uses p,q from class Polynomial-LieAlgebraBase(PolynomialRingInterface):

set up routines to convert polynomials from real to complex and back?

[com.calc_sub_real_into_complex] for each plane take q, p as the position and momentum monomials for a centre return p -iq and q-ip both divided by sqrt 2 use this as a conversion from real to complex

[cvec2r_to_rvec2r cvec2cvec_to_rvec2rvec] This is used to convert (real) polynomial functions of complex coordinates into polynomial functions of real coordinates

[com.calc_sub_complex_into_real] For a conversion from complex to real do as above but return p+iq and q+ip both divided by sqrt 2 This is used to write an "–r-from-c.vpol" file which can be used by the C++ code, and also to convert the Hamiltonian from real to complex.

complex_into_real is a polynomial for real in terms of complex coordinates. Substitute this complex to real transform into h_dr (the polynomial for the original Hamiltonian in terms of real diagonalised coordinates) to give h_dc (the polynomial for the original Hamiltonian in terms of complex diagonalised coordinates)

Remove small coefficients.

Examine the quadratic terms of h_dc, check that any off diagonal terms are below a tolerance level and remove them.

The mathematics of the complexifier is included here for completeness. It is easy to make a mistake with the signs or the direction of convertion.

```
Complexifier
As in notes 17/1/04
{Q} = __1__ {  1 -i } {q}
{P}   sqrt2 { -i  1 } {p}
```

```
{q} = __1__ { 1  i } {Q}
{p}   sqrt2 { i  1 } {P}

Note these are mutual inverse, Det = 1

Note that for the centres:

1/2 ( q^2 + p^2 ) = iQP

{qdot} = { 0 w }  {q}
{pdot}   {-w 0 }  {p}

{Qdot} = __1__ {  1 -i } { 0 w } __1__ { 1 +i } {Q}
{Pdot}   sqrt2 { -i +1 } {-w 0 } sqrt2 { i +1 } {P}

{Qdot} =_1_{  1 -i } { iw   w }  {Q}
{Pdot}   2 { -i +1 } [ -w -iw }  {P}

{Qdot} = { iw  0 }  {Q}
{Pdot}   { 0 -iw }  {P}
```

normalisation

NormalForm and normalisation

The above diagonalize and complexify functionality is implimented in Python but not yet in C++. The Python code can print out a Hamiltonian as a set of Taylor series homogeneous polynomials, together with an "–r-from-c.vpol" file and a "–e-from-d.vpol" file to do the coordinate conversions from equilibrium to diagonal and from real to complex. These can be used by the C++ code.

The functionality described next is the Python normaisation. The C++ code is similar. They are believed to be functionally identical.

In RunConfig.py the compute_normal_form() functiom uses the NormalForm object nf to initialise the tolerance and then run the method perform_all_computations() which itself does some initial setup:

self.diagonalize()

self.complexify()

self.normalize()

self.normalize_real()

self.extract_integrals()

it then calculates and checks the coordinate transforms between diagonal and normal forms and prints them out.

compute_normal_form() then writes out to files the Normal Form series and the Generating Function series

The subroutines are as follows:

[NormalForm.normalize]

Initialise the Lie triangle object tri from class LieTriangle.LieTriangle :

[IsogradeInnerTaylorCoeffs] compute the polynomial factorial conversion coefficients.

[check_quadratic_part] Check that the quadratic Hamiltonian has correct no of variables,

[alg.is_diagonal_polynomial] zero constant and linear part and is diagonal

[extract_fundamental_frequencies] Extract the coefficients of the complex diagonal quadratic Hamiltonian

then normalize() performs the normalisation

First do the initial setup of 2nd degree terms for the Lie triangle. For each desired degree (grade) copy the homogeneous polynomial of that degree from the complex Hamiltonian. compute_normal_form_and_generating_function() will use the Lie triangle recurrence relation to calculate all the terms of the Lie triangle at that degree and the terms of the generating function and normalised hamiltonian.

[compute_normal_form_and_generating_function] multiply the homogeneous polynomial from the complex Hamiltonian by n!. Prepare data to compute the current row of the Lie Triangle.

[triangle_minus_unknown_term] Compute one line of the intermediate Deprit triangle quantities using the known terms H_i^j and w_i (from existing lines) without any unknown term. The current value of the generating function $w_n + 1$ is not known at this stage, it will be calculated later from the homological equation. It is set to zero. Compute each term H_i^j from the recurrence relation recursively.

[known_terms.partition(self.filter)] Partition the resulting H_0^n into normalised terms (powers of axp terms) and the remainder which will be removed by a suitable generating function,

[_correct_row_using_partition] Return to the row of H_i^j terms and correct them by the values to cancel out the remainder terms.

[solve_homological_eqn] Calculate the generating function to cancel out the remainder terms. For each term in the remainder, sum the difference in x,p powers multiplied by the frequency and divide the term by this.

[_check_homological_equation] Check that bracket(k_2,$_w$ [i]) cancels the remainder terms.

generation of output transforms

[self.extract_integrals()] [find_lost_simple_integrals_and_non_simple_integral] From the normal form Hamiltonian sort the simple integrals, those with equal powers in x and p. Everything else is non_simple_integral

[optional computations:

self.check_generating_function_invariant()

self.check_diag_transforms_to_norm()

self.compute_diag_to_norm() Call CoordinateChange.express_norm_in_diag for exch component of x convert to poly and store in self.nc_in_dc

self.compute_norm_to_diag() Call CoordinateChange.express_diag_in_norm for exch component of x convert to poly and store in self.nc_in_dc NB This is WRONG

]

coordinate change maps:

[self.compute_diag_to_norm_real_using_w_real()] Call CoordinateChange.express_norm_in_diag for exch component of x convert to poly and store in self.nr_in_dr_via_wr Calculate and log errors

[compute_diag_to_norm_real_using_conversion] self.nr_in_dr_via_nc_in_dc = self.cvec2cvec_to_r Calculate and log errors

[self.compute_norm_to_diag_real_using_w_real()] Call CoordinateChange.express_diag_in_norm for exch component of x convert to poly and store in nothing Calculate and log errors

[compute_norm_to_diag_real_using_conversion] self.nr_in_dr_via_nc_in_dc = self.cvec2cvec_to_r Calculate and log errors NB This is WRONG

[optional computations:

self.compute_diag_to_norm_real_using_conversion()

self.compute_norm_to_diag_real_using_conversion()

self.write_out_integrals()

self.write_out_transforms()

]

2.6.5 C plus plus

The C plus plus code can be called as a process via a system command from RunConfig.py after the python code has calculated the diagonalize and complexify stages, to double precision, and prited out files of the results

Initially from SystemBathExample.cpp the codes is as follows:

report on context

load diagonalisation polynomials

temporary: zero constant term on taylor series (special)

Read in –h-from-e–grade-0.pol

Check that it is in correct form and is zero

temporary: zero linear term on taylor series (special)

Read in –h-from-e–grade-1.pol

Check that it is in correct form and is zero

temporary: deal with quadratic part (special)

Read in –h-from-e–grade-2.pol

Check that it is in correct form and is degree 2.

import the complexification maps read in the complexification maps combine the diagonalisation polynomials and complexification maps

real-diagonalise the term by applying the transformation

remove off-diagonal part and confirm complex diagonal form

DepritTriangle normalisationTriangle Setup the Deprit triangle

handle the quadratic part specially; we diagonalised it already normalisation file output else diagonalisation normalisation

normalisationTriangle.computeNormalFormAndGenerator(hFromDCG, kTerm, wTerm);

prepare hamiltonian term, removing factorial divisor to give hI

Calculate one line of the Deprit Triangle Compute each term HIJ from the recurrence relation recursively. ignore generating function now , will calculate it later

triangleMinusUnknownTerm

partitionKnownTerms_(knownTerms, normal, remainder);

correctRowUsingPartition_(normal, remainder);

solveHomologicalEquation_(remainder)

calculate generating function

checkHomologicalEquation_(remainder);

report on some sizes

return the polynomial terms, correctly weighted by the preFactor

file output

file output

2.6.6 Environment and directory structure

[ref stuff elsewhere?]

2.6.7 Python Software files

Files from:
nf–unified–0.5/src/py Unit test routines are shown with the files they test without further comment

Accuracy.py AccuracyTest.py Various routines associated with numerical accuracy - truncation

Complex.py ComplexTest.py Implement a few helper methods for complex numbers and vectors.

CoordinateChange.py CoordinateChangeTest.py Compute the coordinate changes relating complex diagonal to complex normal form coordinates.

Diagonal.py DiagonalTest.py Diagonalization of the quadratic part of the Hamiltonian using the Eigensystem.

EigenSystem.py EigenSystemTest.py Eigensystem calculation import and adjustment for Diagonalization.

EigenValueVectorPair.py EigenValueVectorPairTest.py Trivial class to hold a pair of eigenvalue and associated eigenvector.

EquilibriumType.py EquilibriumTypeTest.py Define shorthand for equilibrium point types and provide pretty-printing.

EstimateBounds.py Factorial, binomial coeffficients routines

GmpTest.py Test the basic Python wrappers to the GNU Multi-Precision library. gmpy module

GradedInterface.py Interface for graded algebras.

ImportAllOptimized.py Precompile all Python modules, optionally in optimised compile mode.

IndexPowersTest.py

IntegralExtractor.py IntegralExtractorTest.py Extract the complex integrals of motion from a complex normal form.

IsogradeInnerTaylorCoeffs.py IsogradeInnerTaylorCoeffsTest.py Implement the relationship between a list of "inner" Taylor series coefficients and the corresponding polynomial, in which these

LieAlgebra.py LieAlgebraTest.py Classes to implement graded Lie algebras, based on polynomial rings.

LieTriangle.py LieTriangleTest.py Implements the Lie Triangle for normalization of a complex diagonal Hamiltonian with unknown generating function.

makefile

NameLookupSemanticsTest.py

NewTest.py

NormalFormIO.py Collect together software to read in and write Normal Forms both from C++ and as ascii files from Mathematica.

NormalForm.py NormalFormTest.py The various steps in the overall normalization algorithm, collected together into one class.

PackedPowers.py PackedPowersTest.py Powers built using the packed tuple class.

PackedTuple.py PackedTupleTest.py Tuples of Ints packed together

Polynomial.py PolynomialTest.py New class representing multivariate polynomials with fully tested Ring logic.

PolynomialRingIO.py PolynomialRingIOTest.py Provide output routines for polynomials and polynomial rings in bare ASCII and S-expression formats.

PolynomialRing.py PolynomialRingTest.py An interface and straightforward implementation for polynomial rings.

PowersBase.py A product of coordinates raised to exponents, represented by a tuple of the exponents.

Powers.py At present, this is a nasty mechanism for switching between powers representations.

SemiclassicalNormalForm.py The semi-classical implementation of the normal form algorithm, which includes quantization of the Hamiltonian.

SExprIO.py Basic routines for writing and reading simple S-expressions.

ShelvedDict.py ShelvedDictTest.py Implement a dictionary-like object which resides in a file (shelf).

SparsePowers.py SparsePowersTest.py Implement sparse powers (multi-indices), using the sparse-tuple representation.

SparseTuple.py SparseTupleTest.py A sparse representation of (non-negative) integer tuples; we only store the non-zero elements.

SystemBath.py SystemBathTest.py Used to generate System Bath Hamiltonian, given number of bath modes.

Taylor.py TaylorTest.py Experimental code to implement cached lazy-evaluated Taylor series.

Test.py TuplePowers.py TuplePowersTest.py Implement powers (multi-indices) by a tuple of the exponents; this is the so-called dense representation.

UnitTestTemplate.py

Utility.py UtilityTest.py Miscellaneous utility functions, such as binomial coefficients.

Uuid.py UuidTest.py Generate unique identifiers for use as temporary file name prefixes.

2.6.8 C++ Software files

Files from: nf–unified–0.5/src/cpp

BasicTest.h Check that we can compile code in the current environment, and act as a reminder concerning, for example, the action of fabs.

BasicTestOld.h

ClassicalLieAlgebra.cpp ClassicalLieAlgebra.h ClassicalLieAlgebraTest.h Implement the classical Lie algebra by embedding the q and p components into a polynomial ring and defining the graded structure in terms of the homogeneous polynomial structure of that ring. The Lie bracket here is the usual Poisson bracket. The same implementation will work for semiclassical Poisson bracket which could be used to test the lowest order term of the semiclassical Moyal bracket; this would make a useful additional test.

ClassicalLieAlgebraTestPoisson.h

CoordinateChange.cpp CoordinateChange.h CoordinateChangeTest.h CoordinateChange implementation

CoordinateChangeOld.cpp CoordinateChangeOld.h CoordinateChangeTestOld.h

CppStaticInitializerExample.cpp This was going to be an experient with how various compiler standards deal with initialisation of static member variables, but was later found not to be needed.

CppUnitTestTemplate.h

DefaultNumericalPrecision.cpp DefaultNumericalPrecision.h DefaultNumericalPrecision
Here, we provide a very simple facility for automatically-scoped precision changes to the GMP library.

DepritTriangle.cpp DepritTriangle.h DepritTriangleTest.h Implements the Deprit Triangle for normalization of a complex diagonal Hamiltonian via solution of the corresponding inhomogeneous homological equation, producing as a side-effect the "generator"

DepritTriangleKnown.cpp DepritTriangleKnown.h DepritTriangleKnownTest.h
Implements the Deprit Triangle for a known generator, for the transformation of coordinate selector functions.

GmpTest.h Tests for GNU Multi-Precision support.

GmpTestRunner.cpp GmpTestRunner.h

LieAlgebraBase.cpp LieAlgebraBase.h LieAlgebraTestBase.cpp LieAlgebraTestBase.h
Define the base class (not interface; it has data members!) to all graded lie algebras in our code. This differs from the Python implementation: for simplicity, I do not introduce the concept of grade into PolynomialRing, only at this stage in the LieAlgebra. This removes an extra level of the inheritance

MapPowers.cpp MapPowers.h MapPowersTest.h A product of pure (non-negative) powers, represented efficiently by using a map from coordinate indices to non-zero powers; especially suitable for large numbers of variables. Multiplying two such objects (having the same number of variables) results in adding together the powers componentwise, for example.

Polynomial.cpp Polynomial.h PolynomialTest.h PolynomialTestNotImplementedYet.h P
Implement multivariate polynomials as maps from multi-indices (Powers objects) to coefficients.

PolynomialIO.cpp PolynomialIO.h PolynomialIOTest.h Implement input/output routines for polynomials in s-expression-like format.

PolynomialRing.cpp PolynomialRing.h PolynomialRingTest.h An implementation of PolynomialRingInterface with explicit numVars.

PolynomialRingInterface.cpp PolynomialRingInterface.h Abstract interface for all polynomial rings, including those that will be implemented by graded lie algebras.

PolynomialRingTestRunner.cpp PolynomialRingTestRunner.h Aggregate tests on polynomial rings into an executable test suite.

Random.cpp Random.h

Representations.h

SemiClassicalLieAlgebra.cpp SemiClassicalLieAlgebra.h SemiClassicalLieAlgebraTest.h Implement the semi-classical Lie algebra by embedding the q and p components into a polynomial ring with an additional h-bar variable and defining the graded structure in terms of the homogeneous polynomial structure of that ring plus counting h-bar twice.

SemiClassicalLieAlgebraTestMoyal.h Test the Lie (Moyal) bracket for the semi-classical Lie algebra.

SExprIO.cpp SExprIO.h SExprIOTest.cpp A few experimental routines for parsing of simple s-expressions.

StlInit.h StlInitTest.h It is the purpose of this little bit of code to allow us to fill values into STL vectors in a less painful way.

TestRunner.cpp TestRunner.h Aggregate tests suites together into an executable.

Types.cpp Types.h TypesTest.h Here we define the basic types and utility functions that will be used throughout the code. Some of this could be done more cleanly via templates, but for now this is a viable solution.

Utility.cpp Utility.h UtilityTest.h Here we define the basic utility functions that will be used throughout the code. Some of this could be done more cleanly via templates, but for now this is a viable solution.

2.6.9 Control Software files

Files from:
nf–unified–0.5/src/config-run

Compare.py Compare two Normal Forms and calculate the difference (error) between them from files. Read in and compare results calculated by the python and C++ software.

MakeNormalForm.cpp MakeNormalForm.h Import diagonalisation, Taylor series files from Python version and normalise, etc.

NfExample.py Provide various example configuration and Hamiltonians for the normalization algorithms. These are copied from projects/nf–unified–0.5/src/py/NormalFormTest.py.

NormalFormCompare.py Read in and compare Normal Forms calculated by this software with acsii files from Mathematica when available.

RunConfig.py Provide configuration and run control, with various options, for the normalization algorithms.

run_Crtbp.py RunCrtbp.py A sample setup and configuration for the normalization algorithms.

RunEckartasMMall.py RunEckartasMMclass.py RunEckartasMMdeg4.py RunEckartasM Example run of the semi-classical normal form procedure, applied to the EckartasMM uncoupled system from Holger.

RunHill.py RunHilltest.py A sample setup and configuration for the normalization algorithms.

RunSystemBath.py A setup and configuration for the SystemBath normalization algorithms. Read in the degree and the number of bath modes as parameters.

SystemBathExample.cpp SystemBathExample.h SystemBathExample.py Import system-bath files from Python version and diagonalise, normalise, etc.

2.6.10 Configuration Options

A Description of the Configuration Options The default set of Configuration Options is

```
self._aconfig = { "tolerance" : 5.0e-14 ,
                  "degree" : 10 ,
                  "max_degree" : 20 ,
                  "logfile" : 'NormalFormTest.log' ,
                  "do_stream" : True ,
                  "runprofile" : False ,
                  "compute_diagonalisation" : True ,
                  "run_normal_form_python" : False ,
                  "run_normal_form_cpp" : False ,
                  "system" : "SystemBath" }
```

Any instance of a configuration can be used to override these or add new ones.

tolerance The tolerance used in some calculation and testing not universal

degree the degree to which Lie triangle calculations are performed

max_degree The degree to which the Hamiltonian is truncated for diagonalistion

logfile We use 'NormalFormTest.log'

do_stream Print out the log

runprofile run a profile session

compute_diagonalisation compute the diagonalisation in Python and print for C++

run_normal_form_python run the Normal Form calculations in Python

run_normal_form_cpp run the Normal Form calculations in C++

system pick a problem system from those provided.

2.7 trade offs discussion

Because the calculations are performed in terms of multivariate polynomials of increasing degree the potential number of terms in each polynomial can become very large. Because of the non linear transformations involved often most of the terms can be significant. This manipulation of large polynomials has three effects which present problems:

- The speed of computation and time taken.
- The accuracy required to prevent error accumulation.
- The size of the polynomials themselves and memory requirements.

There is not a single solution to the these since the polynomial representation is a tradeoff between them and each system examined may have a different requirement.

As the problem complexity increases in terms of:

- The degree of freedom and number of variables.
- The degree to which the normal form is calculated.
- The accuracy of the calculations.

the potential number of terms in each polynomial increases as does the memory footprint. Let us look at some examples and approximate numbers. Assuming a few megabytes of memory we start to hit problems at about a thousand terms. Since for a polynomial multiplication we are now dealing with up to a million multiply-add operations which significantly reduces the accuracy of the result.

Internal types

- Polynomials are lists of monomials
- Monomials may be of several forms
- A coefficient which is usually complex but may be real. These both may be multipe precision reals.

- The powers may be a list of integers (tuple) or a dictionary of non zero powers and their indices
- A PolynomialRing is a Polynomial with a fixed number of variables
- A LieAlgebra is

Structure Python C++ compute_normal_form The steps are as follows:-

- 1. confirmation that we have a valid equilibrium point,
- 2. diagonalization of the equilibrium Hamiltonian into real normal form,
- 3. complexification of the real diagonal system into the complex one,
- 4. normalization to complex normal form,
- 5. extraction of the complex integrals,
- 6. realification of the normal form,
- 7. realification of the integrals,
- 8. confirmation that the generating function is invariant under the coordinate changes,
- 9. confirmation that the coordinate changes really do transform the diagonal Hamiltonians into their normalized counterparts and vice versa.

LieAlgebraBase.cpp – base class gets bracket from ClassicalLieAlgebra.cpp SemiClassicalLieAlgebra.cpp DepritTriangle.cpp – Does the Deprit triangle calculations DepritTriangleKnown.cpp – Solves for the ???

calculated normal forms are all stored in

Example software to run the normal forms calculations are stored in projects/nf–unified–0.5/examples/system-bath/src

SystemBathExample.cpp

SystemBathExample.h

SystemBathExample.py

LieTriangle.py does much of the calculation

In order to calculate a normal form you need a formulation of the Hamiltonian and a location of an equilibrium point.

2.7.1 Testing

Testing: unit test software for low level functionality This goes upto and includes runs of Normal Form complete calculations but does not check the results

Also routine to do a run and check the results against previous stored values

Also

Checking:

CheckNormalForm.py Accuracy checks

NormalFormTestAll.py Tests calculated normal forms

2.7.2 Input and Output Files

There are a number of variations of file i/p and o/p formats used in the software. They are based on two different formats, one native to this code, mainly used to print out data from Python to be read by C++ (and vice versa) and an older format compatible with the Mathamatical code. The former uses the phase space coordinate and momentum dimensions in pairs, the xpxp format ordering which is more convenient for higher dimensional systems and semi-classical systems. The latter usually lists all the space coordinates followed by all the momentum coordinates, the xxpp format

The various formats used in the software are as follows.

Note that for conveniance and ease of use polynomials are often written out as a set of homogeneous (or isograde)polynomials with a sparate file holding all terms of a certain degree.

The two file formats are used, the first one produced by e.g. PolynomialRing.seWritePolynomial(): (see PolynomialRing.py) The format is similar to this example:

```
(polynomial
(num-variables 7)
(num-monomials 3)
(powers-format "dense")
(monomials
((1 1 0 0 0 0 0) (-7.349552361081481 1503e-01 +0.0000000000000000000e+00))
...
((0 0 0 0 1 1 0) (+0.0000000000000000000e+00 +1.2672904449679904591e+00))
))
```

with a vector of polynomials (in a .vpol file) in the form:

```
(vector-of-polynomials
(num-polynomials 7)
(polynomials
(polynomial
(num-variables 7)
(num-monomials 4)
(powers-format "dense")
(monomials
((0 1 0 0 0 0 0) (+7.821860663477523e-01 +0.000000000000000e+00))
...
)
)
(polynomial
...
)
)
)
```

The second format (similar to a Mathematica form) in read_ascii_polynomial starts with the number of variables and the number of monomial terms:

```
6
40
0 0 0 0 0 0 -0.98750000000000000000000000000000
2 0 0 0 0 0 -0.30628125000000000000000000000000
...
```

The powers of the variables are followed by a complex or real coefficient. The variables can be grouped by spatial, momenta type, xxxppp, or may be interleaved, xpxpxp, with a quantum mechanical (SemiClassical) hbar term last

There are various files used for the output of the Normal Form software and the names are normally directly related to their contents. These are coordinate transforms between the various coordinate systems. The format used for their contents are polynomials, matrices or vectors of polynomials of appropriate size for the transform. The naming uses the following nemonics for various functions and coordinate systems:

equi equilibrium $e^{\mathbb{R}}$

diag diagonal $d^{\mathbb{R}}$

norm normal form $n^{\mathbb{C}}$

ints integrals I

freq frequency

tham truncated Hamiltonian

tvec truncated vector field

Coordinates:

A coordinate transform such as norm_to_diag.vec is a vector of polinomials each of which is a diagonal coordinate expressed as a function of the normal form coordinates. Thus, when a point in normal form coordinates is substituted into the polinomials, the resultion vector gives the point in diagonal coordinates.

Coordinates transforms for the normal form systems (Mathematica output) are in a set of files named:

equi_to_tham.pol -the Hamiltonian as a polinomial of $e^{\mathbb{R}}$ coords

equi_to_diag.mat -the coordinate change from $e^{\mathbb{R}}$ to $d^{\mathbb{R}}$.

diag_to_equi.mat -the coordinate change from $d^{\mathbb{R}}$ to $e^{\mathbb{R}}$.

diag_to_norm.vec -the coordinate change from $d^{\mathbb{C}}$ to $n^{\mathbb{C}}$.

norm_to_diag.vec -the coordinate change from $n^{\mathbb{C}}$ to$d^{\mathbb{C}}$.

norm_to_ints.vec -the coordinate change from $n^{\mathbb{C}}$ to I

ints_to_tham.pol -the Hamiltonian as a polinomial of I coords

ints_to_freq.vec -the frequencies as a vector of polinomials in I

equi_to_tvec.vec -the vector field as a vector of polinomials in $e^{\mathbb{R}}$

Note that since the truncated vector field, equi_to_tvec.vec, is calculated from J.Hessian of the truncated Hamiltonian it will have degree one less than the Hamiltonian.

Coordinates for nf–unified–0.5 systems start from equi_to_tham.pol

For the Normal Form calculation Python reads: equi_to_tham.pol or obtains the truncated Hamiltonian by some other route e.g. a software generated polynomial.

Python (RunConfig.py) does either or both of:

- calculate the normal form and write these output files and log everything to a log file
- writes diagonalisation and Tayor series for C++ to calculate the normal form and then write the output files.

In the latter case RunConfig.compute_diagonalisation() writes a set of files ending in:

```
--h-from-e--grade-*.pol
--e-from-d.vpol
--r-from-c.vpol
--c-from-r.vpol
```

in the local directory. Here the "*" indicates a set of files with varying numbers for the different polynomials of that degree. Note that here the numbers go from grade-0 to max_degree whereas the results generally start at grade-2. The file names have a prefix added from the configuration.

C++ writes out files of results:

```
--k-from-nc--grade-*.pol
--norm_to_diag.vpol
--diag_to_norm.vpol
--h-from-dc--grade-*.pol
--w--inner-*.pol
--e-from-dc.vpol
```

After the C++ run RunConfig.run_nf reads in the files ending in:

```
--k-from-nc--grade-*.pol
--norm_to_diag.vpol
--diag_to_norm.vpol

and writes out:
diag_to_equi.mat
equi_to_diag.mat
```

```
norm_to_diag.vec
diag_to_norm.vec
norm_to_ints.vec
equi_to_tham.pol
equi_to_tvec.vec
ints_to_freq.vec
ints_to_tham.pol
```

NB the following files contain equivalent information in different formats:

equi_to_tham.pol –h-from-e–grade-*.pol

equi_to_diag.mat

diag_to_equi.mat –e-from-d.vpol

diag_to_norm.vec –diag_to_norm.vpol

norm_to_diag.vec –norm_to_diag.vpol

norm_to_ints.vec

ints_to_tham.pol

ints_to_freq.vec

equi_to_tvec.vec

Python/C++ writes:

```
--e-from-dc.vpol
--e-from-d.vpol
--h-from-dc--grade-*.pol
--h-from-e--grade-*.pol
--k-from-nc--grade-*.pol
--r-from-c.vpol
--w--inner-*.pol
```

```
e-from-dc.vpol
output vector of polynomials for er in terms of dr
e-from-d.vpol   load diagonalisation polynomials
h-from-dc--grade-?.pol
h-from-e--grade-?.pol   Taylor series
k-from-nc--grade-?.pol
r-from-c.vpol
w--inner-?.pol
```

2.8 Normal Forms for Semiclassical Hamiltonians

In (27) it is shown algorithm for computing the normal form of a semiclassical Hamiltonian is "essentially" the same as the algorithm for computing the normal form of a classical Hamiltonian *except* the Poisson bracket is replaced by the Moyal bracket and even powers of $\hbar$ are included in the variables from which homogeneous polynomials are constructed (of course, a substantial amount of theory is required to arrive at this conclusion, to which we refer the users to (27)).

The python code does this in:
import SemiclassicalNormalForm (instead of NormalForm)
The cpp code

For semiclassical Hamiltonian systems the software calculates the normal form and generating function and then stops since it is unclear what the meaning would be of coordinate transforms (which would include the $\hbar$ terms).

Chapter 3

Software Use

This is a description of how to setup and use the software provided here.

Some of the examples given may be run imediately after you have setup the software structure by

- extracting from the tar file or svn as needed.
- move into the directory and:
- type make
- setup environment
- run the test cases
- and then adapt one for your problem

3.1 Install

The software is distributed as a compressed tar file usually called NormalFormSoftware.tgz. You will want to change directory to an installation directory within your filespace which we will now call your_install_dir. You then untar the distributed file using the command:

```
tar xvzf NormalFormSoftware.tgz
cd NormalForm
make
```

You should have a directory NormalForm and if you move into that you will find the software. You should have a makefile, a bin directory containing setup and test scripts, a directory containing code called, for historical reasons, nf–unified–0.5, a file README which should provide useful initial information Running make should perform some setup and run initial tests which may run for some time but can be interupted without harm. The main ssoftware is in nf–unified–0.5/src in three subdirectories: cpp, py, config-run containing C++ code Python code and Python configuration and run utilities respectively. When you run make it will try to write a small file /.NFBASEDIR to your home directory which simply

contains the location of your_install_dir and the extracted software. If you remove your installation and start again then this may ease your path:

```
rm ~/.NFBASEDIR
```

It is envisaged that the running of any new system will take place in a separate project directory and any Python scripts located there needs to be able to find the main software location. The software location can equally be specified by setting an environment variable, NFBASEDIR, to be equal to the software location. This can be achieved from your local startup scripts by sourcing the file bin/setenv.sh or nf–unified–0.5/bin/setenv.sh which should both do the same things.

```
. your_install_dir/NormalForm/bin/setenv.sh
```

If you are on a UOB machine you can use svn to take a checkout from the repository.

3.2 Dependencies

There are a number of software dependencies required before you can run the software. Many will be supplied on a standard Linux install. Some not mentioned here may not be in your Linux install if you are unlucky. The main ones which are likely to be required but are generally easily available are :

Python2.3 or higher for the python half of the code. The shelve module if using the disc for storage (not active by default). You need the unittest module and GUI to run unittests (RECOMMENDED!!!). GNU C++, GNU MP for multi-precision support (active by default in C++). bjam (from BoostC++) or a similar JAM-like build tool for compiling the C++ version, alternatively, you can resort to a make if you are willing to write scripts. The Berkeley DB library if DB support is to be used (this is not active by default). Doxygen for generation of the documentation for the C++ code. Epydoc for generation of the documentation for the Python. libcppunit the C++ unittest framework pyunit The python unittest framework and GUI. gumpy the python interface to GNU GMP [list all others etc]

If you have root access to your machine then these should be installed in their standard location. If you do not then you can insall most of them in your home directory and the software will look for /usr/ and /usr/lib/ and setup environment variables such as LD_LIBRARY_PATH from setenv.sh.

At the moment modules likely to pose difficulties are: libcppunit and gmpy. This is because most of the devellopment and use of this software has been done under SUSE 9.2 or 9.3. The move to SUSE 10.2 proved to have several gotchas. The version of gcc has changed from gcc (GCC) 3.3.4 to gcc (GCC) 4.1.2. The associated library version has changed from libstdc++.so.5 to libstdc++.so.6. Note that version numbers of libs and packages will vary depending upon which updates have been applied. There are also the possibility that the CPU may be 64bit, with or without a 64bit operating system.

In general executables compiled on one system may fail on the other if libraries are mixed. Mainly this will show as a link error but some intermediate versions fail silently. Thus by far the best solution is to get the appropriate packages for your operating systems and load them and then compile.

libcppunit is generally available for SUSE distributions but SUSE does not install it by default Having said all that if your comilation fails to link with -lcppunit this this may make it work:

```
cd ..
cp usr/libcppunit/SuSE_Linux_9.2/libcppunit* usr/lib/
cd NormalForm
make
```

gumpy is not available as an rpm but can be downloaded. For SUSE 10+ and especially for 64 bit machines you will need the new 1.02 version or later from http://code.google.com/p/gmpy/downloads/list. You need to install the rpm python-devel to get the file /usr/lib64/python2.5/config/Makefile. Then do the some version of the following

```
unzip gmpy-1.02-1.zip
cd gmpy-1.02/
python setup.py install
#or if you are not root
python setup.py install --prefix=~/usr
export PYTHONPATH=~/usr/lib64/python2.5/site-packages
#and do the tests to check it works
cd test
python gmpy_test.py
```

3.3 Test

There are several stages to the testing:

- Test the dependencies by making and running the software
- Run the unit tests for Python and c++
- Run the normal form Code on some of the testcases
- Finally test the software itself on a new problem.

make will compile the c++ and should run the test software automatically The files Test.py, for Python, and TestRunner.cpp, for c++, declare the various tests, so you can edit these files to comment-out tests that you don't need, or to add new ones. bin/TestAll.sh is an example script to run the various tests, there's lots of extra instructions in there to point you in the right direction if you want to change it. These unit tests will partly run some test cases themselves. In the directory nf–unified–0.5/src/config-run/ The command:

```
python RunConfig.py
```

will run one of the systems from NfExample.py using a sample configuration and a possible setup for logging etc. It can easily be adapted to run one of the other example systems. You can use RunCrtbp.py as an example of how to run the Crtbp system.

3.4 Run New Systems

These are the requirements to run the Normal Form software on a new system The basic input for a new system is a Hamiltonian as a taylor series expansion about a suitable saddle equilibrium point. For your problem you need to locate the equilibrium point check its form and provide the Hamiltonian, in a local axis set, as a taylor series in the form of a polynomial object. There are a number of examples of this in the file: nf–unified–0.5/src/config-run/NfExample.py. The Hamiltonian polynomial can be read in from a file or explicitly specified term by term. The Hill example shows both the reading of a file and the explicit instantiation of a polynomial (and then checks they are the same). The Hamiltonian polynomial can be specified by a formula as in: nf–unified–0.5/src/py/SystemBath.py and can include standard maths functions where these can be specified as a power series to the required degree. It should be noted that the accuracy to which any constants are specified has implications on the final accuracy of Normal Forms. Once the Hamiltonian polynomial for your system has been generated, any one of the examples can be followed to calculate the Normal Form.

Chapter 4

The Normal Form Theory

In this Chapter we include the theory that was implemented in the code for classical normal forms for completeness sake. In Section 4.1 we describe the essential mathematical elements we require for our use of normal form theory. In Section 4.2 we describe how Lie transforms are computed via the Lie-Deprit method. In Section 4.3 we describe how Lie transforms are used to compute normal forms. In Section 4.4 we discuss the case of main interest to us–the normal form of a Hamiltonian system in the neighborhood of an equilibrium point of saddle-center-...-center stability type. In Section 4.5 we show how this approach can be realized for a specific three-degree-of-freedom Hamiltonian system–the circular restricted three body problem (CRTBP).

Note that in all that follows we will assume that all functions are sufficiently differentiable in order to carry out the required mathematical operations in the domains of interest. Moreover, we assume that the differential equations that we encounter have unique solutions in the domains and time intervals of interest. Clearly, these assumptions will need to be verified in specific applications.

Surveying the history and describing the development of "normal form theory" would be a substantial task that is far beyond the scope of our goals here. Nevertheless, we will highlight a few seminal works and some references that are particularly relevant for out purposes.

Poincaré may be considered a pioneer in developing a method to simplify systems of ordinary differential equations in a neighborhood of an equilibrium point (not necessarily of Hamiltonian nature). Concretely, he found a solution in the case where the matrix corresponding to the linear part of the system (linearized about the equilibrium point) is diagonalizable and its eigenvalues $\gamma_1, \ldots, \gamma_d$ (d being the dimension of the matrix) satisfy a "resonance condition", see (18).

(4) considered the Hamiltonian version of the case treated by Poincaré, giving a method based on action and angle variables to construct normal forms for N degrees of freedom Hamiltonians whose quadratic components have of a special form. Concretely he studied the case in which $\mathcal{H}_0 = \sum_{i=1}^{N} \gamma_i \, x_i \, \xi_i$ and such that the eigenvalues (real or pure imaginary $\pm \gamma_i$ are independent over the integers, i.e. the "non-resonant case". (8) later generalized Birkhoff's results to resonant situations. Other useful references are (6; 9; 12; 7; 13; 14; 17; 15).

There are many references where one can obtain the required basic background on Hamiltonian systems. For example, see (1; 2; 14; 29).

4.1 Overview of the Essential Elements and Features of the Normal Form Method

Here, we will outline the essential elements and features of the normal form method as implemented via Lie transforms.

In a nutshell, the goal of normal form theory is to transform a given Hamiltonian $\mathcal{H} = \mathcal{H}(\mathbf{x})$ into another "simpler" Hamiltonian $\mathcal{K} = \mathcal{K}(\mathbf{y})$ by means of a symplectic change of variables, $\mathbf{x} = \mathcal{X}(\mathbf{y})$. The desired symplectic coordinate changes are constructed via the method of Lie transforms, and what we mean by "simpler" will be defined and play a central role in the nature of the coordinate changes. The "mathematical pieces" that make up this approach are described below.

4.1.1 Notation for the original Hamiltonian system prior to Normalization

We first establish notation for "before" and "after" the transformation to normal form (which is often referred to as "normalization").

For our "original" system (i.e. before "normalization"), we consider an autonomous (that is, time-*independent*) N degrees-of-freedom Hamiltonian $\mathcal{H} = \mathcal{H}(\mathbf{x})$ expressed in coordinates $\mathbf{x} = (x, \xi)$. Here, $x = (x_1, \ldots, x_N)$ gives the configuration space coordinates and $\xi = (\xi_1, \ldots, \xi_N)$ their conjugate momenta. The corresponding Hamiltonian dynamical system is given by

$$\begin{aligned} \dot{x} := \frac{dx}{dt} &= \frac{\partial \mathcal{H}}{\partial \xi}, \\ \dot{\xi} := \frac{d\xi}{dt} &= -\frac{\partial \mathcal{H}}{\partial x}. \end{aligned}$$

4.1.2 The generating function

The change of coordinates will be generated by a map generated by the solutions of a non-autonomous Hamiltonian system. Towards this end, we introduce an auxiliary non-autonomous Hamiltonian $\mathcal{W} = \mathcal{W}(\mathbf{y}; \varepsilon)$ depending on a parameter ε, in the coordinates $\mathbf{y} = (y, \eta) = (y_1, \ldots, y_N, \eta_1, \ldots, \eta_N)$. This additional Hamiltonian is called the "generating function" and its associated Hamiltonian dynamical system is given by

$$\begin{aligned} \frac{dy}{d\varepsilon} &= \frac{\partial \mathcal{W}(\mathbf{y}; \varepsilon)}{\partial \eta}, \\ \frac{d\eta}{d\varepsilon} &= -\frac{\partial \mathcal{W}(\mathbf{y}; \varepsilon)}{\partial y}. \end{aligned}$$

We write the general solution to this dynamical system as

$$\mathcal{X} : (\mathbf{y}; \varepsilon) \mapsto \mathcal{X}(\mathbf{y}; \varepsilon),$$

so that, for example, the solution curve through a particular point $\widehat{\mathbf{y}}$ may be written as

$$\mathcal{X}_{\widehat{\mathbf{y}}} : \varepsilon \mapsto \mathcal{X}(\widehat{\mathbf{y}}; \varepsilon),$$

with initial condition

$$\mathbf{x}(\mathbf{y}; 0) = \mathcal{X}(\mathbf{y}; \varepsilon)|_{\varepsilon=0} = \mathbf{y}.$$

4.1.3 Coordinate changes via the Lie transform

It is a basic result of Hamiltonian systems (see, e.g. (1; 2; 14; 29)) that the solution curves of Hamiltonian vector fields define symplectic maps. The Lie transform uses this general solution $\mathcal{X}(\mathbf{y}; \varepsilon)$ to define a symplectic change of coordinates,

$$\mathbf{x} = \mathcal{X}(\mathbf{y}),$$

with inverse

$$\mathbf{y} = \mathcal{Y}(\mathbf{x}),$$

by means of the time ε map for the flow of the dynamical system defined by $\mathcal{W}$:

$$\mathbf{x} = \mathcal{X}(\mathbf{y}) := \mathcal{X}(\mathbf{y}; \varepsilon) = \mathcal{X}_\varepsilon(\mathbf{y}).$$

Each of the maps $\mathcal{X}_\varepsilon : \mathbf{y} \mapsto \mathcal{X}(\mathbf{y}; \varepsilon)$ is a symplectic map, and these maps form a family parameterised by ε. In what follows we will use the time $\varepsilon = 1$ map in order to make our coordinate changes and will therefore have:

$$\mathbf{x} = \mathcal{X}(\mathbf{y}) := \mathcal{X}(\mathbf{y}; \varepsilon = 1) = \mathcal{X}|_{\varepsilon=1}(\mathbf{y}).$$

Symplectic maps are significant for us since they preserve the form of Hamilton's equations ((1; 2; 14; 29)).

4.1.4 The Lie transform of a general function

In general the Lie transform of any scalar-valued function f (of the vector $(x_1, \ldots, x_N, \xi_1, \ldots, \xi_N)$) with respect to a generating function $\mathcal{W}$ is denoted $\mathcal{L}_\mathcal{W}(f)$ and is defined to be the composition of f with the time ε map of the flow generated by $\mathcal{W}$;

$$\mathcal{L}_\mathcal{W}(f) := f \circ \mathcal{X}_\varepsilon.$$

While we will compute the Lie transformed functions via a formal power series in ε, the actual transformation will be the time $\varepsilon = 1$ map of the flow generated by $\mathcal{W}$. Therefore, we will slightly abuse our notation by also writing:

$$\mathcal{L}_\mathcal{W}(f) := f \circ \mathcal{X}|_{\varepsilon=1}.$$

4.1.5 The transformed Hamiltonian

We may then express the original Hamiltonian $\mathcal{H} = \mathcal{H}(\mathbf{x})$ in the new coordinates $\mathbf{y}$ to give the transformed Hamiltonian $\mathcal{K} = \mathcal{K}(\mathbf{y})$ by means of

$$\mathcal{K}(\mathbf{y}) = \mathcal{L}_{\mathcal{W}}(\mathcal{H})(\mathbf{y}) = (\mathcal{H} \circ \mathcal{X})(\mathbf{y}),$$

or, in more economical notation,

$$\mathcal{K}(\mathbf{y}) = \mathcal{H}(\mathcal{X}(\mathbf{y})),$$

similarly,

$$\mathcal{H}(\mathbf{x}) = \mathcal{K}(\mathcal{Y}(\mathbf{x})).$$

4.1.6 Normal forms: the role of the generating function

Notice that the coordinate transformations, $\mathcal{X}$ and $\mathcal{Y}$, and the transformed Hamiltonian, $\mathcal{K}$, are determined by the generating function, $\mathcal{W}$. So far, we have not put any restrictions on $\mathcal{W}$. We will demonstrate how to construct a generating function in a step-by-step fashion such that the resulting coordinate change $\mathbf{x} = \mathcal{X}(\mathbf{y})$ gives the transformed Hamiltonian $\mathcal{K}$ a simpler ("normal") form than the original one.

We will then be able to use this simplified Hamiltonian to construct geometrical objects in the phase space that are important to the dynamics (in the "normal form coordinates" $\mathbf{y}$) in a straightforward manner. We can then map these objects back into the original coordinates $\mathbf{x}$, via the coordinate change, for application to the original system.

In what follows, we will give a general method for computing the Lie transform $\mathcal{K}$ of a Hamiltonian $\mathcal{H}$ under a generating function $\mathcal{W}$ as outlined above. We will also demonstrate how the coordinate changes $\mathbf{y} = \mathcal{Y}(\mathbf{x})$ and $\mathbf{x} = \mathcal{X}(\mathbf{y})$ are computed explicitly.

We will define precisely what we mean by the phrase "normal form" and will then show how, given an initial Hamiltonian $\mathcal{H}$, we can compute, in a step-by-step manner, both a suitable generating function $\mathcal{W}$ and the corresponding transformed Hamiltonian $\mathcal{K}$ such that $\mathcal{K}$ is in normal form. In order to perform this procedure we make some assumptions on the form of the original Hamiltonian $\mathcal{H}$.

We will then show how this procedure may be applied to Hamiltonian's having equilibrium points of saddle-centre-...-centre type in Section 4.4. This will involve some preliminary steps which are required to put the original Hamiltonian into a suitable form that satisfies the assumptions mentioned above.

The end result will be (1) coordinate transformations, $\mathbf{y} = \mathcal{Y}(\mathbf{x})$ and the inverse $\mathbf{x} = \mathcal{X}(\mathbf{y})$, between the original coordinates $\mathbf{x}$ and "normal form coordinates" $\mathbf{y}$; (2) a transformed Hamiltonian $\mathcal{K}$ which has a simple "normal" form. We will also obtain: (3) expressions for integrals (i.e. constants) of the motion for the normal form Hamiltonian system when the equilibrium point satisfies certain conditions that we will describe.

We will then be in a position to use the normal form Hamiltonian to construct directly various dynamical objects in normal form coordinates. These dynamical objects in normal form coordinates can then be mapped to dynamical objects in the original system via the coordinate change maps.

We now describe some of the important details of the general theory that are used in our software.

4.2 Computation of Lie transforms via the Lie-Deprit Method

In this section we develop the essential features in the Lie-Deprit algorithm for computing Lie transforms.

4.2.1 Preliminaries

First we establish some basic notation and notations that we will use throughout.

Some basic definitions for Hamiltonian systems

Consider an N degrees-of-freedom autonomous (i.e., time-*independent*) Hamiltonian system defined by a Hamiltonian $\mathcal{H} = \mathcal{H}(\mathbf{x})$ expressed in coordinates $\mathbf{x} = (x, \xi)$. We will work with a phase space

$$\mathbb{X} = \mathbb{R}^{2N}, \quad \text{or} \quad \mathbb{X} = \mathbb{C}^{2N},$$

and for any $\mathbf{x} \in \mathbb{X}$, we will write

$$\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_N, \mathbf{x}_{N+1}, \ldots, \mathbf{x}_{2N}),$$

or

$$\mathbf{x} = (x, \xi) = (x_1, \ldots, x_N, \xi_1, \ldots, \xi_N),$$

i.e.,

$$\begin{aligned} \mathbf{x}_j &= x_j \quad \text{for } j = 1, \ldots, N, \\ \mathbf{x}_{N+j} &= \xi_j \quad \text{for } j = 1, \ldots, N. \end{aligned}$$

Here, x gives the configuration space coordinates and ξ gives their associated conjugate momenta. The corresponding Hamiltonian dynamical system is given by

$$\begin{aligned} \dot{x} := \frac{dx}{dt} &= \frac{\partial \mathcal{H}}{\partial \xi}, \\ \dot{\xi} := \frac{d\xi}{dt} &= -\frac{\partial \mathcal{H}}{\partial x}. \end{aligned}$$

Using more economical notation, we may write this as

$$\dot{\mathbf{x}} = J \nabla \mathcal{H}(\mathbf{x}),$$

where J is the matrix,

$$J = \begin{pmatrix} 0_N & I_N \\ -I_N & 0_N \end{pmatrix}. \tag{4.1}$$

In the above, 0_N represents the $N \times N$ zero matrix, and I_N is the $N \times N$ identity matrix. We will write the coordinate changes as

$$\begin{aligned} \mathbf{x} &= \mathcal{X}(\mathbf{y}), \\ \mathbf{y} &= \mathcal{Y}(\mathbf{x}), \end{aligned}$$

with

$$\mathcal{X}(\mathcal{Y}(\mathbf{x})) = \mathbf{x}.$$

where $\mathbf{y} = (\mathbf{y}_1, \ldots, \mathbf{y}_{2N}) = (y_1, \ldots, y_N, \eta_1, \ldots, \eta_N) = (y, \eta)$. We note that a coordinate change $\varphi : \mathbf{y} \mapsto \mathbf{x}$ is symplectic if

$$\left(\frac{\partial \varphi}{\partial \mathbf{y}}\right)^T J \left(\frac{\partial \varphi}{\partial \mathbf{y}}\right) = J,$$

where T denotes the transpose.

Spaces of homogeneous polynomials

For computing the Lie transform of a Hamiltonian about an equilibrium point we will work with Taylor expansions of the Hamiltonian about the equilibrium point. In this setting we will need to consider different terms of the Taylor expansion of the Hamiltonian and the generating function of the Lie transform. It will be useful to establish some notation and terminology that will make this discussion more easy and precise.

To this end, given the phase space $\mathbb{X}$, we will work with polynomials over $\mathbb{X}$ with coefficients taken from the fields $\mathbb{R}$ or $\mathbb{C}$. For each fixed integer degree $d \geq 0$, we consider the homogeneous polynomials in which each term is of degree d, denoted $\mathcal{P}_d$, over the $2N$ variables $\mathbf{x}_1, \ldots, \mathbf{x}_{2N}$. In other words, we consider the set of expressions of the form

$$\sum a_{(m,n)} \prod_{j=1}^{2N} \mathbf{x}_j^{(m,n)_j} = \sum a_{(m,n)} \prod_{j=1}^{N} x_j^{m_j} \xi_j^{n_j},$$

where the sum is taken over all $(m, n) = (m_1, \ldots, m_N, n_1, \ldots, n_N) \in \mathbb{N}^N \times \mathbb{N}^N$ such that

$$\sum_{i=1}^{2N} (m, n)_i = \sum_{i=1}^{N} m_i + \sum_{i=1}^{N} n_i = d, \text{ the degree.}$$

In other words, the sum of the powers to which the variables are raised in each term is always equal to the degree, d. In the above, $a(m, n)$ denotes the coefficient of a single term in the polynomial. We will usually work in the space

$$\mathcal{P} := \bigoplus_{d=2}^{\infty} \mathcal{P}_d,$$

i.e., the space of polynomials in which each term has degree at least 2. We will write, for example,

$$\mathcal{H} = \mathcal{H}_0 + \mathcal{H}_1 + \mathcal{H}_2 + \ldots$$

where

$$\mathcal{H}_d \in \mathcal{P}_{2+d} \quad (d \geq 0).$$

Notice that $\mathcal{P}_{d_1} \mathcal{P}_{d_2} \subseteq \mathcal{P}_{(d_1+d_2)}$.

The Poisson bracket

Writing $\mathbf{x} = (x_1, \ldots, x_N, \xi_1, \ldots, \xi_N)$, and considering functions $F, G : \mathbb{X} \to \mathbb{R}$, we define the Poisson bracket $\{F, G\} : \mathbb{X} \to \mathbb{R}$ by

$$\{F, G\} := \sum_{j=1}^{N} \left(\frac{\partial F}{\partial x_j} \frac{\partial G}{\partial \xi_j} - \frac{\partial F}{\partial \xi_j} \frac{\partial G}{\partial x_j} \right) = \left(\frac{\partial F}{\partial x} \cdot \frac{\partial G}{\partial \xi} \right) - \left(\frac{\partial F}{\partial \xi} \cdot \frac{\partial G}{\partial x} \right),$$

where, in the above, $\cdot$ denotes the usual scalar (dot) product of two vectors.

Consider how the Poisson bracket acts on homogeneous polynomials (i.e., those in which each term has the same fixed degree): Suppose that $F \in \mathcal{P}_{2+r}, G \in \mathcal{P}_{2+s}, (r, s \geq 0)$, then $\{F, G\} \in \mathcal{P}_{2+(r+s)}$, so that the Poisson bracket is well-defined as a map $\{\cdot, \cdot\} : \mathcal{P} \to \mathcal{P}$.

Notice that, in particular, for a quadratic polynomial $F \in \mathcal{P}_2$ and for any $G \in \mathcal{P}_{2+r}$, we have $\{F, G\} \in \mathcal{P}_{2+r}$. In other words, if we define the "Lie operator" $\mathrm{ad}_F := \{F, \cdot\}$ where $F \in \mathcal{P}_2$ then $\mathrm{ad}_F : \mathcal{P}_d \to \mathcal{P}_d$ for all $d \geq 2$, i.e., ad_F maps each of the $\mathcal{P}_d$ into itself. This property will prove to be important later in these notes.

Truncation degree

As we have mentioned, we will work with series representations of the Hamiltonian's and the expressions for the coordinate transformations. Our algorithm will allow is to deduce expressions that relate the coefficients of the various series. In what follows, we will perform all series manipulations formally, that is, without regard to issues of convergence. In general, it is well known that the series associated with the normal form transformations do not converge, except in special cases (20; **?** ; **?** ; **?**). Nevertheless, in practise we may gain useful information by performing our computations up to some finite truncation degree, which we will denote by M.

4.2.2 The Lie transform

The initial Hamiltonian

We begin by considering an initial analytic Hamiltonian function $\mathcal{H}$ depending on a parameter ε:

$$\mathcal{H} = \sum_{i=0}^{\infty} \frac{\varepsilon^i}{i!} \mathcal{H}_i(\mathbf{x}), \tag{4.2}$$

where each Hamiltonian $\mathcal{H}_i(\mathbf{x})$ is an analytic function[1] in the variables $\mathbf{x} = (x, \xi)$. Note that x stands for the configuration space coordinates and ξ denotes their respective conjugate momenta.

A *Lie transform* is a map which converts $\mathcal{H}$ into another Hamiltonian function,

$$\mathcal{K} = \sum_{i=0}^{\infty} \frac{\varepsilon^i}{i!} \mathcal{K}_i(\mathbf{y}), \tag{4.3}$$

[1] In practice it suffices to consider $\mathcal{H}_i \in \mathcal{C}^{(k-i)}(\Omega)$ for some fixed $k \geq i$ and Ω a domain of $\mathbb{R}^6$.

by expressing it in a new system of coordinates, $\mathbf{y} = (y, \eta)$, that are related to the original coordinates via a symplectic map.

The parameter

The parameter ε, in the above series expansions, is introduced for the convenience of this formulation. For now, it suffices to think of ε as a formal "housekeeping" parameter that we will use to group together terms in the series which have the same degree during the various computations. We will perform all series manipulations formally, and will then set $\varepsilon = 1$ afterwards. The parameter ε may be introduced into a general autonomous (time-independent) Hamiltonian H, expressed as a series in quadratic and higher degree terms in coordinates $\mathbf{x}$, by writing

$$\mathcal{H}(\mathbf{x}; \varepsilon) := \varepsilon^{-2} H(\varepsilon \mathbf{x}).$$

We may think of this as a scaling (by the parameter ε) that we apply to the coordinate system.

The generating function

We would like to have a symplectic change of variables for our Lie transform, in order to preserve the Hamiltonian structure of the system. The idea arises to achieve this by using the solution to another (auxiliary) Hamiltonian system, defined by an additional Hamiltonian $\mathcal{W}$ called the *generating function*,

$$\mathcal{W} = \sum_{i=0}^{\infty} \frac{\varepsilon^i}{i!} \mathcal{W}_{i+1}(\mathbf{x}). \tag{4.4}$$

The generating function gives rise to the non-autonomous (it depends on the parameter, ε) Hamiltonian system,

$$\frac{d\mathbf{y}}{d\varepsilon} = J \nabla \mathcal{W}(\mathbf{y}; \varepsilon).$$

We denote the general solution to this system of differential equations by $\mathcal{X}(\mathbf{y}; \varepsilon)$ subject to the initial condition that

$$\mathcal{X}(\mathbf{y}; 0) = \mathbf{y}.$$

Near-identity symplectic changes of variables

Consider any $\mathcal{X}(\mathbf{y}; \varepsilon)$ with

$$\mathcal{X} : \mathbb{X} \times \mathbb{R} \to \mathbb{X}.$$

We say that $\mathcal{X}(\mathbf{y}; \varepsilon)$ *is a "near identity symplectic change of variables"* if the map $\mathcal{X}_\varepsilon : \mathbf{y} \mapsto \mathcal{X}(\mathbf{y}; \varepsilon)$ is symplectic for each fixed ε and if we have

$$\mathcal{X}(\mathbf{y}; \varepsilon) = \mathbf{y} + O(\varepsilon),$$

so that $\mathcal{X}(\mathbf{y}; 0) = \mathbf{y}$.

Recall the dynamical system defined by the generating function $\mathcal{W}$ that we mentioned above, subject to the given initial condition. This system defines just such a near-identity symplectic change of variables, by means of its general solution $\mathcal{X}(\mathbf{y};\varepsilon)$. We denote the inverse by $\mathcal{Y}(\mathbf{x};\varepsilon)$ so that

$$\begin{aligned}\mathcal{X}(\mathcal{Y}(\mathbf{x};\varepsilon);\varepsilon) &= \mathbf{x},\\ \mathcal{Y}(\mathcal{X}(\mathbf{y};\varepsilon);\varepsilon) &= \mathbf{y}.\end{aligned}$$

We may express the original Hamiltonian $\mathcal{H}$ in terms of the new variables $\mathbf{y}$ as $\mathcal{K} = \mathcal{K}(\mathbf{y};\varepsilon)$ by means of

$$\mathcal{K}(\mathbf{y};\varepsilon) \equiv \mathcal{H}(\mathcal{X}(\mathbf{y};\varepsilon);\varepsilon)).$$

The Lie transform

As we have seen above, any parameterised Hamiltonian system, such as that defined by the generating function $\mathcal{W}$ above, defines a family of symplectic transformations by means of the time-evolution map $\mathcal{X} : (\mathbf{y};\varepsilon) \mapsto \mathcal{X}(\mathbf{y};\varepsilon)$ of the system, which takes an initial condition $\mathbf{y}$ and a time period ε and evolves the initial condition under the dynamics for the specified time interval $[0,\varepsilon]$ to give a new point $\mathcal{X}(\mathbf{y};\varepsilon)$. The map $\mathcal{X}$ is the general solution of the Hamiltonian system defined by $\mathcal{W}$.

Each value of the "time" parameter ε gives rise to a symplectic transformation $\mathcal{X}_\varepsilon$ by means of $\mathcal{X}_\varepsilon : \mathbf{y} \mapsto \mathcal{X}(\mathbf{y};\varepsilon)$ and these maps vary continuously with ε.

The method of Lie transforms uses this general solution to make the coordinate change, by means of this time ε map of the flow under $\mathcal{W}$. In other words, a family of coordinate changes is given by the maps $\mathcal{X}_\varepsilon(\mathbf{y}) := \mathcal{X}(\mathbf{y};\varepsilon)$. The Lie transform of a general scalar-valued function $f : \mathbb{X} \to \mathbb{R}$ (or $\mathbb{X} \to \mathbb{C}$) with respect to a generating function $\mathcal{W}$, denoted $\mathcal{L}_\mathcal{W}(f)$, is then simply the composition of f with such a coordinate change;

$$\mathcal{L}_\mathcal{W}(f) := f \circ \mathcal{X}_\varepsilon.$$

We call $\mathcal{L}_\mathcal{W}(f)$ "the Lie Transform of f generated by $\mathcal{W}$". In what follows, we will make use of the time $\varepsilon = 1$ map and thus will have

$$\mathcal{X} := \mathcal{X}(\mathbf{y};1) = \mathcal{X}|_{\varepsilon=1}(\mathbf{y}),$$

and we will use the Lie transform given by

$$\mathcal{L}_\mathcal{W}(f) := f \circ \mathcal{X}|_{\varepsilon=1}.$$

Thus the Lie transform of the original Hamiltonian $\mathcal{H}$ is given by

$$\mathcal{L}_\mathcal{W}(\mathcal{H})(\mathbf{y}) = \mathcal{H}(\mathcal{X}(\mathbf{y})) := \mathcal{H}(\mathcal{X}(\mathbf{y};1)) = \mathcal{K}(\mathbf{y}).$$

4.2.3 The Lie-Deprit method

The Lie-Deprit method provides an algorithm to construct the Lie transform as a series in ε.

Series representations

We represent the functions $\mathcal{H}, \mathcal{W}, \mathcal{K}, \mathcal{X}$ as series expansions in terms of the parameter ε, written as follows:

$$\begin{aligned}
\mathcal{H}(\mathbf{x};\varepsilon) &= \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} \mathcal{H}_n(\mathbf{x}), \\
\mathcal{K}(\mathbf{y};\varepsilon) &= \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} \mathcal{K}_n(\mathbf{y}), \\
\mathcal{W}(\mathbf{y};\varepsilon) &= \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} \mathcal{W}_{n+1}(\mathbf{y}), \\
\mathcal{X}(\mathbf{y};\varepsilon) &= \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} \mathcal{X}_n(\mathbf{y}).
\end{aligned}$$

In the formulation that we will present here, we will take $\mathcal{H}_n, \mathcal{K}_n, \mathcal{W}_n \in \mathcal{P}_{2+n}$ (i.e., terms having degree $2+n$) and $\mathcal{X}_n \in \mathcal{P}_{n+1}$ (i.e., terms having degree $n+1$).

The subscript $n+1$, on the terms of the series for the generating function $\mathcal{W}$, makes our formulation more convenient as we will explain later.

The coordinate change will be a near-identity map which means that $\mathcal{X}_0(\mathbf{y}) = \mathbf{y}$ and thus,

$$\mathcal{X}(\mathbf{y};\varepsilon) = \mathbf{y} + \sum_{n=1}^{\infty} \frac{\varepsilon^n}{n!} \mathcal{X}_n(\mathbf{y}).$$

For this general formulation, we assume that both the original Hamiltonian $\mathcal{H}$ and the generating function $\mathcal{W}$ have both been given and we give a set of recurrence relations that relate all the terms in the above series expansions. In particular, we show how the transformed Hamiltonian $\mathcal{K}$ and the coordinate change $\mathcal{X}$ may be computed in the case where $\mathcal{H}$ and $\mathcal{W}$ have both been given.

We will later show how, for a given Hamiltonian $\mathcal{H}$, a suitable generating function $\mathcal{W}$ may itself be *computed*, degree-by-degree, in order to achieve a special form for the transformed Hamiltonian $\mathcal{K}$.

The intermediate expressions L_j^k

The relations between the coefficients of the various series are expressed in terms of some intermediate quantities that are denoted by L_j^k, with $0 \leq j, k \leq n$ and $j + k = n$. Roughly speaking, these quantities will enable us to compute $\mathcal{K}_n$ in a step-by-step manner from $\mathcal{H}_n$, and from the other quantities already computed, via a chain of relations:

$$\mathcal{H}_n \equiv L_n^0 \to L_{n-1}^1 \to L_{n-2}^2 \to \cdots \to L_1^{n-1} \to L_0^n \equiv \mathcal{K}_n.$$

Note that, at the two ends of this chain, we have

$$\mathcal{H}_n \equiv L_n^0, \quad \text{and} \quad L_0^n \equiv \mathcal{K}_n.$$

Thus, for the case $n = 0$, we have $\mathcal{H}_0 \equiv L_0^0 \equiv \mathcal{K}_0$.

Notice also that, as we proceed along the chain from the left (which is a term of the original Hamiltonian $\mathcal{H}$) to the right (which is a term of the transformed Hamiltonian $\mathcal{K}$), the superscripts on the intermediate quantities L_j^k count upward from 0 to n where as the subscripts count downward from n to 0; the sum of the superscripts and subscripts is always equal to n for each member of the chain.

A derivation that explains with nature of these intermediate quantities is given in Section 4.3.7

The recurrence relations

With notation as before, it may be shown that the following recursion formula relates the terms in $\mathcal{K}$ with those in $\mathcal{H}$ and $\mathcal{W}$ via the intermediate quantities L_j^i:

$$L_j^{(i)} = L_{j+1}^{(i-1)} + \sum_{k=0}^{j} \binom{j}{k} \{L_{j-k}^{(i-1)}, \mathcal{W}_{k+1}\} \tag{4.5}$$

with $j \geq 0$ and $i \geq 1$, where $\binom{j}{k}$ is the usual Binomial coefficient

$$\binom{j}{k} = \frac{j!}{k!(j-k)!}. \tag{4.6}$$

The reason for beginning the indexing of the series for $\mathcal{W}$ with $\mathcal{W}_1$ rather than $\mathcal{W}_0$ now becomes apparent: this gives the equations a nicer form where, in each term (namely, the term L_{j+1}^{i-1} and the term involving $\{L_{j-k}^{i-1}, \mathcal{W}_{k+1}\}$), the superscripts and subscripts always sum to $(i + j)$.

The operator $\{\cdot, \cdot\}$ in the above is the usual Poisson bracket of two scalar fields: given $\mathcal{F}, \mathcal{G} : \mathbb{R}^{2N} \to \mathbb{R}$, the Poisson bracket is defined over an open domain of $\mathbb{R}^{2N}$ as the quantity

$$\{\mathcal{F}, \mathcal{G}\} = \left(\frac{\partial \mathcal{F}}{\partial x} \cdot \frac{\partial \mathcal{G}}{\partial \xi}\right) - \left(\frac{\partial \mathcal{F}}{\partial \xi} \cdot \frac{\partial \mathcal{G}}{\partial x}\right). \tag{4.7}$$

The Lie triangle

$$
\begin{array}{lccccccccc}
(\mathcal{H}_0 \equiv) & L_0^0 & & & & & & & & (\equiv \mathcal{K}_0) \\
& \downarrow & \searrow & & & & & & & \\
(\mathcal{H}_1 \equiv) & L_1^0 & \rightarrow & L_0^1 & & & & & & (\equiv \mathcal{K}_1) \\
& \downarrow & \searrow & \downarrow & \searrow & & & & & \\
(\mathcal{H}_2 \equiv) & L_2^0 & \rightarrow & L_1^1 & \rightarrow & L_0^2 & & & & (\equiv \mathcal{K}_2) \\
& \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & & & \\
(\mathcal{H}_3 \equiv) & L_3^0 & \rightarrow & L_2^1 & \rightarrow & L_1^2 & \rightarrow & L_0^3 & & (\equiv \mathcal{K}_3) \\
& \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots &
\end{array}
$$

Figure 4.1: To compute each term $\mathcal{K}_i$ of the transformed Hamiltonian $\mathcal{K}$, the row of the triangle containing L_j^k with $j + k = i$ is built. One starts at the left with $\mathcal{H}_i \equiv L_i^0$ and ends at the right with $L_0^i \equiv \mathcal{K}_i$. Note that L_0^i cannot be determined unless the term $\mathcal{W}_i$, of the generating function, is already known.

The dependencies between the intermediate quantities L_j^i, and hence the order in which the steps of the recursion process must proceed, are often illustrated by the so-called "Lie-Deprit triangle", shown in figure 4.1. The arrows indicate the direction of computation, thus $a \rightarrow b$ indicates that a must be known (given or computed) before b.

Examples

Here, we will calculate the first few rows of the Lie triangle explicitly, in order to demonstrate how the computation might proceed:

$\mathcal{K}_0$:

$$\mathcal{K}_0 \equiv L_0^0 \equiv \mathcal{H}_0.$$

$\mathcal{K}_1$:

$$
\begin{aligned}
\mathcal{K}_1 \equiv L_0^1 &= L_1^0 + \{L_0^0, \mathcal{W}_1\} \\
&= \mathcal{H}_1 + \{\mathcal{H}_0, \mathcal{W}_1\}.
\end{aligned}
$$

$\mathcal{K}_2$:

$$
\begin{aligned}
\mathcal{K}_2 \equiv L_0^2 &= L_1^1 + \{L_1^0, \mathcal{W}_1\} \\
&= L_2^0 + \{L_1^0, \mathcal{W}_1\} + \{L_0^0, \mathcal{W}_2\} + \{L_1^0, \mathcal{W}_1\} \\
&= \mathcal{H}_2 + \{\mathcal{H}_1, \mathcal{W}_1\} + \{\mathcal{H}_0, \mathcal{W}_2\} + \{\mathcal{H}_1 + \{\mathcal{H}_0, \mathcal{W}_1\}, \mathcal{W}_1\} \\
&= \mathcal{H}_2 + \{\mathcal{H}_1, \mathcal{W}_1\} + \{\mathcal{H}_0, \mathcal{W}_2\} + \{\mathcal{H}_1, \mathcal{W}_1\} + \{\{\mathcal{H}_0, \mathcal{W}_1\}, \mathcal{W}_1\} \\
&= \mathcal{H}_2 + 2\{\mathcal{H}_1, \mathcal{W}_1\} + \{\mathcal{H}_0, \mathcal{W}_2\} + \{\{\mathcal{H}_0, \mathcal{W}_1\}, \mathcal{W}_1\}
\end{aligned}
$$

It is clear that, in the above, one can perform the computations more efficiently by keeping the intermediate results from the previous steps.

4.2.4 Computing the changes of coordinates

In the previous section, we have demonstrated how the transformed Hamiltonian $\mathcal{K}$ may be computed from a given Hamiltonian $\mathcal{H}$ and generating function $\mathcal{W}$. We use the same technique to obtain the coordinate changes that relate the original coordinate system to the transformed one. We now outline the procedure for computing these coordinate changes $\mathcal{X}(\mathbf{y};\varepsilon) = \mathbf{x}$ and $\mathcal{Y}(\mathbf{x};\varepsilon) = \mathbf{y}$.

The Lie transform of the identity

Consider the identity map $\mathrm{Id} : \mathbb{X} \to \mathbb{X}$, $\mathrm{Id}(\mathbf{x}) := \mathbf{x}$. We can adapt the Lie triangle relations in order to transform each component $\mathrm{Id}_j(\mathbf{x}) = \mathbf{x}_j$. This involves modifying our algorithm so that it may be used to transform linear terms, rather than those of quadratic or strictly higher degree.

The coordinate changes that we require are then simply the Lie transforms of the (components of the) identity map under the generating function $\mathcal{W}$ (or $-\mathcal{W}$ for the inverse change). For example, notice that, by definition,

$$\mathcal{L}_{\mathcal{W}}(\mathrm{Id}_s)(\mathbf{y}) := (\mathrm{Id}_s \circ \mathcal{X}|_{\varepsilon=1})(\mathbf{y}) = \mathrm{Id}_s(\mathcal{X}(\mathbf{y})) = \mathcal{X}(\mathbf{y})_s = \mathbf{x}_s.$$

Here, we will demonstrate how to compute the series for the coordinate changes explicitly.

The direct change

The transformation $\mathbf{x} = \mathcal{X}(\mathbf{y};\varepsilon)$ which relates the original or "old" coordinates $\mathbf{x}$ with the normal form ones $\mathbf{y}$ is a near-identity change of variables which we will call the *direct change of coordinates*. We write $\mathcal{X}$ as the series,

$$\mathbf{x} = \mathcal{X}(\mathbf{y};\varepsilon) := \sum_{i=0}^{\infty} \frac{\varepsilon^i}{i!}\, \mathcal{X}_0^{(i)}. \tag{4.8}$$

In a similar manner to the computation of the transformed Hamiltonian, $\mathcal{K}$, we calculate the terms $\mathcal{X}_0^{(i)}$ (for $i \geq 0$) recursively via some intermediate quantities, denoted $\mathcal{X}_k^{(j)}$ with $j + k = i$, using the relation

$$\mathcal{X}_k^{(j)} \;=\; \mathcal{X}_{k+1}^{(j-1)} + \sum_{\ell=0}^{k} \binom{k}{\ell} \{\, \mathcal{X}_\ell^{(j-1)} \,,\, \mathcal{W}_{k+1-\ell} \,\}. \tag{4.9}$$

In the above $\ell \geq 0$, $j \geq 1$, $\mathcal{X}_0^{(0)} := \mathbf{y}$, and $\mathcal{X}_i^{(0)} := \mathbf{0}$ for $i \geq 1$.

The above expression is applied to all components of the vector $\mathbf{y}$. Hence the operator $\{\,\cdot\,,\,\cdot\,\}$ in (4.9) stands for the usual Poisson bracket (with respect to coordinates $\mathbf{y}$) applied to all components of $\mathcal{X}_\ell^{(j-1)}(\mathbf{y})$ and to the generating function expressed in terms of $\mathbf{y}$.

The inverse change

Similar formulae can be used to obtain the coordinate change $\mathbf{y} = \mathcal{Y}(\mathbf{x};\varepsilon)$, which we call the *the inverse change of coordinates*. We express this as the series

$$\mathbf{y} = \mathcal{Y}(\mathbf{x};\varepsilon) = \sum_{i=0}^{\infty} \frac{\varepsilon^i}{i!}\, \mathcal{Y}_i^{(0)}. \tag{4.10}$$

For $i \geq 1$ the terms $\mathcal{Y}_i^{(0)}$ are calculated recursively by means of some intermediate quantities $\mathcal{Y}_k^j$ with $j + k = i$ which satisfy the relation

$$\mathcal{Y}_k^{(j)} \;=\; \mathcal{Y}_{k-1}^{(j+1)} - \sum_{\ell=0}^{k-1} \binom{k-1}{\ell} \{\, \mathcal{Y}_{k-\ell-1}^{(j)}\,,\, W_{\ell+1}(\mathbf{x})\,\}. \tag{4.11}$$

In the above, $i \geq 1$, $j \geq 0$ and this time $\mathcal{Y}_0^{(0)} := \mathbf{x}$, and $\mathcal{Y}_0^{(i)} := \mathbf{0}$ for $i \geq 1$.

The operator $\{\,\cdot\,,\,\cdot\,\}$ in (4.11) refers to the Poisson bracket (with respect to coordinates $\mathbf{x}$) applied to all components of $\mathcal{Y}_{k-\ell-1}^{(j)}(\mathbf{x})$ and to the generating function expressed in terms of $\mathbf{x}$.

Some remarks on the method

At this point we make the following observations:

1. The recursion relations needed to obtain the direct and inverse coordinate changes, given above, can be visualized through Lie-Deprit triangles, similar to that which we gave for the Hamiltonians. We refer the interested reader to (6).

2. There is an alternative method for computing the changes of coordinates which is based on an algorithm to calculate the inverse of a Lie transform. The method is due to (9).

3. The coordinate changes $\mathbf{x} = \mathcal{X}(\mathbf{y};\varepsilon)$ and $\mathbf{y} = \mathcal{Y}(\mathbf{x};\varepsilon)$ are the mappings defining the Lie transform; they are responsible for the transformations of $\mathcal{H}$ into $\mathcal{K}$ and of $\mathcal{K}$ into $\mathcal{H}$.

   Indeed, we may compose the two changes with the Hamiltonians as follows:

   $$\mathcal{H}(\mathbf{x};\varepsilon) = \mathcal{H}(\mathcal{X}(\mathbf{y};\varepsilon);\varepsilon) = \mathcal{K}(\mathbf{y};\varepsilon), \tag{4.12}$$

   and

   $$\mathcal{K}(\mathbf{y};\varepsilon) = \mathcal{K}(\mathcal{Y}(\mathbf{x};\varepsilon);\varepsilon) = \mathcal{H}(\mathbf{x};\varepsilon). \tag{4.13}$$

   Note that these compositions are formal, in the sense that the convergence of the series involved is not considered.

4. Moreover, $\mathcal{X}(\mathbf{y};\varepsilon)$ is formally the inverse mapping of $\mathcal{Y}(\mathbf{x};\varepsilon)$ and vice versa so that, *at least formally*,

   $$\mathcal{X}(\mathcal{Y}(\mathbf{x};\varepsilon);\varepsilon) = \mathbf{x}, \tag{4.14}$$

and

$$\mathcal{Y}(\mathcal{X}(\mathbf{y};\varepsilon);\varepsilon) = \mathbf{y}. \tag{4.15}$$

Equation (4.8) may be used to transform any function expressed in the old variables $\mathbf{x}$ into a function of the new variables $\mathbf{y}$. Similarly, equation (4.10) may be used to express any function of $\mathbf{y}$ as a function of $\mathbf{x}$.

4.3 Using Lie transforms to compute normal forms

So far we have shown how, given an initial Hamiltonian $\mathcal{H}$ and a generating function $\mathcal{W}$, we can compute the Lie transform of $\mathcal{H}$ generated by $\mathcal{W}$.

In other words, we can compute the composition of $\mathcal{H}$ with the time-ε map of the flow generated by $\mathcal{W}$. In particular, we have shown how to compute both the transformed Hamiltonian $\mathcal{K}$ and the family of near-identity coordinate transformations $\mathcal{X}, \mathcal{Y}$ that allow us to change between the original coordinates and the transformed ones.

These explicit changes of variables preserve Hamilton's equations and can be computed, along with the transformed Hamiltonian, by iteration (e.g., via the procedures that we have given, as illustrated by the Lie-Deprit triangle that we have outlined above).

We will now demonstrate how to adapt the above technique for the computation of normal forms, by making a special choice for the generating function $\mathcal{W}$. First, we give a precise definition of the term "normal form" for our purposes.

4.3.1 Definition of "normal form" for our purposes

We consider a Hamiltonian $\mathcal{K}$ written as

$$\mathcal{K} = \mathcal{K}_0 + \mathcal{K}_1 + \mathcal{K}_2 + \cdots \in \mathcal{P},$$

with $\mathcal{K}_i \in \mathcal{P}_{2+i}$ (i.e., $\mathcal{K}_i$ is of degree $d = 2 + i$) for $i \geq 0$,

Definition: Normal form with respect to a given function

We say in general that "$\mathcal{K}$ is in normal form through (terms of) degree $d \geq 2$ *with respect to some given* $F \in \mathcal{P}$" if

$$\{F, \mathcal{K}_r\} = 0, \quad \text{for} \quad r = 0, 1, 2, \ldots, d-2,$$

Definition: Normal form

More specifically, we say simply that "$\mathcal{K}$ is in normal form through degree $d \geq 2$" if $\mathcal{K}$ is in normal form through degree d *with respect to its own quadratic term* $\mathcal{K}_0$, i.e.

$$\{\mathcal{K}_0, \mathcal{K}_r\} = 0 \quad \text{for} \quad r = 0, 1, 2, \ldots, d-2,$$

or, in a different notation that we will use again later in these notes,

$$\mathrm{ad}_{\mathcal{K}_0}(\mathcal{K}_r) = 0 \quad \text{for} \quad r = 0, 1, 2, \ldots, d-2,$$

where $\mathrm{ad}_{\mathcal{K}_0}(F) := \{\mathcal{K}_0, F\}$ for any F. In other words, for $\mathcal{K}$ to be in normal form (through degree d), we require that *all terms of $\mathcal{K}$ (through degree d) commute with the quadratic term $\mathcal{K}_0$ under the Poisson bracket.*

Definition: Normal form of the quadratic terms

In addition to the above definition, we will also insist that the quadratic term $\mathcal{K}_0$ itself has a particular "simple" form, namely that it may be written solely in terms of products $y_j\eta_j$ of configuration space coordinates y_j and their corresponding conjugate momenta η_j. In other words, we insist that $\mathcal{K}_0$ has the form:

$$\mathcal{K}_0(\mathbf{y}) = \sum_{j=1}^{N} \lambda_j (y_j \eta_j),$$

where the coefficients are given by the vector $\lambda = (\lambda_1, \ldots, \lambda_N) \in \mathbb{C}^N$.

We will later put some restrictions on λ that will make our formulation more straightforward, and which will hold in the applications that we are interested in. The definition of "normal form" is in fact more general than this and the assumptions on the form of the quadratic terms of the Hamiltonian may be relaxed. We give on outline of this in section 4.4.3. For now, we will proceed with the most simple case.

4.3.2 The goal of normalization

Suppose that we are given a Hamiltonian

$$\mathcal{H} = \mathcal{H}_0 + \mathcal{H}_1 + \mathcal{H}_2 + \cdots \in \mathcal{P},$$

with $\mathcal{H}_r \in \mathcal{P}_{2+r}$ (so that each $\mathcal{H}_r$ is of degree $2+r$) for $r \geq 0$, in the coordinates $\mathbf{x} = (x, \xi)$. The goal of normalization is to find a symplectic change of coordinates,

$$\mathbf{x} = \mathcal{X}(\mathbf{y}),$$

with inverse

$$\mathbf{y} = \mathcal{Y}(\mathbf{x}),$$

such that expressing $\mathcal{H}$ in terms of $\mathbf{y}$ by means of

$$\mathcal{H}(\mathcal{X}(\mathbf{y})) = \mathcal{K}(\mathbf{y}),$$

with

$$\mathcal{K} = \mathcal{K}_0 + \mathcal{K}_1 + \mathcal{K}_2 + \cdots \in \mathcal{P},$$

($\mathcal{K}_r \in \mathcal{P}_{2+r}$ for $r \geq 0$) *results in a transformed Hamiltonian $\mathcal{K}$ that is in normal form through degree d* (for some $d \geq 2$).

In what follows, we will assume that the quadratic terms $\mathcal{H}_0$ of the original Hamiltonian $\mathcal{H}$ are already in the "simple" form given above and will correspondingly have $\mathcal{K}_0 = \mathcal{H}_0$.

4.3.3 Constructing a generating function for normalization

We have not so far put any restrictions on the generating function $\mathcal{W}$. We will now show how, for a given $\mathcal{H}$, we may construct the terms of $\mathcal{W}$ in order to guarantee that the transformed Hamiltonian $\mathcal{K}$ is in normal form through some given degree d.

The generating function will be constructed degree-by-degree along with the transformed Hamiltonian. This is done by using the Lie triangle relations to rewrite the term $\mathcal{K}_i$ of the transformed Hamiltonian as a sum of known terms plus a Poisson bracket involving the generating function. We will show how the monomials of the known terms can be partitioned easily into those that belong in the normal form $\mathcal{K}_i$ and the remainder. This will leave us with a "homological" equation to solve in order to determine the term $\mathcal{W}_i$ of the generating function and we will indicate how this is solved.

The transformed Hamiltonian

Firstly, recall that the transformed Hamiltonian $\mathcal{K}$ is written as $\mathcal{K} = \mathcal{K}_0 + \mathcal{K}_1 + \dots$ with $\mathcal{K}_i \in \mathcal{P}_{2+i}$ and that we say that $\mathcal{K}$ *is in normal form through degree* d if for $i = 0, \dots, d-2$ we have

$$\{\mathcal{K}_0, \mathcal{K}_i\} = 0, \quad \text{i.e.,} \quad \mathrm{ad}_{\mathcal{K}_0}(\mathcal{K}_i) = 0.$$

Constructing a suitable generating function by induction

We will begin with a given Hamiltonian $\mathcal{H}$ and will for now leave the generating function $\mathcal{W}$ as an unknown; we will construct the generating function $\mathcal{W}$ degree by degree, proceeding by induction.

For the induction step, we will assume that $\mathcal{K}_j$ (and $\mathcal{W}_j$) have already been constructed for $j = 0, \dots, i-1$ in such a way that all the $\mathcal{K}_j$ are in normal form, and we will show how to construct $\mathcal{K}_i$ in normal form and $\mathcal{W}_i$ ready for the next induction step.

As the base case for the induction, we will assume that $\mathcal{H}_0 \equiv \mathcal{K}_0$ is already in normal form. When applying this procedure to a particular Hamiltonian, we must therefore sometimes perform some preliminary steps in order to ensure that the quadratic term $\mathcal{H}_0$ is suitable. We outline these preliminary steps later for three degrees of freedom systems with equilibria of saddle-centre-centre type (Section 4.4), with particular emphasis on the circular restricted three body problems (CRTBP) as a specific example (Section 4.5).

Using the Lie triangle relations for normal forms

Recall the Lie triangle recursion relations given previously. Using these relations, we may write the degree $i-2$ term of the transformed Hamiltonian $\mathcal{K}$ in the following form,

$$\mathcal{K}_i = \mathcal{H}_i + T_i + \{\mathcal{K}_0, \mathcal{W}_i\},$$

where T_i collects together other terms that we already know (or can easily compute) from previous steps in the procedure, and where $\mathcal{W}_i$ is here left as an unknown.

Partitioning the known terms

In order that $\mathcal{K}_i$ should be in normal form, we require that

$$\{\mathcal{K}_0, \mathcal{K}_i\} = 0.$$

In order to achieve this, let us repartition the known terms, namely $\mathcal{H}_i + T_i$, into two pieces:

$$(\mathcal{H}_i + T_i) = N_i + R_i$$

where we wish that N_i contains the parts of $\mathcal{H}_i + T_i$ that are in normal form, and that R_i contains the remaining terms.

Kernel and image of the Lie operator

Since the Poisson bracket $\{\cdot, \cdot\}$ is bilinear, we deduce that the Lie operator $\mathrm{ad}_{\mathcal{K}_0} := \{\mathcal{K}_0, \cdot\}$ is linear. Thus we may consider each monomial F, making up the polynomial $\mathcal{H}_i + T_i$, individually and decide whether it should be added to N_i (the terms in normal form) or to R_i (the remainder).

We require that N_i is the sum of all the monomials F of $\mathcal{H}_i + T_i$ such that $\{\mathcal{K}_0, F\} = 0$. Then R_i is simply the sum of the F such that $\{\mathcal{K}_0, F\} \neq 0$. In other words, $(\mathcal{H}_i + T_i) = N_i + R_i$ with

$$\begin{aligned} \mathrm{ad}_{\mathcal{K}_0}(F) = \{\mathcal{K}_0, F\} = 0 \quad \forall \quad F \in \mathcal{N}_i := \mathrm{monomials}(N_i), \\ \mathrm{ad}_{\mathcal{K}_0}(F) = \{\mathcal{K}_0, F\} \neq 0 \quad \forall \quad F \in \mathcal{R}_i := \mathrm{monomials}(R_i). \end{aligned}$$

Every monomial in $\mathcal{H}_i + T_i$ has the same degree $2 + i$, so that

$$\begin{aligned} \mathcal{N}_i &\subseteq \mathrm{kernel}(\mathrm{ad}_{\mathcal{K}_0}|_{\mathcal{P}_{2+i}}), \\ \mathcal{R}_i &\subseteq \mathrm{image}(\mathrm{ad}_{\mathcal{K}_0}|_{\mathcal{P}_{2+i}}). \end{aligned}$$

Requirements placed on the generating function

In order that $\mathcal{K}_i$ is in normal form, we thus require that

$$\begin{aligned} 0 &= \{\mathcal{K}_0,\ \mathcal{K}_i\} \\ &= \{\mathcal{K}_0,\ \mathcal{H}_i + T_i + \{\mathcal{K}_0, \mathcal{W}_i\}\} \\ &= \{\mathcal{K}_0,\ N_i + R_i + \{\mathcal{K}_0, \mathcal{W}_i\}\} \\ &= \{\mathcal{K}_0,\ R_i + \{\mathcal{K}_0, \mathcal{W}_i\}\} + \{\mathcal{K}_0,\ N_i\} \\ &= \{\mathcal{K}_0,\ R_i + \{\mathcal{K}_0, \mathcal{W}_i\}\} \end{aligned}$$

Finally, in order to satisfy this condition, we will construct the term, $\mathcal{W}_i$, of the generating function in such a way that the following equation holds

$$R_i + \{\mathcal{K}_0, \mathcal{W}_i\} = 0. \tag{4.16}$$

First, we will show how to partition $\mathcal{H}_i + T_i$ into N_i (the parts that are in normal form with respect to $\mathcal{K}_0$) and R_i (the parts that are not in normal form). After which, we will show how the above equation is solved for the unknown $\mathcal{W}_i$.

4.3.4 Partitioning the monomials into resonant and non-resonant terms

The question arises of how to decide whether each monomial F of the known terms $\mathcal{H}_i + T_i$ in the above expressions should go into either N_i or R_i.

Assumptions on the quadratic terms of the original Hamiltonian

Recall that we insisted that $\mathcal{H}_0 = \mathcal{K}_0$ has a simple form in which all coordinates appear only as products of configuration space coordinates and their associated conjugate momenta:

$$\mathcal{H}_0(\mathbf{x}) = \sum_{j=1}^{N} \lambda_j (x_j \xi_j),$$

where $\lambda = (\lambda_1, \ldots, \lambda_N) \in \mathbb{C}^N$.

We now further insist that the λ_j are independent over the integers (also termed "non-resonant"), i.e., for all integer vectors $a = (a_1, \ldots, a_N) \in \mathbb{Z}^N$ we have

$$\langle a, \lambda \rangle = 0 \quad \Rightarrow \quad a = (0, \ldots, 0).$$

In the above, $\langle \cdot, \cdot \rangle$ is the usual inner product (i.e., dot product) of vectors.

As we indicated earlier, we deal with the case where the non-resonance assumption is relaxed in section 4.4.3. For now, we continue with the most simple case.

The general form of a monomial

Now, consider any monomial F of degree $d \geq 2$ (in $\mathcal{H}_i + T_i$):

$$F(\mathbf{x}) = C \prod_{j=1}^{N} x_j^{\mu_j} \xi_j^{\nu_j}.$$

Here, $C = C(\mu, \nu)$ indicates the coefficient of the monomial. The powers on the coordinates are indicated by

$$\begin{aligned} \mu &= (\mu_1, \ldots, \mu_N) \in \mathbb{N}^N, \\ \nu &= (\nu_1, \ldots, \nu_N) \in \mathbb{N}^N, \end{aligned}$$

such that the sum of the powers is equal to the degree, i.e.,

$$\sum_{j=1}^{N} \mu_j + \sum_{j=1}^{N} \nu_j = d.$$

Application of the Lie operator to a monomial

Consider the action of the operator $\mathrm{ad}_{\mathcal{K}_0}$ on the monomial F. A straightforward computation shows that

$$\mathrm{ad}_{\mathcal{K}_0}(F) \equiv \{\mathcal{K}_0, F\} = \langle \mu - \nu, \lambda \rangle F.$$

In other words, each monomial F is an eigenfunction of the operator $\mathrm{ad}_{\mathcal{K}_0}$ with eigenvalue $\langle \mu - \nu, \lambda \rangle$, where μ and ν are the vectors of powers of the coordinates in F. If we would like to have $\{\mathcal{K}_0, F\} = 0$ (so that $F \in \mathcal{N}_i$) then we must therefore have $\langle \mu - \nu, \lambda \rangle F = 0$. Since F is not identically zero, we must therefore have

$$\langle \mu - \nu, \lambda \rangle = 0.$$

Since $(\mu - \nu) \in \mathbb{Z}^N$ we conclude, from our assumption on λ, that $\mu - \nu = 0$, i.e., that $\mu = \nu$ and so F may be written as

$$F(\mathbf{x}) = C \prod_{j=1}^{N} x_j^{\mu_j} \xi_j^{\mu_j} = C \prod_{j=1}^{N} (x_j \xi_j)^{\mu_j}.$$

Resonant and non-resonant terms

The condition $\langle \mu - \nu, \lambda \rangle = 0$ is called "resonance" and such monomials F are sometimes called the "resonant terms" that we must keep and put into $\mathcal{N}_i$, and hence into the term $\mathcal{K}_i$ of the normal form Hamiltonian. Notice that, as a result of the above argument, the condition $\mu = \nu$ on the powers of the monomials in each $\mathcal{K}_i$ means that the normalised Hamiltonian $\mathcal{K}$ may be written as a polynomial in the products $\mathcal{I}_j$ defined by $\mathcal{I}_j(\mathbf{x}) := (x_j \xi_j)$.

The terms with $\langle \mu - \nu, \lambda \rangle \neq 0$ are sometimes called the "non-resonant terms" and are put into the "remainder" $\mathcal{R}_i$.

The reader should note that the terms "resonant" and "non-resonant" are used in more than one context in the literature: in one context, they refer to the commensurability (or otherwise) of the elements of the vector λ, and in another context they refer to the question of whether a (monomial) term belongs in the normal form (in the case where it commutes with the quadratic part under the Poisson bracket) or whether it belongs in the remainder.

4.3.5 Solving the homological equation

The homological equation

In the previous section, we have shown how to partition the monomials from the known terms $\mathcal{H}_i + T_i$ into $N_i = \mathcal{K}_i$ (the normal form) and R_i (the remaining terms). It remains to show how to construct the term $\mathcal{W}_i$ of the generating function in such a way that equation (4.16) is satisfied:

$$R_i + \{\mathcal{K}_0, \mathcal{W}_i\} = 0.$$

In other words, we wish to solve the above equation for the unknown $\mathcal{W}_i$. This partial differential equation is called the *homological equation*.

The monomials occurring in the "remainder"

Recall that R_i is a sum of monomials of the form

$$F(\mathbf{y}) = C_{\mu,\nu} \prod_{j=1}^{N} y_j^{\mu_j} \eta_j^{\nu_j},$$

where $\langle \mu - \nu, \lambda \rangle \neq 0$, so that $\mu \neq \nu$, and where $\sum_j (\mu_j + \nu_j) = 2 + i,$, the degree of the monomial. Recall also that $\mathcal{K}_0$ has the following simple form:

$$\mathcal{K}_0(\mathbf{y}) = \sum_{j=1}^{N} \lambda_j (y_j \eta_j).$$

We require $\{\mathcal{K}_0, \mathcal{W}_i\}$ to be of degree $2 + i$ and $\mathcal{K}_0$ is of degree 2, which implies that $\mathcal{W}_i$ is of degree $d = 2 + i$.

Inverting the Lie operator

We rewrite the homological equation as

$$\mathrm{ad}_{\mathcal{K}_0}(\mathcal{W}_i) = -R_i.$$

Note that if $\mathcal{R}_i$ denotes the monomials of R_i then, by construction, we have

$$\mathcal{R}_i \subseteq \mathrm{image}(\mathrm{ad}_{\mathcal{K}_0}|_{\mathcal{P}_d}),$$

and

$$\mathcal{R}_i \cap \mathrm{kernel}(\mathrm{ad}_{\mathcal{K}_0}|_{\mathcal{P}_d}) = \emptyset.$$

This implies that we can invert the linear operator $\mathrm{ad}_{\mathcal{K}_0}$ on $\mathcal{R}_i$. Recall that, in the notation established above, $\mathrm{ad}_{\mathcal{K}_0}$ acts on monomials F as follows:

$$\mathrm{ad}_{\mathcal{K}_0} : F \mapsto \langle \mu - \nu, \lambda \rangle F.$$

Further, recall that if $F \in \mathrm{image}(\mathrm{ad}_{\mathcal{K}_0}|_{\mathcal{P}_d}) \setminus \mathrm{kernel}(\mathrm{ad}_{\mathcal{K}_0}|_{\mathcal{P}_d})$, then $\langle \mu - \upsilon, \lambda \rangle \neq 0$. Thus we can invert $\mathrm{ad}_{\mathcal{K}_0}|_{\mathcal{P}_d}$ on the monomials that lie in $\mathcal{R}_i$ via

$$\left(\mathrm{ad}_{\mathcal{K}_0}|_{\mathcal{R}_i}\right)^{-1} : F \mapsto \frac{F}{\langle \mu - \nu, \lambda \rangle}.$$

Building the generating function

Thus, we can build the generating function $\mathcal{W}$ degree-by-degree while we are constructing the corresponding transformed Hamiltonian $\mathcal{K}$ in normal form, via

$$\mathcal{W}_i = \left(\mathrm{ad}_{\mathcal{K}_0}|_{\mathcal{P}_{2+i}}\right)^{-1} (-R_i) = \sum_{F \in \mathcal{R}_i} -\frac{F}{\langle \mu - \nu, \lambda \rangle},$$

where $\mathcal{R}_i$ is the set of monomials whose sum is R_i.

4.3.6 Formal integrals of motion

In the previous sections we have demonstrated how to construct a generating function in a degree-by-degree fashion such that the transformed Hamiltonian $\mathcal{K}$ is in normal form. This was done by writing the Lie triangle relations for the term $\mathcal{K}_i$ as a sum of known terms plus a Poisson bracket involving the generating function. We showed how the monomials of the known terms could be partitioned into those which belong in the normal form $\mathcal{K}_i$ and the remainder $\mathcal{R}_i$, which do not. This left us with a "homological" equation to solve in order to determine the term $\mathcal{W}_i$ of the generating function.

We now return to the partitioning of monomials into those that belong in the normal form $\mathcal{K}_i$ and those that do not. We will show that, as a result of our assumptions on the quadratic terms of the Hamiltonian and the resonance condition for a monomial to lie in $\mathcal{K}_i$, we obtain integrals of motion for the normalised Hamiltonian and hence we can calculate, via the coordinate changes, formal integrals for the original Hamiltonian.

Integrals for the normalised Hamiltonian

Recall that the normalised Hamiltonian $\mathcal{K}(\mathbf{y})$ may be written entirely as a polynomial over the *products* $\mathcal{I}_j := (y_j\eta_j)$. It follows that the quantities $\mathcal{I}_j$ are integrals of the motion under the normal form Hamiltonian $\mathcal{K}$.

In order to see this, note that any F with $\{F,\mathcal{K}\} = \{\mathcal{K},F\} = 0$ is an integral of the flow under $\mathcal{K}$, via a basic property of the Poisson bracket;

$$\begin{aligned}
\frac{dF(\mathbf{y})}{dt} &= \sum_{j=1}^{2N} \frac{\partial F(\mathbf{y})}{\partial \mathbf{y}_j}\frac{d\mathbf{y}_j}{dt} \\
&= \sum_{j=1}^{N} \left(\frac{\partial F(\mathbf{y})}{\partial y_j}\frac{dy_j}{dt}\right) + \left(\frac{\partial F(\mathbf{y})}{\partial \eta_j}\frac{d\eta_j}{dt}\right) \\
&= \sum_{j=1}^{N} \left(\frac{\partial F(\mathbf{y})}{\partial y_j}\frac{\partial \mathcal{K}(\mathbf{y})}{\partial \eta_j} - \frac{\partial F(\mathbf{y})}{\partial \eta_j}\frac{\partial \mathcal{K}(\mathbf{y})}{\partial y_j}\right) \\
&= \{F(\mathbf{y}),\mathcal{K}(\mathbf{y})\} = 0.
\end{aligned}$$

In the above, we have used Hamiltonian's equations for $\mathcal{K}(\mathbf{y})$: we have $\dot{\mathbf{y}} = J\nabla\mathcal{K}(\mathbf{y})$.

It now remains to show that if $\mathcal{I}_k = y_k\eta_k$ for $k = 1,\ldots,N$ and $\mathcal{K}$ is a polynomial in the $\mathcal{I}_k$, then $\{\mathcal{I}_k,\mathcal{K}\} = 0$. We have, from bi-linearity of the Poisson bracket,

$$\{\mathcal{I}_k,\mathcal{K}\} = \sum_{F\in\text{monomials}(\mathcal{K})} \{\mathcal{I}_k,F\}.$$

For each monomial F,

$$\{\mathcal{I}_k,F\} = \{y_k\eta_k,F\} = \eta_k\frac{\partial F}{\partial \eta_k} - y_k\frac{\partial F}{\partial y_k},$$

since $\partial\mathcal{I}_k/\partial y_j = \partial\mathcal{I}_k/\partial\eta_j = 0$ whenever $j \neq k$. We write the monomial F as before,

$$F = C\prod_{j=1}^{N}(y_j\eta_j)^{\mu_j},$$

We now show that $\{\mathcal{I}_k, F\} = 0$ for all F, hence $\{\mathcal{I}_k, \mathcal{K}\} = 0$. There are two cases:

1. Case $\mu_k = 0$:

$$\frac{\partial F}{\partial \eta_k} = 0 = \frac{\partial F}{\partial y_k},$$

thus $\{\mathcal{I}_k, F\} = \eta_k \frac{\partial F}{\partial \eta_k} - y_k \frac{\partial F}{\partial y_k} = 0,$

2. Case $\mu_k > 0$:

$$\begin{aligned} \frac{\partial F}{\partial \eta_k} &= \mu_k C \left(\prod_{j \neq k} (y_j \eta_j)^{\mu_j} \right) (y_k^{\mu_k} \eta_k^{\mu_{k-1}}) \\ \frac{\partial F}{\partial y_k} &= \mu_k C \left(\prod_{j \neq k} (y_j \eta_j)^{\mu_j} \right) (y_k^{\mu_{k-1}} \eta_k^{\mu_k}) \end{aligned}$$

Thus

$$\{\mathcal{I}_k, F\} = \eta_k \frac{\partial F}{\partial \eta_k} - y_k \frac{\partial F}{\partial y_k} = \mu_k F - \mu_k F = 0,$$

for each F.

We conclude that $\{\mathcal{I}_k, K\} = 0$ so that each $\mathcal{I}_k$ is an integral of the motion of the normal form Hamiltonian in the normal form coordinates.

Obtaining explicit expressions in the original coordinates

We demonstrated above that the products of configuration space coordinates and their respective conjugate momenta form integrals of the motion in the normal form coordinates.

As a consequence of the above, for each integral $\mathcal{I}(\mathbf{y})$, in the normal form coordinates, it is possible to compute a formal integral $\mathcal{I}^*(\mathbf{x})$ in the original coordinate system, for the original Hamiltonian function $\mathcal{H}$, that is independent of the original Hamiltonian[2]. This is done explicitly in a similar way to the computation of the transformed Hamiltonian and the coordinate changes via the Lie triangle relations.

The formal integral $\mathcal{I}^*(\mathbf{x}; \varepsilon)$ is obtained in terms of the original coordinates $\mathbf{x}$ by writing it as a series,

$$\mathcal{I}^*(\mathbf{x}; \varepsilon) = \mathcal{I}(\mathbf{x}) + \sum_{i=1}^{\infty} \frac{\varepsilon^i}{i!} \mathcal{I}(\mathbf{x})_i^{(0)}, \tag{4.17}$$

where the $\mathcal{I}(\mathbf{x})_i^{(0)}$ are calculated using some intermediate quantities $\mathcal{I}_k^{(j)}$ (with $j + k = i$) via the relation

$$\mathcal{I}(\mathbf{x})_k^{(j)} = \mathcal{I}(\mathbf{x})_{k-1}^{(j+1)} - \sum_{\ell=0}^{k-1} \binom{k-1}{\ell} \{ \mathcal{I}(\mathbf{x})_{k-\ell-1}^{(j)}, \mathcal{W}_{\ell+1}(\mathbf{x}) \}, \tag{4.18}$$

with $k \geq 1$ and $j \geq 0$. Here, we define $\mathcal{I}(\mathbf{x})_0^{(0)} \equiv \mathcal{I}(\mathbf{x})$ and for $i \geq 1$, $\mathcal{I}(\mathbf{x})_0^{(i)} \equiv 0$.

[2]The Poisson bracket between $\mathcal{I}^*$ and $\mathcal{H}$ vanishes and $\mathcal{I}^*$ and $\mathcal{H}$ are functionally independent Hamiltonian functions.

Then $\mathcal{I}^*(\mathbf{x};\varepsilon)$ is a formal integral (which may even be asymptotic in some domain $\Omega' \subseteq \Omega$ and under certain conditions on the size of error) of $\mathcal{H}(\mathbf{x};\varepsilon)$. That is, $\{\,\mathcal{H}\,,\,\mathcal{I}^*\,\} = 0$. For more details of the procedure, see (17).

A brief comment regarding domains of validity

We have given formulae enabling the computation of the coefficients of the series expansions for the transformed Hamiltonian, the coordinate changes, and for the integrals in terms of the original coordinates. We note that, in general, these series do in fact diverge. However, one can gain useful information from these series *truncated to some finite degree*. In any particular application, we determine the domains within which (the truncated series expansions of) the expressions that we obtain are valid up to some specified error tolerance.

4.3.7 Derivation of the Recursion Relations Associated with Lie Transforms

In this section we will show in detail how the recurrence relations in Section 4.2.3 are derived and how the intermediate expressions discussed in Section 4.2.3 arise and what role they play in the recurrence relations.

The Function in the "Original" Coordinates

We will consider scalar valued functions defined on the phase space $\mathbb{R}^{2N}$. For any $\mathbf{x} \in \mathbb{R}^{2N}$, we have

$$\mathbf{x} = (x, \xi) = (x_1, \ldots, x_N, \xi_1, \ldots, \xi_N), \tag{4.19}$$

Thinking in terms of the phase space coordinates of a canonical Hamiltonian vector field, x gives the configuration space coordinates and ξ their conjugate momenta.

Let $\varepsilon \in \mathbb{R}$ denote a (small) scalar parameter and let $f(\varepsilon, \mathbf{x})$ denote a scalar valued function on $\mathbb{R} \times \mathbb{R}^{2N}$. *We will assume that all functions depending on ε can be developed in powers series of ε.* Therefore, we have

$$f(\mathbf{x},\varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} f_n^0(\mathbf{x}), \qquad \text{where} \qquad f_n^0(\mathbf{x}) = \left(\frac{\partial^n}{\partial \varepsilon^n} f(\mathbf{x},\varepsilon) \right)_{\varepsilon=0}. \tag{4.20}$$

The Transformation of Coordinates

In this section we will use a slightly different notation for the coordinate transformations compared to that used in Section 4.2.4. This will make certain of the calculations appear less cumbersome, but we hope that it will also clarify certain variable dependencies which will be important for understanding the meaning of total and partial derivatives.

Consider a function

$$\mathbf{y}(\mathbf{x},\varepsilon), \qquad \mathbf{y}(\mathbf{x},0) = \mathbf{x}, \tag{4.21}$$

and we assume that it has inverse denoted by:

$$\mathbf{x}(\mathbf{y}, \varepsilon,), \qquad \mathbf{x}(\mathbf{y}, 0) = \mathbf{y}, \tag{4.22}$$

where $\mathbf{y} = (\mathbf{y}_1, \ldots, \mathbf{y}_{2N}) = (y_1, \ldots, y_N, \eta_1, \ldots, \eta_N) = (y, \eta)$. We will also assume that these functions can be represented as formal power series in ε:

$$\mathbf{y}(\mathbf{x}, \varepsilon) = \mathbf{x} + \sum_{n=1}^{\infty} \frac{\varepsilon^n}{n!} \mathbf{x}_0^n(\mathbf{x}), \qquad \text{where} \qquad \mathbf{x}_0^n(\mathbf{x}) \equiv \left(\frac{d^n}{d\varepsilon^n} \mathbf{y}(\mathbf{x}, \varepsilon) \right)_{\varepsilon=0} \tag{4.23}$$

and

$$\mathbf{x}(\mathbf{y}, \varepsilon) = \mathbf{y} + \sum_{n=1}^{\infty} \frac{\varepsilon^n}{n!} \mathbf{y}_0^n(\mathbf{y}), \qquad \text{where} \qquad \mathbf{y}_0^n(\mathbf{y}) \equiv \left(\frac{d^n}{d\varepsilon^n} \mathbf{x}(\mathbf{y}, \varepsilon) \right)_{\varepsilon=0} \tag{4.24}$$

The Transformed Function

We will express the function (4.20) as a function of $\mathbf{y}$ by using the transformation (4.22) as follows:

$$f(\mathbf{x}(\mathbf{y}, \varepsilon), \varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} f_0^n(\mathbf{y}) \qquad \text{where} \qquad f_0^n(\mathbf{y}) = \left(\frac{d^n}{d\varepsilon^n} f(\mathbf{x}(\mathbf{y}, \varepsilon), \varepsilon \right)_{\substack{\varepsilon=0 \\ \mathbf{x}=\mathbf{y}}}. \tag{4.25}$$

Observation: *Take particular notice of the partial derivatives with respect to ε in (4.20) and the total derivatives with respect to ε in (4.25).*

Constructing the Coordinate Transformation: The Lie Transform

If we think of $f(\mathbf{x}, \varepsilon)$ as a Hamiltonian function with respect to the coordinates $\mathbf{x}$, then we would like the transformed function of $\mathbf{y}$ to also be a Hamiltonian function with respect to the coordinates $\mathbf{y}$. For this to be the case we will require the transformation to be *canonical*, or *symplectic*. It is well-known that the trajectories generated by a Hamiltonian vector field is a one-parameter family of symplectic transformations. We will define $\mathbf{x}(\mathbf{y}, \varepsilon)$ in this way. This is the method of *Lie transforms*.

Consider a function $W(\mathbf{x}, \varepsilon)$ and the associated Hamiltonian vector field:

$$\begin{aligned} \frac{dx_i}{d\varepsilon} &= \frac{\partial W}{\partial \xi_i}(\mathbf{x}, \varepsilon), \\ \frac{d\xi_i}{d\varepsilon} &= -\frac{\partial W}{\partial x_i}(\mathbf{x}, \varepsilon), \qquad i = 1, \ldots, N. \end{aligned} \tag{4.26}$$

The function $W(\mathbf{x}, \varepsilon)$ is referred to as the *generating function*, and we assume that it can also be expressed as a power series in ε:

$$W(\mathbf{x}, \varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} W_{n+1}(\mathbf{x}), \quad \text{where} \quad W_{n+1}(\mathbf{x}) = \left(\frac{\partial^n}{\partial \varepsilon^n} W(\mathbf{x}, \varepsilon) \right)_{\varepsilon=0}. \tag{4.27}$$

(Note: the reason for choosing the index to be $n+1$ rather than n is (4.27) has to do with a notational convenience that will be explained shortly.)

We denote the trajectories generated by this vector field (4.26) by $\mathbf{x}(\cdot,\varepsilon)$, where the parameter ε is the "time" variable. The phrase "the trajectories generated by a Hamiltonian vector field is a one-parameter family of symplectic transformations" means that for each fixed ε, $\mathbf{x}(\cdot,\varepsilon)$ is a symplectic transformation. The solution of (4.26) with initial condition $\mathbf{y}$ at $\varepsilon = 0$ is denoted by (the "direct" change)

$$\mathbf{x} = \mathbf{x}(\mathbf{y},\varepsilon),\ \mathbf{x}(\mathbf{y},0) = \mathbf{y}. \tag{4.28}$$

We will also require the inverse of this transformation:

$$\mathbf{y} = \mathbf{y}(\mathbf{x},\varepsilon),\ \mathbf{y}(\mathbf{x},0) = \mathbf{y}. \tag{4.29}$$

If we think in terms of the trajectories generated by (4.26), the transformation (4.28) is obtained by taking an initial point $\mathbf{y}$ and letting it evolve for a time ε to a point $\mathbf{x}$. The inverse transformation is obtained by taking that same point $\mathbf{x}$ and letting it evolve for time $-\varepsilon$ to the point $\mathbf{y}$. If we "reverse time" in this way it is equivalent to taking the trajectory generated by the Hamiltonian vector field with $-W(-\mathbf{x},\varepsilon)$. This point will be extremely useful and important later on.

The Lie Derivative

We now ask how functions change "in time" along trajectories of (4.26)? This is obtained by evaluating a function along a trajectory of (4.26) and then computing the derivative with respect to "time" of the resulting function. The derivative is computed by using the chain rule:

$$\frac{d}{d\varepsilon}f(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon) \;=\; \frac{\partial}{\partial\varepsilon}f(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon) + \frac{\partial}{\partial\mathbf{x}}f(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon)\cdot\frac{d}{d\varepsilon}\mathbf{x}(\mathbf{y},\varepsilon), \tag{4.30}$$

where "$\cdot$" represents the usual scalar product. Writing out the scalar product in component form using (4.19) gives:

$$\frac{d}{d\varepsilon}f(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon) \;=\; \frac{\partial}{\partial\varepsilon}f(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon) + \sum_{i=1}^{N}\left(\frac{\partial f}{\partial x_i}(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon)\frac{dx_i}{d\varepsilon}(\mathbf{y},\varepsilon) + \frac{\partial f}{\partial\xi_i}(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon)\frac{d\xi_i}{d\varepsilon}(\mathbf{y},\varepsilon)\right). \tag{4.31}$$

Using (4.26), this takes the form:

$$\frac{d}{d\varepsilon}f(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon) \;=\; \frac{\partial}{\partial\varepsilon}f(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon) + \sum_{i=1}^{N}\left(\frac{\partial f}{\partial x_i}(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon)\frac{\partial W}{\partial\xi_i}(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon) - \frac{\partial f}{\partial\xi_i}(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon)\frac{\partial W}{\partial x_i}(\mathbf{x}(\mathbf{y},\varepsilon),\right.$$

(4

It is possible to simplify the notation further. For any two scalar valued functions $a(\mathbf{x})$, $b(\mathbf{x})$ defined on $\mathbb{R}^{2N}$ the *Poisson Bracket* of $a(\mathbf{x})$ and $b(\mathbf{x})$, denoted $\{a,b\}(\mathbf{x})$ is defined by:

$$\{a,b\}(\mathbf{x}) \equiv \sum_{i=1}^{N} \left(\frac{\partial a}{\partial x_i}\frac{\partial b}{\partial \xi_i} - \frac{\partial a}{\partial \xi_i}\frac{\partial b}{\partial x_i} \right)(\mathbf{x}). \tag{4.33}$$

Using this notation for Poisson Bracket's we have:

$$\frac{d}{d\varepsilon} f(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon) \;=\; \frac{\partial}{\partial \varepsilon} f(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon) + \{f,W\}(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon). \tag{4.34}$$

A Lemma on Poisson Brackets Before going further, we state a lemma that will be useful for later computations.

Consider the formal power series:

$$\sum_{p=0}^{\infty} \frac{\varepsilon^p}{p!} a_p(\mathbf{x}), \quad \text{and} \quad \sum_{q=0}^{\infty} \frac{\varepsilon^q}{q!} b_q(\mathbf{x})$$

where $a_p(\mathbf{x})$ and $b_q(\mathbf{x})$ are functions defined on $\mathbb{R}^{2N}$. Then we have the following lemma.

Lemma 1

$$\left\{ \sum_{p=0}^{\infty} \frac{\varepsilon^p}{p!} a_p \,, \sum_{q=0}^{\infty} \frac{\varepsilon^q}{q!} b_q \right\}(\mathbf{x}) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} \sum_{m=0}^{n} \binom{n}{m} \{a_{n-m}\,, b_m\}(\mathbf{x})$$

This lemma is proved by straightforward calculation.

Recursive Computation of the Transformed Function

Obviously, if we have an explicit transformation $\mathbf{x}(\mathbf{y},\varepsilon)$ then transforming a given function from $\mathbf{x}$ to $\mathbf{y}$ coordinates is simply a matter of substitution. However, our goal will be to choose the transformation in such a way that the new function achieves a "simpler" form. Towards this end, we will develop an iterative approach to generating the coordinate transformation that will allow us the freedom construct a coordinate transformation with the desired properties (and these "desired properties" will be discussed in detail later).

Now we will describe a recursive approach to computing simultaneously the transformation and the transformed function by computing the $f_0^n(\mathbf{y})$ and the $W_{n+1}(\mathbf{x})$ for the generating function.

Here we gather together some earlier established formulae and notation in order to succinctly state the problem and method.

Starting with the function:

$$f(\mathbf{x},\varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} f_n^0(\mathbf{x}), \quad \text{where} \quad f_n^0(\mathbf{x}) = \frac{\partial^n}{\partial \varepsilon^n} f(\mathbf{x},\varepsilon)\Big|_{\varepsilon=0}, \tag{4.35}$$

we wish to "construct" a transformation $\mathbf{x}(\varepsilon,\mathbf{y})$ and express this function in the "new" coordinates $\mathbf{y}$:

$$f(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} f_0^n(\mathbf{y}), \quad \text{where} \quad f_0^n(\mathbf{y}) = \left(\frac{d^n}{d\varepsilon^n} f(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon) \right) \Big|_{\substack{\varepsilon=0 \\ \mathbf{x}=\mathbf{y}}} \tag{4.36}$$

The transformation will be constructed as the "time ε flow map of the flow generated by the Hamiltonian vector field with Hamiltonian $W(\mathbf{x},\varepsilon)$ as given in (4.27).

Towards this end, we assume that $\frac{d^k}{d\varepsilon^k} f(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon)$ can also be expressed as a Taylor series in ε as follows:

$$\frac{d^k}{d\varepsilon^k} f(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} f_n^k(\mathbf{y}), \quad \text{where} \quad f_n^k(\mathbf{y}) = \left(\frac{d^n}{d\varepsilon^n} \left(\frac{d^k}{d\varepsilon^k} f(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon) \right) \right) \Big|_{\substack{\varepsilon=0 \\ \mathbf{x}=\mathbf{y}}} \tag{4.37}$$

Note for clarification. **This is an important step and would appear to be not very well motivated, yet it is crucial for obtaining the recursion relation. It leads to the "intermediate quantities" in the Lie triangle (recall Section 4.2.3).**

In the following calculations we will leave out the arguments for the sake of a less cumbersome notation. Hopefully, with the care we have taken up to this point, they should be clear.

Using (4.37), we have:

$$\sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} f_n^k = \frac{d^k}{d\varepsilon^k} f = \frac{d}{d\varepsilon} \left(\frac{d^{k-1}}{d\varepsilon^{k-1}} f \right) \quad = \quad \frac{d}{d\varepsilon} \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} f_n^{k-1}. \tag{4.38}$$

We use the Lie derivative formula (4.34) to compute the right hand side of this equation to obtain:

$$\frac{d}{d\varepsilon} \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} f_n^{k-1} = \frac{\partial}{\partial\varepsilon} \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} f_n^{k-1} + \left\{ \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} f_n^{k-1}, \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} W_{n+1} \right\}. \tag{4.39}$$

Combining (4.38) and (4.39), and using Lemma 1 gives:

$$\sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} f_n^k = \sum_{n=0}^{\infty} \frac{\varepsilon^{n-1}}{(n-1)!} f_n^{k-1} + \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} \sum_{m=0}^{n} \binom{n}{m} \left\{ f_{n-m}^{k-1}, W_{m+1} \right\} \tag{4.40}$$

The order ε^n term of this series is give by:

$$\boxed{f_n^k = f_{n+1}^{k-1} + \sum_{m=0}^{n} \binom{n}{m} \left\{ f_{n-m}^{k-1}, W_{m+1} \right\}} \tag{4.41}$$

and this provides the necessary recursive rule for computing the f_0^n, which we now describe in some detail[3].

[3]We can now see again why we chose the subscripts for the Taylor coefficients for the generating function to have index $m+1$. If one examines each term on the left and right hand sides of (4.41) (and think of each Poisson bracket as a single term) the sum of all subscripts and superscripts in each term is $n+k$.

Note for clarification. **This is the fundamental recursion relation.** The reader should compare (4.41) with (4.5).

First, for the starting step we have $f_0^0 = f_0^0$. By this apparently obvious statement we mean that the order ε^0 terms in (4.20) and (4.25) are equal.

Degree 2 1st line $\boxed{f_0^1}$

Using (4.41) we have:

$$f_0^1 = f_1^0 + \{f_0^0, W_1\}. \tag{4.42}$$

Now on the right hand side of (4.42), f_0^0 and f_1^0 are known. We have the freedom to choose W_1 is a way that "simplifies" f_0^1. At this point we are describing the recursive construction of the Taylor expansion of the transformed function that is valid for any choice of W_1. We postpone the discussion of the "best way" to choose W_1 until later.

The following triangle is useful to remember how the different terms in the recursion enter in the calculation of f_0^1.

$$\begin{array}{ccc} f_0^0 & & \text{(choosing } W_1\text{)} \\ \downarrow & & \\ f_1^0 & \rightarrow & f_0^1 \end{array} \tag{4.43}$$

At the top of the triangle is f_0^0 obtained at the previous step (so it is known). It is then combined (using (4.42)) with the known f_1^0 and the choice of W_1 to yield the desired f_0^1. We now proceed to calculate f_0^2.

Degree 3, 2nd line $\boxed{f_0^2}$

Using (4.41) gives:

$$f_0^2 = f_1^1 + \{f_0^1, W_1\}. \tag{4.44}$$

Moreover, (4.44) and the quantities that go into that calculation and the fact that, apart from f_1^1, they arose from calculations at previous steps can be understood by considering the yellow highlighted route in the following diagram:

$$\begin{array}{ccccc} f_0^0 & & & & \text{(choosing } W_2\text{)} \\ \downarrow & & & & \\ f_1^0 & \rightarrow & f_0^1 & & \text{(using } W_1\text{)} \\ \downarrow & & \downarrow & & \\ f_2^0 & \rightarrow & f_1^1 & \rightarrow & f_0^2 \end{array} \tag{4.45}$$

Now f_0^1 and W_1 are known from the previous step, but we do not know f_1^1. This can also be calculated using (4.41):

$$f_1^1 = f_2^0 + \{f_1^0, W_1\} + \{f_0^0, W_2\} \tag{4.46}$$

Now f_2^0 is known, f_1^0, W_1, and f_0^0 were known from previous steps, and we choose W_2 at this step. The relation of these quantities from previous steps and how they are used in (4.46)

can be understood diagrammatically by considering the yellow highlighted "route" in the following diagram:

$$\begin{array}{ccccccc}
f_0^0 & & & & & & \text{(choosing } W_2\text{)} \\
\downarrow & & & & & & \\
f_1^0 & \to & f_0^1 & & & & \text{(using } W_1\text{)} \\
\downarrow & & \downarrow & & & & \\
f_2^0 & \to & f_1^1 & \to & f_0^2 & &
\end{array}$$

Degree 2, 3rd line $\boxed{f_0^3}$

Using (4.41) gives:

$$f_0^3 = f_1^2 + \left\{ f_0^2 \, , W_1 \right\} , \tag{4.47}$$

and the relation of these quantities can be understood by the yellow highlighted route in the following diagram:

$$\begin{array}{ccccccc}
f_0^0 & & & & & & \text{(choosing } W_3\text{)} \\
\downarrow & & & & & & \\
f_1^0 & \to & f_0^1 & & & & \text{(using } W_2\text{)} \\
\downarrow & & \downarrow & & & & \\
f_2^0 & \to & f_1^1 & \to & f_0^2 & & \text{(using } W_1\text{)} \\
\downarrow & & \downarrow & & \downarrow & & \\
f_3^0 & \to & f_2^1 & \to & f_1^2 & \to & f_0^3
\end{array} \tag{4.48}$$

Now f_1^2 is unknown, but it can also be computed using (4.41):

$$f_1^2 = f_2^1 + \left\{ f_1^1 \, , W_1 \right\} + \left\{ f_0^1 \, , W_2 \right\} \tag{4.49}$$

and the relation of these quantities can be understood by the yellow highlighted route in the following diagram:

$$\begin{array}{ccccccc}
f_0^0 & & & & & & \text{(choosing } W_3\text{)} \\
\downarrow & & & & & & \\
f_1^0 & \to & f_0^1 & & & & \text{(using } W_2\text{)} \\
\downarrow & & \downarrow & & & & \\
f_2^0 & \to & f_1^1 & \to & f_0^2 & & \text{(using } W_1\text{)} \\
\downarrow & & \downarrow & & \downarrow & & \\
f_3^0 & \to & f_2^1 & \to & f_1^2 & \to & f_0^3
\end{array}$$

Now f_2^1 is unknown, but it can also be computed using (4.41):

$$f_2^1 = f_3^0 + \left\{ f_2^0 \, , W_1 \right\} + 2 \left\{ f_1^0 \, , W_2 \right\} + \left\{ f_0^0 \, , W_3 \right\} \tag{4.50}$$

and the relation of these quantities can be understood by the yellow highlighted route in the following diagram:

The value of W_3 is only determined at the end of these three calculations so it is omitted initially and then chosen at the very end to obtain the Normal Form

$$\begin{array}{ccccccc}
f_0^0 & & & & & & \text{(choosing } W_3\text{)} \\
\downarrow & & & & & & \\
f_1^0 & \to & f_0^1 & & & & \text{(using } W_2\text{)} \\
\downarrow & & \downarrow & & & & \\
f_2^0 & \to & f_1^1 & \to & f_0^2 & & \text{(using } W_1\text{)} \\
\downarrow & & \downarrow & & \downarrow & & \\
f_3^0 & \to & f_2^1 & \to & f_1^2 & \to & f_0^3
\end{array}$$

The reader should compare the expressions derived here with those derived in Section 4.2.3. The diagrams of the form of (4.43), (4.45), and (4.48) that graphically illustrate the recursion relation (4.41) are known as the *Lie triangles*. The reader should compare with Fig. 4.1.

Computation of the Coordinate Transformation and its Inverse

As noted in Section 4.2.4, we can use exactly the same method for computing a series expansion of the transformation (4.28) and its inverse (4.29). (This is why we developed the method in terms of transforming a general (scalar valued) function, rather than in terms of a Hamiltonian function, which tends to obscure the generality). The "trick" will be to cast the computation of the transformation to the form of the very general transformation of a function (4.20) in x coordinates to a function (4.25).

Computation of the "Direct" Change, (4.28)

Consider the "one-parameter family of identity mappings":

$$\mathbf{x} \to \mathrm{id}\,(\mathbf{x}, \varepsilon) = \mathbf{x}. \tag{4.51}$$

and we transform this to the $\mathbf{y}$ coordinates via our coordinate transformation $\mathbf{x} = \mathbf{x}(\mathbf{y}, \varepsilon)$, $\mathbf{x}(\mathbf{y}, 0) = \mathbf{y}$ as follows:

$$\mathbf{y} \to \mathrm{id}\,(\mathbf{x}(\mathbf{y}, \varepsilon), \varepsilon) = \mathbf{x}(\mathbf{y}, \varepsilon). \tag{4.52}$$

This appears identical to the set-up of the transformation of (4.20) to (4.25), and so it would appear that we could proceed directly with the Lie transform/Lie triangle approach for generating a series solution to the transformation. However, there is an important difference (which is easy to account for): the Lie transform approach was developed for transforming a scalar valued function (of a vector variable). The coordinates, and their transformations, are vectors. This is easily dealt with by applying the Lie transform/Lie triangle formalism to the individual components of the transformation.

We let:

$$\mathrm{id_j}(\mathbf{x},\varepsilon) = \mathrm{x_j}, \quad \mathrm{id_j}(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon) = \mathrm{x_j}(\mathbf{y},\varepsilon), \tag{4.53}$$

denote the j^{th} coordinate components of (4.51) and (4.52), and we assume that these functions have the following power series representations:

$$\mathrm{id}_j(\mathbf{x},\varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} \mathrm{id}^0_{n,j}(\mathbf{x}), \quad \text{where} \quad \mathrm{id}^0_{n,j}(\mathbf{x}) = \left(\frac{\partial^n}{\partial \varepsilon^n} \mathrm{id}_j(\mathbf{x},\varepsilon) \right) \Big|_{\varepsilon=0} = \left\{ \begin{array}{cc} x_j & \text{for} \quad n=0, \\ 0 & \text{for} \quad n \geq 1 \end{array} \right. , \tag{4.54}$$

$$\mathrm{id}_j(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} \mathrm{id}^n_{0,j}(\mathbf{y}), \quad \text{where} \quad \mathrm{id}^n_{0,j}(\mathbf{y}) = \left(\frac{d^n}{d\varepsilon^n} \mathrm{id}_j(\mathbf{x}(\mathbf{y},\varepsilon),\varepsilon) \right) \Big|_{\substack{\varepsilon=0 \\ \mathbf{x}=\mathbf{y}}}. \tag{4.55}$$

The general recursion relation (4.41) written in terms of (4.54) and (4.55) is:

$$\mathrm{id}^k_{n,j} = \mathrm{id}^{k-1}_{n+1,j} + \sum_{m=0}^{n} \binom{n}{m} \left\{ \mathrm{id}^{k-1}_{n-m,j} \, , W_{m+1} \right\}, \tag{4.56}$$

which the reader should compare with Eq. (4.9).

Now our goal is to use (4.56) to derive $\mathrm{id}^n_{0,j}(\mathbf{y})$.

$\boxed{\mathrm{id}^0_{0,j}}$

In the beginning we have:

$$\mathrm{id}^0_{0,j} = y_j. \tag{4.57}$$

$\boxed{\mathrm{id}^1_{0,j}}$

Using (4.56) gives:

$$\begin{aligned} \mathrm{id}^1_{0,j} &= \mathrm{id}^0_{1,j} + \left\{ \mathrm{id}^0_{0,j} \, , W_1 \right\}, \\ &= \left\{ y_j \, , W_1 \right\}, \quad \text{using} \quad (4.54) \quad \text{and} \quad (4.57). \end{aligned} \tag{4.58}$$

$\boxed{\mathrm{id}^2_{0,j}}$

Using (4.56) gives:

$$\begin{aligned} \mathrm{id}^2_{0,j} &= \mathrm{id}^1_{1,j} + \left\{ \mathrm{id}^1_{0,j} \, , W_1 \right\} \\ &= \mathrm{id}^1_{1,j} + \left\{ \left\{ y_j \, , W_1 \right\} \, , W_1 \right\} \quad \text{using} \quad (4.58). \end{aligned} \tag{4.59}$$

Using (4.56) again gives:

$$\mathrm{id}^1_{1,j} = \mathrm{id}^0_{2,j} + \left\{ \mathrm{id}^0_{1,j} \, , W_1 \right\} + \left\{ \mathrm{id}^0_{0,j} \, , W_2 \right\} = \left\{ y_j, W_2 \right\}, \quad \text{using} \quad (4.54), \tag{4.60}$$

and therefore we have:

$$\mathrm{id}^2_{0,j} \;=\; \{y_j, W_2\} + \{\{y_j\,, W_1\}\,, W_1\}\,. \tag{4.61}$$

$\boxed{\mathrm{id}^3_{0,j}}$

Using (4.56) gives:

$$\begin{aligned} \mathrm{id}^3_{0,j} &= \mathrm{id}^2_{1,j} + \left\{\mathrm{id}^2_{0,j}\,, W_1\right\}, \\ &= \mathrm{id}^2_{1,j} + \{\{y_j, W_2\}\,, W_1\} + \{\{\{y_j\,, W_1\}\,, W_1\}\,, W_1\}, \quad \text{using} \quad (4.61) \end{aligned} \tag{4.62}$$

Using (4.56) again gives:

$$\begin{aligned} \mathrm{id}^2_{1,j} &= \mathrm{id}^1_{2,j} + \left\{\mathrm{id}^1_{1,j}\,, W_1\right\} + \left\{\mathrm{id}^1_{0,j}\,, W_2\right\}, \\ &= \mathrm{id}^1_{2,j} + \{\{y_j\,, W_2\}\,, W_1\} + \{\{y_j\,, W_1\}\,, W_2\}, \quad \text{using} \quad (4.60) \quad \text{and} \quad (4.58) \end{aligned} \tag{4.63}$$

Using (4.56) again gives:

$$\begin{aligned} \mathrm{id}^1_{2,j} &= \mathrm{id}^0_{3,j} + \left\{\mathrm{id}^0_{2,j}\,, W_1\right\} + 2\left\{\mathrm{id}^0_{1,j}\,, W_2\right\} + \left\{\mathrm{id}^0_{0,j}\,, W_3\right\}, \\ &= \{y_j\,, W_3\}, \quad \text{using} \quad (4.57) \quad \text{and} \quad (4.54). \end{aligned} \tag{4.64}$$

Putting (4.62), (4.63) and (4.64) together gives:

$$\mathrm{id}^3_{0,j} \;=\; \{y_j\,, W_3\} + \{\{y_j\,, W_1\}\,, W_2\} + 2\,\{\{y_j, W_2\}\,, W_1\} + \{\{\{y_j\,, W_1\}\,, W_1\}\,, W_1\} \tag{4.65}$$

Clearly, we can proceed and compute each component of the transformation to $\mathbf{y}$ to a high an order in ε for which we have the resources.

Therefore, for $1 \le j \le 2N$ we have:

$$x_j(\mathbf{y}, \varepsilon) = \mathrm{id}_j(\mathbf{x}(\mathbf{y}, \varepsilon), \varepsilon) \;=\; y_j + \sum_{n=1}^{\infty} \frac{\varepsilon^n}{n!} \mathrm{id}^n_{0,j}(\mathbf{y}) \tag{4.66}$$

which the reader should compare with Eq. (4.8).

Computation of the Inverse Change, (4.29)

We now show how the inverse transformation (4.29) can be easily computed. Proceeding as above, we define the maps:

$$\mathbf{y} \to \mathrm{id}(\mathbf{y}, \varepsilon) = \mathbf{y}. \tag{4.67}$$

and

$$\mathbf{x} \to \mathrm{id}(\mathbf{y}(\mathbf{x},\varepsilon),\varepsilon) = \mathbf{y}(\mathbf{x},\varepsilon). \tag{4.68}$$

As above, we consider coordinates by letting:

$$\mathrm{id_j}(\mathbf{y},\varepsilon) = \mathrm{y_j}, \quad \mathrm{id_j}(\mathbf{y}(\mathbf{x},\varepsilon),\varepsilon) = \mathrm{y_j}(\mathbf{x},\varepsilon), \tag{4.69}$$

denote the j^{th} coordinate components of (4.67) and (4.68), and we assume that these functions have the following power series representations:

$$\mathrm{id}_j(\mathbf{y},\varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} \mathrm{id}^0_{n,j}(\mathbf{y}), \quad \text{where} \quad \mathrm{id}^0_{n,j}(\mathbf{y}) = \left(\frac{\partial^n}{\partial \varepsilon^n} \mathrm{id}_j(\mathbf{y},\varepsilon) \right) \Big|_{\varepsilon=0} = \begin{cases} y_j & \text{for} \quad n = 0, \\ 0 & \text{for} \quad n \geq 1 \end{cases}, \tag{4.70}$$

$$\mathrm{id}_j(\mathbf{y}(\mathbf{x},\varepsilon),\varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} \mathrm{id}^n_{0,j}(\mathbf{x}), \quad \text{where} \quad \mathrm{id}^n_{0,j}(\mathbf{x}) = \left(\frac{d^n}{d\varepsilon^n} \mathrm{id}_j(\mathbf{y}(\mathbf{x},\varepsilon),\varepsilon) \right) \Big|_{\substack{\varepsilon=0 \\ \mathbf{x}=\mathbf{y}}}. \tag{4.71}$$

The general recursion relation (4.41) written in terms of (4.70) and (4.71) is:

$$\mathrm{id}^k_{n,j} = \mathrm{id}^{k-1}_{n+1,j} + \sum_{m=0}^{n} \binom{n}{m} \left\{ \mathrm{id}^{k-1}_{n-m,j}, \overline{W}_{m+1} \right\}, \tag{4.72}$$

where $\overline{W}(\mathbf{y},\varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} \overline{W}_{n+1}(\mathbf{y})$ is the generating function for the j^{th} component of the transformation (4.68). From the remarks immediately following (4.29), this is given by:

$$\overline{W}(\mathbf{y},\varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} \overline{W}_{n+1}(\mathbf{y}) = -W(-\mathbf{y},\varepsilon) = \sum_{n=0}^{\infty} \frac{\varepsilon^n}{n!} (-1)^{n+1} W_{n+1}(\mathbf{y}), \tag{4.73}$$

where $W(\mathbf{x},\varepsilon)$ is the generating function for the j^{th} component of the transformation (4.52).

Summary on the Method and Generating Function

- In computing the transformed function a the Taylor coefficients of the generating function are chosen (the manner in which they are chosen depends on the applications and is discussed later on).
- In computing the coordinate transformation to the "new" variables, the same generating function is used. The Taylor coefficients of the coordinate transformation are computed through a specific algebraic formula.
- The "same" generating function is used to compute the inverse transformation from the "new" to the "original" coordinates. However, negative signs are inserted in front of the even order (in ε) Taylor coefficients for the expansion of the generating function.

4.4 Application to general saddle-centre-centre equilibria for three degrees-of-freedom

In this section we provide additional details on how to calculate the normal form corresponding to a three degrees of freedom Hamiltonian in a neighborhood of an equilibrium point of saddle-center-center stability type. This will require some preliminary transformations in order to bring the original Hamiltonian into a form that is suitable to apply the above procedure. The generalization to additional (centre) degrees-of-freedom is straightforward.

4.4.1 The required form of the Hamiltonian

Recall that, in order to apply the procedure outlined previously in these notes, we assumed that our initial Hamiltonian may be written as a series in quadratic and higher terms,

$$\mathcal{H} = \mathcal{H}_0 + \mathcal{H}_1 + \mathcal{H}_2 + \ldots,$$

with $\mathcal{H}_i \in \mathcal{P}_{2+i}$, such that the quadratic terms have the form

$$\mathcal{H}_0(\mathbf{x}) = \lambda_1(x_1\xi_1) + \lambda_2(x_2\xi_2) + \lambda_3(x_3\xi_3),$$

with the $\lambda_j \in \mathbb{C}$ independent over the integers, i.e., for all $a = (a_1, a_2, a_3) \in \mathbb{Z}^3$:

$$\langle a, \lambda \rangle = 0 \quad \Rightarrow \quad a = (0, 0, 0).$$

where $\lambda = (\lambda_1, \lambda_2, \lambda_3) \in \mathbb{C}^3$.

In Section 4.5 we will explicitly show how to bring a Hamiltonian into this form about a suitable equilibrium point of saddle-centre-centre type, by using the CRTBP as an example.

4.4.2 Outline of the preliminary steps

The following steps are required in order to bring the Hamiltonian into the desired form:

- We begin with the "original Hamiltonian", which we refer to as being expressed in the "physical coordinates".
- The first step will involve a translation to a new coordinate system, which we call the "equilibrium coordinates" in which the equilibrium point of interest lies at the origin. At this stage it is also convenient to remove any constant term from the Hamiltonian (but for applications it may be very important to include it later on).
- The second step is to expand the Hamiltonian in the new coordinates as a Taylor series about the origin, up to some desired truncation degree M.
- Thirdly, we make a linear change of variables to bring the matrix associated with the quadratic part of the Hamiltonian (i.e., with the linearised form of Hamilton's equations) into a "real diagonal" form. We call the resulting coordinate system the "real diagonal coordinates".

- The final preliminary step is to make a change to complex coordinates, which we call the "complex diagonal coordinates", in which the quadratic part of the Hamiltonian has the desired form (that is, it may be written as a linear combination of products of configuration space coordinates with their respective momenta).

After these steps, we will be ready to apply the normalization procedure described in detail earlier. Since various coordinate changes have already taken place at this point, we must be careful to use the relevant inverse transformations before we can apply any results that we compute using the normalized Hamiltonian to the original Hamiltonian back in the "physical" coordinates.

4.4.3 The resonant case for general saddle-centre-centres

In the preceeding sections, we have shown that, whenever the matrix associated to the quadratic part of the Hamiltonian $\mathcal{H}$ is semisimple (i.e., diagonalizable), the calculation of the normal form can lead both to a reduction in the number of degrees of freedom and also to the construction of formal integrals.

In the above, we have assumed that the coefficients λ of the quadratic part of the original Hamiltonian $\mathcal{H}$ are independent over the integers. That is, they satisfy a non-resonance condition. In that case, the normal form that we calculate defines a system of zero-degrees of freedom and consequently it leads to the construction of three integrals of the normalized system, as illustrated previously.

The question arises as to what happens when this is not the case. That is, what happens when there is a resonance between the eigenvalues? We deal with this question briefly here.

The quadratic part of the Hamiltonian

Recall that, for an equilibrium point of saddle-centre-centre type, the quadratic part of the complex diagonal Hamiltonian $\mathcal{H} \equiv \mathbf{D}^{\mathbf{c}}$ has the following form

$$\mathcal{H}_0(\mathbf{x}) = \lambda_1 (x_1\xi_1) + \lambda_2 (x_2\xi_2) + \lambda_3 (x_3\xi_3) ,$$

where the coefficients are given by

$$\lambda = (\lambda_1, \lambda_2, \lambda_3) = (\omega_1, \imath\omega_2, \imath\omega_3),$$

with $\omega_1, \omega_2, \omega_3 \in \mathbb{R}$. Notice that we need therefore only consider the possibility of resonance between the quantities ω_2 and ω_3; since there cannot be a resonance between a purely real and a purely imaginary value, we do not need to consider ω_1.

Partitioning the monomials revisited

Recall that, in order to compute the normal form, we performed a partitioning of the monomials appearing the expression for $\mathcal{K}$ (at each degree) into two sets: those that commute with the quadratic part and therefore belong in the normal form, and those that do not and therefore belong in the Homological equation that we used to find the generating function.

The above mentioned partitioning procedure relied on the fact that the eigenvalues were non-resonant. Therefore, we must now re-examine that procedure in the case where a resonance might exist.

The general form of a monomial

In what follows we consider any monomial F (in the Hamiltonian $\mathcal{H} \equiv \mathbf{D}^{\mathsf{c}}$, prior to normalization) of degree $d \geq 2$:

$$F(x,\xi) = C(\mu,\nu)\,(x_1^{\mu_1}\xi_1^{\nu_1})\,(x_2^{\mu_2}\xi_2^{\nu_2})\,(x_3^{\mu_3}\xi_3^{\nu_3})$$

where $C = C(\mu,\nu)$ indicates the coefficient, and the powers on the coordinates (x,ξ) are indicated by $\mu = (\mu_1,\mu_2,\mu_3) \in \mathbb{N}^3$, and $\nu = (\nu_1,\nu_2,\nu_3) \in \mathbb{N}^3$, such that the sum of the powers is equal to d, the degree of the monomial.

The resonant case

Whenever ω_2 and ω_3 satisfy a resonant condition, that is, whenever the quotient ω_2/ω_3 belongs to $\mathbb{Q}$ (rational), then a monomial F of the form given above commutes with $\mathcal{H}_0 \equiv \mathcal{K}_0$ if and only if

$$\omega_2\,(\nu_2 - \mu_2) = \omega_3\,(\mu_3 - \nu_3) \qquad \text{and} \qquad \mu_1 = \nu_1. \tag{4.74}$$

Note that this condition is weaker than the earlier condition that $\mu = \nu$. Thus, in this case, the number of terms that will be part of the normalized Hamiltonian $\mathcal{K} \equiv \mathbf{N}^{\mathsf{c}}{}_i$ is larger than in the non-resonant case. Furthermore, the number of formal integrals obtained through the normal form process is reduced to two. (One still obtains the "saddle integral" corresponding to $\mathcal{I}_1(\mathbf{y}) = y_1\eta_1$.)

The near-resonant case; small divisors

If ω_2/ω_3 is *close* to a rational number, it can occur that one has powers μ and ν such that $\omega_2\,(\nu_2 - \mu_2) + \omega_3\,(\nu_3 - \mu_3)$ is close to zero.

Whether this situation occurs in practice depends on the degree to which the computations are being taken and, therefore, on the range of values that can occur for the powers μ, ν.

Recall that when a monomial F is chosen to contribute to the generating function, rather than to the normalized Hamiltonian, then the contribution is given by

$$\frac{F}{\langle \mu - \nu, \lambda \rangle} = \frac{F}{\lambda_1\,(\mu_1 - \nu_1) + \imath\,[\omega_2\,(\mu_2 - \nu_2) + \omega_3\,(\mu_3 - \nu_3)]}.$$

Consider the case of a near-resonance between ω_2 and ω_3. If we have a monomial with $\mu_1 = \nu_1$ then the denominator in the above term of the generating function would be $\imath\,[\omega_2\,(\nu_2 - \mu_2) + \omega_3\,(\nu_3 - \mu_3)]$.

In the case where the computation is carried to high degree, the powers μ and ν may take on a larger range of values, making the occurrence of a small denominator in the generating function more likely. This would lead to the appearance of small denominators in the terms of the Hamiltonian $\mathcal{K}$ of larger degree.

It may therefore be advisable to instead leave the corresponding monomial F in the normal form $\mathcal{K} \equiv \mathbf{N}^{\mathsf{c}}$, in order to avoid this problem.

The loss of formal integrals

In the above cases, if one chooses to include the extra terms in the normalized Hamiltonian, then one does so at the expense of losing the formal integrals corresponding to the pairs (y_2, η_2) and (y_3, η_3).

4.5 Example: the collinear equilibria of the circular restricted three body problem

In this Section we present an example, the circular restricted three body problem (CRTBP), that will illustrate the steps required for normalizing a Hamiltonian system in the neighborhood of an equilibrium point of saddle-centre-centre stability type.

4.5.1 The original CRTBP Hamiltonian

In this section we define the Hamiltonian system for the Circularly Restricted Three Body Problem (CRTBP) and describe the relevant equilibria and their linearized stability. Background information can be found in, for example, (21; 16; 11; 10).

The Hamiltonian

The coordinates (in a rotating reference frame) are denoted by

$$(\boldsymbol{x}, \boldsymbol{\xi}) = (x_1, x_2, x_3, \xi_1, \xi_2, \xi_3). \tag{4.75}$$

We will refer to this coordinate system as the "physical coordinates". The Hamiltonian in these coordinates is denoted

$$\mathbf{H} = \mathbf{H}(\boldsymbol{x}, \boldsymbol{\xi}). \tag{4.76}$$

The Hamiltonian function for the spatial circular restricted three-body problem is given by:

$$\mathbf{H} = \frac{1}{2}\left(\xi_1^2 + \xi_2^2 + \xi_3^2\right) - \left(-x_2\,\xi_1 + x_1\,\xi_2\right) - \frac{1-\mu}{d_1} - \frac{\mu}{d_2} - \frac{\mu\,(1-\mu)}{2}. \tag{4.77}$$

where

$$d_1 = \sqrt{(\mu + x_1)^2 + x_2^2 + x_3^2}, \tag{4.78}$$

and

$$d_2 = \sqrt{(\mu - 1 + x_1)^2 + x_2^2 + x_3^2}, \tag{4.79}$$

Note that this Hamiltonian depends upon a parameter $\mu \in (0, 1/2)$. We will assume that the value of this parameter has been fixed for a particular application and will therefore proceed assuming that μ is a suitable constant.

The collinear equilibria

This Hamiltonian has three equilibria of suitable stability type. These are conventionally denoted L_1, L_2, and L_3. In the coordinate system given above, they lie at positions:

$$(\bar{x}_1, \bar{x}_2, 0, -\bar{x}_2, \bar{x}_1, 0), \tag{4.80}$$

which are solutions the following equations:

$$F(x_1;\mu) = \begin{cases} x_1 + \frac{\mu}{d_2^2} + \frac{1-\mu}{d_1^2} & \text{for} \qquad x_1 < -\mu, \\ x_1 + \frac{\mu}{d_2^2} - \frac{1-\mu}{d_1^2} & \text{for} \qquad -\mu < x_1 < 1-\mu, \\ x_1 - \frac{\mu}{d_2^2} - \frac{1-\mu}{d_1^2} & \text{for} \qquad x_1 > 1-\mu. \end{cases} \tag{4.81}$$

It may be shown that $F(x_1;\mu)$ has a unique root for $x_1 < -\mu$, which corresponds to the equilibrium point known as L_3.

A similar argument can be made in the interval $-\mu < x_1 < 1-\mu$, from which we conclude the existence of the equilibrium point known as L_1.

Finally, in the interval $x_1 > 1-\mu$, we conclude the existence of the equilibrium point known as L_2.

These equilibria are the solutions of a quintic equation given by $F(x_1;\mu) = 0$. (The remaining two solutions correspond to equilibrium points known as L_4 and L_5 which are not of the correct stability type for our example application of normal form transformation.)

Linearised stability of the collinear equilibria

Hence, we begin by translating the collinear equilibrium point of interest to the origin. If $(\bar{x}_1, 0, 0, 0, \bar{x}_1, 0)$ are the coordinates of one of the collinear equilibria, we define the new set of variables $(x'_1, x'_2, x'_3, \xi'_1, \xi'_2, \xi'_3)$:

$$\begin{array}{lll} x'_1 = x_1 - \bar{x}_1, & x'_2 = x_2, & x'_3 = x_3, \\ \xi'_1 = \xi_1, & \xi'_2 = \xi_2 - \bar{x}_1, & \xi'_3 = \xi_3. \end{array} \tag{4.82}$$

We substitute this change of variables into the Hamiltonian (4.77) and then expand in Taylor series around the origin up to order two:

$$\mathbf{E} = \frac{1}{2}\left(\xi'^2_1 + \xi'^2_2 + \xi'^2_3\right) - \left(-x'_2\,\xi'_1 + x'_1\,\xi'_2\right) + \frac{\alpha}{2}\left(-2\,x'^2_1 + x'^2_2 + x'^2_3\right), \tag{4.83}$$

where

$$\alpha = \frac{\mu}{d_2^3} + \frac{1-\mu}{d_1^3}, \tag{4.84}$$

and $d_1 = |\mu + \bar{x_1}|$, $d_2 = |\mu - 1 + \bar{x_1}|$. Constant terms have been dropped from the expression of $\mathbf{E}$ since they are irrelevant for the computation of the stability. Since the Taylor expansion has been carried out around an equilibrium point, linear terms do not appear in $\mathbf{E}$. Finally, we explicitly point out the (hopefully obvious) fact that the value of $\bar{x_1}$, and hence d_1, d_2, and α, depends upon whether we are considering L_1, L_2, or L_3. For the sake of maintaining

a more readable notation, we will not explicitly denote this fact as it should be clear from the context.

Now we calculate the Hamiltonian matrix A associated to $\mathbf{E}$: $A = J\,B$, where J is defined in (4.1) and B is the Hessian matrix associated with $\mathbf{E}$:

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 2\alpha & 0 & 0 & 0 & 1 & 0 \\ 0 & -\alpha & 0 & -1 & 0 & 0 \\ 0 & 0 & -\alpha & 0 & 0 & 0 \end{pmatrix}. \tag{4.85}$$

The eigenvalues λ of this matrix determine the linear stability of the equilibria. The characteristic equation associated to A is:

$$(\lambda^2 + \alpha)\,\left(\lambda^4 - (\alpha - 2)\,\lambda^2 - 2\alpha^2 + \alpha + 1\right) = 0, \tag{4.86}$$

and the six eigenvalues are given by:

$$\lambda^2 = -\alpha, \qquad \lambda^2 = \frac{\alpha - 2 \pm \sqrt{\alpha\,(9\alpha - 8)}}{2}. \tag{4.87}$$

From its definition, α is positive, so that the expression on the left corresponds to a pair of complex conjugate purely imaginary eigenvalues. In fact, we will argue that $\alpha > 1$ for the three collinear equilibria and, hence, from the expression of the eigenvalues on the right, we can then infer that there is another pair of purely imaginary eigenvalues and a pair of real eigenvalues.

We provide some details that for L_3 the parameter α is greater than one. For L_1 and L_2 the reasoning is similar, and therefore we will not include it here. Recall that $\bar{x}_1$ is the solution of $F(x_1; \mu) = 0$. We consider the expression for F and the values of d_1 and d_2 corresponding to L_3:

$$\bar{x}_1 = -\frac{\mu}{d_2^2} - \frac{1-\mu}{d_1^2}, \qquad \bar{x}_1 = -\mu - d_1. \tag{4.88}$$

By equating both expressions we obtain:

$$\frac{1-\mu}{d_1^2} = -\frac{\mu}{d_2^2} + \mu + d_1. \tag{4.89}$$

Now we substitute (4.89) into the expression for α, and use the fact that for L_3 we have $d_2 = 1 + d_1$, $\mu, d_1 > 0$, and $d_2 > 1$ to obtain:

$$\alpha = 1 + \mu\,\left(\frac{d_1 - d_2 + d_2^3}{d_1\,d_2^3}\right) = 1 + \mu\,\left(\frac{-1 + d_2^3}{d_1\,d_2^3}\right) > 1. \tag{4.90}$$

Consequently, we can conclude that the collinear equilibrium point L_3 has a pair of real eigenvalues (equal in magnitude, but with opposite signs) and two pairs of complex conjugate, purely imaginary eigenvalues, for all values of μ. Such equilibria are said to have *saddle-center-center* stability type, and each is often referred to just as a "saddle-center-center." The same conclusions hold for L_1 and L_2.

4.5.2 Coordinates centered on the equilibrium point

The reader will note, that the CRTBP Hamiltonian is not in a suitable form to apply our normalization procedure.

As already indicated we first need to translate the equilibrium point of interest (either L_1, L_2 or L_3) to the origin and express the Hamiltonian $\mathbf{H}$ in the corresponding coordinates. We will also take this opportunity to remove any constant terms from the Hamiltonian. In other words, we make the following transformation

$$\mathbf{H}(\boldsymbol{x},\boldsymbol{\xi}) \mapsto \mathbf{H}\left[(\boldsymbol{x},\boldsymbol{\xi}) - (\boldsymbol{x},\boldsymbol{\xi})^*\right] - \mathbf{H}((\boldsymbol{x},\boldsymbol{\xi})^*),$$

where

$$(\boldsymbol{x},\boldsymbol{\xi})^* = (x_1^*, 0, 0, 0, x_1^*, 0), \tag{4.91}$$

denotes the phase space coordinates of the relevant equilibrium point (L_1, L_2, or L_3) under consideration. We designate the new coordinates as

$$(\boldsymbol{x'},\boldsymbol{\xi'}) = (x'_1, x'_2, x'_3, \xi'_1, \xi'_2, \xi'_3), \tag{4.92}$$

which we refer to as the "equilibrium coordinates". The change of coordinates will be denoted by

$$\mathcal{E} : (\boldsymbol{x},\boldsymbol{\xi}) \mapsto (\boldsymbol{x'},\boldsymbol{\xi'}) = (\boldsymbol{x},\boldsymbol{\xi}) - (\boldsymbol{x},\boldsymbol{\xi})^*. \tag{4.93}$$

We denote the inverse of this coordinate change by

$$\mathcal{E}^{-1} : (\boldsymbol{x'},\boldsymbol{\xi'}) \mapsto (\boldsymbol{x},\boldsymbol{\xi}). \tag{4.94}$$

Computation of Taylor expansions for the CRTBP

We apply the change of coordinates (4.93) to the Hamiltonian (4.77) and then expand in a Taylor series around the origin.

A convenient formula for achieving this in the case of the CRTBP is the following well-known series expression:

$$\frac{1}{\sqrt{(a-A)^2+(b-B)^2+(c-C)^2}} = \frac{1}{D}\sum_{n=0}^{\infty}\left(\frac{r}{D}\right)^n P_n\left(\frac{a\,A+b\,B+c\,C}{D\,r}\right), \tag{4.95}$$

where $D^2 = A^2+B^2+C^2$, $r^2 = a^2+b^2+c^2$ and $P_n(x)$ refers to the Legendre polynomial in x of degree n.

In order to make the expansion, Formula (4.95) is applied to d_1 and d_2 of $\mathbf{H}$ in the coordinates $(\boldsymbol{x'},\boldsymbol{\xi'})$. In particular we have

$$\begin{aligned} d_1 &= \sqrt{(x'_1+\mu+x_1^j)^2 + x'^2_2 + x'^2_3}, \\ d_2 &= \sqrt{(x'_1+\mu-1+x_1^j)^2 + x'^2_2 + x'^2_3}, \end{aligned} \tag{4.96}$$

and using (4.95) we get

$$\frac{1}{d_1} = \frac{1}{|\mu+x_1^j|}\sum_{n=0}^{\infty}(-1)^{n+1}\left(\frac{r'}{|\mu+x_1^j|}\right)^n P_n\left(\frac{x'_1}{r'}\right), \tag{4.97}$$

and

$$\frac{1}{d_2} = \frac{1}{|\mu - 1 + x_1^j|} \sum_{n=0}^{\infty} (-1)^{n+1} \left(\frac{r'}{|\mu - 1 + x_1^j|} \right)^n P_n \left(\frac{x'_1}{r'} \right), \tag{4.98}$$

where $r' = \sqrt{x'^2_1 + x'^2_2 + x'^2_3}$.

Next we transform Hamiltonian $\mathbf{H}$ using (4.94), (4.97) and (4.98), and arrive at an expression for the Hamiltonian in the coordinates centered on L_1, L_2 or L_3. We denote the new Hamiltonian by $\widehat{\mathbf{E}}$:

$$\begin{aligned} \widehat{\mathbf{E}} &= \frac{1}{2}(\xi'^2_1 + \xi'^2_2 + \xi'^2_3) - (-x'_2\,\xi'_1 + x'_1\,\xi'_2) + \frac{\alpha}{2}(-2\,x'^2_1 + x'^2_2 + x'^2_3) \\ &\quad + \sum_{n=3}^{\infty} \left(\frac{(-1)^{n+1}}{|\mu + x_1^j|^{n+1}} + \frac{(-1)^{n+1}}{|\mu - 1 + x_1^j|^{n+1}} \right) r'^n\, P_n \left(\frac{x'_1}{r'} \right), \end{aligned} \tag{4.99}$$

where the μ–parameter α is given by

$$\alpha = \frac{1}{|\mu + x_1^j|^3} + \frac{1}{|\mu - 1 + x_1^j|^3}, \tag{4.100}$$

and constant terms have been removed from $\widehat{\mathbf{E}}$. We emphasize that for each $n \geq 3$, $r'^n\, P_n(x'_1/r')$ is a homogeneous polynomial in x'_1, x'_2 and x'_3 of degree n.

Moreover, if we denote $Q_n(x'_1, x'_2, x'_3) = r'^n\, P_n(x'_1/r')$ then we have the following recurrence relation

$$Q_n = \frac{2\,n-1}{n}\, x'_1\, Q_{n-1} - \frac{n-1}{n}\, r'^n\, Q_{n-2}, \tag{4.101}$$

with the base case given by $Q_0 = 0$ and $Q_1 = x'_1$. This recurrence relation is directly derived from the usual recurrence among the Legendre polynomials.

Truncation of the Taylor Series Expansion

For brevity we will express the above Taylor expansion of $\widehat{\mathbf{E}}$ as

$$\widehat{\mathbf{E}} := \sum_{n=2}^{\infty} \mathbf{E}_n, \tag{4.102}$$

where each $\mathbf{E}_n$ is a homogeneous polynomial of degree n in the variables $(\boldsymbol{x'}, \boldsymbol{\xi'})$.

Finally, fixing an integer $M \geq 3$, we truncate the Taylor series by taking the partial sum of formula (4.99) up to degree M, dropping terms of higher degree. We denote the Hamiltonian given by this partial sum by $\mathbf{E}$ (dropping the hat, in order to indicate truncation):

$$\mathbf{E} := \sum_{n=2}^{M} \mathbf{E}_n. \tag{4.103}$$

We defer consideration of the error made in truncating the Taylor expansion until a later section.

Alternative Coordinate Systems and Coordinate Scalings

In the literature, coordinate changes other than those defined through (4.93) and (4.94) are sometimes used.

For example, it is often convenient to put the Hamiltonian $\mathbf{H}$ in terms coordinates centered on one of the primaries (usually the smallest). This approach is especially useful when studying the problem of the capture of bodies (for example, asteroids and primordial moons) by the smaller primary, via transport through the nearby bottlenecks associated with L_1 and L_2; see, for instance, (3).

In addition, it is sometimes desirable to use scalings of the coordinate system, either of canonical type or by scaling the coordinates of the original Hamiltonian so as to overcome numerical problems that may arise from the presence of the nearby singularity due to one or other of the primary bodies; see, for example, (10).

In the remainder of this treatment we will maintain the formulation given above; that is, we will leave the transformation between original and equilibrium-centered coordinates as a simple translation and will forgo any scaling.

4.5.3 Passage to Real Diagonal Coordinates

Our next task is to find a change of coordinates that puts the quadratic terms of $\mathbf{E}$ into their so-called "real normal form". We can accomplish this by means of a symplectic linear coordinate change into what we call the "real diagonal coordinates". After this, a simple change to complex coordinates puts the Hamiltonian into a suitable form for the normalization procedure. We now give the details.

The quadratic terms of the Taylor expansion

The quadratic terms of the Taylor expansion $\mathbf{E}$, for the CRTBP, are given by

$$\mathbf{E}_2 = \frac{1}{2}\,(\xi'^2_1 + \xi'^2_2 + \xi'^2_3) - (-x'_2\,\xi'_1 + x'_1\,\xi'_2) + \frac{\alpha}{2}\,(-2\,x'^2_1 + x'^2_2 + x'^2_3), \tag{4.104}$$

The eigensystem of the quadratic part

We recall that the eigenvalues of the matrix associated with the linearisation of Hamilton's equations (which corresponds to $\mathbf{E}_2$) have been computed already. We enumerate them here as follows:

$$\begin{aligned}
\gamma_{1,4} &= \pm\sqrt{\frac{\alpha - 2 + \sqrt{\alpha\,(9\,\alpha - 8)}}{2}},\\
\gamma_{2,5} &= \pm\sqrt{\alpha}\,\imath,\\
\gamma_{3,6} &= \pm\sqrt{\frac{\alpha - 2 - \sqrt{\alpha\,(9\,\alpha - 8)}}{2}}.
\end{aligned} \tag{4.105}$$

The positive signs of $\pm$ refer to γ_1, γ_2 and γ_3 while the negative signs refer to γ_4, γ_5 and γ_6.

Real versus purely imaginary eigenvalues

Since $\alpha > 1$ we can conclude that the eigenvalues $\gamma_{2,5}$ and $\gamma_{3,6}$ are purely imaginary whereas $\gamma_{1,4}$ are always real. From now on we use the following definitions:

$$\lambda = (\lambda_1, \lambda_2, \lambda_3) := (\gamma_1, \gamma_2, \gamma_3) \in \mathbb{R} \times (\imath\mathbb{R}) \times (\imath\mathbb{R}),$$

thus λ_1 is real and λ_2 and λ_3 are pure imaginary eigenvalues. We denote the magnitudes of these quantities by the vector

$$\omega = (\omega_1, \omega_2, \omega_3) := (\gamma_1, \Im\gamma_2, \Im\gamma_3) \in \mathbb{R}^3$$

Notice that the purely imaginary eigenvalues γ_2 and γ_3, which are those of "centre" type, are replaced by the real quantities $\Im\gamma_2$ and $\Im\gamma_3$ in the vector ω. Note also that for all values of α, i.e., for all values of $\mu \in [0, 1/2)$ we have that ω_1, ω_2, ω_3 are strictly positive.

The real normal form and the change to real diagonal coordinates

Using the theory of linear changes of coordinates for quadratic Hamiltonian functions, see for instance (14), we know that there is a set of variables that we denote by

$$(\boldsymbol{q}, \boldsymbol{p}) = (q_1, q_2, q_3, p_1, p_2, p_3), \tag{4.106}$$

and call the "diagonal real coordinates", and a linear change of variables that we call $\mathcal{D}$,

$$\mathcal{D} : (\boldsymbol{x'}, \boldsymbol{\xi'}) \mapsto (\boldsymbol{q}, \boldsymbol{p}), \tag{4.107}$$

under which $\mathbf{E}_2$ maps to its real normal form defined by

$$\mathbf{D}_2 = \omega_1\, q_1\, p_1 + \frac{\omega_2}{2}\,(q_2^2 + p_2^2) + \frac{\omega_3}{2}\,(q_3^2 + p_3^2). \tag{4.108}$$

Note that the new coordinates are q_1, q_2, q_3 and their associated momenta are p_1, p_2, p_3.

Constructing the linear coordinate change

In order to construct $\mathcal{D}$, we first determine the eigenvectors related to the eigenvalues given in (4.105). By direct computation and after simplifying the expressions somewhat we conclude that two linearly independent complex eigenvectors related to $\pm\lambda_2 = \pm\omega_2\,\imath$ are

$$\mathbf{y}_{2,5} = \big(0\,,\,0\,,\,\pm 1\,,\,0\,,\,0\,,\,\omega_2\,\imath\big), \tag{4.109}$$

and two linearly independent complex eigenvectors of $\pm\lambda_3 = \pm\omega_3\,\imath$ are given by

$$\begin{aligned}\mathbf{y}_{3,6} = \big(&(-1+\alpha)\,(1-2\,\alpha+\lambda_1^2)\,,\,\mp\omega_3\,(1-\alpha+\lambda_1^2)\,\imath\,,\,0\,,\\ &\pm\alpha\,\omega_3\,(2-2\,\alpha+\lambda_1^2)\,\imath\,,\,(-1+\alpha)\,\alpha\,,\,0\big).\end{aligned} \tag{4.110}$$

The corresponding eigenvectors related to $\pm\lambda_1 = \pm\omega_1$ are real and can be written as

$$\mathbf{y}_{1,4} = \big(\mp(1-\alpha)\,(1+\alpha+\lambda_1^2)\,,\,-\lambda_1\,(1+\lambda_1^2)\,,\,0\,,\,\alpha\,\lambda_1\,(\alpha+\lambda_1^2)\,,\,\pm(1-\alpha)\,\alpha\,,\,0\big). \tag{4.111}$$

Note that the upper signs of $\pm$ and $\mp$ are taken for $\mathbf{y}_1$, $\mathbf{y}_2$ and $\mathbf{y}_3$ whereas the lower ones are taken by $\mathbf{y}_4$, $\mathbf{y}_5$ and $\mathbf{y}_6$. Moreover, each $\mathbf{y}_i$ is an eigenvector associated to γ_i for $1 \leq i \leq 6$.

Now, using the eigenvectors $\mathbf{y}_i$, we seek a linear change of variables in the form of a matrix Σ under which $\mathbf{E}_2$ is transformed into its real normal form. As we require a real change, Σ must have real entries. It is convenient to associate the real eigenvalues for $\mathbf{E}_2$ with the components (q_1, p_1) of the new coordinates, whilst we relate $\omega_2\,\imath$ and $\omega_3\,\imath$ with the components (q_2, p_2) and (q_3, p_3) of the new coordinates, respectively. Hence, a possible candidate for Σ is given by

$$\Sigma = \big(\,\mathbf{y}_1\,,\,\Re\,\mathbf{y}_2\,,\,\Re\,\mathbf{y}_3\,,\,\mathbf{y}_4\,,\,\Im\,\mathbf{y}_5\,,\,\Im\,\mathbf{y}_6\big)^T. \tag{4.112}$$

Ensuring that the change is symplectic

We also require that the change of coordinates that we are proposing must be symplectic. This is achieved provided that $\Sigma^T\,J\,\Sigma = J$ where J is the skew symmetric matrix in $\mathbb{R}^6$:

$$J = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \end{pmatrix}. \tag{4.113}$$

However, Σ as given above does not fulfill this property; we need to amend it. In fact we define

$$\Sigma := \big(\,c_1\,\mathbf{y}_1\,,\,c_2\,\Re\,\mathbf{y}_2\,,\,c_3\,\Re\,\mathbf{y}_3\,,\,c_4\,\mathbf{y}_4\,,\,c_5\,\Im\,\mathbf{y}_5\,,\,c_6\,\Im\,\mathbf{y}_6\big)^T. \tag{4.114}$$

for some constant coefficients c_i to be determined. Note that the columns of Σ are also eigenvectors of the matrix related to $\mathbf{E}_2$.

Imposing the condition that $\Sigma^T\,J\,\Sigma - J = 0$ we need that $c_4 = c_1$, $c_5 = c_2$ and $c_6 = c_3$. Furthermore, for c_1, c_2 and c_3 we obtain the following values

$$\begin{aligned} c_1 &= -\frac{1}{\sqrt{2\,(-1+\alpha)\,\alpha\,\lambda_1\,(-2+3\,\alpha^2-2\,\lambda_1^2+3\,\alpha\,\lambda_1^2)}}, \\ c_2 &= -\frac{1}{\alpha^{1/4}}, \\ c_3 &= -\frac{1}{\sqrt{(-1+\alpha)\,\alpha\,\omega_3\,(2-8\,\alpha+6\,\alpha^2+2\,\lambda_1^2-3\,\alpha\,\lambda_1^2)}}. \end{aligned} \tag{4.115}$$

For all $\alpha > 1$ we know that $c_1 < 0$, $c_2 < 0$ and $c_3 < 0$. Thus, after replacing the values of c_i in Σ, it is straghtforward to prove that all entries of Σ are real, that the identity $(\Sigma)^T\,J\,\Sigma = J$ holds, and that $\det(\Sigma) = 1$ which demonstates that the change is canonical. We then define $\mathcal{D}^{-1} := \Sigma$ and $\mathcal{D} := \Sigma^{-1}$.

The real diagonal Hamiltonian

Finally, we express $\mathbf{E}$ as a function of the new coordinates, giving a Hamiltonian $\mathbf{D}$ that can be written as

$$\mathbf{D} = \sum_{i=2}^{M} \frac{1}{(i-2)!} \mathbf{D}_i(\boldsymbol{q}, \boldsymbol{p}), \tag{4.116}$$

where each $\mathbf{D}_i$ is a homogenous polynomial in $\boldsymbol{q}, \boldsymbol{p}$ of degree i with real coefficients, and where the quadratic part $\mathbf{D}_2$ is in real normal form.

The reader should note that care must be taken with all indices and quantities and with all factorials to ensure that a consistent scheme is used throughout all computations.

4.5.4 Passage to Complex Coordinates

The final stage in preparing the Hamiltonian for our normalisation procedure is to convert the diagonal real form into a complex one. In the complex form the quadratic part of the Hamiltonian may be written solely in terms of the products of configuration space coordinates and their respective conjugate momenta, as we require to begin the normalization process detailed ealier.

The complexifying change

To this end, we introduce the complex variables,

$$(\boldsymbol{Q}, \boldsymbol{P}) = (Q_1, Q_2, Q_3, P_1, P_2, P_3) \tag{4.117}$$

formed by three coordinates Q_1, Q_2, Q_3 plus their respective conjugate momenta P_1, P_2, P_3. We refer to these coordinates as the "diagonal complex coordinates". We denote the transformation from real coordinates to complex coordinates by

$$\mathcal{C} : (\boldsymbol{q}, \boldsymbol{p}) \mapsto (\boldsymbol{Q}, \boldsymbol{P}), \tag{4.118}$$

which is given explicitly by:

$$\begin{aligned} Q_1 &= q_1, \quad Q_2 = \tfrac{q_2 - \imath\, p_2}{\sqrt{2}}, \quad Q_3 = \tfrac{q_3 - \imath\, p_3}{\sqrt{2}}, \\ P_1 &= p_1, \quad P_2 = \tfrac{-\imath\, q_2 + p_2}{\sqrt{2}}, \quad P_3 = \tfrac{-\imath\, q_3 + p_3}{\sqrt{2}}, \end{aligned} \tag{4.119}$$

The inverse linear change denoted by $\mathcal{C}^{-1} : (\boldsymbol{Q}, \boldsymbol{P}) \mapsto (\boldsymbol{q}, \boldsymbol{p})$ is defined by

$$\begin{aligned} q_1 &= Q_1, \quad q_2 = \tfrac{Q_2 + \imath\, P_2}{\sqrt{2}}, \quad q_3 = \tfrac{Q_3 + \imath\, P_3}{\sqrt{2}}, \\ p_1 &= P_1, \quad p_2 = \tfrac{\imath\, Q_2 + P_2}{\sqrt{2}}, \quad p_3 = \tfrac{\imath\, Q_3 + P_3}{\sqrt{2}}. \end{aligned} \tag{4.120}$$

The complex diagonal Hamiltonian

Now, the real diagonal Hamiltonian (4.116) is converted into a new Hamiltonian,

$$\mathbf{D}^{\mathbf{c}} = \sum_{i=2}^{M} \frac{1}{(i-2)!} \mathbf{D}^{\mathbf{c}}{}_i(\boldsymbol{Q}, \boldsymbol{P}). \tag{4.121}$$

As the changes (4.120) and (4.119) are linear, it is straightforward to deduce that each $\mathbf{D}^{\mathbf{c}}{}_i$ is a homogeneous polynomial of degree i in the coordinates $(Q_1, Q_2, Q_3, P_1, P_2, P_3)$ with complex or real coefficients. More importantly, $\mathbf{D}^{\mathbf{c}}{}_2$ has the required form of a quadratic Hamilton function to which Birkhoff's normal form transformation may then be applied. Specifically, we arrive at

$$\begin{aligned} \mathbf{D}^{\mathbf{c}}{}_2 &= \lambda_1 (Q_1 P_1) + \imath\, \omega_2 (Q_2 P_2) + \imath\, \omega_3 (Q_3 P_3), & (4.122) \\ &= \lambda_1 (Q_1 P_1) + \lambda_2 (Q_2 P_2) + \lambda_3 (Q_3 P_3). & (4.123) \end{aligned}$$

Note that all of the terms in the quadratic part are now products of pairs of complex coordinates.

4.5.5 Introducing the parameter

In order to adapt the formulae of Section 4.2.2 to our normal form construction we "stretch" the coordinates following Section 4.2.2, writing $(\boldsymbol{Q}, \boldsymbol{P}) = \varepsilon\,(x, \xi)$ and defining

$$\mathcal{H}(\mathbf{x}; \varepsilon) = \mathcal{H}((x, \xi); \varepsilon) := \varepsilon^{-2} \mathbf{D}^{\mathbf{c}}(\varepsilon(x, \xi)) = \varepsilon^{-2} \mathbf{D}^{\mathbf{c}}(\boldsymbol{Q}, \boldsymbol{P}),$$

so as to introduce the parameter ε; after our formal computations we simply set $\varepsilon = 1$. (Note that we have chosen here that the parameter measures the order of perturbation in the coordinates.)

Finally we have arrived at a Hamiltonian $\mathcal{H}$ that is in a suitable form to apply the normalization procedure.

We note that, in a practical computation of a normal form, it is not actually necessary to deal with the parameter ε at all; instead, one can keep track of the various monomials by using the total degree to which the coordinates are raised, and all computations may then be structured to treat such monomials in a consistent way.

4.5.6 The normalization procedure

In the above, we have shown how to take a Hamiltonian with an equilibrium point of saddle-centre-centre type and to make a series of coordinate changes under which the new Hamiltonian has a quadratic part that may be written solely as a polynomial in the products of configuration space coordinates with their respective conjugate momenta. We now demonstrate how to apply the normalization procedure to this Hamiltonian.

Complex normal form coordinates

The normalization of $\mathcal{H}(\mathbf{x})$ yields the normalised Hamiltonian $\mathcal{K}(\mathbf{y})$ in terms of (complex) normal form coordinates $\mathbf{y} = (y, \eta)$. We denote $\mathbf{N}^{\mathsf{c}} := \mathcal{K}$ and call $\mathbf{N}^{\mathsf{c}}$ the *complex normal form Hamiltonian*. We denote the new variables by $(\boldsymbol{Q'}, \boldsymbol{P'}) := \mathbf{y}$ by

$$(\boldsymbol{Q'}, \boldsymbol{P'}) = ({Q'}_1, {Q'}_2, {Q'}_3, {P'}_1, {P'}_2, {P'}_3), \tag{4.124}$$

which we will call the "complex normal form coordinates".

The direct and inverse coordinate changes

The direct and inverse changes, which take us from complex diagonal to complex normal form coordiantes and back again, will be denoted by

$$\mathcal{N} : (\boldsymbol{Q}, \boldsymbol{P}) \mapsto (\boldsymbol{Q'}, \boldsymbol{P'}), \tag{4.125}$$

and

$$\hat{\mathcal{N}} : (\boldsymbol{Q'}, \boldsymbol{P'}) \mapsto (\boldsymbol{Q}, \boldsymbol{P}), \tag{4.126}$$

respectively. Thus $\mathcal{N} := \mathcal{Y}$ and $\hat{\mathcal{N}} := \mathcal{X}$ in our earlier notation.

Note that we deliberately avoid using the misleading notation $\mathcal{N}^{-1}$ for $\hat{\mathcal{N}}$ because in practice the maps that we compute will not be exactly mutual inverses due to the error incurred in performing the normalisation procedure. We consider matters of convergence and error bounds in a later section.

4.5.7 The complex normal form and integrals

Taking the complex diagonal Hamiltonian as the input, we perform the normalization procedure, as detailed in the previous section. After a numerical computation of the terms ${\mathbf{N}^{\mathsf{c}}}_i$ (of the complex normal form Hamiltonian) and W_i (of the corresponding generating function), it is advisable to check that the homological equation is really satisfied at each step. This ensures that the computations are consistent.

The complex normal form Hamiltonian

If the procedure is carried up to terms of degree M, then the normal form Hamiltonian reads as follows,

$$\begin{aligned} \mathbf{N}^{\mathsf{c}} &= \sum_{i=2}^{M} \frac{1}{(i-2)!}\, {\mathbf{N}^{\mathsf{c}}}_i(\boldsymbol{Q'}, \boldsymbol{P'}) \\ &= \sum_{j,k,\ell}^{M} a(j,k,\ell)\, ({Q'}_1\, {P'}_1)^j\, ({Q'}_2\, {P'}_2)^k\, ({Q'}_3\, {P'}_3)^\ell, \end{aligned} \tag{4.127}$$

where the $a(j,k,l)$ are complex or real coefficients.

Note that $\mathbf{N}^{\mathsf{c}}$ is a polynomial of degree M in $(\boldsymbol{Q'}, \boldsymbol{P'})$.

The complex generating function

The generating function W reads as

$$W = \sum_{i=2}^{M-1} \frac{1}{(i-2)!} W_{i+1}(\boldsymbol{Q'}, \boldsymbol{P'}), \tag{4.128}$$

where each W_i is a homogeneous polynomial in the coordinates $(\boldsymbol{Q'}, \boldsymbol{P'})$, or equivalently in the coordinates $(\boldsymbol{Q}, \boldsymbol{P})$, of degree i.

The complex changes of coordinates

The direct change, that is, the passage of the complex coordinates prior to normalization to the normalized complex coordinates is obtained through

$$(\boldsymbol{Q}, \boldsymbol{P}) = \mathcal{N}(\boldsymbol{Q'}, \boldsymbol{P'}) = (\boldsymbol{Q'}, \boldsymbol{P'}) + \sum_{i=2}^{M} \frac{1}{(i-1)!} (\boldsymbol{Q'}, \boldsymbol{P'})_0^{(i)}. \tag{4.129}$$

In the above, the terms $(\boldsymbol{Q'}, \boldsymbol{P'})_0^{(i)}$ are calculated from $(\boldsymbol{Q'}, \boldsymbol{P'})$ and W using equation (4.9).

Note that for all $1 \leq i \leq M$, the six components of $(\boldsymbol{Q'}, \boldsymbol{P'})_0^{(i)}$ are homogeneous polynomials of degree i.

The inverse change, that is, the transformation of the normalized complex coordinates to the complex coordinates prior to normalization, is determined analogously with the aid of

$$(\boldsymbol{Q'}, \boldsymbol{P'}) = \hat{\mathcal{N}}(\boldsymbol{Q}, \boldsymbol{P}) = (\boldsymbol{Q}, \boldsymbol{P}) + \sum_{i=2}^{M} \frac{1}{(i-1)!} (\boldsymbol{Q}, \boldsymbol{P})_0^{(i)}. \tag{4.130}$$

The terms $(\boldsymbol{Q}, \boldsymbol{P})_0^{(i)}$ are obtained from $(\boldsymbol{Q}, \boldsymbol{P})$ and W using equation (4.11).

Again, for all $1 \leq i \leq M$, the six components of $(\boldsymbol{Q'}, \boldsymbol{P'})_0^{(i)}$ are homogeneous polynomials of degree i.

Exact integrals for the complex normal form Hamiltonian

As we proved in the earlier section, under the assumption that the eigenvalues in the vector λ are non-resonant, the normalized Hamiltonian $\mathbf{N}^{\mathbf{c}}$ defines a zero-degrees of freedom dynamical system for which the three products

$$\mathcal{I}_1 = {Q'}_1 {P'}_1, \qquad \mathcal{I}_2 = {Q'}_2 {P'}_2, \qquad \mathcal{I}_3 = {Q'}_3 {P'}_3, \tag{4.131}$$

become, by construction, exact integrals.

Asymptotic integrals for the complex diagonal Hamiltonian

These three integrals of the complex normal form Hamiltonian $\mathbf{N}^{\mathbf{c}}$ correspond to three asymptotic integrals of the complex diagonal Hamiltonian, in other words, of the Hamiltonian $\mathbf{D}^{\mathbf{c}}$ prior to normalization.

In particular, we may obtain explicit expressions for the asymptotic integrals of $\mathbf{D}^{\mathrm{c}}$ by applying the change of coordinates $(\boldsymbol{Q'}, \boldsymbol{P'}) = \hat{\mathcal{N}}(\boldsymbol{Q}, \boldsymbol{P})$ to the products $Q'_1 P'_1$, $Q'_2 P'_2$ and $Q'_3 P'_3$ using the generating function $\mathcal{W}$. These integrals may also be expressed in real coordinates $(q_1, q_2, q_3, p_1, p_2, p_3)$ using the transformation (4.119).

4.5.8 The real normal form and integrals

The real normal form coordinates

We now show how to express the normal form Hamiltonian in terms of real variables.

Thus we introduce the so-called *real* normal form coordinates,

$$(\boldsymbol{q'}, \boldsymbol{p'}) = (q'_1, q'_2, q'_3, p'_1, p'_2, p'_3), \tag{4.132}$$

which can be related to the complex normal form ones $(\boldsymbol{Q'}, \boldsymbol{P'})$ by using (4.119).

Recall that we denote this change by $\mathcal{C}^{-1}$;

$$\mathcal{C}^{-1} : (\boldsymbol{Q'}, \boldsymbol{P'}) \mapsto (\boldsymbol{q'}, \boldsymbol{p'}), \tag{4.133}$$

and its inverse

$$\mathcal{C} : (\boldsymbol{q'}, \boldsymbol{p'}) \mapsto (\boldsymbol{Q'}, \boldsymbol{P'}). \tag{4.134}$$

The real normal form Hamiltonian, generating function, and coordinate changes

In this way we write the normal form Hamiltonian $\mathbf{N}$ (in terms of the real variables) as

$$\begin{aligned} \mathbf{N} &= \sum_{i=2}^{M} \frac{1}{(i-2)!} \mathbf{N}_i(\boldsymbol{q'}, \boldsymbol{p'}) \\ &= \sum_{j,k,\ell}^{M} b(j,k,\ell)\, (q'_1 p'_1)^j\, (q'^2_2 + p'^2_2)^k\, (q'^2_3 + p'^2_3)^\ell, \end{aligned} \tag{4.135}$$

where this time $b(j,k,\ell)$ are real coefficients.

The same change can be applied to W with the goal of determining the real version of the generating function.

Thus, one could construct the direct and inverse change of coordinates for the real diagonal coordinates, thereby relating the coordinates $(\boldsymbol{q'}, \boldsymbol{p'})$ and $(\boldsymbol{q}, \boldsymbol{p})$.

The real form of the integrals

In addition, the three integrals expressed in terms of the real normal form coordinates are as follows:

$$\begin{aligned} \mathcal{I}_1 &= q'_1 p'_1 = Q'_1 P'_1, \\ \mathcal{I}_2 &= \frac{1}{2}\left(q'^2_2 + p'^2_2\right) = \imath\, Q'_2 P'_2, \\ \mathcal{I}_3 &= \frac{1}{2}\left(q'^2_3 + p'^2_2\right) = \imath\, Q'_3 P'_3. \end{aligned} \tag{4.136}$$

The real normal form Hamiltonian in terms of the integrals

We can see the trivial form acquired by $\mathbf{N}$ in terms of these intergrals:

$$\mathbf{N} = \sum_{i=2}^{M} \frac{1}{(i-2)!} \mathbf{N}_i = \sum_{j,k,\ell}^{M} b(j,k,\ell)\, \mathcal{I}_1^j\, \mathcal{I}_2^k\, \mathcal{I}_3^\ell, \tag{4.137}$$

Note that Hamiltonian $\mathbf{N}$ is a polynomial of degree $[M/2]$ (integer part) in terms of the integrals $\mathcal{I}'s$.

We call the integral $\mathcal{I}_1$ the "saddle integral" and the integrals $\mathcal{I}_2$, $\mathcal{I}_3$ are the "centre integrals".

4.5.9 Small denominators for the CRTBP

We now examine the possible occurrence of small denominators in the Hamiltonian systems corresponding to L_1, L_2 and L_3 of the CRTBP.

The eigenvalues of the system

Recall the form of the eigenvalues given by (4.105); there are two purely imaginary (conjugate) pairs and one purely real pair. Purely real and purely imaginary values cannot be related by a *real*-integer resonance relation. Thus, the only possibility for the appearance of resonances is that the four purely imaginary eigenvalues $\gamma_{1,2}$ and $\gamma_{3,4}$ satisfy $\gamma_{1,2}/\gamma_{3,4} \approx r$ with $r \in \mathbb{Q}$, equivalently $\omega_2/\omega_3 = r$.

Possible resonances

Substituting the values of ω_2 and ω_3 in (4.105) and solving the equations $\omega_2/\omega_3 = r$ for r we arrive at two possible values for r:

$$r = \pm\sqrt{\frac{2\,\alpha}{2-\alpha+\sqrt{\alpha\,(9\,\alpha-8)}}}. \tag{4.138}$$

Thus, if we fix a value of $\alpha > 1$, there are two possible values of $r \in \mathbb{Q}$ such that small divisors can occur. Hence there are possible rational values of r for which $\alpha > 1$ and some resonances may appear.

If, on the contrary, we fix a certain r we would obtain two, one or zero values of α as a function of r, which would tell us for a given resonance the possible values of α for which the small denominators appear.

The normal form in the resonant case

As we mentioned above for the resonant case, when dealing with Birkhoff's normal form construction, one should retain in the new Hamiltonian those monomials causing the occurrence of small divisors. In general, this leads to a different form for the normalized Hamiltonian and to the loss of formal integrals.

In practice, this situation does not happen very often in the spatial restricted three body problem for small values of μ, in the sense that any resonance are of high order and thus do not affect the procedure unless the computations are taken to an exceptionally high-degree.

Bibliography

[1] V. I. Arnold, *Mathematical methods of classical mechanics*, Graduate Texts in Mathematics, vol. 60, Springer, New York, Heidelberg, Berlin, 1978.

[2] V. I. Arnol'd, V. V. Kozlov, and A. I. Neishtadt, *Mathematical aspects of classical and celestial mechanics*, Dynamical Systems III (V. I. Arnol'd, ed.), Encyclopaedia of Mathematical Sciences, vol. 3, Springer, Berlin, 1988.

[3] S.A. Astakhov, A.D. Burbanks, S. Wiggins, and D. Farrelly, *Chaos assisted capture of irregular moons*, Nature **423** (2003), 264–267.

[4] G.D. Birkhoff, *Dynamical systems*, A.M.S. Coll. Publications, vol. 9, American Mathematical Society, Rhode Island, Providence, 1927.

[5] R. C. Churchill and M. Kummer, *A unified approach to linear and nonlinear normal forms for Hamiltonian systems*, J. Symb. Comp. **27** (1999), 49–131.

[6] A. Deprit, *Canonical transformations depending on a small parameter*, Celestial Mech. **1** (1969), 12–30.

[7] A.J. Dragt and J.M. Finn, *Lie series and invariant functions for analytic symplectic maps*, J. Math. Phys. **17** (1976), no. 12, 2215–2227.

[8] F.G. Gustavson, *On constructing formal integrals of a hamiltonian system near an equilibrium point*, Astron. J. **71** (1966), no. 8, 670–686.

[9] J. Henrard, *The algorithm of the inverse lie transform*, Recent Advances in Dynamical Astronomy (Dordrecht) (V. Szebehely and B.D. Tapley, eds.), D. Reidel Publishing Company, 1973, pp. 250–259.

[10] A. Jorba and J. Masdemont, *Dynamics in the center manifold of the collinear points of the restricted three body problem*, Physica D **132** (1999), 189–213.

[11] C. Marchal, *The three-body problem*, Elsevier, Amsterdam, 1990.

[12] K.R. Meyer, *Normal forms for hamiltonian systems*, Celestial Mech. **9** (1974), 517–522.

[13] _____, *A lie transform tutorial ii*, Computer Aided Proofs in Analysis (New York, Heidelberg, Berlin) (K.R. Meyer and D.S. Schmidt, eds.), The IMA Volumes in Mathematics and its Applications, vol. 28, Springer-Verlag, 1991, pp. 190–210.

[14] K.R. Meyer and G.R. Hall, *Introduction to hamiltonian dynamical systems and the n-body problem*, Applied Mathematical Sciences, vol. 90, Springer-Verlag, Berlin, Heidelberg, New York, 1992.

[15] J. Murdock, *Normal Forms and Unfoldings for Local Dynamical Systems*, Springer, New York, 2003.

[16] C.D. Murray and S.F. Dermott, *Solar system dynamics*, Cambridge University Press, Cambridge, New York, Melbourne, Madrid, 1999.

[17] J. Palacián and P. Yanguas, *Reduction of polynomial hamiltonians by the construction of formal integrals*, Nonlinearity **13** (2000), 1021–1054.

[18] H. Poincaré, *Sur les courbes définies par les équations différentielles*, J. Math. Pures App. **quatrième série, tome premier, troisième partie** (1885), 167–244.

[19] R. Schubert, H. Waalkens, and S. Wiggins, *Efficient computation of transition state resonances and reaction rates from a quantum normal form*, Phys. Rev. Lett. **96** (2006), 218302.

[20] C.L. Siegel and J.K. Moser, *Lectures on celestial mechanics*, Springer-Verlag, New York, Heidelberg, Berlin, 1971.

[21] V. Szebehely, *The theory of orbits*, Academic Press, New York, 1967.

[22] T. Uzer, C. Jaffe, J. Palacian, P. Yanguas, and S. Wiggins, *The geometry of reaction dynamics*, Nonlinearity **15** (2002), 957–992.

[23] H. Waalkens, A. Burbanks, and S. Wiggins, *A computational procedure to detect a new type of high-dimensional chaotic saddle and its application to the 3D Hill's problem*, J. Phys. A **37** (2004), L257–L265.

[24] ———, *Phase space conduits for reaction in multidimensional systems: HCN isomerization in three dimensions*, J. Chem. Phys. **121** (2004), no. 13, 6207–6225.

[25] ———, *Efficient procedure to compute the microcanonical volume of initial conditions that lead to escape trajectories from a multidimensional potential well*, Physical Review Letters **95** (2005), 084301.

[26] ———, *A formula to compute the microcanonical volume of reactive initial conditions in transition state theory*, J. Phys. A **38** (2005), L759–L768.

[27] H. Waalkens, R. Schubert, and S. Wiggins, *Wigner's dynamical transition state theory in phase space: Classical and quantum*, Nonlinearity **21** (2008), R1–R118.

[28] H. Waalkens and S. Wiggins, *Direct construction of a dividing surface of minimal flux for multi-degree-of-freedom systems that cannot be recrossed*, J. Phys. A **37** (2004), L435–L445.

[29] S. Wiggins, *Introduction to applied nonlinear dynamical systems and chaos, second edition*, Springer-Verlag, New York, 2003.