Manual for FPP: The Fully Polymorphic Package

Étienne Forest
Tsukuba, Japon

August 30, 2018

Contents

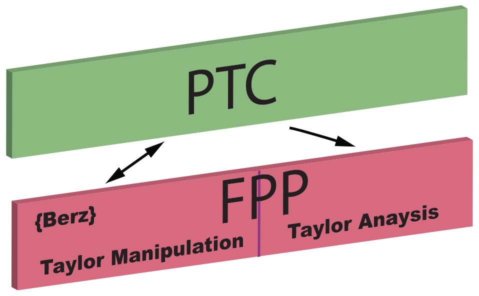

Figure 1: The Fully Polymorphic Package (FPP) part of the FPP/PTC library provides manipulation and analysis of Taylor series and maps and the Polymorphic Tracking Code (PTC) part provides the physics from which accelerators can be analyzed.

1 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 within FPP. This package was originally coded by Martin Berz.

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

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 polymorphic types used in any code

In this section we list all the simple types that exist in FPP in order of increasing complexity. They are needed if one write a simple tracking code.

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. Its direct use is discouraged. Tracking programs should use the real polymorph.

1.5.2 Type `complextaylor`

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

This complex Taylor 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`

This type is the most important type if you write a tracking code of respectable length. Imagine that your code tracks in one degrees of freedom (1-d-f). Then you will push two 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 our 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 include some parameters, such as quadrupole strengths in the calculation? Since this is a user decision, it is best if the code decides at execution time using the type `real_8`. This was explained already in Sec. (1.4) but here we provide a small complete example.

```
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
```

The situation presented here is very similar to the matrix codes “Transport” or “Marylie”. This little program produces Taylor series to second order in the phase space variables $\mathbf{z} = (z_1, z_2)$.

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 tun with 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[2] codes, not using TPSA, are limited to a small set of Taylor variables, usually the six phase space variables.

So in summary a polymorph can be as mentioned in Sec. (1.4):

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

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

1.5.4 Type `complex_8`

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 equation is written in cylindrical coordinate. In such a case, if $\mathbf{z} = (x, p_x, y, p_y)$, then most intermediate calculation involve a quantity $q = x + i\,y$. Therefore it is extremely useful inside one's code to declare `q` as a complex polymorph. This happens in some magnets of PTC but it is generally hidden from a normal user.

```
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 of Sec. (1.5) are useful to anyone who decides to use FPP to write his own tracking code. In fact there is nothing "tracking" about these types. Once 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 a real tracking code must deal with.

The code PTC tracks the following Fortran structure:

```
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
```

As the reader can see, it is made out of other structures which are themselves made of `real_8` or simple real numbers. I will describe each component.

1.6.1 The x(6) component of `probe_8`

This represents the orbital part of the ray. Thus we have the following associations:

- In PTC units, when time is used:

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

- in BMAD units

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

where

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

It does not take too much imagination to see that 1-d-f, 2-d-f and 3-d-f Taylor maps will be created easily with the appropriate initialization of the polymorphic `probe_8`.

1.6.2 The spin and quaternion components of `probe_8`

PTC, as BMAD, 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 direction 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 `spinor_8` called `s(3)`. A `spinor_8` is just

```
  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 following initial conditions

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

the end vectors will allow us to construct the one-turn 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 map for the spin: this will be discussed later.

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 matrix, 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.

1.6.3 The components of type `rf_phasor_8_8`

These objects are a little more complex to explain but their Fortran 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 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 use that pseudo-clock to modulate their multipole components. In the end, as we will see, the components `rf_phasor_8%x(2)` is 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 this except that allow 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 finally 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 uses the polymorphs `real_8` and nothing else. However this is not what PTC does. PTC has a `real(dp)` version as well as a `real_8` embedded within itself. This is an issue of speed. Consider the following Fortran statement

```
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 of 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. Therefore 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). This most operations are acting directly and quickly on Fortran intrinsic types.

1.7 The type c_taylor and the Taylor maps c_damap for analysis

PTC is an integrator just like our little example of Sec. (1.5.3). This little code calls the tracking subroutine

```
      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 seems to compute the effect of a “drift” followed by a multipole “kick” in the jargon of accelerator physicists. And indeed, if one substitutes it with

```
      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.

It so happens that PTC tracks `probe` or `probe_8`. It is very easy to modify the above routines to reflect this fact. We insert them in a module which uses the FPP module `tree_element_module` where `probe` or `probe_8` are defined:

```
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[3] 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` containing the same information 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,

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

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[4] observed.

Conversely we define a new operator when the creation of a new type would too cumbersome due to its infrequent usage.

This will now be explained in Sec. (2) where we describe the important types and associated operators.

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 `taylors` 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:

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

In fact, the output from the code, a `c_damap`, is made of two `c_taylor`.

[4]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.

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.[5]

1.7.2 The type `c_damap`

The type `c_damap` is defined as

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

~~There is an obvious resemblanc~~e with `probe_8` which we recall is:

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

```
    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 an new operator must be defined.

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[6] 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 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

[6]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.

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` in Sec. (1.7.2).

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 associated operators

2.1 Obsolete Types

3 overloading

4 subpackage