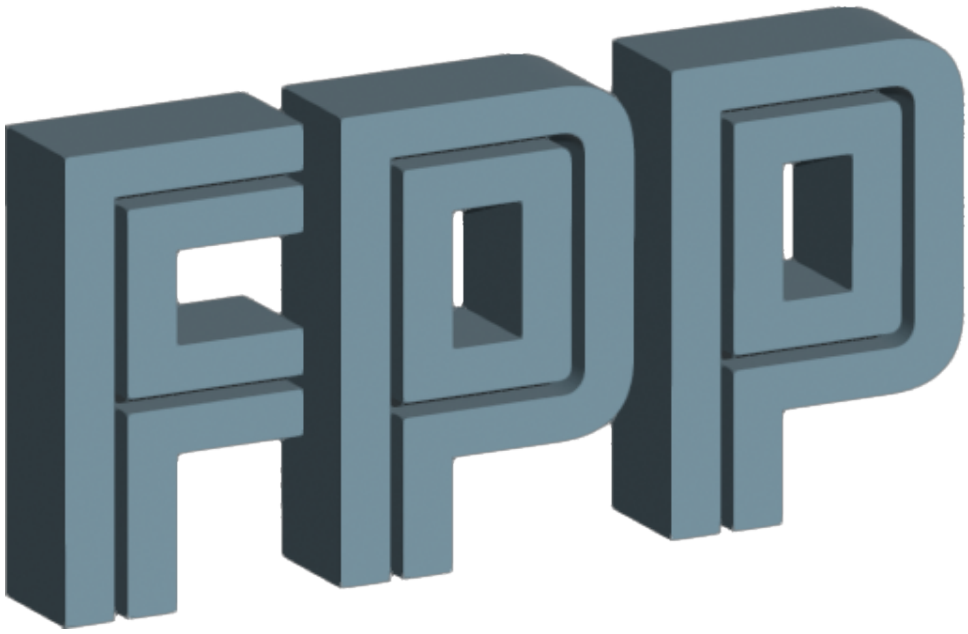
Fully Polymorphic Package Manual



Etienne Forest and David Sagan
March 25, 2023

Contents

1 Introduction to packages within FPP/PTC

The name PTC is commenly used to designate a code written by Etienne Forest with the help of Frank Schmidt and later David Sagan. PTC is an object oriented, open source, subroutine library for

- The manipulation and analysis of Taylor series and Taylor maps.
- Modeling of charged particle beams in accelerators using Taylor maps.

The popularity of PTC can be attested to by its use with the Bmad[11] toolkit for accelerator and X-ray simulations as well as its use within the MAD[8] simulation program.

PTC is linked with the overloaded Truncated Power Series Algebra (TPSA) package of Martin Berz. Berz's package is called **Polymorphic Package** (PP). By virtue of using the polymorphic package and by virtue of being able to do tracking, PTC stands for "Polymorphic Tracking Code". PTC is can be decomposed into two parts:

1. The Fully Polymorphic Package (FPP) part handles Taylor series. FPP is pure mathematics detached from any "physics".
2. The Tracking Code (TC) part is where the physics of PTC is located. It is simply an integrator pushing the phase space and spin of particles through magnets. TC relies on FPP for handling Taylor series. In particular, via the magic of FPP (specifically the magic of PP), a Taylor map can be produced if wanted when tracking with TC.

FPP is the subject of this manual. FPP itself can be subdivided into two parts:

1. The "Polymorphic Package" (the PP of FPP) deals with the production of Taylor series.
2. The Analysis Package (AP) analyses physically sensible Taylors maps: Taylor maps that approximate the tracking of the code. Spin and magnet modulation can also be analyzed.

To appreciate the role of PP, we can show a tiny piece of code TC representing a drift.

```
                        .
                        .
                        .
  TYPE(REAL_8),INTENT(INOUT):: X(6)
  TYPE(REAL_8),INTENT(IN):: L
                        .
                        .
                        .
        PZ=SQRT(1.0_dp+2.0_dp*X(5)/b+x(5)**2-X(2)**2-X(4)**2)
        X(1)=X(1)+L*X(2)/PZ
        X(3)=X(3)+L*X(4)/PZ
```

For completeness, here is type **real_8**:

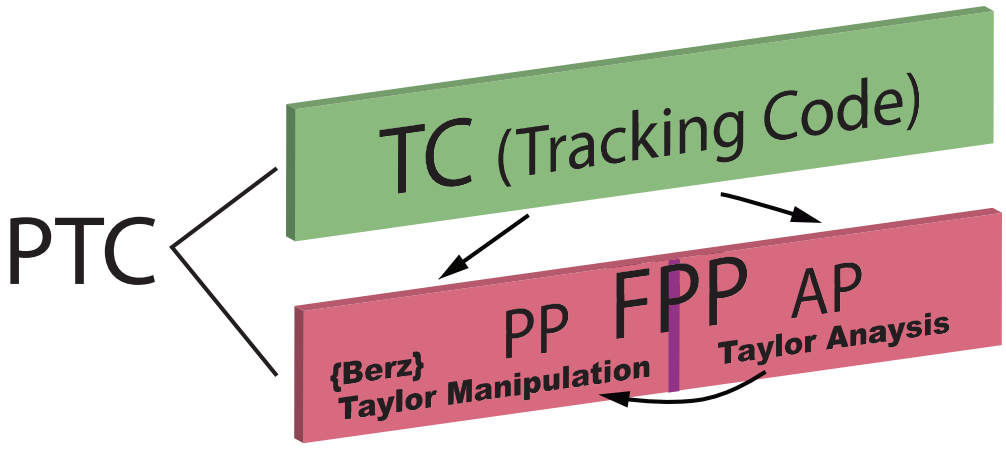


Figure 1: The Fully Polymorphic Package (FPP) part of the FPP/PTC library provides manipulation and analysis of Taylor series and maps and the Tracking Code (TC) part contains the physics of the accelerators. Arrows indicate code dependencies. The Taylor analysis code uses the Taylor manipulation code but not vice versa. FPP does not use PTC but PTC code uses both FPP's Taylor manipulation and potentially analysis as the red arrow indicates.

```
  TYPE REAL_8
    TYPE (TAYLOR) T       !  USED IF TAYLOR
    REAL(DP) R            !    USED IF REAL
    INTEGER KIND  !  0,1,2,3 (1=REAL,2=TAYLOR,3=TAYLOR KNOB, 0=SPECIAL)
    INTEGER I    !   USED FOR KNOBS AND SPECIAL KIND=0
    REAL(DP) S    !   SCALING FOR KNOBS AND SPECIAL KIND=0
    LOGICAL(lp) :: ALLOC  ! IF TAYLOR IS ALLOCATED IN DA-PACKAGE
END TYPE REAL_8
```

In the above definition, we see **real_8%t** and **real_8%r**. They are respectively the Taylor and double precision entries of the polymorph. If the polymorph must for some reason become a Taylor series, then its value is stored in **%t**, otherwise it is stored in **%r**. A **real_8** can also be a "knob": this is described in section (§7.2).

If type **REAL_8** was simply **real(8)**, then this code fragment would simply be the exact formula for a drift in canonical variables. However **REAL_8** can become a Taylor series in certain chosen variables at execution time and thus Taylor series can be produced. The actual package that does the Taylor calculation is the package of Martin Berz[2] , namely the old version from LBNL properly modified by Johan Bengtson and overloaded by Etienne Forest using Fortran90.

This manual is focused on how to use FPP. Since FPP is designed to serve PTC or a code like PTC, there will be some mention of PTC but only so far as how PTC relates to FPP. In particular, tracking through lattices is not discussed and the reader is referred to the PTC documentation for this.

The red arrow in Figure 1 refers to a potential dependency. PTC can dump on files Taylor series which can be analyzed or tracked by another code. Conversely, PTC (via FPP), can read maps

produced by other codes and track them. For example, a person might have a COSY-INFINITY version of a damping ring and the subsequent injection line. The Taylor map describing this structure can be an input to PTC in the absence of a PTC lattice for this structure: this is equivalent to inputting linear uncoupled lattice functions in a regular code but much more general.

In fact most analysis in PTC in done by people (including the authors) who write independent modules for generalized Twiss calculations. This will be illustrated later and is the main topic of reference [6].

The FPP package can be used with any tracking code including TC. Indeed one can write an integrator in Fortran90 using the polymorphic types of FPP for the production and analysis of Taylor maps: Forest has written many such examples in lectures.

2 Where to Obtain FPP/PTC

FPP/PTC can be downloaded from the web via the Bmad web site[11] or at

```
https://github.com/jceepf/fpp_book
```

Taylor manipulation routines are contained in the following code files:

```
a_scratch_size.f90                j_tpsalie.f90
b_da_arrays_all.f90               k_tpsalie_analysis.f90
b_da_arrays_all_pancake.f90       l_complex_taylor.f90
c_dabnew.f90                      m_real_polymorph.f90
c_dabnew_pancake.f90              n_complex_polymorph.f90
d_lielib.f90                      o_tree_element.f90
h_definition.f90                  Sa_extend_poly.f90
i_tpsa.f90
```

Taylor analysis routines are contained in the following code files

```
cb_da_arrays_all.f90               Ci_tpsa.f90
cc_dabnew.f90                      Su_duan_zhe_map.f90
```

The library

```
Su_duan_zhe_map.f90
```

can actually be used separately. It allows for tracking using Taylor maps: remember that PTC proper (TC) does not use Taylor maps for tracking, it is a "kick code" or integrator.

3 Concepts: Taylor series, polymorphs and Taylor maps

The purpose of this section is to explain why we would like to write an integrator equipped with polymorphism. Of course the purpose exists before the code and, for the most part, is not the result of "Darwinian evolution".

It is true from a goal driven point of view (teleological) that PTC was created with the production of Taylor maps of phase space in mind. This is simply because Forest is convinced, see references [5] and [6], that an analytical theory primarily based on finite "s" maps is superior than a theory primarily based on the s-dependent Hamiltonian of the Courant-Snyder theory in the context of accelerators. This s-dependent Hamiltonian is far from the code and thus not adequate as the primary theory for most applications.

But can we guess from TC's structure, the tracking part, that PTC prioritizes a map-based theory for analysis? The answer as we will see is :no.

There are two options if we have Taylor maps in mind:

1. We can write a beam line code whose goal is the production of Taylor series approximate maps around some chosen orbit and whose internal structure reflects this goal. The prime representative of such codes is COSY-INFINITY[10, 9]. This is also true of MARYLIE, TRANSPORT[4] and the Twiss modules part of MAD8 ported into MAD-X. All these codes choose a "design" orbit and then compute Taylor maps around this orbit. If a user examine these codes, especially TRANSPORT, Taylor series maps will jump in their faces: arrays of monomial coefficients for each phase space variables are defined and they contain increasingly complex formulae as the degree increases. The TRANSPORT approach reached a grotesque end point with the code COSY 5.0 [1] of Berz : Fortran formulae at the fifth order Taylor coefficients covered pages of computer generated algebra.

2. Forest, following mainly the philosophy of Talman, prefers the usage of integrators, in particular, symplectic integrator in the simulations of LINACs, recirculators and rings. (See Forest's review article [7] for a comprehensive opinionated description of this topic.) As a result, and irrespective of teleological consideration i.e., wanting Taylor maps for analysis– Twiss, etc..., TC is a (mostly symplectic) integrator whose main purpose is the tracking of rays. It is only via polymorphism that this integrator can, under some very specific conditions, produce a Taylor map which approximates the phase space maps produced by the likes of COSY-INFINITY or MARYLIE.

In fact Forest produced a "kick code". Imagine if the type **real_8** was a Fortran90 intrinsic. Anyone could write a kick code using the floating point **real(8)**. The fact that same person could recompile the code using **real_8** would not automatically elevate the code to a code which produces sensible phase space maps. COSY-INFINITY, MARYLIE or TRANSPORT on the other hand always produce sensible maps when applied to magnets: this is their end purpose and their internal structure can be guessed from their purpose.

Looking internally at TC, it is not possible to see any maps [2]: only tracking of **real_8** completely mimicking the tracking of **real(8)**.

Therefore it is not mathematically true that PTC produces "Taylor maps" despite Forest's motives. Sometimes PTC produces Taylor series and it is not clear to the author of PTC (Forest)

[1] This is not to be confused with COSY-INFINITY which uses Berz's TPSA package. COSY 5.0[3] was really the end of the dinosaur line of TRANSPORT like codes. It is mercifully extinct.

[2] A now retired CERN scientist exclaimed upon looking in the guts of PTC: "where are the beta functions?" The reader should understand upon reading this manual that this question is ridiculous: they are nowhere. PTC is unaware of them even when it tracks them! This is the central topic of reference [6].

what kind of maps should be created. An example of this is provided in sections (§5.2) and (§5.3): from the Taylors series produced by PTC we construct a contracting map to be used in a Newton search. In that case, the user must write the interface since no one can guess *a priori* what the user has in mind.

We will see that PTC provides an interface between TC and AP which privileges the production of phase space maps so that the users can *de facto* state that for the most part PTC facilitates the production of Taylor maps related to phase space via the derived type **real_8**. But this is not visible in TC proper and is only true because Forest has programmed a very specific interface between polymorphs and Taylor maps. He could have left that task to the user (Bmad and MAD-X programmers) but they would have complained. Forest himself as the prime user would have complained schizophrenically to himself.

In the rest of this section we explain TC's interaction with FPP. It is thus necessary to introduce certain concepts in very broad terms. Since PTC is an existing code, we can use its existing structures to describe the general approach. Someone else, writing in a different language would most likely be tempted to use similar concepts and perhaps do a better job.

After general considerations to set the tone and provide some context, we will examine 3 cases in section (§3.7):

1. In section (§5.1), we extract a phase space map. We use special constructs and routines that facilitates this for the user. Namely they connect TC to AP. In AP, normal forms for example have very special assumptions : they deal with linearly stable spin-orbit maps for example.
2. In sections (§5.2) and (§5.3), we create a contraction map that fits the output of TC that has no connection with accelerator theory proper. FPP is used directly without the interface to AP. This program can be understood without the slightest understanding of TC and what it allegedly computes.
3. In section (§5.4), we fit the orbit (as in item 2) and the tunes/phase-slip. This is a typical accelerator physics case. We create the usual one-turn maps as well as the Langrangian contraction maps of item 2.

3.1 Conventions

FPP/PTC is written in Fortran90. It is assumed that the reader has some familiarity with this language. In particular, it is assumed that the reader knows what a **structure** is (roughly corresponding to a **class** in Python or C++) which is also called a **derived type**. Additionally it is assumed that the reader knows about operator overloading.

FPP/PTC uses double precision numbers. The kind type parameter "**dp**" is defined in FPP/PTC to correspond to double precision numbers. For example:

```
real(dp) abc, xyz              ! Declare double precision vars abc and xyz.
xyz = 3.4_dp * abc / 1e9_dp    ! 3.4_dp and 1e9_dp are double precision.
```

In this manual, **real(dp)** is often written as **real(8)** in case the meaning of **dp** is forgotten. However **dp** could be **16** if quadruple precision is used.

3.2 Some useful operators on Taylor series

This section contains simple operations which are useful mainly to programmers using FPP.

It must be remarked that the integral operator **.i.** loses one order in the taylor series. It is better to use the **c_universal_taylor** if integrals are needed to obtained Poisson bracket operators.

	Description (t: c_taylor)	Fortran Operator
1	Real and imaginary part of c_taylor	t=real(t) and t= aimag(t)
2	Extract the coefficient of x_i (constructing matrices)	r=t.index.i
3	$\frac{dt}{dx_i}$ and $\int t\, dx_i$	t=t.d.i and t=t.i.i
4	Extract order "i"	t=t.sub.i
5	Truncates above "i"	m=m.cut.i
6	Extracts coefficient r of monomials $x_1^{j_1} \cdots x_1^{j_{nv}}$	r=t.sub.j or r=t.sub.j
7	t=a + b x_i where a and b are real or complex	t=a+b dz_c(i)
8	Create Monomials $r\ x_1^{j_1} \cdots x_1^{j_{nv}}$ $r\ x_1^{j_1} \cdots x_1^{j_{nv}}$ $r\ x_i$	t = r.cmono.j(1 : nv) t = r.cmono.'j1...jnv' t = r.cmono.i
9	Peek coefficient t , as a c_taylor, of monomial $x_1^{j_1} \cdots x_n^{j_n}$	t = r.par.j(1 : nv) t = r.par.'j1...jn'
10	Generalization of .par. using a type called sub_taylor (inf)	t=r.par.inf or t=r.part.inf
11	Shift exponents downwards by k	t=t<=k
12	Peek and shift (operators 9 and 11 combined)	t=t<=k
13	Pseudo derivative : $d(x_i^n) = x_i^{n-1}$	t=t.k.i
14	Poisson Bracket	t=t.pb.t'

3.3 Some useful operators on c_damap and c_vector_field

	Description (M,A: c_damap and F,F': c_vector_field $F \cdot \nabla$)	Fortran Operator
1	Extract order "i"	M=M.sub.i
2	Truncates above "i"	M=M.cut.i
3	Extracts order "i"	F=F.sub.i
4	Truncates above "i"	F=F.cut.i
5	Exponenting a vector field	M=exp(F,M) or M=exp(F)
6	Logarithm of a map (look at link for optional variables)	F=ln(M)
7	Transforming a vector field with a vector field	F=exp_ad(F,F')
8	$\mathcal{A}\left(F\cdot\nabla+\hat{f}\right)\mathcal{A}^{-1}=\widetilde{F}\cdot\nabla+\hat{\widetilde{f}}$ $\widetilde{F}_k=\left(F_i\partial_i a_k^{-1}\right)\circ a$ $\widetilde{f}=\left(\widetilde{F}_k\partial_k\alpha^{-1}\right)\alpha+\alpha^{-1}f\circ a\,\alpha$ a is the orbital map and α is the quaternion	F=A*F

3.4 TPSA Versus DA

TPSA stands for "Truncated Power Series Algebra" and DA stands for "Differential Algebra." What does it mean when applied to a typical accelerator ring? Once we cut the mathematical jargon, we will see that

- TPSA operations take into account the constant part and the results change as a function of the order (see below).
- DA operations are equivalent to normal TPSA operations used around the closed orbit and thus the constant part of the map is ignored. All the coefficients of the Taylor series stay the same independently of the order invoked. It so happens that the computation of nonlinear differential operators (Lie vector fields for example), are self-consistent because they form a differential algebra. But it is much simpler in our field to state that they are self-consistent because there are no feed down terms.

It is important to understand why a non-zero constant part of a Taylor series can be problematical. To see this, consider two TPSA maps of order N. One maps x to y and the second maps y to

z:

$$y = \sum_{j=0}^{N} a_j x^j \tag{1}$$

$$z = \sum_{j=0}^{N} b_j y^j + \mathcal{O}(y^{N+1}) \tag{2}$$

In Eq. (2) it is made explicit that there are terms of order $N+1$ or higher that are being neglected. The two maps can be concatenated to form a map of w as a function of x:

$$w(x) = \sum_{j=0}^{N} c_j x^j \equiv z(y(x)) \tag{3}$$

If there is a neglected term in Eq. (2) that looks, for example, like $b_m y^m$ with $m > N$, substituting Eq. (1) into this term will result in modification of all lower order terms in Eq. (3) if, and only if, a_0 is non-zero. This is called "feed-down". That is, terms of higher order will affect the coefficients of lower order terms when TPSA maps are combined. To avoid this, maps with zero constant term (DA maps) should be used. With simulations, this generally means computing maps with respect to the orbit. For lattices with a closed geometry, this generally means computing maps with respect to the actual period-one closed orbit. For lattices with an open geometry (EG: Linacs), the reference orbit can be some orbit defined by tracking a beam from some wisely user-specified initial position.

3.5 Polymorphism

In computer programming "**polymorphism**" is the property that a given variable, object, or function can act in different ways depending upon the context. With FPP/PTC, polymorphism is used to define types that can act as if the structure components were real valued numbers or Taylor series. See the documentation of the **real_8** type (§11) as an example.

Note: Many **PTC** tracking code routines come in pairs. One routine of the pair, typically having a "r" suffix in its name, will use real variables while the other routine, typically having a "p" suffix in its name. One could track real numbers with the polymorphic routines but this with entail a substantial speed decrease and therefore we opted for a duplication of the routines.

Polymorphic types always have structure names that have a **_8** suffix.

3.6 Operator Overloading

FPP/PTC heavily uses operator overloading for the manipulation of Taylor series and polymorphs. Not only are the standard arithmetical operators $(+, -, *, /, **)$ overloaded as well as the equal sign $(=)$, but there are a number of custom operators that are defined as well. Below is a partial list.[3]

[3]Note: Fortran mandates that custom operator names begin and end with a dot ".". These operators have the lowest priority and therefore require parentheses. The usual Fortran operators, if overloaded, inherit the priorities of the

+, -
Standard addition and subtraction of Taylor series.

*
$M1 * M2$ is used with two maps for a DA concatenation while $M1$.o.M2 is a TPSA concatenation with constant part retained.

.o.

**
Powers of Taylor maps

.oo.
Powers of Taylor maps using .o. for concatenation.

The difference between **.o.** and * was explained in section (§4) using a one-dimensional example for simplicity.

3.7 Tracking Versus Analysis

An important distinction here is the difference between **tracking** and **analysis**. By "tracking" it is meant the propagation through a lattice of a single particle which typically involves six **real_8** numbers for the orbital motion and four **real_8** for spin represented via a quaternion.

However, with objects ending in **_8**, the orbital part and the quaternion can internally become Taylor series in some variables via polymorphism. *De facto*, if these polymorphs are properly initialized, then we can end up tracking Taylor series which can be turned into *bona fide* Taylor maps which approximate the beam line as they do in COSY-INFINITY or MARYLIE. Typically this is used to either compute a one-turn Taylor map or propagate a canonical transformation such as the Courant-Snyder transformation. However it could also produce a map unrelated to the usual one-turn map, for example, the closed orbit as a Taylor series in dipole strengths or anything ... [4] This is opposed to "analysis" which is the study of a Taylor transport map to extract such things as tunes, lattice functions, resonance driving terms, etc. With **FPP**, analysis is always done on Taylor maps.

3.8 TC tracks elements of the algebra ${}_{no}D_{nv}$

The purpose of this section is to unpack clearly the statement that PTC can track Taylor maps while TC tracks polymorphs which are members of an algebraic extensions of the real numbers. We do this with examples because in the end, all of this is very trivial if we focus on applications

intrinsic operators.

[4] Mathematically, single particle tracking is just tracking using Taylor series truncated at zeroth order. From the code perspective, due to the speed reduction with dealing with Taylor series, the two are distinct: the routines with suffixes **_r** and **_p**.

rather than theory. For the theory, the reader can look at Berz's book [1] and in particular the section on Levi-Civita fields[5] which provide inverses to the infinitesimals.

Since any function is a map, we usually reserve the word map to a function of phase space which, in TC, is of dimension 2 at a minimum. Therefore when PTC invokes TPSA it will always track Taylor "maps" but they can be nonsensical as a beam line map. We do allow PTC to track without any TPSA monomials reserved for the orbital motion, but this is only permitted if the TPSA package is initialized with a subroutine proper to FPP: it forces the programmer to view TC as just an "unknown" black box subroutine. We will see what it implies in section (§5.2).

Mathematically what does TC track?

A normal "kick" code tracks a subset of the field $(\mathbb{R}, +, \times)$, namely floating point numbers. When embedded in the code Bmad, the TC subroutine "**propagate**" , which we will encounter in the examples, is a function (or map in the usual mathematical sense) from :

$$\mathbb{R}^6 \stackrel{TC}{\longmapsto} \mathbb{R}^6 \tag{4}$$

$$z \longrightarrow z^f = TC(z) \tag{5}$$

$$\text{where in BMAD } z = (x, P_x/p_0, y, P_y/p_0, \beta ct, \delta P/p_0) \in \mathbb{R}^6 \tag{6}$$

PTC or **propagate** is a map of $\mathbb{R}^6$ in PTC at a minimum. (Spin and other goodies can be included).

Polymorphism extends $(\mathbb{R}, +, \times)$ to an algebra called by Berz[1] ${}_{no}D_{nv}$, i.e., to $({}_{no}D_{nv}, +, \times)$. This is the TPSA algebra described in §3.4. This algebra is a ring, not a field, i.e., not all elements have inverses. In fact all the so-called infinitesimals have no inverses. Colloquially, we might say that we deal with Taylor series not Laurent series. Thus we can say very clearly that

$${}_{no}D_{nv}{}^6 \stackrel{TC}{\longmapsto} {}_{no}D_{nv}{}^6 \tag{7}$$

$$z \longrightarrow z^f = TC(z) \tag{8}$$

$$\text{where } z = (x, p_x/p_0, y, p_y/p_0, \beta ct, \delta p/p_0) \in {}_{no}D_{nv}{}^6 \tag{9}$$

As an example, we can use a 1-d example as in Eq. (40). Consider two second order polynomials in one-variable, i.e., elements of ${}_2D_1$:

$$\begin{aligned} &\text{if} \quad m = m_0 + m_1\Delta + m_2\Delta^2 \\ &\text{and} \quad n = n_0 + n_1\Delta + n_2\Delta^2 \end{aligned} \tag{10}$$

then we can write

$$\begin{aligned} &\text{if} \quad m \equiv (m_0, m_1, m_2) \in {}_2D_1 \\ &\text{and} \quad n \equiv (n_0, n_1, n_2) \in {}_2D_1 \end{aligned} \tag{11}$$

In Eq. (11), we use the array notation for elements of the ring ${}_2D_1$ which is more in line with the actual Fortran90 code. Then we have:

$$m \pm n = (m_0 \pm n_0, m_1 \pm n_1, m_2 \pm n_2) \tag{12}$$

[5] We think that it is better for most readers to get a practical view of Berz's idea before reading his exposition of such fields.

and for multiplication,

$$m \times n = (m_0 n_0, m_0 n_1 + m_1 n_0, m_2 n_0 + m_1 n_1 + m_0 n_2) \tag{13}$$

and finally for division we have,

$$m \div n = \left(\frac{m_0}{n_0}, -\frac{m_0 n_1 - m_1 n_0}{n_0^2}, -\frac{m_0 n_0 n_2 - m_0 n_1^2 + m_1 n_0 n_1 - m_2 n_0^2}{n_0^3} \right) \tag{14}$$

It should be clear from this example that we regain the field of real (or complex) numbers if only the entry of the element of ${}_2D_1$ is used. Secondly, if $n_0 = 0$, then the division of Eq. (14) is not defined: we have a ring, not a scalar field (See Berz[1]).

When TC tracks elements of ${}_{no}D_{nv}$, it is totally "unaware" that these objects can be turned into *bona fide* phase space Taylor maps: this was shown in the drift code fragment of section (§1) where no "Taylor maps" are to be seen. In order to explain this dichotomy between tracking and analysis, between polymorphs and Taylors maps, we need to define different structures of FPP/PTC that are optimized to handle one or the other. Structures that have been designed to handle tracking are

```
TC structures

real_8
complex_8    (only used internally in TC)
probe_8
probe    (this is the real version of probe_8)
```

and for analysis here is an important subset

```
AP structures

c_taylor
c_damap
c_normal_form
c_vector_field
```

Finally we have structures which allow us to save Taylor series outside the TPSA package(s) of Berz. This is a permanent kind of storage that does not depend on the order and number of variables used by the TPSA of Berz which underpins FPP. They are totally independent.

```
Storage structures

universal_taylor
c_universal_taylor
```

While the structures that PTC uses for tracking are discussed here, the details of how to track through a lattice are deferred to the PTC documentation. Here the primary concern is FPP and analysis as well as the interaction between analysis and tracking, between AP and TC.

The interaction between the world of polymorphs **_8**, which are elements of ${}_{no}D_{nv}$ and Taylor maps, we will illustrate below in section (§5.1). In summary, the code TC produces the

usual orbital and spin but, via polymorphism, it can be fed into a structure **c_damap**, which is a Taylor map and thus can be analyzed if sensible. The result of this analysis, for example a Courant-Snyder transformation, in the form of a **c_damap**, can then be fed into a polymorphic ray (**probe_8**, and tracked again. This is how one does any type of Twiss calculation, i.e., propagates canonical transformations including spin and nonlinearities. This is the topic of reference [6].

Nota Bene:

In summary, there are 3 ways to look at a TPSA variable. They are all isomorphic mathematically but they serve very different purposes in the human brain.

1. As a polynomial in some variable, say Δ of Eq. (10), with automatic truncation at order $no+1$. Our brains are in physics mode when doing this.
2. As a polynomial in some abstract object, say Δ of Eq. (10), where $\Delta^{no+1}=0$. In that case we are dealing with an abstract representation of ${}_{no}D_{nv}$. This is useful when thinking about $({}_{no}D_{nv},+,\times)$. Our brains are in mathematical mode.
3. Finally, the algebra $({}_{no}D_{nv},+,\times)$ can be represented as an n-tuple as in Eq. (11). Here we are in a computer science mode since ultimately the computer stores all the TPSA variables in arrays with specific rules under addition and multiplication.

Item 3 is fully demonstrated if one downloads from the git site the following folder:

```
https://github.com/jceepf/fpp_book/my_demo_package
```

This folder contains a mini-TPSA and analysis package as well as a nonlinear Twiss example.

Indeed, the TSPA package of that site (no polymorph but only Taylor) contains the following definition of a Taylor derived type:

```
 TYPE my_taylor
    complex(dp) a(0:n_mono)
END TYPE my_taylor
```

This package can represent $({}_0D_3,+,\times)$, $({}_1D_3,+,\times)$, $({}_2D_3,+,\times)$, $({}_3D_3,+,\times)$ and $({}_4D_3,+,\times)$. Obviously $({}_0D_3,+,\times)$ is just the field of complex numbers $(\mathbb{C},+,\times)$ where each number is stored **my_taylor%a(0)**.

Moreover addition and subtraction are defined respectively by overloading the Fortran90 intrinsic operators via the following functions:

```
  FUNCTION add( S1, S2 )
    implicit none
    TYPE (my_taylor) add
    TYPE (my_taylor), INTENT (IN) :: S1, S2

     add%a=S1%a + S2%a

     call clean(add)
```

```
END FUNCTION add


FUNCTION subs( S1, S2 )
  implicit none
  TYPE (my_taylor) subs
  TYPE (my_taylor), INTENT (IN) :: S1, S2

   subs%a=S1%a - S2%a

   call clean(subs)
END FUNCTION subs
```

This is exactly Eq. (12). We encourage the reader to compile the files and run the main program.

4 DA and TPSA maps

The code for this section is located at **z_track_da_tpsa.f90**.

A code like TC always produces TPSA objects by default. To make the explanations simple, we will assume here that TC is tracking for one turn in a ring. The closed orbit in that case is a natural special orbit demanded by theory. For example, all textbooks assume some version of Hill's equation **around the ideal closed orbit** when discussing Courant-Snyder theory.

Calling this orbit f — for fixed point — we have:

$$f = TC(f) \tag{15}$$

We now, via polymorphism, track the following ray z_0:

$$z_0 = f + \Delta \tag{16}$$

Δ is an array of infinitesimals:

$$\Delta = (\Delta_1, \cdots, \Delta_{nv}) \quad \text{where} \quad \Delta_k^{no+1} = 0 \tag{17}$$

If we substitute Eq. (16) in $TC(z)$, we get:

$$TC(f + \Delta) = TC(f) + t_f(\Delta) = f + t_f(\Delta) \tag{18}$$

$t_f(\Delta)$ is a (vector) polynomial in Δ of maximum degree no and such that $t_f(0) = 0$

We can evaluate TC at a different ray not involving the closed orbit f, for example, we can use some arbitrary input in the units of TC:

$$TC(z + \Delta) = TC(z) + t_z(\Delta) = z_1 + t_z(\Delta) \tag{19}$$

In Eq. (18) and Eq. (19), the variable Δ represents a different expansion. If we want Δ to represent the same variable, we need to translate these expressions:

$$m_f(\Delta) = f + t_f(\Delta - f) \quad \text{and} \quad m_z(\Delta) = z_1 + t_z(\Delta - z) \tag{20}$$

In Eq. (20), the maps m_f and m_z are the same if and only if $no = \infty$. For example, we have:

$$m_f(f) = f = m_z(f) \; + \; O(|z - f|^{no+1}) \tag{21}$$

Eq. (21) is again exposing a symptom to TPSA maps: inconstant in the order no is finite.

Next we see how to such maps can be concatenated.

4.1 Concatenating two TPSA maps

Consider two TPSA maps:

$$m_{w_0}(z) = w_1 + M_{w_0}(z - w_0) \quad \text{and} \quad n_{z_0}(z) = z_1 + N_{z_0}(z - z_0) \tag{22}$$

For the record here, in FPP, the map m_{w_0} of Eq. (22) is stored as follows:

```
type(c_damap) m  ! Here nv=nd2 + # of parameters
.
m%x0(1:nd2)=w0     ! these are complex(dp), nd2 is the size of phase space
m%v(1:nd2) = w1(1:nd2)+Mw0(Δ(1:nv))     ! these are complex Taylor, i.e., in Berz's no Dnv
```
(23)

We compute now the map $t = m \circ n$:

$$\begin{aligned} t(x) &= (m_{w_0} \circ n_{z_0})(x) \\ &= w_1 + M_{w_0}(n_{z_0}(x) - w_0) \\ &= w_1 + M_{w_0}(N_{z_0}(x - z_0) + z_1 - w_0) \end{aligned} \tag{24}$$

This operation is done in FPP between two **c_damap**'s using the operator **.o.**

4.2 Inverse of a TPSA map in terms of a DA map

The inverse of a DA map is well known. First one inverses the linear part, a matrix, and then one inverses the nonlinear part by an iterative method that has a finite number of steps, namely no at most. This routine was programmed by Berz and is in **c_dabnew.f90**. In FPP is called as follows:

```
type(c_damap) m, m_inverse
        .
        .
m_inverse= m**(-1)
        .
        .
```

We work out the TPSA inverse of m_{w_0} of Eq. (22). To do so we solve the following equation for y in terms of x:

$$\begin{aligned} m_{w_0}(x) = w_1 + M_{w_0}(x - w_0) = y \\ \Rightarrow\ x = M_{w_0}^{-1}(y - w_1) + w_0 \\ \text{or}\ \ m_{w_0}^{-1}(x) = M_{w_0}^{-1}(x - w_1) + w_0 \end{aligned} \tag{25}$$

In FPP, the TPSA inverse is invoked by the following call:

```
type(c_damap) m, m_tpsa_inverse
        .
        .
m_tpsa_inverse= m.oo.(-1)
        .
        .
```

4.3 Concatenating two DA maps using their TPSA representation

In a "DA" situation, the exit value of the first map must be the entrance value of the second map,i.e., they must be evaluated on the same orbit. Therefore we must have $z_1 = w_0$:

$$m_{w_0}(\Delta) = w_1 + M_{w_0}(\Delta - w_0) \quad \text{and} \quad n_{z_0}(\Delta) = w_0 + N_{z_0}(\Delta - z_0) \tag{26}$$

For example the first map could be from the entrance of a wiggler to its center followed by the second half of the wiggler. Obviously, the exit value of n_{z_0} must be the entrance value of the ray of map m_{w_0}. Here we verified, that in this case, we can ignore completely these constant values. In an integrator, this happens completely naturally since we do not deal with **c_damap** but with the rays themselves in polymorphic form, i.e., type **probe_8**.

$$\begin{aligned} (m_{w_0} \circ n_{z_0})(\Delta) &= w_1 + M_{w_0}(n_{z_0}(\Delta) - w_0) \\ &= w_1 + M_{w_0}(N_{z_0}(\Delta - z_0)) \\ &= w_1 + (M_{w_0} \circ N_{z_0} \circ (I - z_0))(\Delta) \\ &= w_1 + \underbrace{(M_{w_0} \circ N_{z_0})}^{DA\ \circ}(\Delta - z_0) \end{aligned} \tag{27}$$

We see in Eq. (27), where $I - z_0$ represents a translation, that only the concatenation of the map around the orbit is necessary, i.e., $M_{w_0} \circ N_{z_0}$. This is why, in FPP, the DA concatenation does

not keep track of the constant parts at all. This is the case, for example, during all the normal form operations: in the linear case for example, all the the matrices are around the closed orbit, no feed-down effects are present and thus constant parts can be ignored in diagonalizing the matrix.

This DA operation is done in FPP between two **c_damap**'s using the operator ***.

4.3.1 Where are TPSA map and DA stored: one two Fortran type but two operations

A code like TC naturally produces TPSA maps of the form

$$TC(z) = m_{w_0}(z) = w_1 + M_{w_0}(z - w_0) \tag{28}$$

This map is produced by tracking a particle starting at w_0 and using TPSA to extract the Taylor representation denoted by M_{w_0}. However there is an infinite number of choices for w_0:

$$TPSA = \left\{ w_0 \in \mathbb{R}^6 \,|TC(z) = m_{w_0}(z) = w_1 + M_{w_0}(z - w_0) \right\}_{No=\infty} \tag{29}$$

If the order No is infinite, then all these maps are identical and anyone could be used to represent the map of the code, namely $TC(z)$. You are simply expanding the same function around a different point.

In fact their Taylor representations form a natural equivalent class. Things start to degenerate if No is finite, as it is in all realistic applications:

$$TPSA_{No} = \left\{ w_0 \in \mathbb{R}^6 \,|TC(z) \approx m_{w_0}(z) = w_1 + M_{w_0}(z - w_0) \right\}_{No<\infty} \tag{30}$$

With a finite No, there is a need to be close to the design orbit (LINAC) or closed orbit (ring). This is to avoid feed down issues. This is achieved, in a ring, by choosing the map expressed around the actual closed orbit of the code TC.

It is clear that any member of that set can be stored to order No in the type **c_damap**. Therefore we use this type for both DA and TPSA maps.

The difference is in the operator used: both are types **c_damap**. If **M** and **N** are DA maps, then the resulting DA map **K** will be:

```
K=M*N
```

If we intend **M** and **N** to be TPSA maps, then the code is

```
K=M.o.N
```

The code for **M*N** completely ignore the constant parts when performing the required substitutions. For example, in a linear map, we are just multiplying matrices.

As we will see in section (§4.7)), w_0 is stored in **c_damap%x0(:)**. It is ignored by the DA concatanation "$*$".

4.4 Finding the closed orbit of a one-turn TPSA map

In an integrator such as Bmad or PTC, we normally find the exact closed orbit f. However what if we have a TPSA map m_{w_0} which is **not** around the closed orbit? We need to solve the following equation:

$$m_{w_0}(f) = w_1 + M_{w_0}(f - w_0) = f \tag{31}$$

Eq. (31) into an equation involving a map:

$$\begin{aligned}
m_{w_0}(f) &= f \\
w_1 + M_{w_0}(f - w_0) - f &= 0 && \text{(32a)} \\
w_1 + M_{w_0}(f - w_0) - (f - w_0) - w_0 &= 0 \\
w_1 - w_0 + M_{w_0}(f - w_0) - (f - w_0) &= 0 \\
w_1 - w_0 + (M_{w_0} - I)(f - w_0) &= 0 && \text{(32b)} \\
c_{w_0}(f) &= 0 && \text{(32c)}
\end{aligned}$$

Eq. (32b) is exactly in the TPSA form of our **c_damap**, i.e., the right hand term of Eq. (20). We can solve for the fixed point f:

$$f \quad = c_{w_0}^{-1}(0) \tag{33}$$

Once again, the value of f will depend on the order of truncation *no* while the fixed point of the integrator is "exact".

4.5 Computing the DA map from the TPSA map: inconsistent

Consider the map of Eq. (31)

$$m_{w_0}(z) = w_1 + M_{w_0}(z - w_0) \tag{34}$$

and its fixed point given by Eq. (33). We can re-expressed around the fixed point f. It is done by a strange similarity transformation. Consider this representation of the identity map:

$$a_f(z) = f + I(z - f) \tag{35}$$

Eq. (35) is the identity "expressed" around f. The two f's obviously cancel. We also notice, not too surprisingly, following Eq. (25), we have

$$a_f^{-1}(z) = a_f(z) \tag{36}$$

We are now ready to make the following similarity transformation on m_{w_0}. To do so, we keep the identity explicitly in Eq. (35) and apply twice the concatenation formula of Eq. (24). It is just

a matter of substituting carefully variables:

$$\left(a_f \circ m_{w_0} \circ a_f\right)(\Delta) = w_1 + M_{w_0}(I(\Delta - f) - w_0 + f) \tag{37a}$$

$$= \left\{ \underbrace{(w_1 + M_{w_0} \circ (I - w_0 + f))}_{\text{DA map around } f} \circ \underbrace{(I - f)}_{\text{shift}} \right\}(\Delta) \tag{37b}$$

In Eq. (37a)[6], if we use the definition of I, clearly all f's disappear and we are back where we started. However we use the definition of a **c_damap** and extract $-f$ from the map and use a shift. This is consistent with the definition shown in blue in Eq. (23).

This last formula allows for normal form on TPSA maps. Of course, if $no/ = \infty$ inconsistencies will appear. For example otherwise symplectic maps will become non-symplectic or, worse, radiation effects will be wiped out. This is why we love integrators with DA maps around the true closed orbit.

We now illustrates all of this in a one dimensional example and also with a real map from a ring.

Finally we substitute $\Delta = 0$ is the left factor (DA map) of Eq. (37b) and get

$$w_1 + M_{w_0}(-w_0 + f)$$

which is, according to Eq. (32a), precisely the fixed orbit if f is selected according to Eq. (32c).

4.6 Concatenating two "DA" maps from the "code" of Eq. (38)

We provide an actual one-dimensional example using AP. This is a map in one dimension which is the result of the concatenation of two maps. It imitates a ring made of two parts.

$$\begin{aligned} m_1(x) &= 0.05 + \sin(0.5x) + 0.3\sin^2(x) \\ m_2(x) &= 0.03 + \sin(0.3x) + 0.2\sin^2(x) \\ m_{12} &= m_2 \circ m_1 \end{aligned} \tag{38}$$

As we can see, $m(0) \neq 0$: therefore the numbers 0.05 and 0.03 represent a deliberate weird placement of a magnet or the inevitable misalignments found in every beam line. Suppose we want the linear properties of this map, in a code like PTC, it is recommended to first solve for the closed orbit,

$$m_{12}(x_0) = x_0 \;\rightarrow\; x_0 = 0.05469119581164052\cdots \tag{39}$$

and then we expand the one-turn map around this closed orbit:

$$m_{12}(x_0 + \Delta) = x_0 + m_{12;1}\Delta + m_{12;2}\Delta^2 + \cdots \tag{40}$$

[6] We avoided abuses of notation in Eq. (37a). Some people may want to confuse the identity I with the dummy variable Δ if that helps.

The actual "DA" map is made of the coefficients $m_{12;i}$ of Eq. (40). We can compute this map to second order which is very common in ring dynamics: the linear part gives the usual lattice functions and the second order part gives us the sextupoles distortions and/or chromaticities in a real ring. Here is the code fragment:

```
type(c_taylor)  x   ! In PTC, x would be a probe_8

write(6,*) "Imitating PTC: tracking through  "

x=x_closed+(1.d0.cmono.1)
x=sin(x/2.d0)+0.3d0*sin(x)**2+0.05d0
x_closed1=x     ! recording orbit for future use
x=sin(0.3d0*x)+0.2d0*sin(x)**2+0.03d0

call print(x)
```

The result, for **x**, is:

```
 Imitating PTC: tracking through

          1, NO =    2, NV =    1, INA =    9
 *******************************************

    I   COEFFICIENT           ORDER   EXPONENTS
      NO =     2      NV =     1
   0   0.5469119581164052E-01   0.000000000000000        0
   1   0.1763235586477631       0.000000000000000        1
   2   0.1533323662801814       0.000000000000000        2
    -3   0.000000000000000        0.000000000000000        0
```

We see that the real part of the Taylor series is the closed orbit indeed.

The code fragment is:

```
! computing map1
x=x_closed+(1.d0.cmono.1)
x=sin(x/2.d0)+0.3d0*sin(x)**2+0.05d0
map1%v(1)=x
! computing map2
x=x_closed1+(1.d0.cmono.1)
x=sin(0.3d0*x)+0.2d0*sin(x)**2+0.03d0
map2%v(1)=x
one_turn_map_AP = map2*map1
Write(6,*) " This is map1 "
call print(map1)
Write(6,*) " This is map2 "
call print(map2)
write(6,*) "one_turn_map_AP = map2*map1 "
call print(one_turn_map_AP)
!!! Saving DA coefficients to imitate a TRANSPORT-like code
m(1,1)=map1%v(1).sub.'1'
m(1,2)=map1%v(1).sub.'2'
```

```
m(2,1)=map2%v(1).sub.'1'
m(2,2)=map2%v(1).sub.'2'
write(6,*) "Coefficients using Taylor Map Multiplication "
write(6,*) m(1,1)*m(2,1), m(2,1)*m(1,2)+m(2,2)*m(1,1)**2
```

and the result is

```
 Creating two DA maps : map1 and map2
  This is map1
  tpsa status for tracking type(c_ray)  F
           1  Dimensional DA map (around chosen orbit in map%x0)

          1, NO =    2, NV =    1, INA =   10
*****************************************

   I  COEFFICIENT           ORDER   EXPONENTS
     NO =     2      NV =     1
  0  0.7823863368603357E-01   0.000000000000000       0
  1  0.5325623875161611       0.000000000000000       1
  2  0.2947893387720240       0.000000000000000       2
   -3   0.000000000000000       0.000000000000000       0
  No Spin Matrix
  No c_quaternion
No Stochastic Radiation
  This is map2
  tpsa status for tracking type(c_ray)  F
           1  Dimensional DA map (around chosen orbit in map%x0)

          1, NO =    2, NV =    1, INA =   24
*****************************************

   I  COEFFICIENT           ORDER   EXPONENTS
     NO =     2      NV =     1
  0  0.5469119581164052E-01   0.000000000000000       0
  1  0.3310852639633932       0.000000000000000       1
  2  0.1965003538431843       0.000000000000000       2
   -3   0.000000000000000       0.000000000000000       0
  No Spin Matrix
  No c_quaternion
No Stochastic Radiation
one_turn_map_AP = map2*map1
  tpsa status for tracking type(c_ray)  F
           1  Dimensional DA map (around chosen orbit in map%x0)

          1, NO =    2, NV =    1, INA =   66
*********************************************

   I  COEFFICIENT           ORDER   EXPONENTS
     NO =     2      NV =     1
  0  0.5469119581164052E-01   0.000000000000000       0
```

```
   1   0.1763235586477631        0.000000000000000        1
   2   0.1533323662801814        0.000000000000000        2
    -3    0.000000000000000         0.000000000000000         0
 No Spin Matrix
 No c_quaternion
No Stochastic Radiation
Coefficients using Taylor Map Multiplication
 0.176323558647763       0.153332366280181
```

Since the maps **map1** and **map2** are around the closed orbit, we expect them to obey the usual rules of DA concatenation which ignore the constant parts. Thus, given $m_{12} = m_2 \circ m_1$,

$$\text{if} \quad m_1(x_0 + \Delta) = x_1 + m_{1;1}\Delta + m_{1;2}\Delta^2 + \cdots \tag{41a}$$

$$\text{and } m_2(x_0 + \Delta) = x_2 + m_{2;1}\Delta + m_{2;2}\Delta^2 + \cdots \tag{41b}$$

$$\text{then } m_{12}(x_0 + \Delta) = x_0 + m_{12;1}\Delta + m_{12;2}\Delta^2 + \tag{41c}$$

$$\text{where } m_{12;1} = m_{1;1}m_{2;1} \text{ and } m_{12;1} = m_{2;1}m_{1;2} + m_{2;2}m_{1;1}^2 \tag{41d}$$

We see perfect agreements in the numerical results. Now we will turn this into a TPSA concatenation problem.

4.7 Concatenating the two "TPSA" maps from the "code" of Eq. (38)

Let us compute the one-turn map by concatenating the two maps map1 and map2 around, respectively, the weird orbits $x1 = 0.015$ and $x2 = 0.02$. As we explained before these are the same as the DA maps of section (§4.6) is $no = \infty$ but here $no = 2$.

The map is computed using the code fragment

```
!  Creating TPSA maps around the design orbit

x1= 0.015d0
map1%x0(1)=x1
x=map1%x0(1)+(1.d0.cmono.1)
map1%v(1)=sin(x/2.d0)+0.3d0*sin(x)**2+0.05d0

x2=.02d0
map2%x0(1)=x2
x=map2%x0(1)+(1.d0.cmono.1)
map2%v(1)=sin(0.3d0*x)+0.2d0*sin(x)**2+0.03d0

!!! multiplying the TPSA maps
 map12=map2.o.map1


x=map12%v(1)+i_*one_turn_map_AP%v(1)
 write(6,*)  "    The map map12 for one turn"
 call print(map12)
 write(6,*)  "    Comparing the TPSA one-turn map with the DA one-turn map"
 write(6,*)  "    TPSA coefficients      DA coefficient ",k
 call print(x)
```

The results are:

```
     The map map12 for one turn
  tpsa status for tracking type(c_ray)  F
           1  Dimensional DA map (around chosen orbit in map%x0)
 Initial orbit for TPSA calculations
 (1.500000000000000E-002,0.000000000000000E+000)

          1, NO =    2, NV =    1, INA =   38
 *******************************************

    I  COEFFICIENT           ORDER   EXPONENTS
      NO =     2      NV =     1
   0  0.4793209256475234E-01   0.000000000000000        0
   1  0.1643954713972499       0.000000000000000        1
   2  0.1482514114953167       0.000000000000000        2
    -3   0.000000000000000        0.000000000000000        0
  No Spin Matrix
  No c_quaternion
 No Stochastic Radiation
     Comparing the TPSA one-turn map with the DA one-turn map
     TPSA coefficients      DA coefficient            0

          1, NO =    2, NV =    1, INA =    9
 *******************************************

    I  COEFFICIENT           ORDER   EXPONENTS
      NO =     2      NV =     1
   0  0.4793209256475234E-01  0.5469119581164052E-01   0
   1  0.1643954713972499      0.1763235586477631       1
   2  0.1482514114953167      0.1533323662801814       2
    -3   0.000000000000000        0.000000000000000        0
```

The two maps, map12 and one_turn_map_AP, are identical but expressed around different orbits. The map one_turn_map_AP is expressed around the closed orbit. We will use the results of section (§4.4) to move map12 to its fixed point. This is done with the code fragment:

```
type(c_ray) rayon
      .
      .
      .
! TPSA maps re-expressed around the closed orbit !

write(6,*) " closed orbit computation "
write(6,*) map12%x0(1)
c_w0%x0=map12%x0
c_w0%v(1)=map12%v(1)-(1.d0.cmono.1)-map12%x0(1)
c_w0_inv=c_w0.oo.(-1)

f_map%x0=0
f_map%v(1)=0.d0
```

```
f_map=c_w0_inv.o.f_map
f_ray=0
f_ray=c_w0_inv.o.f_ray
call print(c_w0_inv)
call print(f_map)
write(6,*) "exact ", x_closed
x_closed_tpsa=map1%v(1)
write(6,*) "TPSA  ", x_closed_tpsa
write(6,*) "f_ray ", f_ray%x(1)

go_to_orbit%v(1)= (1.d0.cmono.1)+x_closed_tpsa
go_to_orbit%x0=x_closed_tpsa

map12=(go_to_orbit.o.map12).o.(go_to_orbit.oo.(-1))

call print(map12)

x=map12%v(1)+i_*one_turn_map_AP%v(1)
 write(6,*)  "    Maps around the TPSA closed orbit "
 write(6,*)  "    Comparing TPSA one-turn map around the TPSA fixed point "
write(6,*)  "    with the DA one-turn map"
 write(6,*)  "    TPSA coefficients      DA coefficient "

 call print(x)
```

The answer for the TPSA map around the approximately computed closed orbit is:

```
      Maps around the TPSA closed orbit
     Comparing TPSA one-turn map around the TPSA fixed point
     with the DA one-turn map
     TPSA coefficients      DA coefficient

          1, NO =    2, NV =    1, INA =    9
 ******************************************

    I  COEFFICIENT           ORDER   EXPONENTS
      NO =     2      NV =     1
   0  0.5469069935363461E-01  0.5469119581164052E-01   0
   1  0.1761640230032753       0.1763235586477631       1
   2  0.1482514114953167       0.1533323662801814       2
    -3   0.000000000000000        0.000000000000000       0
```

We see that the agreement has improved. Indeed if $no = \infty$, the agreement would be perfect. This is why a code like COSY-INFINITY usually runs at a high order. If we rerun this example with $no = 10$, we get:

```
     Maps around the TPSA closed orbit
     Comparing TPSA one-turn map around the TPSA fixed point
     with the DA one-turn map
     TPSA coefficients      DA coefficient
```

```
          1, NO =    10, NV =     1, INA =    17
*****************************************

   I   COEFFICIENT           ORDER    EXPONENTS
      NO =     10       NV =      1
  0   0.5469119581164050E-01  0.5469119581164052E-01    0
  1   0.1763235586477634       0.1763235586477631       1
  2   0.1533323662802094       0.1533323662801814       2
  3   0.4375700106665827E-01  0.4375700106455089E-01    3
  4  -0.3637622533509953E-01 -0.3637622544112402E-01    4
  5  -0.3834044616082886E-01 -0.3834044989384196E-01    5
  6  -0.1063392477923387E-01 -0.1063401860744071E-01    6
  7   0.1162532131698485E-01  0.1162363855314815E-01    7
  8   0.1090970864911106E-01  0.1088862478020678E-01    8
  9   0.2833241323816774E-03  0.1079668683861611E-03    9
 10  -0.3309462539739653E-02 -0.4174183972224262E-02   10
```

We can see that we are closing on the DA map section (§4.6).

In a kick code or integrator, we can compute maps at a low order while preserving perfect self-consistency provided all the maps are computed around the closed orbit in a ring. Therefore, in an integrator, if you like your model, you are assured of self-consistency.

One additional issue is "symplecticity" which cannot be analyzed with a fake map like that of Eq. (38). This can become a severe issue especially if radiation is turned on: the TPSA maps lack of consistency can be greater than radiation damping, thus erasing a very important effect. This does not happen in a (symplectic) integrator since the maps are always self-consistent.

5 PTC examples of DA vs TPSA maps

The code for this section is located at **z_track_map_code.f90**. If you run the code, you can select case 1,2,3 or 4 which correspond respectively to sections (§5.1), (§5.2), (§5.3) and (§5.4).

5.1 Extracting Taylor orbital maps around the closed orbit: case 1

Now we look at a "PTC" piece of code which mixes, in a trivial but very fundamental way, the analysis part (AP) with the tracking part (TC) of PTC with the specific intent of creating a *bona fide* Taylor series map which approximates the beam line in PTC, i.e., the output of the subroutine **propagate**.

Consider:

```
p=>ring%start
closed_orbit=0
call find_orbit_x(closed_orbit,bmad_state, 1.0e-7_dp, fibre1=p)
write(6,*) " closed orbit "
write(6,"(6(1x,g11.4))") closed_orbit
```

The above code is pure tracking of real numbers. The closed orbit is computed at the start of the ring. The **find_orbit_x** routines finds the closed orbit and returns :

```
  closed orbit
 -0.1788E-03 -0.9790E-05 -0.1328E-04 -0.6548E-04 -0.1866E-05 -0.5154E-01
```

The object **Bmad_state**, of type **internal_state**, is a PTC structure where the particulars of Bmad are fed for compatibility : units of Bmad, phase space dimension of 6 for **c_damap**'s, etc...

internal_state's are objects which control the initialization interface between TC and AP when *bona fide* Taylor maps are needed for analysis. They do not need to exist and could be left to the user if the interaction between TC and AP was a once-in-a-blue-moon event.

The new code fragment is interesting:

```
select case(case_section)
case(1)
p=>ring%start
call move_to(ring,p,"QF1")   !  Locating QF1 in TC
call make_it_knob(p%magp%bn(1),1)  !  PP : making a Polymorph into a knob
call move_to(ring,p,"QF2")   !  Locating QF2 in TC
call make_it_knob(p%magp%an(1),2)  !  PP : making a Polymorph into a knob

p=>ring%start
closed_orbit=0
call find_orbit_x(closed_orbit,bmad_state, 1.0e-7_dp, fibre1=p)
write(6,*) " closed orbit "
write(6,"(6(1x,g11.4))") closed_orbit

no=1 ; np=2 ;  ! nv=6+2=8
call init(bmad_state,no,np)   ! Berz's TPSA package is initialized
```

The TPSA variables will be of order no=1 and the phase space dimension will be six via **Bmad_state**. Additionally, via np=2, the polynomials will have two extra variables—knob 1 and 2. Therefore the total number of variables will be 8. Then it is decided that the dipole components of the first QF1 and QF2 will be the 7th (6+1) and 8th (6+2) variables representing respectively the normal dipole component of QF1 and the skew component of QF2. These things are called knobs and can turned off at will. If they are active, then the dipoles components of these quadrupoles are TPSA variables otherwise they stay **real(8)** variables.

Now we mix TC and AP:

```
type(probe) ray_TC
type(probe_8) ray_8_TC
type(c_damap)  one_turn_map_AP, identity_AP,two_turn_map_AP
real(dp)  closed_orbit(6)
type(internal_state),target :: Bmad_state
   .
   .
   .

ray_TC=closed_orbit   !  For TC
```

```
identity_AP=1  !  For AP

ray_8_TC=ray_TC+identity_AP   ! Connect AP with TC
write(6,*) " The initial x "
call print(ray_8_TC%x(1))
call propagate(ray_8_TC,+bmad_state,fibre1=p) ! TC of PTC proper

one_turn_map_AP=ray_8_TC ! TC into AP : makes a map out of ray_8_TC

write(6,*) "x_final for one turn "
call print(one_turn_map_AP%v(1))

two_turn_map_AP=one_turn_map_AP*one_turn_map_AP
write(6,*) "x_final for two turns : squaring the map "
call print(two_turn_map_AP%v(1))

!  Tracking two turns using TC
ray_8_TC=ray_TC+identity_AP   ! Connect AP with TC
call propagate(ray_8_TC,+bmad_state,fibre1=p) ! TC of PTC proper
call propagate(ray_8_TC,+bmad_state,fibre1=p) ! TC of PTC proper
two_turn_map_AP=ray_8_TC ! TC into AP : makes a map out of ray_8_TC
write(6,*) "x_final for two turns : tracking two turns "
call print(two_turn_map_AP%v(1))
```

identity_AP is an AP object of type **c_damap**. It is set to a phase space identity by equating it to 1. Indeed in any code which tracks Taylor maps (COSY-INFINITY, TRANSPORT, MARYLIE, etc…) the initial value of a map must be the identity in 6 dimensions. In the linear case, it amounts to the identity matrix in the 6 orbital variables of Bmad. The first variable is the monomial z_1 representing x, the second variable is z_2 will represent p_x and so on, i.e., the variables of the Bmad phase space. (See Bmad manual)

The next line is crucial: it mixes AP and TC. **ray_8_TC** is a **probe_8**: we feed in it the closed orbit buried in ray_TC which is made of **real(8)** and add to this the identity map. This operation reveals Forest intent: generally make useful Taylor series phase space maps from TC. It could have been left to the user. Indeed if a user links TC to a library which finds the linear eigenvalues of a matrix, neither Forest nor the authors of that library would provide the interface: it must be written by the user based on the nature of these matrices. Here the object **Bmad_state**, of type **internal_state**, contains all that is needed to make of TC and AP consistent packages.

So now the polymorphic probe **ray_8_TC** contains a polymorph consistent with the identity map expressed around the closed orbit. The code prints this polynomial:

```
  The initial x

 etall    1, NO =    1, NV =    8, INA =   29
 ****************************************

    I  COEFFICIENT           ORDER   EXPONENTS
      NO =     1      NV =     8
   0 -0.1787724612820044E-03   0  0  0  0  0  0  0  0
```

```
   1    1.000000000000000        1  0  0  0  0  0  0  0
    -2    0.000000000000000          0  0  0  0  0  0  0  0
```

We recognize the closed orbit value of x and the monomial $1.0 * z_1^1$.

Therefore the tracking will take place correctly around the closed orbit and the feed down effects will be correctly computed. This is to be contrasted with *bona fide* Taylor codes (COSY-INFINITY, TRANSPORT, MARYLIE, etc...) where feed-down effects are not self-consistent[7]. Here they are totally self-consistent with the code.

The next is TC proper: **propagate(ray_8_TC,+Bmad_state,fibre1=p)** tracks through the total beam line— actually one turn in this example. The + sign in front of **Bmad_state** activates the "knobs" connected to QF1 and QF2. In TC, knobs are deactivated by default. This is not true of PP proper.

The next line **one_turn_map_AP=ray_8_TC** sends the final TC structure **ray_8_TC** into a **c_damap**: it connects TC to AP. Finally, we attempt a pure AP operation

```
two_turn_map_AP=one_turn_map_AP*one_turn_map_AP
```

This operation is the concatenation of two "differential algebraic" maps: the constant part is ignored.

This is followed by a two turns tracking of TC. The results are identical to machine precision. This is the virtue of using "DA" maps around the closed orbit in an "kick code" or integrator like TC. We print the results for the skeptics: notice machine precision agreement—only the last two digits are different.

```
x_final for one turn
          1, NO =     1, NV =     8, INA =   230
 *****************************************

    I   COEFFICIENT           ORDER    EXPONENTS
       NO =      1       NV =      8
   0  -0.1787724612819064E-03    0.000000000000000        0  0  0  0  0  0  0  0
   1  -0.3889828486539758        0.000000000000000        1  0  0  0  0  0  0  0
   1    11.40736328972829        0.000000000000000        0  1  0  0  0  0  0  0
   1  -0.3464662557908923E-02    0.000000000000000        0  0  1  0  0  0  0  0
   1  -0.4171271427570414E-01    0.000000000000000        0  0  0  1  0  0  0  0
   1   0.6963086125527715E-10    0.000000000000000        0  0  0  0  1  0  0  0
   1   0.8036598951947091E-01    0.000000000000000        0  0  0  0  0  1  0  0
   1   -4.224142793166476        0.000000000000000        0  0  0  0  0  0  1  0
   1  -0.3125492004102120E-05    0.000000000000000        0  0  0  0  0  0  0  1
    -9    0.000000000000000        0.000000000000000        0  0  0  0  0  0  0  0
 x_final for two turns : squaring the map

          1, NO =     1, NV =     8, INA =   179
 *****************************************

    I   COEFFICIENT           ORDER    EXPONENTS
       NO =      1       NV =      8
   0  -0.1787724612819064E-03    0.000000000000000        0  0  0  0  0  0  0  0
```

[7] COSY-INFINITY is often more accurate than PTC but is not self-consistent with its own model. In rings, this can be a problem. In a single pass system, it might be better to have accurate inconsistent models with complex fringe effects, then simplistic integrators like TC.

```
   1  -0.7056565190107400        0.000000000000000        1   0   0   0   0   0   0   0
   1   -8.751690962380659        0.000000000000000        0   1   0   0   0   0   0   0
   1  -0.3542316696944531E-01    0.000000000000000        0   0   1   0   0   0   0   0
   1   0.2562552716914303E-02    0.000000000000000        0   0   0   1   0   0   0   0
   1   0.5610356685840276        0.000000000000000        0   0   0   0   1   0   0   0
   1   0.8726398593300869E-01    0.000000000000000        0   0   0   0   0   1   0   0
   1   -2.218998803166924        0.000000000000000        0   0   0   0   0   0   1   0
   1  -0.1965421389017537E-01    0.000000000000000        0   0   0   0   0   0   0   1
    -9    0.000000000000000         0.000000000000000         0   0   0   0   0   0   0   0
 x_final for two turns : tracking two turns

          1, NO =    1, NV =    8, INA =  179
 ****************************************

    I   COEFFICIENT           ORDER    EXPONENTS
       NO =     1       NV =     8
   0  -0.1787724612818773E-03    0.000000000000000        0   0   0   0   0   0   0   0
   1  -0.7056565190107397        0.000000000000000        1   0   0   0   0   0   0   0
   1   -8.751690962380632        0.000000000000000        0   1   0   0   0   0   0   0
   1  -0.3542316696944601E-01    0.000000000000000        0   0   1   0   0   0   0   0
   1   0.2562552716913452E-02    0.000000000000000        0   0   0   1   0   0   0   0
   1   0.5610356685840368        0.000000000000000        0   0   0   0   1   0   0   0
   1   0.8726398593301046E-01    0.000000000000000        0   0   0   0   0   1   0   0
   1   -2.218998803166917        0.000000000000000        0   0   0   0   0   0   1   0
   1  -0.1965421389017552E-01    0.000000000000000        0   0   0   0   0   0   0   1
    -9    0.000000000000000         0.000000000000000         0   0   0   0   0   0   0   0
```

5.2 Extracting Taylor Series which are not orbital maps: case 2

Nota Bene : In this section, we call FPP outside the constructs of PTC, namely without the use of **bmad_state**. *Of course, we could still use* **bmad_state** *and thus produce a phase space map with null transverse entry. In this case we waste, in Bmad, 6 TPSA variables. This might be a memory/speed issue if a large number of variables or a higher order are required. Otherwise it is perfectly fine. See section (§5.4) to see how the equivalent calculation is done when running with Bmad variables as Taylor series.*

The function **propagate** of TC computes a final position $z^f = (z_1^f, z_2^f, z_3^f, z_4^f)$ in terms of an initial position $z^0 = (z_1^0, z_2^0, z_3^0, z_4^0)$.

Furthermore, we can choose 6 magnets to have TPSA parameters. Namely 3 of them will be normal dipole kicks and 3 of them will be skew dipole kicks.

```
p=>ring%start
call move_to(ring,p,"QF1")    !  Locating QF1 in TC
call make_it_knob(p%magp%bn(1),1)   !  PP : making a Polymorph into a knob
call move_to(ring,p,"QF2")   !  Locating QF2 in TC
call make_it_knob(p%magp%an(1),2)   !  PP : making a Polymorph into a knob
call move_to(ring,p,"QD1")      !  Locating QD1 in TC
call make_it_knob(p%magp%bn(1),3)   !  PP : making a Polymorph into a knob
call move_to(ring,p,"QD2")      !  Locating QD2 in TC
call make_it_knob(p%magp%an(1),4)   !  PP : making a Polymorph into a knob
call move_to(ring,p,"BEND")      !  Locating BEND in TC
call make_it_knob(p%magp%bn(1),5)   !  PP : making a Polymorph into a knob
```

```
call move_to(ring,p,"BEND1")     !  Locating BEND1 in TC
call make_it_knob(p%magp%an(1),6)  !  PP : making a Polymorph into a knob
```

Even if z^0 starts as **real(8)**, by virtue of polymorphism, it will become a vector of Taylor series if **propagate** visits any of the above magnets..

Here is the code computing z^f. Notice that in this example we start with the entrance orbit $z^0 = 0$. We will try to fit the output z^f also to 0. None of this will depend on the meaning of TC. It could compute the values of stocks for all that we know.

```
bmad_state=bmad_state+nocavity0 ! turns cavities into drifts

p=>ring%start
closed_orbit=0
call find_orbit_x(closed_orbit,bmad_state, 1.0e-7_dp, fibre1=p)
write(6,*) " closed orbit "
write(6,"(6(1x,g11.4))") closed_orbit

np=6     ! number of free parameters to solve these equations
no=1; nV=NP ;  ! nv=6
call c_init_all(no,nv)   ! FPP command to initialize the TPSA

call alloc(ct);call alloc(ray_8_TC);

ray_TC=entrance_orbit   !  For TC
ray_8_TC=ray_TC  ! Connect AP with TC: no identity added
write(6,*) "The variable x = z_1^0 "
call print(ray_8_TC%x(1))
p=>ring%start
call propagate(ray_8_TC,+bmad_state,fibre1=p) !  TC of PTC proper
call print(ray_8_TC%x(1))
```

The important difference is the line **ray_8_TC=ray_TC**: the initial orbit made of 6 **real(8)** numbers is simply put into the polymorphic **probe_8** structure **ray_8_TC**. At this stage, there are no Taylor series in sight. Indeed the code print the initial value of x and it is the simply a **real(8)** unlike the example of section (§5.1):

```
 The variable x = z_1^0
  0.000000000000000E+000
```

However, TC will encounters some Taylor series during the tracking, namely the dipole components of the magnets QF1, QF2, QD1, QD2, BEND and BEND1. Thus, via polymorphism, a Taylor series in these components will appear as the 1st to 6th variables of the TPSA package.

Additionally, the command to initialize TPSA, **call c_init_all(no,nv)**, is pure FPP and is unware of the analysis package (symplectic maps) or of the code PTC. Indeed while **propagate** is a PTC command, we could have used any code computing anything for the example of this section and section (§5.3).

The output for the variable x = z_1 is

```
The variable x = z_1^f

etall     1, NO =     1, NV =    10, INA =    44
*********************************************

   I   COEFFICIENT            ORDER    EXPONENTS
      NO =      1       NV =     10
  0   0.6109701219350949E-02    0   0   0   0   0   0   0   0   0   0
  1    -4.962724533009230       1   0   0   0   0   0   0   0   0   0
  1  -0.2968892804328157E-05    0   1   0   0   0   0   0   0   0   0
  1    -1.421420498227908       0   0   1   0   0   0   0   0   0   0
  1  -0.1231462510798852E-05    0   0   0   1   0   0   0   0   0   0
  1    -1.116448300584394       0   0   0   0   1   0   0   0   0   0
  1  -0.4314330647295780E-04    0   0   0   0   0   1   0   0   0   0
   -7    0.000000000000000        0   0   0   0   0   0   0   0   0   0
```

Naive attempts to create a Taylor map using the standard interface between TC and AP, in this example, will lead to nonsense. Instead we will use the resulting polymorphic **probe_8** to create a kind of closed orbit bump by fitting the final z^f to z^0:

This is the topic of the next section using the example of this section.

5.3 Fitting the output of section (§5.2): case 3

In section (§5.2), the Taylor series of the output **probe_8** called **ray_8_TC** have 6 variables for the dipole strengths. Consider the following functions

$$G = \frac{1}{2}\sum_{i=1,6} k_i^2 + \sum_{i=1,4} \lambda_i \left(z_i^f(\mathbf{k}) - \mathbf{g}_i \right) \tag{42}$$

The vector $\mathbf{v} = (k_1, k_2, k_3, k_4, k_5, k_6, \lambda_1, \lambda_2, \lambda_3, \lambda_4)$ contains the 6 dipole strengths and 4 Lagrange multipliers. The first term in Eq. (42) is the function we will minimize namely the norm of the total dipole strengths. The second term represents the equations we want to set equal to a goal function namely $\mathbf{g}$. In our example, they are the values of x, p_x, y, and p_y.

The solutions are found by solving the following set of equations:

$$lagrange\_map\_ap(\mathbf{v}) = \nabla_v G = 0 \tag{43}$$

lagrange_map_ap is a map in 10 TPSA variables. We can solve with by TPSA inversion since the constant part is significant:

$$\mathbf{v} = lagrange\_map\_ap^{-1}(0) \tag{44}$$

If the order of the TPSA was infinite, then we would get an exact solution. Otherwise this is simply the first step of a Newton search. We show the actual algorithm and the results:

```
bmad_state=bmad_state+nocavity0 ! turns cavities into drifts

p=>ring%start
closed_orbit=0
call find_orbit_x(closed_orbit,bmad_state, 1.0e-7_dp, fibre1=p)
write(6,*) " closed orbit "
write(6,"(6(1x,g11.4))") closed_orbit

 goal=0.d0
entrance_orbit=0
goal(1:6)=entrance_orbit

do k=1,4    ! performing 4 iterations

   !!!!!! Part 1 : calling TC !!!!!!
   neq=4    ! number of equations to solve
   np=6     ! number of free parameters to solve these equations
   no=1; nV=NP ;  ! nv=6
   call c_init_all(no,nv)   ! FPP command to initialize the TPSA

   call alloc(ct);call alloc(ray_8_TC);

   ray_TC=entrance_orbit   !  For TC
   ray_8_TC=ray_TC  ! Connect AP with TC: no identity added
   !write(6,*) "The variable x = z_1^0 "
   !call print(ray_8_TC%x(1))
   p=>ring%start
   call propagate(ray_8_TC,+bmad_state,fibre1=p) !  TC of PTC proper
   !call print(ray_8_TC%x(1))

   !!!!!! Part 2 : saving the relevant Taylor series  !!!!!!

   call ALLOC(tpos,1,NV,0) ! c_universal_taylor
   do i=1,4
    ct=ray_8_TC%x(i)%t   ! saving final transverse positions and momenta
    tpos(i)=ct
   enddo
   call kill(ct);call kill(ray_8_TC);

!!!!!!!! Part 3 : creating a map of some sort and using it !!!!!!


   nV=NP+neq ;  ! nv=6+4=10
   call c_init_all(no,nv)   ! FPP command to initialize the TPSA
   call alloc(ct);call alloc(eq);

   lagrange_map_ap%n=nv
   call alloc(lagrange_map_ap)

   do i=1,4
```

```
 eq(i)=tpos(i)   ! putting the c_universal_taylor into c_taylor
enddo

!!!  Construction of Lagrange function
do i=1,np
 lagrange_map_ap%v(i)=1.d0.cmono.i
 do j=np+1,np+neq

  ct=((eq(j-np).d.i)*(1.d0.cmono.j))
  lagrange_map_ap%v(i)=lagrange_map_ap%v(i)+ct
 enddo
enddo

do i=np+1,np+neq
 lagrange_map_ap%v(i)=eq(i-np)-goal(i-np)
enddo

lagrange_map_ap=lagrange_map_ap.oo.(-1)

f_ray=0
f_ray=lagrange_map_ap.o.f_ray

p=>ring%start
call move_to(ring,p,"QF1")   !  Locating QF1 in TC
 k1=f_ray%x(1)
 call add(p,1,1,k1)
call move_to(ring,p,"QF2")
 k1=f_ray%x(2)
 call add(p,-1,1,k1)
call move_to(ring,p,"QD1")   !  Locating QF1 in TC
 k1=f_ray%x(3)
 call add(p,1,1,k1)
call move_to(ring,p,"QD2")   !  Locating QF2 in TC
 k1=f_ray%x(4)
 call add(p,-1,1,k1)
call move_to(ring,p,"BEND")   !  Locating QF2 in TC
 k1=f_ray%x(5)
 call add(p,1,1,k1)
call move_to(ring,p,"BEND1")   !  Locating QF2 in TC
 k1=f_ray%x(6)
 call add(p,-1,1,k1)

p=>ring%start

ray_TC=entrance_orbit   !  For TC

call propagate(ray_TC,bmad_state,fibre1=p) !  TC of PTC proper

write(6,*) " exit orbit "
write(6,"(4(1x,g11.4))") ray_TC%x(1:4)
```

```
   call kill(lagrange_map_ap);call kill(ct);call kill(eq);call kill(tpos);

enddo
```

This code fragment has two parts:

1. In the first part, TC is called with the FPP initialization **call c_init_all(no,nv)** with **nv=6**.
2. In the second part, some Taylor series are saved for future used in 4 **c_universal_taylor**'s. These are the Taylor series for the transverse phase space as a function of 6 dipole strengths.
3. A map is created to fit the final positions and momenta reconstructed from the 4 **c_universal_taylor**'s. FPP is initialized with with **nv=10**.

First we are taking the derivative of the output of the orbital ray. Secondly we multiply is by **(1.d0.mono.j)** which represents the variables $v_{7,8,9,10}$, i.e., the Lagrange multipliers. There is no chance in hell that this could have been guessed by the programmers of either PTC or Bmad. This map is truly a construction of the user and is not the output of PTC. It is created from the output of PTC the same way a matrix can be computed from the output of an ordinary **real(8)** code via numerical differentiation. What for? Up to the user.

Here is the result of 4 iterations with no=1:

```
  exit orbit
  0.5673E-03  0.4982E-04 -0.1294E-06  0.4597E-08
  exit orbit
  0.1903E-04 -0.2253E-07 -0.1233E-09  0.1115E-08
  exit orbit
  0.6044E-09  0.5405E-09 -0.1040E-11 -0.1279E-12
  exit orbit
  0.1130E-14 -0.3989E-16  0.1440E-19  0.5051E-19
```

With no=5, the convergence takes two iterations:

```
  exit orbit
 -0.1216E-06  0.9864E-08  0.1979E-09  0.6521E-10
  exit orbit
 -0.1139E-15  0.1102E-16 -0.2711E-19 -0.1347E-19
```

In conclusion, the subroutine **propagate** returns Taylor series which have no apparent special meaning. The user creates in this case a map which the author of PTC, Etienne Forest, could never have anticipated. For this reason, the creation of this Lagrangian-Newton map is entirely the responsibility of the user.

On the other hand, in section (§5.1), the code TC was initialized with the construct

```
identity_AP=1  !  For AP
ray_8_TC=ray_TC+identity_AP   ! Connect AP with TC
```

This construction is provided by PTC under the assumption that Taylor maps produced by PTC are the usual phase space maps. The **probe_8** is constructed by adding phase space identity to the closed orbit or entrance orbit. In that sense, PTC tracks phase space maps because of the default interface between TC and FPP.

In fact, as we pointed out, not only the identity can be added but any canonical transformation which represent the shape of a beam or lattice functions— linear, nonlinear, spin, etc...

But it is important to remember that any change of variables within PTC involves elements of Berz's TPSA algebra ${}_{no}D_{nv}$ and manipulations which are identical to the manipulations of **real(8)**. There are no Taylor maps, no **c_damap**.

5.4 Fitting the orbital ray and the tunes: case 4

This example is a more in tune (pun intended) with the usual usage of PTC. Suppose we want to fit the final position of the ray, as in section (§5.3), as well as the transverse tunes and the phase slip. Then, as in Eq. (42), a contracting map can be constructed from a Lagrange function:

$$G = \frac{1}{2}\sum_{i=1,12} k_i^2 + \sum_{i=1,3} \mu_i \left(\nu_i\left(\mathbf{k}\right) - \nu_{0i}\right) + \sum_{i=1,4} \lambda_i \left(z_i^f\left(\mathbf{k}\right) - \mathbf{g}_i\right) \tag{45}$$

This time the vector $\mathbf{v} = (k_1, \cdots, k_{12}, \mu_1, \mu_2, \mu_3, \lambda_1, \lambda_2, \lambda_3, \lambda_4)$ contains the 12 dipole strengths, 3 Lagrange multipliers for the tunes and phase slip and finally 4 Lagrange multipliers for the orbit.

The code is similar to the code of section (§5.3). However this time a "real map" is computed with 6 phase space variables (Bmad) and 12 parameters for a total of 18 variables. This map is normalized and the transverse tunes in **phase(1:2)** and the phase slip in **phase(3)** are extracted.

In part 2, the equations to solve are saved via the **c_universal_taylor**'s **tpos(:)**. Finally the Lagrange map with the 19 variables of **v** is computed and inverted as part of the Newton search.

Here is the code. This concludes our discussion on "Taylor maps" coming out of TC.

```
p=>ring%start
call move_to(ring,p,"QF1")    !  Locating QF1 in TC
call make_it_knob(p%magp%bn(2),1)   !  PP : making a Polymorph into a knob
call move_to(ring,p,"QF2")    ! Locating QF2 in TC
call make_it_knob(p%magp%an(2),2)   !  PP : making a Polymorph into a knob
call move_to(ring,p,"QD1")     !  Locating QD1 in TC
call make_it_knob(p%magp%bn(2),3)   !  PP : making a Polymorph into a knob
call move_to(ring,p,"QD2")     !  Locating QD2 in TC
call make_it_knob(p%magp%an(2),4)   !  PP : making a Polymorph into a knob
call move_to(ring,p,"BEND")     !  Locating BEND in TC
call make_it_knob(p%magp%bn(2),5)   !  PP : making a Polymorph into a knob
call move_to(ring,p,"BEND1")    !  Locating BEND1 in TC
call make_it_knob(p%magp%an(2),6)   !  PP : making a Polymorph into a knob

p=>ring%start
call move_to(ring,p,"QF1")    !  Locating QF1 in TC
call make_it_knob(p%magp%bn(1),7)   !  PP : making a Polymorph into a knob
call move_to(ring,p,"QF2")    ! Locating QF2 in TC
call make_it_knob(p%magp%an(1),8)   !  PP : making a Polymorph into a knob
call move_to(ring,p,"QD1")     !  Locating QD1 in TC
call make_it_knob(p%magp%bn(1),9)   !  PP : making a Polymorph into a knob
call move_to(ring,p,"QD2")     !  Locating QD2 in TC
call make_it_knob(p%magp%an(1),10)   !  PP : making a Polymorph into a knob
call move_to(ring,p,"BEND")     !  Locating BEND in TC
call make_it_knob(p%magp%bn(1),11)   !  PP : making a Polymorph into a knob
call move_to(ring,p,"BEND1")    !  Locating BEND1 in TC
call make_it_knob(p%magp%an(1),12)   !  PP : making a Polymorph into a knob
```

```
write(6,*) " %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%"

bmad_state=bmad_state+nocavity0

p=>ring%start
closed_orbit=0
call find_orbit_x(closed_orbit,bmad_state, 1.0e-7_dp, fibre1=p)
write(6,*) " closed orbit "
write(6,"(6(1x,g11.4))") closed_orbit

goal=0.d0
entrance_orbit=closed_orbit
goal(1)=0.28d0
goal(2)=0.79d0
goal(3)=(-1)**ndpt_bmad*2.5d-2
goal(4:7)=entrance_orbit(1:4)

do k=1,8

!!!!!! Part 1 : calling TC !!!!!!
neq=7     ! number of equations to solve
np=12     ! number of free parameters to solve these equations
no=2 ;    ! nv=6+12=18
call init(bmad_state,no,np)   ! Berz's TPSA package is initialized
nV=c_%nv   !
call alloc(ct)
call alloc(identity_AP,one_turn_map_AP);call alloc(normal_form);
call alloc(ray_8_TC);call alloc(phase);call alloc(eq)
p=>ring%start

ray_TC=entrance_orbit   !  For TC
identity_AP=1
ray_8_TC=ray_TC + identity_AP ! Connect AP with TC: identity added

p=>ring%start
call propagate(ray_8_TC,+bmad_state,fibre1=p) !  TC of PTC proper

one_turn_map_AP=ray_8_TC

! compute 2 tunes and the phase slip
call c_normal(one_turn_map_AP,normal_form,phase=phase)
ray_TC=ray_8_TC
write(6,"(3(1x,g17.10,1x))") normal_form%tune(1:3)
write(6,"(6(1x,g17.10,1x))") ray_TC%x(1:6)
 identity_AP=0

phase(3)=phase(3).d.(5+ndpt_bmad)    ! compute phase slip dbeta*t/ddelta

phase(1)=(phase(1).o.identity_AP)<=6  ! removing delta dependence
phase(2)=(phase(2).o.identity_AP)<=6  ! removing delta dependence
phase(3)=(phase(3).o.identity_AP)<=6  ! removing delta dependence


eq(1)=phase(1)-goal(1)  ! tune x
eq(2)=phase(2)-goal(2)  ! tune y
eq(3)=phase(3)-goal(3)  ! pahse slip

do i=1,4
ct=ray_8_TC%x(i)%t
eq(3+i)=((ct.o.identity_AP)<=6)-goal(3+i)
enddo

!!!!!! Part 2 : saving the relevant Taylor series  !!!!!!
   call ALLOC(tpos,1,NV,0)
do i=1,7
 tpos(i)=eq(i)
enddo
```

```
call kill(identity_AP,one_turn_map_AP);call kill(normal_form);
call kill(ray_8_TC);call kill(phase);call kill(eq);call kill(ct)  ;
!!!!!! Part 3 : creating a map of some sort and using it !!!!!!

neq=7    ! number of equations to solve
np=12     ! number of free parameters to solve these equations
no=no-1; nV=NP+neq ;  ! nv=7+12=19
call c_init_all(no,nv)   ! FPP command to initialize the TPSA
lagrange_map_ap%n=nV   ! size of the c_damap must be explicitely stated
call alloc(ct);call alloc(eq);call alloc(lagrange_map_ap);
do i=1,7
 eq(i)=tpos(i)
enddo
!!!  Construction of Lagrange function
do i=1,np
 lagrange_map_ap%v(i)=1.d0.cmono.i
 do j=np+1,np+neq
  ct=(eq(j-np).d.i)*(1.d0.cmono.j)
   lagrange_map_ap%v(i)=lagrange_map_ap%v(i)+ct
 enddo
enddo

do i=np+1,np+neq
 lagrange_map_ap%v(i)=eq(i-np)
enddo

lagrange_map_ap=lagrange_map_ap.oo.(-1)

f_ray=0
f_ray=lagrange_map_ap.o.f_ray

p=>ring%start
call move_to(ring,p,"QF1")   !  Locating QF1 in TC
 k1=f_ray%x(1)
 call add(p,2,1,k1)
call move_to(ring,p,"QF2")
 k1=f_ray%x(2)
 call add(p,-2,1,k1)
call move_to(ring,p,"QD1")   !  Locating QF1 in TC
 k1=f_ray%x(3)
 call add(p,2,1,k1)
call move_to(ring,p,"QD2")   !  Locating QF2 in TC
 k1=f_ray%x(4)
 call add(p,-2,1,k1)
call move_to(ring,p,"BEND")   !  Locating QF2 in TC
 k1=f_ray%x(5)
 call add(p,2,1,k1)
call move_to(ring,p,"BEND1")   !  Locating QF2 in TC
 k1=f_ray%x(6)
 call add(p,-2,1,k1)

p=>ring%start
call move_to(ring,p,"QF1")   !  Locating QF1 in TC
 k1=f_ray%x(7)
 call add(p,1,1,k1)
call move_to(ring,p,"QF2")   ! Locating QF2 in TC
 k1=f_ray%x(8)
 call add(p,-1,1,k1)
call move_to(ring,p,"QD1")     !  Locating QD1 in TC
 k1=f_ray%x(9)
 call add(p,1,1,k1)
call move_to(ring,p,"QD2")     !  Locating QD2 in TC
 k1=f_ray%x(10)
 call add(p,-1,1,k1)
call move_to(ring,p,"BEND")     !  Locating BEND in TC
 k1=f_ray%x(11)
```

```
 call add(p,1,1,k1)
call move_to(ring,p,"BEND1")    !  Locating BEND1 in TC
 k1=f_ray%x(12)
 call add(p,-1,1,k1)

call kill(ct) ;call kill(eq);call kill(lagrange_map_ap);
enddo
```

In this code we fit the orbit to the original closed orbit.

The results are for the tunes, phase slip and the orbit:

```
  0.2828310304        0.7875313802       -0.2508933816E-01
  0.4575879602E-02   -0.8650459066E-05   -0.1701393632E-04   -0.4623033694E-04
-0.1176909853E-02     0.000000000
  0.2799916973        0.7900014703       -0.2499706314E-01
  0.4581594167E-02   -0.8590151609E-05   -0.1700070068E-04   -0.4623484201E-04
-0.1214079405E-02     0.000000000
  0.2799981589        0.7900002081       -0.2500013027E-01
  0.4575879632E-02   -0.8650379343E-05   -0.1701358606E-04   -0.4623024223E-04
-0.1212988306E-02     0.000000000
  0.2799999976        0.7899999994       -0.2500000017E-01
  0.4575879600E-02   -0.8650459057E-05   -0.1701393633E-04   -0.4623033694E-04
-0.1213016115E-02     0.000000000
  0.2800000000        0.7900000000       -0.2500000000E-01
  0.4575879602E-02   -0.8650459066E-05   -0.1701393632E-04   -0.4623033694E-04
-0.1213016151E-02     0.000000000
  0.2800000000        0.7900000000       -0.2500000000E-01
  0.4575879602E-02   -0.8650459067E-05   -0.1701393632E-04   -0.4623033694E-04
-0.1213016151E-02     0.000000000
  0.2800000000        0.7900000000       -0.2500000000E-01
  0.4575879602E-02   -0.8650459066E-05   -0.1701393632E-04   -0.4623033694E-04
-0.1213016151E-02     0.000000000
  0.2800000000        0.7900000000       -0.2500000000E-01
  0.4575879602E-02   -0.8650459066E-05   -0.1701393632E-04   -0.4623033694E-04
-0.1213016151E-02     0.000000000
```

6 Normalizing a TPSA map

The code for this section is located at **z_track_normal_tpsa.f90**.

6.1 Normalizing as a DA map around the closed orbit

Here we run a code where we can choose to get a one-turn map around the closed orbit (DA map) or around the orbit $x = 0$ which in this ring would correspond to the so-called "design orbit". As in section (§5.1) we used to knobs corresponding to quadrupole components:

```
p=>ring%start
call move_to(ring,p,"QF1")    !  Locating QF1 in TC
```

```
call make_it_knob(p%magp%bn(2),1)  !  PP : making a Polymorph into a knob
call move_to(ring,p,"QF2")   ! Locating QF2 in TC
call make_it_knob(p%magp%an(2),2)  !  PP : making a Polymorph into a knob
```

The next step is to decide whether we want a DA map or a TPSA map around the so-called "design orbit" **x0=0** here:

```
 bmad_state=only_4d
     .
     .
     .
 p=>ring%start
closed_orbit=0
call find_orbit_x(closed_orbit,bmad_state, 1.0e-7_dp, fibre1=p)
write(6,*) " closed orbit "
write(6,"(6(1x,g11.4))") closed_orbit
 x0=closed_orbit

write(6,*) " For TPSA map around f=(0,...,0) type 1 "
write(6,*) " For DA map around closed orbit type 0 "
read(5,*) ignore_closed_orbit
write(6,*) " Give order no "
read(5,*) no
if(ignore_closed_orbit==1) then
 closed_orbit=0
endif
```

We first show the results for the DA map:

```
  closed orbit
  0.4576E-02 -0.8650E-05 -0.1701E-04 -0.4623E-04   0.000       0.000
  For TPSA map around f=(0,...,0) type 1
  For DA map around closed orbit type 0
0
  Give order no
2
 Transverse tunes
  0.2828310304       0.7875313802
 Dampings (numerical noise)
 -0.4329869796E-14   0.1332267630E-14

z_1-component of the Linear Vector Field in Phasors Variables

          1, NO =    3, NV =    6, INA =  358
 ****************************************

    I  COEFFICIENT           ORDER   EXPONENTS
       NO =     3      NV =     6
   1   0.000000000000000       -1.777079774508286       1  0  0  0  0  0
    -1   0.000000000000000        0.000000000000000       0  0  0  0  0  0

 z_1-component of the Non-Linear Vector Field in Phasors Variables (2nd order)

          1, NO =    3, NV =    6, INA =  659
```

```
 ****************************************

    I   COEFFICIENT           ORDER    EXPONENTS
       NO =      3       NV =      6
   2    0.000000000000000       -2.580306677113468        1   0   0   0   1   0
   2    0.000000000000000       0.3425829877097634E-01    1   0   0   0   0   1
   3    0.000000000000000       -2900.949436382287        2   1   0   0   0   0
   3    0.000000000000000        6556.211092278534        1   0   1   1   0   0
   3    0.000000000000000       -0.2020888563626094       1   0   0   0   2   0
   3    0.000000000000000       -0.2080839310516064       1   0   0   0   1   1
   3    0.000000000000000        5.902822986039759        1   0   0   0   0   2
    -7    0.000000000000000         0.000000000000000         0   0   0   0   0   0
```

In the above example, the vector field is given by:

$$F \cdot \nabla = i \left\{ -1.777 - 2.580 dk_1^n - 0.034 dk_2^s - 2900.949 \underbrace{z_1 z_2}_{J_1} + \cdots \right\} z_1 \partial_{z_1} + \text{other planes} \qquad (46)$$

In fact, it is a purely imaginary vector field: it represents a tune. In fact $1.777079774508286/2/\pi = 0.2828310304\cdots$ which is just the tune in the first plane.

There is no real part in this vector field since this would represent damping. The state of PTC, **bmad_state=only_4d**, is a cavity-less radiation-less, state which restricts Taylor maps to the transverse orbital dimensions as the **4d** in **only_4d** indicates.

6.2 Normalizing as a TPSA map around the $(0,0,0,0,0,0)$

Because the map is not around the closed orbit, we must first move the TPSA map around the closed orbit using the algorithm of section (§4.4) and the same code as in page 25. This is given by the code:

```
if(ignore_closed_orbit==1) then
!   Compute TPSA closed orbit
one_turn_map_AP%x0(1:c_%nd2)=x0(1:c_%nd2)
c_w0%x0(1:c_%nd2)=x0(1:c_%nd2)

!!! inversion only need in the transverse plane in this example
do i=1,c_%nd2
 c_w0%v(i)=one_turn_map_AP%v(i)-(1.d0.cmono.i)-one_turn_map_AP%x0(i)
enddo

 c_w0_inv=c_w0.oo.(-1)

f_map%x0=0
do i=1,c_%nd2
f_map%v(i)=0.d0
f_map=c_w0_inv.o.f_map
enddo

f_ray=0
f_ray=c_w0_inv.o.f_ray
```

```
! Similarity transformation to the closed orbit

go_to_orbit= 1
go_to_orbit%x0(1:c_%nd2)=f_ray%x(1:c_%nd2)
do i=1,c_%nd2
 go_to_orbit%v(i)= go_to_orbit%v(i)+f_ray%x(i)
enddo

one_turn_map_AP=(go_to_orbit.o.one_turn_map_AP).o.(go_to_orbit.oo.(-1))

 write(6,"(a18,6(1x,g11.4))") "Exact closed orbit",closed_orbit(1:c_%nd2)
 write(6,"(a18,6(1x,g11.4))") "TPSA  closed orbit",real(f_ray%x(1:c_%nd2))

endif
```

The result of this normalization is to second order:

```
Exact closed orbit  0.4576E-02 -0.8650E-05 -0.1701E-04 -0.4623E-04
TPSA  closed orbit  0.4575E-02 -0.8523E-05 -0.1716E-04 -0.4624E-04
  EIG6: Eigenvalues off the unit circle!
   1.00159643192902
  EIG6: Eigenvalues off the unit circle!
   1.00159643192902
  EIG6: Eigenvalues off the unit circle!
   1.00056073888672
  EIG6: Eigenvalues off the unit circle!
   1.00056073888672
 Transverse tunes
  0.2826834097       0.7872786253
 Dampings (numerical noise)
 -0.1595158986E-02  -0.5605817314E-03

 z_1-component of the Linear Vector Field in Phasors Variables

          1, NO =    3, NV =    6, INA =  358
 *****************************************

    I  COEFFICIENT           ORDER   EXPONENTS
       NO =     3       NV =     6
   1  0.1595158986163770E-02  -1.776152246232164       1  0  0  0  0  0
    -1   0.000000000000000        0.000000000000000       0  0  0  0  0  0

 z_1-component of the Non-Linear Vector Field in Phasors Variables (2nd order)

          1, NO =    3, NV =    6, INA =  667
 *****************************************

    I  COEFFICIENT           ORDER   EXPONENTS
       NO =     3       NV =     6
   2 -0.1819669443106595       -2.657937823855120       1  0  0  0  1  0
   2  0.1187338977469338E-02  0.3373570996746057E-01   1  0  0  0  0  1
   3   1707.209464520280       -2338.602046283265       2  1  0  0  0  0
   3   782.8781372860180        6922.042189398233       1  0  1  1  0  0
   3   3.114213005812829      -0.5998691420874549       1  0  0  0  2  0
   3 -0.2927537938038314E-01  -0.1634680876623557       1  0  0  0  1  1
   3  0.7397499373591118E-01   5.985271299438016       1  0  0  0  0  2
    -7   0.000000000000000        0.000000000000000       0  0  0  0  0  0
```

One notices the problems: apparitions of damping, i.e., non-symplectic terms. For example, the (anti)-damping in the first plane is **-0.1595158986E-02**. This is an unacceptable artifact of the feed-down being completely wrong. We see also non-linear anti-damping. This is also a phenomenon of non-symplectic integrators.

Of course we can run at a higher order, for example $no = 11$, printing only numbers greater than 10^{-10}:

```
Exact closed orbit  0.4576E-02 -0.8650E-05 -0.1701E-04 -0.4623E-04
TPSA  closed orbit  0.4576E-02 -0.8650E-05 -0.1701E-04 -0.4623E-04
 Transverse tunes
  0.2828310304        0.7875313802
 Dampings (numerical noise)
 -0.1043609643E-13    0.2309263891E-13

 z_1-component of the Linear Vector Field in Phasors Variables

          1, NO =   11, NV =    6, INA =  494
 *****************************************

    I  COEFFICIENT           ORDER   EXPONENTS
       NO =    11       NV =     6
   1    0.000000000000000       -1.777079774507927        1  0  0  0  0  0
    -1    0.000000000000000        0.000000000000000        0  0  0  0  0  0

 z_1-component of the Non-Linear Vector Field in Phasors Variables (2nd order)

          1, NO =   11, NV =    6, INA =  605
 *****************************************

    I  COEFFICIENT           ORDER   EXPONENTS
       NO =    11       NV =     6
   2    0.000000000000000       -2.580306677112761        1  0  0  0  1  0
   2    0.000000000000000      0.34258298776735O7E-01     1  0  0  0  0  1
   3  0.74029694860210 55E-08   -2900.949436547272        2  1  0  0  0  0
   3  0.56521092794384 09E-06    6556.211092675170        1  0  1  1  0  0
   3 -0.22485613356797 24E-08  -0.2020888596108968        1  0  0  0  2  0
   3  0.68702461249911 54E-09  -0.2080839306933067        1  0  0  0  1  1
   3    0.000000000000000        5.902822986046178        1  0  0  0  0  2
    -7    0.000000000000000        0.000000000000000        0  0  0  0  0  0
```

We see that the situation as improved and approaches the DA result. It is not surprising that users of COSY-INFINITY must run at very higher order in rings. This is not necessary with a symplectic integrator provided all the maps are around the closed orbit, i.e., DA maps.

7 Taylors and Polymorphs

The PP of FPP defines two fundamental structures that are the building blocks of many other structures: **taylor** and **complextaylor**. The **direct** use of either **taylor** or **complextaylor** in a tracking program (in a simulation code like TC) is highly discouraged. Polymorphic types like **real_8** and **complex_8** should be used.

taylor and **complextaylor** are used via the polymorphic types **real_8** and **complex_8** in the tracking code TC. They are no longer used in AP. Indeed the analysis used in FPP uses the package **c_dabnew.f90** which is a "complexification" of Berz's **dabnew.f90**: it uses the complex Taylor **c_taylor**. The glue between these two world is described in section (§5). These types delay the usage of a Taylor and this save memory: the program can decide during execution if a quantity is real or Taylor.

They are defined as follows:

```
type real_8
 type (taylor) t ! used if taylor
 real(dp) r ! used if real
!
 integer kind !  1,2,3 (1=real,2=taylor,3=taylor knob )
 integer i ! used for knobs and special kind=0
 real(dp) s ! scaling for knobs and special kind=0
 logical(lp) :: alloc ! true if taylor is allocated in c_dabnew.f90 of Berz
end type real_8


type complex_8
 type (complextaylor) t    -> t= t%r +i  t%i are 2 real taylors
 complex(dp) r
 logical(lp) alloc
 integer kind
 integer i,j
 complex(dp) s
end type complex_8
```

7.1 The various Taylor types of the analysis package

We describe below the taylor (**taylor**) and complex Taylor (**complextaylor**) used in the polymorphs. They are all based on the real package of Berz: **dabnew.f90**.

7.1.1 Taylor Type

The **taylor** structure stores an integer which points to a stored Taylor (or element of ${}_{no}D_{nv}$) in Berz's original package.

The structure is:

```
type taylor
   type (taylor) r     ! Real part of complex Taylor series.
   type (taylor) i     ! Imaginary part of complex Taylor series.
end type complextaylor
```

We can look at the function **add**, from **i_tpsa.f90**, which overloads addition:

```
  function add( s1, s2 )
```

```
    implicit none
    type (taylor) add
    type (taylor), intent (in) :: s1, s2
    integer localmaster
    if(.not.c_%stable_da) then
       add%i=0
      return
    endif
    localmaster=master

    call ass(add)

 ! Old Fortran77 called on the integer pointers of
 ! s1, s2 and add, all type taylors

    call daadd(s1%i,s2%i,add%i)

    master=localmaster

  end function add
```

7.1.2 ComplexTaylor Type

The **complextaylor** structure stores two Taylor series which represent the real and imaginary parts. The structure is:

```
type complextaylor
   type (taylor) r     ! Real part of complex Taylor series.
   type (taylor) i     ! Imaginary part of complex Taylor series.
end type complextaylor
```

Again, showing the function **add**, from **l_complex_taylor.f90**, which overloads addition, is self-explanatory:

```
  function add( s1, s2 )
    implicit none
    type (complextaylor) add
    type (complextaylor), intent (in) :: s1, s2
    integer localmaster
    localmaster=master
    call ass(add)

    add%r=s1%r+s2%r  ! adds using overloading
    add%i=s1%i+s2%i

    master=localmaster
  end function add
```

7.1.3 C_taylor Type

The **c_taylor** structure stores Taylor series in the complexified version of Berz's TPSA package: **c_dabnew.f90**. This type is used only in AP and in the glue between AP and TC. (See section §5.1)

The structure is:

```
type c_taylor
 integer i ! integer i is a pointer to the complexified berz package
end type c_taylor
```

Everything that was said about **taylor** in section (§7.1.1) applies to **c_taylor** if one substitutes **c_dabnew.f90** for Berz's package. In this new package, all the coefficients are **complex(dp)**.

7.2 Examples of the polymorphs real_8 and complex_8

The Taylor series embedded in types **real_8** and **complex_8**, types **taylor** and **complextaylor** respectively, are all coming from the real package **dabnew.f90** of Berz.

A polymorph can be an ordinary floating point number, a Taylor series from **c_dabnew.f90** or a knob which we will now describe. A knob is a latent simple Taylor:

$$t = t\%r + t\%s\,\Delta_{t\%i+nd2} \tag{47}$$

To understand these three instantiations of the **real8/complex_8** types, we propose the code **z_track_real_8.f90**. Consider these fragments from the program **z_track_real_8.f90**:

1. **Creation of knobs**

```
p=>ring%start
call move_to(ring,p,"QF1")   !  Locating QF1 in TC
!  PP : making a Polymorph into a knob
call make_it_knob(p%magp%bn(2),1)
p_QF1=>p
call move_to(ring,p,"QF2")   ! Locating QF2 in TC
call make_it_knob(p%magp%bn(2),2)
p_QF2=>p

write(6,*);write(6,*) "Printing knob 1  "
call print(p_QF1%magp%bn(2))
write(6,*);write(6,*) "Printing knob 2  "
call print(p_QF2%magp%bn(2))
```

2. **Using knobs with no phase space**

```
write(6,*) "A polyphorm powered by a knob : FPP no phase space maps "
x1=p_QF2%magp%bn(2)
write(6,*) " x1 "
```

```
call print(x1)
x2=p_QF2%magp%bn(2)+i_*p_QF1%magp%bn(2)
write(6,*) " x2"
call print(x2,6)
call kill(x1);call kill(x2);
```

3. **Using knobs with phase space**

```
no=2
nd=3
np=2
ndpt=0
knob=.true.
call c_init_all(no,nd,np,ndpt)
write(6,*) " Jordan Normal Form ", c_%ndpt

call alloc(x1);call alloc(x2);



write(6,*) "Taylors powered by a knob : FPP with phase space maps "
x1=p_QF2%magp%bn(2)
call print(x1)
x2=p_QF2%magp%bn(2)+i_*p_QF1%magp%bn(2)
call print(x2)
call kill(x1);call kill(x2);
```

4. **Using knobs with phase space via PTC**

```
call in_bmad_units()
bmad_state=default+time0      !+nocavity0

call init_all(bmad_state,no,np)
write(6,*) " Jordan Normal Form ", c_%ndpt


call alloc(x1);call alloc(x2);

call print(bmad_state)


write(6,*) "Taylors created by a knob : PTC with phase space maps "
x1=p_QF2%magp%bn(2)
call print(x1)
x2=p_QF2%magp%bn(2)+i_*p_QF1%magp%bn(2)
call print(x2)
```

5. **Ignoring knobs PTC**

```
knob=.false.
write(6,*) "Knobs are ignored "

x1=p_QF2%magp%bn(2)
call print(x1)
x2=p_QF2%magp%bn(2)+i_*p_QF1%magp%bn(2)
call print(x2)

call kill(x1);call kill(x2);
```

6. **Removing knobs permanently**

```
write(6,*) "Knobs are eliminated "
knob=.true.
if(p_QF1%magp%bn(2)%kind==3) p_QF1%magp%bn(2)%kind=1
if(p_QF2%magp%bn(2)%kind==3) p_QF2%magp%bn(2)%kind=1
x1=p_QF2%magp%bn(2)
call print(x1)
x2=p_QF2%magp%bn(2)+i_*p_QF1%magp%bn(2)
call print(x2)
```

7.2.1 Initializing a polymorph: fragment 1

All polymorphs must be initialized prior to their usage: this done with the call

```
call alloc(x1}
```

or in the depth of PTC by

```
subroutine zero_anbn_p(el,n)
 implicit none
 type(elementp), intent(inout) ::el
 integer, intent(in) ::n

 if(n<=0) return
 if(associated(el%an)) deallocate(el%an)
 if(associated(el%bn)) deallocate(el%bn)
 el%p%nmul=n
 allocate(el%an(el%p%nmul),el%bn(el%p%nmul))
 call alloc(el%an,el%p%nmul);    ! polymorphic an constructed
 call alloc(el%bn,el%p%nmul);    ! polymorphic bn constructed

end subroutine zero_anbn_p
```

This allocation is done, in the case of **an**,**bn**, prior to any call to the TPSA package. Polymorphs can be allocated as soon as the space in memory exists.

Next we create two knobs, for example,

```
p=>ring%start
```

```
call move_to(ring,p,"QF1")   !  Locating QF1 in TC
!  PP : making a Polymorph into a knob
call make_it_knob(p%magp%bn(2),1)
p_QF1=>p
```

This implies tthat the quadrupole component **bn(2)** of the magnet QF1 can become the $nd2+1$ variable of a Taylor series. If we run the code of item 1, we get:

```
 Printing knob 1
  printing a real polymorph (real_8)
2.25384743526091+1.00000000000000(x_1)

 Printing knob 2
  printing a real polymorph (real_8)
2.24740000000000+1.00000000000000(x_2)
```

This shows that the two polymorphs are latent Taylor series consistent with Eq. (47). Now we see how that is actualized in a real code.

7.2.2 Knobs with no phase space: item 2

Also the TPSA package must be initialized otherwise the code will not know the values *no* and *nv*: remember that the code tracks an element of the algebra ${}_{no}D_{nv}$ (see section §3.8).

In fragment of item 2, as in section (§5.2), the call **c_init_all(no,nv)** creates polynomials in **nv** variables but **no** phase space, i.e., $nd2 = 0$, therefore the parameters for **QF1** and **QF2** are respectively 1 and 2. The output from item 2 is:

```
 A polyphorm powered by a knob : FPP no phase space maps
  x1

 etall     1, NO =     2, NV =     7, INA =    28
 *****************************************

    I   COEFFICIENT           ORDER    EXPONENTS
       NO =      2       NV =      7
   0    2.247400000000000        0  0  0  0  0  0  0
   1    1.000000000000000        0  1  0  0  0  0  0
    -2    0.000000000000000          0  0  0  0  0  0  0
  x2
  Real Part

 etall     1, NO =     2, NV =     7, INA =    35
 *****************************************

    I   COEFFICIENT           ORDER    EXPONENTS
       NO =      2       NV =      7
   0    2.247400000000000        0  0  0  0  0  0  0
   1    1.000000000000000        0  1  0  0  0  0  0
    -2    0.000000000000000          0  0  0  0  0  0  0
  Imaginary Part

 etall     1, NO =     2, NV =     7, INA =    36
```

```
****************************************

   I   COEFFICIENT           ORDER    EXPONENTS
     NO =      2       NV =      7
  0    2.253847435260914        0   0   0   0   0   0   0
  1    1.000000000000000        1   0   0   0   0   0   0
   -2    0.000000000000000          0   0   0   0   0   0   0
```

We see that the knobs are the first and second variable of the Taylor series.

7.2.3 Knobs with phase space: item 3

In fragment of item 3, as in section (§5.2), the call **c_init_all(no,nd,np,ndpt=0)** creates polynomials in **nv=2*nd+np** variables. The huge difference with the call from the previous section (§7.2.2) is the creation of a phase space of **nd2=2*nd** dimension, **nd** degrees of freedom. According to Eq. (47), the position of knob 1 and 2 will be respectively 7 and 8.

The variable **ndpt** can be give the values 0, 5 or 6. It affects on the normal form. If ndpt is 0 (**ndpt** is optional, default=0), the 3 planes execute pseudo-harmonic oscillations and the normal form is 3 rotations. If the **ndpt** is 5 or 6, than the normal form is a drift-like matrix in the longitudinal plane. In that case, **ndpt** is the position of the energy-like variable. It is 6 in Bmad and that variable is $\frac{\delta p}{p_0}$. In this section, since there are no normal form displayed, this is completely irrelevant.

Here is the result of running item 3:

```
Taylors powered by a knob : FPP with phase space maps

etall    1, NO =    2, NV =    8, INA =   30
****************************************

   I   COEFFICIENT           ORDER    EXPONENTS
     NO =      2       NV =      8
  0    2.247400000000000        0   0   0   0   0   0   0   0
  1    1.000000000000000        0   0   0   0   0   0   0   1
   -2    0.000000000000000          0   0   0   0   0   0   0   0
 Real Part

etall    1, NO =    2, NV =    8, INA =   37
****************************************

   I   COEFFICIENT           ORDER    EXPONENTS
     NO =      2       NV =      8
  0    2.247400000000000        0   0   0   0   0   0   0   0
  1    1.000000000000000        0   0   0   0   0   0   0   1
   -2    0.000000000000000          0   0   0   0   0   0   0   0
 Imaginary Part

etall    1, NO =    2, NV =    8, INA =   38
****************************************

   I   COEFFICIENT           ORDER    EXPONENTS
     NO =      2       NV =      8
  0    2.253847435260914        0   0   0   0   0   0   0   0
```

```
   1    1.000000000000000        0   0   0   0   0   0   1   0
    -2    0.000000000000000          0   0   0   0   0   0   0   0
```

7.2.4 Knobs with phase space via PTC: item 4

The result below is identical to that of section (§7.2.3). The difference lies in the way the FPP is activated. Here we pass a so-called **internal_state**. First, in item 4, we see that main program invoked Bmad units: therefore energy will be the sixth variable. Secondly, by default, the cavity is turned on. This corresponds to the FPP call **c_init_all(no,nd,np,ndpt=0)**. If the **internal_state** invokes **nocavity0**, than cavities are generally turned into drifts and this is equivalent to **c_init_all(no,nd,np,ndpt=6)** in Bmad units.

```
 Jordan Normal Form            0
************ State Summary ****************
MADTHICK=>KIND =   32  DRIFT-KICK-DRIFT
 Rectangular Bend: input arc length (rho alpha)
 Default integration method     2
 Default integration steps      8
This is an electron (positron actually if charge=1)
      EXACT_MODEL =  TRUE
      TOTALPATH   =     0
      RADIATION   =  FALSE
      STOCHASTIC  =  FALSE
      ENVELOPE    =  FALSE
      NOCAVITY    =  FALSE
      TIME        =  TRUE
      FRINGE      =  FALSE
      PARA_IN     =  FALSE
      ONLY_2D     =  FALSE
      ONLY_4D     =  FALSE
      DELTA       =  FALSE
      SPIN        =  FALSE
      MODULATION  =  FALSE
      RAMPING     =  FALSE
      ACCELERATE  =  FALSE
Taylors created by a knob : PTC with phase space maps

etall    1, NO =    2, NV =    8, INA =   30
*********************************************

   I  COEFFICIENT          ORDER   EXPONENTS
     NO =     2      NV =     8
  0   2.247400000000000        0  0  0  0  0  0  0  0
  1   1.000000000000000        0  0  0  0  0  0  0  1
   -2   0.000000000000000        0  0  0  0  0  0  0  0
 Real Part

etall    1, NO =    2, NV =    8, INA =   37
```

```
 ******************************************

    I   COEFFICIENT           ORDER    EXPONENTS
       NO =      2       NV =      8
   0    2.247400000000000        0  0  0  0  0  0  0  0
   1    1.000000000000000        0  0  0  0  0  0  0  1
    -2    0.000000000000000         0  0  0  0  0  0  0  0
  Imaginary Part

 etall     1, NO =     2, NV =     8, INA =    38
 ******************************************

    I   COEFFICIENT           ORDER    EXPONENTS
       NO =      2       NV =      8
   0    2.253847435260914        0  0  0  0  0  0  0  0
   1    1.000000000000000        0  0  0  0  0  0  1  0
    -2    0.000000000000000         0  0  0  0  0  0  0  0
```

8 The c_damap type: fundamental to analysis

The **c_damap** type is given by:

```
type c_damap
! Orbital part
 type (c_taylor) v(lnv) !@1 orbital part of the map
 type(c_quaternion) q
! Stochastic part
 complex(dp) e_ij(6,6) !@1 stochastic fluctuation in radiation theory
! Number of planes allocated
 integer :: n=0 !@1 number of planes allocated
!  Initial orbit if tpsa =  true
 complex(dp) x0(lnv)
 logical :: tpsa=.false.
!  Lie map matrix  (Yu's square matrix)
 complex(dp), pointer :: cm(:,:)=> null()
!  Moment matrix (transpose of Yu Matrix)
 real(dpn),pointer :: m(:,:)=> null()
! SO(3) is deprecated in FPP
! and computed from quaternion if needed
 type(c_spinmatrix) s !@1 spin matrix
end type c_damap
```

8.1 The c_ type: storing global data for AP

Before discussing the **c_damap**, we need to understand a few parameters that are crucial to that map and are global in AP. We can show a code fragment that prints them in particularly complex

cases. This is from the main program **z_track_c_.f90**.

There are two basic ways to track in TC: the RF-cavity is on or the RF-cavity is off. Even when RF cavity are absolutely necessary, it is useful to compute a 6-d map with the RF cavity off, i.e., with $\delta p =$ constant. Why?

state=nocavity In that case, it is possible, via simple normal form, to compute things like the chromaticities, the dispersions, the phase slip, etc...Strictly speaking these concepts are ill-defined in the presence of longitudinal oscillations. However they are approximately defined for a small synchrotron (longitudinal) tune which is the case in normal rings.

state=default In that case, the cavities are turned on. This is usually the "physical" case. It is necessary when extracting from the map the longitudinal tunes and the damping decrements due to radiation. PTC also computes the quadratic fluctuations $\langle z_i z_j \rangle$, 21 of them, from which beam sizes can be extracted. We can also extract the so-called "equilibrium" emittances which are also approximate concepts, i.e., valid in the limit of small damping.

Consider the following call:

```
p=>ring%start
call move_to(ring,p,"QF1")   !  Locating QF1 in TC
!  PP : making a Polymorph into a knob
call make_it_knob(p%magp%bn(2),1,s=1.0_dp)
p_QF1=>p
call move_to(ring,p,"QF2")   ! Locating QF2 in TC
call make_it_knob(p%magp%bn(2),2,s=1.0_dp)
p_QF2=>p
        .
        .
no=2
nd=3
np=2
        .
        .
call in_bmad_units()
bmad_state=default+time      !+nocavity

call init_all(bmad_state,no,np)
```

This initializes FPP and AP in particular to track with cavities. The basic phase space will be 6 dimensional with 3 tunes. We print the relevant parts of **c_**:

```
  Taylors created with two knobs : PTC with phase space maps
 Cavities turned on
        c_%no =   2   !  Order of TPSA
        c_%nv =   8   !  Total number of variables
        c_%nd =   3   !  Orbital degrees of freedom
       c_%nd2 =   6   !  c_%nd2=2*c_%nd
   c_%nd2harm =   6   !  All oscillating planes including magnet modulation
      c_%ndpt =   0   !  Energy is not constant
```

```
 c_%ndpt_bmad =   1   !  Bmad units used
        c_%np =   2   !  Number of parameters
  c_%npara_fpp=   6   !  Position of parameters minus 1
       c_%nd2t=   6   !  Oscillating orbital planes
      c_%ndc2t=   0
 c_%pos_of_delta  6
         c_%rf=   0
    c_%setknob=   T
       c_%knob=   T
```

These numbers are all encoded in **bmad_state=default+time**. Bmad by default using 3 degrees of freedom. Moreover, these 3 planes are oscillating with a cavity. This is encoded in **c_%nd2t=6**. Notice that there are 8 variables total: 2 quadrupole strengths at location **c_%npara_fpp+1** and **c_%npara_fpp+2**.

Before we go into a detailed explanations of type **c_damap**, we show a more complex call to **init_all**. We run PTC with the **internal_state**

```
         .
         .
 call in_bmad_units()
!call in_ptc_units()
bmad_state=default+time0+nocavity0
bmad_state=bmad_state+modulation0+spin0
         .
         .
```

and then we get:

```
 Taylors created with 2 knobs : PTC with phase space maps
  without a cavity and modulation on a magnet
        c_%no =   2
        c_%nv =  10
        c_%nd =   4
       c_%nd2 =   8
   c_%nd2harm =   6
      c_%ndpt =   6
 c_%ndpt_bmad =   1
        c_%np =   2
  c_%npara_fpp=   8
       c_%nd2t=   4
      c_%ndc2t=   2
 c_%pos_of_delta  6
         c_%rf=   1
    c_%setknob=   T
       c_%knob=   T
```

The first thing we notice is that **c_%ndpt=6**. This means that the sixth variable is the energy variable and is a constant. The orbital phase space has 6 dimensions. However we notice that the number of oscillating dimensions is 4, i.e., **c_%nd2t=4** that is the two transverse orbital planes.

The total phase space is eight since **c_%nd2=8** and thus the maps have 8 entries. This is because of the presence of a modulating magnet. Additional parameters, the two quadrupoles, must occupy the 9^{th} and 10^{th} variables. The total number of TPSA variables and indeed we have **c_%nv=10**.

These numbers are necessary for AP. However this is automatically set up by the call to **init_all(bmad_state,no,np)**.

FPP does not attempt to do a forensic analysis of a map and then determine the right settings. Indeed if a map is created outside PTC, the user must either call **init_all(state,no,np)** with the proper state or directly call **c_init_all**. For example, the above case can be reproduced by the pure AP call:

```
no=2
nd=4
np=2
ndpt=6
knob=.true.
ac_modulation=1
call c_init_all(no,nd,np,ndpt,ac_modulation)
```

Even if you do not use PTC, calls to initialize FPP via a PTC **internal_state** like **bmad_state** are far less cryptic.

8.2 The c_damap in detail

The Taylor maps can be used for all sorts of purposes (see §5). But its main function, in TC proper, is analysis and **not** tracking. This is not true of "matrix" codes like COSY-INFINITY or MARYLIE where the ultimate object of the code is a one-turn Taylor map. In TC, the tracking is achieved normally by pushing through individual magnets using a variety of integrators. This is also true in the case of Bmad which not only includes PTC, but also also a collection of tracking methods which for the most part are integrators. For example, Bmad has a variable step Runge-Kutta method which handles complex magnets without the paraxial approximation,i.e., does not assume Hills equations.

It is true that most users of PTC seek it for its ability to produce Taylor maps, but it is fundamentally an integrator whose algebra is extended from $(\mathbb{R}, +, \times)$ to $({}_{no}D_{nv}, +, \times)$ via polymorphism. What it is and what it is used for — often confused by users— are two different things.

Here we concentrate on the DA map, i.e., a map around the closed orbit. All constant parts are ignored and that includes **c_damap%x0** whose functions is described in section (§4.1) and Eq. (22): it is the w_0 of map m_{w_0}.

The **c_damap** contains primarily phase space and a quaternion:

```
 type c_damap
! Orbital part
 type (c_taylor) v(lnv) !@1 orbital part of the map
 type(c_quaternion) q
       .
```

```
          .
```

The array **%(lnv)** contains the phase space variables. For example, in a normal Bmad run, it will contain the six polynomials for $(x, p_x, y, p_y, -\beta ct, \delta)$, the usual Bmad variables. If a "clock" is present for sinusoidal magnet modulation, then it will contain eight polynomials $(x, p_x, y, p_y, -\beta ct, \delta, q_c, p_c)$.

The type **c_quaternion** is made of four **c_taylor**:

```
 type  c_quaternion
  type(c_taylor) x(0:3)
end type c_quaternion
```

Two DA **c_damap**'s **S1** and **S2** are concatenated by the following code fragment:

```
function c_concat(s1,s2)
implicit none
type (c_damap) c_concat
type (c_damap), intent (in) :: s1, s2
t1=s1;t2=s2;

! code for s1 o s2
! removes the constant part since these are da maps
do i=1,t1%n
t1%v(i)=t1%v(i)-(t1%v(i).sub.'0')
enddo
do i=1,t2%n
t2%v(i)=t2%v(i)-(t2%v(i).sub.'0')
enddo

! calls to the concatanator in c_dabnew.f90
call c_etcct(t1%v%i,t1%n,t2%v%i,t2%n,tempnew%v%i)
! the constant part at the end of the orbit (not necessary)
if(add_constant_part_concat) then
do i=1,t1%n
tempnew%v(i)=tempnew%v(i)+(s1%v(i).sub.'0')
enddo
endif
t1%q = t1%q*t2 !!! substituting s2 into quaternion of s1
tempnew%q=t1%q*t2%q !!! multiplying quaternions
```

Symbolically, we can write the above code fragment as:

$$\begin{aligned} c\_concat\%x &= S1\%x \circ S2\%x \\ c\_concat\%q &= \underbrace{\{S1\%q \circ S2\%x\}}_{\text{transformed quaternion}} \times S2\%q \end{aligned} \tag{48}$$

The $\times$ in Eq. (48) represents quaternion multiplication.

We know that the motion can also include radiation. The classical radiation, which is deterministic, is stored in the orbital map. However, the stochastic part, the quadratic fluctuation of the

moments, is stored in **c_damap%e_ij(1:6,1:6)**. Denoting the linear part of **S1** and **S2** by the matrices **M1** and **S2**, we have:

$$c\_concat\%E\_ij \;=\; M2\; S1\%E\_ij\; M2^{\top} \;+\; S2\%E\_ij \tag{49}$$

Eq. (49) is realized by the code:

```
        .
        .
      t1=t1.sub.1
      m2=t1
      m2t=transpose(m2)
      tempnew%e_ij=t1%e_ij + matmul(matmul(m2,t2%e_ij),m2t)
```

Other parts of the **c_damap** are experimental. For example, in **c_damap%cm(:,:)** one can compute and store the matrix representation of the Lie map associated to the orbital map: this is a huge matrix acting on space of polynomials of order **no** and dimension **nv**. The huge matrix **c_damap%m(:,:)** is the matrix acting on moment. In a deterministic case, it is the transposed of **c_damap%cm(:,:)**.

These matrices are not used in AP for analysis. It is possible in theory to compute a normal form directly using the matrix **c_damap%cm(:,:)**, see [12]for example.

9 The normal form and the vector field

It is in the normal form where a lot of perturbation theory is embedded. This type can permit the extraction of any lattice function, linear and nonlinear. For example, it can **trivially** extract the dependence of the coupled Courant-Snyder invariants in terms of parameters including δp if energy is a constant. It can compute all quantities related to spin with equal ease using quaternions. Obviously it follows that phase advances for both the orbital and the spin are trivially contructed once these tools exist. This is the topic of Forest's new book [6] which unfortunately uses $SO(3)$ instead of quaternions.

Normal forms can also leave a "single resonance" in the map. This is topologically equivalent to the one-resonance Hamiltonian theory. It can be done with either an orbital resonance or a spin resonance.

Here is the normal form structure. We skip variables of the structure which are obsolescent.

```
type c_normal_form
!!!  full transformation with spin
 type(c_damap) atot  !   for spin   (m = atot n atot^-1)
 type(c_vector_field) h,h_l,h_nl
!!!envelope radiation stuff to normalize radiation (sand's like theory)
 complex(dp) s_ij0(6,6)  !   equilibrium beam sizes
 complex(dp) s_ijr(6,6)  !   equilibrium beam sizes in resonance basis
 complex(dp) b_ijr(6,6)  !    stochastic kick in resonance basis
!  equilibrium emittances as defined by chao
! (computed from s_ijr(2*i-1,2*i) i=1,2,3 )
```

```
 real(dp) emittance(3)
!! controls resonances left in normal form
! stores resonances to be left in the map, including spin (ms)
 integer nres,m(ndim2t/2,nreso),ms(nreso)
!! redundant stuff
! stores simple information
 real(dp) tune(ndim2t/2),damping(ndim2t/2),spin_tune,quaternion_angle
 logical positive ! forces positive tunes (close to 1 if <0)
        .
        .
        .
end type c_normal_form
```

9.1 The normal form type

See program **z_track_normal_tpsa.f90** for a simple call to the normal form.

The normal form rewrites a **c_damap** as follows:

$$n = a^{-1} \circ m \circ a \tag{50}$$

where n is called a normal form. In the simplest case, in the presence of a cavity, the orbital map is an amplitude dependent rotation. If classical radiation is present, it is an amplitude dependent dilation (rotation and damping akin to a circular drain). The simplest call to achieve this is

```
type(c_normal_form) normal_form
type(c_damap)  one_turn_map_AP
        .
        .
        .
call c_normal(one_turn_map_AP,normal_form)
```

We deduce that in Fortran90, the equivalent of Eq. (50) is

```
type(c_normal_form) normal_form
type(c_damap) n , one_turn_map_AP
        .
        .
        .
call c_normal(one_turn_map_AP,normal_form)
n=normal_form%a**(-1)*one_turn_map_AP*normal_form%a
```

The analysis AP has two global parameters of importance:

```
logical :: remove_tune_shift=.false.
complex(dp) :: n_cai=-2*i_
```

If the parameter **remove_tune_shift** is true, then in the case of a damped map, the code will attempt to remove all nonlinearities and create a linear sink. This is generally not recommended

in accelerator physics since damping decrements are often very small. On a map without radiation, this will create a crash since division by zero will occur: it is impossible to remove tune shifts with amplitude by similarity transformations in the symplectic case.

The parameter **n_cai** can be $-i = -\sqrt{(-1)}$ or $-2i$. This parameter selects the phasors' definition which is the complex transformation which diagonalizes the linear part of n which is rotation. That is to say that matrix part of n, denoted N, obeys, in 1-d-f:

$$\Lambda = C^{-1}NC \quad \text{where} \quad \mathrm{R} = \begin{pmatrix} \mathrm{e}^{-\mathrm{i}\mu} & 0 \\ 0 & \mathrm{e}^{i\mu} \end{pmatrix} \tag{51}$$

If **n_cai = -2i**, then

$$\begin{aligned} \text{map } c^{-1} : x^{new} &= x + ip \\ p^{new} &= x - ip \end{aligned} \tag{52}$$

Notice that the Poisson bracket $[x^{new}, p^{new}] = n_{cai} = -2i$. More importantly, we have for Eq. (52):

$$x^{new}p^{new} = x^2 + p^2 = 2J \tag{53}$$

If on the other hand **n_cai = -i**, then we have:

$$\begin{aligned} \text{map } c^{-1} : x^{new} &= \frac{1}{\sqrt{2}}(x + ip) \\ p^{new} &= \frac{1}{\sqrt{2}}(x - ip) \end{aligned} \tag{54}$$

$$\text{and} \quad [x^{new}, p^{new}] = -i \tag{55}$$

with

$$x^{new}p^{new} = x^2 + p^2 = J \tag{56}$$

It can be argued that the choice **n_cai = -i** is the best since the action is simply the product of the phase space variables.

In phasors, the full normalized map is:

$$d = c^{-1} \circ n \circ c = c^{-1} \circ a^{-1} \circ m \circ a \circ c \tag{57}$$

The maps c and c^{-1} are obtained using the function **c_phasor()** and **ci_phasor()** respectively.

9.2 The type c_vector_field type and Lie maps

The normal form is defined, in FPP, via vector fields:

```
type c_vector_field
! n dimension used v(1:n) (nd2 by default) ;
```

```
! nrmax some big limiting integer to use for exponentiation
 integer :: n=0,nrmax
! if eps=-integer  then |eps| # of lie brackets are taken
! otherwise eps=eps_tpsalie=10^-9
 real(dp) eps
! orbital part
 type (c_taylor) v(lnv)
! quaternion part
 type(c_quaternion) q
 end type c_vector_field
```

The type **c_quaternion** is defined as

```
type  c_quaternion
 type(c_taylor) x(0:3)
end type c_quaternion
```

A general quaternion is defined from type **c_quaternion** as

$$q = q\%x(0) + q\%x(1)b_1 + q\%x(2)b_2 + q\%x(3)b_3 \tag{58}$$

$$\text{where } \mathrm{b} = (i, j, k) \tag{59}$$

i, j and k are the 3 usual quaternions which define the quaternion algebra.

In FPP, the action of a **c_vector_field** on a

$$F = F\%\vec{v} \cdot \vec{\nabla} + F\%q = F\%v(i)\partial_i + F\%q \tag{60}$$

where in Eq. (60) it is summed over the index i.

A general vector field as in Eq. (60), acts on a map m as follows:

$$\begin{aligned} (Fm)\,\%x(k) &= F\%v(i)\partial_i m\%x(k) \\ (Fm)\,\%q &= F\%v(i)\partial_i m\%q + m\%q \times F\%q \end{aligned} \tag{61}$$

A map near the identity always have a vector field representation. So if m is near the identity

$$\exists F \quad m = \exp(F)\, I \qquad \text{where } I \text{ is the identity map} \tag{62}$$

If we have a function f of phase space, then the Lie maps have the following "virtue":

$$\text{if } \; m = \exp(F)\, I \implies f \circ m = \exp(F)\, f \tag{63}$$

Eq. (63) implies that transforming a function by a map m can be done with a Lie operator. It also implies that a Lie operator is a "compositional" map.

Here is a list of operation using the vector field.

Now we define the normal form in terms of Lie operators.

9.3 The Lie operator of the normal form

The map d of Eq. (57) is, in the simplest symplectic case, an amplitude dependent rotation. Using Dragt's notation for Lie maps (calligraphic font), we can write the Lie map associated to d, i.e., $\mathcal{D}$ as :

$$\mathcal{D} \;=\; \exp\left(H\cdot\nabla + q\right) \tag{64}$$

where

$$\begin{aligned} q &= \cos\left(\theta/2\right) + \sin\left(\theta/2\right) \\ \theta &= \theta(J_1,\cdots,J_{nd}) \;\text{ and }\; J_i = z_{2i-1}z_{2i} \quad \text{per} \quad Eq.\ (56) \end{aligned} \tag{65}$$

and

$$H_1\partial_1 = -iz_1\mu_1\left(J_1,\cdots,J_{nd}\right)\partial_1 \quad \text{and} \quad H_2\partial_2 = iz_1\mu_1\left(J_1,\cdots,J_{nd}\right)\partial_2 \tag{66}$$

In the case of damping, a real piece is added to Eq. (66):

$$\text{real part} = -z_1\alpha_1\left(J_1,\cdots,J_{nd}\right)\partial_1 \quad \text{and} \quad H_2\partial_2 = -z_1\alpha_1\left(J_1,\cdots,J_{nd}\right)\partial_2 \tag{67}$$

Finally in the absence of a cavity, the longitudinal normal form, in the Bmad units, have the form:

$$\mathrm{H}_5\frac{\partial}{\partial z_5} = z_6\phi\left(J_1,J_2;z_6\right)\partial_1 \quad \text{and} \quad H_6 = 0 \tag{68}$$

In Eq. (68), ϕ is the phase/time slip. z_5 is the time in the units of Bmad and z_6 is $\frac{\delta p}{p_0}$— a conserved quantity.

The **c_vector_field** of Eq. (64) is stored in the field **c_normal_form%H**.

It is also possible to leave a resonance in the map via a normal form: please look into reference[6]. If the normal form $\mathcal{D}$ is not a rotation then in FPP it can be written as

$$\mathcal{D} \;=\; \exp\left(H_{nl}\cdot\nabla + q_{nl}\right)\exp\left(H_l\cdot\nabla + q_l\right) \tag{69}$$

These vector fields are stored in **c_normal_form%h_nl** and **c_normal_form%h_l**.

9.4 Call to the normal form

```
subroutine c_normal(xyso3,n,dospin,no_used,rot,phase,nu_spin,canonize)

type(c_damap) , intent(inout) :: xyso3    ! map to be normalized
type(c_normal_form), intent(inout) ::  n
type(c_damap), optional :: rot  !  normal form rot=n%Atot**(-1)*xy*n%Atot
type(c_taylor), optional :: phase(:),nu_spin
logical(lp), optional :: dospin,canonize
integer,optional :: no_used
```

dospin is false by default: spin is ignored. In the symplectic case, **phase(1:nd)** will contain the tunes. If the last plane, 3 in Bmad, is not oscillating, this contains the time slip. If **canonize** is true, the map **n%atot** is put in a special form, i.e., the Courant-Snyder-Teng-Edwards form in the linear case (see [6]).

The normalization is done to order**no** of the algebra ${}_{no}D_{nv}$. But we can perform the normalization to lower order: this is specified by **no_used**. To perform a normalization to higher order than **no** is possible but it is very unusual: for that the map has to be stored in a "**c_universal_taylor**'' and then the TPSA package has to be re-initialized to higher order.

Finally the type **c_normal_form** has structures which can be modified in order to change the calculation. Here is the list assuming that **n** is a **c_normal_form**:

- If **n%nres=0**, the maps are turns into rotations and perhaps a time slip if no cavity are present.
- If **n%nres**$/=0$, then **n%nres** resonances, labelled by k, are left in the map:

$$n\%m(1,k)\nu_1 + \cdots + n\%m(ndharm,k)\nu_{ndharm} + n\%ms(k)\nu_{spin} = \text{integer} \tag{70}$$

  ndharm is the number of oscillating plane, typically 2 or 3 in Bmad. If a magnet is modulated, then **ndharm** is increased by 1, the tune of the modulation enters in the modulation.
- If **n%positive=.false.**, the fractional part of the tune stored in **n%tune** is between $-1/2, 1/2$. This does not apply to the time/phase slip. This is useful in "one-resonance" normal form.
- **force_spin_input_normal** is a global which is normally false. If true, it makes the optional parameter **dospin** obligatory.
- As pointed out above, the parameter **remove_tune_shift** is false by default. On a very damped map, one can set this parameter to true. The normal form is a linear sink around the origin. It is not recommended in accelerators.

9.5 Factorizing the map n%atot

Any Taylor map a can be factorized as follows:

$$a = a_0 \circ a_1 \circ a_2 \circ a_s \text{ if dir} = 1 \tag{71}$$

using the command

```
!# at = a_0 o a_1 o a_2 o a_s  for dir=1
 call c_full_factorise(at,as,a0,a1,a2,dir=1)
```

If **dir=-1**, the order is reverse.

This factorization is general. However we can explain it in the context of a normal form where

it is useful for finding the lattice functions around the parameter dependent closed orbit.

$$\begin{aligned} n &= a^{-1} \circ m \circ a \\ &= a_s^{-1} \circ a_2^{-1} \circ a_1^{-1} \circ \underbrace{a_0^{-1} \circ m \circ a_0}_{m_c} \circ a_1 \circ a_2 \circ a_s \\ &= a_s^{-1} \circ a_2^{-1} \circ \underbrace{a_1^{-1} \circ m_c \circ a_1}_{n_c} \circ a_2 \circ a_s \end{aligned} \tag{72}$$

1. The map a_0 is the identity in the harmonically oscillating parts of phase space, either 4 (no cavity) or 6 (with RF cavity). It brings the map to the parameter dependent fixed point where δ, if constant, is included.
2. The map a_1 turns the linear part of the map around the fixed , to all orders in the parameters, into a rotation.
3. The map a_2 is purely nonlinear.

An example of this useful call is in program **z_track_normal_fact.f90**. This example computes the so-called beta function in two different ways using a_1, First if defines a_1 as the coefficient of p_x^2 in the quadratic invariant. Secondly it defines it has the coefficient of J_1 in the computation of $< x^2 >$. In the Ripken formalism, these are the same functions.

```
!# normal_form%atot = a_0 o a_1 o a_2 o a_s  for dir=1
call c_full_factorise(normal_form%atot,as,a0,a1,a2,dir=1)

! computing the "beta function" two different ways
! x^2+p^2 is invariant of normal form n
ct=(1.d0.cmono.'2')+(1.d0.cmono.'02')
! (x^2+p^2) o a^-1 is invariant around the fixed point
ct=ct*a1**(-1)
allocate(jindex(c_%nd2harm))
call clean(ct,ct,prec=prec)
jindex=0
jindex(2)=2
ct=ct.par.jindex
Write(6,*) "Beta via Coefficient of p^2 in invariant"
call print(ct)
!  function x^2 is constructed
ct=(1.d0.cmono.'2')
!  function x^2 is averaged using a_1
call C_AVERAGE(ct,a1,ct)
jindex=0
jindex(1)=1
jindex(2)=1
!  coefficient of J_1 extracted
ct=ct.par.jindex
call clean(ct,ct,prec=prec)
```

```
write(6,*) " Beta via average "
call print(ct)
```

The result if **no=3** is:

```
 Beta via Coefficient of p^2 in invariant

          1, NO =     3, NV =     7, INA =   209
 ****************************************

    I   COEFFICIENT           ORDER    EXPONENTS
       NO =      3       NV =      7
   0    13.57348336645041        0.000000000000000        0   0   0   0   0   0   0
   1    25.47460380089213        0.000000000000000        0   0   0   0   1   0   0
   1   -7.011940286158579        0.000000000000000        0   0   0   0   0   1   0
   1   0.9489986322495689        0.000000000000000        0   0   0   0   0   0   1
    -4    0.000000000000000         0.000000000000000         0   0   0   0   0   0   0
 Beta via average

          1, NO =     3, NV =     7, INA =   209
 ****************************************

    I   COEFFICIENT           ORDER    EXPONENTS
       NO =      3       NV =      7
   0    13.57348336645041        0.000000000000000        0   0   0   0   0   0   0
   1    25.47460380089214        0.000000000000000        0   0   0   0   1   0   0
   1   -7.011940286158588        0.000000000000000        0   0   0   0   0   1   0
   1   0.9489986322495378        0.000000000000000        0   0   0   0   0   0   1
    -4    0.000000000000000         0.000000000000000         0   0   0   0   0   0   0
```

9.6 Putting a transformation in "canonical" form

If the normal form n is free of resonance driving terms, then it commutes with any similar normal forms. This means that we can multiply a by a rotation (phase slip and/or damping depending on the case).

$$n = a^{-1} \circ m \circ a \Rightarrow n = r^{-1} \circ a^{-1} \circ m \circ \underbrace{a \circ r}_{a_c} \tag{73}$$

In accelerator physics, we often desire a special form for the canonical transformation a. This special form is of special significance in the linear case, i.e., the map a_1 of (§9.5). Namely, we require that the matrix of a_c, denoted by A, obeys:

$$A_{12} = A_{34} = 0 \text{ and } A_{11}, A_{33} > 0 \tag{74}$$

If we select the transformation a_c according to Eq. (74), then the phase difference of two position monitors will be the Hamiltonian phase advance. There is no known extension of this property[8] to the nonlinear problem: FPP, for the nonlinear part, selects a using an arbitrary construction explained in reference [6].

[8] It should be possible to investigate this in 1-d-f without too much trouble.

Obviously if the longitudinal plane is oscillating, the same condition can be applied : $A_{56} = 0$. In the case of a phase slip, we do something else (See routine . **extract_a0** in **Ci_tpsa.f90** or section 7.3 of reference [6]).

We refer to Eq. (74) as the Courant-Snyder(-Teng-Edwards)'s condition. As for the other planes, spin and nonlinear effects, our choice is, at present, arbitrary (see reference [6]).

There are two calls to "canonize": a nonlinear routine and a faster linear routine for standard usage.

9.6.1 Nonlinear Call

The full call is given by:

```
type(c_damap) a0,as,a1,a2,a_cs,rot,a_tracked
type(c_taylor) phase(3),phase_spin
        .
        .
        .

!  Usual computation of a Taylor maps within TC
ray_TC=x0    !  For TC
identity_AP=1
ray_8_TC=ray_TC + identity_AP ! Connect AP with TC: identity added

p=>ring%start
call propagate(ray_8_TC,bmad_state,fibre1=p) !  TC of PTC proper

one_turn_map_AP=ray_8_TC


! compute 2 tunes and the phase slip
call c_normal(one_turn_map_AP,normal_form,phase=phase,dospin=bmad_state%spin) ! #1

write(6,*) "Transverse tunes "
write(6,"(3(1x,g17.10,1x))") normal_form%tune(1:c_%nd)




!!!! Canonize  !!!!
  phase(1)=0.d0
  phase(2)=0.d0
  phase(3)=0.d0
phase_spin=0.d0

call c_full_canonise(normal_form%atot,a_cs)  !  initial A ! #2
!  Usual computation of a Taylor maps within TC
ray_TC=x0    !  For TC
ray_8_TC=ray_TC + a_cs ! Connect AP with TC: identity added

p=>ring%start
call propagate(ray_8_TC,+bmad_state,fibre1=p,fibre2=p2) !  TC of PTC proper ! #3a
a_tracked=ray_8_TC

        .
```

```
        .
        .
call c_full_canonise(a_tracked,a_cs,as=as,a0=a0,a1=a1,a2=a2,rotation=rot  & ! #3b
,phase=phase,nu_spin=phase_spin)
```

The map **a_tracked** is now:

```
a_tracked=as*a0*a1*a2*rot=a_cs*rot
```

In the above code fragment, which is in **z_track_normal_fact.f90**, there three important steps.

1. At # 1, the one-turn map is normalized. In FPP, the normalization map is **normal_form%atot**. No special attention if given to that map except that it produces a normal form.
2. At # 2, the map is put in a special form denoted **a_cs**. In the linear case, it is given by Eq. (74).
3. At # 3a, the canonical transformation **a_cs** is tracked by the code PTC from **fibre1** to **fibre2**. It is extremely important to realize that what PTC is in detail, what a fibre is in detail is irrelevant. The code PTC could be full of bugs, full of errors of physics, etc... This particular call works for any tracking code— in fact people reading this manual should try writing a little code and convince themselves.
4. At # 3b, the object tracked by the code PTC, after having been converted into a **c_damap**, is now rotated into the form of Eq. (74). The angles of that rotation, here **rot**, are the phase advances (or time slip) between the positions represented by **fibre1** and **fibre2**. They are stored in **phase(1:3)** and **phase_spin**.

9.6.2 Linear fast call phase_spin and type c_linear_map

For a typical calculation, we want to be fast. For this purpuse we have a type **c_linear_map**.

```
type c_linear_map
 complex(dp) mat(6,6)          ! orbital
 complex(dp)  q(0:3,0:6)       ! spin
end type c_linear_map
```

The field **c_linear_map%q** contains the four quaternions components in **q(0:3,0)**. The orbital dependence is in **q(0:3,1:6)**.

```


call c_fast_canonise(normal_form%atot,a_cs,dospin=bmad_state%spin)

 p2=>ring%start
 call move_to(ring,p2,"SF",reset=.true.)   !  Locating SF in TC


 !  Usual computation of a Taylor maps within TC
ray_TC=x0   !  For TC
```

```
ray_8_TC=ray_TC + a_cs ! Connect AP with TC: identity added

p=>ring%start

call propagate(ray_8_TC,bmad_state,fibre1=p,fibre2=p2) !  TC of PTC proper

 a_tracked=ray_8_TC


call c_fast_canonise(a_tracked,a_cs,phase=pha,damping=damp,q_cs=q_cs, &
q_as=q_as,q_orb=q_orb,q_rot=q_rot,spin_tune=spin_tune ,dospin=bmad_state%spin)

end type c_linear_map
```

10 more...

10.1 Overloaded Operators for Taylor and Complex Taylor Types

Overloaded operators for **taylor** and **complextaylor** types include the standard functions like **sin** and **tanh** as well as the operators used in arithmetic expressions +, -, *, /, **.

11 Real_8 Type

FPP defines the **taylor** type (§7.1.1) which holds a Taylor series. Since computations with Taylor series would soon exhaust all the memory of Berz's package, FPP defines a "polymorphic" type called **real_8**. Indeed, in PTC, almost anything can become a Taylor series including multipole strengths, wiggler profiles, voltages, frequencies, etc... This would totally overload the memory of the TPSA package for any lattice with the exception of tiny ones. [9] In general, a "polymorphic" variable is a variable that can act in different ways depending on the context of the code. In this case, a **real_8** variable can act as if it were a real number or it can act as if it were a Taylor series depending upon how it is initialized. An example program will make this clear.

```
program real_8_example
use pointer_lattice   ! Read in structure definitions, etc.
implicit none

type (real_8) r8      ! Define a real_8 variable named r8
real(dp) x            ! Define a double precision number

!

nice_taylor_print = .true.    ! Nicely formatted "call print" output
call init (only_2d0, 3, 0)    ! Initialize: #Vars = 2, Order = 3

call alloc(r8)          ! Initialize memory for r8

x = 0.1d0
r8 = x                  ! Init r8 to a real => r8 will act as a real.
print "(/,a)", "r8 is now acting as a real:"
call print (r8)         ! Will print a real number.

r8 = 0.7d0 + dz_8(1) + 2*dz_8(2)**3   ! Init r8 as a Taylor series
print "(/,a)", "r8 is now acting as a Taylor series:"
call print(r8)                        ! Will print a Taylor series.

r8 = r8**4  ! Raise the Taylor series to the 4th power
print "(/,a)", "This is r8^4:"
call print (r8)

call kill(r8)
end program
```

The variable **x** is defined as a double precision real number. The line

```
type (real_8) r8
```

defines **r8** as an instance of a **real_8** variable and the line

```
call alloc(r8)
```

[9] The "8" here is an allusion to the fact that double precision numbers are generally represented by 8 bytes and indeed the **dp** parameter defined by FPP/PTC to designate double precision numbers has, on most systems, a value of 8.

initializes **r8**. This initialization must be done before **r8** is used. After **r8** is used, any memory that has been allocated for use with **r8** is reclaimed by calling the **kill** routine.

```
call kill(r8)
```

This illustrates a general rule: All calls to **alloc** must have a corresponding **kill**.[10] When **r8** is set to the real number **x** in the line[11]

```
  r8 = x
```

This initialization of **r8** will cause **r8** to act as a real number. This is verified by printing the value of **r8** in the lines

```
print '(/,a)', "r8 is now acting as a real:"
call print (r8)
```

The output is just a single real number indicating that **r8** is acting as a real:

```
r8 is now acting as a real:"
0.100000000000000
```

Notice that the **print** statement uses the Fortran intrinsic print function while the **call print** statement uses the overloaded print subroutine defined by **FPP**.

When **r8** is set to a Taylor series in the line

```
r8 = 0.7d0 + dz_8(1) + 2*dz_8(2)**3 ! Init r8 as a Taylor series
```

this will cause **r8** to act as a Taylor series. To understand how this initialization works, first consider the initialization of FPP/PTC which was done by the line

```
call init (only_2d0, 3, 0)  ! Initialize FPP/PTC. #Vars = 2, Order = 3
```

The first argument, **only_2d0**, is a parameter defined by **PTC** of type **internal_state** (§14).[12] When **only_2d0** is used as the first argument to **init**, the number of variables will be two which is appropriate for simulations involving motion along one axis (typically involving phase space (x, p_x)).[13] The second argument, **3**, gives the order at which the Taylor series is truncated to.

[10] Strictly speaking, **kill** is not necessary here since memory cleanup is automatically done at the end of the program. However, in a subroutine or function, all local instances of **real_8** variables must be killed otherwise there will be a memory leak.

[11] All sets like this where the variable type on the LHS is different from the RHS is, by necessity, done with an overloaded equal sign.

[12] When an **internal_state** type is used as the first argument in the overloaded routine **init**, both **FPP** and **PTC** will be initialized.

[13] There is also an **only_4d0** parameter for configuring using 4 variables for simulations with transverse phase space (x, p_x, y, p_y). However, there is no **only_6d0** parameter since configuring for the full 6D phase space is a bit more complicated (involving consideration like whether there are powered RF cavities or not). If only **FPP** needs to be initialized (as is the case at hand), the initialization here could have been done in the above example via:

```
call init (3, 1,np1=0) ! Init just FPP. Order = 3, #Vars = 2, degrees of freedom = 1
```

or

```
call init (3, 2) ! Init just FPP. Order = 3, #Vars = 2, degrees of freedom = not needed
```

Notice that here the order comes before the number of variables which is the reverse of the order when **init** is called with an **internal_state** as the first argument. In the first instance, the analysis AP is aware than a Taylor map in two variables can be normalized. In the second instance maps can be created but they are generic without any normal form as far as AP is concerned.

That is, after this initialization, all Taylor series t will be of the form:

$$t = \sum_{i,j}^{0 \le i+j \le 3} C_{ij}\, z_1^i\, z_2^j \tag{75}$$

where z_1 is the first variable and z_2 is the second variable. **FPP** sets up a **real_8** array named **dz_8** such that **dz_8(N)** represents the N^{th} variable.[14] Thus in the above code **r8** is initialized to the Taylor series:

$$t = 0.7 + z_1 + 2\, z_2^3 \tag{76}$$

This is confirmed by printing **r8** after it has been set via the lines

```
print '(/,a)', "r8 is now acting as a Taylor series:"
call print(r8)
```

The output is:

```
r8 is now acting as a Taylor series:
Out  Order  Coef                        Exponents
-------------------------------------------------------------
         0  0.7000000000000000        0  0
         1   1.000000000000000        1  0
         3   2.000000000000000        0  3
```

Each line in the above output, after the line with dashes, represents one term in the Taylor series. The general form for printing a Taylor term is:

```
<output-index> <order>  <coef>  <z1-exponent>  <z2-exponent>, ...
```

The <**output-index**> is the index of the output variable when there is an array of variables. Here, since **r8** is not an array, the <**output-index** column is blank. The <**order**> is the order of the term. That is, the sum of the exponents. For example, the last line in the above printout is

```
         3   2.000000000000000        0  3
```

This line represents the term $2\, z_1^0\, z_2^3$ which is order 3. The <**coef**> column is the term coefficient and <**z1-exponent**>, and <**z2-exponent**> collums are the exponents z_1 and z_2 in the term. The number of exponent columns will be equal to the number of variables.

Once **r8** has been initialized, it can be used in expressions. Thus the line

```
r8 = r8**4  ! Raise the Taylor series to the 4th power
```

raises **r8** to the 4th power and puts the result back into **r8**. This is confirmed by the final print which produces

```
This is r8^4:
Out  Order  Coef                        Exponents
-------------------------------------------------------------
         0  0.2400999999999999        0  0
```

[14] More accurately, **dz_8(N)** is the Tayor series $t = z_i$.

```
          1    1.372000000000000        1   0
          2    2.940000000000000        2   0
          3    2.800000000000000        3   0
          3    2.743999999999999        0   3
```

Notice that the map has been truncated so that no term has an order higher than 3 as expected. Expressions using **real_8** variables involve overloaded operators as discussed in section §**??**.

11.1 Real_8 Under the Hood

The particulars of how the **real_8** structure is defined are generally not of interest to the general user. However, it is instructive to take a quick look. In the **FPP** code the **real_8** structure is defined as:

```
type real_8
   type (taylor) t    ! Used if taylor
   real(dp) r         ! Used if real
   integer kind       ! 0,1,2,3 (1=real,2=taylor,3=taylor knob)
   integer i          ! Used for knobs and special kind=0
   real(dp) s         ! Scaling for knobs and special kind=0
   logical(lp) :: alloc  ! True if taylor is allocated in da-package
end type real_8
```

The **t** component of the structure is of type **taylor** (§7.1.1) and is used if a **real_8** variable is acting as a Taylor series. The **r** component is used if a **real_8** variable is acting as a real number. The **kind** component is an integer that sets the behavior of a **real_8** variable. Besides behaving as **real** or a Taylor series, a **real_8** variable may behave as a "**knob**" which will be explained later. The reason for hiding a Taylor series under the hood is to defer the decision of its use to run time.

This is useful when tracking since manipulating Taylor series is computationally more expensive than using real numbers.

12 Complex_8 Type

The type **complex_8** is the polymorphic version of the **complextaylor** type just as the **real_8** type is the polymorphic version of the **taylor** type.

The type **complex_8** is rarely used in a tracking code since all quantities we compute are ultimately real. However once in a while it is useful to go into complex coordinates temporarily. The **complex_8** type in useful in several cases. For example, when fields are expressed in cylindrical coordinates. In such a case, if $\mathbf{z} = (x, p_x, y, p_y)$, then most intermediate calculations involve a quantity $q = x + i\,y$. and the complex polymorph is useful.

The definition of **complextaylor** is:

```
type complex_8
  type (complextaylor) t
  complex(dp) r
  logical(lp) alloc
  integer kind
  integer i,j
  complex(dp) s
end type complex_8
```

As in the case of the real polymorph, the **t** component contains the complex Taylor series and the **r** component contains the complex number if the polymorph is not a Taylor series.

13 Real_8 and Complex_8 Functions and Operators

Operators that act on **real_8** and **complex_8** types:

```
exp(t)                      ! Exponentiation
log(t)                      ! Log
sin(t), cos(t), tan(t)      ! Trig functions
asin(t), acos(t), atan(t) ! Inverse trig functions
sinh(t), cosh(t), tanh(t) ! Hyperbolic functions
atan2(ty, tx)
sinx_x(t)
sinhx_x(t)
abs(t)
full_abs(t)???? What is this
dble ???? is this the same as real?
real(ct), aimag(ct)
cmplx(t_re, t_im)
```

Question: Do all functions act on real_8, taylor, complex_8, complextaylor?

Include dz_8

14 Internal_State Type

Components of the **internal_state** structure define parameters that affect such things as whether RF cavities are considered to be on or off or how phase space variables are treated. The structure definition is:

```
type internal_state
   integer totalpath      ! T => total time or path length is used
   logical(lp) time       ! T => Time is used instead of path length
   logical(lp) radiation  ! T => Radiation is turned on
   logical(lp) nocavity   ! T => Cavity is turned into a drift
   logical(lp) fringe     ! T => Fringe fields on? (mainly for quadrupoles)
   logical(lp) stochastic ! T => Random Stochastic kicks to x(5)
   logical(lp) envelope   ! T => Stochastic envelope terms tracked in probe_8
   logical(lp) para_in    ! T => Parameters in the map are included
   logical(lp) only_4d    ! T => Real_8 Taylor in (x,p_x,y,p_y)
   logical(lp) delta      ! T => Real_8 Taylor in (x,p_x,y,p_y,delta)
   logical(lp) spin       ! T => spin is tracked
   logical(lp) modulation ! T => One modulated family tracked by probe
   logical(lp) only_2d    ! T => Real_8 taylor in (x,p_x)
   logical(lp) full_way   !
end type internal_state
```

For each structure component except **param_in** and **full_way**, there is a global parameter defined

Explain Bmad units

In PTC, when "**time**" units are being used (the **time** component is set true), the orbital phase space is:

$$x(1:6) = \left(x, \frac{p_x}{p_0}, y, \frac{p_y}{p_0}, \frac{\Delta E}{p_0 c}, c\,T \text{ or } c\,\Delta T \right) \tag{77}$$

where cT is used for $x(6)$ if the **totalpath** component is set True, and $c\Delta T$ is used for $x(6)$ if **totalpath** is set False.

When **Bmad** units are used, the orbital phase space is:

$$x(1:6) = \left(x, \frac{p_x}{p_0}, y, \frac{p_x}{p_0}, -\beta c T \text{ or } -\beta\, c\, \Delta T, \frac{\Delta p}{p_0} \right) \tag{78}$$

where $-\beta cT$ is used for $x(5)$ if the **totalpath** component is set True, and $-\beta c\Delta T$ is used for $x(5)$ if **totalpath** is set False.

With 1-d-f tracking, only the first two phase space coordinates are used. With 2-d-f, only the first four phase space coordinates are used.

where

$$\Delta T = T - T_{ref} \tag{79}$$

15 Other Polymorphic Types: spinor_8, quaternion_8, and rf_phasor_8

The types discussed above are useful for anyone who decides to use FPP to write their own tracking code. In fact there is nothing "tracking" about these types. One could write a code to solve a problem in finance or biology using **real_8**. However, since our ultimate goal is to describe the analysis part of FPP, we need to say a little bit more about some of the structures used in PTC. Here we introduce three.

The **spinor** type is the represents a particle's spin in (x, y, z) coordinates:

```
type spinor
  real(dp) x(3)  ! x(3) = (s_x, s_y, s_z)   with  |s|=1
end type spinor
```

and the **spinor_8** is the polymorphic equivalent:

```
type spinor_8
  type(real_8) x(3)  ! x(3) = (s_x, s_y, s_z)   with  |s|=1
end type spinor_8
```

The **quaternion** type is used to store the quaternion representing spin transport:

```
type  quaternion
  real(dp) x(0:3)
end type quaternion
and the \vn{quaternion_8} is the polymorphic equivalent:
\begin{code}
type  quaternion_8
  type(real_8) x(0:3)
end type quaternion_8
```

The **rf_phasor** type is used for RF phase modulation:

```
type rf_phasor
   real(dp) x(2)
   real(dp) om
   real(dp) t
end type rf_phasor
```

and the corresponding **rf_phasor_8** polymorphic equivalent is:

```
type rf_phasor_8
  type(real_8) x(2)   ! The two hands of the clock
  type(real_8) om     ! the omega of the modulation
  real(dp) t          ! the pseudo-time
end type rf_phasor_8
```

These structures are used as components of the **probe** and **probe_8** types (§16) that are used for tracking.

16 Probe and Probe_8 Types

The **proble** and **probe_8** types are used for tracking in **PTC**. The **probe** structure looks like:

```
type probe
  real(dp) x(6)
  type(spinor) s(3)
  type(quaternion) q
  type(rf_phasor)  ac(nacmax)
  integer:: nac=0
    ...
end type probe
```

And **probe_8** is the polymorphic equivalent (§3.5):

```
type probe_8
  type(real_8) x(6)              ! Polymorphic orbital ray
  type(spinor_8) s(3)            ! Polymorphic spin s(1:3)
  type(quaternion_8) q           ! Spin transport quaternion
  type(rf_phasor_8)  ac(nacmax)  ! Modulation of magnet
  integer:: nac=0                ! Number of modulated clocks <= nacmax
  real(dp) E_ij(6,6)             ! Envelope for stochastic radiation
  real(dp) x0(6)                 ! Initial ray for TPSA calc with c_damap
  ....
end type probe_8
```

That is, the **x**, **s**, **q** and **ac** components of the **probe** structure are replaced in the **probe_8** structure with their polymorphic analogues. Since **probe_8** can do everythong that **probe** does, why bother to define the **probe** structure? The reason is speed as will be discussed below.

The way PTC uses **probe** and **probe_8** is that for a given a PTC tracking routine that uses **probe_8** there is a duplicate tracking routine that uses **probe**. The **probe_8** routine is to be used when tracking is to be done using Taylor maps and the **probe** routine is to be used for tracking ordinary real rays. **Probe** is equivalent to tracking **probe_8** setting the truncation order to 0.

The same routine duplication happens with routines that use **real_8**. In this case the corresponding routine will use a **real(dp)** variable. As an example, consider the subroutine in the example of §17:

```
subroutine trackp(z)
implicit none
type(real_8) :: z(2)
z(1) = z(1) + L*z(2)
z(2) = z(2) - B - K_q*z(1) - K_s*z(1)**2
end subroutine track
```

This routines computes the effect of a "drift" followed by a multipole "kick" in the jargon of accelerator physicinsts. The corresponding **real(dp)** version is:

```
subroutine trackr(z)
implicit none
```

```
real(dp) :: z(2)             <-------- instead of type(real_8) :: z(2)
z(1)=z(1)+L*z(2)
z(2)=z(2)-B-K_q*z(1)-K_s*z(1)**2
end subroutine track
```

The naming of the two routines **trackp** and **trackr** follow a common PTC convention that routines that involve polymorphic types have a name ending in **p** and corresponding non-polymorphic routines have a name ending in **r**.

16.1 The x(6) component

The **x(6)** component represents the orbital phase space part of the tracked particle.

16.2 The spin and quaternion components

PTC can track spin. There are two ways to track spin: one method uses a regular spin matrix and the other uses a quaternion. Orignally, only the matrix based method was implemented. The quaternion representation was subsiquently implemented since it is a more efficient representation for the spin and it simplifies the analysis. From the theory of rotations in three dimensions, we know that there is one invariant unit direction and one angle of rotation around this axis. The unit quaternion has exactly the same freedom: four numbers whose squares add up to one. Once more we claim that if its polymorphic components are properly initialized, a generic Taylor map for the quaternions emerges. This is described in §**??**.

For the spin matrix representation, Since there are three independent directions of spin, PTC tracks three directions: this saves time if one wants to construct a spin matrix. The three directions are represented by the three **s(3)** components which are of type **spinor_8**:

```
type spinor_8
  type(real_8) x(3)  ! x(3) = (s_x, s_y, s_z)   with  |s|=1
end type spinor_8
```

For example, if one tracks on the closed orbit, the initial conditions are

$$\begin{aligned} s(1) &= (1,0,0) \\ s(2) &= (0,1,0) \\ s(3) &= (0,0,1) \end{aligned} \tag{80}$$

The tracking of these vectors for one turn will allow us to construct the one-turn spin matrix around the closed orbit. Thus if the orbital polymorphs are powered to be appropriate Taylor series in n-d-f (n=1,2, or 3), we can produce a complete approximate 3 by 3 matrix for the spin. This is shown in §**??**.

16.3 The components of type rf_phasor_8

The **rf_phasor_8** type is somewhat complex to explain but its definition is simple:

```
type rf_phasor_8
  type(real_8)  x(2)  ! The two hands of the clock
  type(real_8) om     ! the omega of the modulation
  real(dp) t          ! the pseudo-time
end type rf_phasor_8
```

The variables **x(2)** represents a vector rotating at a frequency **om** based on a pseudo-time related to the reference time of the "design" particle. As the the **probe_8** traverses a magnet, in the integration routines, magnets can use that pseudo-clock to modulate their multipole components. In the end, as we will see, the components **rf_phasor_8%x(2)** are used to add two additional dimensions to a Taylor map. This will be explained later when we discuss the types germane to analysis.

16.4 The real components E_ij(6,6) and equilibrium moments

The **E_ij(6,6)** structure allows us to store the quantum fluctuations due to radiation. PTC, does not attempt to go beyond linear dynamics when dealing with photon fluctuations. When radiation is present, the **x(6)** polymorphic component of **probe_8** contain the closed orbit (with classical radiation) and the **E_ij(6,6)** component stores the fluctuations $< x_i\, x_j >\ (i,j = 1,6)$ due to photon emission. Notice that **E_ij** is not polymorphic. Radiation fluctuations are always approximated to zeroth order around the reference orbit.

If a linear matrix M is extracted from **probe_8**, then the one-turn linear map for moments is given by:

$$\Sigma_{\text{final}} = M\,(\Sigma_{\text{initial}} + E)\,M^{\text{T}} \tag{81}$$

where superscript **T** indicates transpose and

$$\Sigma_{ij} = \left\langle x_i x_j \right\rangle, \qquad i,j = 1,\ldots 6. \tag{82}$$

However when a **probe_8** is converted into a **c_damap** with the syntax **c_damap** = **probe_8**, then E is redefined as:

$$\Sigma_{\text{final}} = M\left(\Sigma_{\text{initial}} + E_{\text{probe\_8}}\right) M^{\text{T}} = M\,\Sigma_{\text{initial}} M^{\text{T}} + E_{\text{c\_damap}}. \tag{83}$$

Thus, in FPP, the equation from the moments is:

$$\Sigma_{\text{final}} = \ M\,\Sigma_{\text{initial}} M^{\text{T}} + E, \tag{84}$$

where E has been redefined.

To get the equilibrium moments, we can first diagonalized the matrix M:

$$M = B\Lambda B^{-1} \quad \text{where}$$
$$\Lambda = \begin{pmatrix} \lambda_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \lambda_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & \lambda_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & \lambda_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & \lambda_6 \end{pmatrix} \quad \text{and} \quad \begin{array}{l} \lambda_1 = \exp\left(-\alpha_1 - i2\pi\mu_1\right) \\ \lambda_2 = \exp\left(-\alpha_1 + i2\pi\mu_1\right) \\ \lambda_3 = \exp\left(-\alpha_2 - i2\pi\mu_2\right) \\ \lambda_4 = \exp\left(-\alpha_2 + i2\pi\mu_2\right) \\ \lambda_5 = \exp\left(-\alpha_3 - i2\pi\mu_3\right) \\ \lambda_6 = \exp\left(-\alpha_3 + i2\pi\mu_3\right) \end{array} . \tag{85}$$

We can apply the transformation B on Eq. (84):

$$\begin{aligned} B\Sigma_{\text{final}}B^{\text{T}} &= BM\Sigma_{\text{initial}}M^{\text{T}}B^{\text{T}} + BEB^{\text{T}} \\ \sigma_{\text{final}} &= \Lambda\sigma\Lambda + \varepsilon \end{aligned} \tag{86}$$

And since Λ is diagonal, we can easily get the equilibrium σ^{inf} in this basis:

$$\begin{aligned} \sigma^{\infty} &= \Lambda\sigma^{\infty}\Lambda + \varepsilon \\ &\Downarrow \\ \sigma^{\infty}_{ij} &= \frac{\varepsilon_{ij}}{1 - \lambda_i\lambda_j} \end{aligned} \tag{87}$$

The terms σ_{12},σ_{34} and σ_{56} correspond to the so-called equilibrium emittances and they dominate when damping is small and when the map is far from linear resonances. For example, the horizontal emittance is

$$\sigma^{\infty}_{12} = \frac{\varepsilon_{12}}{1 - e^{-2\alpha_1}} \approx \frac{\varepsilon_{12}}{2\alpha_1}. \tag{88}$$

ε_{12} is called, in accelerator jargon, the horizontal H-function. It is the fluctuatuation of the horizontal invariant summed over the entire machine. The final beam sizes are given by:

$$\Sigma^{\infty} = B^{-1}\sigma^{\infty}B^{-1\,\text{T}}. \tag{89}$$

In §**??** we explain how one can use E in stochastic tracking.

16.5 Real(dp) type probe specific to PTC

In theory, it is possible to have a code which always uses the polymorphs **real_8** and nothing else. However this is not what PTC does due to computational speed considerations. To see this consider the following code fragment

```
type(real_8) a,b,c
     .
     .
c=a+b
```

How many internal questions does the + operation requires? First it must decide if **a** is real, Taylor or knob? The same thing applies to the polymorph **b**. On the basis of the answer, it must branch into 9 possibilities before it can even start to compute this sum. This overhead slows down a polymorphic calculation even if all the variables are real. To get around this, PTC has a type **probe**...

As we said, PTC tracks **probe** or **probe_8**. It is very easy to modify the above routines to mimic this feature of PTC. This is done in the module **my_code** in the file **z_my_code.f90**. Notice that the cell is repeated **nlat** times which is defaulted to 4:

```
module my_code
use tree_element_module
implicit none
private trackr,trackp
type(real_8)  :: L ,B, K_q , K_s
real(dp) :: L0 , B0, K_q0 , K_s0
real(dp) par(4)
integer ip(4)
integer :: nlat = 4

interface track
  module procedure trackr
  module procedure trackp
end interface

contains
      .
      .
      .
    subroutine trackr(p) ! for probe
    implicit none
    type(probe) :: p
    integer i
     do i=1,nlat
     p%x(1)=p%x(1)+L0*p%x(2)
     p%x(2)=p%x(2)-B0-K_q0*p%x(1)-K_s0*p%x(1)**2
     enddo
    end subroutine trackr

    subroutine trackp(p) ! for probe_8
    implicit none
    type(probe_8) :: p
    integer i
     do i=1,nlat
     p%x(1)=p%x(1)+L*p%x(2)
     p%x(2)=p%x(2)-B-K_q*p%x(1)-K_s*p%x(1)**2
     enddo
    end subroutine trackp
```

```
          .
          .
          .
end module my_code
```

Then a call to **track(p)** will either call

- **trackr(p)** if **p** is a **probe**
- or **trackp(p)** if **p** is a **probe_8**

If we call **track(p)** where **p** is a **probe_8**, then the resulting **p** could be a Taylor series which approximates the true map[15] of the code.

For example, in §17, we got the following results for the final polymorphs:

```
Properties, NO =    2, NV =    2, INA =   20
*******************************************

  1    1.0000000000000000        1   0
  1    1.0000000000000000        0   1


Properties, NO =    2, NV =    2, INA =   21
*******************************************

  1 -0.1000000000000000        1   0
  1  0.9000000000000000        0   1
```

It is clear that one could deduce from the above result:

$$\mathbf{z} = \begin{pmatrix} 1 & 1 \\ -0.1 & 0.9 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} + O\left(z^2 \right) \tag{90}$$

Therefore we could say that **z** or equivalently a **probe_8** is a "Taylor map." But this is not done in PTC for several reasons:

1. The variables (z_1, z_2) could be infinitesimal with respect to machine parameters, in which case any attempt to concatenate the matrix is pure nonsense.
2. One should not confuse a set with an algebraic structure which the set itself.

Item 2 requires an explanation. Take for example a pair of real numbers from the set $\mathbb{R} \times \mathbb{R}$. A priori we have no idea what structures are imposed on this pair of numbers. Indeed the structure could be a complex number field, a ring of differentials (running TPSA to order 1 with one parameter), a one-dimensional complex vector space, a two dimensional real vector space, a twice infinite dimensional vector space on the field of rationals, etc... In a code (or in a

[15] The true map of the code is always what you get by calling **track**.

mathematical article), we could decide to distinguish these structures by using a different "plus" sign depending on the structure: $+$ if complex numbers and say a $\oplus$ if they are vectors.

If the object in FPP is extremely important, the solution in FPP is to define a new type and keep the $+, *, \ldots$ signs for this new type. Therefore we do not allow the concatenation of **probe_8** even when it is reasonable. Instead we construct a map, type **c_damap** described in §8, only if this construction is meaningful. FPP does not prevent the construction of meaningless Taylor maps. The **c_damap** of PTC will be meaningful if the rules between the **probe_8** and **c_damap** types are religiously[16] observed.

Conversely we define a new operator when the creation of a new type is too cumbersome due to its infrequent usage. We do this on **c_damap** allowing "DA" concatenation and "TPSA" concatenation via a different symbol rather than a different type. This will be explained in §**??**.

[16] Neither FPP, nor PTC nor Bmad prevents a user to do crazy things and shove a **probe_8** into a **c_damap** anyway he sees fit. But beware of the results.

17 Knobs

This type is the most important type if you write a tracking code of respectable length. Imagine that your code tracks in one degree of freedom (1-d-f). Then you will push two phase space variables through your magnets, let us call them $\mathbf{z} = (z_1, z_2)$. These variables will denote the position and the tangent of an angle in our little example. If it is your intention to always extract a Taylor series around a special orbit, then it would suffice to declare as **taylor** only the phase space variables $\mathbf{z} = (z_1, z_2)$ and any temporary variables the code might used during its calculations.

But what if we want to have a Taylor map that also depends upon some parameter or parameters of the lattice. For example, a map can include quadrupole strengths as independent variables in the maps. Such variables are called "**knobs**." Since this is a user decision, it is best if the code decides at execution time using the type **real_8**. As an example, consider the code **z_why_polymorphism.f90**:

```
program my_small_code_real_8
use polymorphic_complextaylor
implicit none
type(real_8) :: z(2)
real(dp) :: z0(2) = [0, 0]  ! special orbit
type(reaL_8)  :: L , B, K_q , K_s
integer :: nd = 1 , no = 2 , np = 0, ip
longprint = .false.         ! Shorten "call print" output
! nd = number of degrees of freedom
! no =  order of Taylor series
! Number of extra variables beyond 2*nd

call alloc(z)
call alloc( L , B, K_q , K_s )
np=0
print * , "Give  L and parameter ordinality (0 if not a parameter)"
read(5,*) L%r , ip
np=np+ip
call make_it_knob(L,ip);  np=np+ip;
print * , "Give  B  and parameter ordinality (0 if not a parameter)"
read(5,*) K_q%r , ip
print * , "Give  K_q and parameter ordinality (0 if not a parameter)"
read(5,*) K_q%r , ip
call make_it_knob(K_q,ip);  np=np+ip;
print * , "Give  K_s and parameter ordinality (0 if not a parameter)"
read(5,*) K_s%r , ip
call make_it_knob(K_s,ip); np=np+ip;
print * , "The order of the Taylor series ?"
read(5,*) no

call init(no,nd,np) ! Initializes TPSA

z(1)=z0(1) + dz_8(1) ! <--- Taylor monomial z_1 added
z(2)=z0(2) + dz_8(2) ! <--- Taylor monomial z_2 added
```

```
call track(z)

call print(z)

contains

subroutine track(z)
implicit none
type(real_8) :: z(2)
 z(1)=z(1)+L*z(2)
 z(2)=z(2)-B-K_q*z(1)-K_s*z(1)**2
end subroutine track

end program my_small_code_real_8
```

In this little code, there is one drift of length **L** followed by a multipole kick that contains a dipole of strength **B**, a quadrupole of strength **K_q** and a sextupole of strength **K_s**. We run the code ignoring the parameters:

```
 Give  L and parameter ordinality (0 if not a parameter)
1 0
 Give  B  and parameter ordinality (0 if not a parameter)
0 0
 Give  K_q and parameter ordinality (0 if not a parameter)
.1 0
 Give  K_s and parameter ordinality (0 if not a parameter)
0 0
 The order of the Taylor series ?
2

 Properties, NO =    2, NV =    2, INA =   20
 *********************************************

   1   1.000000000000000        1  0
   1   1.000000000000000        0  1


 Properties, NO =    2, NV =    2, INA =   21
 *********************************************

   1 -0.1000000000000000        1  0
   1  0.9000000000000000        0  1
```

This little program produces Taylor series to second order in the phase space variables $\mathbf{z} = (z_1, z_2)$ similar to the programs **Transport** and **Marylie**. However, we can now require that the multipole strengths be variables of the Taylor series without recompiling the program. In this example, we make the quadrupole strength the third variable of TPSA: $K_q = 0.1 + dz_3$.

```
 Give  L and parameter ordinality (0 if not a parameter)
1 0
```

```
 Give  B  and parameter ordinality (0 if not a parameter)
0 0
 Give  K_q and parameter ordinality (0 if not a parameter)
.1 1
 Give  K_s and parameter ordinality (0 if not a parameter)
0 0
 The order of the Taylor series ?
2

 Properties, NO =    2, NV =    3, INA =   22
 *******************************************

   1   1.000000000000000        1  0  0
   1   1.000000000000000        0  1  0


 Properties, NO =    2, NV =    3, INA =   23
 *******************************************

   1 -0.1000000000000000        1  0  0
   1  0.9000000000000000        0  1  0
   2  -1.000000000000000        1  0  1
   2  -1.000000000000000        0  1  1
```

Again, we must emphasize that while it would have been easy here to use the type **taylor** for all the variables, it is totally unfeasible in a real tracking code to either recompile the code or allow all parameters of the systems to be Taylor series. This is why typical matrix[17] codes, not using TPSA, are limited to a small set of Taylor variables, usually the six phase space variables.

So in summary a **real_8** polymorph can be as mentioned in §11.1:

1. A real number
2. a Taylor series with real coefficients (**taylor**)
3. a knob which is a simple temporary Taylor series activated only if needed

[17] This is not true of Berz's COSY INFINITY which handles variable memory of TPSA within its own internal language.

18 Manual To Do List

Note: calling init again wipes out existing Taylor series. (Does is wipe out universial taylor?)

Explain knobs

References

[1] M. Berz. *Modern Map Methods in Particle Beam Physics*. San Diego, San Francisco, New York, Boston, London, Sydney, Tokyo: Academic Press, 1999.

[2] M. Berz. "The method of power series tracking for the mathematical description of beam dynamics". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 258.3 (1987), pp. 431–436. ISSN: 0168-9002. DOI: https://doi.org/10.1016/0168-9002(87)90927-2. URL: https://www.sciencedirect.com/science/article/pii/0168900287909272.

[3] M. Berz, H. C. Hoffmann, and H. Wollnik. In: *Nucl. Instr. and Meth.* A258 (1987), p. 402.

[4] K. L. Brown. *A First and Second Order Matrix Theory for the Design of Beam Transport Systems and Charged Particle Spectrometers*. Tech. rep. SLAC Report 75. SLAC, June 1982.

[5] E. Forest. *Beam Dynamics: A New Attitude and Framework*. Amsterdam, The Netherlands: Harwood Academic Publishers, 1997.

[6] E. Forest. *From Tracking Code to Analysis: Generalised Courant-Snyder Theory for Any Accelerator Model*. Tokyo, Japan: Springer, 2016.

[7] E. Forest. "Geometric integration for particle accelerators". In: *J. Phys. A: Math. Gen.* 39 (2006), pp. 5321–5377.

[8] F. Christoph Iselin. "The MAD program (Methodical Accelerator Design)". In: *CERN/SL/92* (1992).

[9] K. Makino and M. Berz. In: *Nucl. Instr. and Meth.* A558 (2005), pp. 346–350.

[10] K. Makino and M. Berz. *COSY INFINITY Version 9.1*. Tech. rep. Michigan State University, June 2004, pp. 346–350.

[11] D. Sagan. "Bmad: A relativistic charged particle simulation library". In: *Nuc. Instrum. and Methods in Phys Research A* 558 (2006). http://www.lepp.cornell.edu/~dcs/bmad, pp. 356–359.

[12] Li Hua Yu. "Analysis of nonlinear dynamics by square matrix method". In: *Phys. Rev. ST Accel. Beams* 20 (2017), p. 034001.