Contents

Manual for FPP: The Fully Polymorphic Package

Étienne Forest
Tsukuba, Japon

September 13, 2018

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. (4)), 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.

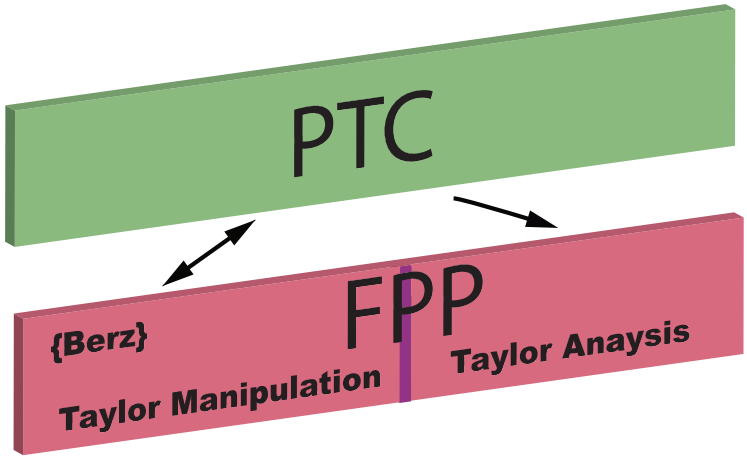

Figure 1: The Fully Polymorphic Package (FPP) part of the FPP/PTC library provides manipulation and analysis of Taylor series and maps and the Polymorphic Tracking Code (PTC) part provides the physics from which accelerators can be analyzed.

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 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. (3).

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:

```
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 -0.10000000000000000       1  0
   1  0.90000000000000000       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 infeasible 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):

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

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
  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 their 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_i_j(6,6)

We will say little about the `E_i_j(6,6)` structure except that it 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 final closed trajectory (with classical radiation) and `E_i_j(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.

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 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.

As we said, PTC tracks `probe` or `probe_8`. It is very easy to modify the above routines to mimic this feature of PTC:

```
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)

interface track
  module procedure trackr
  module procedure trackp
end interface

contains
      .
      .
      .
subroutine trackr(p) ! for probe
implicit none
type(probe) :: p
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
end subroutine trackr

subroutine trackp(p) ! for probe_8
implicit none
type(probe_8) :: p
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
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[4] 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:

$$\mathbf{z} = \begin{pmatrix} 1 & 1 \\ -0.1 & 0.9 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} + O\left(z^2 \right) \tag{7}$$

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

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

2. One should not confuse a set with an algebraic structure which the set itself.

Item 2 requires an explanation. Take for example a pair of real numbers from the set $\mathbb{R} \times \mathbb{R}$. A priori we have no idea what structures are imposed on this pair of numbers. Indeed the structure could be a complex number field, a ring of differentials (running TPSA to order 1 with one parameter), a one-dimensional complex vector space, a two dimensional real vector space, a twice infinite dimensional vector space on the field of rationals, etc... In a code (or in a 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 and 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[5] 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

[5]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.

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. (7) can be diagonalize into:

$$\Lambda = \begin{pmatrix} \exp(-i\ \mu) & 0 \\ 0 & \exp(\ i\ \mu) \end{pmatrix} \text{ where } \mu\ =\ 0.317560429291521 \qquad (8)$$

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.[6]

1.7.2 The type `c_damap`

~~The type `c_damap` is defined as~~

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

```
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
```

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. (8) 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[7] 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

[7]This is view point of COSY INFINITY of Berz but we reject it in ring dynamics 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[8] 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 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

- DA operations are 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.
- TPSA operations take into account the constant part and the results change as a function of the order.

Let us look at our little code again

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

```
  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
```

with the parameters

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

We run the code and invoke DA maps:

```
 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  9.951207934035656E-020
 The order of the Taylor series ?
1
 Tpsa =0 DA=1
1

           2  Dimensional map

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

   0 -0.1127016653792583E-01   0.000000000000000       0  0
   1   1.000000000000000       0.000000000000000       1  0
   1   1.000000000000000       0.000000000000000       0  1


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

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

  0 -0.1084202172485504E-18   0.000000000000000       0  0
  1 -0.7745966692414835E-01   0.000000000000000       1  0
  1  0.9225403330758517       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:

```
           2  Dimensional map

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

  0 -0.1127016653792583E-01   0.000000000000000       0  0
  1   1.000000000000000       0.000000000000000       1  0
  1   1.000000000000000       0.000000000000000       0  1


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

  0 -0.1084202172485504E-18   0.000000000000000       0  0
  1 -0.7745966692414835E-01   0.000000000000000       1  0
  1  0.9225403330758517       0.000000000000000       0  1
  2  -1.000000000000000       0.000000000000000       2  0
  2  -2.000000000000000       0.000000000000000       1  1
  2  -1.000000000000000       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 map

Properties, NO =    2, NV =    2, INA =   64
******************************************

  0 -0.1127016653792583E-01   0.000000000000000       0  0
  1  0.9225403330758517       0.000000000000000       1  0
```

```
  1   1.922540333075852       0.000000000000000       0  1
  2  -1.000000000000000       0.000000000000000       2  0
  2  -2.000000000000000       0.000000000000000       1  1
  2  -1.000000000000000       0.000000000000000       0  2


Properties, NO =    2, NV =    2, INA =   63
*********************************************

  0 -0.1084202172485504E-18   0.000000000000000       0  0
  1 -0.1489193338482967       0.000000000000000       1  0
  1  0.7736209992275551       0.000000000000000       0  1
  2  -1.773620999227555       0.000000000000000       2  0
  2  -5.392322664606813       0.000000000000000       1  1
  2  -4.618701665379258       0.000000000000000       0  2

 No Spin Matrix
 c_quaternion is identity
No Stochastic Radiation
 Two periods map

          2  Dimensional map

Properties, NO =    2, NV =    2, INA =   64
*********************************************

  0 -0.1127016653792583E-01   0.000000000000000       0  0
  1  0.9225403330758517       0.000000000000000       1  0
  1   1.922540333075852       0.000000000000000       0  1
  2  -1.000000000000000       0.000000000000000       2  0
  2  -2.000000000000000       0.000000000000000       1  1
  2  -1.000000000000000       0.000000000000000       0  2


Properties, NO =    2, NV =    2, INA =   63
*********************************************

  0 -0.1084202172485504E-18   0.000000000000000       0  0
  1 -0.1489193338482967       0.000000000000000       1  0
  1  0.7736209992275550       0.000000000000000       0  1
  2  -1.773620999227555       0.000000000000000       2  0
  2  -5.392322664606813       0.000000000000000       1  1
  2  -4.618701665379258       0.000000000000000       0  2
```

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

```
two_period_map=one_period_map*one_period_map
print *, " one_period_map*one_period_map "
call PRINT(two_period_map)
call track(polymorphic_probe)
two_period_map=polymorphic_probe
print *, " Two periods map "
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_map .o. one_period_map

          2  Dimensional map

Properties, NO =    2, NV =    2, INA =   64
*******************************************

  0 -0.2254033307585166E-01   0.000000000000000       0  0
  1  0.9225403330758517       0.000000000000000       1  0
  1   1.922540333075852       0.000000000000000       0  1
  2  -1.000000000000000       0.000000000000000       2  0
  2  -2.000000000000000       0.000000000000000       1  1
  2  -1.000000000000000       0.000000000000000       0  2


Properties, NO =    2, NV =    2, INA =   63
*******************************************

  0  0.7459666924148337E-03   0.000000000000000       0  0
  1 -0.1281249674648599       0.000000000000000       1  0
  1  0.8169556986868436       0.000000000000000       0  1
  2  -1.796161332303407       0.000000000000000       2  0
  2  -5.437403330758516       0.000000000000000       1  1
  2  -4.641241998455110       0.000000000000000       0  2


 Two periods map

          2  Dimensional map

Properties, NO =    2, NV =    2, INA =   64
*******************************************

  0 -0.1127016653792583E-01   0.000000000000000       0  0
  1  0.9225403330758517       0.000000000000000       1  0
  1   1.922540333075852       0.000000000000000       0  1
  2  -1.000000000000000       0.000000000000000       2  0
  2  -2.000000000000000       0.000000000000000       1  1
  2  -1.000000000000000       0.000000000000000       0  2


Properties, NO =    2, NV =    2, INA =   63
*******************************************

  0 -0.1084202172485504E-18   0.000000000000000       0  0
  1 -0.1489193338482967       0.000000000000000       1  0
  1  0.7736209992275550       0.000000000000000       0  1
  2  -1.773620999227555       0.000000000000000       2  0
```

```
  2  -5.392322664606813       0.000000000000000       1  1
  2  -4.618701665379258       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=(0,0)`.

```
 Closed Orbit from Tracking  -1.127016653792583E-002  9.951207934035656E-020
 Closed Orbit using TPSA map -1.000000000000000E-002  0.000000000000000E+000
 Linear using TPSA map around z=(0,0)

          2  Dimensional map

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

  1   1.000000000000000       0.000000000000000       1  0
  1   1.000000000000000       0.000000000000000       0  1


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

  0 -0.1000000000000000E-02   0.000000000000000       0  0
  1 -0.1000000000000000       0.000000000000000       1  0
  1  0.9000000000000000       0.000000000000000       0  1

 No Spin Matrix
 c_quaternion is identity
No Stochastic Radiation

 Linear map around computed by TPSA inversion

          2  Dimensional map

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

  1   1.000000000000000       0.000000000000000       1  0
  1   1.000000000000000       0.000000000000000       0  1
```

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

   1 -0.1000000000000000       0.000000000000000       1  0
   1  0.9000000000000000       0.000000000000000       0  1
```

The approximate closed orbit is found by solving

$$F(\mathbf{z}) = \mathrm{T}(\mathbf{z}) - \mathbf{z} = 0 \qquad \Longrightarrow \qquad \mathbf{z}_{\mathrm{c}} = \mathrm{F}^{-1}(0) \tag{9}$$

This is done by the code (the maketpsa routine is explained near Eq. (13)):

```
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 * ," Linear using TPSA map around z=(0,0)"
newton_map=one_period_map.cut.2
call print(newton_map)

one_period_map= one_period_map.o.to_closed_orbit  <---- notice .o. NOT *

one_period_map=(to_closed_orbit.oo.(-1)).o.one_period_map  <---- notice .o. and .oo.

newton_map=one_period_map.cut.2       <---- order higher than 1 are cut


 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

```
  Linear map around computed by TPSA inversion

           2  Dimensional map

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

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

  1   1.000000000000000       0.000000000000000       1  0
  1   1.000000000000000       0.000000000000000       0  1


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

  0 -0.2007594574035456E-08   0.000000000000000       0  0
  1 -0.7745971876000000E-01   0.000000000000000       1  0
  1  0.9225402812400000       0.000000000000000       0  1
```

To the extent that we consider the integrator model to be realistic, i.e. exact in the Talman sense (see reference [?]), 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.

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:

$$M(x) = ax + b \qquad \text{and} \qquad N(x) = cx + d \tag{10}$$

Then $M * N$ will be:

$$(M * N)(x) = ac\ x + b \tag{11}$$

The TPSA concatenation will be given by

$$(M.o.N)\,(x) = ac\;x + ad + b \tag{12}$$

In general, Eq. (12) makes little sense because the variable x is around some orbit while the TPSA map represents the full orbit in the original coordinates. However, if the original coordinates are known, they can be fed into the `c_damap` component `x0` and the map can be transformed as follows:

$$\begin{aligned}\widetilde{M}\,(x) &= \text{maketpsa}\,(M)\,(x) = a(x - M\%x0) + b\\ \widetilde{N}\,(x) &= \text{maketpsa}\,(N)\,(x) = c(x - N\%x0) + d\end{aligned} \tag{13}$$

The component flag `tpsa` of `c_damap` is set to true and the map concatenation behaves as in Eq. (13). 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 is an example code which does a calculation 4 differents ways in addition to exact tracking from PTC.

```
M1=MAKETPSA(my_map)

x=1.01_dp*closed
ray=x
ray%x0(1:6)=closed
ray=m1.o.ray
write(6,*) " Using a TPSA map "
                                    call print(ray)

write(6,*) " Using a DA map "
ray=x
ray%x0(1:6)=closed
ray=my_map.o.ray
                                    call print(ray)

write(6,*) " Using a TPSA map and SO(3)"
CALL MAKESO3(m1)
use_quaternion=.FALSE.
ray=x
ray%x0(1:6)=closed
ray=m1.o.ray
                                    call print(ray)
write(6,*) " Using a DA map and SO(3) "
CALL MAKESO3(MY_MAP)
use_quaternion=.FALSE.
ray=x
ray%x0(1:6)=closed
ray=my_map.o.ray
                                    call print(ray)

use_quaternion=.true.
xs0=x
call propagate(als,xs0,state,fibre1=1)
                                   call print(xs0)
```

`my_map` is a one-turn map produced by PTC. `m1` is a TPSA map as described by Eq. (13). This particular run used quaternions. So we also turn both `my_map` and `m1` into maps using $SO(3)$. Finally we compute the exact result as seen by PTC. In the limit of large order, the Taylor series results converge, as expected, to PTC's results.

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 complex and we refer the reader to reference [?] 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.

$$M = \exp\left(F \cdot \nabla\right) I \tag{14}$$

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

$$P = \exp\left(F \cdot \nabla\right) N \Longleftrightarrow P = N \circ M \tag{15}$$

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

$$\widehat{F} q = F \cdot \nabla q + q f \tag{16}$$

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

$$\widehat{F}q = F \cdot \nabla q + qf \tag{17}$$

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 5

Please take note of the of the reverse ordering in Eq. (17). It is a consequence of Eq. (15) 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)`
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 Sec. (7) and Eq. (33).
6. `c_quaternion=exp(c_vector_field,c_quaternion)` $\rightarrow exp\left(\widehat{F}\right) q$
7. `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.
8. `c_spinmatrix=exp(c_spinor,c_spinmatrix)` $\rightarrow exp\left(\omega\times\right) s$ Useful for constant spinmatrix: used in the linear normal form with $SO(3)$.
9. `c_vector_field = exp(c_damap,h:`0`,epso:computed )`

   xxxxxxxxxxxxxxxxxx
10. `exp(c_factored_lie,c_spinmatrix)`

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

2.4 Obsolete Types

3 overloading

4 subpackage

5 Lie Map and Lie operator

The operator

$$\widehat{F}q = F \cdot \nabla q + qf \tag{18}$$

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):

$$\begin{aligned} &\text{if} \quad I_{total} = (I_{orbital}, 1) \\ &\text{then} \quad \exp\left(\widehat{F}\right) I_{total} = \left(\exp\left(F \cdot \nabla\right) I_{orbital}, \exp\left(\widehat{F}\right) 1\right) = (m, q) \end{aligned} \tag{19}$$

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:

$$U = (n, l)\,(m, q) = (n \circ m, l \circ m\ q) \tag{20}$$

However, thanks to our right to left quaternion operator in Eq. (18), 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

$$l \circ m\ q = \exp\left(\widehat{F}\right) l \tag{21}$$

6 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. (5)) following the formula:

$$\mathcal{A}\,(m, q) = (m \circ \alpha, q \circ a\alpha) \tag{22}$$

How does a vector field $\widehat{F}$ as in Eq. (18) if transformed by the map of Eq. (22)? This has to be done by a similarity transformation:

$$\mathcal{A}\left(F \cdot \nabla + \hat{f}\right)\mathcal{A}^{-1} = \widetilde{F} \cdot \nabla + \hat{\widetilde{f}} \tag{23}$$

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

$$\begin{aligned} \widetilde{F}_k &= \left(F_i \partial_i a_k^{-1}\right) \circ a \\ \hat{f} &= \left\{\left(\widetilde{F}_k \partial_k \alpha^{-1}\right) + \alpha^{-1} f \circ a\right\} \circ a \end{aligned} \tag{24}$$

7 Lie bracket and Lie bracket operator

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

$$\begin{aligned} \left[\widehat{F}, \widehat{H}\right] &= \left[F \cdot \nabla + \hat{f}, H \cdot \nabla + \hat{h}\right] \\ &= G \cdot \nabla + \hat{g} \end{aligned} \tag{25}$$

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

$$\begin{aligned} G &= \langle F, H \rangle = F \cdot \nabla H - H \cdot \nabla F \\ g &= [h, f] + F \cdot \nabla h - G \cdot \nabla f \end{aligned} \tag{26}$$

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

$$\underbrace{(G, g) = \langle (F, f), (H, h) \rangle}_{\text{FPP's c\_vector\_field}} \tag{27}$$

In the code, FPP, it is truly Eq. (27) 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 the Lie which acts on the vector field. Going back to Eq. (29), let us assume that $\mathcal{A}$ is represented by a Lie exponent $\widehat{A}$,

$$\mathcal{A} = \exp\left(\widehat{A}\right) \tag{28}$$

then we must have:

$$\mathcal{A}\left(F\cdot\nabla + \hat{f}\right)\mathcal{A}^{-1} = \widetilde{F}\cdot\nabla + \hat{\widetilde{f}} \tag{29}$$

$$\begin{aligned}\mathcal{A}\widehat{F}\mathcal{A}^{-1} &= \exp\left(\widehat{A}\right)\widehat{F}\exp\left(-\widehat{A}\right)\\ &= \exp\left({}^{\#}_{\#}\widehat{A}^{\#}_{\#}\right)\widehat{F}\end{aligned} \tag{30}$$

where

$${}^{\#}_{\#}\widehat{A}^{\#}_{\#}\widehat{F} = \left[\widehat{A},\widehat{F}\right] = \widehat{A}\widehat{F} - \widehat{F}\widehat{A} \tag{31}$$

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

$$\widehat{\widetilde{F}} = \exp\left({}^{\#}_{\#}\widehat{A}^{\#}_{\#}\right)\widehat{F} = \text{Operator}\left\{\exp\left(:\,(A,a)\,:\right)(F,f)\right\} \tag{32}$$

The word "Operator" in front of the curly bracket in Eq. (32) 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. (26) and (27). This Lie bracket is in FPP and is defined by the Fortran operator "`.lb.`". So we have:

$$\left(\widetilde{F},\widetilde{f}\right) = \text{Operator}\left\{\exp\left(:\,(A,a)\,:\right)(F,f)\right\} \tag{33}$$

Eq. (33) is the complete equivalent of Eq. (24). In Eq. (33) the Lie operator (vector field) of $\mathcal{A}$ is known while in Eq. (24) we know only the Taylor spin-orbital maps (A,a).

8 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 check weed out theory or programming mistakes.

8.1 The logarithm of a Lie spin-orbital map

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

$$\log(M) = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{(M-1)^n}{n} \tag{34}$$

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:

$$\begin{aligned}(m, q) &= \exp\left(F \cdot \nabla + \hat{f}\right)(I, 1) \\ &\approx \left(I \ + \ F \cdot \nabla I + \cdots, 1 + \hat{f}1+\right) \\ &\approx (I \ + \ F + \cdots, 1 + f)\end{aligned} \tag{35}$$

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 I pointed out in reference [**?**], is the transpose of the matrix which propagates the moments.

Here I do not have matrices in the nonlinear case, so I must be a little more resourceful. First one notices that Eq. (35) gives us a trivial approximation of the vector field:

$$(F, f) \approx (m, q) - (I, 1) \tag{36}$$

I 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). I can write:

$$\exp\left(-F^k \cdot \nabla - f^k\right)(m, q) = (I, 1) + (t, u) \tag{37}$$

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), I can write:

$$\exp\left(\tau_3 \cdot \nabla + \hat{\theta}_3\right)(I, 1) = (I, 1) + (t, u) \tag{38}$$
$$\text{where} \quad (\tau_3, \hat{\theta}_3) = (t, u) + \varepsilon_2$$

Solving for ε_2, I get:

$$\varepsilon_2 = -\frac{1}{2}\left\{t \cdot \nabla + \hat{u}\right\}(t, u) = \left(t \cdot \nabla t, t \cdot \nabla u + u^2\right) \tag{39}$$

I can then rewrite Eq. (38) using Eq. (39):

$$\exp\left(-F^k \cdot \nabla - \hat{f}^k\right)\mathcal{M} \approx \exp\left(\tau_3 \cdot \nabla + \hat{\theta}_3\right)$$
$$\Downarrow$$
$$\mathcal{M} = \exp\left(F^k \cdot \nabla + \hat{f}^k\right)\exp\left(\tau_3 \cdot \nabla + \hat{\theta}_3\right) \tag{40}$$

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

$$\left(F^{k+1}, f^{k+1}\right) = \left(F^k, f^k\right) + (\tau_3, \theta_3) + \frac{1}{2}\left\langle\left(F^k, f^k\right), (\tau_3, \theta_3)\right\rangle + \text{higher order} \cdots \tag{41}$$

The Lie bracket in Eq. (41) was defined in Eqs. (26) and (27).

8.2 Validation of all these "DA" operators

Below is a code fragment which allows us to illustrate the results of Secs. (5), (6) and (7):

```
A=exp(vf)                                   implements Eq. (19)
f=c_logf_spin(my_map)                       implements the result of Sec. (8.2)

id=A**(-1)*my_map*A
f_tilde=c_logf_spin(id)    Uses the logarithm to get the results of Eq. (29)


vf_s=A*f                                    Implements Eq. (24)
vf_a=exp(vf,f)                              Implements Eq. (33)

vf_s=vf_s-f_tilde          Compares the logarithm of Eq. (29) with Eq. (24)
call c_full_norm_vector_field(vf_s,norm)
print *, "norm",norm
vf_a=vf_a-f_tilde          Compares the logarithm of Eq. (29) with Eq. (33)
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.