Contents

Manual for FPP: The Fully Polymorphic Package

Étienne Forest
KEK, Japan

January 15, 2020

1 Introduction to FPP/PTC

1.1 FPP and PTC

FPP/PTC is an object oriented, open source, subroutine library for

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

FPP/PTC has two parts. The Fully Polymorphic Package (FPP) is the part that deals with Taylor series and maps. FPP is pure math independent[1] of any "physics". The Polymorphic Tracking Code (PTC) part of the library deals with the modeling of particle beams and accelerators. PTC contains the "physics" and relies on FPP for producing and manipulating Taylor maps. This is illustrated in Figure 1. Roughly, FPP can be subdivided into two parts(see Sec. (6)), a Taylor manipulation part for basic manipulations of Taylor series and an analysis part to do things like normal form analysis. PTC uses the Taylor manipulation part of FPP for things like the construction of Taylor maps. Additionally, PTC uses the analysis tools of FPP. A closer look at FPP shows the existence of a Differential Algebra (DA) package[2] within FPP. This package was originally coded by Martin Berz.

[1] It does not contain any physical description of what an accelerator is.

[2] Should really be called the TPSA package of Berz. We will demystify this jargon latter.

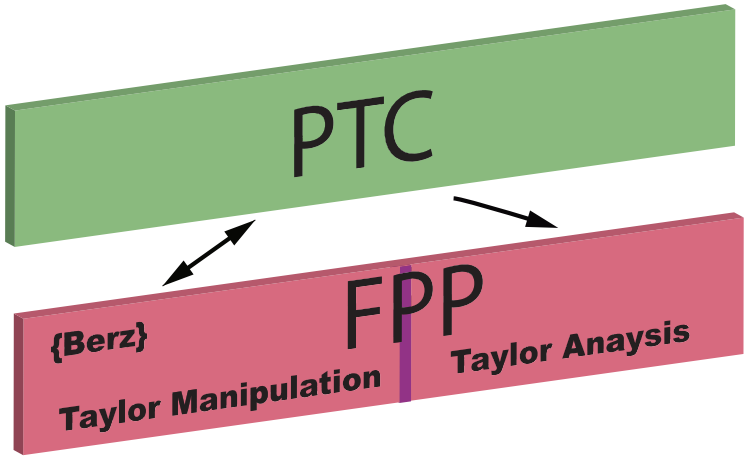

Figure 1: The Fully Polymorphic Package (FPP) part of the FPP/PTC library provides manipulation and analysis of Taylor series and maps and the Polymorphic Tracking Code (PTC) part provides the physics from which accelerators can be analyzed.

1.2 Where to Obtain FPP/PTC

1.3 Concepts

FPP/PTC is written in object oriented Fortran2008.

FPP/PTC uses double precision real numbers defined using the type "real(dp)" "dp" is defined in FPP/PTC to correspond to double precision. For example, to define in a program a real number named "time" one would write:

```
real(dp) time
```

In Fortran, a "structure" (also called a "derived type") is like a struct in C or a class in C++. A structure holds a set of components as defined by the programmer. With FPP, the "taylor" structure is used to hold a taylor series. For practical calculations it is often not convenient to deal directly with the taylor structure. For reasons that will be discussed later, the preferred structure to use is a polymorphic structure called "`real_8`". In general, a "polymorphic" variable is a variable that can act in different ways depending on the context of the program. Here, 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

!

longprint = .false.          ! Shorten "call print" output
call init (only_2d0, 3, 0)  ! Initialize: #Vars = 2, Order = 3

x = 0.1d0
call alloc(r8)          ! Initialize memory for r8
r8 = x                  ! This will make r8 act as a real

print *, 'r8 is now acting as a real:'
call print (r8)

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

r8 = r8**4  ! Raise the Taylor series to the 4th power
print *, '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)
```

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)
```

Strictly speaking, the `kill` is not necessary here since `kill` is called at 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.

When `r8` is set to the real number `x` in the line

```
r8 = x
```

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

```
print *, '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`, configured FPP/PTC to construct any Taylor series as a function of two phase space variables. These two variables will be called z_1 and z_2 here. The second argument, `3`, gives the order at which the Taylor series is truncated to. That is, after this initialization, all Taylor series t will be of the form

$$t = \sum_{i,j}^{0 \le i+j \le 3} C_{ij}\, z_1^i\, z_2^j \tag{1}$$

In the above initialization of `r8`, `dz_8(1)` represents the variable z_1 and `dz_8(2)` represents the variable z_2. Thus `r8` is initialized to the Taylor series

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

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

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

The output is:

```
r8 is now acting as a Taylor series:
Properties, NO =    3, NV =    2, INA =   21
 *****************************************

   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 the asterisks, represents one term in the Taylor series. The general form for printing a Taylor term is:

```
<term-order>   <term-coefficient>    <z1-exponent>  <z2-exponent>
```

The <term-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
```

and this line represents the term $2\, z_1^0\, z_2^3$.

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:
Properties, NO =    3, NV =    2, INA =   23
*********************************************

  0  0.2400999999999999       0  0
  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 Sec. (5).

1.4 Real_8 Under the Hood

The particulars of how the `real_8` structure is defined are generally not of interest to the general user. But 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  1 IF TAYLOR IS ALLOCATED IN DA-PACKAGE
END TYPE REAL_8
```

The `t` component of the structure is of type `taylor` 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 `real_8` structure contains a component of type `taylor`. The definition of the taylor structure is

```
TYPE TAYLOR
   INTEGER I       !  integer I is a pointer in old da-package of Berz
END TYPE TAYLOR
```

The component `i` is a pointer to the differential algebra package of Berz that is contained in FPP. The details of how a Taylor series is stored in the structure is not important here. What is important is that this structure can be used to hold a Taylor series.

The reason for hiding a Taylor series under the hood is to defer its use to running time. We really do not know in a tracking code like PTC when the user will need a Taylor series and even what the parameters of that series might be: phase space, if so how many (2,4,6,?), and parameters such as quadrupole strengths which are taken from the huge set of magnets present in the system.

1.5 Fundamental Types

In this section we list all the simple types that exist in FPP in order of increasing complexity.

1.5.1 Type taylor

```
TYPE TAYLOR
   INTEGER I
END TYPE TAYLOR
```

This type overloads the Taylor series of the original real "DA-Package" of Berz. See §1.4. Its direct use is discouraged. Tracking programs should use the real polymorph type `real_8`.

1.5.2 Type complextaylor

```
TYPE COMPLEXTAYLOR
   type (taylor) r
   type (taylor) i
END TYPE COMPLEXTAYLOR
```

The `complextaylor` is made of two Taylor series which represent the real and imaginary parts. Its direct use is also discouraged as was already explained in Sec. §1.4.

Consider this code fragment:

```
type(complextaylor) z1,z2

call init(2,4)
.
.
.
z1 = dz_t(1) + i_ * dz_t(2)
z2=z1**2

write(6,*) " this is z1 "
call PRINT(z1)
write(6,*) " this is z2 "
call PRINT(z2)
```

The variable `z1` is, in Leibnitz mathematical language, $dx+i\,dy$. The variable `z2` must simply be $dx^2 - dy^2 + i\,2\,dxdy$.

```
 this is z1

Properties, NO =    2, NV =    4, INA =   23
*****************************************

  1   1.000000000000000       1  0  0  0
```

```
Properties, NO =    2, NV =    4, INA =   24
*******************************************

  1   1.000000000000000       0  1  0  0

 this is z2

Properties, NO =    2, NV =    4, INA =   25
*******************************************

  2   1.000000000000000       2  0  0  0
  2  -1.000000000000000       0  2  0  0


Properties, NO =    2, NV =    4, INA =   26
*******************************************

  2   2.000000000000000       1  1  0  0
```

1.5.3 Type `real_8`

The `real_8` type was discussed in Sec. §1.3 and Sec. §1.4. 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[3] 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 Sec. (1.4):

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

1.5.4 Type complex_8

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. If the type `complex_8` did not exist, then a code could become extremely difficult to write. This happens a lot when Maxwell's equations are written in cylindrical coordinates. In such a case, if $\mathbf{z} = (x, p_x, y, p_y)$, then most intermediate calculations involve a quantity $q = x + i\,y$. It in such a case, the complex polymorph is useful. This happens in the tracking code for some magnets in PTC but it is generally hidden from a normal programmer using PTC.

```
TYPE COMPLEX_8
  TYPE (COMPLEXTAYLOR) T
  COMPLEX(DP) R
  LOGICAL(LP) ALLOC
  INTEGER KIND
```

[3]This is not true of Berz's COSY INFINITY which handles variable memory of TPSA within its own internal language.

```
    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.

1.6 Type probe_8 specific to PTC

The types discussed in Sec. (1.5) are useful to 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 type `real_8`. But since our ultimate goal is to describe the analysis part of FPP, we need to say a little bit more about the objects used in PTC.

PTC uses the following `probe_8` structure for tracking

```
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
   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 value of the ray for TPSA calculations with c_damap
          .
          .
          .
end type probe_8
```

As the reader can see, it is made out of other structures which are themselves made of `real_8` or simple real numbers These components are described later.

1.6.1 The x(6) component of probe_8

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

- In `PTC`, when "`time`" units is used, the orbital phase space is:

$$x(1:6) = \left(x, \frac{p_x}{p_0}, y, \frac{p_y}{p_0}, \frac{\Delta E}{p_0 c}, cT \text{ or } \mathrm{c}\Delta T \right) \tag{3}$$

- When `BMAD` units are used, the orbital phase space is:

$$x(1:6) = \left(x, \frac{p_x}{p_0}, y, \frac{p_x}{p_0}, -\beta cT \text{ or } -\beta \mathrm{c}\Delta T, \frac{\Delta p}{p_0} \right) \tag{4}$$

where

$$\Delta T = T - T_{ref} \tag{5}$$

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.

1.6.2 The spin and quaternion components of probe_8

PTC can track spin. There are two ways to track spin: one method uses a regular spin matrix and the other uses a quaternion. 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`. The components of a `spinor_8` are

```
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

$$\begin{aligned} s(1) &= (1,0,0) \\ s(2) &= (0,1,0) \\ s(3) &= (0,0,1) \end{aligned} \tag{6}$$

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 Sec. (1.7.3).

The polymorphic quaternion `quaternion_8`, not surprisingly, is given by:

```
type  quaternion_8
  type(real_8) x(0:3)
end type quaternion_8
```

As we will see, 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 Sec. (1.7.3).

1.6.3 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.

1.6.4 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, like most codes, does not attempt to go beyond linear

dynamics when dealing with photon fluctuations. When radiation is present, the polymorphs `x(6)` contain the closed orbit (with classical radiation) and `E_ij(6,6)` measures the fluctuations $< x_i\, x_j >$ $(i, j = 1, 6)$ due to photon emission. PTC does not attempt any polymorphic computations: you get the zeroth order results around the orbit computed.

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

$$\Sigma^f = M\,(\Sigma + E)\,M^{\mathrm{T}} \tag{7}$$

where

$$\Sigma_{ij} = \langle x_i x_j \rangle\,. \tag{8}$$

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

$$\Sigma^f = M\left(\Sigma + E^{probre\_8}\right)M^{\mathrm{T}} = M\Sigma M^{\mathrm{T}} + E^{c\_damap}. \tag{9}$$

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

$$\Sigma^f = \; M\Sigma M^{\mathrm{T}} + E, \tag{10}$$

where E has been redefined.

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

$$M = B\Lambda B^{-1} \quad \text{where}$$
$$\Lambda = \begin{pmatrix} \lambda_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \lambda_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & \lambda_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & \lambda_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & \lambda_6 \end{pmatrix} \text{and} \quad \begin{array}{l} \lambda_1 = \exp\left(-\alpha_1 - i2\pi\mu_1\right) \\ \lambda_2 = \exp\left(-\alpha_1 + i2\pi\mu_1\right) \\ \lambda_3 = \exp\left(-\alpha_2 - i2\pi\mu_2\right) \\ \lambda_4 = \exp\left(-\alpha_2 + i2\pi\mu_2\right) \\ \lambda_5 = \exp\left(-\alpha_3 - i2\pi\mu_3\right) \\ \lambda_6 = \exp\left(-\alpha_3 + i2\pi\mu_3\right) \end{array} . \tag{11}$$

We can apply the transformation B on Eq. (10):

$$\begin{aligned} B\Sigma^f B^{\mathrm{T}} = \; & BM\Sigma M^{\mathrm{T}} B^{\mathrm{T}} + BEB^{\mathrm{T}} \\ \sigma^f = \; & \Lambda\sigma\Lambda + \varepsilon \end{aligned} \tag{12}$$

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

$$\begin{aligned} \sigma^{\infty} &= \Lambda\sigma^{\infty}\Lambda + \varepsilon \\ &\Downarrow \\ \sigma^{\infty}_{ij} &= \frac{\varepsilon_{ij}}{1-\lambda_i\lambda_j} \end{aligned} \tag{13}$$

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

$$\sigma^{\infty}_{12} = \frac{\varepsilon_{12}}{1-e^{-2\alpha_1}} \approx \frac{\varepsilon_{12}}{2\alpha_1}. \tag{14}$$

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:

$$\Sigma^{\infty} = \ B^{-1}\sigma^{\infty}B^{-1\ \mathrm{T}}\ . \tag{15}$$

In appendix E we explain how one can use E in stochastic tracking.

1.6.5 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` defined as

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

The components of the `probe` are all analogous to the components of the `probe_8` with `real_8` replaced by `real(dp)`. With this, most operations are acting directly and quickly[4] on Fortran intrinsic types.

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)`. As an example, consider the subroutine in the example of Sec. (1.5.3):

```
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
```

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

```
subroutine track(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
```

this is indeed a trivial drift-kick routine.

[4] "Quickly" is a relative term. PTC is a slow code even with real numbers.

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[5] of the code.

For example, in Sec. (1.5.3), we got the following results for the final polymorphs:

```
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
```

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

$$\mathbf{z} = \begin{pmatrix} 1 & 1 \\ -0.1 & 0.9 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} + O\left(z^2 \right) \tag{16}$$

Therefore we could say that $\mathbf{z}$ 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

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

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 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 Sec. (1.7.1), 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[6] 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 Sec. (2).

1.7 The type c_taylor and the Taylor maps c_damap for analysis

1.7.1 The type `c_taylor` and the complex Berz's package

The reader will notice two seemingly identical Fortran files : `c_dabnew.f90` and `cc_dabnew.f90`. The first package creates and manipulates real Taylor series. In other words, the coefficients of the monomials are `real(dp)`. The complex objects of Secs. (1.5.2) and (1.5.4) are made of two real `taylor`s or `real_8`s respectively. The individual coefficients inside Berz' package `c_dabnew.f90` are real.

It turns out that this is very inconvenient in accelerator physics when we analyze maps. Since most of our maps are stable, their diagonalized representation contains complex numbers. For example, the map of Eq. (16) can

[6]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.

be diagonalize into:

$$\Lambda = \begin{pmatrix} \exp(-i\ \mu) & 0 \\ 0 & \exp(\ i\ \mu) \end{pmatrix} \text{ where } \mu\ =\ 0.317560429291521 \qquad (17)$$

In fact, the output from the code, a `c_damap`, is made of two `c_taylor`. Please notice that the coefficient have a real and imaginary part corresponding to the cosine and sine of μ :

```
           2  Dimensional map

 Properties, NO =    1, NV =    2, INA =  139
 ********************************************

   1  0.9499999999999998     -0.3122498999199199       1  0


 Properties, NO =    1, NV =    2, INA =  138
 ********************************************

   1  0.9499999999999998      0.3122498999199199       0  1

  No Spin Matrix
  c_quaternion is identity
 No Stochastic Radiation
```

Finally if FPP were to be used in electron microscopy, the eigenfunctions of L_z representing symmetry around the axis of the microscope are very useful: $x \pm iy$. We can see how a complex TPSA package is almost unavoidable in the field of beam dynamics.[7]

1.7.2 The type `c_damap`

The type `c_damap` is defined as

```
type c_damap
  type (c_taylor) v(lnv)
  integer :: n=0
```

[7] Of course, Berz's COSY INFINITY handles complex maps.

```
  type(c_spinmatrix) s
  type(c_quaternion) q
  complex(dp) e_ij(6,6)
  complex(dp) x0(lnv)
  logical :: tpsa=.false.
end type c_damap
```

There is an obvious resemblance with `probe_8` which we recall is:

```
   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
    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
          .
          .
 end type probe_8
```

The first thing one notices is that `c_damap` contains `lnv=100` complex Taylor series. Indeed the analysis part of FPP does not put any limit on the dimensionality of phase space beyond `lnv`. For example, if the user tracks six-dimensional phase space (as in BMAD) and adds two modulated clocks, then the component `v(lnv)` will use 10 Taylor series to represent the map leaving 90 unused.

The `c_spinmatrix` is a matrix of `c_taylor` for the three spin directions

```
 type c_spinmatrix
    type(c_taylor) s(3,3)
 end type c_spinmatrix
```

and the `c_quaternion` follows the polymorphic quaternion:

```
type  c_quaternion
 type(c_taylor) x(0:3)
END TYPE c_quaternion
```

It is important to realize that `c_damap` can be concatenated. For example, the diagonal matrix Λ of Eq. (17) was obtained with a Fortran statement which represents a similarity transformation where `normal%atot` turns the original map into a rotation and `c_phasor()` diagonalizes the rotation:

```
diag=c_phasor(-1)*normal%atot**(-1)*one_period_map*normal%atot*c_phasor()
```

The Fortran symbol `*` is overloaded to represent the "differential algebraic" (DA) concatenation of five maps in this example. When we deal with maps around a closed orbit, the maps and the various differential operators we will later discuss, form a self-consistent differential algebra if the constant part is ignored. These complicated operators we will later define: Lie vector fields, etc ...

Most of perturbation theory deals with differential algebraic operators. On the other hand, it is possible to concatenate `c_damap` using truncated power series algebra (TPSA). In the case of TPSA, the constant part of a map is taken into account. In that case, the map concatenation uses the symbol "`.o.`". As we alluded at the beginning of Sec. (1.7), it is sometimes preferable to use a single type for two different purposes: then a new operator must be defined. The component `x0(lnv)` and `tpsa` are related to the TPSA usage of `c_damap` and will be explained later.

A tracking code like PTC produces TPSA maps if we do not compute them around the closed orbit because of feed down issues. Thus most calculations of lattice functions must be preceded by a computation of the closed orbit for the obtention of self-consistent results. If the maps were of infinite or very large order, then we could always deal with TPSA[8] maps and the closed orbit search would be part of the map analysis.

1.7.3 Interaction between the worlds of probes and Taylor maps

Let us assume that we want to compute a `probe_8`, called `polymorphic_probe`, and that we want to track it around some orbit `z0=(z0(1),z0(2))`. For example, `z0` might contain the closed orbit. Since PTC deals with `probe`, we

[8]This is view point of COSY INFINITY of Berz but we reject it in the context of symplectic integrators although its has its place in other area of beam physics.

first stick this real initial trajectory into a real `probe`, say `probe0`. The syntax would be:

```
real(dp) :: z0(2) = (/0,0/)  ! special orbit
type(probe) :: probe0
type(probe_8) :: polymorphic_probe
              .
              .
 probe0=z0
 polymorphic_probe = probe0    <------ initial value of polymorphic_probe
 call track(polymorphic_probe)
```

However the reader will notice that nothing is said about the initial Taylor value of `polymorphic_probe`: it will simply acquire the value of `probe0`. The rays will start as real numbers and the final value of `polymorphic_probe` after tracking will be two real numbers: no Taylor information. So how do we initialize the `polymorphic_probe` correctly to obtain a map? Here is the code:

```
real(dp) :: z0(2) = (/0,0/)  ! special orbit
type(probe) :: probe0
type(probe_8) :: polymorphic_probe
type(c_damap) Identity, one_period_map
              .
              .
 probe0=z0
 Identity=1     <------World of c_damap

 polymorphic_probe = probe0 + Identity   <------probes and c_damap are mixing
 call track(polymorphic_probe)
 one_period_map=polymorphic_probe        <------probes and c_damap are mixing
```

`Identity` is a `c_damap`. The line `Identity=1` turns it into an identity map with **no** constant part. This is the differential algebraic world. Then `probe0` is "added" to `Identity` and the appropriate `probe_8` is created with the `=` sign.

Finally, once the tracking is done, it returns the final value of `polymorphic_probe`. Now, if we want to analyze the corresponding map, we perform the reverse assignment `one_period_map=polymorphic_probe`.

Someone may wonder why we insist on adding an identity map rather than the Taylor monomials as we did in Sec. (1.5.3). Indeed

```
Identity=1
```

is equivalent to

```
Identity%v(1)=dz_c(1)
```

```
Identity%v(2)=dz_c(2)
```

and thus `dz_8(1)` and `dz_8(1)` could have been added directly into the `probe_8` as we did in Sec. (1.5.3).

```
polymorphic_probe%x(1)=z0(1)+dz_8(1)
polymorphic_probe%x(2)=z0(2)+dz_8(2)
```

The answer will become obvious when we discuss analysis. When we track lattice functions, linear, nonlinear, with or without spin, the initial value involves a canonical transformation. For example, `Identity` would be replaced by `normal%atot`. A map like `normal%atot` if it is nonlinear or parameter dependent is a huge beast. But even in a coupled linear system in 2-d-f, `normal%atot` involves 16 values. It makes perfect sense to allow its addition to a `probe` to create the appropriate `probe_8` to track.

We are left with spin: how is spin transferred from the `probe_8` to a `c_damap`. This depends on the choice : quaternions versus 3 by 3 rotations. In the case of a rotation, the following code fragments gives us the answer:

```
DO I=1,3
 DO J=1,3
   t=R%S(J)%X(I)    ! a probe_8 in converted from real_8 to c_taylor
   DS%S%s(I,J)=t   ! It is  put into the spin matrix of a c_damap
  ENDDO
 ENDDO
```

The three vectorial spin directions of the `probe_8` `R` are fed into the `c_damap` `DS`.

In the case of quaternions, the polymorphic `quaternion_8` is a mirror image of the `c_quaternion` contained in the `c_damap`. Thus the conversion code is trivial:

```
DO I=0,3
  t=r%q%x(i)
  DS%q%x(i)=t
ENDDO
```

This concludes our overview of the quintessential types: the various Taylor types, the polymorphs, the probes of PTC and the maps which can be concatenated and analyzed.

2 Important types and their operators

Tracking codes such as PTC, BMAD, TEAPOT, Sixtrack, etc...track particles via brute[9] force integration. The tracking includes phase space and potentially other things such as spin. This is why PTC has a `probe` entity so as to accommodate anything "trackable." The `probe_8` was invented to accommodate Taylor series. For example, the phase space variables and the spin can be expanded as a Taylor series in some variables. For a `probe_8`, the extra information carried by the type `probe_8` is sometime passive like the spin. As we explained in Sec. (1.7), `probe_8` can be promoted to a *bona fide* Taylor approximation of the map of PTC.

We will now assume that we have a proper map of the type `c_damap`. This map has at least one degree of freedom and represents some dynamical system. We describe the types and the operations associated to `c_damap`.

2.1 TPSA? DA? What does that mean?

TPSA stands for "Truncated Power Series Algebra" and DA stands for "Differential Algebra." But 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.
- 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.

[9] From a certain point of view, symplectic integration is not so brute, but integration nevertheless. The reader is invited to read reference [1] for a complete discussion of the "Talman" view point of symplectic integration which the primary tool of PTC.

Let us look at our little code again

```
  subroutine track(z)     ! 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 subroutine track
```

with the parameters

```
1     ! L
0.001 ! B
.1    ! K_q
1.0   ! K_s
```

We run the code and invoke DA maps. The lattice is read from a file called `lattice-2.txt`. The main programme is called `z_concat_da_c_damap.f90`:

```
 L
   1.00000000000000
 K_b
  1.000000000000000E-003
 K_q
  0.100000000000000
 K_s
   1.00000000000000
 Lattice Read and polymorphs read
 closed orbit found
  Closed Orbit from Tracking -1.127016653792583E-002  4.073729261518271E-019
 The order of the Taylor series ?
1
 Tpsa =0 DA=1
1
```

```
          0

 One period DA map: around the closed orbit

          2  Dimensional DA map (around chosen orbit in map%x0)

Properties, NO =    1, NV =    2, INA =  154
*******************************************

  0 -0.1127016653792583E-01   0.000000000000000       0  0
  1  0.5647772404535650       0.000000000000000       1  0
  1   3.260938572756972       0.000000000000000       0  1


Properties, NO =    1, NV =    2, INA =  153
*******************************************

  1 -0.2525912157058627       0.000000000000000       1  0
  1  0.3121860247477024       0.000000000000000       0  1
```

The initial closed orbit is computed exactly by a Newton search and we compute the map around it. Notice that the DA map, type `c_damap`, contains the correct closed orbit. But it is most important to realize that the Taylor coefficients of the map are the correct ones: there is now feed down effect. If we run the same program to second order, we get:

```
 The order of the Taylor series ?
2
 Tpsa =0 DA=1
1
          0

 One period DA map: around the closed orbit

          2  Dimensional DA map (around chosen orbit in map%x0)

Properties, NO =    2, NV =    2, INA =  155
*******************************************

  0 -0.1127016653792583E-01   0.000000000000000       0  0
  1  0.5647772404535650       0.000000000000000       1  0
```

```
  1   3.260938572756972       0.000000000000000       0  1
  2  -4.930887690126961       0.000000000000000       2  0
  2  -16.38365252576786       0.000000000000000       1  1
  2  -17.07146650102016       0.000000000000000       0  2


 Properties, NO =    2, NV =    2, INA =  154
 *******************************************

  1 -0.2525912157058627       0.000000000000000       1  0
  1  0.3121860247477024       0.000000000000000       0  1
  2  -2.094295104116132       0.000000000000000       2  0
  2  -11.40566537033463       0.000000000000000       1  1
  2  -20.76413510185940       0.000000000000000       0  2
```

Moreover, we can square the map and call the tracking code for an additional turn:

```
  one_period_map*one_period_map

         2  Dimensional DA map (around chosen orbit in map%x0)

 Properties, NO =    2, NV =    2, INA =  191
 *******************************************

  0 -0.1127016653792583E-01   0.000000000000000       0  0
  1 -0.5047111071004802       0.000000000000000       1  0
  1   2.859723338385714       0.000000000000000       0  1
  2  -9.938986380329304       0.000000000000000       2  0
  2  -51.31013528451668       0.000000000000000       1  1
  2  -148.1284897283459       0.000000000000000       0  2


 Properties, NO =    2, NV =    2, INA =  192
 *******************************************

  1 -0.2215132172865710       0.000000000000000       1  0
  1 -0.7262243243870512       0.000000000000000       0  1
  2  0.2259717122158635       0.000000000000000       2  0
  2   3.521937616442731       0.000000000000000       1  1
  2  -38.07518299996050       0.000000000000000       0  2

 No Spin Matrix
 c_quaternion is identity
No Stochastic Radiation

 Two period map by tracking

         2  Dimensional DA map (around chosen orbit in map%x0)
```

```
Properties, NO =    2, NV =    2, INA =  191
*******************************************

  0 -0.1127016653792583E-01   0.000000000000000       0  0
  1 -0.5047111071004802       0.000000000000000       1  0
  1   2.859723338385714       0.000000000000000       0  1
  2  -9.938986380329304       0.000000000000000       2  0
  2  -51.31013528451668       0.000000000000000       1  1
  2  -148.1284897283459       0.000000000000000       0  2


Properties, NO =    2, NV =    2, INA =  192
*******************************************

  1 -0.2215132172865710       0.000000000000000       1  0
  1 -0.7262243243870513       0.000000000000000       0  1
  2  0.2259717122158634       0.000000000000000       2  0
  2   3.521937616442731       0.000000000000000       1  1
  2  -38.07518299996050       0.000000000000000       0  2
```

The results are identical to machine precision. The code fragment is

```
 print *,' ' ;print *," One period DA map: around the closed orbit "
call PRINT(one_period_map)
 two_period_map=one_period_map*one_period_map
 print *, " ";print *, " one_period_map*one_period_map "
 call PRINT(two_period_map)
 call track(polymorphic_probe)
 two_period_map=polymorphic_probe
  print *, " ";print *, " Two period map by tracking"
 call PRINT(two_period_map)
```

Notice the syntax for the concatenation of the maps:

```
two_period_map=one_period_map * one_period_map
```

In the FPP package, the operator `*` always deals with "DA" operations. The symbol `.o.` makes a TPSA operation. For example, suppose we *erroneously* replace `*` by`.o.` in the same code, the result becomes:

```
 one_period_m.o.one_period_map

          2  Dimensional DA map (around closed orbit)

Properties, NO =    2, NV =    2, INA =  191
*******************************************

  0 -0.1826160494929466E-01   0.000000000000000       0  0
  1 -0.4885796717061110       0.000000000000000       1  0
```

```
  1   3.279800663360453       0.000000000000000       0  1
  2  -10.87372847170263       0.000000000000000       2  0
  2  -55.23709362281821       0.000000000000000       1  1
  2  -153.8599029713216       0.000000000000000       0  2


 Properties, NO =    2, NV =    2, INA =  192
 *****************************************

  0  0.2580734710843199E-02   0.000000000000000       0  0
  1 -0.2273213028258177       0.000000000000000       1  0
  1 -0.5321585402511647       0.000000000000000       0  1
  2 -0.2760048530853147       0.000000000000000       2  0
  2   1.282402148707428       0.000000000000000       1  1
  2  -41.55016026602691       0.000000000000000       0  2
```

The x-component of the closed orbit was incorrectly substituted into the map upon concatenation. The results for p^{final} are simply wrong. So the moral of this story:

When using an integrator, find the closed orbit, then the maps around it and only use "DA" operations in the analysis package. The closed orbit search must not be done by the TPSA package but by the original tracking code.

Concatenation must use `*` and powers `**`. For TPSA maps, the equivalent operators are `.o.` and `.oo.`. For example, let us run the same example computing the map around `z0=(0.001,0.001)`.

```
 The order of the Taylor series ?
1
 Tpsa =0 DA=1
0
           0

  Closed Orbit from Tracking  -1.127016653792583E-002 -2.244213701843388E-019

  Closed Orbit using TPSA map -9.840325887876448E-003 -3.509210588784647E-005

  Linear using TPSA map around z=(0.001,0.001)

           2  Dimensional DA map (around chosen orbit in map%x0)

 Properties, NO =    1, NV =    2, INA =  108
 *****************************************

  0 -0.2018736094076522E-02   0.000000000000000       0  0
  1  0.4318728048252188       0.000000000000000       1  0
  1   3.033496081674794       0.000000000000000       0  1
```

```
Properties, NO =    1, NV =    2, INA =  107
*******************************************

  0 -0.3230112164086397E-02   0.000000000000000       0  0
  1 -0.3087523692190016       0.000000000000000       1  0
  1  0.1468045615699475       0.000000000000000       0  1

 No Spin Matrix
 c_quaternion is identity
No Stochastic Radiation

 Linear map around computed by TPSA inversion

          2  Dimensional DA map (around chosen orbit in map%x0)

Properties, NO =    1, NV =    2, INA =  108
*******************************************

  1  0.4318728048252188       0.000000000000000       1  0
  1   3.033496081674794       0.000000000000000       0  1


Properties, NO =    1, NV =    2, INA =  107
*******************************************

  1 -0.3087523692190016       0.000000000000000       1  0
  1  0.1468045615699475       0.000000000000000       0  1
```

The approximate closed orbit is found by solving

$$F(\mathbf{z}) = \mathrm{T}(\mathbf{z}) - \mathbf{z} = 0 \qquad \Longrightarrow \qquad \mathbf{z}_c = \mathrm{F}^{-1}(0) \tag{18}$$

This is done by the code (the maketpsa routine is explained near Eq. (22)):

```
newton_map=one_period_map; newton_map=maketpsa(newton_map);
newton_map%v(1)=newton_map%v(1)-(1.0_dp.cmono.1)
newton_map%v(2)=newton_map%v(2)-(1.0_dp.cmono.2)


newton_map=newton_map.oo.(-1)    <---- notice .oo. NOT **

print *," Closed Orbit from Tracking ",z1
z0(1)=newton_map%v(1)
z0(2)=newton_map%v(2)
 print *," Closed Orbit using TPSA map",z0
```

The new map is computed by similarity transformation and only the zeroth order and linear part is printed:

```
  print *,' ';print *," Linear using TPSA map around z=(0.001,0.001)"
newton_map=one_period_map.cut.2
call print(newton_map)
newton_map= maketpsa(one_period_map).o.to_closed_orbit <---- notice .o. NOT *
```

```
newton_map=(to_closed_orbit.oo.(-1)).o.newton_map <----- notice .o. and .oo.
newton_map=newton_map.cut.2 <----- order higher than 1 are cut

  print *,' ';print * ," Linear map around computed by TPSA inversion "

call print(newton_map)
```

The reader will notice that the linear matrix is very wrong. Of course, this being TPSA, things can be improved by using a higher order. For example, if we increase the order to 10, the result is

```
 The order of the Taylor series ?
10
 Tpsa =0 DA=1
0
           0

  Closed Orbit from Tracking  -1.127016653792583E-002 -2.244213701843388E-019

  Closed Orbit using TPSA map -1.127016228741658E-002 -4.413505319085650E-011

  Linear using TPSA map around z=(0.001,0.001)

           2  Dimensional DA map (around chosen orbit in map%x0)

 Properties, NO =   10, NV =    2, INA =  117
 ********************************************

   0 -0.2018736094076522E-02   0.000000000000000       0  0
   1  0.4318728048252188       0.000000000000000       1  0
   1   3.033496081674794       0.000000000000000       0  1


 Properties, NO =   10, NV =    2, INA =  116
 ********************************************

   0 -0.3230112164086397E-02   0.000000000000000       0  0
   1 -0.3087523692190016       0.000000000000000       1  0
   1  0.1468045615699475       0.000000000000000       0  1

  No Spin Matrix
  c_quaternion is identity
 No Stochastic Radiation

  Linear map around computed by TPSA inversion

           2  Dimensional DA map (around chosen orbit in map%x0)

 Properties, NO =   10, NV =    2, INA =  117
 ********************************************

   0 -0.1993840146805037E-08   0.000000000000000       0  0
   1  0.5647771992590910       0.000000000000000       1  0
   1   3.260938504625006       0.000000000000000       0  1
```

```
Properties, NO =   10, NV =    2, INA =  116
*****************************************

  0 -0.1043284625709833E-08   0.000000000000000       0  0
  1 -0.2525912330061146       0.000000000000000       1  0
  1  0.3121859781006693       0.000000000000000       0  1
```

For example, the last two polynomials should be compared with the DA results: we have 6 or 7 digits of agreement with a 10th order map for the linear matrix. Therefore to the extent that we consider the integrator model to be realistic, i.e. exact in the Talman sense (see reference [1]), there is no doubt that the computation of the closed orbit should be done exactly by the integrator and NOT by the Taylor series package as we have just done here.

2.2 List of DA and TPSA operators and associated types

1. `c_damap*c_damap` and `c_damap.o.c_damap`
2. `c_taylor*c_damap` and `c_taylor.o.c_damap`
3. `c_spinmatrix*c_damap` and `c_spinmatrix.o.c_damap`
4. `c_quaternion*c_damap` and `c_quaternion.o.c_damap`
5. `c_spinor*c_damap` and `c_spinor.o.c_damap`
6. `c_taylor.o.c_ray`
7. `c_damap.o.c_ray`
8. `c_spinor.o.c_ray`
9. `c_spinmatrix.o.c_ray`
10. `c_quaternion.o.c_ray`

The type `c_ray` is essentially a zeroth order map, i.e., very similar to the type `probe` but associated to FPP and not to the tracking code. It is defined as

```
TYPE c_ray
 complex(dp) x(lnv)            !# orbital and/or magnet modulation clocks
 complex(dp) s1(3),s2(3),s3(3) !# 3 spin directions
 type(complex_quaternion) q    !# quaternion
 integer n                     !# of dimensions used in x(lnv)
 complex(dp) x0(lnv)           !# the initial orbit around which the map is computed
end type c_ray
```

The type `c_spinor` consists of 3 spin directions:

```
type c_spinor
   type(c_taylor) v(3)
end type c_spinor
```

It can be used with the $SO(3)$ or the quaternion algebra to store spin directions.

2.2.1 Operation .o. and * on c_damap

Let us start with the type `c_damap` which is

```
type c_damap
  type (c_taylor) v(lnv)
  integer :: n=0
  type(c_spinmatrix) s
  type(c_quaternion) q
  complex(dp) e_ij(6,6)
  complex(dp) x0(lnv)
  logical :: tpsa=.false.
end type c_damap
```

When it is created with the assignment

```
c_damap=probe_8
```

then the constant part of the map is the final trajectory of the initial ray. In any "DA" concatenation using the operator `*`, this constant part is ignored. Let us do a one-dimensional linear example:

$$M(x) = ax + b \qquad \text{and} \qquad N(x) = cx + d \tag{19}$$

Then $M * N$ will be:

$$(M * N)(x) = ac\ x + b \tag{20}$$

The TPSA concatenation will be given by

$$(M.o.N)\,(x) = ac\;x + ad + b \tag{21}$$

In general, Eq. (21) makes little sense because we do not know around which coordinates `x0` the variable x is expressed. However, if the original coordinates `x0` are known, they can be fed into the `c_damap` and the map can be transformed as follows:

$$\begin{aligned}\widetilde{M}\,(x) &= \text{maketpsa}\,(M)\,(x) = a(x - M\%x0) + b\\ \widetilde{N}\,(x) &= \text{maketpsa}\,(N)\,(x) = c(x - N\%x0) + d\end{aligned} \tag{22}$$

The component flag `tpsa` of `c_damap` is set to true and the map concatenation behaves as in Eq. (21). Finally we notice that if the maps M and N were extracted from the same integrator, we expect the variable $N\%x0$ to be the constant b of the map M: the final value of the ray when $x = 0$ must be the initial value of the ray entering N. This is true whether we integrate through magnets or compute trajectories of bullets: it is a mathematical fact.

The reverse operation is also available. Consider this piece of code:

```
type{c_damap} m1,m2,mtot_tpsa,mtot_da,mtot
      .
      .
mtot_da=m2*m1
call print(mtot_da)

m1=maketpsa(m1)
m2=maketpsa(m2)
mtot_tpsa=m2.o.m1
call print(mtot_tpsa)
mtot=makeda(mtot_tpsa)
call print(mtot)
```

We would expect `mtot` and `mtot_da` to be identical. However this is only true if the order of the calculation is 1 or infinite: there is no feed down in a linear calculation and they are exactly computed if the order is infinite. Again, we emphasize that the user of an integrator should always manipulate "DA" maps. On occasion, we need to evaluate maps or Taylor series for a certain value of the ray, this is the only time we should use TPSA operators.

2.2.2 Operation .o. with type c_ray

We recall the type `c_ray`:

```
TYPE c_ray
 complex(dp) x(lnv)            !# orbital and/or magnet modulation clocks
 complex(dp) s1(3),s2(3),s3(3) !# 3 spin directions
 type(complex_quaternion) q    !# quaternion
 integer n                     !# of dimensions used in x(lnv)
 complex(dp) x0(lnv)           !# the initial orbit around which the map is computed
end type c_ray
```

The first obvious thing to do is to calculate the effect of a `c_damap` on a `c_ray`. All the so-called matrix codes, from COSY INFINITY to MARYLIE, must have that function since they are not integrators. The syntax of FPP is simple: `c_ray`= `c_damap.o.c_ray`.

Here are two examples from the end of the code `z_concat_da_c_damap.f90`. The first example concerns "DA" maps, maps properly computed around the closed orbit of the code. The code fragment is:

```
one_period_map=polymorphic_probe
m1=one_period_map
m2=one_period_map
             .
             .
             .
write(6,*) " Degree of polynomials ",c_%no
z0=0.001d0;
if(ipol==1) then
!!!!!!!!!!!!!!   checking squaring a DA  map  !!!!!!
write(6,*) " Using a DA map and checking various options "
two_period_map=m2*m1

probe0=z_closed+z0
 write(6,fm49) "initial ray for code tracking         ",probe0%x(1:2)
call track(probe0);call track(probe0);
write(6,fm49) "final ray using code tracking        ",probe0%x(1:2)

ray=0
ray%x(1:2)=z0
ray%x0(1:2)=0
rayf=two_period_map.o.ray
write(6,fm49)"final ray using relative coordinates",real(rayf%x(1:2))

ray=0
ray%x(1:2)=z_closed+z0
ray%x0(1:2)=two_period_map%x0(1:2)
rayf=two_period_map.o.ray
write(6,fm49) "final ray using absolute coordinates",real(rayf%x(1:2))

two_period_map=maketpsa(two_period_map)
rayf=two_period_map.o.ray
write(6,fm49) "final ray using TPSA map             ",real(rayf%x(1:2))
```

`m1` and `m2` are the one-turn map produced by our little code. In the code fragment, the initial coordinates around the exact closed orbit are $z0 = (0.001, 0.001)$. Obviously `two_period_map=m2*m1` indicates that we are dealing with the 2-turns map. The results are as follows if we track to order 2 and to order 6.

First to order 2:

```
Degree of polynomials           2
Using a DA map and checking various options
          initial ray for code tracking         -0.1027016653793E-01  0.1000000000000E-02
           final ray using code tracking        -0.9119049524440E-02 -0.9792013681693E-03
           final ray using relative coordinates -0.9124531918034E-02 -0.9820648153449E-03
           final ray using absolute coordinates -0.9124531918034E-02 -0.9820648153449E-03
           final ray using TPSA map             -0.9124531918034E-02 -0.9820648153449E-03
```

Secondly to order 6:

```
Degree of polynomials           6
Using a DA map and checking various options
```

```
initial ray for code tracking         -0.1027016653793E-01  0.1000000000000E-02
 final ray using code tracking        -0.9119049524440E-02 -0.9792013681693E-03
 final ray using relative coordinates -0.9119049525235E-02 -0.9792013715064E-03
 final ray using absolute coordinates -0.9119049525235E-02 -0.9792013715064E-03
 final ray using TPSA map             -0.9119049525235E-02 -0.9792013715064E-03
```

A proper scaling test with order would reveal that the improvement is consistent with the order used. That is to say, the difference between the Taylor maps and the tracking code scales like the power of the order plus one.

Just to make things clear:

1. The first result is from the tracking code as indicated in the output.
2. The second result (third line) comes from putting directly the relative coordinates inside the DA map coming from the integrator. This is the "preferred mode" from the code PTC where everything is assumed to be expressed in around the closed orbit.
3. However, we can put into the full coordinate into DA map. This is done by sticking the initial orbit in the object `ray%x0=%two_period_map%x0(1:2)`. The concatenation operator `.o.` will subtract this initial vector from input automatically before evaluation: *de facto* recreating the results of item 2.
4. Finally we can create a true TPSA map where the Taylor series is expressed around $z = (0.0)$ and the input is the full ray. This is done with the function `maketpsa(c_damap)`. This is incidentally the preferred option of BMAD internally.

The next example is tracking with a *bona fide* TPSA map computed **not** around the closed orbit, but around $z = (0.001, 0.001)$.

```
m1=maketpsa(m1)
m2=maketpsa(m2)

!!!!!!!!!!!!!!    checking squaring a TPSA map  !!!!!!
two_period_map=m2.o.m1

ray=0
ray%x(1:2)=z_closed+z0
ray%x0(1:2)=two_period_map%x0(1:2)

rayf=two_period_map.o.ray
write(6,fm49) "final ray using TPSA map             ",real(rayf%x(1:2))
```

```
probe0=z_closed+z0
 write(6,fm49) "initial ray for code tracking        ",probe0%x(1:2)

call track(probe0)
call track(probe0)

write(6,fm49) "final ray using code tracking       ",probe0%x(1:2)
```

The results are as follows:

```
 Degree of polynomials           2
           final ray using TPSA map            -0.9082808079376E-02 -0.9847918957181E-03
          initial ray for code tracking        -0.1027016653793E-01  0.1000000000000E-02
           final ray using code tracking       -0.9119049524440E-02 -0.9792013681693E-03
```

Secondly to order 6:

```
 Degree of polynomials           6
           final ray using TPSA map            -0.9119049403298E-02 -0.9792013513959E-03
          initial ray for code tracking        -0.1027016653793E-01  0.1000000000000E-02
           final ray using code tracking       -0.9119049524440E-02 -0.9792013681693E-03
```

The results results are not improving as fast since there are two sources of errors in the Taylor series: it is not computed around the correct orbit and it is truncated. Again this is why in an accelerator, it pays to evaluate the Taylor map around the closed orbit. In a Taylor series code, the user must try to evaluate the Taylor map around an orbit close to location of the final closed orbit and one is usually forced to deal with high order Taylor maps.

2.2.3 The ** and .oo. operators

These operators do the expected thing: they raise a `c_damap` to a power where the multiplication is either `*` or `.o.`. If the power is negative, they compute the inverse map. This map will not be unique in the TPSA case `.oo.` because as usual feed down effects will depend on the order of truncation.

2.3 Types related to analysis

The types described here are complicated and we refer the reader to reference [2] for detailed examples.

2.3.1 c_vector_field

```
TYPE c_vector_field  !@1
 !@1 n dimension used v(1:n) (nd2 by default)
 !@1 nrmax iterations some big integer if eps<1
 integer :: n=0,nrmax
 !@1 if eps=-integer  then |eps| # of Lie brackets are taken
 !@ otherwise eps=eps_tpsalie=10^-9
 real(dp) eps
 type (c_taylor) v(lnv)   ! vector field denoted by F below
 type(c_quaternion) q   ! quaternion operator denoted by f below
END TYPE c_vector_field
```

The type `c_vector_field` is extremely important in perturbation theory. It is its analysis that permits someone to extract resonances, tunes shifts, damping, etc.... out of the `c_damap`. In general, if you have a `c_vector_field` you can produce a `c_damap` by the process of exponentiation.

$$M = \exp\left(F \cdot \nabla\right) I \tag{23}$$

Here I is the identity. But one can also act on any map:

$$P = \exp\left(F \cdot \nabla\right) N \Longleftrightarrow P = N \circ M \tag{24}$$

The type `c_vector_field` is general: it will create a map with a constant coefficient if it has itself a constant coefficient. In that case, it is a TPSA object and feed down matters. If it has no constant part, then it is a DA object with total self-consistency.

We now list the various operations, italic objects are the default value of an optional variable. The operator $\widehat{F}$ below stands really for

$$\widehat{F}q = F \cdot \nabla + \widehat{f} \tag{25}$$

where $\hat{f}$ is a quaternion operator. The $\widehat{F}$ acts as on a quaternion as:

$$\widehat{F}q = F \cdot \nabla q + qf \tag{26}$$

or $\hat{f}q = qf$. This reversal order is necessary for the self-consistency of the regular maps and the Lie operators. This is explained in appendix A. It is a consequence of Eq. (24) when extended to quaternions.

1. `c_vector_field=c_vector_field * c_taylor` $\rightarrow F \cdot \nabla t$
2. `c_damap =c_vector_field * c_damap` $\rightarrow \widehat{F} M$
3. `c_taylor=exp(c_vector_field,c_taylor)` $\rightarrow exp\left(\widehat{F}\right) t$
4. `c_damap=exp(c_vector_field,c_damap:`*Identity*`)` $\rightarrow exp\left(\widehat{F}\right) M$
5. `c_vector_field=exp(c_vector_field,c_vector_field)` See appendix C and Eq. (93).
6. `c_quaternion=exp(c_vector_field,c_quaternion)` $\rightarrow exp\left(\widehat{F}\right) q$
7. `exp(c_factored_lie,c_damap:`*Identity*`)`

$$\rightarrow \underbrace{exp\left(\widehat{F}_1\right) \cdots exp\left(\widehat{F}_n\right)}_{dir=1} M \tag{27}$$

$$\rightarrow \underbrace{exp\left(\widehat{F}_n\right) \cdots exp\left(\widehat{F}_1\right)}_{dir=-1} M \tag{28}$$

```
TYPE c_factored_lie
      integer :: n = 0
      integer :: dir= 0
       type (c_vector_field), pointer :: f(:)=>null()
END TYPE c_factored_lie
```

8. `exp_inv(c_factored_lie,c_damap:`*Identity*`)` Generates the inverse of the maps of item 7.
9. `c_factor_map(U=c_damap ,L=c_damap, F=c_vector_field,dir=-1,1)`

   This subroutine factors a map into the Dragt-Finn or reversed Dragt-Finn factorisation.

$$M = \underbrace{exp\left(\widehat{F}\right) I_{total} \circ L}_{dir=1} = \underbrace{L \circ exp\left(\widehat{F}\right) I_{total}}_{dir=-1} \tag{29}$$

10. `c_full_factor_map(U,Q,U_0,U_1,U_2)` ; all `c_damap`.

    This is a very special factorisation of the `c_damap`.

$$U = Q \circ U_0 \circ U_1 \circ U_2 \tag{30}$$

    The map Q is a pure quaternion map. The map U_0 is a pure translation as a function of the parameters of TPSA which might include the energy variable if it is a constant. The map U_1 is a linear map as a function of the parameters. The map U_2 is a pure non-linear map in phase space variables.

    It should be said that the situation is a little more complex in the case when the energy variable (the fifth one in PTC) is a constant. In that case, $z_5 = \delta$ is a canonical variable, therefore the map U_0 contains in the time variable[10] terms which depend on position if U_0 is to be symplectic.

    This factorisation is particularly useful if applied to the transformation of a normal form algorithm:

$$M = Q \circ U_0 \circ U_1 \circ U_2 \circ R \circ U_2^{-1} \circ U_1^{-1} \circ U_0^{-1} \circ Q^{-1} \tag{31}$$

    Then U_0 takes us to the fixed point to all orders in the parameters, U_1 gives us the "Courant-Snyder" to all orders in the parameters. U_2 constructs the non-linear invariant. Q gives us the invariant spin field in the original coordinates.

For most ordinary applications, items 2 and 7 are most useful. Eq. (27) represents the so-called Dragt-Finn perturbation theory. Eq. (28) is the so-called reversed Dragt-Finn factorisation and enters naturally in normal form theory: calculations are performed order by order leading to a factored canonical transformation.

[10] One should consult reference [2] on this issue: the section on Jordan normal form.

2.3.2 Normal Form

The type normal form is one of the most useful. It is at the basis of the usual Courant-Snyder theory and its extension to coupled systems, non-linear systems, spin, etc... Reference [2] deals with the topic of general Courant-Snyder theory.

```
TYPE c_normal_form
 type(c_damap) a1   !@1 brings to fix point at least linear
 type(c_damap) a2   !@1 linear normal form
 type(c_factored_lie) g   !@1 nonlinear part of a in phasors
 type(c_factored_lie) ker !@1  kernel i.e. normal form in phasors
 type(c_damap) a_t !@1 transformation a (m=a n a^-1)
 type(c_damap) n   !@1 transformation n (m=a n a^-1)
 type(c_damap) As  !@1  For Spin   (m = As a n a^-1 As^-1)
 type(c_damap) Atot  !@1  For Spin   (m = Atot n Atot^-1)
 integer NRES,M(NDIM2t/2,NRESO),ms(NRESO) !@1 stores resonances to be left in the map, including spin (ms)
 real(dp) tune(NDIM2t/2),damping(NDIM2t/2),spin_tune !@1 Stores simple information
 logical positive ! forces positive tunes (close to 1 if <0)
!!!Envelope radiation stuff to normalise radiation (Sand's like theory)
 complex(dp) s_ij0(6,6)  !@1  equilibrium beam sizes
 complex(dp) s_ijr(6,6)  !@1  equilibrium beam sizes in resonance basis
 real(dp) emittance(3)   !@1  Equilibrium emittances as defined by Chao (computed from s_ijr(2*i-1,2*i) i=1,2,3)
END TYPE c_normal_form
```

Explanation of the normal form and its propagation through a lattice, the so-called Courant-Snyder loop (or Twiss loop), is beyond the scope of this manual. Reference [2] is entirely on that topic.

We will concentrate on the part of the normal form which work with spin. First the call statement:

`c_normal(M:c_damap, N:c_normal_form,dospin:`*false*`,no_used:`*Identity*`,rot:`*nothing computed*`,phase:`*nothing computed*`,nu_spin:`*nothing computed*`)`

1. `M=N%Atot*N%n*N%Atot**(-1)`
2. `M=N%As*N%A_t*N%n*N%A_t**(-1)*N%As**(-1)`

The map N%As either contains a quaternion (N%As%q) or a spin matrix N%AS%S(1:3,1:3). The usage of quaternion is encouraged in PTC although the flag use_quaternion is set to false by default for backward compatibility.

2.4 Normal Form into a Circle Example

The example contains two normal form calculations and the calculations of the vector field of the map.

1. From the normal form, the tunes, we construct the invariant vector field of the map including the quaternion part.
2. We compute the invariant spin field.
3. we compute the "horizontal invariant", i.e., the invariant associated with the first tune.

All these calculations are checked through tracking. We now explain the calculations using theory and fragments from a program.

```
closed=0.d0
call find_orbit(als,closed(1:6),1,state,1.d-5)   ! (1)
!!!!!!!!!!!!!
!!!! polymorphic probe is created in the usual manner
   xs0=closed(1:6)
   id_s=1
   xs=xs0+id_s                                   ! (2)

!!!! get  polymorphic probe after one turn
call propagate(als,xs,state,fibre1=1)            ! (3)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! copy probe_8 into a complex damap
c_map=xs                                         ! (4)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

call c_normal(c_map,c_n,dospin=my_true,phase=phase,nu_spin=nu_spin)      ! (5)
Write(6,*) " tunes in resonance basis "
call print(phase(1:c_%nd))
Write(6,*) " Spin tune in resonance basis "
call print(nu_spin)
```

In the above code fragment, the map is computed using PTC. First, as explained in Sec. (1.7.3), the probe `xs0` defined by the type `probe`

```
type probe
   real(dp) x(6)
   type(spinor) s(3)
   type(quaternion) q
   type(rf_phasor)  AC(nacmax)
   integer:: nac=0
   logical u,use_q
   type(integration_node),pointer :: last_node=>null()
    real(dp) e
end type probe
```

ORBIT Variable 1 0.000000000000 Variable 2 0.000000000000 Variable 3 0.000000000000 Variable 4 0.000000000000 Variable 5 0.000000000000 Variable 6 0.000000000000 quaternion real quaternion 1.00000000000000 0.000000000000000E+000 0.000000000000000E+000 0.000000000000000E+000

3 Tracking with maps outside PTC

We assumed that a `c_damap` m was obtained from a tracking code. we recall the definition of the map in FPP:

```
type c_damap
  type (c_taylor) v(lnv)
  integer :: n=0
  type(c_spinmatrix) s
  type(c_quaternion) q
  complex(dp) e_ij(6,6)
  complex(dp) x0(lnv)
  logical :: tpsa=.false.
end type c_damap
```

Here we assume that the map is "DA" and thus around the closed orbit. The initial value around which the map was computed, `x0` of the map is also supplied by the tracking code. So we get from the unknown source, the map and the initial ray around which it was computed. In our case, the unknown source is often the code PTC.

Thus we have, around the orbit produced by classical radiation, a nonlinear deterministic map U composed of an orbital map and a quaternion:

$$M = (m, q)\,. \tag{32}$$

This map U acts on a phase space vector $\vec{z}$ and a quaternion q_0 in the usual fashion:

$$M\,(\vec{z}, q_0) = (m(\vec{z}), q(\vec{z})q_0)\ \ . \tag{33}$$

We also have, as explained in Sec. (1.6.4), Eq. (10), a map for the moments which includes a stochastic term. If we denoted by L, the linear part of m, then the maps for the moments is:

$$\Sigma^f = L\Sigma L^{\mathrm{T}} + E^{c\_damap}. \tag{34}$$

As explained in appendix E, it is possible to generate a stochastic kick of the ray at every turn which will reproduce upon averaging the effect of Eq. (34). This map is:

$$\vec{z}_s = \vec{z} + F\delta\vec{r} \tag{35}$$

where the meaning of δ, F and $\vec{r}$ are explained in appendix E.

The nonlinear orbital map m is stored in the `c_damap` structure `v(lnv)`. The stochastic matrix E is stored in `e_ij`. The original ray $\vec{z}_0$ is passed in the structure `x0`. We are now ready for tracking.

3.1 Tracking "as is" or COSY-INFINITY tracking

If a map is of high enough order, we can track with the map "as is". In that case we simply evaluate the Taylor series. Thus the code performs the following operation in the routine of the package `Su_duan_zhe_map.f90` called `track_tree_probe_complex_zhe`:

```
m%x0(1:6)=x
 ! FPP
call fill_tree_element_line_zhe_outside_map(m,mapfile,as_is=.true.,stochprec=1.d-8)
! Su_duan_zhe_map
call read_tree_zhe(T(1:3),mapfile)
```

```
                                .
                                .
                                .

call track_tree_probe_complex_zhe(T:1:3),xs,spin,rad,stoch,linear)  !

!  spin,rad, stoch, and linear are logicals (true or false)
! xs is of type probe_zhe in PTC but simply probe is used outside PTC
! T(1:3) is of type  tree_element_zhe in PTC but tree_element if used alone
```

In the above code fragment, one sees that the package `Su_duan_zhe_map.f90` duplicates types which exist within PTC and thus must be renamed if the package is used inside PTC. Why is that? This is because Taylor maps can be used to substitute the integrator models of PTC. For example one can replace a quadupole by a Taylor map. These operations involve types which are duplicated in `Su_duan_zhe_map.f90`. In other words, one can write a code totally outside PTC which uses `Su_duan_zhe_map.f90` and thus the necessary types must match. However if you use `Su_duan_zhe_map.f90` within PTC, the types must be renamed. For example, the above code used

```
use duan_zhe_map, probe_zhe=>probe,tree_element_zhe=>tree_element, &
                  .
                  .
                  .
```

Therefore, the FPP routine creates from the `c_damap m` an FPP object of type `tree_element` and prints it on the file `mapfile`. It is then read back in the `tree_element_zhe` which is totally independent of FPP. The routine `track_tree_probe_complex_zhe` has several flags. However the flags `rad` and `linear` are superfluous. If a map is read "as is", then you cannot remove radiation or nonlinearities.

The tracking procedure has the following steps starting with the ray $(\vec{z}, q_0)$

$$\vec{z}_f = \vec{z} - \vec{f} \tag{36}$$

$$\vec{z}_s = \vec{z}_f + F\delta\vec{r} \tag{37}$$

$$\overline{q} = q\left(\vec{z}_s\right) q_0 \tag{38a}$$

$$q^f = \overline{q}/\left|\overline{q}\right| \tag{38b}$$

$$\vec{z}^f = m\left(\vec{z}_s\right) + \vec{f}. \tag{39}$$

So the code first goes around the closed orbit with radiation in Eq. (36). Then stochastic radiation is added in Eq. (37) using the trick of appendix E.

The quaternion is evaluated at Eq. (38a) using the Taylor series for the quaternion map q. The result is normalized at Eq. (38b).

Finally the Taylor map is evaluated "as is" and put back around the origin of phase space by Eq. (39).

3.2 Comments on "as is" tracking with radiation and stochasticity

There is a common belief in accelerator physics that "damping" solves all problems. However with a nonlinear map, truncated at some order, one has to realize that if a particle migrates in an area where the violations of the symplectic condition are bigger than the physical linear damping produced by radiation, then results could become unphysical.

In a particular case, we tracked a distribution of particles with a third order map "as is". We discovered that the final beam sizes in the vertical plane where similar to those in the horizontal plane. The linear theory had predicted sizes 3 order of magnitude smaller!

It should be said that this is certainly possible in the presence of a large Hamiltonian resonance. A particle drifts into the island of the resonance and is immediately propelled to larger amplitudes. In a lattice with only distortions, it is only possible if huge sextupoles, for example, are made to cancel— like the famous π-apart sextupoles. In the region inside the canceling pair of sextupoles, the amplitude could grow very big.

Except for the aforementioned pathologies, we do not expect nonlinearities to affect the beam sizes by order of magnitudes. However if the map is **not** symplectic, then it can happen. Specifically nonlinear antidamping can occur and propel a particle upwards as a function of position in phase space. We verified that to be the case in the third order example.

Therefore when you use a map "as is" preliminary care or a very high order map (as in COSY-INFINITY) are necessary.

3.3 Tracking the Taylor map in factored form

This factored form is used on an arbitrary map just as the "as is" method of Sec. (3.1). The call used to prepare the map is the same with the obvious difference that optional parameter `as_is` must be set to false.

```
m%x0(1:6)=x
call fill_tree_element_line_zhe_outside_map(m ,mapfile,as_is=.false.,stochprec=1.d-8)
```

3.3.1 Factoring the map

Let us concentrate on a map $M = (m, q)$ expressed around the closed orbit. Imagine that we can factorize the map as follows:

$$m = \underbrace{L_{ns} \circ N_{ns}}_{m_{ns}} \circ \underbrace{L_s \circ N_s}_{m_s} . \tag{40}$$

Here "ns" means "non-symplecitic and "s" stands for "symplectic". The algorithm to factor Eq. (40) starts with the linear part. We use a contraction map invented by Miguel Furman of LBNL. One starts with the full linear matrix L and apply the following map to it:

$$\mathcal{F}L = \left(\frac{3}{2} I + \frac{1}{2} L S L^{\mathrm{T}} S \right) L \ . \tag{41}$$

The matrix L_s is gotten by applying the map $\mathcal{F}$ until convergence:

$$\mathcal{F}^{\infty} L = L_s \ . \tag{42}$$

Furman's contraction map, inspired by a more complex contraction map due to Dragt, produces a symplectic matrix in the neighborhood of the original nonsymplectic matrix. Obviously, we can get L_{ns} by simple inversion:

$$L_{ns} = L \circ L_s^{-1} \ . \tag{43}$$

To get a symplectification of the nonlinear part— whose linear part is the identity— we take the "logarithm" of the map using a procedure described in reference [2]:

$$\exp\left(\vec{F}\cdot\vec{\nabla}\right) I = L_s^{-1}\circ L_{ns}^{-1}\circ m. \tag{44}$$

We then convert $\vec{F}\cdot\vec{\nabla}$ into a Poisson bracket operator. This can be done by a simple integral over phase space:

$$\text{if} \quad :f:=\vec{F}\cdot\vec{\nabla} \quad \Longrightarrow f=\int_0^{\vec{z}} S\vec{F}(\vec{\zeta})\cdot d\vec{\zeta}. \tag{45}$$

Here S is the symplectic form which defines the Poisson bracket. Eq. (45) seems to depend on the path of integration. However the integrand is a closed form if and only if the underlying map is symplectic. In our case, the map is not symplectic and therefore we must be careful about the path of integration. We know that radiation will distribute itself over one-turn in particular along all of phase space, therefore we choose a path along the diagonal of phase space:

$$f=\int_0^1 S\vec{F}(\alpha\vec{z})\cdot\vec{z}d\alpha. \tag{46}$$

In the case of a polynomial, the diagonal path is pretty simple since it amounts to multiplying each monomial by $=1/(n+1)$ where n is the order of the monomial. The integration is carried out to the order of truncation of the TPSA package.

It follows that N_s is just:

$$N_s=\exp\left(:f:\right) I. \tag{47}$$

3.3.2 Symplectic evaluation of N_s

We know through the manipulation of Sec. (3.3.1), that the map N_s is a Taylor series expansion perfectly consistent with the symplectic condition because it is generated by a Poisson Lie exponent namely $:f:$ of Eq. (46).

Unfortunately this representation is not adequate for numerical tracking since it would require an infinite number of Poisson brackets or an evaluation of the vector field flow due to f to machine precision using high order integrators. Thus we perform a partial inversion of the map to obtain a mixed generating function. This is done in the following code fragment:

```
js=0
js(1)=1;js(3)=1;js(5)=1; ! q_i(q_f,p_i) and p_f(q_f,p_i)

ms=n_s
ms=ms**js
```

The `c_damap N_s` is copied in a real map `ms` of type `damap` which is then "partially inverted" along the non-zero planes of the integer array `js`. The result is

$$\begin{aligned} q_i^0 &= M_{2i-1}^s(\vec{q}^f, \vec{p}^0) \\ p_i^f &= M_{2i}^s(\vec{q}^f, \vec{p}^0) \ . \end{aligned} \tag{48}$$

However because N_s was a symplectic map to the order of truncation, we are certain that Eq. (48) is the gradient of a single function g:

$$\begin{aligned} q_i^0 = M_{2i-1}^s(\vec{q}^f, \vec{p}^0) \ &= \frac{\partial}{\partial p_i^0} g \\ p_i^f = M_{2i}^s(\vec{q}^f, \vec{p}^0) &= \frac{\partial}{\partial q_i^f} g \ . \end{aligned} \tag{49}$$

Solving Eq. (49) using a Newton search exactly on the computer leads to an exactly symplectic map to machine precision.

3.4 The PTC-based factorization in BMAD

The BMAD factorization uses the true symplectic map of PTC rather than the *ad hoc* factorization of Sec. (3.3).

```
call fill_tree_element_line_zhe0_node(state0_sagan,state_sagan,it,itn,    &
                                      no,orb0, orb,filef=mapfile,stochprec = 1d-10)
```

Here the states `state0_sagan` and state `state_sagan` are set to:

```
state0_sagan = time0+spin0
state_sagan  = state0_sagan + radiation0 + envelope0
```

This means that the symplectic state will be a state using time as the longitudinal variable in the presence of a cavity. The non-symplectic state will be the state with radiation and stochasticity computed via the envelope, i.e., Eq. (35).

`orb0` is the closed orbit in `state0_sagan` at the initial `integration_node` denoted by `it` in the call to `fill_tree_element_line_zhe0_node`. Conversely `orb` is the closed orbit for the state `state_sagan` at the same point in the ring. All the maps are computed from the entrance integration node `it` to the front of the final node `itn`.

The tracking call is the same as before:

```
track_tree_probe_complex_zhe(t(1:3),xs(k),spin=.true.,rad=.true.,stoch=.true.)
```

The map used in BMAD is factorized using the known model of PTC. Therefore the symplectic part of the map is the map computed in the state `state0_sagan` properly factorized into a linear part and a nonlinear part where, as in Sec. (3.3.2), a generating function is used.

The spin map used, whether quaternion or $SO(3)$, is the map of the state `state_sagan` by default. If there is no spin in that state, then the spin of the map of the state `state0_sagan` is used. Of course if there is no spin at all, then spin tracking is not possible.

3.5 Handling the spin part of the map

If the quaternion is used, then the tracking routine simply enforces a unit quaternion:

```
xs%q=qu*xs%q
xs%q%x=xs%q%x/sqrt(xs%q%x(1)**2+xs%q%x(2)**2+xs%q%x(3)**2+xs%q%x(0)**2)
```

In the above code `qu` is the quaternion of the map.

If the map is an $SO(3)$ rotation, the unitary of the matrix, which is phase space dependent, is enforced by a Furman-like procedure similar to the one described in Eq. (41). If R is a matrix near the group $SO(3)$, then the following contraction will produce a matrix in $SO(3)$:

$$\mathcal{F}R = \left(\frac{3}{2}I - \frac{1}{2}RR^{\mathrm{T}}\right) R \ . \tag{50}$$

3.6 Spin depolarization

If one tracks with radiation, one notices spin depolarization due to the stochastic nature of radiation. Assuming a small beam aligned along the invariant direction $\vec{n}$, we expect a slow depolarization of the beam. Mathematically we expect the following average

$$\langle \vec{s} \rangle = \frac{1}{N} \sum_{i=1,N} \vec{s^i} \tag{51}$$

to slowly decay away from a vector in the neighbourhood $\vec{n}_0$.

The scalar polarization is:

$$P = |\langle \vec{s} \rangle| = \sqrt{\langle s_1 \rangle^2 + \langle s_2 \rangle^2 + \langle s_3 \rangle^2}. \tag{52}$$

3.6.1 The vectors $\vec{n}$, $\vec{m}$ and $\vec{l}$

Let us examine a single spin vector $\vec{s}$ near $\vec{n}_0$. We know that in the absence of radiation, $\vec{s}$ can be decomposed as:

$$\vec{s} = \vec{s} \cdot \vec{n}\ \vec{n} + \alpha \vec{m}\ +\ \beta \vec{l}. \tag{53}$$

Secondly, we assume that a normal form has been performed on the full spin-orbit map. Here a is a quaternion.

$$(m, q) = (I, a) \circ (m, q_y) \circ (I, a)^{-1} \tag{54a}$$

$$q_y = \cos\left(\frac{\theta}{2}\right)\ +\ j \sin\left(\frac{\theta}{2}\right)\ . \tag{54b}$$

The factorization of Eq. (54) is created by FPP using the one-turn map. Then we produce the spin matrix corresponding to the quaternion a. Let us call that 3 by 3 matrix A. We can construct an orthogonal basis out of A:

$$\vec{n} = A \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \qquad \vec{m} = A \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \qquad \vec{l} = A \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}. \tag{55}$$

By construction, it is clear that $\vec{n}$ is the Invariant Spin Field (ISF) and the other vectors are perpendicular to the ISF.

Of course the matrix A depends on the orbital vector $\vec{z}$. Therefore, each one of these vectors can be expanded in $\vec{z}$: this comes out of the normal form process for free.

$$\vec{n} = \vec{n}_0 + \vec{\nu}^i z_i \quad \vec{m} = \vec{m}_0 + \vec{\mu}^i z_i \quad \vec{l} = \vec{l}_0 + \vec{\lambda}^i z_i. \tag{56}$$

Notice that these vectors obey certain relations because they form an orthonormal basis:

$$\vec{n}_0 \cdot \vec{\nu}^i = \vec{m}_0 \cdot \vec{\mu}^i = \vec{l}_0 \cdot \vec{\lambda}^i = 0 \tag{57a}$$

$$\vec{n}_0 \cdot \vec{\mu}^i + \vec{m}_0 \cdot \vec{\nu}^i = 0 \tag{57b}$$

$$\vec{n}_0 \cdot \vec{\lambda}^i + \vec{l}_0 \cdot \vec{\nu}^i = 0 \tag{57c}$$

$$\vec{m}_0 \cdot \vec{\lambda}^i + \vec{l}_0 \cdot \vec{\mu}^i = 0. \tag{57d}$$

Notice that the vector $\vec{\nu}^6 = \frac{\partial \vec{n}}{\partial z_6}$, if "6" is the index of the energy variable, is the vector that enters in most formulas concerning polarization in electron rings. We will see later that it does not enter naturally in the formula based on the solution of the SDE. However it can be thrown back in using Eqs. (57b) and (57c).

3.6.2 Computation of the depolarization rate

Here I more or less follow Barber's idea in his lectures "An Introduction to Spin Polarisation in Accelerators and Storage Rings."

Consider a spin vector $\vec{s}$. It can be represented using the basis of Eq. (55) as is done in Eq. (53). The average of this vector over time is simply $\vec{s} \cdot \vec{n}\, \vec{n}$ since it rotates around the ISF $\vec{n}$.

Following partly Barber, we write the polarization P of Eq. (52) as:

$$\begin{aligned} P &= \left\langle \sqrt{\langle s_1 \rangle^2 + \langle s_2 \rangle^2 + \langle s_3 \rangle^2} \right\rangle = \langle \vec{n} \cdot \vec{s} \rangle \\ &= \left\langle \sqrt{1 - \alpha^2 - \beta^2} \right\rangle \end{aligned} \tag{58}$$

We need to compute the change in polarization due to a sudden change in α and β resulting from a sudden radiation process. So to proceed we expand Eq. (58):

$$\Delta P = \left\langle -\frac{\alpha\Delta\alpha + \Delta\alpha^2 + \beta\Delta\beta + \Delta\beta^2}{2\ \vec{n}\cdot\vec{s}} + \frac{\{\alpha\Delta\alpha + \beta\Delta\beta\}^2}{2\ \vec{n}\cdot\vec{s}^{\,3}} \right\rangle \tag{59}$$

At this stage we follow Barber by assuming that we are very near full polarization and therefore α and β are very small. Then we get

$$\Delta P = -\left\langle \frac{\Delta\alpha^2 + \Delta\beta^2}{2\ \vec{n}_0\cdot\vec{s}} \right\rangle = \left\langle \frac{1}{2\ \vec{n}_0\cdot\vec{s}} \left\langle \Delta\alpha^2 + \Delta\beta^2 \right\rangle_{\text{orbital}} \right\rangle_{\text{spin}} \tag{60}$$

Notice that in Eq. (60) the ISF was replaced by $\vec{n}_0$ consistent with a large degree of polarization. This means that we average $\Delta\alpha^2$ and $\Delta\beta^2$ without having to worry about the dependence of the ISF on phase space.

Here since we intend to use the one-turn map gotten through the integration of the SDE, we look at the one-turn map fluctuations. We start with Eq. (106) from appendix E augmented by the deterministic map orbital M:

$$\vec{z}^1 = M\left(\vec{z}^0 + F\delta\vec{r}\right) \tag{61}$$

In the absence of damping and fluctuation, the map M preserves the polarization. Now consider the difference between the map with and without radiation evaluated at the initial position of the tracking

$$\begin{aligned} \Delta\vec{z} &= M_s^{-1}\left\{M\left(\vec{z}^0 + F\delta\vec{r}\right) - M_s\vec{z}^0\right\} \\ &= \underbrace{\left\{M_s^{-1}M - I\right\}\vec{z}^0}_{\text{damping}} + F\delta\vec{r} \end{aligned} \tag{62}$$

Using Eq. (62), following Barber, we computed the change of $\vec{s}$ due to changes of the vectors $\vec{m}$ and $\vec{l}$. Let us compute for $\vec{m}$, the result being the same for $\vec{l}$. We start with Eqs. (56) and (62):

$$\begin{aligned} (\alpha + \Delta\alpha)^2 = \vec{s}\cdot\vec{m}^{\ 2} &= \left(\alpha + \vec{s}\cdot\vec{\mu}^i\Delta z_i\right)^2 \\ &= \alpha^2 + 2\alpha\vec{s}\cdot\vec{\mu}^i\Delta z_i + \left(\vec{s}\cdot\vec{\mu}^i\Delta z_i\right)^2 \end{aligned} \tag{63}$$

The average $\Delta\alpha$ over the distribution at equilibrium is zero by construction. So we finally get:

$$\begin{aligned}\left\langle \Delta\alpha^2 \right\rangle_{\text{orbital}} &= \left\langle \vec{s}\cdot\vec{\mu}^i \Delta z_i \ \vec{s}\cdot\vec{\mu}^j \Delta z_j \right\rangle \\ &= \left\langle \vec{s}\cdot\vec{\mu}^i \left|\left\{M_s^{-1}M - I\right\}\vec{z}^0\right|_i \vec{s}\cdot\vec{\mu}^j \left|\left\{M_s^{-1}M - I\right\}\vec{z}^0\right|_j \right\rangle \\ &\quad + \left\langle \vec{s}\cdot\vec{\mu}^i |F\delta\vec{r}|_i \vec{s}\cdot\vec{\mu}^j |F\delta\vec{r}|_j \right\rangle \end{aligned} \tag{64}$$

Denoting the damping matrix by

$$D = M_s^{-1}M - I \tag{65}$$

we can simplify Eq. (64) for a distribution at equilibrium:

$$\begin{aligned}\left\langle \Delta\alpha^2 \right\rangle_{\text{orbital}} &= \left\langle \vec{s}\cdot\vec{\mu}^i \Delta z_i \ \vec{s}\cdot\vec{\mu}^j \Delta z_j \right\rangle \\ &= \left\langle \vec{s}\cdot\vec{\mu}^i D_{ia} z_a^0 \vec{s}\cdot\vec{\mu}^j D_{jb} z_b^0 \right\rangle + \left\langle \vec{s}\cdot\vec{\mu}^i E_{ij} \vec{s}\cdot\vec{\mu}^j \right\rangle \\ &= \left\langle \vec{s}\cdot\vec{\mu}^i D_{ia}\Sigma_{ab} D_{jb} \vec{s}\cdot\vec{\mu}^j \right\rangle + \left\langle \vec{s}\cdot\vec{\mu}^i E_{ij} \vec{s}\cdot\vec{\mu}^j \right\rangle \\ &= s_c s_d \mu_c^i \mu_d^j \left\{D_{ia}\Sigma_{ab}D_{jb} + E_{ij}\right\} \end{aligned} \tag{66}$$

We are now ready for the final step using Eqs. (66) and (60). In particular we must perform the spin average:

$$\left\langle \frac{s_c s_d}{n_0\cdot s} \right\rangle \approx \left\langle \frac{(n_0\cdot s)^2 n_{0c} n_{0d}}{n_0\cdot s} \right\rangle \approx P\ n_{0c} n_{0d}. \tag{67}$$

In Eq. (67), again following Barber, we assumed that α and β are small. So the final result is:

$$\begin{aligned}\frac{1}{P}\frac{dP}{dn} &= -\frac{1}{2}\left\{\vec{n}_0\cdot\vec{\mu}^i\vec{n}_0\cdot\vec{\mu}^j + \vec{n}_0\cdot\vec{\lambda}^i\vec{n}_0\cdot\vec{\lambda}^j\right\}\left\{D_{ia}\Sigma_{ab}D_{jb} + E_{ij}\right\} \\ &\approx \underbrace{\frac{1}{2}\left\{\vec{n}_0\cdot\vec{\mu}^i\vec{n}_0\cdot\vec{\mu}^j + \vec{n}_0\cdot\vec{\lambda}^i\vec{n}_0\cdot\vec{\lambda}^j\right\}E_{ij}}_{\text{Neglecting damping}}. \end{aligned} \tag{68}$$

Eq. (68) expresses the damping rate per turn. It is trivial to find the damping rate in seconds using the circumference C and the speed of light c: the result is:

$$\tau_{dep}^{-1} = \frac{c}{C}\frac{1}{P}\frac{dP}{dn} = -\frac{c}{C}\frac{1}{2}\left\{\vec{n}_0\cdot\vec{\mu}^i\ \vec{n}_0\cdot\vec{\mu}^j + \vec{n}_0\cdot\vec{\lambda}^i\ \vec{n}_0\cdot\vec{\lambda}^j\right\}E_{ij}. \tag{69}$$

3.6.3 Depolarization rate in terms of $\vec{\nu} = \frac{d\vec{n}}{d\vec{z}}$

In general we know that the depolarization rate cannot depend on the vectors $\vec{m}$ or $\vec{l}$: these vectors are not uniquely defined. Ultimately averages over the beam can only depend on $\vec{n}$. In our expressions, the derivatives of $\vec{m}$ and $\vec{l}$ enter. These derivatives can be explicitly related to $\vec{\nu} = \frac{d\vec{n}}{d\vec{z}}$ using the relations in Eqs. (57b) and (57c). Then Eq. (69) becomes:

$$\tau_{dep}^{-1} = \frac{c}{C}\frac{1}{P}\frac{dP}{dn} = -\frac{c}{C}\frac{1}{2}\left\{\vec{m}_0\cdot\vec{\nu}^{i}\ \vec{m}_0\cdot\vec{\nu}^{j} + \vec{l}_0\cdot\vec{\nu}^{i}\ \vec{l}_0\cdot\vec{\nu}^{j}\right\}E_{ij}. \tag{70}$$

It is interesting to compare the formula of Eq. (70) with the formula I dug out of the BMAD manual, namely formula (20.21), which if I am not mistaken, was extracted from Barber and Ripken's contribution to the famous "Chao and Tigner" hanbook:

$$\tau_{dep}^{-1} = \frac{5\sqrt{3}}{8}\frac{r_e\gamma^5\hbar}{m}\frac{1}{C}\oint ds\ \left\langle g^3\frac{11}{18}\left|\frac{\partial\vec{n}}{\partial\delta}\right|^2\right\rangle. \tag{71}$$

These things look so different at first glance. However the formula applicable to PTC or PTC maps presupposed that we can solve for the one-turn fluctuation matrix E_{ij}. However, since it is a result of first order perturbation theory, we can apply it to an infinitesimal fluctuation.

Let us first assume, that the fluctuation is only in the energy direction, i.e., the index "6" in BMAD. This not always true in canonical variables but it is true for standard magnets outside fringe field areas. Then Eq. (70) becomes:

$$d\tau_{dep}^{-1} = -\frac{c}{C}\frac{1}{2}\left\{\left(\vec{m}_0\cdot\vec{\nu}^{6}\right)^2 + \left(\vec{l}_0\cdot\vec{\nu}^{6}\right)^2\right\}E_{66}ds. \tag{72}$$

Next we use again the equations in (57) to rewrite $\vec{\nu}^6 = \frac{\partial\vec{n}}{\partial\delta}$ in terms of $\vec{m}_0$ and $\vec{l}_0$:

$$\frac{\partial\vec{n}}{\partial\delta} = \underbrace{\vec{n}_0\cdot\vec{\nu}^{6}}_{=0\ \text{Eq. (57a)}}\ \vec{n}_0\ +\ \vec{m}_0\cdot\vec{\nu}^{6}\ \vec{m}_0\ +\ \vec{l}_0\cdot\vec{\nu}^{6}\ \vec{l}_0$$

$$\Downarrow$$

$$\left|\frac{\partial\vec{n}}{\partial\delta}\right|^2 = \left(\vec{m}_0\cdot\vec{\nu}^{6}\right)^2 + \left(\vec{l}_0\cdot\vec{\nu}^{6}\right)^2. \tag{73}$$

$$d\tau_{dep}^{-1} = -\frac{c}{C}\frac{1}{2}\left\{\left(\vec{m}_0\cdot\vec{\nu}^6\right)^2+\left(\vec{l}_0\cdot\vec{\nu}^6\right)^2\right\}E_{66}ds. \tag{74}$$

It then follows that

$$\tau_{dep}^{-1} = -\frac{c}{C}\frac{1}{2}\oint\left|\frac{\partial\vec{n}}{\partial\delta}\right|^2 E_{66}ds. \tag{75}$$

Eq. (75) is obviously the same as Eq. (71) of the BMAD manual.

It is obvious from our discussion here that full depolarization of Eq. (69) (or Eq. (70)) can be rewritten as:

$$\tau_{dep}^{-1} = \frac{c}{C}\frac{1}{P}\frac{dP}{dn} = -\frac{c}{C}\frac{1}{2}\frac{\partial\vec{n}}{\partial z_i}\cdot\frac{\partial\vec{n}}{\partial z_j}E_{ij}. \tag{76}$$

3.7 What code can use Eq. (76)?

To use Eq. (70) a tracking code must provide

1. A full solution of the stochastic equations namely E_{ij}
2. Spin tracking including orbital effects
3. Normal form for spin at least around $\vec{n}_0$ to first order: the SLIM formalism at a minimum.

The code BMAD (without PTC) fails on item 1 and 3. The code SAD has item 1 but no spin at all. The code SLIM fails on item 1.

If only the fluctuation along the "principal solution" is computed, then we can make large error in the computation of the depolarization. This is not true for plain beam size calculations: equilibrium emittances and the Ripken-Lattice function suffices.

Therefore it appears that only PTC at this point can compute depolarization with the stochastic map.

3.8 How it is trivially done in PTC

In the code below, I compute (putting back the implicit sums of repeated indices):

$$\tau_{dep}^{-1} = \frac{c}{C}\frac{1}{P}\frac{dP}{dn} = -\frac{c}{C}\frac{1}{2}\sum_{i=1,6}\sum_{j=1,6}\frac{\partial \vec{n}}{\partial z_i}\cdot\frac{\partial \vec{n}}{\partial z_j}E_{ij}. \tag{77}$$

Line 0 computes the normal form with stochastic radiation : E_{ij} is extracted.

Line 1 Extract the spin map properly factorized as in Eq. (54). The SO(3) matrix is computed from the quaternion "a" in `as`.

Line 2 computes the ISF using the normal form.

Below line 3, is the computation of Eq. (77): the ".d." operator takes the derivatives of the ISF with respect to `i1`.

The final result of this calculation is, on my example from Oleksii Beznozov, `P(1 million turn) = 0.99011`. The numerical results using tracking with a linear map for 1 million turns is : `P=0.99014` using 1000 particles.

```
!!!   0
!  calculation of beam size using envelope theory
call radia_new(r,1,state0,"radia.dat",sij=sij,sijr=sijr,e_ij=e_ij)
                    .
!!! 1
call c_full_canonise(nf%atot,U_1,as,a0,a1,a2)
call makeso3(as)

!!!   2
ISF=2
ISF=as%s*ISF


!!! 4
cut=0
do i1=1,3
 do i2=1,6
  do i3=1,6

    cut =  cut - (ISF%v(i1).d.i2)*e_ij(i2,i3)*(ISF%v(i1).d.i3)/2

  enddo
 enddo
enddo


write(6,*)"Polarization with dn/dz+final polarization after ",nturns," turns"
write(6,*)  cut,exp(nturns*cut)
```

4 Obsolete Types

5 overloading

6 subpackage

1. `c_spinmatrix=exp(c_spinor,c_spinmatrix)` $\rightarrow exp\left(\omega\times\right)s$ Useful for constant spin matrix: used in the linear normal form with $SO(3)$.
2. `c_vector_field = exp(c_damap,h:`0`,epso:computed )`
3. `c_quaternion=exp(c_quaternion,c_quaternion)` $\rightarrow exp\left(q_2\right)q_1$ Useful for constant quaternion map: used in the linear normal form with quaternion.

A Lie and Composition Operator

The operator

$$\widehat{F}q = F\cdot\nabla q + qf \tag{78}$$

is the general operator used by FPP. If we let this operator act on the identity map I_{total}, which contains the orbital identity and the unit quaternion, we obtain a map denoted by (m,q):

$$\begin{aligned} &\text{if} \quad I_{total} = (I_{orbital}, 1) \\ &\text{then} \quad \exp\left(\widehat{F}\right) I_{total} = \left(\exp\left(F\cdot\nabla\right) I_{orbital}, \exp\left(\widehat{F}\right) 1\right) = (m,q) \end{aligned} \tag{79}$$

The map (m,q) is a regular spin-orbital map. Let us assume that we concatenate this map with another spin-orbital map (n,l). Then the total map is just:

$$U = (n,l)\circ(m,q) = (n\circ m, l\circ m\ q) \tag{80}$$

However, thanks to our right to left quaternion operator in Eq. (78), the full Lie map preserves the substitution rules which are known to be correct for a pure orbital Lie operator:

$$U = \exp\left(\widehat{F}\right)(n, l) = (n \circ m, l \circ m\ q)$$

or

$$l \circ m\ q = \exp\left(\widehat{F}\right) l \tag{81}$$

The map of Eq. (79) is known in mathematics as a composition operator. In dynamical systems, which is a branch of physics, it is known as the Koopman operator. It arises naturally if one is after invariants of the motion, see reference [3]. For example, suppose ε is an orbital invariant,like the Courant-Snyder invariant, then by definition:

$$\epsilon = \epsilon \circ m = \exp\left(\widehat{F}\right) \epsilon \tag{82}$$

B Transformation of a Lie operator

Consider the Lie map $\mathcal{A}$ operator associated with the phase space map $A = (a, \alpha)$. It acts on a phase space map $M = (m, q)$ (see Sec. (A)) following the formula:

$$\mathcal{A}(m, q) = (m \circ \alpha, q \circ a\alpha) \tag{83}$$

How does a vector field $\widehat{F}$, as in Eq. (78), is transformed by the map of Eq. (83)? This has to be done by a similarity transformation:

$$\mathcal{A}\left(F \cdot \nabla + \hat{f}\right)\mathcal{A}^{-1} = \widetilde{F} \cdot \nabla + \hat{\widetilde{f}} \tag{84}$$

The answer for $\widetilde{F}$ and $\widetilde{f}$ is:

$$\begin{aligned} \widetilde{F}_k &= \left(F_i \partial_i a_k^{-1}\right) \circ a \\ \hat{f} &= \left\{\left(\widetilde{F}_k \partial_k \alpha^{-1}\right) + \alpha^{-1} f \circ a\right\} \circ a \end{aligned} \tag{85}$$

C Lie bracket and Lie bracket operator

We first evaluate the commutator of two line operators $\widehat{F}$ and $\widehat{H}$:

$$\begin{aligned}\left[\widehat{F},\widehat{H}\right] &= \left[F\cdot\nabla+\hat{f},H\cdot\nabla+\hat{h}\right]\\ &= G\cdot\nabla+\hat{g}\end{aligned} \tag{86}$$

The answer for $\widehat{G}$ is:

$$\begin{aligned}G &= \langle F,H\rangle = F\cdot\nabla H - H\cdot\nabla F\\ g &= [h,f]+F\cdot\nabla h - G\cdot\nabla f\end{aligned} \tag{87}$$

We can define a generalised Lie bracket between the vectors and the quaternions defining the Lie operators:

$$\underbrace{(G,g)=\langle(F,f),(H,h)\rangle}_{\text{FPP's c\_vector\_field}} \tag{88}$$

In the code, FPP, it is truly Eq. (88) which is implemented: vector fields and Lie maps are not represented by a big linear matrix acting on monomials but by differential operators.

Finally we can defined a Lie map which acts on the vector field. Going back to Eq. (84), let us assume that $\mathcal{A}$ is represented by a Lie exponent $\widehat{A}$,

$$\mathcal{A}=\exp\left(\widehat{A}\right) \tag{89}$$

then we must have:

$$\begin{aligned}\mathcal{A}\widehat{F}\mathcal{A}^{-1} &= \exp\left(\widehat{A}\right)\widehat{F}\exp\left(-\widehat{A}\right)\\ &= \exp\left({}^{\#}_{\#}\widehat{A}^{\#}_{\#}\right)\widehat{F}\end{aligned} \tag{90}$$

where

$${}^{\#}_{\#}\widehat{A}^{\#}_{\#}\widehat{F}=\left[\widehat{A},\widehat{F}\right]=\widehat{A}\widehat{F}-\widehat{F}\widehat{A} \tag{91}$$

and finally, thanks to the homomorphism between the commutators and the Lie bracket:

$$\widehat{\widehat{F}} = \exp\left(\begin{matrix}\# & \widehat{A}^{\#} \\ \# & \#\end{matrix}\right) \widehat{F} = \text{Operator}\left\{\exp\left(:\,(A,a)\,:\right)(F,f)\right\} \tag{92}$$

The word "Operator" in front of the curly bracket in Eq. (92) indicates that the object inside the brackets has a "hat" on top of it and is thus a Lie operator. The colon around $:\,(A,a)\,:$ indicates, à la Dragt, that a Lie bracket must be taken with what follows as defined by Eqs. (87) and (88). This Lie bracket is in FPP and is defined by the Fortran operator "`.lb.`". So we have:

$$\left(\widetilde{F},\widetilde{f}\right) = \text{Operator}\left\{\exp\left(:\,(A,a)\,:\right)(F,f)\right\} \tag{93}$$

Eq. (93) is the complete equivalent of Eq. (85). In Eq. (93) the Lie operator (vector field) of $\mathcal{A}$ is known while in Eq. (85) we know only the Taylor spin-orbital maps (A,a).

D Logarithm of a map and "DA" self-consistency

We first display the recursive loop for the calculation of the Lie operator of a spin-orbital map with quaternions. Then we explain the type of validation of all our operators in the differential algebraic (no feed down) that can be performed by FPP to weed out the theory of programming mistakes.

D.1 The logarithm of a Lie spin-orbital map

If a matrix M is near the identity, the following series converges:

$$\log(M) = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{(M-1)^n}{n} \tag{94}$$

Of course this applies trivially to a linear map of phase space. What about a Lie map? Consider a nonlinear map (M,q) and let us assume that it can

be approximated by a Lie exponent:

$$\begin{aligned}(m,q) &= \exp\left(F\cdot\nabla+\hat{f}\right)(I,1)\\ &\approx \left(I\ +\ F\cdot\nabla I+\cdots,1+\hat{f}1+\right)\\ &\approx (I\ +\ F+\cdots,1+f)\end{aligned}\tag{95}$$

The Lie map $\exp\left(F\cdot\nabla+\hat{f}\right)$ acts on the space of functions and thus it is possible to write a matrix for it using a basis made of monomials. For example, in 1-d-f, for polynomials of degree n_0, the space of polynomials is of dimension $\frac{(n_0+2)!}{n_0!2!}$. Therefore the matrix for the orbital of $\exp(F\cdot\nabla)$ is of dimension $(\frac{(n_0+2)!}{n_0!2!})^2$. This matrix, as we pointed out in reference [3], is the transpose of the matrix which propagates the moments.

Here we do not have matrices in the nonlinear case, so we must be a little more resourceful. First one notices that Eq. (95) gives us a trivial approximation of the vector field:

$$(F,f)\approx(m,q)-(I,1)\tag{96}$$

We proceed further by assuming that the vector field (F,f) is already known to some order and that the result is (F^k,f^k). We can write:

$$\exp\left(-F^k\cdot\nabla-f^k\right)(m,q)=(I,1)+(t,u)\tag{97}$$

This algorithm starts with $(t,u)=(m-I,q-1)$. At the end we expect t and u to be zero. My goal is to find a relatively fast algorithm. The first step is to find a vector field which can reproduce the map $(I+t,1+u)$. To second-order in (t,u), We can write:

$$\begin{gathered}\exp\left(\tau_3\cdot\nabla+\ \hat{\theta}_3\right)(I,1)=(I,1)+(t,u)\\ \text{where}\quad(\ \tau_3,\hat{\theta}_3\ )=(t,u)+\varepsilon_2\end{gathered}\tag{98}$$

Solving for ε_2, we get:

$$\varepsilon_2=-\frac{1}{2}\left\{t\cdot\nabla+\hat{u}\right\}(t,u)=\left(t\cdot\nabla t,t\cdot\nabla u+u^2\right)\tag{99}$$

We can then rewrite Eq. (98) using Eq. (99):

$$\exp\left(-F^k\cdot\nabla-\hat{f}^k\right)\mathcal{M}\approx\exp\left(\tau_3\cdot\nabla+\hat{\theta}_3\right)$$
$$\Downarrow$$
$$\mathcal{M}=\exp\left(F^k\cdot\nabla+\hat{f}^k\right)\exp\left(\tau_3\cdot\nabla\ +\hat{\theta}_3\right) \tag{100}$$

Now we can apply the Baker-Campbell-Hausdorff formula (CBH) to go to the next step in the iteration:

$$\left(F^{k+1},f^{k+1}\right)=\left(F^k,f^k\right)+\left(\tau_3,\theta_3\right)+\frac{1}{2}\left\langle\left(F^k,f^k\right),\left(\tau_3,\theta_3\right)\right\rangle+\text{higher order}\cdots \tag{101}$$

The Lie bracket in Eq. (101) was defined in Eqs. (87) and (88).

D.2 Validation of all these "DA" operators

Below is a code fragment which allows us to illustrate the results of Secs. (A), (B) and (C):

```
A=exp(vf)                                   implements Eq. (79)
f=c_logf_spin(my_map)                       implements the result of Sec. (D.2)

id=A**(-1)*my_map*A
f_tilde=c_logf_spin(id)    Uses the logarithm to get the results of Eq. (84)


vf_s=A*f                                    Implements Eq. (85)
vf_a=exp(vf,f)                              Implements Eq. (93)

vf_s=vf_s-f_tilde          Compares the logarithm of Eq. (84) with Eq. (85)
call c_full_norm_vector_field(vf_s,norm)
print *, "norm",norm
vf_a=vf_a-f_tilde          Compares the logarithm of Eq. (84) with Eq. (93)
call c_full_norm_vector_field(vf_a,norm)
```

```
print *, "norm",norm
```

The norms produced in the above fragment are zero to machine precision provided that map and vector fields involved have **no** constants parts. All differential operators produce self-consistent results. For the mathematicians, this result comes from the fact that all the Lie operator form a differential algebra. For physicists, they are self-consistent because they are all around an orbit with no feed down effects.

E Stochastic tracking using the beam envelope map of PTC

Starting with the one-turn stochastic map of Eq. (7), i.e,

$$\Sigma^f = M\left(\Sigma + E\right) M^{\mathrm{T}} \tag{102}$$

we want to construct a random number generator which, coupled with the matrix M, will generate the equilibrium beam sizes of Eq. (15). To do this, we diagonalize the fluctuation matrix E rather than M:

$$E = F\delta F^{\mathrm{T}}. \tag{103}$$

where δ is a diagonal matrix such that $\delta_{11} = \delta_{22} = k_1$, $\delta_{33} = \delta_{44} = k_2$ and $\delta_{55} = \delta_{66} = k_3$.

For a one-turn map, the moment kick E lies on some hyper-ellipsoid in 6 dimension and therefore the diagonalization of E by a symmetric matrix is equivalent to the diagonalization a stable harmonic oscillator by a symplectic matrix.

Finally, suppose we generate a stochastic ray $\vec{x}_s$ as follows:

$$\vec{x}_s = \vec{x} + F\delta^{1/2}\vec{r} \tag{104}$$

where $\vec{r}$ is a random vector composed of 6 random variables of variance one:

$$\vec{r} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \end{pmatrix} \tag{105}$$

then it is easy to show that

$$\langle x_{s;i} x_{s;j} \rangle = \left\langle \left(x_i + F\delta^{1/2}\vec{r}\big|_i \right) \left(x_j + F\delta^{1/2}\vec{r}\big|_j \right) \right\rangle = E_{ij}. \tag{106}$$

Therefore if Eq. (104) is used during stochastic tracking with a linear map, we expect that the correct distribution will be produced. This theory is valid only for a linear map. It is possible to produce a nonlinear beam envelope map for all the moments to a given order, however we are not aware of a diagonalization procedure similar to Eq. (103).

```
C:\document\my_tex_papers\fpp_manual\polymorphism_only\z_why_polymorphism.f90
C:\document\my_tex_papers\fpp_manual\polymorphism_only\z_concat_da_c_damap.f90
C:\document\my_tex_papers\fpp_manual\polymorphism_only\z_my_code.f90
resonance
C:\document\my_tex_papers\fpp_manual\polymorphism_only\z_spin_m1_1_1_quaternion.f90
C:\document\my_tex_papers\fpp_manual\polymorphism_only\z_spin_normal_form.f90
```

References

[1] E. Forest, J. Phys. A: Math. Gen. **39**, 5321 (2006).

[2] E. Forest, *From Tracking Code to Analysis, -Generalised Courant-Snyder Theory for any Accelerator Model* (Springer Japan, Tokyo, Japan, 2016).

[3] E. Forest, *Beam Dynamics: A New Attitude and Framework* (Harwood Academic Publishers, Amsterdam, The Netherlands, 1997).