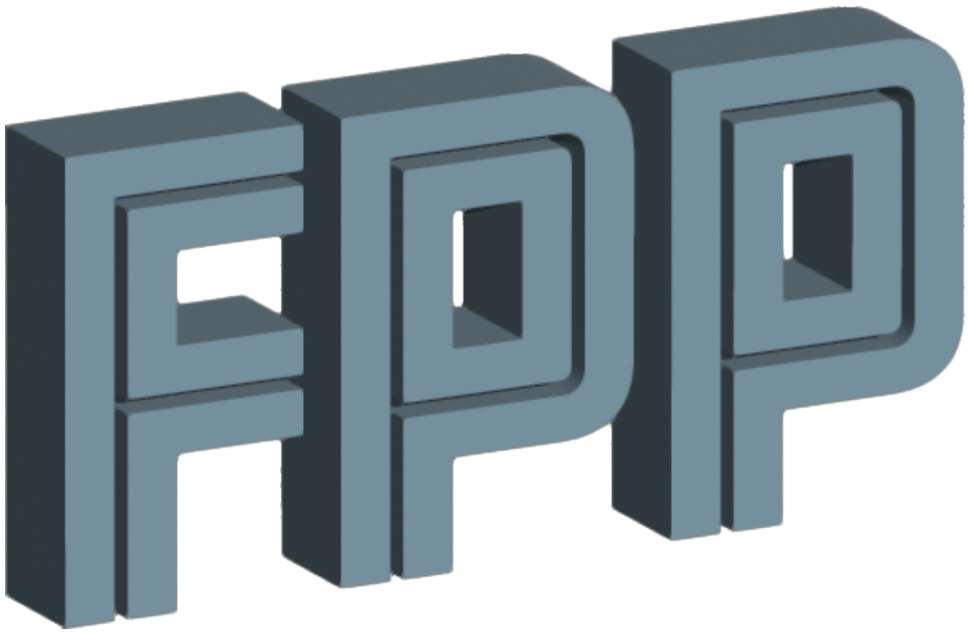
Fully Polymorphic Package Manual



Etienne Forest and David Sagan
March 25, 2023

Contents

1 Introduction to FPP/PTC

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

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

The popularity of FPP/PTC can be attested to by its use with the Bmad[1] toolkit for accelerator and X-ray simulations as well as its use within the MAD[2] simulation program.

The FPP/PTC library has two parts. The Fully Polymorphic Package (FPP) is the part that deals with Taylor series and maps. FPP is pure mathematics detached from any "physics". The Polymorphic Tracking Code (PTC) part 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, 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 within FPP. This package was originally coded by Martin Berz.[**?**]

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

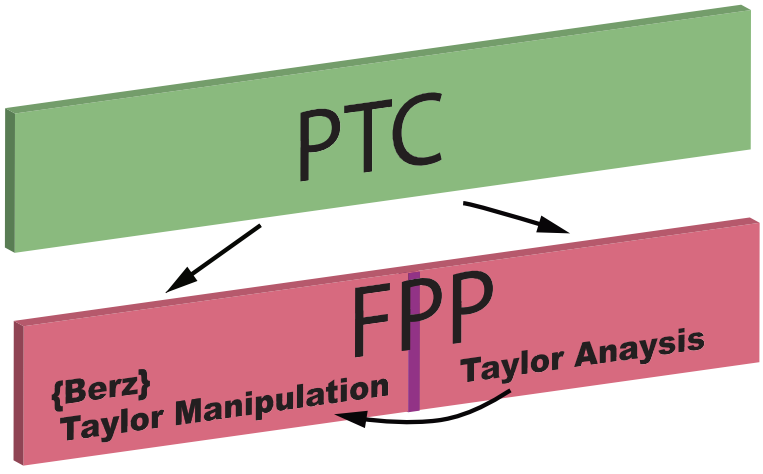


Figure 1: The Fully Polymorphic Package (FPP) part of the FPP/PTC library provides manipulation and analysis of Taylor series and maps and the Polymorphic Tracking Code (PTC) part provides the physics from which accelerators can be analyzed. Arrows indicate code dependencies. The Taylor analysis code uses the Taylor manipulation code but not vice versa. FPP does not use PTC but PTC code uses both FPP's Taylor manipulation and analysis.

2 Where to Obtain FPP/PTC

FPP/PTC can be downloaded from the web via the Bmad web site[1] or at

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

Taylor manipulation routines are contained in the following code files:

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

Taylor analysis routines are contained in the following code files

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

3 Concepts

3.1 Conventions

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

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

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

3.2 TPSA Versus DA

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

- TPSA operations take into account the constant part and the results change as a function of the order (see below).

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

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

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

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

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

$$z = \sum_{j=0}^{N} c_j x^j \tag{3}$$

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

3.3 Polymorphism

In computer programming "**polymorphism**" is the property that a given variable, object, or function can act in different ways depending upon the context. With FPP/PTC, polymorphism is used to define types that can act as if the structure components were real valued numbers or Taylor series. See the documentation of the **real_8** type (§5) as an example.

Note: Many **PTC** tracking code routines come in pairs. One routine of the pair, typically having a "r" suffix in its name, will use real variables while the other routine, typically having a "p" suffix in its name, will use Taylor series variables with the exact same calculation as its real counterpart.

Polymorphic types (§3.3) generally (always???) have structure names that have a **_8** suffix.

3.4 Operator Overloading

FPP/PTC heavily uses operator overloading for Taylor series manipulation. Not only are the standard arithmetical operators $(+, -, *, /, **)$ overloaded as well as the equal sign $(=)$, but there are a number of custom operators that are defined as well. Below is a partial list.[1]

+, -
: Standard addition and subtraction of Taylor series.

*****
: $M1 * M2$, when used with two maps, is TPSA concatenation $M1 \circ M2$ with constant part retained.

.o.

3.5 Tracking Versus Analysis

An important distiction here is the difference between **tracking** and **analysis**. By "tracking" it is meant either the propagation through a lattice of a single particle or a transport map (which here is always an array of truncated Taylor power series).[2] This is opposed to "analysis" which is the study of a transport map to extract such things as resonance driving terms, etc. With **fpp**, analysis is always done on maps and not single particle results.

As a result of this dicotomy between tracking and analysis, FPP/PTC defines different structures that are optimized to handle one or the other. Structures that have been designed tohandle tracking are

```
real_8
complex_8
probe_8
```

and for analysis there is

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

[1] Note: Fortran mandates that custom operator names begin and end with a dot ".".

[2] Mathematically, single particle tracking is just tracking of a transport map using Taylor series truncated at zeroth order. From the code perspective, due to the speed penelely with dealing with Taylor series, the two are distinct as will be seen in this manual.

4 Taylor and ComplexTaylor Fundamental Types

FPP defines two fundamental structures that are the building blocks of many other structures: **Taylor** and **complextaylor**. The **direct** use of either in a tracking program (that is, when doing simulations with PTC) is discouraged. Rather, other types like **real_8** and **complex_8** should be used.

4.1 Taylor Type

The **taylor** structure overloads the Taylor series of the original real "DA-Package" of Berz. See §5.1. The structure is

```
type taylor
   integer i    ! Pointer to Berz
end type taylor
```

The Berz package uses a positive integer to differentiate different Taylor series and the **taylor** structure just stores a reference integer.

4.2 ComplexTaylor Type

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

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

4.3 Overloaded Operators for Taylor and ComplexTaylor Types

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

5 Real_8 Type

FPP defines the **taylor** type (§4.2) which holds a Taylor series. Since computations with Taylor series can be slow, FPP defines "polymorphic" type called **real_8**.[3] In general, a "polymorphic" variable is a variable that can act in different ways depending on the context of the code. In this case, a **real_8** variable can act as if it were a real number or it can act as if it were a Taylor series depending upon how it is initialized. An example program will make this clear.

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

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

!

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

call alloc(r8)          ! Initialize memory for r8

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

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

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

call kill(r8)
end program
```

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

```
type (real_8) r8
```

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

```
call alloc(r8)
```

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.

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

```
call kill(r8)
```

This illustrates a general rule: All calls to **alloc** must have a corresponding **kill**.[4] When **r8** is set to the real number **x** in the line[5]

```
  r8 = x
```

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

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

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

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

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

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

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

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

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

The first argument, **only_2d0**, is a parameter defined by **PTC** of type **internal_state** (§8).[6] When **only_2d0** is used as the first argument to **init**, the number of variables will be two which is appropriate for simulations involving motion along one axis (typically involving phase space (x, p_x)).[7] 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:

$$t = \sum_{i,j}^{0 \le i+j \le 3} C_{ij}\, z_1^i\, z_2^j \tag{4}$$

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

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

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

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

```
call init (3, 2) ! Init just FPP. Order = 3, #Vars = 2
```

Notice that here the order comes before the number of variables which is the reverse of the order when **init** is called with an **internal_state** as the first argument.

where z_1 is the first variable and z_2 is the second variable. **FPP** sets up a **real_8** array named **dz_8** such that **dz_8(N)** represents the N^{th} variable.[8] Thus in the above code **r8** is initialized to the Taylor series:

$$t = 0.7 + z_1 + 2\,z_2^3 \tag{5}$$

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

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

The output is:

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

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

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

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

```
          3    2.000000000000000        0   3
```

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

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

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

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

```
This is r8^4:
Out   Order   Coef                        Exponents
-------------------------------------------------------------
          0   0.2400999999999999        0   0
          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 §**??**.

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

5.1 Real_8 Under the Hood

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

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

The **t** component of the structure is of type **taylor** (§4.2) and is used if a **real_8** variable is acting as a Taylor series. The **r** component is used if a **real_8** variable is acting as a real number. The **kind** component is an integer that sets the behavior of a **real_8** variable. Besides behaving as **real** or a Taylor series, a **real_8** variable may behave as a "**knob**" which will be explained later. The reason for hiding a Taylor series under the hood is to defer the decision of its use to run time.

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

6 Complex_8 Type

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

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

The definition of **complextaylor** is:

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

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

7 Real_8 and Complex_8 Functions and Operators

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

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

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

Include dz_8

8 Internal_State Type

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

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

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

Explain Bmad units

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

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

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

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

$$x(1:6) = \left(x, \frac{p_x}{p_0}, y, \frac{p_x}{p_0}, -\beta c T \text{ or } -\beta\, c\, \Delta T, \frac{\Delta p}{p_0} \right) \tag{7}$$

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

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

where

$$\Delta T = T - T_{ref} \tag{8}$$

9 Other Polymorphic Types: spinor_8, quaternion_8, and rf_phasor_8

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

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

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

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

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

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

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

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

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

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

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

These structures are used as components of the **probe** and **probe_8** types (§10) that are used for tracking.

10 Probe and Probe_8 Types

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

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

And **probe_8** is the polymorphic equivalent (§3.3):

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

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

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

The same routine duplication happens with routines that use **real_8**. In this case the corresponding routine will use a **real(dp)** variable. As an example, consider the subroutine in the example of §11:

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

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

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

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

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

10.1 The x(6) component

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

10.2 The spin and quaternion components

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

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

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

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

$$\begin{aligned} s(1) &= (1,0,0) \\ s(2) &= (0,1,0) \\ s(3) &= (0,0,1) \end{aligned} \tag{9}$$

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 §**??**.

10.3 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.

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

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

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

$$\Sigma_{\text{final}} = M\,(\Sigma_{\text{initial}} + E)\,M^{\text{T}} \tag{10}$$

where superscript **T** indicates transpose and

$$\Sigma_{ij} = \left\langle x_i x_j \right\rangle, \qquad i,j = 1,\ldots 6. \tag{11}$$

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

$$\Sigma_{\text{final}} = M\left(\Sigma_{\text{initial}} + E_{\text{probe\_8}}\right) M^{\text{T}} = M\,\Sigma_{\text{initial}} M^{\text{T}} + E_{\text{c\_damap}}. \tag{12}$$

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

$$\Sigma_{\text{final}} = \ M\,\Sigma_{\text{initial}} M^{\text{T}} + E, \tag{13}$$

where E has been redefined.

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

$$M = B\Lambda B^{-1} \quad \text{where}$$
$$\Lambda = \begin{pmatrix} \lambda_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \lambda_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & \lambda_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & \lambda_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & \lambda_6 \end{pmatrix} \quad \text{and} \quad \begin{array}{l} \lambda_1 = \exp\left(-\alpha_1 - i2\pi\mu_1\right) \\ \lambda_2 = \exp\left(-\alpha_1 + i2\pi\mu_1\right) \\ \lambda_3 = \exp\left(-\alpha_2 - i2\pi\mu_2\right) \\ \lambda_4 = \exp\left(-\alpha_2 + i2\pi\mu_2\right) \\ \lambda_5 = \exp\left(-\alpha_3 - i2\pi\mu_3\right) \\ \lambda_6 = \exp\left(-\alpha_3 + i2\pi\mu_3\right) \end{array} . \tag{14}$$

We can apply the transformation B on Eq. (13):

$$\begin{aligned} B\Sigma_{\text{final}}B^{\text{T}} &= BM\Sigma_{\text{initial}}M^{\text{T}}B^{\text{T}} + BEB^{\text{T}} \\ \sigma_{\text{final}} &= \Lambda\sigma\Lambda + \varepsilon \end{aligned} \tag{15}$$

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

$$\begin{aligned} \sigma^{\infty} &= \Lambda\sigma^{\infty}\Lambda + \varepsilon \\ &\Downarrow \\ \sigma^{\infty}_{ij} &= \frac{\varepsilon_{ij}}{1 - \lambda_i\lambda_j} \end{aligned} \tag{16}$$

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

$$\sigma^{\infty}_{12} = \frac{\varepsilon_{12}}{1 - e^{-2\alpha_1}} \approx \frac{\varepsilon_{12}}{2\alpha_1}. \tag{17}$$

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

$$\Sigma^{\infty} = B^{-1}\sigma^{\infty}B^{-1\,\text{T}}. \tag{18}$$

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

10.5 Real(dp) type probe specific to PTC

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

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

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

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

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

interface track
  module procedure trackr
  module procedure trackp
end interface

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

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

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

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

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

If we call **track(p)** where **p** is a **probe_8**, then the resulting **p** could be a Taylor series which approximates the true map[9] of the code.

For example, in §11, 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:

$$\mathbf{z} = \begin{pmatrix} 1 & 1 \\ -0.1 & 0.9 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} + O\left(z^2 \right) \tag{19}$$

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

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

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

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

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

If the object in FPP is extremely important, the solution in FPP is to define a new type and keep the $+, *, \ldots$ signs for this new type. Therefore we do not allow the concatenation of **probe_8** even when it is reasonable. Instead we construct a map, type **c_damap** described in §**??**, 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[10] 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 §**??**.

[10] 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.

11 Knobs

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

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

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

call alloc(z)
call alloc( L , B, K_q , K_s )
np=0
print * , "Give  L and parameter ordinality (0 if not a parameter)"
read(5,*) L%r , ip
np=np+ip
call make_it_knob(L,ip);  np=np+ip;
print * , "Give  B  and parameter ordinality (0 if not a parameter)"
read(5,*) K_q%r , ip
print * , "Give  K_q and parameter ordinality (0 if not a parameter)"
read(5,*) K_q%r , ip
call make_it_knob(K_q,ip);  np=np+ip;
print * , "Give  K_s and parameter ordinality (0 if not a parameter)"
read(5,*) K_s%r , ip
call make_it_knob(K_s,ip); np=np+ip;
print * , "The order of the Taylor series ?"
read(5,*) no

call init(no,nd,np) ! Initializes TPSA

z(1)=z0(1) + dz_8(1) ! <--- Taylor monomial z_1 added
z(2)=z0(2) + dz_8(2) ! <--- Taylor monomial z_2 added
```

```
call track(z)

call print(z)

contains

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

end program my_small_code_real_8
```

In this little code, there is one drift of length **L** followed by a multipole kick that contains a dipole of strength **B**, a quadrupole of strength **K_q** and a sextupole of strength **K_s**. We run the code ignoring the parameters:

```
 Give  L and parameter ordinality (0 if not a parameter)
1 0
 Give  B  and parameter ordinality (0 if not a parameter)
0 0
 Give  K_q and parameter ordinality (0 if not a parameter)
.1 0
 Give  K_s and parameter ordinality (0 if not a parameter)
0 0
 The order of the Taylor series ?
2

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

   1   1.000000000000000       1  0
   1   1.000000000000000       0  1


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

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

This little program produces Taylor series to second order in the phase space variables $\mathbf{z} = (z_1, z_2)$ similar to the programs **Transport** and **Marylie**. However, we can now require that the multipole strengths be variables of the Taylor series without recompiling the program. In this example, we make the quadrupole strength the third variable of TPSA: $K_q = 0.1 + dz_3$.

```
 Give  L and parameter ordinality (0 if not a parameter)
1 0
```

```
 Give  B  and parameter ordinality (0 if not a parameter)
0 0
 Give  K_q and parameter ordinality (0 if not a parameter)
.1 1
 Give  K_s and parameter ordinality (0 if not a parameter)
0 0
 The order of the Taylor series ?
2

 Properties, NO =    2, NV =    3, INA =   22
 *******************************************

   1   1.000000000000000       1  0  0
   1   1.000000000000000       0  1  0


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

   1 -0.1000000000000000       1  0  0
   1  0.9000000000000000       0  1  0
   2  -1.000000000000000       1  0  1
   2  -1.000000000000000       0  1  1
```

Again, we must emphasize that while it would have been easy here to use the type **taylor** for all the variables, it is totally unfeasible in a real tracking code to either recompile the code or allow all parameters of the systems to be Taylor series. This is why typical matrix[11] 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 §5.1:

1. A real number
2. a Taylor series with real coefficients (**taylor**)
3. a knob which is a simple temporary Taylor series activated only if needed

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

12 Manual To Do List

Note: calling init again wipes out existing Taylor series. (Does is wipe out universial taylor?)
Explain knobs

13 References

[1] D. Sagan, "Bmad: A relativistic charged particle simulation library," *Nuc. Instrum. and Methods in Phys Research A*, vol. 558, pp. 356–359, 2006. http://www.lepp.cornell.edu/~dcs/bmad.

[2] F. C. Iselin, "The MAD program (Methodical Accelerator Design)," *CERN/SL/92*, 1992.