

# Introduction to the Polymorphic Tracking Code

## Fibre Bundles, Polymorphic Taylor Types and “Exact Tracking”

July 4, 2002

Étienne Forest

National High Energy Research Organization (KEK)

1-1 Oho, Tsukuba, Ibaraki, 305-0801, Japan

and

Frank Schmidt

SL-AP Group, CERN, Geneva, CH

Eric McIntosh

IT-API Group, CERN, Geneva, CH

### Abstract

This is a description of the basic ideas behind the “Polymorphic Tracking Code” or PTC. PTC is truly a “kick code” or symplectic integrator in the tradition of TRACYII, SixTrack, and TEAPOT. However it separates correctly the mathematical atlas of charts and the magnets at a structural level by implementing a “restricted fibre bundle.” The resulting structures allow backward propagation and recirculation, something not possible in standard tracking codes.

Also PTC is polymorphic in handling real (single, double and even quadruple precision) and Taylor series. Therefore it has all the tools associated to the TPSA packages: Lie methods, Normal Forms, Cosy-Infinity capabilities, beam envelopes for radiation, etc., as well as parameter dependence on-the-fly. However PTC is an integrator, and as such, one must, generally, adhere to the Talman “exactness” view of modelling. Incidentally, it supports exact sector and rectangular bends as well. Of course, one can certainly bypass its integrator and the user is free to violate Talman’s principles on his own; PTC provides the tools to dig one’s grave but not the encouragement.

The reader will find in Appendix B a PowerPoint presentation of FPP. The presentation is a bit out of date but it gives a good idea of FPP which is essential to PTC. FPP is a stand-alone library and can be used by anyone with a FORTRAN90 compiler.

This presentation is also, to be honest, a place where the authors intend to document very incompletely nearly two years of work: the development of FPP and subsequently that of PTC.

Our ultimate intention is to morph PTC completely into MAD-X. The code MAD-X is an upgrade of MAD-8 and not of the C++ CLASSIC based code MAD-9. The present document does not address when and how this will be done. It is also our goal to link, if possible, PTC with CAD programs for the design of complex follow-the-terrain beam lines. So far FPP and PTC have been used in the design of beam separators (complex polymorphs) and recirculators. They have also been linked with the code BMAD from Cornell. There is still a lot of work to be done if these tools are to be generally usable by a wide range of people.

In addition, more complex structures will be needed to handle effects beyond single particle dynamics in a way which respects the fundamental mathematical integrity of the structures of PTC.

**N.B.-This document will be slowly corrected and upgraded on our website as FPP, PTC, and MAD-X evolve.**

# Contents

<b>1</b>	<b>OVERVIEW WITH EXAMPLES</b>	<b>8</b>
<b>A</b>	<b>A Survey of PTC</b>	<b>8</b>
A.1	Introduction . . . . .	8
A.2	A Short Description Using Los Alamos PSR . . . . .	8
A.2.1	The Layout and an Example: PSR Storage Ring . . . . .	9
A.2.2	The Importance of the Charts: Misalignments and the Euclidean Group . . . . .	13
A.2.3	Tracking a Single Fibre . . . . .	17
A.3	Comparing and Contrasting C++ CLASSIC Classes and Multiple Inheritance with Composition (Forward Delegation) . . . . .	17
A.3.1	Fibre Bundles (or not) . . . . .	17
A.3.2	Composition, Inheritance and Algorithmic Classes . . . . .	19
A.3.3	More on delegation and PTC . . . . .	22
A.4	ELEMENTP and Polymorphism . . . . .	23
<b>B</b>	<b>Non Trivial Examples</b>	<b>30</b>
B.1	Example 1: Two Siamese Rings . . . . .	30
B.2	Example 2: Making a Figure “8” . . . . .	33
<b>2</b>	<b>FILE BY FILE DESCRIPTION</b>	<b>35</b>
<b>C</b>	<b>A few aspects of FPP and PTC</b>	<b>35</b>
C.1	ELEMENT and ELEMENTP: EL and ELP . . . . .	35
C.2	Real*8 array X(6) . . . . .	36
C.3	Real Polymorph Array Y(6) . . . . .	36
C.4	Beam Envelope Array YS(6) . . . . .	36
C.4.1	Definition of Beam Envelope . . . . .	37
C.4.2	Normalizing the Beam Envelope in PTC . . . . .	37
C.5	The Type DAMAP: M . . . . .	38
C.6	The Type NORMALFORM . . . . .	38
C.7	The Type UNIVERSAL_TAYLOR . . . . .	38
C.8	The Module Precision_Constants . . . . .	39
C.9	The Module File_Handler . . . . .	39
<b>D</b>	<b>Sa_ROTATION_MIS.f90: The Module Rotation_mis</b>	<b>40</b>
D.1	Basic Description of the Module . . . . .	40
D.2	Operations on Type Matrix_PTC . . . . .	41
<b>E</b>	<b>Sb_EXTEND_POLY.f90</b>	<b>43</b>
E.1	The Explosive Functions . . . . .	43
E.2	The REAL_8 Type . . . . .	43
E.3	The (=) Assignment . . . . .	43
E.3.1	REAL_8REAL6: Y=X . . . . .	43
E.3.2	REAL6REAL_8: X=Y . . . . .	43
E.3.3	REAL_8REAL_8: Y2=Y1 . . . . .	44
E.3.4	ENV_8MAP: YS=M . . . . .	44
E.3.5	REAL6ENV_8: X=YS . . . . .	44
E.3.6	Initializing an Envelope with ENV_8T or ENV_8BENV: YS=T or YS=ENV . . . . .	44
E.3.7	Extracting the Tracked Envelope with TENV_8: T=YS . . . . .	44
E.4	The Operator + . . . . .	44
E.5	The PRINT and DAPRINT Interface . . . . .	44
E.5.1	Printing and Reading Arrays . . . . .	44
E.5.2	Printing a Beam Envelope YS . . . . .	45
E.6	The ALLOC and KILL Interface . . . . .	45
E.6.1	Allocating Arrays . . . . .	45
E.7	The Context Routine . . . . .	45

<b>F</b>	<b>Sc_i_POL_TEMPLATE.f90 and Sg_i_template_MY_KIND.f90</b>	<b>46</b>
F.1	Dealing with POL_BLOCK . . . . .	46
F.2	Dealing with WORK . . . . .	48
<b>G</b>	<b>Sd_EUCLIDEAN.f90</b>	<b>49</b>
G.1	Coping with the Square Root . . . . .	50
G.2	Translating an Element: TRANS(A,X,B,EXACT,CTIME) . . . . .	50
G.3	Rotating an Element: ROT_YZ, ROT_XZ, and ROT_XY . . . . .	50
G.3.1	The Rotation ROT_XY in the Transverse Plane . . . . .	51
G.3.2	The Rotations ROT_XZ and ROT_YZ . . . . .	51
G.4	A Matter of Perspective: Are we Patching Here? . . . . .	52
G.4.1	Defining Patching . . . . .	52
G.4.2	An Example: First, Using the Compression Trick . . . . .	53
G.4.3	Same Example: Now, Using Patching . . . . .	53
G.4.4	Failure of Compression: Decompression! . . . . .	54
G.5	Routines of Sd_EUCLIDEAN . . . . .	55
<b>H</b>	<b>Se_FRAME.f90</b>	<b>56</b>
H.1	The Charts and the Patches . . . . .	56
H.2	Subroutines of S_FRAME . . . . .	57
<b>I</b>	<b>Sf_STATUS.f90</b>	<b>59</b>
I.1	Constants of Sf_STATUS.f90 . . . . .	59
I.2	TYPE WORK . . . . .	61
I.3	TYPE POL_BLOCK . . . . .	61
I.4	TYPE MAGNET_CHART . . . . .	61
I.5	TYPE INTERNAL_STATE . . . . .	61
I.6	Defined States and Operations on States . . . . .	61
I.6.1	The Basic States . . . . .	62
I.6.2	The Eternal Basic States . . . . .	62
I.6.3	MAKE_STATES: Initializes PTC and Solves Maxwell's Equations . . . . .	62
I.6.4	Addition of States: S1+S2 . . . . .	63
I.6.5	Subtraction of States: S1-S2 . . . . .	63
I.6.6	Unary Plus: +S1 . . . . .	63
I.6.7	STATE%EXACTMIS versus ELEMENT%EXACTMIS . . . . .	63
I.6.8	STATE%FRINGE versus ELEMENT%PERMFRINGE . . . . .	63
I.6.9	Printing a State: PRINT_S . . . . .	63
I.6.10	UPDATE_STATES . . . . .	64
I.6.11	CLEAR_STATES . . . . .	64
I.7	Initializing FPP within PTC: INIT . . . . .	64
I.7.1	INIT in FPP . . . . .	64
I.7.2	INIT in PTC . . . . .	65
I.8	User Defined Integrators or Strange <i>s</i> -Dependence . . . . .	66
<b>J</b>	<b>Sg_0_FITTED.f90</b>	<b>68</b>
<b>K</b>	<b>Sh_DEF_KIND.f90</b>	<b>69</b>
K.1	List of Magnet Types . . . . .	69
K.2	Inheritance of Element Properties: Delegation in FORTRAN90 . . . . .	70
K.3	Some Maintenance Routines: Zeroing, ALLOC and KILL . . . . .	71
K.3.1	The (=) Assignment . . . . .	71
K.3.2	More Interfaces for ALLOC and KILL . . . . .	72
K.4	More about the Magnets . . . . .	72
K.4.1	DRIFT1: Drift . . . . .	72
K.4.2	DKD2: Drift-Kick-Drift Element . . . . .	72
K.4.3	KICKT3: Thin Multipole Kick . . . . .	73
K.4.4	CAV4: RF Cavity . . . . .	73
K.4.5	SOL5: The Combined Function Solenoid . . . . .	74

K.4.6	KTK: Delta-dependent Quadratic Hamiltonian and Multipole Kicks	74
K.4.7	TKTF : Quadratic Hamiltonian, Delta-Corrections, and Multipole Kicks	75
K.4.8	NSMI and SSMI: Single Multipole Thin Kicks	76
K.4.9	TEAPOT: The Exact Sector Bend	76
K.4.10	MON : Monitors	77
K.4.11	ESEPTUM: Electric Septum	78
K.4.12	STREX: The Exact Generic Rectangular Bend	78
K.4.13	SOLT: Delta-dependent Quadratic Hamiltonian with a Solenoidal Term and Multipole Kicks	79
<b>L</b>	<b>Si_DEF_ELEMENT.f90</b>	<b>80</b>
L.1	Constants and Internal Routines of Si_DEF_ELEMENT.f90	80
L.1.1	ZERO_ANBN	80
L.1.2	ALWAYS_EXACTMIS and ALWAYS_FRINGE	80
L.1.3	The Logical FEED_P0C	80
L.1.4	BERZ and ETIENNE	80
L.1.5	MOD_N(I,J)	80
L.1.6	RESET31(ELP), TPSAFIT(LNV), and SET_TPSAFIT	80
L.1.7	VERBOSE and GEN	80
L.2	Types whose functionality is defined in Si_DEF_ELEMENT.f90	81
L.3	Copying ELEMENT and ELEMENTP: COPY and EQUAL	81
L.4	The Assignment (=)	82
L.4.1	EL(P)=INTEGER : ZERO_EL and ZERO_ELP	82
L.4.2	EL(P)=STATE : MAGSTATE and MAGPSTATE	82
L.4.3	The Type(WORK): Design Energy	82
L.4.4	The Type(MUL_BLOCK): Changing the AN and BN	83
L.4.5	Example of Non-Trivial Use of Types MUL_BLOCK and WORK	84
L.4.6	Setting the Knobs Using a POL_BLOCK: Routines BLPOL_0 and ELP_POL	86
L.4.7	EL=X(6) : Subroutine MIS_, MIS_P, and FIBRE_MIS for Misalignments	89
L.5	The SETFAMILY Interface: Pointing from ELEMENT to Magnet Types	90
L.6	Adding Multipole Components: ADD	91
<b>M</b>	<b>Sj_ELEMENTS.f90</b>	<b>92</b>
<b>N</b>	<b>Sk_LINK_LIST.f90</b>	<b>93</b>
N.1	The fundamental types FIBRE and LAYOUT	93
N.2	The various routines of S_FIBRE_BUNDLE	97
<b>O</b>	<b>SI_FAMILY.f90</b>	<b>99</b>
O.1	More on POL_BLOCK	99
O.1.1	Assigning Polymorphs to a Layout with SETPOL_L: LAYOUT=POL_BLOCK	99
O.1.2	Why is the Polymorph ELP%L not in POL_BLOCK?	101
O.1.3	Removing Parameters: KILL_PARA	101
O.2	Routines extended from EL(P) to FIBRE	101
O.2.1	The Interface ADD: ADDP_ANBN	101
O.2.2	FIBRE_POL: FIBRE=POL_BLOCK	101
O.2.3	FIBRE_BL: FIBRE=MUL_BLOCK	101
O.2.4	FIBRE_WORK: FIBRE=WORK	101
O.2.5	MISALIGN_FIBRE: FIBRE=X(6)	102
O.3	Copying all ELPs into ELs and Vice Versa	102
O.4	Copying Layouts: COPY and EQUAL	102
O.5	Standard Surveys	102
O.5.1	Full Standard Survey	102
O.5.2	Partial Standard Survey	102
O.6	Moving a Layout	103
O.6.1	Rotating a Layout	103
O.6.2	Moving a Layout	103
O.7	Routines of S_FAMILY	103

<b>P</b>	<b>Sm_TRACKING.f90</b>	<b>105</b>
P.1	TRACK for a Layout . . . . .	105
P.2	TRACK for a Fibre . . . . .	105
P.3	The MIS_FIB Routines: Misaligning a Magnet . . . . .	107
P.4	The Variables ALWAYS_EXACT_PATCHING . . . . .	108
P.5	The Routines of S_TRACKING . . . . .	108
<b>Q</b>	<b>Sn_MAD_LIKE.f90</b>	<b>109</b>
Q.1	Example of the PSR Revisited . . . . .	109
Q.2	Operations on types LAYOUT and FIBRE . . . . .	110
Q.2.1	EL+EL . . . . .	110
Q.2.2	EL+BL and BL+EL (BL stands for a layout) . . . . .	110
Q.2.3	BL+BL . . . . .	111
Q.2.4	N*EL and N*BL . . . . .	111
Q.2.5	-BL . . . . .	111
Q.3	Am I a Dumb or Smart User? . . . . .	111
Q.4	The Rules . . . . .	112
Q.4.1	Making the States and the Logicals: MAD and MADLENGTH . . . . .	112
Q.4.2	The Subroutine SET_MAD and MADKIND2 (MADTHICK) . . . . .	113
Q.4.3	Logicals and Integer Flags . . . . .	113
Q.4.4	More on MADKINDs . . . . .	114
Q.4.5	Cleaning Up . . . . .	114
Q.5	The Elements . . . . .	115
Q.5.1	The Marker . . . . .	115
Q.5.2	The Drift . . . . .	115
Q.5.3	The Monitors and the Instruments . . . . .	115
Q.5.4	The Quadrupole and Tilting . . . . .	115
Q.5.5	The Solenoid . . . . .	116
Q.5.6	Other Straight Elements . . . . .	116
Q.5.7	More Straight Elements: HKICKER,VKICKER, and KICKER . . . . .	116
Q.5.8	The Rectangular Bend . . . . .	116
Q.5.9	The Sector Bend . . . . .	117
Q.5.10	The General Bend . . . . .	118
Q.5.11	The RF Cavity . . . . .	118
Q.5.12	The Single Lens or SixTrack's SMI . . . . .	118
Q.5.13	The Thin Multipole Block . . . . .	118
Q.6	Operators Acting On MAD-Like Input . . . . .	118
Q.6.1	Adding Multipole Components . . . . .	119
Q.7	The Final Step: the Creation of a Layout . . . . .	119
<b>R</b>	<b>So_FITTING.f90: Non-core Routines</b>	<b>120</b>
R.1	Changing the Integration Method of a Magnet . . . . .	120
R.2	Fixed Point Routines: Polymorphi Delendi Sunt . . . . .	120
R.2.1	FIND_ORBIT_LAYOUT(RING,X,LOC,STATE) . . . . .	121
R.2.2	FIND_ORBIT_LAYOUT_NODA(RING,X,LOC,STATE,EPS) . . . . .	121
R.2.3	FIND_ORBIT_M_LAYOUT(RING,Y,LOC,STATE) . . . . .	121
R.2.4	FIND_ENV_LAYOUT(RING,YS,X,LOC,STATE) . . . . .	122
R.2.5	Parameter Dependence: FIND_ENVELOPE(RING,YS,A,FIX,LOC,STATE) . . . . .	122
<b>A</b>	<b>Postface by Etienne Forest</b>	<b>125</b>
<b>B</b>	<b>APPENDIX: THE POWERPOINT PRESENTATION OF FPP</b>	<b>129</b>

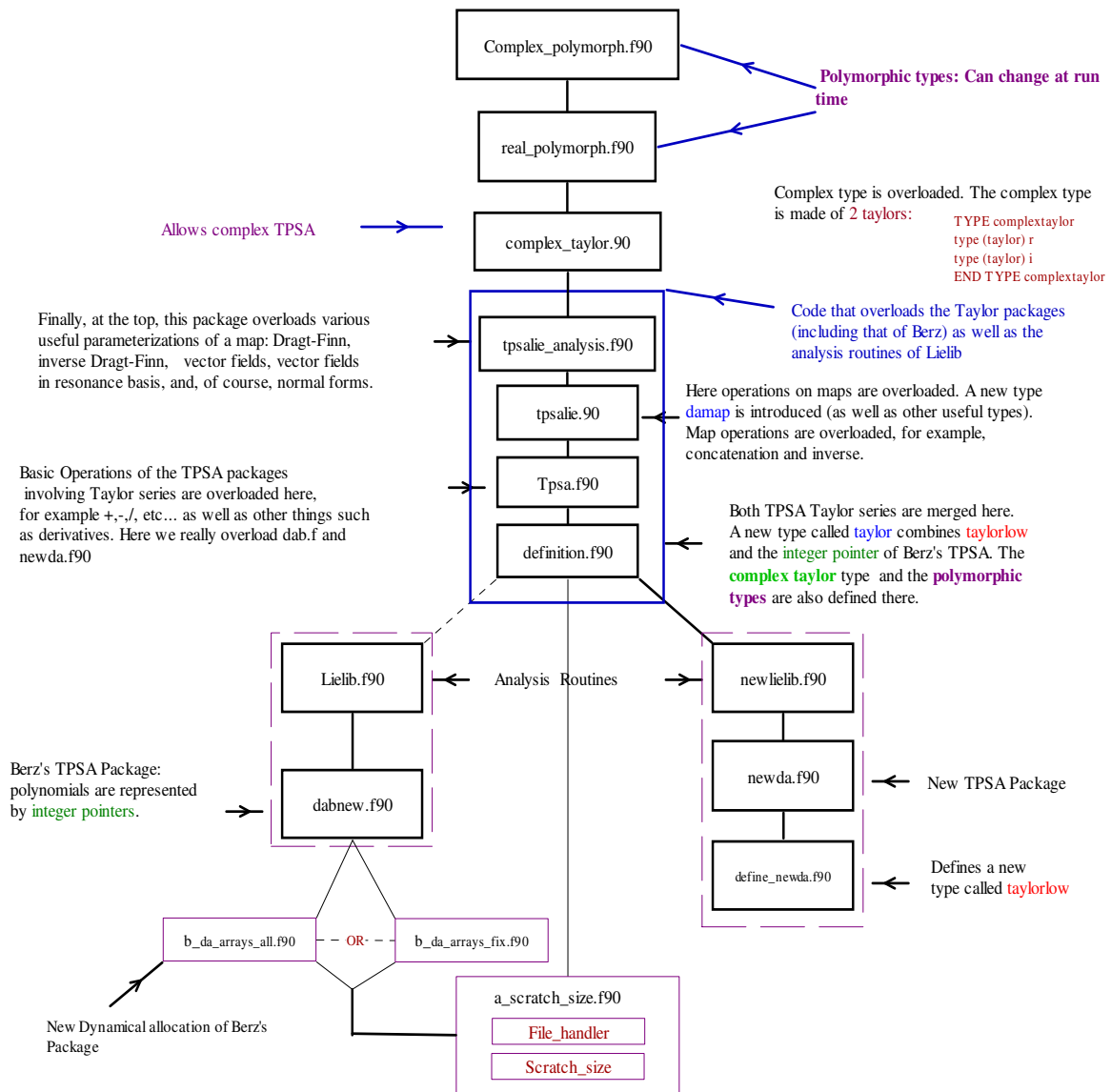


Figure 1: Structure of FPP

The above picture is from the PowerPoint presentation on the “Fully Polymorphic Package ” located at the URL

[http://bc1.lbl.gov/CBP\\_pages/educational/TPSA\\_DA/Introduction.html](http://bc1.lbl.gov/CBP_pages/educational/TPSA_DA/Introduction.html)

## PTC STRUCTURE

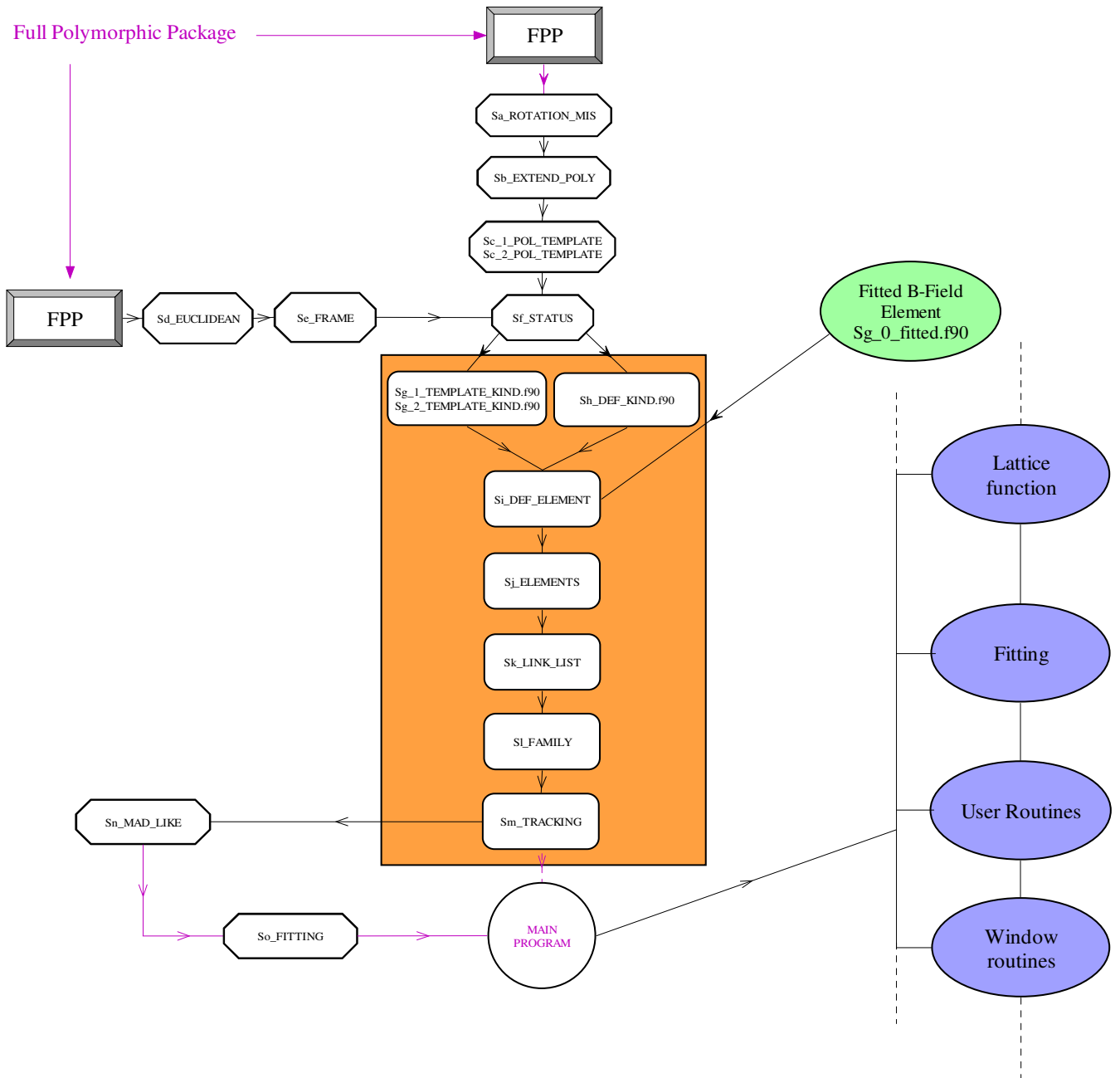


Figure 2: Structure of PTC

# 1 OVERVIEW WITH EXAMPLES

## A A Survey of PTC

### A.1 Introduction

The program PTC was started as an educational exercise at DESY in collaboration with Aimin Xiao under the name `Small_Code`. Our goal was to implement two things that are missing in traditional codes. The first thing and perhaps the most fundamental was the implementation of a `Layout` class which allows a clear separation between the ideal bend angle and the actual field which attempts to produce this bend. This `Layout` type is now a doubly linked list whose nodes, of type `FIBRE`, contains the magnet per se, the local charts and the patches. The reader will realize that the `FIBRE` is the central entity which makes up a beam line inside a tracking code. Traditionally a beam line is thought of as being made up of magnet propagators. Unfortunately this simple point of view does not permit the creation of a fibre bundle which is the correct mathematical structure needed to fully support single particle dynamics in a tracking code in the “s”-dependent (or magnet dependent) formalism rather than the more general “t”-dependent Newton-Einstein parameterization. This aspect is why we do not yet discuss the details of the implementation of PTC into MAD-X. At present all versions of MAD deal with the traditional collection of magnet propagators and thus cannot handle an arbitrary PTC lattice. It should be pointed out that it is relatively easy, as we shall see, to create monstrosities within PTC, but it is not trivial to imagine doing such things within a MAD-like input file with no assumed access to the programming language. There are some pointers within the fibre which point (sorry about the bad pun) to the issue of copying, manipulating and storing PTC’s more bizarre lattices. It is interesting, as we shall also see, that it is possible within the context of a programming language to generate lattices which cannot be copied, unlike the standard beam lines, unless one adds additional pointers, “unnecessary” from the point of view of pure tracking (see Sect. N.1). All of this and the actual implementation in MAD-X, when and if finally done, will be fully documented in a separate document. In the present article we will show simple fictitious beam lines which display the power of the PTC structures.

The second thing is the implementation of polymorphism for TPSA calculations (Truncated Power Series Algebra) where parameter dependence is done on-the-fly. This has been possible for years in the context of COSY-Infinity style interpreters. It is also possible, with messy programming, to put a subset of such capabilities in a compiled FORTRAN77 tracking code (as was done in `DESPOT` and `SixTrack`). However modern languages such as C++ and FORTRAN90 facilitate greatly this task provided someone goes to the trouble of creating a suitable library: this is `FPP`.

In this note, we will describe the core of PTC as it exists. Certainly the basic model overlaps with codes such as `TRACYII`, `LEGO`, `MAPA`, and `TEAPOT` just to name a few. `LEGO`<sup>1</sup>, in particular, provides some of the patching functionality found in PTC. But the complete implementation of a fibre bundle is perhaps unique to PTC. `MAPA`<sup>2</sup> seems to have some appealing features in its class hierarchy as it seems that by adding one extra-layer it becomes more or less PTC, but honestly we are not capable of judging all the pros and cons of these new products. The purpose of this paper is to describe how we think it should be done and to document a modest attempt in FORTRAN90 rather than C++.

PTC now supersedes completely the FORTRAN90 version of `SixTrack` which was a superset of the old `Small_Code`. It has the functionality of `SixTrack`, `TRACYII` (Old `Despot`) and small machine capabilities. Because of the clean separation between charts and magnets, there is no limit to its ability to handle small rings and multiple beam lines as well as recirculators.

### A.2 A Short Description Using Los Alamos PSR

We will glance over the code in general terms. We will use the PSR as an example. In this section we do not describe in detail all the commands; we merely demonstrate a few commands. In the second part of this paper (starting with Section 2), we will describe the commands and the various internal procedures more systematically. In this first example we use a standard lattice which any code could produce easily. It is however a lattice famous for displaying “small ring” effects.

---

<sup>1</sup>LEGO was developed at SLAC by Yunhai Cai in C++.

<sup>2</sup>The program MAPA is a C++ piece of software from the TECH-X company.



### A.2.1 The Layout and an Example: PSR Storage Ring

The layout is a FORTRAN90 type (it would be a class in C++). It represents a standard beam line, i.e., a collection of magnets with the necessary charts at the entrance and exit of the magnets (drifts are magnets for the purpose of this discussion). As we will see, the layout is realized by a doubly linked list in FORTRAN90. Each node of the list is of type *fibre*. Each fibre contains the magnet, the charts and the patches associated to the variable “s” which happens to be discrete in a tracking code. One can think of the layout as being the ordered collection of the variable  $s_i$  that positions a magnet in the standard theory. Here however, we do not insist at all for a continuous connection between the variable  $s_i$  and the Hamiltonian used in the integration of each magnet. This is truly the power of a map based theory: continuous  $s$ -dependent Hamiltonians for the ring are just special cases. In a sense the layout type is really the fibre bundle of the mathematicians: perhaps the reader prefers here to stick to the more descriptive “layout.”

It has been argued by Forest since circa 1990, that tracking codes are deficient because they usually do not contain an atlas. What is an atlas? An atlas is a collection of charts.<sup>3</sup>

For each element in the beam line we associate as a minimum an entrance chart and an exit chart. The routine (the transfer map) that tracks the ray assumes that the entrance coordinates are with respect to the entrance chart and the exit coordinates are with respect to the exit chart.

It has been shown that if the magnets do not interact (no true space charge for example), then the magnet or the element as we call it, takes a dynamical existence of its own; more precisely the flow through the magnet becomes a mathematical object with rotational and translational properties similar to that of the physical object. The ontological nature of a magnet, when single particle maps are used, can be realized **if and only if** the atlas and the collection of magnets are theoretically unrelated. This is neither the case of TEAPOT nor MAD8/9 nor SAD[1, 2].

We start with the definition of the layout. Comments are preceded by the exclamation point (!) in the FORTRAN90 style. Before displaying the layout, we should display each node of the layout, namely the type fibre:

```
TYPE FIBRE
  ! BELOW ARE THE DATA CARRIED BY THE NODE
  INTEGER, POINTER :: DIR
  REAL(DP), POINTER :: POC, BETA0
  TYPE(PATCH), POINTER :: PATCH
  TYPE(CHART), POINTER :: CHART
  TYPE (ELEMENT), POINTER :: MAG
  TYPE (ELEMENTP), POINTER :: MAGP
  ! END OF DATA
  ! POINTER TO THE MAGNETS ON EACH SIDE OF THIS NODE
  TYPE (FIBRE), POINTER :: PREVIOUS
  TYPE (FIBRE), POINTER :: NEXT
  ! POINTING TO PARENT LAYOUT AND PARENT FIBRE DATA
  TYPE (LAYOUT), POINTER :: PARENT_LAYOUT
  TYPE (FIBRE), POINTER :: PARENT_PATCH
  TYPE (FIBRE), POINTER :: PARENT_CHART
  TYPE (FIBRE), POINTER :: PARENT_MAG
END TYPE FIBRE
```

The type fibre is recursively defined. This will allow the creation of a linked list. Of course a list is interesting only if it contains data. One can think of a linked list as a chain; on each link hangs potentially some data. In our case the fundamental datum is the object CHART of type CHART. This contains three actual charts (affine frames of reference): one at each end of the fibre and one in the middle. In addition we have the beam element MAG and its polymorphic version MAGP. MAG is the generic magnet to which we will attach a single particle propagator. MAGP is almost a carbon copy of MAG. The integer DIR defines the direction of propagation through the fibre. We can either enter the magnet from the front or from the back. POC and BETA0 define a preferential frame of reference for the energy of this fibre. It is usually the same as the energy of the Element MAG. We will come back later to this apparent departure from accelerator physics, the relegation of the magnet to a secondary role behind the charts.

<sup>3</sup>In English we have the word chart and the word map. Obviously in accelerator physics we want to avoid using the word map when talking about a chart.

Finally we have the pointers PREVIOUS and NEXT which must point to the previous and next link of the chain respectively. The pointers of the PARENT type are best explained later. Essentially they help us uncover the true nature of the data within the fibre: in PTC data can be cloned (the usual way in accelerator physics) or pointed at in the case of recirculators and elements shared between beam lines. Without the PARENT pointers it is not possible to copy or print a recirculator, for example, even though it exists inside PTC at execution time! See again Sect. N.1 and particularly the metaphor of the door to door peddler.

Now the chain itself is of type Layout. It is given by:

```

TYPE LAYOUT
  CHARACTER(120), POINTER :: NAME ! IDENTIFICATION
  INTEGER, POINTER :: INDEX, CHARGE ! IDENTIFICATION, CHARGE SIGN
  LOGICAL, POINTER :: CLOSED
  INTEGER, POINTER :: N ! TOTAL ELEMENT IN THE CHAIN
  INTEGER, POINTER :: NTHIN ! NUMBER OF THIN LENSES IN COLLECTION (FOR SPEED ESTIMATES)
  REAL(DP), POINTER :: THIN ! PARAMETER USED FOR AUTOMATIC CUTTING INTO THIN LENS
  !POINTERS OF LINK LAYOUT
  INTEGER, POINTER :: LASTPOS ! POSITION OF LAST VISITED
  TYPE (FIBRE), POINTER :: LAST ! LAST VISITED
  !
  TYPE (FIBRE), POINTER :: END
  TYPE (FIBRE), POINTER :: START
  TYPE (FIBRE), POINTER :: START_GROUND ! STORE THE GROUNDED VALUE OF START DURING CIRCULAR SCANNING
  TYPE (FIBRE), POINTER :: END_GROUND ! STORE THE GROUNDED VALUE OF END DURING CIRCULAR SCANNING
END TYPE LAYOUT

```

This linked list is explained in Figure 3.

Here we displayed a layout with four elements. Obviously this number could be enormous in an actual beam line. First we see unimportant data specific to the layout itself: data concerning steps of integration statistics. Secondly, besides the fibres, the two most important quantities in this layout are N and CLOSED. The variable N is simply the number of fibres in the layout which is often the number of magnets in non-recirculating objects. It is updated each time magnets are inserted or deleted. The boolean CLOSED refers to the topology of the so-called “base space” or, in accelerator parlance, the  $s$ -variable. Is the variable “ $s$ ” periodic (a ring) or is the variable  $s$  simply defining an interval (a straight beam line)? In the case of a ring, CLOSED is set to true, otherwise it is set to false. This does not happen automatically: the user must set it to the desired value.

In standard accelerator physics, the connection between the variable  $s$ , the local phase space coordinates  $\vec{z}$  and the global coordinates of the underlying topological space (in our case usually the global  $R^6$ ) is continuous. Therefore an  $s$ -dependent smooth Hamiltonian is written and thus pops the standard Courant-Snyder theory and its nonlinear extensions. Of course this is highly restrictive and mathematically unwarranted. Here there is no assumed existence of any smooth theory. It is absolutely unnecessary since all the quantities of interest (lattice functions, etc...) will be extracted using a finite “time” perturbation theory, i.e., a theory based on approximate Taylor maps.

The second type of variables in the linked list are the pointers. Their role is best explained looking at Figure 3. First of all, a linked list must have at least one pointer, this allows the creation of a simple linked list rather than a doubly linked (two-way) list. In our case this role is played by the pointer PREVIOUS. In a simple linked list, each node has the red “previous” pointer. The pointer END points to the end of the list, in our case fibre #4. Then the list is traversed backwards using the pointer “previous.” The reader, not familiar with linked lists may think that this is the result of poor programming— unfortunately this is what happens if one attempts to develop the simplest linked list.

Therefore a more complex structure must be introduced to handle two-way traversing. To do this we introduce the pointer “START” and at each node we add the green pointer “NEXT.” The green pointer points to the next element until it finally ends on the START pointer. The START pointer as well as the END pointer are nullified or grounded. Thus one can perform an “ASSOCIATED” check on either NODE%NEXT or NODE%PREVIOUS to determine if the extremities of the list have been reached.

Finally, we want to have a *circular* linked list for tracking a ring. When circular, the magenta link pointing to the grounded START is cut. The last fibre’s green pointer now points to fibre #1 (blue link). Thus the list is made circular in the forward direction. The purpose of the START\_GROUND pointer is precisely to remember the “location” of the grounded START pointer to re-establish the ordinary two-way terminated list if necessary. The same type of operation is performed on the magenta link pointing to the

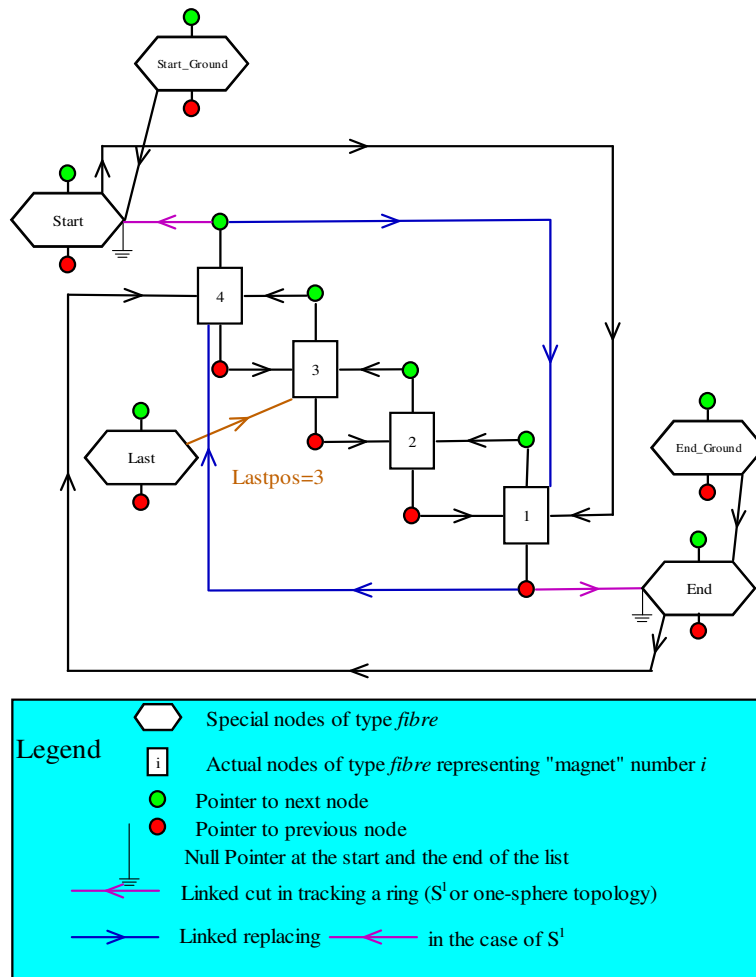


Figure 3: A linked list Layout in PTC

END pointer. The list is then fully circular. PTC has routines which permit a programmer to toggle back and fourth between terminated and circular lists; they are shown in Sect. N.1.

Finally, we have the LAST pointer. This pointer simply remembers the last fibre which was accessed by any of the maintenance routines. This allows the routine MOVE\_TO, which locates an actual fibre, to find its target faster. Obviously in our case speed is not of great importance: we tend to traverse a LAYOUT in the order of tracking— this is not a phone book. Let us try an actual example, a single cell of the PSR storage ring at Los Alamos. In this example, the layout has 7 elements or fibres (which includes drifts). Each fibre has a variable of type CHART which contains frames at the beginning in blue, in the middle in green, and at the end in red.

The following program is a real example of PTC where the lattice and the operations on it are done in the main program called RUN\_PSR.

```

PROGRAM RUN_PSR
USE S_TRACKING
USE S_FITTING
USE MAD_LIKE
IMPLICIT NONE
INTEGER ND2,NPARA
TYPE(REAL_8) Y(6)
TYPE(NORMALFORM) NORMAL
TYPE(LAYOUT) PSR
TYPE(LAYOUT) CELL,RING
TYPE(FIBRE) D1,QD,QF,D2,B
REAL(DP) X(6),KF,KD,ANG,BRHO,MIS_ROT(6)

CALL MAKE_STATES(.FALSE.)

```

```

EXACT_MODEL=.TRUE.
DEFAULT=DEFAULT+NOCAVITY+EXACTMIS
CALL UPDATE_STATES
MADLENGTH=.FALSE.

ANG=(TWOPI*36.DO/360.DO); BRHO=1.2D0*(2.54948D0/ANG);
CALL SET_MAD(BRHO=BRHO,METHOD=6,STEP=10)
MADKIND2=DRIFT_KICK_DRIFT

KF=2.72D0/BRHO; KD=-1.92D0/BRHO;

D1 = DRIFT("D1",2.28646D+00);D2 = DRIFT("D2",0.45D+00);
QF = QUADRUPOLE("QF",0.5D0,KF);QD = QUADRUPOLE("QD",0.5D0,KD);
B = RBEND("B",2.54948D0,ANG);
CELL= D1+QD+D2+B+D2+QF+D1; PSR=10*CELL;
PSR=.RING.PSR
CALL SURVEY(PSR)
CALL CLEAN_UP

X=0.DO; CALL FIND_ORBIT(PSR,X,1,DEFAULT) ! DEFAULT IS A STATE

CALL INIT(DEFAULT,3,0,BERZ,ND2,NPARA)
CALL ALLOC(Y);CALL ALLOC(NORMAL); ! ALLOCATE VARIABLES
Y=NPARA
Y=X

CALL TRACK(PSR,Y,1,DEFAULT)

NORMAL=Y; WRITE(6,*) NORMAL%TUNE;
CALL DAPRINT(NORMAL%DHDJ%V(1),6) ;CALL DAPRINT(NORMAL%DHDJ%V(2),6);

CALL KILL(Y);CALL KILL(NORMAL);

END PROGRAM RUN_PSR

```

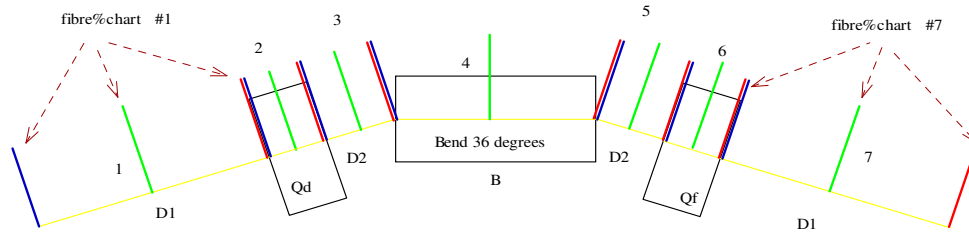


Figure 4: A Cell of PSR

The program RUN\_PSR uses PTC pseudo-MAD overloaded input style. The results of PTC can be compared to that of reference [3] in which Dragt did a careful study of the chromaticity of small rings. We must look at Table V and equations (3.19) and (3.20) of Dragt's paper. The set of calls

```

CALL MAKE_STATES(.FALSE.)
EXACT_MODEL=.TRUE.
DEFAULT=DEFAULT+NOCAVITY+EXACTMIS
CALL UPDATE_STATES
MADLENGTH=.FALSE.

```

are very important in PTC. Certain tracking internal states are set by the call to MAKE\_STATES. These states affect either the actual ray tracking or the type of TPSA calculation one intends to do. The input to MAKE\_STATES is true if we deal with electrons (positrons actually) and false for protons. There is also a global parameter concerning the nature of the Hamiltonian being<sup>4</sup> used. If EXACT\_MODEL is set to true, then the full "square root" Hamiltonian will be used. This is supported so far for the straight elements, the

<sup>4</sup>The MAD-like input of PTC looks at EXACT\_MODEL to figure out which model to use for a given magnet.

rectangular bends, and sector<sup>5</sup> bends. In his paper, Dragt used the correct Hamiltonian with a first order fringe field effect at the pole face (See Forest’s book [5], page 383. Equation (13.13.f), for the final path length, is incorrect, please interchange  $\ell^f$  and  $\ell$ .)

DEFAULT is something called an internal state (type INTERNAL\_STATE). The user can create his own and certain operations are allowed on them; we will discuss that later. Then all the possible predefined states can be upgraded with “CALL UPDATE\_STATES.” Finally, a global flag relevant to the MAD-like input is changed from TRUE to FALSE. This flag, if true, accepts the Cartesian length in the command for the rectangular bend. Thus, in this example, the command

```
B = RBEND("B", 2.54948D0,ANG)
```

uses the arc length of 2.54948 meters rather than the Cartesian distance between the parallel faces of the bend. We perform a standard survey in this lattice (CALL SURVEY(PSR)).

When we create the bend B using the standard command RBEND, the code defines an ideal Cartesian length (B%P%LC), an ideal  $1/\rho$  (B%P%B0), an ideal arc length<sup>6</sup> (B%P%LD), and an ideal tilt angle (B%P%TILTD). The code also defines the ideal relativistic  $\beta_0$  (B%P%BETA0) which it uses for the computation of time<sup>7</sup> if desired by the user. These are really internal survey data. The integration of a rectangular magnet in the exact model uses Cartesian variables. It is not done around the arc LD. Thus the code explicitly needs the actual  $B_y$ . In the case of the ideal magnet it is also numerically equal to B%P%B0. It is stored in the multipole content array B%MAG%BN(1). Now, consider the following interesting experiment. We increased the value<sup>8</sup> of B%MAG%BN(1) by 25% and recomputed the survey and plotted the new closed orbit.

```
B = RBEND("B", 2.54948D0,ANG); B%MAG%BN(1)=B%MAG%BN(1)*1.25d0; CALL COPY(B%MAG,B%MAGP);
```

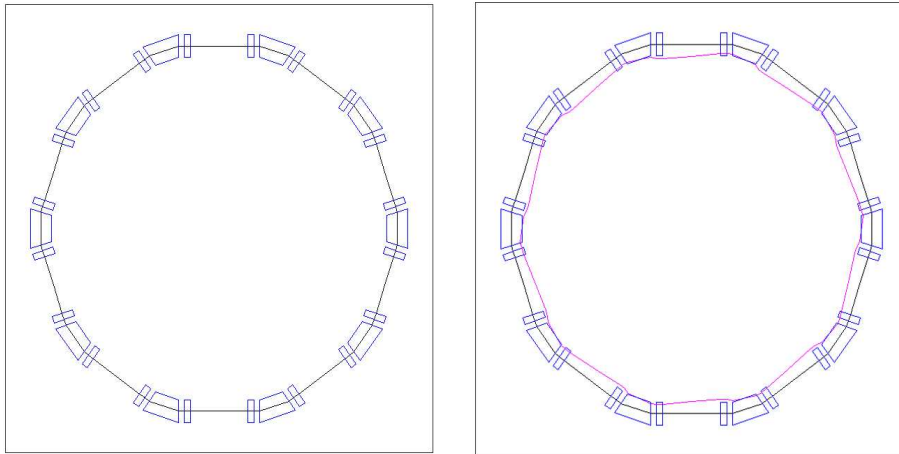


Figure 5: Changing the B-field of the Bends in PTC

As is seen in Figure 5, the survey remained at the same place since it depends on the survey variables LC, LD,B0, and TILTD. However the closed orbit, in magenta, shrunk considerably because of the stronger magnetic field. This example shows the ability of the layout type to decouple the ideal field of a bend, which determines the layout in a standard survey, from the true B field of the magnet. In fact there is no relation whatsoever between layout and field. A relation, if any, is imposed by standard survey commands.

### A.2.2 The Importance of the Charts: Misalignments and the Euclidean Group

Each fibre contains 7 data objects: an ELEMENT (MAG), an ELEMENTP (MAGP), a CHART (CHART), a PATCH (PATCH), the directional integer pointer DIR and two less important real variables<sup>9</sup> P0C and

<sup>5</sup>Sector bends are easy to support if we ignored the curvilinear effects of the bend on Maxwell’s equations. Then we simply have the model of TEAPOT[4]. In PTC we assume a circular geometry and solve Maxwell’s equations to an order specified by the user.

<sup>6</sup>LD is really the ideal path length or time of flight.

<sup>7</sup>Actually the code also keeps redundant quantities such as  $p_{0c}$  in addition to  $\beta_0$  in the element definition.

<sup>8</sup>It is not always possible to simply change the multipole array as done here; PTC provides certain procedures and types to do it reliably. However it is correct in the present case.

<sup>9</sup>They are not used in PTC at the moment.

BETA0. ELEMENT is simply a magnet of some sort while ELEMENTP allows for the flow of Taylor series through polymorphism. Without the chart "CHART," the fibre is simply a magnet at "the factory." It does not know how to misalign itself with respect to the beam pipe. In mathematical terms, the chart connects the local coordinates of the magnet (in type MAGNET\_CHART) to the local coordinates attached to the pipe. The quantity PATCH contains additional elements of the Euclidean group which permit the patching of a fibre with the preceding and following fibre. This patching is geometrical (SO(3)) using the dynamical Euclidean group as well as "energetic" in case the magnets in surrounding fibres have different reference energies. This is the case in a recirculator for example. The directional integer DIR refers to the direction of propagation inside this fibre as PTC fully supports forward and reverse propagation. Finally it is possible to give the fibre a reference energy different from that contained in the magnet. This energy information is stored in P0C and BETA0 and is defaulted to the magnet energy. Again this is useful in a recirculator since the reference energy we may want to use, for displaying purposes, may vary each time we recirculate through a magnet. Since the fibre is different (remember it is really the "s" variable) but the magnet is by definition the same, a different reference energy, if desired, must be attached to the fibre. At this point PTC does not use these energy variables but they are there for the user.

Thus in practice, the fibre structure permits rigorous translations and rotations of the element in physical space. In contrast, standard accelerator physics assumes a continuous connection between the layout variable  $s$ , the local magnet variables, and the global space in which the magnet is immersed. This is assumed in order **to enforce**, *manu militari*, a continuous  $s$ -dependent Hamiltonian theory. This approach prevents the use of discontinuous patches that are necessary for a sensible descriptions of complex magnets and for the elevation of the magnet into a dynamical object having well-defined properties under rotations and translations.

For example, many tracking codes, without our structure, require the introduction of internal changes of the magnet as a result of misalignments such as the introduction of feed-down multipole terms: this clearly invalidates the "ontological" nature of the magnet-object. Other more exact codes, such as MAD8/9, introduce two new pathological elements on both sides of a magnet to misalign it.

Many people have looked at our ideas and call them "novel." Actually we have been pushing them for a long time and they are well-known in other fields. In addition, they can be found in the work of other accelerator physicists, for example Michelotti in his book[6], on page 10, refers to related ideas in what he calls "two minor comments." Actually Michelotti talks explicitly about the freedom one has in choosing an "atlas of charts" and that all of this probably done unconsciously in accelerator tracking codes: we could not agree more.

Now let us illustrate the use of the type ELEMENT by calling directly the tracking routine<sup>10</sup> on the fourth element of the layout PSR, which is just the bend B. For example we start with the ray  $X(5) = \delta p/p_0$  and track it through the fourth element (here we require a pointer P to fetch this element):

```
TYPE(FIBRE), POINTER :: P
INTEGER, TARGET :: UNO=1
.
.
.
NULLIFY(P)
CALL MOVE_TO(PSR,P,4)
WRITE(6,*) "THE NAME IS ",P%MAG%NAME
X(:)=0.DO; X(5)=0.1D0;
P%MAG%P%DIR=>UNO;P%MAG%P%CHARGE=>UNO;
CALL TRACK(P%MAG,X)
WRITE(6,*) X
```

The result of this piece of code is:

```
THE NAME IS B
 7.117497437015063E-002  6.437143094882025E-002  0.000000000000000E+000
0.000000000000000E+000  0.10000000000000000  1.633652773351549E-002
```

The routine TRACK is the fundamental routine which controls ordinary tracking as well as map computations used in perturbation theory. Thanks to TPSA and polymorphism all the quantities of ordinary perturbation theory are computed by using Normal Form theory and the subroutine TRACK. This subroutine is found in Sj\_ELEMENTS.f90.

<sup>10</sup>Actually in the most recent version of PTC, this call cannot even track unless one gives to the magnet the direction of propagation (FIBRE property) and the sign of the charge (global LAYOUT property). It can be done with P%MAG%P%DIR=>UNO;P%MAG%P%CHARGE=>UNO;

The reader will notice that, in the above example, there is no mention of the layout. When only the magnet is mentioned, the tracking is performed as if the magnet is literally sitting on the factory bench with no concern about its surrounding. At this point the PSR is in its ideal state, therefore the equivalent tracking through the layout will produce the same results. This would not be true in recirculators or other bizarre lattices where patches are required to connect the magnet's intrinsic frames to that of the machine where it is used.

CALL TRACK(P%MAG,X) is replaced by CALL TRACK(PSR,X,4,5,DEFAULT)

THE NAME IS B

```
7.117497437015063E-002 6.437143094882025E-002 0.000000000000000E+000
0.000000000000000E+000 0.100000000000000 1.633652773351549E-002
```

Now, let us misalign this fourth element by a translation of  $dz = 10^{-4}m$  in the longitudinal direction. Notice that the default state uses EXACTMIS. This ensures that misalignments are done exactly. Now consider the following piece of code:

```
NULLIFY(P)
CALL MOVE_TO(PSR,P,4)
WRITE(6,*) "THE NAME IS ",P%MAG%NAME

MIS_ROT(:)=0.DO;MIS_ROT(3)=1.D-4;
P=MIS_ROT;      ! Same as CALL MISALIGN_FIBRE(P,MIS_ROT)

X(:)=0.DO
P%MAG%P%DIR=>UNO;P%MAG%P%CHARGE=>UNO;
CALL TRACK(P%MAG,X)
WRITE(6,*)"ELEMENT TRACKING :", X(1),X(2)
X(:)=0.DO
CALL TRACK(PSR,X,4,5,DEFAULT)
WRITE(6,*)"LAYOUT TRACKING :", X(1),X(2)
```

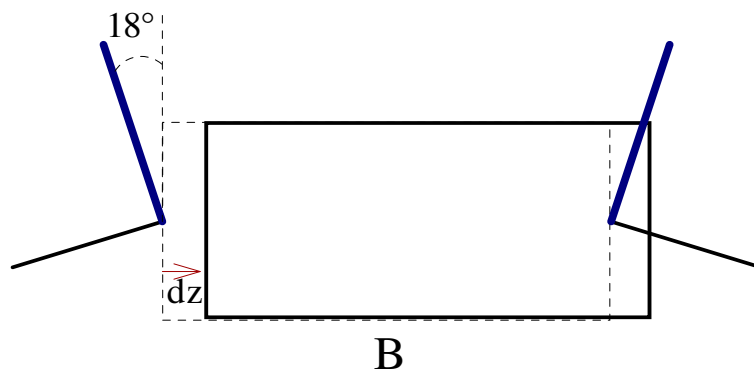


Figure 6: Translation of a Bend

In this piece of code, a double precision (real(dp)) array is used to put the misalignments in the fibre P using an overloaded equal sign. MIS\_ROT(1:3) contains the translation and MIS\_ROT(4:6) the rotation. (Some may simply prefer the actual call “CALL MISALIGN\_FIBRE(P,MIS\_ROT).”) The result of the above code is

```
THE NAME IS B
ELEMENT TRACKING : -5.939352236872518E-017 5.551115123125783E-017
LAYOUT TRACKING : 6.180339887489015E-005 5.551115123125783E-017
```

As we said, an element inherits from the FIBRE%CHART the knowledge necessary for “misaligning itself.” Therefore the call to TRACK(P%MAG,X) is incapable of producing misalignments. However the call to TRACK(PSR,X,4,5,DEFAULT) takes this data into account. It is interesting to look at the way the first obsolete versions of PTC approached this problem:

```

IF(C%MAG%MIS) THEN
  CALL ROT_XZ( C%CHART%ALPHA/2.DO ,X, C%MAG%P%BETAO,OUR,C%MAG%P%TIME)
  CALL TRANSZ( C%CHART%L/2.DO ,X, C%MAG%P%BETAO,OUR,C%MAG%P%TIME)
  CALL ROT_YZ(C%MAG%R(1),X,C%MAG%P%BETAO,OU,C%MAG%P%TIME)
  CALL ROT_XZ(C%MAG%R(2),X,C%MAG%P%BETAO,OU,C%MAG%P%TIME)
  CALL ROT_XY(C%MAG%R(3),X,OU)
  CALL TRANS(C%MAG%D,X,C%MAG%P%BETAO,OU,C%MAG%P%TIME)
  CALL TRANSZ( -C%CHART%L/2.DO ,X, C%MAG%P%BETAO,OUR,C%MAG%P%TIME)
  CALL ROT_XZ( -C%CHART%ALPHA/2.DO ,X, C%MAG%P%BETAO,OUR,C%MAG%P%TIME)
ENDIF

```

```
CALL TRACK(C%MAG,X)
```

```

IF(C%MAG%MIS) THEN
  CALL ROT_XZ( -C%CHART%ALPHA/2.DO ,X, C%MAG%P%BETAO,OUR,C%MAG%P%TIME)
  CALL TRANSZ( -C%CHART%L/2.DO ,X, C%MAG%P%BETAO,OUR,C%MAG%P%TIME)
  CALL TRANS(-C%MAG%D,X,C%MAG%P%BETAO,OU,C%MAG%P%TIME)
  CALL ROT_XY(-C%MAG%R(3),X,OU)
  CALL ROT_XZ(-C%MAG%R(2),X,C%MAG%P%BETAO,OU,C%MAG%P%TIME)
  CALL ROT_YZ(-C%MAG%R(1),X,C%MAG%P%BETAO,OU,C%MAG%P%TIME)
  CALL TRANSZ( C%CHART%L/2.DO ,X, C%MAG%P%BETAO,OUR,C%MAG%P%TIME)
  CALL ROT_XZ( C%CHART%ALPHA/2.DO ,X, C%MAG%P%BETAO,OUR,C%MAG%P%TIME)
ENDIF

```

The reader will notice the mathematical structure of these calls. They are of the form:

$$E(T) = E_{-1}^{out} \circ T \circ E^{in} \quad (1)$$

where the map  $T$  represents the call TRACK(C%MAG,X)— the magnet “on the factory bench.” Moreover the maps  $E_{-1}^{out}$  and  $E^{in}$  are made of rotations and translations. They depend **only** on the layout charts and not on the element C%MAG. We can be a bit more precise and expand this expression further:

$$E(T) = L_t \circ E^{-1} \circ \underbrace{L_t^{-1} \circ T \circ L^{-1}}_{T \text{ compressed}} \circ E \circ L \quad (2)$$

Standard Operation
   

  
 $T$  re-expanded

The operator  $T$  is just our element “on the factory bench,” the operators  $L$  depend **only** on the layout, and finally, the operator  $E$  is just the standard Euclidean group operators in their dynamical representation. The entire gymnastics creates a “thin” or “compressed” Cartesian element. Such an element transforms under the Euclidean group in the usual geometrical manner.

The present version of PTC uses a “PTC standard factorization” of the Euclidean operator and it is used for all Euclidean group operations—misalignments and patches:

```

IF(C%MAGP%MIS) THEN
  ou = K%EXACTMIS.or.C%MAGP%EXACTMIS
  CALL MIS_FIB(C,X,OU,.TRUE.)
ENDIF

```

```
CALL TRACK(C%MAGP,X)
```

```

IF(C%MAGP%MIS) THEN
  CALL MIS_FIB(C,X,OU,.FALSE.)
ENDIF

```

The subroutine MIS\_FIB is just, in the case of forward propagation,

```

IF(ENTERING) THEN
  CALL ROT_YZ(C%CHART%ANG_IN(1),X,C%MAGP%P%BETAO,OU,C%MAGP%P%TIME)
  CALL ROT_XZ(C%CHART%ANG_IN(2),X,C%MAGP%P%BETAO,OU,C%MAGP%P%TIME)
  CALL ROT_XY(C%CHART%ANG_IN(3),X,OU)
  CALL TRANS(C%CHART%D_IN,X,C%MAGP%P%BETAO,OU,C%MAGP%P%TIME)
ELSE
  ! EXITING

```



```

CALL ROT_YZ(C%CHART%ANG_OUT(1),X,C%MAGP%P%BETA0,OU,C%MAGP%P%TIME)
CALL ROT_XZ(C%CHART%ANG_OUT(2),X,C%MAGP%P%BETA0,OU,C%MAGP%P%TIME)
CALL ROT_XY(C%CHART%ANG_OUT(3),X,OU)
CALL TRANS(C%CHART%D_OUT,X,C%MAGP%P%BETA0,OU,C%MAGP%P%TIME)

```

ENDIF

Of course the arrays (ANG\_IN,D\_IN) and (ANG\_OUT,D\_OUT) are computed on the basis of the magnet arrays (C%MAG%R,C%MAG%D) at the moment of misalignment. This is done in the module ROTATION\_MIS using solely the group SO(3) acting on the affine space  $R^3$ . In that space, all the operators of the original “compressed” Cartesian element are well behaved, and thus this new version of PTC can handle magnets near 180 degrees as discussed in Sect. G.4.4. It should be said that one can use the routines of ROTATION\_MIS to preprocess (C%MAG%R,C%MAG%D) and thus use a magnet body frame different from that of PTC.

### A.2.3 Tracking a Single Fibre

In PTC, it is possible to track<sup>11</sup> a single fibre. Therefore the mathematical expression of Equation (2) is a single call in PTC, namely:

```
CALL TRACK(C,X,STATE,CHARGE)
```

## A.3 Comparing and Contrasting C++ CLASSIC Classes and Multiple Inheritance with Composition (Forward Delegation)

In this section we present, what is in our opinion, an obvious flaw of CLASSIC/MAD9; namely, the failure to create a fibre bundle. We also present a description of composition (forward delegation) in FORTRAN90. (A more in-depth treatment of the OO aspects of FORTRAN90, as applied in PTC, can be found in Ref.[7].) We actually contrast three techniques, fibre bundles by composition (PTC), fibre bundles by multiple inheritance (but are not aware of any existing implementation), and the definition of lots of classes to be used by algorithms to manage standard beam lines(MAD9/CLASSIC) without the fibre bundle concept.

### A.3.1 Fibre Bundles (or not)

It is a key feature of the design that our type CHART transforms geometrically in the usual manner under the Euclidean group. By expressing the map as a “thin map” around the center chart (the green one in Figure 4), we create a map which transforms under the Euclidean group like a geometrical object. Thus, the fibre inherits the transformational properties of the chart. (See explicitly the fibre propagator in Equation (2) and Sect. A.2.3.) The fibre then inherits its tracking methods from either the element MAG (or the polymorphic element MAGP). They, in turn, inherit their tracking methods from the various on-the-bench magnets. This is done by forward delegation (or composition) rather than multiple inheritance (which is not supported in FORTRAN90 anyway). This will be described below with the help of Figure 11.

In this representation, the most fundamental datum within the fibre is the variable FIBRE%CHART of type CHART. A magnet may well exist as a complicated piece of metal within a layout. As such it can be moved and displayed by a CAD-like program, but it cannot be tracked. Secondly, it may also exist as a generator of a magnetic field but not as a well defined propagator. This happens in the presence of “true” space-charge or other types of collective effects. In this case the propagator given by the subroutine TRACK, inherited from the magnet.i’s of Figure 11, is strictly speaking meaningless and thus its transformational properties under the Euclidean group are also meaningless. Finally, it is possible that the magnet has an elastic structure: it is not an inflexible solid. In this case the magnet is modified by internal misalignments. Since PTC gives full access to the internal parts of a magnet (all variables are public), one can certainly handle elastic magnets. Furthermore, in the case of true collective effects, the layout can be used as a database for a more complex time-base propagation ignoring the single particle propagators; the fibre bundle becoming, like the so-called design orbit, a figment of our imagination.

In conclusion, we have developed a type which takes full advantage of the “ontological” nature of the magnet propagator when it is valid, while still leaving enough freedom to allow more complex situations. In PTC, thanks to this layout structure, single particle calculations are always done correctly in the presence

<sup>11</sup>A suggestion of David Sagan of Cornell.

of misalignments simply because the propagators are never modified internally: *ad hoc* fudges like feed-down, introduction of fake B-field, etc. are no longer necessary. It is only when physics dictates a radical modification of the magnet that its internal integrity is violated.

In PTC it is possible to do a standard survey. This is equivalent to the Surveyor visitor function of the CLASSIC class structure in MAD9. In PTC all the elements have an implicit entrance and exit chart described by the parameters LD, LC, B0, and TILTD. PTC does not necessarily assume, in its structure, that the connection between the charts attached to a magnet (type MAGNET.CHART) and those of the layout is the identity map. However, if that is the case, then PTC can perform a survey. In the CLASSIC case, the Surveyor visitor function does the same thing: it takes the internal geometry of a magnet and performs a survey based on it.

In PTC, since we implemented essentially a fibre bundle structure, patches can be loaded to connect charts. They are not elements. This simply reflects, that from the start, we designed support for a mathematical framework in which the connection between the global space and the local magnet coordinates is neither assumed to be identity nor continuous; moreover, this connection is *not a property of the magnet* but a property of the fibre, i.e., the layout. This last point is crucial in recirculators and when elements are shared between two beam lines. In fact, if a fibre bundle is implemented and two beam lines with shared elements are read in, as in the LHC rings, then fitting two beam lines simultaneously is not only possible, but it is the only thing possible. Thus the MAD9 problems about fitting two rings simultaneously, are not in PTC *by construction*. It should be obvious that, if our objects are correctly implemented, then if a common quadrupole is modified, both rings will see the effect. In a real collider, one is not concerned about only ring #1 seeing a change if an element common to rings #1 and #2 is modified. In such a case it would be difficult, if not impossible, to isolate the rings.

Therefore the proper implementation of a fibre bundle was a **sine qua non**, non-negotiable point, which Forest and Bengtsson insisted upon in the early days of pre-CLASSIC C++ collaborations. A quick look at CLASSIC (MAD9) shows that the CLASSIC structure does not satisfy this condition. In particular, patches are derived from the beam element class and are thus of the same nature as the element. Patches are generally a figment of one’s mathematical imagination, useful tools, but they are not physical elements which can be ordered from a factory. We believe this is a major flaw in the CLASSIC design. It is perhaps the result of placing too much emphasis on implementation using C++ capabilities, rather than the basic mathematical framework. We believe this accounts for the excessive number of classes and the complexity of CLASSIC (MAD9).

**One can philosophize further about emphasizing algorithms rather than flow. PTC tracks rays and Taylor series through polymorphism, function interfaces and operator overloading. In addition PTC tracks stochastic moments using the so-called “beam envelop theory.” This means that an expression such as**

```
CALL TRACK(PSR,X,5,10,DEFAULT)
```

**could represent the flow of a myriad of things. It could be the flow of a simple ray in  $R^6$ : the usual function of a tracking code. It could also be the flow of a jet(see [5], p.223 ) , i.e., a truncated power series, thanks to TPSA (usually Berz’s “DA-package” in our case). Then, by adding to this package the polymorphism of FPP, the jet could become a jet in system variables such as quadrupole strengths.**

Therefore the first thing to realize with polymorphism and TPSA is that the concept of flow is potentially extended further than can be imagined. Secondly, thanks to the so-called “Hamiltonian-free” perturbation theory, jets can be used to compute the various quantities of accelerator physics: lattice functions, equilibrium beam sizes, etc... it is simply a matter of selecting the proper X and its subsequent tracking and analysis. The flow of X is of central importance. Algorithms based on TPSA act correctly on the flow and not on the magnet. Thus, if the strength  $K_f$  of a quadrupole<sup>12</sup> is declared as a parameter of the polymorph, the object being computed will be a jet in  $K_f$ . This will be done correctly, exactly, without knowledge of the inside state of this quadrupole. The quadrupole retains its inviolate property as a dynamical object. The design, based on flow, allows magnets to be well defined objects since “algorithms” are performed on the flow and not on the magnets themselves. Of course

<sup>12</sup>Of course this  $K_f$  points to something inside the magnet but one does not need to know what it is exactly. Indeed this quadrupole could be a horribly complex magnet with fringe fields, errors, correlations between  $K_f$  and some other factor— we do not care. In contrast, the analytical formulas of perturbation theory as found in most MAD modules must know all these things to be correct and they break down as lattices and magnets deviate from some ideal state.

this capability will break down at some point. But given that the propagators found in codes like MAD, TEAPOT, SixTrack, TRACYII, etc. all assume single particle dynamics in normal operational mode, it is a major weakness, in our view, that they fail to exploit the “object orientedness” of the theory. In any case, as mentioned previously, with PTC it is always possible to ignore the single particle propagator and use the magnets as pure database objects for simulation conditions incompatible with the magnet-object paradigm.

We do not necessarily believe that PTC is the ideal alternative to CLASSIC classes. We simply think that the CLASSIC classes have a design flaw, in addition to being rather complex and rigidly hierarchical. We have yet to hear a reasoned argument as to why our insistence on giving the flow of an individual magnet a mathematically sound object-oriented character would be a mistake. Given the power of a polymorphic TPSA, we cannot see any drawback in our general approach.

Finally, on reading further, it will become clear that our way of implementing multiple inheritance is close to what is called “forward delegation” or “composition” in computer science terminology. Inheritance works best when a subclass is truly ontologically an “is-a” of the class which it extends. In our case, the “is-a” relationship between the generic fibre, the chart (the geometrical magnet description) and the dynamical magnet/flow is not always realized. Certain conditions must be met before the flow through the magnet can acquire the status of an object. When composition is used, interfaces must be written to ensure polymorphism (interfaces for rays, Taylor and Beam Envelopes). The situation thus becomes more complex. However, CLASSIC has a far too complex and too rigid structure, and lacks the fundamental mathematical object of single particle dynamics.

### A.3.2 Composition, Inheritance and Algorithmic Classes

For the purposes of illustration we imagine the existence of a “Superbird”, a kind of domesticated flying horse, capable of flying while carrying a person in its claws.

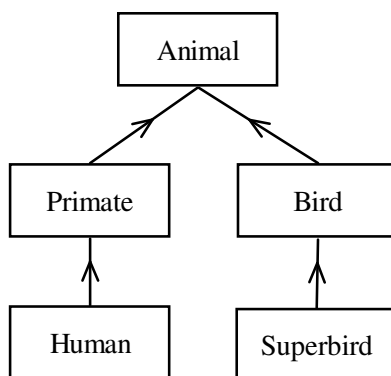


Figure 7: Normal Class Hierarchy for Humans and Superbirds

In Figure 7, we see the normal class hierarchy prior to any attempt to make humans fly with the help of superbirds, analogous to PTC without its single particle propagators. It might be the hierarchy of a program which ignores completely human flying (single particle tracking).

Our goal is to now implement human superbird flying. We will see that there are three ways which come to our attention. The first method, multiple inheritance, is highly intrusive and fundamentally modifies the hierarchical structure. The second method, composition, is less intrusive and preserves the inner sanctity of both the superbird and human classes while at the same time using the superbird class. This is how PTC is implemented.

Finally the last way simply ignores the Superbird class and uses algorithms visiting the Human class. These algorithms will produce Superbird flying without using or even creating this object. This is analogous to the MAD9/CLASSIC technique.

First we examine multiple inheritance. Clearly it is correct to use inheritance and say that the Human class extends the Primate class (JAVA syntax). But of course primates cannot fly. Furthermore, when our humans fly, they do so by being the equivalent of a weight carried by a superbird. Thus, in the language of multiple inheritance, we may say that Human inherits from Superbird. The advantage of this is that the code for Superbird can be re-used and that polymorphism is automatic. We already have the interface `superbird.fly()` and therefore `human.fly()` will correctly invoke `superbird.fly()`. The problem with this construction is that humans are not superbirds; in fact not even birds. In order to have code re-usability

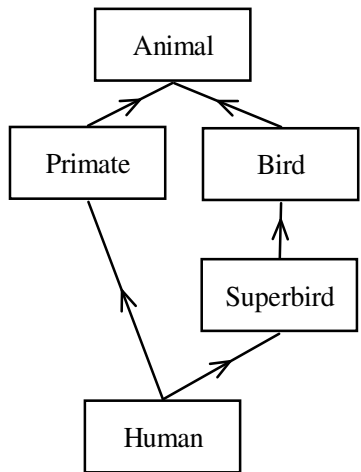


Figure 8: Human flying through Multiple Inheritance

and polymorphism, we create a logical category confusion. Will humans have feathers in this model? This is the “diamond problem” of multiple inheritance.

In addition, Superbird becomes a superclass of human and thus any change to the return value of `superbird.fly()` will ripple down in any code using the `human.fly()` method. This is nasty since superbirds ought to be able to change their color, for example, quite independently of humans.

The proper solution is to use composition methods as one would do in JAVA. In this case, the Human

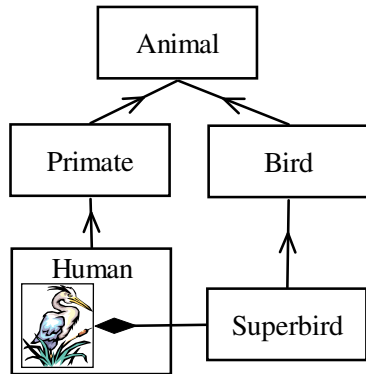


Figure 9: Human flying through Composition

class contains an instance of the Superbird class. The `Human.fly()` method is defined by an interface which calls directly the method `Superbird.fly()` on the instance of Superbird defined for each Human object. If one changes the return value of `Superbird.fly()`, then it is only necessary to change the interface to ensure that old code using the `Human.fly()` method can still be used. Thus composition provides a far stronger encapsulation than inheritance.

Of course if humans were actually superbirds, then inheritance would be more appropriate. It would still be true that a change in Superbird would ripple down the Human codes. However in that case the change in Superbird would imply that the original code was inadequate and badly designed for superbirds as well as for humans since humans are truly superbirds. The ripple down effect would be an unfortunate but necessary consequence of our mistakes in the Superbird superclass. And yes, in such a case, we would definitely have feathers.

It should also be said that as neither FORTRAN90 nor JAVA support multiple inheritance, but both support interfaces, we are obliged to use composition.

The effect of the `Human.fly()`, in either a composition or inheritance scheme, is to return some data consistent with a superbird flying a human between point A and point B. The code will perhaps return a time of flight, the energy used by the superbird, and potentially many other things which depend on the actual state of the human and of the superbird.

The third (algorithmic) option is used in CLASSIC/MAD9. Since humans are not superbirds, we may

want to describe the effect of a superbird flight as a complicated algorithm acting on the data contained inside the Human object. We do not deal at all with the superbird nor do we make Human a contrived subclass of Superbird. CLASSIC rather constructs a large class hierarchy of visitor algorithms. Why are these required? In procedural programming, new algorithms, i.e., new functions can be added without disturbing the data structures, i.e. without disturbing the objects. For example, in a code like SixTrack, one can do all sorts of calculations using its massive collection of common blocks, without a need to “rewrite” the common blocks. This is an extreme case of structureless procedural programming. In object oriented programming the situation is rather reversed. New data types can be added without affecting the existing functions. Unfortunately there are cases where we want to add a new function without disturbing the data type, i.e., the object. In standard accelerator physics, all algorithms are generated from formulas. Formulas are symbolic manipulations on the abstract “s”-dependent equations of motion. They are not fundamental objects<sup>13</sup> of PTC nor are they fundamental objects of CLASSIC. There are of course a nearly infinite collection of these formulas. It is therefore impracticable to incorporate each formula as a data type within the object beam element or, in our analogy, the object human. The concept of visitor functions is intended to provide a solution.

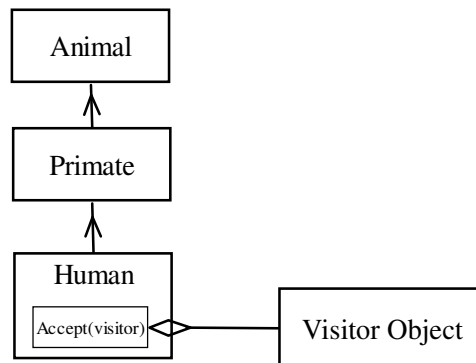


Figure 10: A particular aspect of Superbird flying comes for a visit!

Since algorithms are in general quite arbitrary and not necessarily ontologically related to the object on which they are applied, the concept of visitors is a valuable construct of object oriented programming. To the extent that we may have algorithms acting in a drastic way on our objects, for example flexible magnets deforming under rotations, it is certainly desirable for a CLASSIC-like structure to have a visitor class. In addition, there are certain computations, particularly first order computations, in which a formula related to mild collective effects, requires an integral over single particle propagators. This type of integral is not normally obtainable from the polymorphic magnet/flow object.

The draw back of implementing Human Superbird flying (or single particle magnet/flow) through visitor functions, is related to the fact that when flying (or when pushing single particles), our objects are temporarily inheriting from a superclass. Moreover, in the case of the magnet/flow, the algorithm which would normally act on the magnet data, really simply acts on the magnet/flow alone, once properly extended to include Real/Taylor polymorphs. The draw back is precisely the absence of a superbird. If a superbird is misfed (read misaligned magnet), loses a feather (read mispowered), it is not really a superbird any more; what guarantees do we have that a visitor function will return something consistent with true superbird flying? It is just an algorithm acting on the data of the Human class or, in CLASSIC, visiting the beam line objects. This is confirmed by the fact that MAD9, previous versions of MAD, and other codes based on this model fail to give the correct dispersion when some bizarre magnet is introduced or give wrong synchrotron integrals when cavities are misaligned, etc.

PTC avoids this completely because a large (very large) portion of the algorithms of single particle dynamics can be made to act not on the Human class but to act on the quantity returned by Human.fly() namely the single particle polymorphic(Real/TPSA) flow. Therefore if the Superbird was correctly implemented to react correctly when losing feathers or misfed, then Human.fly() will also react correctly. By extending the flow to include Berz’s TPSA variables polymorphically, we can get rid of a vast array of algorithms acting on the magnet and replace them by algorithms acting on the flow. Of course it is still possible to have algorithms acting on the individual magnets – analytical formulas are typical examples, but it is of-

<sup>13</sup>In PTC we promised no algorithms beyond tracking the flow and this is indeed the case. In CLASSIC, it is hard to imagine that the computation of the third order momentum compaction would suddenly become a fundamental object simply because someone decided that the knowledge of this number was relevant to his design efforts.

ten unnecessary. People who know only standard Courant-Snyder theory (99% of all accelerator physicists), understand only analytical formulas and are therefore sure to generate procedural code in relation to the magnet or flow object irrespective of their choice of programming language!

### A.3.3 More on delegation and PTC

In an accelerator, the single particle magnet propagators (magnet\_i of Figure 11) are various types of birds. These are our Superbirds. The ELEMENT is the actual magnet. It is in our Human class. In the case of single particle dynamics, it will then inherit its tracking method from one of the magnet\_i depending on the variable ELEMENT%KIND. The interface is provided in the module S\_ELEMENTS (see Sect. L).

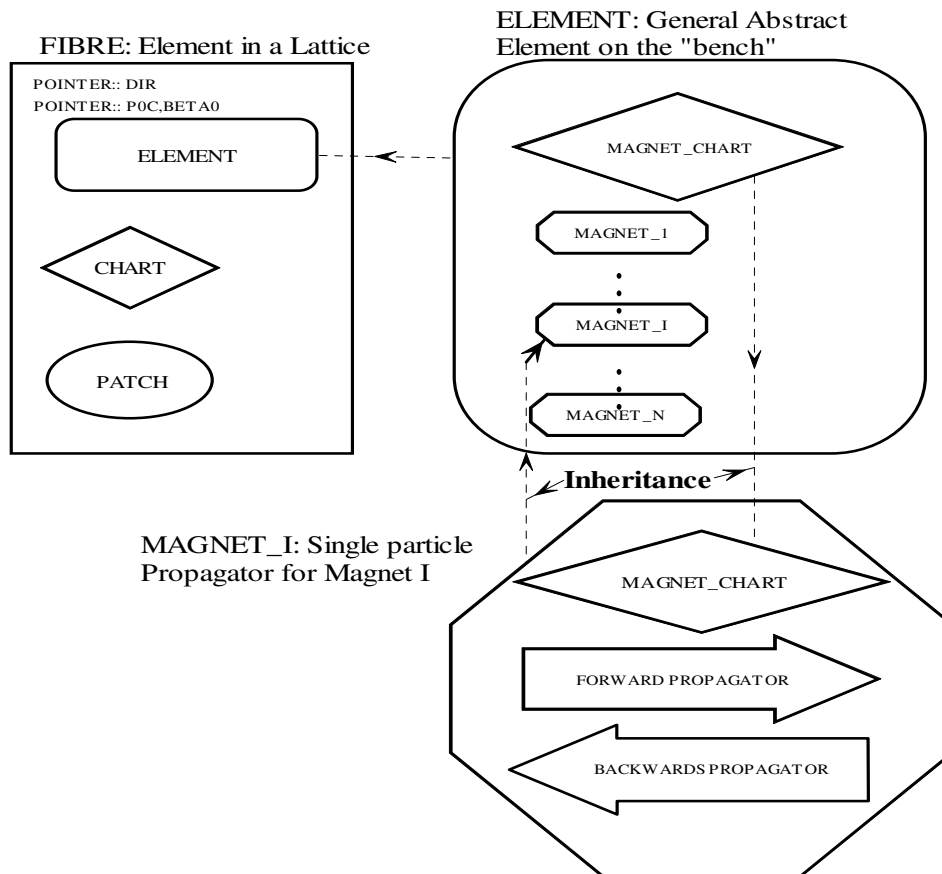


Figure 11: Fibre-Element-Magnet linked through Composition

Thus, in PTC, the flow through an element is inherited from one of the magnet\_i's through composition. While the ELEMENT inherits its single particle tracking properties from the magnets in S\_DEF\_KIND as well as some geometrical properties (MAGNET\_CHART), it is not a subclass of any of the magnet types. In fact we could write code which uses ELEMENT and totally ignores the propagators of magnet\_i. Moreover, the single particle propagators can be changed at will. In particular, we could change the interface without breaking any code beyond the interface module S\_ELEMENTS. It is important to realize that we elevated the single particle flow to an *occasionally* well-defined object without compromising the fact that perhaps the objects of type ELEMENT are not always really compatible with single particle propagators: humans are not always superbirds. In other words, type ELEMENT is the generic magnet in the absence of single particle propagators. This type can be used when our propagators are either inadequate or plainly irrelevant. For example, one may want to write a fast tracking structure made out of thin lens kicks to save time, or to study collective effects. The resulting structure is closer to the global Hamiltonian  $H(s)$  or even  $H(t)$ . Such an entity will have pointers to objects of type ELEMENT but is unlikely to use their single particle propagators.

In PTC the story does not end with the ELEMENT. We do not define a beam line as a sequence of elements as other codes do, but rather create a fibre bundle. Initially the term fibre bundle was never used by Forest. It appears only once in a paper[8] written with Hirata. This lack of mathematical terminology results from the emphasis on the very realistic idea of making the magnet flow rotate and translate like a

physical magnet, as in the types CHART and MAGNET\_CHART. Therefore in Forest’s papers one always finds emphasis on the issue of decoupling the layout (or tunnel) coordinates from the geometry of the magnet. There is a little of this idea in Dragt’s paper on chromaticities[3] in small rings in relation to the proper integration methods for parallel face bends. Mathematically, one simply needs to support a fibre bundle structure, which is an arbitrary connection between the outside world of  $R^6$  and the local coordinates used by the magnet propagators magnet.i’s used by ELEMENT. But this is not general relativity, this generality can be restricted to the group  $SO(3)$ , the group of rotations in three dimensions. Thus in PTC we have a “restricted” fibre bundle built into the type FIBRE.

The element FIBRE, through composition, inherits properties from the CHART and the ELEMENT(P). The geometrical properties and the dynamical Euclidean group are inherited from CHART. The single particle propagator is inherited from ELEMENT(P). Together these give us the FIBRE propagator. Suddenly a magnet knows how to misalign itself. The third important object in the FIBRE is the PATCH. The patches are the hooks and eyelets which will permit the arbitrary connection of fibres into a layout – the true beam line of PTC. Each fibre of a layout is the discontinuous “s” variable of standard Courant-Snyder theory. A fibre may contain a magnet or simply a pointer to an existing magnet. The same is true of a patch contained by a fibre. This flexibility allows for the complete management of recirculators and the common beam lines.

Because there is no strict inheritance, only interfaces, one can certainly ignore the propagators within a given magnet.i or invent new ones. In addition, the magnetic field and electric field structure of these magnets are totally irrelevant. PTC does not concern itself with the form or the look of the internal fields. This ensures maximum flexibility. A magnet designer may create a complicated structure: super bend at ALS, FFAG as generated with TOSCA, etc... all that PTC needs is that certain interfaces are provided. **The internal details are irrelevant; the user can develop hundreds of classes for describing his new magnet in enormous detail. The internal symmetries of a magnet are of no concern to PTC.** Of course the interfaces must be written, but the added flexibility, in complete agreement with the physics, is why one should not use a strict inheritance mechanism in cases where the “is-a” relationship is not complete. Composition methods are more powerful and preferable, simply because they follow the physics. At the very least, one should beware inheritance when it is dictated only by a wish to have automatic polymorphism. Interfaces require some work, but are often more appropriate.

## A.4 ELEMENTP and Polymorphism

Besides the layout and its charts, the other new aspect of PTC is its reliance on TPSA (LBNL version of Berz’s DA-package in particular) for the calculation of Taylor series maps. We just babbled above on the importance of TPSA and polymorphism, in particular, their tremendous impact on a “correct” class structure. These maps are used for all the lattice function calculations through the use of Normal Form theory. PTC uses something called the “Fully Polymorphic Package” or FPP. This set of FORTRAN90 tools creates a polymorphic type that can change from real to Taylor at execution time; it handles complex polymorphs as well, but PTC does not use them directly. We will now explain in general terms how this works within PTC.

### A Simple Example

The first thing is the concept of a real polymorphic variable. This is defined in the FPP package and has little to do with PTC. The reader will notice the following declaration in PSR.f90:

```
TYPE(REAL_8) Y(6)
```

The variable Y(6) is a polymorph. Consider the following piece of code

```
NULLIFY(P)
CALL MOVE_TO(PSR,P,4)
WRITE(6,*) "THE NAME IS ",P%MAG%NAME
```

```
MIS_ROT(:)=0.DO;MIS_ROT(3)=1.D-4;
P=MIS_ROT;
```

```
X(:)=0.DO
CALL TRACK(PSR,X,4,5,DEFAULT)
```

```
WRITE(6,*)"LAYOUT TRACKING :", X(1),X(2)
```

```
CALL ALLOC(Y);
CALL TRACK(PSR,Y,4,5,DEFAULT)
CALL PRINT(Y(1),6);CALL PRINT(Y(2),6)
```

As before, we perform a misalignment of the fourth magnet and track the origin of phase space. This is then followed with an initialization of Y, which fills the array with zeros, and an identical tracking call applied to Y rather than X.

Since Y is not a real variable but something else, we need a special routine to print the result. This piece of code gives us the output

```
THE NAME IS B
LAYOUT TRACKING : 6.180339887489015E-005 5.551115123125783E-017
6.180339887494295E-005
0.000000000000000E+000
```

Certainly the reader may wonder what this is all about. Nothing changed except for the annoying call to ALLOC(Y) and a special print routine. What is going on here? In fact things are even worse, speed tests would even show that on a typical computer architecture, the tracking of Y takes at least about 4 times longer than that of X!

Now, obviously a slightly more complex syntax with a speed reduction is not our goal. Consider instead the following problem. Suppose we would like to know the dependence to first order in the B-field BN(1) of this trajectory. If by miracle it were possible to declare BN(1) as a TPSA parameter, then our task would be over. In this context, we look at this modified piece<sup>14</sup> of code:

```
CALL INIT(1,1,BERZ)
CALL ALLOC(Y);
P%MAGP%BN(1)%KIND=3;P%MAGP%BN(1)%I=1
CALL TRACK(PSR,Y,4,5,+DEFAULT)
CALL PRINT(Y(1),6);CALL PRINT(Y(2),6)
```

The result is:

```
Berz's Package
etall 1, NO = 1, NV = 1, INA = 222
*****
I COEFFICIENT ORDER EXPONENTS
NO = 1 NV = 1
0 0.6180339887489018E-04 0
1 -3.144178731616748 1
-2 0.000000000000000 0
etall 1, NO = 1, NV = 1, INA = 226
*****
I COEFFICIENT ORDER EXPONENTS
NO = 1 NV = 1
1 -2.636803553144651 1
-1 0.000000000000000 0
```

The result if expressed in more standard notation is:

$$\begin{aligned} X(1) &= 0.6180339887489018 \cdot 10^{-4} - 3.144178731616748 \Delta b_1 \\ X(2) &= -2.636803553144651 \Delta b_1 . \end{aligned} \tag{3}$$

What happened here? First the TPSA package of Berz was initialized with degree 1 and with one variable using a standard call to the FPP package; this was done with the call INIT(1,1,BERZ). Secondly the variable BN(1), which is of type REAL\_8 in ELEMENTP, was set to “kind=3.” This is a TPSA knob. A knob is something which is fixed and should not change during a calculation. This knob was indexed with the value 1 by the assignment BN(1)%I=1. Finally the unary operator “+” acted on the internal state DEFAULT: this tells PTC (actually FPP) to take into account the knobs rather than ignoring them.

<sup>14</sup>Again this is not a safe piece of code, but it does work for the exact rectangular bend while being closer to FPP than a safer call to the recommended PTC procedure.



It is clear that this provided us with a sensitivity analysis of the ray with respect to BN(1).

## The Real Power of Polymorphism

The main power behind polymorphism and TPSA is their ability to carry out Normal Form calculations. These kinds of calculations are at the center of all the generalizations of the Courant-Snyder theory to non-linear, non-Hamiltonian and even non-deterministic effects. In particular Normal Form theory applies not only to the equations of motion but to maps of phase space. The Courant-Snyder theory and its Hamiltonian generalizations always deal with the equations of motion, i.e., the infinitesimal generator of a map. This is not what a tracking code produces. In fact it is not even what a typical experiment measures.

Let us go back to the above example and consider the following piece of code:

```
NULLIFY(P)
CALL MOVE_TO(PSR,P,4)
WRITE(6,*) "THE NAME IS ",P%MAG%NAME

MIS_ROT(:)=0.DO;MIS_ROT(3)=1.D-4;
P=MIS_ROT;

X=0.d0; CALL FIND_ORBIT(PSR,X,1,DEFAULT,1.D-7)

CALL INIT(DEFAULT,2,1,BERZ,ND2,NPARA)
CALL ALLOC(Y); CALL ALLOC(NORMAL);
Y=NPARA; Y=X      ! MAKES Y = CLOSED ORBIT + IDENTITY MAP

P%MAGP%BN(1)%KIND=3;P%MAGP%BN(1)%I=NPARA+1
CALL TRACK(PSR,Y,1,+DEFAULT)

NORMAL=Y
WRITE(6,*) NORMAL%TUNE
CALL DAPRINT(NORMAL%DHDJ%V(1),6)
CALL DAPRINT(NORMAL%DHDJ%V(2),6)

CALL KILL(Y)
CALL KILL(NORMAL)
```

We have already seen the meaning of the misalignment commands. In the next line, the code finds the closed orbit in the default state. Incidentally, in this example we used a different call to FIND\_ORBIT that **does not** use TPSA. Therefore it could be moved prior to the line Y=X.

The call INIT(DEFAULT,2,1,BERZ,ND2,NPARA) is most important. It is a call similar to INIT(1,1,BERZ) of the previous example. However this is a subroutine of PTC rather than just a routine of FPP. It will initialize the polymorphic package, including the Normal Form routines, in a way compatible with the state DEFAULT. The degree of the polynomials will be 2 and there will be one system parameter possible. ND2 is the returned number of phase space variables. Here that will be 6. NPARA is the index of the last variable that is not a system parameter (i.e., not a kind=3 of FPP). Here it is also 6. If delta (X(5)) is a parameter<sup>15</sup>, ND2 is 4 and NPARA is then 5. The array Y(6) must start as the identity map around the closed orbit; it is achieved here by equating the fixed point X with Y. Clearly there is a lot of mystery happening here under the overloading of that equal<sup>16</sup> sign and it involves the integer NPARA. The call TRACK(PSR,Y,1,+DEFAULT) produces the one-turn map at position 1 in the DEFAULT state. The unary + activates the parametric dependence on the BN(1) of the fourth magnet.

The next line is NORMAL=Y. NORMAL is of type NORMALFORM. It performs a normal form on the mapping represented by Y. Since there are no cavities and the phase space dimension is 6, the result will be a reduction of the approximate Taylor map into a rotation in the transverse plane and a drift-like map in the longitudinal plane. This means that we get tunes and momentum compaction from this operation. NORMAL contains all sorts of information. We will show in the next example how one extracts and tracks

<sup>15</sup>It is never in PTC an FPP parameter, i.e., kind=3, but it is simply a constant Taylor series of kind=2.

<sup>16</sup>This involves the FPP package as well as PTC; we delay this discussion. See Sect. E.3.1.

the lattice functions using the normal form and the routine TRACK. In the mean time, this is the result of the above code:

```

THE NAME IS B
Berz's Package
      NO      ND      ND2      NP      NDPT      NV
      1      2      4      0      0      4
4.549303376691718E-005
4.951861254696703E-010
5.395960987324300E-016
1.653307271249937E-016
3.221926252210130E-016
Berz's Package
      NO      ND      ND2      NP      NDPT      NV
      2      3      6      1      5      7
0.254100405239239      0.255432440298811      18.8741689756373
ETALL  1, NO = 2, NV = 7, INA = 298
*****
I COEFFICIENT      ORDER  EXPONENTS
NO = 2      NV = 7
0 0.2541004052392394      0 0 0 0 0 0 0
1 -0.9281612961338238      0 0 0 0 1 0 0
1 -0.4893556444627437      0 0 0 0 0 0 1
-3 0.0000000000000000      0 0 0 0 0 0 0
ETALL  2, NO = 2, NV = 7, INA = 299
*****
I COEFFICIENT      ORDER  EXPONENTS
NO = 2      NV = 7
0 0.2554324402988110      0 0 0 0 0 0 0
1 -2.111310651000039      0 0 0 0 1 0 0
1 0.4317083900993576      0 0 0 0 0 0 1
-3 0.0000000000000000      0 0 0 0 0 0 0

```

The last two polynomials represent the tune in the x and y planes. For example, the first polynomial, rounded up in normal notation is

$$\nu_x = 0.2541 - 0.9282 \delta p/p_0 - 0.4894 \Delta b_1 . \quad (4)$$

What is a normal form? A normal form is a statement about the (approximate<sup>17</sup>) topological nature of the flow. Mathematically it often<sup>18</sup> assumes the existence of a transformation  $A$  at position 1 such that the one-turn map from 1 to 1, can be transformed into an amplitude dependent rotation  $R$ :

$$R = A^{-1} \circ M \circ A \quad (5)$$

It is much better to look at this process geometrically as mathematicians and children would do. The tracking of phase space around the fixed point will produce a figure simultaneously in the  $X(1) - X(2)$  and the  $X(3) - X(4)$  planes. In the absence of coupling and nonlinearities, we all know that the usual ellipses will appear in each plane, therefore it is not hard to imagine that there exists a transformation  $A_i$  which will turn the ellipse into a circle in the  $i^{\text{th}}$  plane.

Figure 12 represents an ellipse rotated by 30 degrees and stretched in the x-direction by 2 while shrunk by 1/2 in the vertical direction. The blue area is preserved because  $A$  is a symplectic transformation.

Normal form theory is an attempt to generalize the kind of transformation displayed in the above figure. Such transformations are useful because it is easier in a ring (or linac) to perform calculations in the normalized space. For example ergodic averages and equilibrium averages are all simple averages over the angle

<sup>17</sup>Exact nature if the map is linear!

<sup>18</sup>FPP supports at least four normal forms relevant to accelerators. We do not have the space to explain all of Hamiltonian-free perturbation theory in this manual, nor do we have the will. And if it was not explained to you in the last summer school, well, this is not our fault.

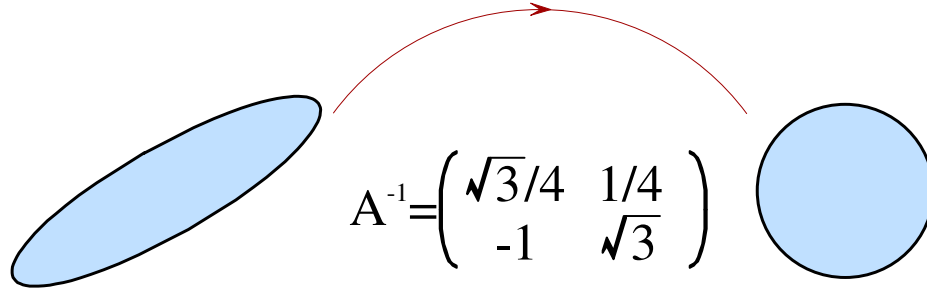


Figure 12: Geometrical Meaning Normal Form

around these circles. The results in the normalized space can then be transformed into results in the actual physical space with the help of the map  $A$ . In fact the relationship between the blue area and the desired quantities in real space define rigorously invariant lattice functions.

The reader should not try to “translate” these results into Hamiltonian theory. Of course the connection is possible and he could consult Forest’s book[5] if he is interested. Rather one should focus directly on the geometric approach and how an approximate Taylor map helps us uncover this transformation  $A$ . Let us see again, on an example, how PTC proceeds in this manner.

```

Y=(NORMAL%A_T+X)

P=>PSR%START
BETAX=(Y(1).SUB.'10')**2+(Y(1).SUB.'01')**2
WRITE(6,*) 1,' ',P%MAG%NAME,BETAX

DO I=1,PSR%N
BETAX=(Y(1).SUB.'10')**2+(Y(1).SUB.'01')**2
WRITE(6,*) I,' ',P%MAG%NAME,BETAX
CALL TRACK(PSR,Y,I,I+1,+DEFAULT)
P=> P%NEXT
ENDDO

BETAX=(Y(1).SUB.'10')**2+(Y(1).SUB.'01')**2
WRITE(6,*) 1,' ',P%MAG%NAME,BETAX

```

The above piece of code follows the normalization statement `NORMAL=Y` of the previous example. The map `NORMAL%A_T` is the canonical transformation  $A$  expressed around the closed orbit  $X$ . It was computed by the assignment `NORMAL=Y`. The polymorphic array is thus initialized as the transformation  $A$  added to the fixed point. Since the FPP package deals normally with Lie/Differential algebraic quantities, the closed orbit<sup>19</sup> is not part of the normalization process and must be added later. The above piece of code is based on the relation between the normal form transformation  $A_i$  at position  $i$ , the one-turn map  $M_i$ , the map  $M_{i+1}$  between  $i$  and  $i+1$ , and finally the one-turn map at  $i+1$ ,  $M_{i+1}$ . Using the following equalities

$$\begin{aligned}
M_i &= A_i \circ R \circ A_i^{-1} \text{ and } M_{i+1} = M_{i+1} \circ M_i \circ M_{i+1}^{-1} \\
&\Downarrow \\
M_{i+1} &= M_{i+1} \circ A_i \circ R \circ A_i^{-1} \circ M_{i+1}^{-1}.
\end{aligned} \tag{6}$$

$$\begin{aligned}
M_i &= A_i \circ R \circ A_i^{-1} \text{ and } M_{i+1} = M_{i+1} \circ M_i \circ M_{i+1}^{-1} \\
&\Downarrow \\
M_{i+1} &= M_{i+1} \circ A_i \circ R \circ A_i^{-1} \circ M_{i+1}^{-1},
\end{aligned} \tag{7}$$

<sup>19</sup>Operations on Taylor series maps are self-consistent only if they are around the closed orbit. In such a case the Lie algebra of truncated operators forms a graded differential algebra. Everything becomes self-consistent. In simple jargon-free terms it means that we must expand our operators and maps around the closed orbit to get self-consistent results. PTC deals with exact tracking and thus it is important to keep track correctly of the closed orbit whatever it may be.

we conclude that the transformation  $B_{i+1}$  given by

$$B_{i+1} = M_{i+1} \circ A_i \quad (8)$$

also normalizes the one-turn map  $M_{i+1}$  at position  $i+1$ . This important result shows that the normalizing transformation  $A_i$  can be tracked, and thus all lattice functions can be tracked. In the above example, we computed a real\*8 BETAX. Thus we obtained the so-called beta function. The results are extracted with the operator “.SUB.” of the FPP package; for example the string  $Y(1).SUB.'10'$  means the coefficient of  $x_1^1$ , thus

$$\beta_x = (Y(1).SUB.'10') **2 + (Y(1).SUB.'01') **2 \equiv A_{11}^2 + A_{12}^2 . \quad (9)$$

The results are

1	D1	6.43760505409452
2	QD	3.77333523226422
3	D2	3.95410307833322
4	B	4.59516372714596
.	.	.
.	.	.
.	.	.
1	D1	6.43760505409440

The reader may wonder how the parameter dependence enters in all of this. Indeed, since the map depends on the seventh TPSA variable, we ought to be able to get BETAX as a polynomial in this variable. This can be done if BETAX is replaced by a REAL\_8 polymorph. In the following piece of code BETX is a polymorph. Notice that the operator .SUB. is replaced<sup>20</sup> by the operator .PAR.:

```
BETX=(y(1).PAR.'1000')**2+(y(1).PAR.'0100')**2
```

The results are

0	6.437605054094520	0	0	0	0	0	0	0
1	-2.019537138724649	0	0	0	0	1	0	0
1	-0.2584365546574744	0	0	0	0	0	0	1
2	0.3059396954101024	0	0	0	0	2	0	0
2	-2.699159311243196	0	0	0	0	1	0	1
2	12.71997024647950	0	0	0	0	0	0	2
-6	0.0000000000000000	0	0	0	0	0	0	0
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
0	6.437605054094401	0	0	0	0	0	0	0
1	-2.019537138724240	0	0	0	0	1	0	0
1	-0.2584365546572856	0	0	0	0	0	0	1
2	230.6165041134295	0	0	0	0	2	0	0
2	128.6268879897636	0	0	0	0	1	0	1
2	17.93909544717098	0	0	0	0	0	0	2
-6	0.0000000000000000	0	0	0	0	0	0	0

We notice that results are periodic to first order in  $X_5$  and  $X_7$  ( $\delta p/p_0$  and  $\Delta b_1$ ). The higher order terms are incorrect because our calculation was only second order in the TPSA variables.

## Phase Advance

Consider the simple diagram of Figure 13. It expresses compactly the concept of normal form and its related little brother “the phase advance.”

We have already discussed the tracking of the canonical transformation  $A$ . However the phase advance is the result of a strict definite “recipe” for the calculation of  $A$ . In the first example, the matrix  $A^{-1}$  was **not** in the “ordinary” Courant-Snyder form, that is to say in the form:

$$A^{-1} = \begin{pmatrix} \frac{1}{\sqrt{\beta}} & 0 \\ \frac{\alpha}{\sqrt{\beta}} & \sqrt{\beta} \end{pmatrix} \quad (10)$$

<sup>20</sup>SUB returns a real(dp) coefficient; PAR could have been used instead of SUB since the syntax REAL\*8=REAL\_8 will simply return the constant part of the polymorph.

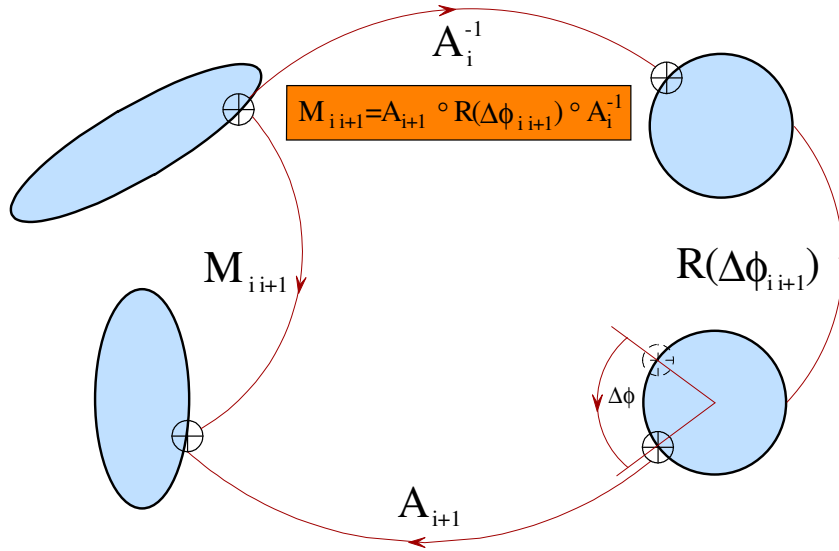


Figure 13: Geometrical Definition of the Phase Advance

The reader can check that a simple rotation can be put in that form. The answer is then

$$\begin{aligned}
 A_{cs}^{-1} &= \begin{pmatrix} \frac{4\sqrt{3}}{7} & -\frac{1}{7} \\ \frac{1}{7} & \frac{4\sqrt{3}}{7} \end{pmatrix} \begin{pmatrix} \frac{\sqrt{3}}{4} & \frac{1}{4} \\ -1 & \sqrt{3} \end{pmatrix} \\
 &= \begin{pmatrix} \frac{4}{7} & 0 \\ -\frac{15\sqrt{3}}{28} & \frac{7}{4} \end{pmatrix}
 \end{aligned} \tag{11}$$

Similarly from the above figure, we can write an equation for the phase advance:

$$\underbrace{M_{i+1} \circ A_i}_{\text{Tracked transformation}} \circ R(\Delta\phi_{i+1}) = \underbrace{A_{i+1}}_{\text{Courant - Snyder}} \tag{12}$$

The phase advance can be viewed as the difference between the tracked transformation and the transformation which depends on the one-turn map through a specific recipe. Generally if

$$\mathcal{R} = A^{-1} \circ M \circ A = B^{-1} \circ M \circ B \tag{13}$$

the product

$$B^{-1} \circ A$$

is also a rotation. In general this product is of the same nature as the normal form, hence in the nonlinear case, it is an amplitude dependent rotation. As a last example, consider the piece of code

```

Y=(NORMAL% A_T+X)

PHASE(:)=0.DO
TESTOLD(:)=0.DO
P=>PSR%START
DO I=1,PSR%N

CALL TRACK(PSR,Y,I,I+1,+DEFAULT)

TEST=DATAN2((Y(1).SUB.'01'),(Y(1).SUB.'10'))/TWOPI
IF(TEST<0.DO.AND.DABS(TEST)>1.E-10)TEST=TEST+1.DO
DPH=TEST-TESTOLD(1)
IF(DPH<0.DO)DPH=DPH+1.DO
PHASE(1)=PHASE(1)+DPH

```

```

TESTOLD(1)=TEST

TEST=DATAN2((Y(3).SUB.'0001'),(Y(3).SUB.'0010'))/TWOPI
IF(TEST<0.DO.AND.DABS(TEST)>1.E-10)TEST=TEST+1.DO
DPH=TEST-TESTOLD(2)
IF(DPH<0.DO)DPH=DPH+1.DO
PHASE(2)=PHASE(2)+DPH
TESTOLD(2)=TEST
P=>P%NEXT
ENDDO

WRITE(6,*) " Total Phase Advances ", PHASE(1),PHASE(2)

```

The above code computes the angle necessary to rotate the tracked normalizing transformation into the standard Courant-Snyder<sup>21</sup> transformation. The result is simply

```

Total Phase Advances      2.25410040523924      2.25543244029881

```

**The geometrical interpretation of the normal form is completely encapsulated in Figure 13. Quite often this interpretation looks “non rigorous” or “unphysical” to accelerator physicists. Actually this is both the most rigorous way to look at a normal form as well as the most physical way. It is the most rigorous because, the normal form when truly achievable, is a statement on the intrinsic (local) topology of the motion, i.e., of the manifold on which the motion resides. It is also the most physical because it is detached from any equation of motion. Recent experiment at the ALS concerning the diffusion of particles across resonances measure directly the failure of the normal form to exist. These techniques, initiated by Laskar in solar system dynamics, do not rely on any particular equation of motion. Accelerator physicists unfortunately are very much “equation prone” and feel comfortable if concepts are defined in terms of equations. This is why some of the ideas connected to PTC, namely “Hamiltonian-free” perturbation theory are extremely difficult to communicate; without them the entire PTC edifice falls flat because we are dragged back to the continuous/smooth  $s$ -dependent perturbation theory where accelerator theorists feel at home. Obviously this is not compatible with the fibre bundle structure of PTC.**

## B Non Trivial Examples

In this section we create directly in FORTRAN90 some lattices which are not possible in a standard tracking code without cumbersome logic linking magnet clones. We will first examine two versions of the PSR glued by the “waist.” We will later turn these two layouts into a single “figure-8” layout.

### B.1 Example 1: Two Siamese Rings

It is best to look at the picture of the beastly lattices, in Figure 14, before trying to discuss their creation. One sees a common region that consists of (D2,QF,D1,D1,QD,D2). The only real physical magnets are QF and QD. These are the two grossly misaligned elements in the figure.

The structure of the main program is very simple:

```

DIR=1 ; CALL MAKE_PSR(PSR(1),DIR)
DIR=-1; CALL MAKE_PSR(PSR_REV,DIR)
CALL SURVEY(PSR(1)); CALL SURVEY(PSR_REV);

OMEGA=0.DO;A=0.DO; A(2)=TWOPI/2.DO;
CALL ROTATION(PSR_REV,OMEGA,A)

CALL MAKE_SIAMESE(PSR(1),PSR_REV,PSR(2))

```

In this PTC example we pretty much follow the typical procedure of constructing a non-trivial entity from MAD-like beam lines. Of course, a powerful code should be able to generate directly through its input the

<sup>21</sup>With coupling, this algorithm would give the “Teng-Edwards” phase advance.

double monstrosity. But, for the time being, what follows provide a simple example of the creation of a non-trivial topology from pasting together MAD-like standard lines.

Here the first two calls create two separate rings with no common elements. The subroutine MAKE\_PSR is pretty much the main program we saw previously. Here it is again, reduced to a minimum (notice that MADKIND2=DRIFT\_KICK\_DRIFT is a default to the Drift-Kick-Drift model and thus can be omitted):

```

SUBROUTINE MAKE_PSR(PSR,DIR)
USE MAD_LIKE
IMPLICIT NONE
TYPE(LAYOUT) PSR
TYPE(FIBRE) D1,QD,QF,D2,B
REAL(DP) X(6),KF,KD,ANG,BRHO,MIS_ROT(6)
INTEGER DIR

FIBRE_DIR=DIR ; FIBRE_FLIP = .TRUE. ;

ANG=(TWOPI*36.D0/360.D0) ; BRHO=1.2D0*(2.54948D0/ANG) ;
CALL SET_MAD(BRHO=BRHO,METHOD=6,STEP=10)

KF=2.72D0/BRHO ; KD=-1.92D0/BRHO ;
D1 = DRIFT("D1",2.28646D+00) ; D2 = DRIFT("D2",0.45D+00) ;
QF = QUADRUPOLE("QF",0.5D0,KF) ; QD = QUADRUPOLE("QD",0.5D0,KD) ;
B = RBEND("B", 2.54948D0,ANG) ;

PSR=.RING.(10*(D1+QD+D2+B+D2+QF+D1))

CALL CLEAN_UP
END SUBROUTINE MAKE_PSR

```

The integer DIR is set to either 1 or -1. Generally, if set to -1, the MAD-like input of PTC will also invert B-fields so as to ensure closure if FIBRE\_FLIP is true. This is done in this run.

Going back to the main program, we notice that PSR(1) and PSR\_REV are created. PSR(1) is the normal forward lattice and PSR\_REV is a carbon copy but made of reversed propagators. At this stage the two beam lines are completely independent. In the next step, since we have no CAD input, we simply proceed with a standard MAD-like survey using the commands CALL SURVEY(PSR(1)) and CALL SURVEY(PSR\_REV). Using this particular technique, the two rings will in fact be right on top of each other. This is not exactly what we want. To correct this problem we actually rotate the PSR\_REV by 180 degrees around its origin which happens to be the origin of space. This is done by the commands:

```

OMEGA=0.D0 ; A=0.D0 ; A(2)=TWOPI/2.D0 ;
CALL ROTATION(PSR_REV,OMEGA,A)

```

If plotted, the two rings would appear like the final “Siamese” system. However the common area would be made of physically different QF and QDs unphysically superimposed on one another.

The next step consists in using the existing rings, PSR(1) and PSR\_REV, to create the new ring PSR(2) to replace PSR\_REV. Let us look in detail into the routine “MAKE\_SIAMESE” which will perform this surgery.

```

SUBROUTINE MAKE_SIAMESE(PSR,PSR_REV,TWIN)      ! PSR here is PSR(1) and TWIN will be PSR(2)
USE MAD_LIKE
IMPLICIT NONE
TYPE(LAYOUT) PSR,PSR_REV,TWIN ;
TYPE(FIBRE), POINTER :: P,PN
INTEGER I

TWIN=0 ; CALL SET_UP(TWIN) ;                  ! (1)

NULLIFY(P,PN) ; P=>PSR%START ; PN=>PSR_REV%START ; ! (2)

DO I=1,3                                     !

```

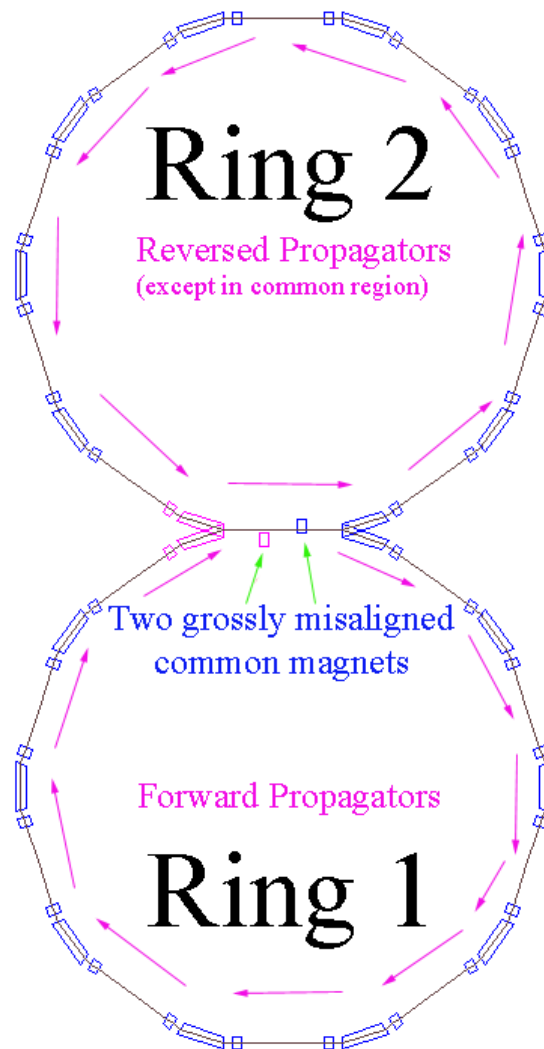


Figure 14: Two Rings joined like Siamese Twins

```

CALL APPEND_POINT(TWIN,P)           !(3)
P=>P%NEXT; PN=>PN%NEXT;             !
ENDDO                               !

DO I=4,PSR%N-3                      !
CALL APPEND(TWIN,PN)                 !(4)
P=>P%NEXT;PN=>PN%NEXT;               !
ENDDO

DO I=1,3                             !
CALL APPEND_POINT(TWIN,P)           !(5)
P=>P%NEXT;PN=>PN%NEXT;               !
ENDDO

TWIN%CLOSED=.TRUE. ; CALL RING_L(TWIN,.TRUE.);  !(6)

NULLIFY(P) ;P=>TWIN%START            !(7)

DO I=1,3; P=>P%NEXT; ENDDO;          !(8)

CALL FIND_PATCH(P%PREVIOUS,P,NEXT=.TRUE.)  !(9)

```



```

DO I=4,PSR%N-3; P=>P%NEXT; ENDDO; !(10)

CALL FIND_PATCH(P%PREVIOUS,P,NEXT=.FALSE.) !(11)

END SUBROUTINE MAKE_SIAMESE

```

Let us go one by one through the above routine keeping in mind the topology of the Siamese rings. The routine MAKE\_SIAMESE will transfer the MAD-like independent beam line PSR\_REV into a new layout called TWIN. The only difference will be that the fibres of the “jointed” region will contain data belonging to PSR rather than clones as they do now. Most codes, at present, can only handle clones, that is to say, copies of a magnet. This is inappropriate for common beam lines or recirculators. In a common beam line, the common magnets are the same entities, not clones (identical twins) which can theoretically be powered on their own. The entire virtue of the fibre construct is to ensure our ability on the computer to create true common structures rather than clones with horrible logic mimicking the true system. This is the case, unfortunately, of all present design codes.

We will now describe line by line what PTC is doing in this routine:

1. Here, TWIN is nullified and then initialized. TWIN is a dummy variable; it is used to pass PSR(2) from the main program.
2. The pointers P and PN are set at the beginning of the two existing lattices.
3. The first 3 elements of TWIN must belong to ring PSR. Therefore the fibre are created and the data is pointed to rather than cloned. This is done by the routine APPEND\_POINT which was first written for recirculators.
4. We now clone around the lattice until we reach the last 3 elements. This operation is similar to the duplication of a beam line in traditional codes.
5. The operations of item 3 are repeated on the last 3 elements.
6. The new ring TWIN is now closed rather than remaining a terminated beam line.
7. The pointer P is set at the beginning of TWIN.
8. We move to the fourth element which belongs only to TWIN.
9. A patch is computed between element 3 and 4. The patch is set on element 4 with the keyword “NEXT=.TRUE.”.
10. We again travel down the cloned part of the lattice.
11. A patch is computed between last cloned element and the following pointed element. The patch is put on the pointed element with the keyword “NEXT=.FALSE.”.

The reader may wonder what the patch is doing since both rings have the same energy and the elements are perfectly lined up. Actually the  $x$  and  $z$  coordinates of the backward and forward propagators are defined differently. A rotation of 180 degrees around the  $y$ -axis is necessary to glue a forward and backward fibre. In our example, the only effect of the patch was the rotation by 180 degrees.

## B.2 Example 2: Making a Figure “8”

The following horror is only a slight modification of the previous example. Here rather than having two rings sharing a common section, we turn the entire thing into a single object with the topology of a “figure 8.” It is still a closed ring but with a common section: a big dog bone.

We do not need to describe the making of this object in great detail since it is similar to the previous example. Here the original standard lattices PSR and PSR\_REV can be exterminated on exit. Ideally, in the future version of MAD-X, these rings would have never existed in the first place. The sole survivor is the final layout FIGURE\_8.

```

SUBROUTINE MAKE_FIGURE_8(PSR,PSR_REV,FIGURE_8)
USE MAD_LIKE
IMPLICIT NONE
TYPE(LAYOUT) PSR,PSR_REV,FIGURE_8
TYPE(FIBRE), POINTER :: P,PN
INTEGER I

FIGURE_8=0; CALL SET_UP(FIGURE_8);

NULLIFY(P); P=>PSR%END
DO I=1,2
P=>P%PREVIOUS
ENDDO
WRITE(6,*) P%PREVIOUS%MAG%NAME,P%MAG%NAME

DO I=1,PSR%N
CALL APPEND(FIGURE_8,P)
P=>P%NEXT;
ENDDO

NULLIFY(P); P=>FIGURE_8%START
DO I=1,6
WRITE(6,*) P%MAG%NAME
CALL APPEND_POINT(FIGURE_8,P)
P=>P%NEXT;
ENDDO

NULLIFY(P); P=>PSR_REV%START;
DO I=1,3
P=>P%NEXT;
ENDDO
WRITE(6,*) P%MAG%NAME

DO I=4,PSR%N-3
CALL APPEND(FIGURE_8,P)
P=>P%NEXT;
ENDDO

FIGURE_8%CLOSED=.TRUE. ; CALL RING_L(FIGURE_8,.TRUE.);

P=>FIGURE_8%START
PN=>P%NEXT
DO I=1,FIGURE_8%N
IF(P%DIR/=PN%DIR) CALL FIND_PATCH(P,PN,NEXT=.TRUE.)
P=>P%NEXT
PN=>PN%NEXT
ENDDO

END SUBROUTINE MAKE_FIGURE_8

```

This concludes our set of crazy examples.

## 2 FILE BY FILE DESCRIPTION

We will list the standard commands more or less on a file by file basis. The reader is invited to look at the original routines.

## C A few aspects of FPP and PTC

We will describe a few types and a few conventions of FPP and PTC used in this document.

### C.1 ELEMENT and ELEMENTP: EL and ELP

The ELEMENT and the ELEMENTP are actually defined in module S\_DEF\_ELEMENT. Generally we will use EL for a generic ELEMENT and ELP for a generic polymorphic element. We give here the polymorphic ELEMENTP for reference. In the definition of ELEMENT, polymorphic entities are replaced by their real\*8 equivalent, for example real\_8 becomes real\*8 and TYPE(DRIFT1P) becomes TYPE(DRIFT1). Only the logical knob is peculiar to ELEMENTP; all the rest can be found in type ELEMENT as well. Notice that in PTC, double precisions are defined as real(dp). “dp” is itself defined in a module containing constants.

```
TYPE ELEMENTP
  INTEGER, POINTER :: KIND ! WHAT IT IS
  LOGICAL, POINTER :: KNOB ! FALSE IF NO KNOB
  CHARACTER(16), POINTER :: NAME,VORNAME ! IDENTIFICATION NAME AND FIRST NAME
  LOGICAL, POINTER :: PERMFRINGE
  !
  !
  !
  TYPE(REAL_8), POINTER :: L ! LENGTH OF INTEGRATION OFTEN SAME AS LD, CAN BE ZERO
  TYPE(REAL_8), DIMENSION(:), POINTER :: AN,BN !MULTIPOLE COMPONENT
  TYPE(REAL_8), POINTER:: FINT,HGAP !FRINGE FUDGE FOR MAD
  TYPE(REAL_8), POINTER:: H1,H2 !BOUNDARY FUDGE FOR MAD
  !
  TYPE(REAL_8), POINTER :: VOLT, FREQ,PHAS ! CAVITY INFORMATION
  REAL(DP), POINTER :: DELTA_E ! CAVITY ENERGY GAIN
  !
  TYPE(REAL_8), POINTER :: B_SOL
  LOGICAL, POINTER :: THIN

  ! MISALIGNEMENTS AND ROTATION
  LOGICAL, POINTER :: MIS,EXACTMIS
  REAL(DP), DIMENSION(:), POINTER :: D,R

  TYPE(MAGNET_CHART), POINTER :: P

  ! TYPES OF POLYMORPHIC MAGNETS
  TYPE(FITTED_MAGNETP), POINTER :: BEND ! MACHIDA'S FITTED MAGNET
  TYPE(DRIFT1P), POINTER :: D0 ! DRIFT
  TYPE(DKD2P), POINTER :: K2 ! INTEGRATOR
  TYPE(KICKT3P), POINTER :: K3 ! THIN KICK
  TYPE(CAV4P), POINTER :: C4 ! DRIFT
  TYPE(SOL5P), POINTER :: S5 ! CAVITY
  TYPE(KTKP), POINTER :: T6 ! INTEGRATOR SIXTRACK STYLE
  TYPE(TKTFP), POINTER :: T7 ! INTEGRATOR THICK FAST
  TYPE(NSMIP), POINTER :: S8 ! NORMAL SMI
  TYPE(SSMIP), POINTER :: S9 ! SKEW SMI
  TYPE(TEAPOTP), POINTER :: TP10 ! SECTOR BEND WITH CYLINDRICAL GEOMETRY
  TYPE(MONP), POINTER :: MON14 ! MONITOR OR INSTRUMENT
  TYPE(ESEPTUMP), POINTER :: SEP15 ! ELECTROSTATIC SEPARATOR
  TYPE(STREXP), POINTER :: K16 ! EXACT STRAIGHT INTEGRATOR
  TYPE(SOLTP), POINTER :: S17 ! SOLENOID SIXTRACK STYLE AS IN TYPE(KTKP)
  TYPE(USER1P), POINTER :: U1 ! USER DEFINED
  TYPE(USER2P), POINTER :: U2 ! USER DEFINED
```

```
END TYPE ELEMENTP
```

Here type MAGNET\_CHART contains mostly information about the internal chart of a magnet:

```
TYPE MAGNET_CHART
  TYPE(MAGNET_FRAME), POINTER :: F
  INTEGER, POINTER :: CHARGE ! PROPAGATOR
  INTEGER, POINTER :: DIR ! PROPAGATOR
  REAL(DP), POINTER :: LD,BO,LC ! REAL(DP), POINTER :: TILTD ! INTERNAL FRAME
  REAL(DP), POINTER :: BETA0,GAMMA0I,GAMBET,POC
  REAL(DP), DIMENSION(:), POINTER :: EDGE ! INTERNAL FRAME
  ! INTEGER, POINTER :: TOTALPATH !
  LOGICAL, POINTER :: EXACT,RADIATION,NOCAVITY ! STATE
  LOGICAL, POINTER :: FRINGE,TIME !
  !
  INTEGER, POINTER :: METHOD,NST ! METHOD OF INTEGRATION 2,4,OR 6 YOSHIDA
  INTEGER, POINTER :: NMUL ! NUMBER OF MULTIPOLE
END TYPE MAGNET_CHART
```

Please notice the appearance of type MAGNET\_FRAME within MAGNET\_CHART. It is defined as:

```
TYPE MAGNET_FRAME
  REAL(DP), POINTER, DIMENSION(:, :):: ENT
  REAL(DP), POINTER, DIMENSION(:) :: A
  REAL(DP), POINTER, DIMENSION(:, :):: EXI
  REAL(DP), POINTER, DIMENSION(:) :: B
  REAL(DP), POINTER, DIMENSION(:, :):: MID
  REAL(DP), POINTER, DIMENSION(:) :: O
END TYPE MAGNET_FRAME
```

The frames of reference in MAGNET\_FRAME are normally equal to that of the fibre in which the magnet sits. However, as one misaligns a magnet, MAGNET\_FRAME contains the true position of the magnet in space.

## C.2 Real\*8 array X(6)

Generally we will use X to designate a real\*8 array. Thus X means

```
REAL(DP) X(6)
```

## C.3 Real Polymorph Array Y(6)

Generally we will use Y to designate a polymorphic array. Thus Y means

```
TYPE(REAL_8) Y(6)
```

## C.4 Beam Envelope Array YS(6)

Generally we will use YS to designate a beam envelope array. Thus YS means

```
TYPE(ENV_8) YS(6)
```

It is defined in the FPP package as

```
TYPE ENV_8
  TYPE(REAL_8) V
  TYPE(REAL_8) E(NDIM2)
  TYPE(REAL_8) SIGMA0(NDIM2)
  TYPE(REAL_8) SIGMAF(NDIM2)
END TYPE ENV_8
```

**Routines involving the beam envelopes now support parameter dependence. However this is a tricky topic because the theory is only linear; therefore the parameter dependent fixed point with classical radiation must be provided to PTC prior to calculation. So beware: we hope to document these subtleties eventually. See Sect. R.2.5.**

The code includes stochastic effects only from elements whose ideal bending angle (EL%P%B0) is non zero. This is to ensure a self-consistent result when parameter dependence is turned on. This can be relaxed by setting the global variable STOCH\_IN\_REC to true.

#### C.4.1 Definition of Beam Envelope

The beam envelope YS in PTC is a collection of quadratic moments and the linear map. Theoretically one can write a one-turn map for the moments:

$$\begin{aligned} \langle x_a x_b \rangle^{final} &= \underbrace{\sum_{i,j} M_{ai} M_{bj} \langle x_i x_j \rangle}_{\text{Deterministic}} + \underbrace{B_{ab}}_{\text{Stochastic}} \\ &= \underbrace{\sum_{n=1}^{\infty} M_{ai} \{ \langle x_i x_j \rangle + B_{ij}^0 \} M_{bj}}_{\text{Form in PTC}} . \end{aligned} \quad (14)$$

The map  $M$  is stored implicitly in the polymorph YS(6)%V. The stochastic kick  $B_{ab}^0$ , added to the initial envelope if any, IS located in YS(a)%E(b). The arrays YS(a)%SIGMA0(b) and YS(a)%SIGMAF(b) contain the initial and final tracked envelopes respectively. The equilibrium beam is obtained by solving the equation

$$\langle x_a x_b \rangle^{\infty} = \sum_{i,j} M_{ai} \{ \langle x_i x_j \rangle^{\infty} + B_{ij}^0 \} M_{bj} . \quad (15)$$

This equation is solved by normal form techniques.

#### C.4.2 Normalizing the Beam Envelope in PTC

The normal form corresponding to type ENV\_8 is type BEAMENVELOPE of FPP.

```

TYPE BEAMENVELOPE          ! RADIATION NORMALIZATION
TYPE (DAMAP) TRANSPOSE     ! TRANSPOSE OF MAP WHICH ACTS ON POLYNOMIALS
TYPE (TAYLOR) BIJ          ! REPRESENTS THE STOCHASTIC KICK AT THE END OF THE TURN
                           ! ENV_F=M ENV_F M^T + B
TYPE (PBRESONANCE) BIJNR  ! EQUILIBRIUM BEAM SIZES IN RESONANCE BASIS
TYPE (TAYLOR) SIJ0        ! EQUILIBRIUM BEAM SIZES
REAL(DP) EMITTANCE(3),TUNE(3),DAMPING(3)
LOGICAL AUTO,STOCHASTIC
REAL(DP) KICK(3)
TYPE (DAMAP) STOCH
END TYPE BEAMENVELOPE

```

Therefore one can produce the equilibrium beam envelope with the syntax

```
ENV=YS
```

where ENV is of type BEAMENVELOPE. The equilibrium beam sizes, an exact linear concept, are in ENV%SIJ0. The equilibrium emittances are in ENV%EMITTANCE. These are not exact linear concepts. They depend critically on the smallness of the damping decrements compared to the tunes and all the linear resonances. The calculation of the equilibrium emittances in PTC uses a projection of the operator  $B_{ab}$  along the eigen-directions of the map  $M$ . This can be shown to be equivalent to the Chao[9] theory of radiation—a better theory than Sands but nevertheless not as good as the beam envelope approach.

The normalization routine can also produce the information necessary to produce the stochastic kicks of a Monte Carlo calculation. This could be useful in brute force simulations. That information is stored in ENV%KICK and ENV%STOCH. It is triggered by the logical ENV%STOCHASTIC.

## C.5 The Type DAMAP: M

We will use M to denote a generic DAMAP. The DAMAP is defined as

```
TYPE DAMAP
  TYPE (TAYLOR) V(NDIM2)
END TYPE DAMAP
```

The DAMAP is a collection of 2, 4, or 6 Taylor series. Many operations are permitted on the DAMAP. These operations are part of the FPP package, not of PTC per se. However there is one type of convenient initialization operation permitted in PTC which involves a polymorphic array Y(6), a DAMAP M and an orbit X(6) which is generally the closed orbit:

```
Y=NPARA
Y=X+M
```

Here the polymorphic array Y is prepared as before with Y=NPARA where NPARA is the special integer computed by the INIT routine. We remind the reader that NPARA+1 is the location of the first system parameter of TPSA, i.e., of something that is not part of X(6). The DAMAP M is added to the fixed point and Y(6) is properly initialized. Notice that M can be a 4 or 6 dimensional map in PTC. Generally this syntax is used if M is actually a canonical transformation normalizing the map (in a ring) or defining the distribution (in a linac).

## C.6 The Type NORMALFORM

We will use NORMAL in this manual. The concept of normal forms is central to ring perturbation theory. A normal form is triggered by the assignment:

```
NORMAL = M
```

However it is also possible for convenience to write

```
NORMAL = Y
```

The basic idea of a normal form is to rewrite a map, around its fixed point, i.e., a DAMAP, as

$$M = A \circ R \circ A^{-1} . \quad (16)$$

Basically an object of type NORMALFORM contains A, R as well as other useful goodies. NORMALFORM is defined as

```
TYPE NORMALFORM
  TYPE (DAMAP) A_T
  TYPE (DAMAP) A1
  TYPE (REVERSEDRACTFINN) A
  TYPE (DRACTFINN) NORMAL
  TYPE (DAMAP) DHDJ
  REAL(8) TUNE(NDIM), DAMPING(NDIM)
  INTEGER NORD, JTUNE
  INTEGER NRES, M(NDIM, NRESO), PLANE(NDIM)
  LOGICAL AUTO
END TYPE NORMALFORM
```

NORMAL%A\_T and NORMAL%NORMAL contain A and R respectively. Other things are described in our presentation on FPP which can be found in the Appendix B.

([http://bc1.lbl.gov/CBP\\_pages/educational/TPSA\\_DA/Introduction.html](http://bc1.lbl.gov/CBP_pages/educational/TPSA_DA/Introduction.html))

## C.7 The Type UNIVERSAL\_TAYLOR

The type UNIVERSAL\_TAYLOR is a convenient type for storing permanently some important Taylor series. This can be used in lieu of printing the series on a file. This was suggested by David Sagan of Cornell.

```

TYPE UNIVERSAL_TAYLOR
  INTEGER, POINTER:: N,NV      ! Number of coefficients and number of variables
  REAL(DP), POINTER :: C(:)    ! Coefficients C(N)
  INTEGER, POINTER:: J(:, :)   ! Exponents of each coefficient J(N,NV)
END TYPE UNIVERSAL_TAYLOR

```

The following syntax is permissible:

```

TYPE (UNIVERSAL_TAYLOR) U
TYPE (TAYLOR) T1,T2
.
.
.
U=0      ! NULLIFIES
U=T1
.
.
.
T2=U
U=-1    ! DESTROYS U

```

In the above example, the Taylor series T1 is stored in U and resurrected later in T2. Of course, calls to INIT are permissible between these two lines: that is the entire raison d'être of the UNIVERSAL\_TAYLOR type.

## C.8 The Module Precision\_Constants

This module contains the whole pile of constants used by FPP and PTC including two Ruth/Yoshida sets of constants that defined the fourth and sixth order symplectic integrators of PTC. FD1, FD2, FK1, and FK2 are constants related to the fourth order Ruth-Yoshida integrator. YOSK(0:4) and YOSD(4) are used in the sixth order Yoshida scheme.

It also contains some routines that are used to access the console (unit=5 or 6). This is done so that neither FPP nor PTC calls the console directly. This might become important if these codes are to be linked to a Windows style application.

## C.9 The Module File\_Handler

The file a\_scratch\_size.f90 contains three modules. Besides the module scratch\_size that defines the sizes of the scratch variables of FPP and Precision\_Constants defined above, this file contains the module File\_Handler. This module is a convenient file handling module in case one would like to open and close files. Failure to use this facility may lead to conflict between a user's own file and the files that PTC may open. The syntax is rather easy.

```

MF=NEWFILE
OPEN(UNIT=MF, FILE='STOCH.TXT', STATUS='UNKNOWN')
.
.
.
MF=CLOSEFILE

```

MF is an integer which will contain the file unit number. MF=CLOSEFILE will close the unit MF and return its negative. At this points units from 20 to 99 are used by File\_Handler; it should be enough for most usage. We should add that for safety, the following initialization should be done:

```

NEWFILE%MF=.FALSE.
CLOSEFILE%MF=.FALSE.

```

In PTC this is done in the compulsory routine MAKE\_STATES, thus the user need not worry. It is also possible for the file handler to reserve certain files for GUI applications. For example, the popular GUI WINTERACTER commercial package uses units 40,41, and 42. Thus File\_handler never uses these files. This is done in the array "winterfile" which can be modified to accommodate other reserved unit.

## D Sa\_ROTATION\_MIS.f90: The Module Rotation\_mis

The module Rotation\_Mis manipulates the  $R^3$  representation of the Euclidean group, in other words the usual one acting on geometrical figures.

### D.1 Basic Description of the Module

In Sect. A.2.2, we introduced the techniques for making a magnet into a thin compressed element. These were used directly in the first versions of PTC. At that time only the dynamical versions of the Euclidean operators were used. The purpose of the module Rotation\_mis is to rewrite these operators in  $R^3$  and then use the local isomorphism between the dynamical and  $R^3$  operators for near-identity maps. Therefore let us start with what used to precede the call to TRACK(C%MAG,X) in the old PTC version:

```
CALL ROT_XZ( C%CHART%ALPHA/2.DO ,X, C%MAG%P%BETA0,OUR,C%MAG%P%TIME)
CALL TRANSZ( C%CHART%L/2.DO ,X, C%MAG%P%BETA0,OUR,C%MAG%P%TIME)
CALL ROT_YZ(C%MAG%R(1),X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)
CALL ROT_XZ(C%MAG%R(2),X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)
CALL ROT_XY(C%MAG%R(3),X,OU)
CALL TRANS(C%MAG%D,X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)
CALL TRANSZ( -C%CHART%L/2.DO ,X, C%MAG%P%BETA0,OUR,C%MAG%P%TIME)
CALL ROT_XZ( -C%CHART%ALPHA/2.DO ,X, C%MAG%P%BETA0,OUR,C%MAG%P%TIME)
```

In Lie map order, these calls correspond to the map:

$$\begin{aligned} \mathcal{T}_{ent} = & \exp\left(: -\frac{\alpha}{2}L_y :\right) \exp\left(: \frac{L}{2}p_z :\right) \\ & \times \exp\left(: r_x L_x :\right) \exp\left(: -r_y L_y :\right) \exp\left(: r_z L_z :\right) \exp\left(: \vec{d} \cdot \vec{p} :\right) \\ & \times \exp\left(- : \frac{L}{2}p_z :\right) \exp\left(: \frac{\alpha}{2}L_y :\right) . \end{aligned} \quad (17)$$

In the obsolete versions of PTC, the operators used in Equation (17) are the dynamical operators of Equation (26). The purpose of the module Rotation\_mis is to manipulate sequences of dynamical operations such as those of Equation (17) into a standard factorized order. The manipulations are done in the ordinary  $R^3$  Euclidean group and then the result is transferred to dynamical operators. Therefore the old sequence of above is replaced by a standardized sequence

```
CALL ROT_YZ(C%CHART%ANG_IN(1),X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)
CALL ROT_XZ(C%CHART%ANG_IN(2),X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)
CALL ROT_XY(C%CHART%ANG_IN(3),X,OU)
CALL TRANS(C%CHART%D_IN,X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)
```

Thus PTC now ignores the dynamical expanded calls and instead use a standard ordering:

$$\mathcal{T}_{ent} = \exp\left(: \beta_x^{in} L_x :\right) \exp\left(: -\beta_y^{in} L_y :\right) \exp\left(: \beta_z^{in} L_z :\right) \exp\left(: \vec{e}^{in} \cdot \vec{p} :\right) . \quad (18)$$

The passage between Equations (17) and (18) is done entirely in  $R^3$ . For that purpose a new type in Rotation\_mis is created. It is the type matrix\_PTC:

```
TYPE MATRIX_PTC
REAL(DP) R(D,D)
REAL(DP) T(D)
END TYPE
```

This type represents the following map:

$$T(\vec{x}) = R\vec{x} + \vec{d} . \quad (19)$$

And of course the composition obeys the usual rules:

$$(T_2 \circ T_1)(\vec{x}) = R_2 R_1 \vec{x} + R_2 \vec{d}_1 + \vec{d}_2 . \quad (20)$$



The Lie operators associated to these objects operate in the reverse order:

$$T_2 \circ T_1 = T_1 T_2. \quad (21)$$

Amongst a whole slew of routines, PTC provides a routine for the factorization of any operator of type `matrix_PTC` into a standard product, namely:

$$\begin{aligned} T(\vec{x}) &= R_3 R_2 R_1 \vec{x} + \vec{e} \\ \text{where } R_i &= \exp(\beta_i L_i) . \end{aligned} \quad (22)$$

The factorization in Equation (22) corresponds to the Lie ordering of Equation (18) and thus to PTC standard ordering. Thus we can identify the quantities  $\beta_i$  and the vector  $\vec{e}$  with one another.

When a fibre is misaligned, PTC calls the routine `FACTORIZE_ROTATION` of the module `Rotation_mis`. The inputs are the standard misalignments of the magnet. Then it computes  $(\vec{\beta}^{in}, \vec{e}^{in})$  and  $(\vec{\beta}^{out}, \vec{e}^{out})$  for the entrance and exit operator respectively.

```
CALL FACTORIZE_ROTATION(S1,S2%CHART%L,S2%CHART%ALPHA,S2%CHART%D_IN,    &
& S2%CHART%ANG_IN,S2%CHART%D_OUT,S2%CHART%ANG_OUT)
CALL ADJUST_INTERNAL(S2)
```

These new misalignments are properties of the chart and not of the magnet since they depend on the layout length `CHART%L` and the layout angle `CHART%ALPHA`.

Obviously there are a lot of operators and tools inside module `Rotation_mis`. Undoubtedly more could, and will be, added. The user could, for example, redefine the meaning of `MAG%R(3)` and `MAG%D(3)` to correspond to some other parameterization of the Euclidean group: Euler's angles are not unique. In that case, he could actually change PTC, or perhaps better, write a (`Rotation_mis`) routine which preprocesses his definition into the `MAG%R(3)` and the `MAG%D(3)` of present PTC. In addition the routines in this module can be used for complex patching and survey adjustments.

Finally, because the Euclidean group is global in  $R^3$  and because the misalignments lead to maps near the identity, the `FACTORIZE_ROTATION` call works well on  $180^\circ$  fibres as well. In other words the excluded phase space is always minimal. (See Sect. G.4.4 on the topic of "decompression.")

## D.2 Operations on Type `Matrix_PTC`

Let us use the notation `M` for a `Matrix_PTC` object and  $r$  for a real number. `Li` ( $i=1,2,3$ ) represents the generator of rotations in `SO(3)`. `Vi` ( $i=1,2,3$ ) represents an array of real numbers often multiplying the generators of rotation.

```
INTERFACE ASSIGNMENT (=)
  MODULE PROCEDURE EQUAL      ! M2=M1
  MODULE PROCEDURE EQUALINT ! M2=0 (Zeroes M), M=-1,-2,OR -3 (Creates L1,2,3), M=1 Makes an identity matrix
  MODULE PROCEDURE EQUALD    ! M2=r1 makes a translation of length r in the 3rd direction ( a drift);
END INTERFACE

INTERFACE OPERATOR (*)
  MODULE PROCEDURE MUL      ! M3=M2*M1
  MODULE PROCEDURE MULR    ! M3=r1*M2
  MODULE PROCEDURE RMUL    ! M3=M1*r2
END INTERFACE

INTERFACE OPERATOR (**)
  MODULE PROCEDURE POW      ! M3=M1**n2 but only n=-1 is accepted (inverse)
END INTERFACE

INTERFACE OPERATOR (/)
  MODULE PROCEDURE DIVR    ! M3=M1%/r2 ( Translation part set to zero)
END INTERFACE

INTERFACE OPERATOR (-)
  MODULE PROCEDURE SUB      ! M3=M2-M1 (Matrix part only ; Translation part set to zero)
  MODULE PROCEDURE USUB    ! M2=-M1 (Matrix part only ; Translation part set to zero)
END INTERFACE

INTERFACE OPERATOR (+)
  MODULE PROCEDURE ADD      ! M3=M2+M1 (Matrix part only ; Translation part set to zero)
  MODULE PROCEDURE UADD    ! M2=M1
END INTERFACE

INTERFACE EXP
  MODULE PROCEDURE EXPMAT  ! M2=exp(M1)
```

```

MODULE PROCEDURE EXPVEC      ! M2=exp(V1_i L_i)
MODULE PROCEDURE EXPVEC2    ! M3=exp(-V2_1 L_1) exp(-V2_2 L_2)exp(-V2_3 L_3)exp(V1_i L_i)
MODULE PROCEDURE TEXPVEC    ! M2= exp(V2_3 L_3) exp(V2_2 L_2) exp(V2_1 L_1)  if IFAC=1
END INTERFACE

INTERFACE TEXTP
MODULE PROCEDURE EXPMAT     ! Same routines as above
MODULE PROCEDURE EXPVEC
MODULE PROCEDURE EXPVEC2
MODULE PROCEDURE TEXPVEC
END INTERFACE

```

Just as an example, we can look at some parts of the subroutine FACTORIZE\_ROTATION which is used in the misalignment of a magnet.

```

!      CALL ROT_XZ( C%LOCAL%ALPHA/2.DO ,X, C%MAG%P%BETA0,OUR,C%MAG%P%TIME)  !MAKE MAGNET THICK AGAIN  1
!      CALL TRANSZ( C%LOCAL%L/2.DO ,X, C%MAG%P%BETA0,OUR,C%MAG%P%TIME)  !MAKE MAGNET THICK AGAIN  2
!      CALL ROT_YZ(C%MAG%r(1),X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)           ! rotations
!      CALL ROT_XZ(C%MAG%r(2),X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)
!      CALL ROT_XY(C%MAG%r(3),X,OU)
!      CALL TRANS(C%MAG%d,X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)               !translation
!      CALL TRANSZ( -C%LOCAL%L/2.DO ,X, C%MAG%P%BETA0,OUR,C%MAG%P%TIME) !MAKE MAGNET THIN  3
!      CALL ROT_XZ( -C%LOCAL%ALPHA/2.DO ,X, C%MAG%P%BETA0,OUR,C%MAG%P%TIME) !MAKE MAGNET THIN  4

a= y**(-1)*t_d**(-1)*R*t_d*y      ! SO(3) representation of the above dynamical code
ai= y*t_d*R**(-1)*t_d**(-1)*y**(-1) !
call factorize_m2( a,beta_in)
t_in=a/t
call factorize_m2( ai,beta_out)
t_out=ai/t
beta_in(2)=-beta_in(2)      ! PTC convention
beta_out(2)=-beta_out(2)    ! PTC convention

```

Here we can see in green the old code of PTC appearing as a comment in FACTORIZE\_ROTATION. The first line of actual code,

```
a= y**(-1)*t_d**(-1)*R*t_d*y
```

is a succession of operations on matrix\_PTC types in SO(3) using the overloaded operators described in this section. The map “a” is the SO(3) representation of the dynamical sequence commented out. In the dynamical sequence, certain maps can be ill-defined, in particular the first one which is a rotation by half the layout angle  $\alpha$ . As explained in Sect. G.4.4, near  $\alpha = \pi$  this map is completely meaningless since it represents backwards propagation even though the misalignments are small and the total misalignment map is certainly an operator near the identity. The SO(3) representation does not have this problem and thus “a” is well-defined. The map “a” is then factorized again in the standard order of PTC. The result is used in the tracking routine MIS\_FIB where the dynamical operators resurface.

## E Sb\_EXTEND\_POLY.f90

This file contains two modules : the module S\_extend\_poly and the module ANBN. S\_extend\_poly extends a few of the FPP interfaces and assignments to things specific to PTC. It is described here in detail.

The module ANBN solves Maxwell's equation for the "exact sector bend." It is well known that the regular harmonic solution for multipoles is the correct solution for a straight (infinite) element. For an element with rotational (cyclotron-like) symmetry, corrections proportional to power of  $EL\%P\%B0$  must be introduced. ANBN solves this problem to a user-specified order when the code boots in the compulsory routine MAKE\_STATES of Sect. I. It is used in the element TEAPOT (See Sect. K.4.9).

### E.1 The Explosive Functions

PTC, in exact mode, is likely to encounter square roots. Therefore to prevent disastrous overflows, we have explicitly replaced the real square root by a new function ROOT(X). In addition, in wiggler calculations, hyperbolic functions do appear and require special treatment. We list here the four beasts presently defined:

```
REAL(DP) FUNCTION  ROOT(X)      ! REPLACES SQRT(X)
REAL(DP) FUNCTION  SINEHX_X(X) ! REPLACES SINH(X)/X
REAL(DP) FUNCTION  COSEH(X)    ! REPLACES COSH(X)
REAL(DP) FUNCTION  SINEH(X)    ! REPLACES SINH(X)
```

These functions for arguments exceeding some values (and for roots of negatives) set the global parameter negative CHECK\_STABLE to false. It is then intercepted by the routine TRACK\_FIBRE\_R of module S\_TRACKING. Tracking is then interrupted.

### E.2 The REAL\_8 Type

This is found in the file f\_definition.f90 of the FPP package. It is displayed here for completeness.

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

### E.3 The (=) Assignment

We list the routines which define the interfaces for the (=) assignment. The reader can then look them up easily.

#### E.3.1 REAL\_8REAL6: Y=X

$Y=X$  is permitted. The definition is the obvious one if the array Y is made of kind=1 or kind=2 polymorphs. However, if prior to this assignment, the assignment  $Y=NPARA$  is used, then  $Y=X$  will produce

$$\begin{aligned} Y(i) &= X(i) + x_i \\ Y(i) &= X(i) \text{ for } i > NPARA . \end{aligned} \tag{23}$$

In the above equation,  $x_i$  is simply a monomial in the  $i^{th}$  variable in the TPSA package. See also Sect. E.4 for another PTC way to initialize a polymorph in terms of a DAMAP.

#### E.3.2 REAL6REAL\_8: X=Y

This allows the  $X=Y$  assignments. It puts simply the real part of  $Y(i)$  into  $X(i)$ .

### E.3.3 REAL\_8REAL\_8: Y2=Y1

This allows the Y2=Y1 assignments: one array of polymorphs is copied into another one.

### E.3.4 ENV\_8MAP: YS=M

This allows the copying of a map M in the part of YS which contains the “ray,” which is just the Taylor map. This is used only in a fixed point routine for the beam envelope (in So\_FITTING.f90).

### E.3.5 REAL6ENV\_8: X=YS

This simply permits one to dump the constant part of the map YS(6)%V into the polymorphic array X(6).

### E.3.6 Initializing an Envelope with ENV\_8T or ENV\_8BENV: YS=T or YS=ENV

This routines feed the quadratic part of a Taylor series T in the YS(6)%SIGMA0(6) array. It works as follows:

$$\text{if } \sum_{i,j}^6 T_{ij} x_i x_j \implies T_{ij} = \text{YS}(i)\%SIGMA0(j) \quad (24)$$

This uses the operator .PAR. of FPP. Thus the array  $T_{ij}$  is a polynomial in the polymorphic knobs.

The purpose of this routine is to initialize an envelope YS with the initial value of the quadratic moments  $\langle x_i x_j \rangle$  stored as the coefficients  $T_{ij}$  of T. For example, if ENV is a normal form of the beam envelope (type beamenvelope), then the equilibrium beam sizes are in ENV%SIJ0. One can initialize YS using the assignment YS=ENV%SIJ0. This is a first step towards the propagation of a beam envelope.

One can also use the syntax YS=ENV which will automatically invoke YS=ENV%SIJ0.

### E.3.7 Extracting the Tracked Envelope with TENV\_8: T=YS

Whenever an envelope YS is tracked, at the end of a tracking call, PTC evaluates the final moments using Equation (14) and stores the results in YS(6)%SIGMAF(6). As in Sect. E.3.6, the following polynomial is created.

$$\text{if } \sum_{i,j}^6 T_{ij} x_i x_j \implies T_{ij} = \text{YS}(i)\%SIGMAF(j) \quad (25)$$

using the syntax T=YS. T is of type Taylor.

## E.4 The Operator +

The following lines are permissible in PTC.

```
Y=NPARA          ! Makes the first NPARA polymorph KIND=0
Y=NORMAL%A_T+X   ! Adding a real array with a DAMAP; only allowed in PTC
CALL TRACK(LAYOUT,Y, I,J, STATE)
```

The line Y=NORMAL%A\_T+X or Y=X+NORMAL%A\_T add to a map an array of six double precision numbers. In general this allows the addition of the fixed point to a DAMAP such as the transformation  $A$  of a normal form. It is important to precede this by the Y=NPARA assignment. This will create REAL\_8 variables of kind=0. It should be said that Taylor series are assigned here to a kind=0 polymorph; this is normally forbidden by FPP. Here PTC overrules this restriction setting the FPP variable “INSANE\_PTC” to true in the MAKE\_STATES routine.

## E.5 The PRINT and DAPRINT Interface

### E.5.1 Printing and Reading Arrays

The statement

```
CALL PRINT(Y,MF) or CALL DAPRINT(Y,MF)
```

will print on file number MF the 6 variables of Y for human or machine consumption.

### **E.5.2 Printing a Beam Envelope YS**

One can use “CALL PRINT(YS,MF)” or “CALL DAPRINT(YS,MF)” to print the entire content of a beam envelope for human consumption.

## **E.6 The ALLOC and KILL Interface**

### **E.6.1 Allocating Arrays**

Calls to ALLOC(Y) and KILL(Y) will allocate the polymorphic array or kill it. These are constructor and destructor routines for the polymorphic arrays. The same syntax works on beam envelopes: ALLOC(YS) and KILL(YS). The functions are extensions of similar procedures in the FPP package.

## **E.7 The Context Routine**

Call to CONTEXT(String) replaces all the lower case letters of String with upper case letters:

```
character*5 ggg  
ggg='RTss1'  
call context(ggg)  
write(6,*) ggg
```

This produces RTSS1.

## F Sc\_i\_POL\_TEMPLATE.f90 and Sg\_i\_template\_MY\_KIND.f90

Please read this chapter later as it will make little sense at this stage. This is its proper position in the PTC hierarchy but not in a pedagogical sense!

Sg\_i\_template\_MY\_KIND.f90 ( $i=1$  or  $2$ ) contains the modules USER\_KIND $i$  where “ $i$ ” is either 1 or 2. Obviously, these blank templates cannot be tracked, but they act as a guide as well as being necessary for compilation. The reader can consult the file Sg\_2\_arbitrary.f90 for an actual implementation of a WIGGLer in USER2. In addition, we show how one can connect internal variables of these user kinds to the type POL\_BLOCK. This is done in ELP\_POL\_USER1 and ELP\_POL\_USER2. In the template, the variable INTERNAL of the template is also associated to the VOLT variable of POL\_BLOCK.

Furthermore one can modify the fields USER1 and USER2 of a POL\_BLOCK to provide a flexible connection to anything in the user defined element. The fields are defined in Sc\_i\_POL\_TEMPLATE.f90. Again, in our example, USER1 and USER2 connect to the variable INTERNAL. **This is the safest way to deal with the interaction of POL\_BLOCK and the user defined kinds. However, USER1 and USER2 also provide an automatic interface to The arrays AN and BN of EL and ELP.**

Generally the user defined element should be written for real(dp) variables, REAL\_8 polymorphs, and ENV\_8 envelopes. This will ensure that the map based theory carries over. This is essential, as we should not have to point out here.

There are two very useful types in PTC whose properties and functions should also work for a user defined element: POL\_BLOCK and WORK. Once more, we must provide the methods manually which will enforce a certain degree of inheritance. We will discuss this PTC-specific in the following two sections.

### F.1 Dealing with POL\_BLOCK

POL\_BLOCKS are very useful in dealing with polymorphic variables. First let us write down the USER1P element as defined in the template:

```
TYPE USER1P
  TYPE(MAGNET_CHART) P
  TYPE(REAL_8), POINTER ::L                ! MUST ALWAYS BE THERE
  TYPE(REAL_8), DIMENSION(:), POINTER :: AN,BN !Multipole component
                                              !(OPTIONAL but always defined)
  !   ADD INTERNAL STUFF HERE AS POINTERS
  !   .....
  TYPE(REAL_8), POINTER ::INTERNAL ! Example of a variable specific to user1p
END TYPE USER1P
```

The variable P of type MAGNET\_CHART is always present. It passes important knowledge between ELEMENTP and USER1P. The variable L is always there. It is defined, allocated and killed in S\_DEF\_ELEMENT. The variables AN and BN are also handled in the same module. You may use them if you have some multipoles in your element. These objects will be automatically handled by POL\_BLOCK, MUL\_BLOCK, and WORK as if your element was KIND2 or KIND6. For example, it assumes that AN and BN are used directly in the tracking loop (unlike KIND7 and KIND10) and that these are variables scaled by POC.

The really new thing here is the variable INTERNAL. In a real application, we would have perhaps several such variables. Let us see how the routine ELP\_POL\_USER1 handles its interaction with a POL\_BLOCK.

```
SUBROUTINE ELP_POL_USER1(S2,S1,DONEIT)
  IMPLICIT NONE
  TYPE (POL_BLOCK),INTENT(IN):: S1
  TYPE(USER1P),INTENT(INOUT):: S2
  LOGICAL,INTENT(INOUT):: DONEIT

  ! ONE CAN LINK INTERNAL POLYMORPHS TO PART OF POL_BLOCK WHICH IS NOT USED
  ! HERE THE VARIABLE "INTERNAL" IS LINKED TO VOLT

  IF(S1%IVOLT>0) THEN
```

```

        S2%INTERNAL%I=S1%IVOLT+S1%NPARA
        S2%INTERNAL%S=S1%SVOLT
        S2%INTERNAL%KIND=3
        DONEIT=.TRUE.
        IF(S1%SET_TPSAFIT) THEN
            S2%INTERNAL%R=S2%INTERNAL%R+S2%INTERNAL%S*S1%TPSAFIT(S1%IVOLT)
        ENDIF
    ENDIF

! OR TRY
    IF(S1%USER1%IINTERNAL>0) THEN
        S2%INTERNAL%I=S1%USER1%IINTERNAL+S1%NPARA
        S2%INTERNAL%S=S1%USER1%SINTERNAL
        S2%INTERNAL%KIND=3
        DONEIT=.TRUE.
        IF(S1%SET_TPSAFIT) THEN
            S2%INTERNAL%R=S2%INTERNAL%R+S2%INTERNAL%S*S1%TPSAFIT(S1%USER1%IINTERNAL)
        ENDIF
    ENDIF

END SUBROUTINE  ELP_POL_USER1

```

The first part shows a case where the variable INTERNAL is tied to VOLT of ELP. One can connect single variables to VOLT, PHAS, FREQ, and B.SOL. Otherwise one can connect a variable of USER1 by using the type POL\_BLOCK1. The template connection routine is in Sc.POL.TEMPLATE.f90. Here is the entire template module as an example.

```

MODULE S_POL_USER1
IMPLICIT NONE
PRIVATE BLPOL1_0

    TYPE POL_BLOCK1
        INTEGER IINTERNAL
        REAL(DP) SINTERNAL
    END TYPE POL_BLOCK1

INTERFACE ASSIGNMENT (=)
    MODULE PROCEDURE BLPOL1_0
END INTERFACE

CONTAINS

SUBROUTINE  BLPOL1_0(S2,S1)
    IMPLICIT NONE
    TYPE (POL_BLOCK1), INTENT(INOUT):: S2
    INTEGER, INTENT(IN):: S1
        S2%SINTERNAL=1.DO
        S2%IINTERNAL=0
    END SUBROUTINE BLPOL1_0

END MODULE S_POL_USER1

```

For example, if one want to make the variable INTERNAL the first parameter, then the syntax POLB=NPARA;POLB%USER1%IINTERNAL=1; should work.

## F.2 Dealing with WORK

A variable of type WORK can be used to set up energy patches and rescale variables by some energy-like quantity such as P0C. In PTC, it is automatically assumed that the AN and BN arrays are scaled by the P0C of the ELEMENT. This is not true of the voltage. Now suppose that WORK is set to some energy P0C\_NEW, i.e., `WORK%P0C=P0C_NEW`, then the assignment

```
P=WORK
```

where P is a fibre will call the routine

```
SUBROUTINE  SCALE_USER1R(S2,POC_OLD,POC_NEW)
  IMPLICIT NONE
  TYPE(USER1),INTENT(INOUT):: S2
  REAL(DP),INTENT(IN)::  POC_OLD,POC_NEW

  ! EXAMPLE
  S2%INTERNAL= S2%INTERNAL*POC_OLD/POC_NEW

END SUBROUTINE  SCALE_USER1R
```

as well as its polymorphic counterpart. Of course this is user-specified code and it could be anything. **N.B. If the work field `WORK%RESCALE` is set to false, then the rescaling routines are not called.**



## G Sd\_EUCLIDEAN.f90

This file contains the module S\_EUCLIDEAN. The module S\_EUCLIDEAN deals with the dynamical operators of the Euclidean group.

The module S\_EUCLIDEAN contains various operators that will give the FIBRE type all its power as a “dynamical object.”

The Euclidean group is simply the group of translations and rotations in 3-dimensional space. In regular 3-dimensional affine space, these are the usual translations and rotations. They are given by the usual Lie operators of translation and rotation. In the following table we list both the time-Hamiltonian and the dynamical (or layout) operators corresponding to the Euclidean transformations:

Operators	Time	Dynamical	
$p_x$	$: p_x : = -\frac{\partial}{\partial x}$	$: p_x : = -\frac{\partial}{\partial x}$	
$p_y$	$: p_y : = -\frac{\partial}{\partial y}$	$: p_y : = -\frac{\partial}{\partial y}$	
$p_z$	$: p_z : = -\frac{\partial}{\partial z}$	$: \sqrt{(1+\delta)^2 - p_x^2 - p_y^2} :$	(26)
$L_x$	$: y p_z - z p_y :$	$: y \sqrt{(1+\delta)^2 - p_x^2 - p_y^2} :$	
$-L_y$	$: x p_z - z p_x :$	$: x \sqrt{(1+\delta)^2 - p_x^2 - p_y^2} :$	
$L_z$	$: x p_y - y p_x :$	$: x p_y - y p_x :$	

Here the notation of Dragt is used for Poisson bracket operator. The time operators are the usual ones; if restricted to position space they simply translate and rotate a geometrical figure in 3-dimensions. Of course, in the Hamiltonian framework, they also rotate the momenta.

The amazing thing about the operators of Equation (26) is that the time and dynamical versions have the same Lie algebra; but maybe not so surprising<sup>22</sup> since they represent the same physical process albeit on different objects. These operators and the time operators act on the map of a compressed magnet isomorphically in the neighborhood of the identity, i.e., for small misalignments. One notices that three of the dynamical Lie operators are identical to their time-like equivalent. These are the operators which move the magnet in the transverse plane. Obviously the  $x$  and  $y$  translations as well as the  $x - y$  rotation are decoupled neatly from the longitudinal direction. On the other hand, anything coupling with the longitudinal is nonlinear. In fact all these nonlinear operators are drifts (free space propagators); one drift in Cartesian coordinates and two drifts in polar, one around the  $y$ -axis and one around the  $x$ -axis.

Before we describe the dynamical operators in PTC, it is important to understand the concept of a “compressed” map. Often the idea of a thin lens comes to mind. Certainly a thin lens is thin! But it is not really compressed: it is thin by design and has no length. The compressed element is an element which is sandwiched between negative drifts, i.e., negative Euclidean operators so that from a layout point of view it has zero length and it is straight like a kicker. It looks like a thin lens. However it is still an exact map. For example, if we take a straight element of ideal layout length  $L_D$ , then the reader will agree that the map

$$D\left(\frac{L_D}{2}\right) \circ \diamond \circ \underbrace{D\left(-\frac{L_D}{2}\right) \circ M \circ D\left(-\frac{L_D}{2}\right)}_{\text{Compressed } M} \circ \diamond^{-1} \circ D\left(\frac{L_D}{2}\right) \quad (27)$$

is unchanged by the drifts  $D$  sandwiching  $M$  provided  $\diamond$  is identity. However if we put a member of the dynamical Euclidean group instead of  $\diamond$ , the map  $M$  will transform correctly, that is to say, like its associated 3-dimensional shape. This is why we say that the time and the dynamical operators act isomorphically on the compressed map<sup>23</sup>  $D(-L_D/2) \circ M \circ D(-L_D/2)$ . We will now list these operators as they are defined in PTC. All the operators in PTC move the magnet actively. Thus a translation of  $d_x$  moves the magnet in the positive  $x$ -direction by  $d_x$ . An  $x - y$  rotation of angle  $\alpha$  turns the magnet towards the  $y$  direction by  $\alpha$ . PTC has  $x - y$ ,  $x - z$ , and  $y - z$  rotations.

**N.B. PTC no longer uses Equation (27) directly. However it does use it in module Rotation\_mis while manipulating SO(3) operators.**

<sup>22</sup>Actually the isomorphism depends crucially on the isotropic nature of free space. It would fall apart if the magnets were not immersed in free space. Still magnets could be misaligned but one could not willy-nilly commute transverse translations and forward propagation because the transverse momenta and  $H$ , which is  $p_z$ , would not commute.

<sup>23</sup>One could consider the compressed maps as parts of a different but mathematically equivalent atlas; this is precisely the kind of flexibility Michelotti alluded to in his “two minor comments” in page 10 of his book.

## G.1 Coping with the Square Root

Before describing in some detail the operators of S\_EUCLIDEAN, we point out the existence of a “dynamical aperture” check within it. The following objects are defined in S\_EXTEND\_POLY:

```
LOGICAL :: ROOT_CHECK=.TRUE.
LOGICAL :: CHECK_STABLE=.TRUE.
PRIVATE ROOTR  INTERFACE ROOT
  MODULE PROCEDURE ROOTR
END  INTERFACE
```

It is a fact that the Euclidean operators include “square roots” in the exact option; one needs only to look at the collection of Lie operators in Equation (26) to convince oneself of the potential trouble. Therefore it is necessary to have a global flag to stop tracking whenever the argument of this square root is less than zero. The reader will notice that all square roots of PTC, for the type real(dp), have been replaced by the function “ROOT.” This function is just

```
REAL(DP) FUNCTION  ROOTR(X)
  IMPLICIT NONE
  REAL(DP), INTENT(IN) : X

  IF(X<=0.DO.AND.ROOT_CHECK) THEN
    ROOTR=1.DO
    CHECK_STABLE=.FALSE.
  ELSE
    ROOTR=SQRT(X)
  ENDIF

END FUNCTION ROOTR
```

Tracking terminates if CHECK\_STABLE is false. Without this, PTC could crash during any type of dynamic aperture search. This check is equivalent to the infamous Teapot check on the velocity of a particle exceeding the speed of light. It is a paradox that, in the “exact” option of PTC, one needs to check such things; indeed this corresponds to a particle reversing its direction of propagation within a fibre. In that case, the equations of motion are invalid. Thus both PTC (EXACT\_MODEL=.TRUE.) and Teapot are actually detecting a logical inconsistency. The laws of logic tell us that false may imply false as well as true. This is exactly what happens in this case.

## G.2 Translating an Element: TRANS(A,X,B,EXACT,CTIME)

The call TRANS(A,X,B,EXACT,CTIME), in Lie operator language, produces the map

$$\begin{aligned} \mathcal{T}(\vec{A}) &= \exp(: A_1 p_x + A_2 p_y + A_3 p_z :) \\ &= \exp\left(: A_1 p_x + A_2 p_y + A_3 \sqrt{(1 + \delta)^2 - p_x^2 - p_y^2} : \right). \end{aligned} \quad (28)$$

The logical EXACT, generally derived from EXACTMIS (see Sect. I.6.7), ensures that the drift ( $p_z$ ) is handled with the square root. If false, then the small angle approximation is used for the drift. The logical CTIME is derived from EL%TIME. In that case the time of flight is computed rather than the path length.

## G.3 Rotating an Element: ROT\_YZ, ROT\_XZ, and ROT\_XY

These are the three rotations corresponding to  $L_x$ ,  $-L_y$ , and  $L_z$  of table (26). These transformations have the syntax:

```
ROT_YZ(A,X,B,EXACT,CTIME)
ROT_XZ(A,X,B,EXACT,CTIME)
ROT_XY(A,X,EXACT)
```

where  $A$  is a real(dp) angle. The  $x - y$  rotation, which is in the transverse plane, does not affect time or path length. However it can be evaluated approximately as we will see. The  $y - z$  and  $x - z$  rotations<sup>24</sup> involve the logical CTIME; if true, time is computed, otherwise path length is computed. We remind the reader that these rotations are actually drifts in polar coordinates.

### G.3.1 The Rotation ROT\_XY in the Transverse Plane

Now let us look at the exact and approximate versions of ROT\_XY. The code is given by

```

SUBROUTINE ROT_XYR(A,X,EXACT)
  IMPLICIT NONE
  REAL(DP),INTENT(INOUT):: X(6)
  REAL(DP) XN(4)
  REAL(DP),INTENT(IN):: A
  LOGICAL,INTENT(IN):: EXACT
  IF(EXACT) THEN
    XN(1)=COS(A)*X(1)+SIN(A)*X(3)
    XN(3)=COS(A)*X(3)-SIN(A)*X(1)
    XN(2)=COS(A)*X(2)+SIN(A)*X(4)
    XN(4)=COS(A)*X(4)-SIN(A)*X(2)
    X(1)=XN(1)
    X(2)=XN(2)
    X(3)=XN(3)
    X(4)=XN(4)
  ELSE
    X(1)=X(1)+A*HALF*X(3)
    X(4)=X(4)-A*HALF*X(2)
    X(2)=X(2)+A*X(4)
    X(3)=X(3)-A*X(1)
    X(1)=X(1)+A*HALF*X(3)
    X(4)=X(4)-A*HALF*X(2)
  ENDIF
END SUBROUTINE ROT_XYR

```

The reader may wonder why the approximate ROT\_XY has this strange form: we want the prescription reversible that is to say  $R(-\alpha) = R^{-1}(\alpha)$ . This property, if true for any operator, will ensure that a drift stays invariant under a misalignment.

### G.3.2 The Rotations ROT\_XZ and ROT\_YZ

These two rotations involve the longitudinal direction  $z$ . We will list here ROT\_XZ also known in Dragt's circles as PROT. It is also found, although totally cluttered, in the main tracking loop of TEAPOT. It is given by

```

SUBROUTINE ROT_XZR(A,X,B,EXACT,CTIME)
  IMPLICIT NONE
  REAL(DP),INTENT(INOUT):: X(6)
  REAL(DP) XN(6),PZ,PT
  REAL(DP),INTENT(IN):: A,B
  LOGICAL,INTENT(IN):: EXACT,CTIME
  IF(EXACT) THEN
    IF(CTIME) THEN
      PZ=ROOT(ONE+TWO*X(5)/B+X(5)**2-X(2)**2-X(4)**2)
      PT=ONE-X(2)*TAN(A)/PZ
      XN(1)=X(1)/COS(A)/PT
      XN(2)=X(2)*COS(A)+SIN(A)*PZ
      XN(3)=X(3)+X(4)*X(1)*TAN(A)/PZ/PT
      XN(6)=X(6)+X(1)*TAN(A)/PZ/PT*(ONE/B+X(5))
    ELSE
      PZ=ROOT((ONE+X(5))**2-X(2)**2-X(4)**2)

```

---

<sup>24</sup>The  $x - z$  rotation is Dragt's famous PROT. It is used in the file Sh\_DEF\_KIND.f90 since it is needed in the exact parallel face bend. It is also central to the exact sector bend integrator as in the type and the code TEAPOT; there it was more convenient to re-implement it under the interface name SPROT for the same reason that  $z$ -translations are re-implemented as DRIFT.

```

PT=ONE-X(2)*TAN(A)/PZ
XN(1)=X(1)/COS(A)/PT
XN(2)=X(2)*COS(A)+SIN(A)*PZ
XN(3)=X(3)+X(4)*X(1)*TAN(A)/PZ/PT
XN(6)=X(6)+(ONE+X(5))*X(1)*TAN(A)/PZ/PT
ENDIF
X(1)=XN(1)
X(2)=XN(2)
X(3)=XN(3)
X(6)=XN(6)
ELSE
IF(CTIME) THEN
PZ=ROOT(ONE+TWO*X(5)/B+X(5)**2)
X(2)=X(2)+A*PZ
X(6)=X(6)+A*X(1)*(ONE/B+X(5))/PZ
ELSE
X(2)=X(2)+A*(ONE+X(5))
X(6)=X(6)+A*X(1)
ENDIF
ENDIF
END SUBROUTINE ROT_XZR

```

In the small angle approximation the tilt of an element is simply a translation of the momentum by  $A(1 + \delta)$ .

## G.4 A Matter of Perspective: Are we Patching Here?

### G.4.1 Defining Patching

First of all we define patching. Patching refers to the connection between two elements. The rays at the exit plane of an element are transported to the entrance plane of the following element. In PTC, in a standard survey mode, all the elements, including drifts, are patched to one another automatically. In fact it is assumed that no patches are needed; this is how a standard “MAD” survey is done. But this is not necessarily the case. One simple example is that of the extraction from one beam line to another.

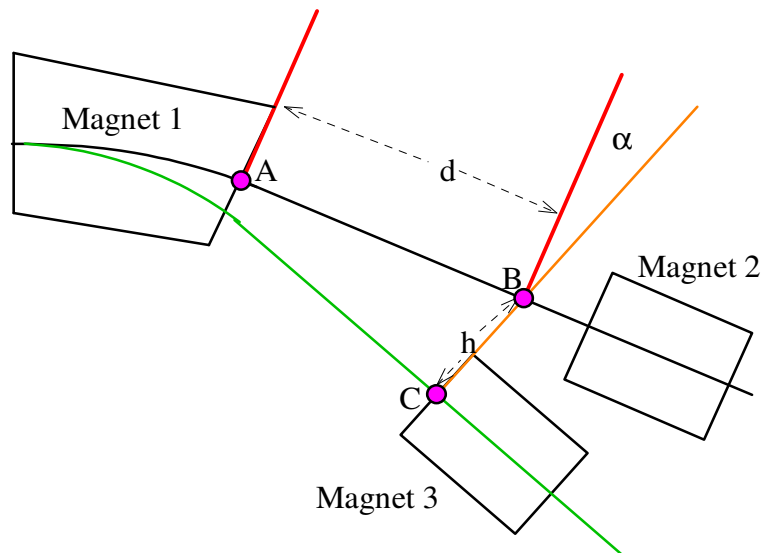


Figure 15: Patching two Beam Lines

In Figure 15, we show two beam lines inspired by an actual problem. The black line shows the ideal trajectory of a 10 GEV muon. The green line shows the trajectory of a 4 GEV muon. At the beginning both particles are in the “10 GEV” line and share the first magnet (magnet 1). The 10 GEV particle continues towards magnet 2. The 4 GEV particle is bent further by magnet 1 and thus leaves the 10 GEV beam line as it propagates towards magnet 3— the first magnet of its dedicated beam line. PTC handles this kind of beam line switching naturally thanks to the fibre bundle structure of the layout.

Part of the problem is to transfer the 4 GEV ray from point A to point C. More precisely to express the rays in the orange frame at point C. Geometrically one sees a first translation from A to B in a frame which is perpendicular to line AB. This is just a drift of length  $d$ . The next step is a rotation around B of angle  $\alpha$  in the  $x - z$  plane, i.e., ROT\_XZ. Finally the only thing left to do is a simple translation  $h$  in the  $x$  direction. All these operation can be deduced from geometry, from the position of the exit plane at A and

the entrance plane at C. The reader can imagine other combinations of operators producing the same results. The “geometrical” connections are called patches and they are useful when dealing with strange beam lines.

#### G.4.2 An Example: First, Using the Compression Trick

The following three sections are perhaps not truly germane to the present version of PTC. Obviously by using the local isomorphism between  $SO(3)$  and the dynamical operators we are de-facto replacing compression by patching. Nevertheless it is still instructive to go through the presentation because PTC in fact compresses in  $SO(3)$ .

Now let us come back to misalignments. Are we patching there too? The answer is yes; however the trick of the “compressed element” permits us to hide this “under the carpet.” Let us see on an example how our theory is indeed equivalent to a set of patches. Consider the following rotation of a straight element. Here we will first use the “compressed element trick.” The reader will notice that for this example it is easier to compress the element at the entrance. Using the Lie operator order, we can thus rotate the map using the formula:

$$\mathcal{M}_\theta = \underbrace{\mathcal{Y}_\theta \underbrace{\mathcal{M}\mathcal{D}^{-1}}_{\text{Compressed}} \mathcal{Y}_{-\theta}}_{\text{Rotated Compressed Map}} \mathcal{D}$$

where

$$\mathcal{Y}_\theta = \exp\left(:\theta x \sqrt{(1+\delta)^2 - p_x^2 - p_y^2}:\right)$$

$$\mathcal{D} = \exp\left(:O_{12} \sqrt{(1+\delta)^2 - p_x^2 - p_y^2}:\right). \quad (29)$$

Here  $O_{12}$  is obviously the distance between  $O_1$  and  $O_2$ .

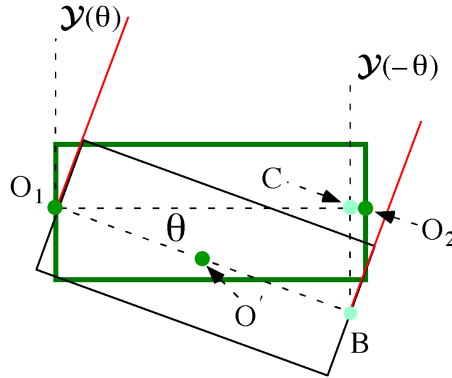


Figure 16: Rotating a Straight Element at the Entrance Face

#### G.4.3 Same Example: Now, Using Patching

Of course, in this case, it is also possible to use the figure and derive the patches from the geometry. We get the following equivalent result.

$$\mathcal{M}_\theta = \mathcal{Y}_\theta \mathcal{M} \mathcal{Y}_{-\theta} \mathcal{T}_x(BC) D(CO_2)$$

where

$$\mathcal{T}_x(BC) = \exp(:BC p_x:)$$

$$D(CO_2) = \exp\left(:CO_2 \sqrt{(1+\delta)^2 - p_x^2 - p_y^2}:\right)$$

$$BC = O_{12} \sin \theta$$

$$CO_2 = O_{12} (1 - \cos \theta). \quad (30)$$

Clearly Equation (30) is the result of a patching operation. We rotate back from the exit frame to the BC frame. We then translate in the  $x$ -direction to C. Finally we drift to  $O_2$ . Now the fun part is to see how

the manipulation of the Lie operators in Equation (29) gives the same results. Better, because of the local isomorphism, it does not matter if we use the time or the dynamical operators. Here we chose the dynamical since they are certainly those to be used in the tracking code. So let us see if Equation (29) is identical to Equation (30). If it is, then we must have

$$\begin{aligned}
& \exp(: -O_{12}p_z :) \mathcal{Y}_{-\theta} \exp(: O_{12}p_z :) = \mathcal{Y}_{-\theta} \exp(: BCp_x :) \exp(: CO_2p_z :) \\
\Rightarrow & \mathcal{Y}_{\theta} \exp(: -O_{12}p_z :) \mathcal{Y}_{-\theta} \exp(: O_{12}p_z :) = \exp(: BCp_x :) \exp(: CO_2p_z :) \\
\Rightarrow & \exp(: -O_{12}\mathcal{Y}_{\theta}p_z :) \exp(: O_{12}p_z :) = \exp(: BCp_x + CO_2p_z :). \tag{31}
\end{aligned}$$

However we have

$$\mathcal{Y}_{\theta}p_z = \sqrt{(1 + \delta)^2 - (p_x \cos \theta + p_z \sin \theta)^2 - p_y^2} \tag{32}$$

and substituting in Equation (31) and equating the Lie operators, which all commute:

$$\begin{aligned}
-O_{12}\sqrt{(1 + \delta)^2 - (p_x \cos \theta + p_z \sin \theta)^2 - p_y^2} + O_{12}p_z &= CO_2p_z + BCp_x \\
\Rightarrow -\sqrt{(1 + \delta)^2 - (p_x \cos \theta + p_z \sin \theta)^2 - p_y^2} &= (\sin \theta p_x - \cos \theta p_z)
\end{aligned}$$

The reader can square the above expression on both sides of the equal sign and check indeed that this equality holds.

*Remarkably, the reader will notice that the present version of PTC would produce exactly the ordering of Equation (30). Of course it is not done through an analytical derivation as in this section. Instead the operators in the compression formulas are re-factored in the module Rotation\_mis using some iterative algorithms in SO(3).*

#### G.4.4 Failure of Compression: Decompression!

We now discuss why PTC, using module Rotation\_mis, goes back and forth between the dynamical Euclidean group and the ordinary  $R^3$  representation.

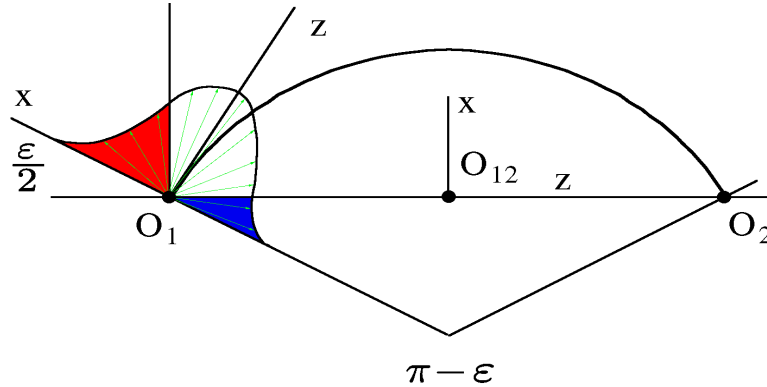


Figure 17: Why Dynamical Compression fails in general

The reader will notice that ROT\_XZ diverges for  $\alpha = 90^\circ$ ; thus the dynamical representation is not global. This means that if the layout angle of a bend is around  $180^\circ$  then the entire compression approach fails. Thus generally, for big magnets, one should re-express the theory around  $180^\circ$ . The magnets should be “decompressed” into  $180^\circ$  bends. Although one could support this in PTC it is better to use a different approach altogether.

Figure 17 shows the difference between compression and decompression. For this magnet, of layout angle  $\pi - \epsilon$ , we displayed a distribution of incoming rays which are propagating forward in the layout. The red part corresponds to rays lost by compression into a thin magnet. This is because these rays would suddenly propagate backwards, something forbidden in the  $z$ -parameterized Hamiltonian. However, if a “decompression” technique is used, then the blue rays are lost.

Thus we do need the two approaches in a code capable of handling large bending angles unless the isomorphism to  $SO(3)$  is used. This is why PTC does all its compression work in  $SO(3)$ .

## G.5 Routines of Sd\_EUCLIDEAN

This module contains simply the Euclidean group operators: Drifts and Rotations. The so-called “NEW ROUTINES” are never used in the present version of PTC. In these routines, the input parameters (angles and lengths) are polymorphs. With these routines and complex programming in S\_DEF\_KIND, one could imagine making the actual length of a magnet a true polymorph. For a short discussion of PTC’s inability to make the integration length L a true polymorph, please read Sect. O.1.2.

```
INTERFACE TRANS
  MODULE PROCEDURE TRANSR
  MODULE PROCEDURE TRANSP
  MODULE PROCEDURE TRANSP_P ! NEW ROUTINES WITH POLYMORPHS ONLY
  MODULE PROCEDURE TRANSS
  MODULE PROCEDURE TRANSS_S ! NEW ROUTINES WITH POLYMORPHS ONLY
INTERFACE ROT_XZ
  MODULE PROCEDURE ROT_XZR
  MODULE PROCEDURE ROT_XZP
  MODULE PROCEDURE ROT_XZP_P ! NEW ROUTINES WITH POLYMORPHS ONLY
  MODULE PROCEDURE ROT_XZS
  MODULE PROCEDURE ROT_XZS_S ! NEW ROUTINES WITH POLYMORPHS ONLY
INTERFACE ROT_YZ
  MODULE PROCEDURE ROT_YZR
  MODULE PROCEDURE ROT_YZP
  MODULE PROCEDURE ROT_YZP_P ! NEW ROUTINES WITH POLYMORPHS ONLY
  MODULE PROCEDURE ROT_YZS
  MODULE PROCEDURE ROT_YZS_S ! NEW ROUTINES WITH POLYMORPHS ONLY
INTERFACE ROT_XY
  MODULE PROCEDURE ROT_XYR
  MODULE PROCEDURE ROT_XYP
  MODULE PROCEDURE ROT_XYP_P ! NEW ROUTINES WITH POLYMORPHS ONLY
  MODULE PROCEDURE ROT_XYS
  MODULE PROCEDURE ROT_XYS_S ! NEW ROUTINES WITH POLYMORPHS ONLY
SUBROUTINE ROT_YZR(A,X,B,EXACT,CTIME)
SUBROUTINE ROT_YZP(A,X,B,EXACT,CTIME)
SUBROUTINE ROT_YZP_P(A,X,B,EXACT,CTIME)
SUBROUTINE ROT_YZS(A,Y,B,EXACT,CTIME)
SUBROUTINE ROT_YZS_S(A,Y,B,EXACT,CTIME)
SUBROUTINE TRANSR(A,X,B,EXACT,CTIME)
SUBROUTINE TRANSP(A,X,B,EXACT,CTIME)
SUBROUTINE TRANSP_P(A,X,B,EXACT,CTIME)
SUBROUTINE TRANSS(A,Y,B,EXACT,CTIME)
SUBROUTINE TRANSS_S(A,Y,B,EXACT,CTIME)
SUBROUTINE ROT_XYR(A,X,EXACT)
SUBROUTINE ROT_XYP(A,X,EXACT)
SUBROUTINE ROT_XYP_P(A,X,EXACT)
SUBROUTINE ROT_XYS(A,Y,EXACT)
SUBROUTINE ROT_XYS_S(A,Y,EXACT)
SUBROUTINE ROT_XZR(A,X,B,EXACT,CTIME)
SUBROUTINE ROT_XZP(A,X,B,EXACT,CTIME)
SUBROUTINE ROT_XZP_P(A,X,B,EXACT,CTIME)
SUBROUTINE ROT_XZS(A,Y,B,EXACT,CTIME)
SUBROUTINE ROT_XZS_S(A,Y,B,EXACT,CTIME)
```

## H Se\_FRAME.f90

This module contains all the geometric operations associated with the charts. It was written very early, in collaboration with Aimin Xiao, while trying to use PTC for some useful HERA calculations.

### H.1 The Charts and the Patches

This type contains the information which locates the ideal position of an element in 3-dimensional space. It is given by

```

TYPE CHART
  TYPE(MAGNET_FRAME), POINTER :: F
  REAL(DP), POINTER :: A_XY
  REAL(DP), POINTER :: L
  REAL(DP), POINTER :: ALPHA
  ! FIBRE MISALIGNMENTS => NOW ALWAYS TRUE!
  REAL(DP), DIMENSION(:), POINTER :: D_IN, ANG_IN
  REAL(DP), DIMENSION(:), POINTER :: D_OUT, ANG_OUT
END TYPE CHART

```

where the MAGNET\_FRAME is defined as

```

TYPE MAGNET_FRAME
  REAL(DP), POINTER, DIMENSION(:,:) :: ENT
  REAL(DP), POINTER, DIMENSION(:) :: A
  REAL(DP), POINTER, DIMENSION(:,:) :: EXI      REAL(DP), POINTER, DIMENSION(:) :: B
  REAL(DP), POINTER, DIMENSION(:,:) :: MID
  REAL(DP), POINTER, DIMENSION(:) :: O
END TYPE MAGNET_FRAME

```

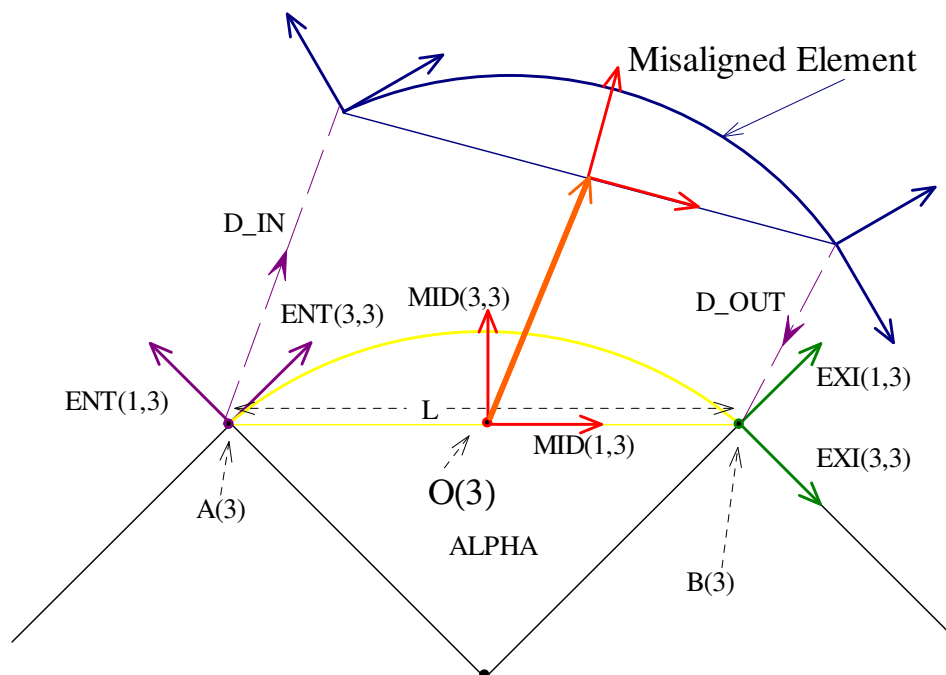


Figure 18: The Chart

Figure 18 shows the charts attached to an element. The map (subroutine TRACK(EL,X) of Sect. L) must send a ray from the purple affine basis (A,ENT) to the green exit affine basis (B,ENT). Whoever programs new elements must ensure that this is the default for an element of PTC. If necessary, internal patches deep inside the coding for the magnet must be used. This is why module S\_DEF\_KIND contains calls to ROT\_XZ in the routines for type STREX— the correct parallel face bend. The length  $L$  of the chart must coincide with



the Cartesian length  $EL\%LC$  while the path length along the arc corresponds to  $EL\%LD$ . Finally  $ALPHA$  is  $EL\%P\%B0*EL\%P\%LD$ . The above conditions are assumed to be true during the execution of a standard survey. Finally the angle  $A\_XY$  allows a rotation of the yellow figure in the case of a non planar bend (non zero  $EL\%TILTD$ ). Actually there is a simple planar usage of  $A\_XY$ , namely  $A\_XY = \pi$ : it corresponds to an outward bend.

Finally the variables  $(D\_IN,ANG\_IN)$  and  $(D\_OUT,ANG\_OUT)$  are the actual displacements and rotations at the entrance and exit of the fibre. They are symbolically represented by the purple dash arrows. They are computed on the basis of the arrays  $MAG\%R(3)$  and  $MAG\%D(3)$  as explained in Sect. D.

```

TYPE PATCH
  LOGICAL, POINTER:: PATCH
  LOGICAL, POINTER:: ENERGY
  LOGICAL, POINTER:: TIME
  REAL(DP), POINTER:: A_T,B_T
  REAL(DP),DIMENSION(:), POINTER:: A_D,B_D,A_ANG,B_ANG
END TYPE PATCH

```

The patch variables are used in the layout tracking if patching is necessary. For example it is useful when connecting beam lines as shown in Sect. G.4.1. Patches are also necessary when a vertical bump is put in place. In such a case, standard surveys are not reliable. Finally, in advanced forms of design, it might be useful to position magnets using some CAD tools. This is the case of spreaders in muon colliders. For such rings the CAD program will create the layout; putting the magnets may require patching.

In addition, PTC supports energy and temporal patches. The energy patch is useful if the reference  $EL\%P\%BETA0$  is upgraded along a beam line. A non-trivial example of this can be seen in Sect. L.4.5. In that case the momenta and the energy variable  $X(5)$  are rescaled according to the new and old energy, i.e., according to the present and the following magnet settings. The temporal patch is a trivial change of reference time. The variable  $X(6)$  loses a quantity  $A\_T$  at the entrance and  $B\_T$  at the exit of the magnet.

In summary, the variables  $(A,ENT)$ ,  $(O,MID)$ , and  $(B,EXI)$  define three affine bases attached to the beam pipe. They locate the element in space. The variables  $L$  and  $ALPHA$  characterize the yellow plane of the magnet. At the center of this plane lies the affine basis at  $\Omega$ . This is where a magnet is compressed for misalignment purposes. Finally the patches are variables used in the main tracking loop to perform certain adjustments if the exit chart of one element does not connect smoothly with the entrance chart of the element that follows it: beam line transfer, vertical bumps, CAD produced layout, reference energy changes, etc...

## H.2 Subroutines of S\_FRAME

There are some potentially very useful routines in module `S_FRAME`. These routines include those needed for a standard survey, some routines which allow the displacement of entire layouts and some routines for patching. The patching routine `FIND_PATCH_B`, which is overloaded later to act on a fibre, is particularly important in a non trivial lattice.

We list here the routines and their interfaces:

```

INTERFACE ASSIGNMENT (=)
  MODULE PROCEDURE ZERO_CHART
  MODULE PROCEDURE ZERO_PATCH
INTERFACE EQUAL
  MODULE PROCEDURE COPY_CHART
  MODULE PROCEDURE COPY_PATCH
INTERFACE COPY
  MODULE PROCEDURE COPY_CHART1
  MODULE PROCEDURE COPY_PATCH1
INTERFACE GEO_ROT
  MODULE PROCEDURE GEO_ROTA
  MODULE PROCEDURE GEO_ROT B
  MODULE PROCEDURE GEO_ROT V
INTERFACE FIND_PATCH
  MODULE PROCEDURE FIND_PATCH_B
SUBROUTINE COPY_PATCH(B,A) ! B<-A
SUBROUTINE COPY_CHART(B,A) ! B<-A
SUBROUTINE COPY_CHART1(A,B) ! A->B
SUBROUTINE COPY_PATCH1(A,B) ! A->B
SUBROUTINE ZERO_PATCH(F,R) !R=0 NULLIFIES AND ALLOCATES ; R=-1 DEALLOCATES
SUBROUTINE ZERO_CHART(F,R) !R=0 NULLIFIES AND ALLOCATES ; R=-1 DEALLOCATES

```

```
SUBROUTINE COPY_VECTOR(R,F) ! R%ENT,R%MID,R%EXI -> F%ENT,F%MID,F%EXI
SUBROUTINE GEO_TRA(A,ENT,D,I) ! ADDS/SUBTRACTS D TO A WHERE D IS EXPRESSED IN THE ENT FRAME
SUBROUTINE GEO_AVE(ENT,A,EXI,B,MID,O) ! FINDS MID AND O
SUBROUTINE GEO_ROTA(ENT,A,I) ! ROTATES FRAME ENT BY A(3) IN THE PTC OR REVERSE PTC ORDER
SUBROUTINE GEO_ROT(B,ENT,EXI,A_XY,A_XZ,A_YZ) ! ROTATES ENT INTO EXI
SUBROUTINE GEO_ROT(V,A) ! ROTATES V BY ANGLES A(3) IN PTC ORDER
SUBROUTINE ROTATE_V(E,F,DIRVER) ! ROTATES EXIT INTO ENTRANCE OR VICE VERSA (FOR SURVEY)
SUBROUTINE ROTATE_C(E,F,DIRVER) ! PUTS E AT THE END OF F FOR SURVEY PURPOSES. E AND F MUST HAVE SAME DIRVER
SUBROUTINE ROTATE_E(A,ENT,O,MID,B,EXI,B_LAT,ROT) ! THIS ROUTINE IS USED IN ROTATE_E
SUBROUTINE FIND_PATCH_B(A,ENT,B,EXI,D,ANG) ! FINDS PATCH BETWEEN ENT AND EXI : INTERFACED LATER FOR FIBRES
```

# I Sf\_STATUS.f90

This module is primarily concerned with definitions and the operations on internal states. It contains the routines which initialize PTC, including the states, the integration methods and the Maxwellian template for the B-field in cylindrical geometry. A few important types are defined here, which are in order of increasing importance, TILTING, WORK, POL\_BLOCK and MAGNET\_CHART.

Finally two operations on the ray are conveniently located here: DTILTD and B2PERP. These are respectively the design tilts and the  $B_{\perp}^2$  used in the computation of radiation effects.

## I.1 Constants of Sf\_STATUS.f90

There are a few useful variables defined in Sf\_STATUS.f90. We list them here.

1. Integers KIND0 to KIND16 are the standard magnets. They will be described in Sect. K.
2. KINDFITTED refers to an experimental fitted magnet. It is defined in the file Sg\_0\_FITTED.f90. It is not yet documented.
3. KINDUSER1 and KINDUSER2 refer to user defined elements. One can compile PTC with an empty template or fill it in with an element.
4. DRIFT\_KICK\_DRIFT, MATRIX\_KICK\_MATRIX and KICK\_SIXTRACK\_KICK are KIND2, KIND7 and KIND6 respectively. Although, for example, DRIFT\_KICK\_DRIFT is equal to KIND2, the routine which creates the element in the module MAD\_LIKE may assign a kind different from KIND2 to the element. For this reason, it is better to use DRIFT\_KICK\_DRIFT since the name is more representative of the properties of the element created.
5. In connection with item 4, PTC has three variables that are used in the MAD-like language of Sect. Q to indicate the type of model desired by the user. There are also three pointers which automatically assign a better name to these variables while keeping backward compatibility.

```
INTEGER, TARGET :: MADKIND2=KIND2
INTEGER, TARGET :: MADKIND3N=KIND3
INTEGER, TARGET :: MADKIND3S=KIND3
INTEGER, POINTER :: MADTHICK
INTEGER, POINTER :: MADTHIN_NORMAL
INTEGER, POINTER :: MADTHIN_SKEW
.
.
.
MADTHICK=>MADKIND2
MADTHIN_NORMAL=>MADKIND3N
MADTHIN_SKEW=>MADKIND3S
```

6. The global variable EXACT\_MODEL, if true, enforces the correct treatment for the body of the magnet. We support the straight elements, including the standard rectangular bends and the sector bend. This is useful particularly in small machines although it might also be necessary in strongly focusing interaction regions and other extreme situations. This flag is used at creation time. Once an element is created, it is not possible to change its model with a simple flag.
7. The logical STOCH\_IN\_REC removes any attempt to evaluate stochastic effects in elements with a vanishing  $EL\%P\%B0$ . This is necessary if one wants parameter dependence of the beam envelope. PTC has expressions which are only valid in elements with big and rather constant bending. Generally it does not hurt to evaluate this effect in straight elements as well, however, the formula is not differentiable around zero B-field. The user can force the evaluation by setting STOCH\_IN\_REC to TRUE. In such a case, the parameter dependence of the beam envelope using TPSA (FPP) will be slightly off. This happens because, in straight elements, the stochastic effect is evaluated as a “double precision” entity rather than as a polymorph. Therefore the dependence of that quantity on knobs or phase space is ignored.

8. NMAX, set to 20, is used in types and routines for manipulating AN and BN. Although PTC has no software limitation on the size of NMUL, the highest multipole in a magnet, these user-friendly routines are limited to NMAX. NMAX can be changed but the code must be recompiled.
9. SECTOR\_B of type B\_CYL contains the solution of Maxwell's equations for a magnet of cylindrical geometry to order SECTOR\_NMUL, which has a default value of 4. This is used in KIND10 (type TEAPOT). The equations are solved by the module ANBN using a TPSA based technique described in [5], Interlude XIV, p.362.
10. MADKIND2 selects the kind of integration methods used for the usual magnets in Sn\_MAD\_LIKE.f90; this module contains the MAD-like input of PTC. MADKIND2 is defaulted to DRIFT\_KICK\_DRIFT, which is described in Sect. K. (There are also the more dangerous MADKIND3N and MADKIND3S. See Sect. Q.)
11. MADLENGTH is a logical. Rectangular bends accept the Cartesian length in MAD. It is defaulted to false for the MAD-like input of PTC, which means that rectangular bends accept the ideal arc length by default.
12. MAD is another logical. It is defaulted to false. It is connected to normal and skew multipole input in the MAD-like input of PTC.

$$\begin{aligned}
 \text{mad} = \text{true} \quad \Rightarrow B_y/(B\rho) &= \sum_n \frac{b_n}{(n-1)!} x^{n-1} \\
 \text{mad} = \text{false} \quad \Rightarrow B_y/(B\rho) &= \sum_n b_n x^{n-1}
 \end{aligned} \tag{33}$$

13. The integers NSTD and METD refer to the default values of the number of integration steps and the order if relevant to the kind of magnet used. The defaults are NSTD=1 and METD=2.
14. Three logical variables SETKNOB, KNOB and INSANE\_PTC belong to the FPP package. In the FPP package SETKNOB is set to false for security (no knob can be changed) and KNOB is set to true (parametric dependence is always on). In PTC the situation is reversed. KNOB should always be false. It is controlled by a unary +. (See Sect. I.6.6). SETKNOB is set to true. It allows the change of a parametric variable. The user could reverse this situation and default SETKNOB to false. In such a case, the user must make SETKNOB true whenever he is changing the real value of a parametric knob. **(Not recommended!)**  
INSANE\_PTC permits the setting of a Taylor into a virgin REAL\_8 polymorph (kind=0). It is normally false in FPP but true in PTC. (See Sect. E.4)
15. The character array ind\_stoc(6), defined by MAKE\_STATES, is used in connection with the operators .PAR. and .SUB. Feel free to use it if needed.

```

ind_stoc(1)='100000'
ind_stoc(2)='010000'
ind_stoc(3)='001000'
ind_stoc(4)='000100'
ind_stoc(5)='000010'
ind_stoc(6)='000001'

```

16. METHOD\_1,..., METHOD\_4, METHOD\_F as well as the logical NEW\_METHOD are part of five user defined symplectic integrators; arrays are filled up corresponding to these methods. See Sect. I.8.
17. TILT is of TYPE(TILTING). It is defined in the module S\_status but it is used only in the MAD-like input. See again Sect. Q. In the present module the equality of two objects of type TILTING is defined in the obvious way in the routine EQUALTILT.
18. The eternal states described in Sect. I.6.2 are all constants of PTC.

## I.2 TYPE WORK

This type is used to retrieve and set the mass and energy-like variables of the elements. It will be described later in conjunction with the modules S\_DEF\_ELEMENT and S\_FAMILY.

## I.3 TYPE POL\_BLOCK

Type POL\_BLOCK is used to assign and retrieve polymorphic attributes to and from a layout. It will be described later in conjunction with the modules S\_DEF\_ELEMENT and S\_FAMILY.

## I.4 TYPE MAGNET\_CHART

Type MAGNET\_CHART is used to provide a certain level of inheritance between the general abstract ELEMENT (or ELEMENTP) and a particular element such as TEAPOT or USER1. More importantly it contains the geometry of the element, which in PTC, is not necessarily that of the fibre. This type also contains a pointer to the direction of propagation of the fibre in which the magnet sits. The same is true for the charge: PTC also has a switch for the sign of the charge.

## I.5 TYPE INTERNAL\_STATE

This type is defined as follows:

```
TYPE INTERNAL_STATE
  LOGICAL TOTALPATH, TIME, RADIATION, NOCAVITY, FRINGE, EXACTMIS
  LOGICAL PARA_IN, ONLY_4D, DELTA
END TYPE
```

It contains a series of logicals which tell the various tracking routines how to behave.

1. TOTALPATH ensures a computation of the total path length or total time of flight
2. TIME selects time of flight rather than path length. ( $cT$  to be precise)
3. RADIATION turns on classical radiation.
4. NOCAVITY forces the code to ignore RF cavities. It has also implications on the normal form if performed in three degrees of freedom.
5. FRINGE turns on quadrupole fringe fields based on the  $b_2$  and  $a_2$  components in the element.
6. EXACTMIS if true forces the misalignments to be treated exactly.

The following flags are strictly related to TPSA calculations:

7. If PARA\_IN is true TPSA knobs are included in the calculation. It is activated by a unary + on a state.  
(As in TRACK(PSR, Y, 1, +DEFAULT))
8. If ONLY\_4D is true, then neither path length nor time is a TPSA variable. This means that the phase space dimension in the normal form will be 4. Also X(5) will not be TPSA unless DELTA is also true. See next item.
9. If DELTA is true, then ONLY\_4D is also true. However, in this case, X(5) is the 5<sup>th</sup> TPSA variable. The phase space dimension in the normal form will also be 4; momentum compaction cannot be computed.

## I.6 Defined States and Operations on States

In this section we described the built-in states of PTC and the operations allowed on them.

### I.6.1 The Basic States

The basic states are used to create new states using operations defined in Sf\_STATUS.f90. We now list them.

1. DEFAULT: all the logicals of the state are false.
2. TOTALPATH: Only TOTALPATH is true.
3. RADIATION: Only RADIATION is true.
4. NOCAVITY: Only NOCAVITY is true.
5. FRINGE: Only FRINGE is true.
6. TIME: Only TIME is true.
7. EXACTMIS: Only EXACTMIS is true.
8. ONLY\_4D: ONLY\_4D and NOCAVITY are true. A cavity cannot be tracked without path length or time of flight.
9. DELTA: DELTA, ONLY\_4D and NOCAVITY are true.

### I.6.2 The Eternal Basic States

The basic states can be modified. However a copy of the original value is kept in the eternal states. They are

- DEFAULT0
- TOTALPATH0
- RADIATION0
- NOCAVITY0
- FRINGE0
- TIME0
- EXACTMIS0
- ONLY\_4D0
- DELTA0

These states cannot be modified, they are FORTRAN constants. For example, ONLY\_4D0 is defined as:

```
TYPE(INTERNAL_STATE), PARAMETER :: ONLY_4D0 = INTERNAL_STATE(f,f,f,t,f,f,f,t,f)
```

Here “t” stands for true and “f” for false.

### I.6.3 MAKE\_STATES: Initializes PTC and Solves Maxwell’s Equations

Every main program using PTC must start with a call to MAKE\_STATES. There are two possible interfaces.

```
CALL MAKE_STATES(particle)      ! Particle is a logical
```

or as

```
CALL MAKE_STATES(muonfactor)    ! Muonfactor is a real*8
```

Particle is true for an electron (positron actually) and false for a proton. The other case is equivalent to an electron but with a scale factor of “muonfactor” for the mass.

This subroutine sets all the fundamental internal states of Sect. I.6.1 equal to the eternal states of Sect. I.6.2. In addition, it solves Maxwell’s equations in polar coordinates to order SECTOR\_NMUL ([5], Interlude XIV, p.362.). Please set SECTOR\_NMUL to the desired order prior to calling MAKE\_STATES.

Now we show how the state DEFAULT can be modified.

#### I.6.4 Addition of States: S1+S2

S1+S2 is performed by the function ADD. Each logical will be joined by the boolean operator .OR. For example,

```
ADD%FRINGE = S1%FRINGE.OR.S2%FRINGE
```

Then the following corrections are performed on the final state:

```
IF(ADD%DELTA) THEN
    ADD%ONLY_4D = T      ! T stands for .TRUE. and F for .FALSE.
    ADD%NOCAVITY = T
ENDIF
IF(ADD%ONLY_4D) THEN
    ADD%TOTALPATH = F
    ADD%RADIATION = F
    ADD%NOCAVITY = T
ENDIF
```

If we want to track in the default state with fringe fields, we can simply invoke TRACK(PSR,Y,1,DEFAULT+FRINGE).

#### I.6.5 Subtraction of States: S1-S2

S1-S2 is performed by the function SUB. It takes away a certain property. Suppose that the DEFAULT state has been redefined by the user to be DEFAULT%FRINGE=.TRUE. We will see later how this is done. Then we may want to track temporarily without quadrupole fringe fields. This can be done with the call TRACK(PSR,Y,1,DEFAULT-FRINGE0). The user should see that TRACK(PSR,Y,1,DEFAULT-FRINGE) rather than TRACK(PSR,Y,1,DEFAULT-FRINGE0) might do more than just removing fringe fields. This is because a state such as FRINGE might have been updated to include other things common to all states while FRINGE0 is an eternal state. See Sect. I.6.10.

#### I.6.6 Unary Plus: +S1

The unary plus is performed by the function PARA\_REMA. It sets S1%PARA\_IN to true. Tracking will be done with parametric dependence.

#### I.6.7 STATE%EXACTMIS versus ELEMENT%EXACTMIS

When misalignments are put in an element or a fibre, the 3-d arrays EL%D and EL%R are filled in and the appropriate CHART arrays are computed. By default, the logical EL%EXACTMIS is set to ALWAYS\_EXACTMIS, which is a global variable. This means that misalignments are done with exact formulas if ALWAYS\_EXACTMIS is true; otherwise all the angles and lengths are assumed to be small and simpler linear formulas are used.

It is possible to track with all misalignments done exactly. This is achieved with the state EXACTMIS. For example, TRACK(PSR,Y,1,DEFAULT+EXACTMIS) forces the exact computation of misalignments. If an element is set with EL%EXACTMIS = .TRUE. then it cannot be overwritten by a state whose EXACTMIS is false since the logicals are joined by an “or.”

#### I.6.8 STATE%FRINGE versus ELEMENT%PERMFRINGE

The logical EL%PERMFRINGE forces an element to always have the quadrupole fringes turned on. This is useful in colliders where the IP quadrupoles may have important quadrupole fringe fields: one turns them on “permanently” using this logical. The rules controlling STATE%FRINGE and ELEMENT%PERMFRINGE are identical to those of Sect. I.6.7. In addition it is possible to use the global logical ALWAYS\_FRINGE to force ELEMENT%PERMFRINGE to be turned on for all the created magnets.

#### I.6.9 Printing a State: PRINT\_S

Sometimes, in a debugging mode, it is useful to print a state to see if everything is going as planned! The syntax is simply call print(state,file\_number). See an example below in Sect. I.6.10.

### I.6.10 UPDATE\_STATES

After the basic states have been created, they can all be modified on the basis of a new DEFAULT state. This example was used in Sect. A.2.1.

```
CALL MAKE_STATES(.FALSE.)
EXACT_MODEL=.TRUE.
DEFAULT=DEFAULT+NOCAVITY+EXACTMIS
CALL UPDATE_STATES
MADLENGTH=.FALSE.
```

```
CALL PRINT(DEFAULT,6)
```

We used the print routine for a state. The output is

```
***** State Summary *****
MADTHICK => KIND =          32  DRIFT-KICK-DRIFT
Rectangular Bend: input arc length (rho alpha)
Default integration method          6
Default integration steps          10
This is a proton
EXACT_MODEL = TRUE
TOTALPATH   = FALSE
EXACTMIS    = TRUE
RADIATION   = FALSE
NOCAVITY    = TRUE
TIME        = FALSE
FRINGE      = FALSE
PARA_IN     = FALSE
ONLY_4D     = FALSE
DELTA       = FALSE
```

Updating states is not compulsory, but it sometimes make sense to update all the states. They then share all the same default properties.

### I.6.11 CLEAR\_STATES

This subroutines forces all the states to return to their “eternal” value. One simply calls CLEAR\_STATES without any arguments.

## I.7 Initializing FPP within PTC: INIT

### I.7.1 INIT in FPP

The polymorphic package is normally initialized by calling a routine in the module polymorphic\_complex\_taylor. There are two types of calls. The first call is a plain TPSA call. It is invoked by

```
CALL INIT(NO1, NP1, PACKAGE)
```

Here NO1 is the degree of the TPSA polynomials and NP1 is the number of variables. PACKAGE is true if Berz’s package is used.

However, being accelerator physicists, we may want to tell the FPP package that we are dealing with “differential algebraic maps” rather than just doing plain TPSA. This is done with the call

```
CALL INIT(NO1, ND1, NP1, NDPT1, PACKAGE)
```

Here ND1 is the number of degrees of freedom. NP1 is the number of parametric variables. NDPT1 is either 0, 2\*ND1-1, or 2\*ND1. If non zero, then the last plane is non-oscillatory and NDPT1 is the position of the constant energy-like variable. In that case, a Jordan normal form is performed in the last plane and momentum compaction is computed.



## I.7.2 INIT in PTC

The reader will notice that using a direct call to FPP is rather tedious within the context of PTC. For example, in PTC the constant energy-like variable is *always* X(5). Therefore, NDPT1 is either 0 or 5. Someone used to MAD or Marylie notation might choose 6. The code would go nuts!

Instead we have defined internal states. These internal states imply a certain choice of the FPP call to INIT. We automate this using a routine in the module S\_status. Thus one uses

```
CALL INIT(STATE,NO1,NP1,PACKAGE,ND2,NPARA)
```

There are two output variables ND2 and NPARA. ND2 is just the dimension of phase space: 4 or 6 is PTC. NPARA controls the location of the first knob, it is the (NPARA + 1)<sup>th</sup> variable of the TPSA package. This quantity in PTC will be 4, 5, or 6. It is best explained by an example from our PSR runs.

```
WRITE(6,'(6(G14.7))') X
CALL INIT(DEFAULT,2,1,BERZ,ND2,NPARA)
CALL ALLOC(Y);
Y=NPARA
Y=X                ! MAKES Y = CLOSED ORBIT + IDENTITY
CALL PRINT(Y,6)
```

The result is an identity map around the closed orbit. We remind the reader that the DEFAULT state was set to DEFAULT+NOCAVITY+EXACTMIS.

```
-.4431056E-04  -.1182941E-05  .0000000  .0000000  .0000000  .2008143E-04
```

```
etall  1, NO =  2, NV =  7, INA = 304
*****
  I COEFFICIENT      ORDER  EXPONENTS
  NO =  2      NV =  7
0 -0.4431055933810007E-04  0 0 0 0 0 0 0
1  1.0000000000000000    1 0 0 0 0 0 0
-2  0.0000000000000000    0 0 0 0 0 0 0
```

```
etall  1, NO =  2, NV =  7, INA = 305
*****
  I COEFFICIENT      ORDER  EXPONENTS
  NO =  2      NV =  7
0 -0.1182941098476823E-05  0 0 0 0 0 0 0
1  1.0000000000000000    0 1 0 0 0 0 0
-2  0.0000000000000000    0 0 0 0 0 0 0
```

```
etall  1, NO =  2, NV =  7, INA = 306
*****
  I COEFFICIENT      ORDER  EXPONENTS
  NO =  2      NV =  7
1  1.0000000000000000    0 0 1 0 0 0 0
-1  0.0000000000000000    0 0 0 0 0 0 0
```

```
etall  1, NO =  2, NV =  7, INA = 307
*****
  I COEFFICIENT      ORDER  EXPONENTS
  NO =  2      NV =  7
1  1.0000000000000000    0 0 0 1 0 0 0
-1  0.0000000000000000    0 0 0 0 0 0 0
```

```
etall  1, NO =  2, NV =  7, INA = 308
*****
```

```

I COEFFICIENT      ORDER  EXPONENTS
NO =      2      NV =      7
1  1.0000000000000000      0 0 0 0 1 0 0
-1  0.0000000000000000      0 0 0 0 0 0 0

```

etall 1, NO = 2, NV = 7, INA = 309

\*\*\*\*\*

```

I COEFFICIENT      ORDER  EXPONENTS
NO =      2      NV =      7
1  1.0000000000000000      0 0 0 0 0 1 0
-1  0.0000000000000000      0 0 0 0 0 0 0

```

If we replace the above call to INIT with the following one

```
CALL INIT(DEFAULT+ONLY_4D,2,1,BERZ,ND2,NPARA)
```

then the result is the identical except for the longitudinal plane:

```

0.0000000000000000E+000
0.0000000000000000E+000

```

These variables are no longer Taylor series and NPARA=4. Finally

```
CALL INIT(DEFAULT+ONLY_4D+DELTA,2,1,BERZ,ND2,NPARA)
```

will give NPARA=5 and the result is

ETALL , NO = 2, NV = 6, INA = 244

\*\*\*\*\*

```

I COEFFICIENT      ORDER  EXPONENTS
NO =      2      NV =      6
1  1.0000000000000000      0 0 0 0 1 0
-1  .0000000000000000      0 0 0 0 0 0
0.0000000000000000E+000

```

## I.8 User Defined Integrators or Strange $s$ -Dependence

These user-defined methods can be used with KIND2(DKD2), KIND10(TEAPOT), KIND16(STREX), and the cavity KIND4(CAV4) which are built out of a two-step integrator as explained in Sects. K.4.2, K.4.9, K.4.12 and K.4.4. They can also be used in the user defined types USER1 and USER2 if so desired.

A general step of integration is given by

$$M\left(\frac{L}{N}\right) = \prod_{i=1}^{\text{SPN}(\text{Method})} \exp\left(: \text{SPD}(i, \text{Method}) \frac{L}{N} H_1 :\right) \exp\left(: \text{SPK}(i, \text{Method}) \frac{L}{N} H_2 :\right). \quad (34)$$

Let us do a simple example.

```

CALL MAKE_METHOD(10)      ! The arrays SPD and SPK are initialized
J=METHOD_1
DO I=1,10
  SPK(I,J)=1/9.DO
  SPD(I,J)=1/10.DO
ENDDO
SPK(10,J)=0.DO

```

In this example each step of integration cannot contain more than 10 operators. Of course it can have less and this is achieved by simply setting `SPN(METHOD_1)` to something less than 10. Here however, we created a method with 10 drifts and 9 kicks. This method is trivially quadratic due to reversal symmetry. The user can deallocate and start again within a run using the command “`KILL_METHOD.`” The logical `NEW_METHOD` is set to `TRUE` or `FALSE` indicating the presence or absence of user-defined methods.

These user-defined methods serve a dual purpose. Obviously one can try new integration methods without reprogramming PTC. It is well-known in the mathematical community that the Yoshida original schemes are not very efficient for Hamiltonians of the type  $T(p) + V(q)$ . One can now try other things.

Perhaps more importantly for large machines these user-defined schemes allow a person to create a tailored s-distribution of the multipole content. This is necessary, for example, when dealing with the IR quadrupoles of the LHC at CERN. Of course for that application one uses a single step of integration with a complex distribution of kicks described by the array `SPK`.

## J Sg\_0\_FITTED.f90

This is a fitted element not yet completed. It allows the magnetic field data of a magnet, in polar coordinates, to be entered in PTC and tracked. At the moment we simply interpolate between data points and perform non-symplectic tracking. It is also possible to track using a “pseudo-symplectic” algorithm. An approximate interpolated vector potential is constructed that reproduces the main effects of the field; a first order implicit method is then used. This vector potential is not differentiable at the grid points and thus the method is only locally symplectic but globally area preserving. The tracking algorithms are reliable for very large magnet apertures. Of course these magnets are implicitly “exact” in a kinematical sense.

This type of interpolated map does not permit the production of Taylor maps reliable beyond linear maps. Perhaps in the future more fancy algorithms will be implemented if there is a need.

## K Sh\_DEF\_KIND.f90

This is the module containing the basic tracking routines of PTC. This list is bound to increase as more standard magnets are included. This list does not include the user defined types USER1 and USER2 which are fully active elements of PTC if they have been fully and correctly implemented by the user.

### K.1 List of Magnet Types

First let us list these types (for the polymorphic type, simply add the letter “P” at the end, i.e., DRIFT1 becomes DRIFT1P). The integer at the end of the name has a mnemonic value (“1” in DRIFT1 reminds us that drifts are KIND1).

1. DRIFT1 : A drift, either approximate or exact. In canonical variables, it is the drift which carries the burden of the approximation rather than the multipole kick. Mathematically it makes no difference.
2. DKD2 : A second, fourth or sixth order integrator of the Drift-Kick-Drift type. It uses the expanded Hamiltonian; hence quadrupoles and bends become “linear elements.” It does not handle any exact elements anymore. Drift-kick-drift exact elements are handled either by type STREX or type TEAPOT.
3. KICKT3: A thin lens multipole kick, which should be avoided since it has no length.
4. CAV4: An RF cavity. It can have a length.
5. SOL5 is a solenoid. It can also contain multipoles. It is in the expanded Hamiltonian framework.
6. KTK: This is a second order Kick-Matrix-Kick integrator. The matrix is delta-dependent and the path length is quadratic in the transverse variables. This is the code SixTrack. It is slow because the matrix must be recomputed. It uses COSY-INFINITY techniques in the transverse plane (exponential of Lie matrices) and a special integrated exponential in the longitudinal plane. It supports EXACT\_MODEL only if the magnet is straight.
7. TKTF: This is almost your usual Matrix-Kick-Matrix. The matrix does not depend on delta. Therefore it is very fast. To be more precise, the splitting is Matrix-Correction-Kick-Correction-Matrix. The Hamiltonian of the correction is

$$\begin{aligned} H &= \frac{p_x^2 + p_y^2}{2(1 + \delta)} - \frac{p_x^2 + p_y^2}{2} \\ &= -\delta \frac{p_x^2 + p_y^2}{2} \end{aligned} \tag{35}$$

Of course, the delta-dependence is approximate in this case and depends on the number of integration steps. PTC supports 3 types of second order integration. The two more complex types are fourth and sixth order respectively in the action of the matrix on the kick and on the correction. This is achieved by a Simpson and a Bode type integrator. It supports EXACT\_MODEL only if the magnet is straight.

8. NSMI: This is a single normal multipole kick: it is here to mimic a SixTrack capability. Just like KICKT3, it should be avoided.
9. SSMI: This is a single skew multipole kick: it is here to mimic a SixTrack capability. Just like KICKT3, it should be avoided.
10. TEAPOT: This is a sector bend (normal entry bend) with cylindrical geometry. Imagine a perfect cyclotron and slice it like a wheel of Parmesan. This type of bend requires solving Maxwell’s equations in cylindrical coordinates. We call it TEAPOT in honor of the first symplectic integrator which did this magnet correctly for a constant field. The code Teapot however did not solve Maxwell’s equation for small bending radii: but the implementation was otherwise correctly done. One can add the fake wedges of MAD to this magnet. In PTC, we must emphasize, rectangular bends are not sector bends with wedges, although one can certainly create such objects with type TEAPOT.
11. MON: This is just a monitor in both planes. It is actually a drift that records x and y in the center.

12. ESEPTUM: This is the electric septum of MAD. It supports the exact and expanded Hamiltonian. To avoid crashing in the polymorphism part, functions such as  $\sinh(x)/x$  are evaluated by infinite Taylor series à la COSY-INFINITY near zero. This is not a major problem, hopefully, since septa are not numerous in a lattice almost by definition.
13. STREX: This a straight element of the exact variety. It can be a rectangular bend (i.e. Cartesian geometry) with the MAD style wedges at the end. PTC also supports a true rectangular bend for which the sum of the entrance and exit angle must equal the total bending angle.
14. SOLT: This is a solenoid handled with the techniques of type KTK. It is interesting since the source code shows how one can use the techniques developed in this paper for the SixTrack integrator on an arbitrary delta-dependent quadratic Hamiltonian.
15. USER1 and USER2: these are two user defined elements.

## K.2 Inheritance of Element Properties: Delegation in FORTRAN90

FORTRAN90 does not support inheritance directly as C++ does; what we will do here is perhaps closer to *composition* or *forward delegation* in JAVA. We needed to provide a sort of inheritance to manage the various similarities between the various kind of magnets. Here it is best to give a specific example. Consider an ELEMENTP (see Sect. C.1) and suppose that we know that this element is “quadrupole-like” in nature. This means that the variable ELP%BN(2) is allocated. However, looking at the list of possible elements, it appears that several elements are potential quadrupole-like elements: DKD2, KICKT3, SOL5, KTK, TKTF, NSMI, TEAPOT, SOLT, and STREX. In fact, some of these elements are really different models for the same physical type of object. Now suppose that we want to find the parametric dependence on the quadrupole component of ELP. How do we do it? Here it is in an expanded form (**not recommended, use POL\_BLOCK instead**):

```
NULLIFY(P)
CALL MOVE_TO(PSR,P,2)
P%MAGP%BN(2)%KIND=3      ! Here ELP is P%MAGP
P%MAGP%BN(2)%I=NPARA+1
CALL TRACK(PSR,Y,1,+DEFAULT)
```

The important point is that this syntax<sup>25</sup> will always work for an element which is quadrupole-like because we manage to make sure that BN(2) always points to the quadrupole-like variable of this element. Of course, the array BN must be allocated to order 2. This is not necessarily the case if the element is a dipole kick of KIND2 (**Again use POL\_BLOCK instead to avoid these issues**).

How is this achieved in PTC? Let us concentrate on the above example. Here the ELP%KIND was KIND2. This means that the magnet type was DKD2P. The definition of DKD2P is

```
TYPE DKD2P
  TYPE(MAGNET_CHART), POINTER:: P
  TYPE(REAL_8), POINTER :: L
  TYPE(REAL_8), DIMENSION(:), POINTER :: AN,BN      !Multipole component
  TYPE(REAL_8), POINTER:: FINT,HGAP                !FRINGE FUDGE FOR MAD
  TYPE(REAL_8), POINTER:: H1,H2
END TYPE DKD2P
```

We notice a few things. First DKD2P contains P of type MAGNET\_CHART. All magnets in the module S\_DEF\_KIND contain a P of type MAGNET\_CHART. It is defined as

```
TYPE MAGNET_CHART
  TYPE(MAGNET_FRAME), POINTER:: F
  INTEGER,POINTER :: CHARGE ! PROPAGATOR
  INTEGER,POINTER :: DIR   ! PROPAGATOR
  REAL(DP), POINTER :: LD,BO,LC !
  REAL(DP), POINTER :: TILTD ! INTERNAL FRAME
```

---

<sup>25</sup>The POL\_BLOCK type handles all the complex issues automatically. It is always better to change magnets through high level routines unless one knows exactly the inner works of the specific type of PTC under analysis.

```

REAL(DP), POINTER :: BETA0,GAMMA0I,GAMBET,POC
REAL(DP), DIMENSION(:), POINTER :: EDGE          ! INTERNAL FRAME
!
INTEGER, POINTER :: TOTALPATH                    !
LOGICAL, POINTER :: EXACT,RADIATION,NOCAVITY     ! STATE
LOGICAL, POINTER :: FRINGE,TIME                 !
!
INTEGER, POINTER :: METHOD,NST                   ! METHOD OF INTEGRATION 2,4,OR 6 YOSHIDA
INTEGER, POINTER :: NMUL                         ! NUMBER OF MULTIPOLE
END TYPE MAGNET_CHART

```

When PTC creates an element of any kind, it sets the pointer P in a magnet type such as DKD2P to point directly to the equivalent object in ELEMENTP. This communicates the object P of type MAGNET\_CHART to the ELEMENT. Ultimately this permits the connection with the quantity CHART of type CHART in the fibre.

In addition, AN, BN, L, B\_SOL, VOLT, etc.. also exist in ELEMENTP and are pointed at from a compatible magnet. This piece of code is not in the module S\_def\_kind but in the module S\_elements. This is because ELEMENT and ELEMENTP are not yet defined in S\_def\_kind. Nevertheless we can show the actual pointing operation for KIND2 done in the routine SETFAMILY(EL)<sup>26</sup>:

```

.
.
SELECT CASE(EL%KIND)
CASE(KIND1)
  if (.not. ASSOCIATED(EL%DO)) ALLOCATE(EL%DO)
  EL%DO%P=>EL%P
  EL%DO%L=>EL%L
CASE(KIND2)
  if (.not. ASSOCIATED(EL%K2)) ALLOCATE(EL%K2)
  EL%K2%P=>EL%P
  EL%K2%L=>EL%L
  IF(EL%P%NMUL==0) CALL ZERO_ANBN(EL,1)
  EL%K2%AN=>EL%AN
  EL%K2%BN=>EL%BN
  EL%K2%FINT=>EL%FINT
  EL%K2%HGAP=>EL%HGAP          EL%K2%H1=>EL%H1
  EL%K2%H2=>EL%H2
.
.

```

The first line `EL%K2%P=>EL%P` connects the MAGNET\_CHART in EL and the magnet under consideration; notice that K2 is the object of type DKD2P embedded in ELEMENTP. (See Sect. C.1) The beauty of this construction is that the code using the BN(2) of the element # 2 does not care about the model DKD2P. All other things being equal we could have used TKTFP for example. Of course if # 2 is a marker or a cavity, the user gets either a PTC exception<sup>27</sup>, a system exception due to a crash, or just plain aberrant behavior.

### K.3 Some Maintenance Routines: Zeroing, ALLOC and KILL

The maintenance routines of this module are almost of no concern to the user. Of course if someone wanted to add a new type of magnet, then paying close attention to the most complex case in this module would be very useful. So we will list them.

#### K.3.1 The (=) Assignment

The routines ZEROR\_KTK, ZEROP\_KTK, ZEROR\_TKT7, ZEROP\_TKT7, ZEROR\_TEAPOT, ZEROP\_TEAPOT, ZEROR\_MON, ZEROP\_MON, ZEROR\_STREX, ZEROP\_STREX, ZEROR\_SOL, ZEROP\_SOL, ZEROR\_CAV4, and ZEROP\_CAV4 permit the nullification of the pointers of KTK, KTKP, TKT7, TKT7P, TEAPOT,

<sup>26</sup>Here EL is actually ELEMENTP: it is a dummy of type ELEMENTP in the code.

<sup>27</sup>If Etienne Forest can stimulate himself to program the exceptions.

TEAPOTP, MON, MONP, STREX, STREXP, SOLT,SOLTP, CAV4, and CAV4P respectively using the syntax magnet=0. If the assignment magnet=-1 is used, then an existing magnet is destroyed. Polymorphic variables are first deallocated in TKT7P (FPP function KILL) and then the pointers are also deallocated. In KTKP and SOLTP (SixTrack style integrators) the polymorphic variables are deallocated locally before exiting the tracking routine since the delta-dependent matrix is constantly recomputed. The reader should look at the code. These assignments are used in the module S\_DEF\_ELEMENT because equivalent operations exist for ELEMENT and ELEMENTP.

### K.3.2 More Interfaces for ALLOC and KILL

Again we have eight routines corresponding to the magnets of Sect. K.3.1: ALLOCKTK, ALLOCTKT7, ALLOCSOL, ALLOCTEAPOT, KILLKTK, KILLTKT7, KILLSOL, and KILLTEAPOT. They perform the allocation and killing for the polymorphic variables. They are used constantly for a KTKP or SOLTP magnet for reasons explained above. For a TKT7 magnet, since the matrix is constant, the routines are only used if parametric dependence on BN(2) is required; however they are used in the module S\_DEF\_ELEMENT when a TKT7 is created or destroyed since its polymorphic matrix is global.

The reader will perhaps conclude that this is a lot of garbage: indeed maintenance is not pleasant in FORTRAN90 because we do not have automatic constructors and destructors. All of this spells “mess” for the programmer. On the other hand, as we explained before, it is not clear that the present approach is not ultimately the best anyway: forward delegation rather than multiple inheritance or simply visitor functions.

## K.4 More about the Magnets

Of course the best manual is the code itself (it cannot lie) but it is not too eloquent, like a village idiot!

What follows is hardly better, but here it is anyway. First we remind the reader of the connection between  $\delta = (p - p_0)/p_0$  and  $\delta_E = (E - E_0)/(p_0 c)$ :

$$(1 + \delta)^2 = 1 + \frac{2\delta_E}{\beta_0} + \delta_E^2. \quad (36)$$

The time of flight  $cT$  is canonically conjugate to  $-\delta_E$  while path length is conjugate to  $-\delta$ .

In what follows we emphasize the  $\delta$ -dependent Hamiltonians for the magnets out of pure laziness; these produce path length rather than time of flight.

### K.4.1 DRIFT1: Drift

Well this is a drift! Drifts come in two flavors: exact and expanded. The exact Hamiltonian is given by the formula:

$$H = -\sqrt{(1 + \delta)^2 - p_x^2 - p_y^2}. \quad (37)$$

The expanded Hamiltonian is

$$H = \frac{p_x^2 + p_y^2}{2(1 + \delta)} - \delta. \quad (38)$$

Incidentally, the expression in Equation (38) gives us

$$(x', y') = \frac{1}{(1 + \delta)}(p_x, p_y), \quad (39)$$

which, in turn, leads to a multipole kick of the variables  $(x', y')$  proportional to  $1/(1 + \delta)$ : this is the usual non-canonical result.

### K.4.2 DKD2: Drift-Kick-Drift Element

This is the “classic” TRACYII-SixTrack Hamiltonian in the non-exact expanded mode. The Hamiltonian of the body of such a magnet is given by

$$H = \underbrace{\frac{p_x^2 + p_y^2}{2(1 + \delta)} - \delta}_{H_1} + \underbrace{-\frac{x\delta}{\rho_d} + \frac{x^2}{2\rho_d^2}}_{H_2} + V(x, y)$$



$$V(x, y) = \text{Re} \left( \sum_{n=1}^{\infty} \frac{(i a_n + b_n)}{n} (x + iy)^n \right). \quad (40)$$

PTC can integrate this Hamiltonian using the second, fourth, and sixth order splitting method of Yoshida. The Yoshida method takes the fundamental second order method and builds higher order schemes using it. The second order scheme is simply:

$$S_2(dz) = \exp \left( : -\frac{dz}{2} H_1 : \right) \exp \left( : -dz H_2 : \right) \exp \left( : -\frac{dz}{2} H_1 : \right). \quad (41)$$

Higher order schemes are based on this fundamental second order method. For example, the fourth order method, which is equivalent here to Ruth's old integrator, is simply:

$$\begin{aligned} S_4(dz) &= S_2(\xi_0 dz) S_2(\xi_1 dz) S_2(\xi_0 dz) \\ \xi_0 &= \frac{1}{2 - \sqrt[3]{2}} \\ \xi_1 &= -\frac{\sqrt[3]{2}}{2 - \sqrt[3]{2}} \end{aligned} \quad (42)$$

It is possible to incorporate radiation correctly in the Yoshida hierarchy of integrators. However in PTC radiation is incorporated in the lowest order because it is a small effect.

The entire map for the magnet, in the expanded model, is

$$M(L) = R_{xy}^{-1} \circ Q(\theta_2) \circ F_2^{out} \circ B(L) \circ F_2^{in} \circ Q(\theta_1) \circ R_{xy} \quad (43)$$

where  $R_{xy}$  is a layout rotation<sup>28</sup> of angle EL%TILTD,  $Q(\theta_i)$  mimics the entrance and exit angle<sup>29</sup> of an arbitrary bend.  $F_2^{in/out}$  is the quadrupole fringe field from any  $b_2$  or  $a_2$  found in the bend. PTC uses a symplectic rendition of the famous Lee-Whiting formula; the same as used by SAD. Finally  $B(L)$  is the body of the magnet integrated using Yoshida's method.

What about EXACT\_MODEL? In the most recent version of PTC type DKD2 handles only the expanded Hamiltonian elements. The exact integration are handled by types STREX, TEAPOT, KTK, and TKTF. STREX handles straight elements of all types. TEAPOT handles bends of "cyclotronic" symmetry, i.e., mostly invariant along the ideal trajectory, and finally TKTF and KTK handle straight elements without any ideal bending using a different split from STREX, namely the kick-matrix splits.

**N.B.** For all the multipole-like elements which can have a dipole geometry, we added the parameters FINT, HGAP, H1, and H2 of MAD. The implementation of the (FINT,HGAP) and the (H1,H2) effects are explained in two technical notes which will be published with this manual. In the original SLAC-75 report, these expressions are either quoted as results of private derivations or obscure notes. Since we would like PTC and eventually MAD-X to be unambiguous in their modeling and open to criticisms, we went through the trouble of re-deriving and analyzing the sources of these effects.

#### K.4.3 KICKT3: Thin Multipole Kick

This element is a pure thin multipole kick. It is not recommended at all. It applies the map

$$\exp \left( : -\text{Re} \left( \sum_{n=1}^{\infty} \frac{(i a_n + b_n)}{n} (x + iy)^n \right) : \right) \quad (44)$$

to a ray. Such an element cannot radiate.

#### K.4.4 CAV4: RF Cavity

This element can represent a thin or thick cavity. The thin cavity kick is represented by the expression:

<sup>28</sup>This is used to indicate that the bend is perhaps a vertical bend. Theoretically it can be any angle. It is NOT an error tilt.

<sup>29</sup>This thin quadrupole trick is not "physical." It is an accidental feature of a first order correct calculation. The focusing in the horizontal plane is purely dynamical. The focusing in the vertical plane is a result of applying Maxwell's equations. The effects are equal and of opposite sign, what are the deep reasons? Ask the devil!

```

IF(EL%NOCAVITY) RETURN
EL%DELTA_E=X(5)
X(5)=X(5)-EL%DIR*EL%CHARGE*EL%VOLT*1.D-3*DSIN(TWOPI*EL%FREQ*X(6)/CLIGHT+EL%PHAS)/EL%POC
EL%DELTA_E=(X(5)-EL%DELTA_E)*EL%POC

```

The variable `EL%DELTA_E` records the change of energy at a cavity. It is useful during radiation as it gives us the energy loss/gain. Notice that during radiation the new synchronous orbit will be found and the fish structure of the bucket will appear naturally as an output of the simulation; as it does in a real ring. Thus `EL%PHAS` is a design input phase, not a phase resulting from radiation. The units for `EL%VOLT` are MV and those of `EL%POC` are GeV.

Recently, we also added a thick cavity which can be integrated using the same integration methods as `KIND2`. This is controlled by the parameter `THIN` of type `CAV4`. In addition we added the experimental fringe field which is not yet documented nor is it debugged. (There are issues of cavity wavelength and wave types which complicate matters.)

#### K.4.5 SOL5: The Combined Function Solenoid

This is an ideal solenoid in the expanded Hamiltonian framework. The Hamiltonian is given by:

$$\begin{aligned}
H &= \frac{1}{2(1+\delta)} \left\{ \left( p_x + \frac{b_s}{2} y \right)^2 + \left( p_y - \frac{b_s}{2} x \right)^2 \right\} - \delta + V(x, y) \\
&= \underbrace{\frac{\mathbf{p}^2}{2(1+\delta)}}_{H_1} - \underbrace{\delta}_{H_2} - \underbrace{\frac{b_s L_z}{2(1+\delta)}}_{H_3} + \underbrace{\frac{b_s^2 \mathbf{r}^2}{8(1+\delta)}}_{H_4} + \underbrace{V(x, y)}_{H_4}.
\end{aligned} \tag{45}$$

The term  $V(x, y)$  is a regular multipole term. The term in  $L_z$  is a delta-dependent rotation around the z-axis. It commutes with the rest of the solenoid but not with  $V$ . Therefore this element can be integrated using a four-term Yoshida scheme. This is what PTC does. It supports second, fourth, and sixth order Yoshida integrators.

`SOL5` does not support the `EXACT_MODEL` option except in the drift part. In that case, the calls to the drift in the integrator use the full drift Hamiltonian.

#### K.4.6 KTK: Delta-dependent Quadratic Hamiltonian and Multipole Kicks

This is a new and very interesting method: it is the thick element of SixTrack. It support the straight exact element in the same way as type `TKTF`, so please look at Sect. K.4.7 for an explanation of the `EXACT_MODEL` option. This new method solves the expanded Hamiltonian of Equation (40) using a different splitting method. We can rewrite Equation (40) as

$$H = \underbrace{\frac{p_x^2 + p_y^2}{2(1+\delta)} - \frac{x\delta}{\rho_d} + \frac{x^2}{2\rho_d^2}}_{H_1} - \underbrace{\delta + \frac{b_2}{2}(x^2 - y^2)}_{H_2} + \underbrace{V - \frac{b_2}{2}(x^2 - y^2)}_{H_2} + \underbrace{C_{\text{exact}}}_{H_3}. \tag{46}$$

Of course this is the same Hamiltonian as in Equation (40) but a different<sup>30</sup> “model” for the integration. The quantity  $C_{\text{exact}}$  is the missing part of the square root for a drift that is needed in the exact option for a straight element. This enters as a third term of the integrator split. Unlike type `TKTF` discussed in Sect. K.4.7, this term is only used in the exact option.

It should be obvious that  $H_1$  can be solved exactly in the transverse plane. It is a delta-dependent matrix. This is at the center of codes such as `RACETRACK`, `PATRICIA` and `TRACY`. However Ripken and Schmidt noticed that this Hamiltonian admits a quadratic path length as its exact solution in the longitudinal plane. It can be interpreted as the expansion of the standard Pythagoras formula in cylindrical coordinates for the path length. However Hamiltonian theory teaches us here that it is pointless and even incorrect (it violates the symplectic condition) to go beyond the second order term.

<sup>30</sup>Actually I am a proponent of the Talman way of looking at modelling. In other words I view the method of integration as part of the model. All accelerator physicists do this to some extent without acknowledging it. Thus, in that sense, Equations (40) and (46) are not describing the same model.

Ripken and Schmidt painfully derived all the quadratic path lengths for the basic magnets. When analytic formulas are used, one must distinguish between focusing quadrupoles, defocusing quadrupoles, various kinds of bends, etc.. this is because the solution in terms of real functions is not possible without branching. For example a focusing element may require a cosine while the defocusing one requires a hyperbolic cosine. All of this is a mess, but it was religiously implemented by them. However branching is a killer for polymorphic parametric dependence on the quadrupole strength.

Here we adopted a novel solution. First, the transverse dynamics can be evaluated exactly for a given value of delta using the exponential of the Lie matrix corresponding to  $H_1$ : a mini-COSY-INFINITY is written just for this purpose.

The longitudinal is a bit trickier. Rather than having the exponential of a Lie operator acting on the identity map (inexact COSY-INFINITY technique in this particular case), one can actually write the path length as the “integrated exponential” ( $\frac{e^x-1}{x}$ ) of a Lie matrix acting on the space of quadratic polynomials in the transverse variables. This problem is solvable and of finite dimension. It should be pointed out that functions like the integrated exponential and  $\sin(x)/x$  appear in analytical formulas as well and cannot be treated directly by TPSA techniques. This is another problem in addition to branching.

By actually computing these exponentials using the formal Taylor series, PTC can get all the parametric dependence so elusive in the Ripken-Schmidt approach. It is however slow since the transverse matrix and the quadratic polynomials must be recomputed<sup>31</sup> all the time.

The integrator for the body is obtained using the splitting

$$\begin{aligned} \mathcal{S}_2 = & \exp\left(: -\frac{dz}{2} H_2 :\right) \exp\left(: -\frac{dz}{2} C_{\text{exact}} :\right) \exp\left(: -dz H_1 :\right) \\ & \times \exp\left(: -\frac{dz}{2} C_{\text{exact}} :\right) \exp\left(: -\frac{dz}{2} H_2 :\right) . \end{aligned} \quad (47)$$

This integration method gets the so-called linear elements exactly for the expanded Hamiltonian. The total map for the magnet (body, fringe effects, etc.) is given also by Equation (43).

#### K.4.7 TKTF : Quadratic Hamiltonian, Delta-Corrections, and Multipole Kicks

In the last section we discussed the exact treatment of the quadratic delta-dependent Hamiltonian. It is slow because recomputation is necessary whenever delta changes. To alleviate this problem, consider yet another split of  $H$ .

$$\begin{aligned} H = & \underbrace{\frac{p_x^2 + p_y^2}{2} - \frac{x\delta}{\rho_d} + \frac{x^2}{2\rho_d^2} + \frac{b_2}{2}(x^2 - y^2)}_{H_1} + \underbrace{\left\{ -\delta \frac{p_x^2 + p_y^2}{2(1+\delta)} - \delta + C_{\text{exact}} \right\}}_{H_2} \\ & + \underbrace{V - \frac{b_2}{2}(x^2 - y^2)}_{H_3} . \end{aligned} \quad (48)$$

This option supports EXACT\_MODEL if the design curvature is zero ( $\rho_d = 0$ ); in such a case the term  $C_{\text{exact}}$  in Equation (48) supplements the missing part of the drift, i.e., of the square root.

PTC supports three methods of integration for this element. First it permits the usual second order integrator implemented as

$$\exp\left(: -\frac{dz}{2} H_1 :\right) \exp\left(: -\frac{dz}{2} H_2 :\right) \exp\left(: -dz H_3 :\right) \exp\left(: -\frac{dz}{2} H_2 :\right) \exp\left(: -\frac{dz}{2} H_1 :\right) \quad (49)$$

The higher order methods are not Yoshida integrators but biased methods. What is a biased method? Consider the above Hamiltonian with smallness parameters  $\alpha$ ,  $\beta$ , and  $\varepsilon$ :

$$H = \alpha\varepsilon H_1 + \beta\varepsilon (H_2 + H_3) \quad (50)$$

A bias integrator seeks an approximate solution  $B_{n,k}$  such that

$$B(n, k) = \exp(: ds H :) + O(\varepsilon^{n+1}), \quad (51)$$

---

<sup>31</sup>Actually it must be recomputed if delta changes. However PTC recomputes all the time to avoid horrible logic. Anyway magnet type TKTF will resolve these issues.

but the error terms in  $\varepsilon^{n+1}$  are not evenly distributed between  $\alpha$  and  $\beta$ . In our case, we have:

$$\text{Error} = O_{k+1}(\beta\alpha^k) + O_{n+1}(\alpha\beta^n) + \dots \quad (52)$$

Let us list the three methods we have. They are of type  $B(2, 2)$ ,  $B(2, 4)$ , and  $B(2, 6)$ . The first method is not biased since  $n = k$ , in fact it is just our second order split and it is given by (49). The next method is based on a third order Simpson (1,4,1) rule and the error is given by

$$\text{Error} = O_5(\beta\alpha^4) + O_3(\alpha\beta^2) + \dots \quad (53)$$

This means that the “phase advance” effect of the linear map on  $H_2 + H_3$  is computed to fourth order locally. The cross-terms in  $H_2$  and  $H_3$  alone are only accurate to third order locally (second order integrator).

Finally, we also have  $B(2, 6)$ ; this is a fifth order Bode (7,32,12,32,7) scheme. The phase advance effect is locally sixth order.

These methods are quite efficient in accelerators. In PTC they are triggered by `EL%P%METHOD=2,4, or 6`. All of them support radiation. Once more the total map for the magnet (body, fringe effects, etc.) is given by Equation (43).

#### K.4.8 NSMI and SSMI: Single Multipole Thin Kicks

These are SixTrack leftovers. Like the magnet KICKT3 these objects have no length and cannot radiate. We do not recommend their use.

#### K.4.9 TEAPOT: The Exact Sector Bend

This is a normal entry bend with cylindrical symmetry. This kind of bend is the pie slice of a perfect cyclotron, that is to say, a system with rotational symmetry around the  $y$ -axis. **Although it is possible to create a parallel face bend with type TEAPOT, by using wedges, PTC makes a distinction between a rectangular bend created with wedges and a true Cartesian bend (handled by STREX, Sect. K.4.12).** If you add wedges to a bend of type TEAPOT, you are saying that the bend, in its center, is essentially invariant along the “ $\phi$ ” direction and thus obeys Maxwell’s equation in polar coordinates. Conversely, if a normal entry bend is created by adding wedges to a true “RBEND” of type STREX, you are saying that the center of the bend is translationally invariant and thus obeys Maxwell’s equation in Cartesian coordinates. **This distinction is unknown to MAD. More can be found on this topic in Sect. K.4.12.**

For the record, type TEAPOT is defined as

```

TYPE TEAPOT
  TYPE(MAGNET_CHART), POINTER:: P
  REAL(DP), POINTER :: L
  REAL(DP), DIMENSION(:), POINTER :: AN,BN          !MULTIPOLE COMPONENT
  REAL(DP), DIMENSION(:), POINTER :: BF_X,BF_Y      ! B FIELD POLYNOMIAL
  LOGICAL, POINTER :: DRIFTKICK                    ! SPLIT FLAG
  REAL(DP), POINTER:: FINT,HGAP                    !FRINGE FUDGE FOR MAD
  REAL(DP), POINTER:: H1,H2                        !FRINGE FUDGE FOR MAD
END TYPE TEAPOT

```

As we stated earlier, AN and BN are not used directly but through BF\_X and BF\_Y. This is because Maxwell’s equations are harder to solve in a cylindrical geometry than in a Cartesian one. In addition, we have the flag DRIFTKICK which relates to the two possible splits of the Hamiltonian which we now explain.

Let us write the Hamiltonian for the body of the TEAPOT magnet with two different splits:

$$\begin{aligned}
H &= \underbrace{-\left(1 + \frac{x}{\rho_d}\right) \sqrt{(1 + \delta)^2 - p_x^2 + p_y^2}}_{T_1} + \underbrace{V(x, y; \rho_d^{-1})}_{T_2} \leftarrow \text{The Teapot Code Split} \\
&= \underbrace{-\left(1 + \frac{x}{\rho_d}\right) \sqrt{(1 + \delta)^2 - p_x^2 + p_y^2} + b_1 \left(x + \frac{x^2}{2\rho_d}\right)}_{H_1} + \underbrace{V(x, y; \rho_d^{-1}) - b_1 \left(x + \frac{x^2}{2\rho_d}\right)}_{H_2}. \quad (54)
\end{aligned}$$

The Teapot code (DRIFTKICK=.TRUE.) split uses the standard ROT\_XZ (drift in polar coordinates called SPROT in S\_DEF\_KIND) and a multipole kick. In PTC the ideal orbit and the computed orbit do not match perfectly if this is used unless BN(1) is adjusted and differs slightly from EL%P%B0. With the Teapot split it is possible to even do a straight element; of course this is not recommended. The other split (DRIFTKICK=.FALSE.) is an “exact bend-multipole” split. The horizontal bend is handled exactly. Notice that it may even differ from the ideal bend since there is no assumption that  $b_1 = \rho_d$ .

PTC must compute a template from which it extracts the B-field from the multipole components. The potential  $V$  in the limit of  $\rho_d^{-1}$  going to zero becomes the regular harmonic expansion of Equation (40). However for finite values of  $\rho_d^{-1}$ , the potential  $V$  obeys the equation:

$$\left\{ (\rho_d + x) \left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} - \frac{\partial}{\partial x} \right) - \frac{\partial}{\partial x} \right\} V = 0 . \quad (55)$$

The field, scaled by  $p_0/q$  is then given by

$$\begin{aligned} b_x &= -\frac{1}{\rho_d + x} \frac{\partial}{\partial x} V \\ b_y &= \frac{1}{\rho_d + x} \frac{\partial}{\partial y} V \\ b_s &= 0. \end{aligned}$$

Equation (55) is solved in the module ANBN to order SECTOR\_NMUL. It is solved by an iterative procedure for  $\rho_d^{-1} = 1$ . In fact, the beginning of the iteration is the usual harmonic solution for straight elements. The solution for an arbitrary  $\rho_d$  can be regained due to the fact that each successive iterate depends on one extra power of  $\rho_d^{-1}$ . The ugly details can be found in reference [5], Interlude XIV.

In module ANBN, the solution must be tied to a physical definition, a measurement so to speak. Thus we define the usual  $a_n$  and  $b_n$  as follows:

$$\begin{aligned} b_x^{y=0} &= \sum_{n=1}^{\text{Sector\_Nmul}} a_n x^{n-1} \\ b_y^{y=0} &= \sum_{n=1}^{\text{Sector\_Nmul}} b_n x^{n-1}. \end{aligned} \quad (56)$$

Then, on the basis of Equation (56),  $V$  is computed to order SECTOR\_NMUL.

Module ANBN provides the template. Each time the multipole components of an exact sector bend are changed, the power series for  $b_x$  and  $b_y$  are recomputed (BF\_X,BF\_Y).

The symplectic integration proceeds in the usual two-terms Yoshida for the two possible splits. Finally the full map in PTC for this element is:

$$M(L) = R_{xy}^{-1} \circ F_1^{out} \circ F_2^{out} \circ B(L) \circ F_2^{in} \circ F_1^{in} \circ R_{xy} \quad (57)$$

This is similar to the full map for the exact parallel face bend in Equation (60).

Again we repeat, it is possible to add a wedge to this element. The wedge is an exact sector bend of field BN(1) which modifies the geometry so that the entrance and exit angles can be different. This is described in [5] and it is compatible with MAD. This gymnastic is barely PTC compliant.

#### K.4.10 MON : Monitors

This type is just a drift in the middle of which is read an  $x$  and a  $y$  position. The definition of the element is just:

```

TYPE MON
  TYPE(MAGNET_CHART), POINTER :: P
  REAL(DP) , POINTER :: L ! MONITOR AND INSTRUMENT OF MAD
  REAL(DP), POINTER :: X,Y
END TYPE MON

```

#### K.4.11 ESEPTUM: Electric Septum

This is an electric septum defined in PTC as

```

TYPE ESEPTUM
!
TYPE(MAGNET_CHART), POINTER :: P
REAL(DP) , POINTER :: L
REAL(DP) , POINTER :: VOLT ! VOLTAGE IN MV/M
END TYPE ESEPTUM

```

Here we give exceptionally the time rather than the pathlength Hamiltonian, so  $p_t$  is actually the variable X(5) of PTC:

$$H = -\sqrt{\left(\frac{1}{\beta_0} + p_t + ky\right)^2 - \frac{1}{\beta_0^2 \gamma_0^2} - p_x^2 - p_y^2} \quad \text{where } p_t = \frac{E - E_0}{p_0 c} \quad (58)$$

Obviously, one sets  $\beta_0 = 1$  to obtain the pathlength  $H$ . In PTC this magnet, as in MAD, has a straight geometry. The codes supports the exact and approximate option. In both cases the map is computed exactly since the Hamiltonian in Equation (58) is exactly solvable.

#### K.4.12 STREX: The Exact Generic Rectangular Bend

This type represents a magnet with a Cartesian internal geometry that can be potentially a dipole with curved layout geometry. MAD-8,9,X do not differentiate between rectangular and sector bends. One goes from one to the other by wedges sandwiching a sector bend (type TEAPOT in PTC). Although this is also permitted by PTC, it is actually discouraged. PTC forces you to decide on the nature of the symmetries in the center of the magnet. This is explained in detail in [5]. Here we will show with one pictorial example, in Figure 19, the difference between the two type of bends.

Type STREX is defined as

```

TYPE STREX
TYPE(MAGNET_CHART), POINTER:: P
REAL(DP), POINTER :: L
REAL(DP), DIMENSION(:), POINTER :: AN,BN !MULTIPOLE COMPONENT
LOGICAL, POINTER :: DRIFTKICK,LIKEMAD
REAL(DP), POINTER:: FINT,HGAP !FRINGE FUDGE FOR MAD
REAL(DP), POINTER:: H1,H2 !BOUNDARY FUDGES FROM MAD
END TYPE STREX

```

The flag DRIFTKICK also appears in type STREX and the meaning is the same as in type TEAPOT. In the case of STREX, the symmetry of the body is Cartesian, the Hamiltonian for the body is always:

$$\begin{aligned}
H &= \underbrace{-\sqrt{(1+\delta)^2 - p_x^2 - p_y^2}}_{T_1} + \underbrace{V(x,y)}_{T_2} \leftarrow \text{DRIFTKICK} = \text{true} \\
&= \underbrace{-\sqrt{(1+\delta)^2 - p_x^2 - p_y^2 + b_1 x}}_{H_1} + \underbrace{V(x,y) - b_1 x}_{H_2} \leftarrow \text{DRIFTKICK} = \text{false} \\
V(x,y) &= \text{Re} \left( \sum_{n=1}^{\infty} \frac{(i a_n + b_n)}{n} (x + iy)^n \right) . \quad (59)
\end{aligned}$$

The expressions for the multipole components are correct since this bend has Cartesian geometry by assumption. If the ideal bending angle is zero, this is just a straight element done with the exact body Hamiltonian. The entire Yoshida apparatus applies to it as well.

To understand the difference between MAD and PTC with regard to true parallel face bends, it is best to look at a simple example. In Figure 19, the two bends displayed have an entrance angle  $\varepsilon_1 = 0$  and an exit angle  $\varepsilon_2 = \alpha$  where  $\alpha$  is the total bending angle. For the first case, LIKEMAD=.TRUE., PTC constructs this bend using a wedge glued to a true Cartesian bend. In PTC this can be done with type TEAPOT, in which case the glueing is done on a sector bend or on type STREX, as it is the case here. These two

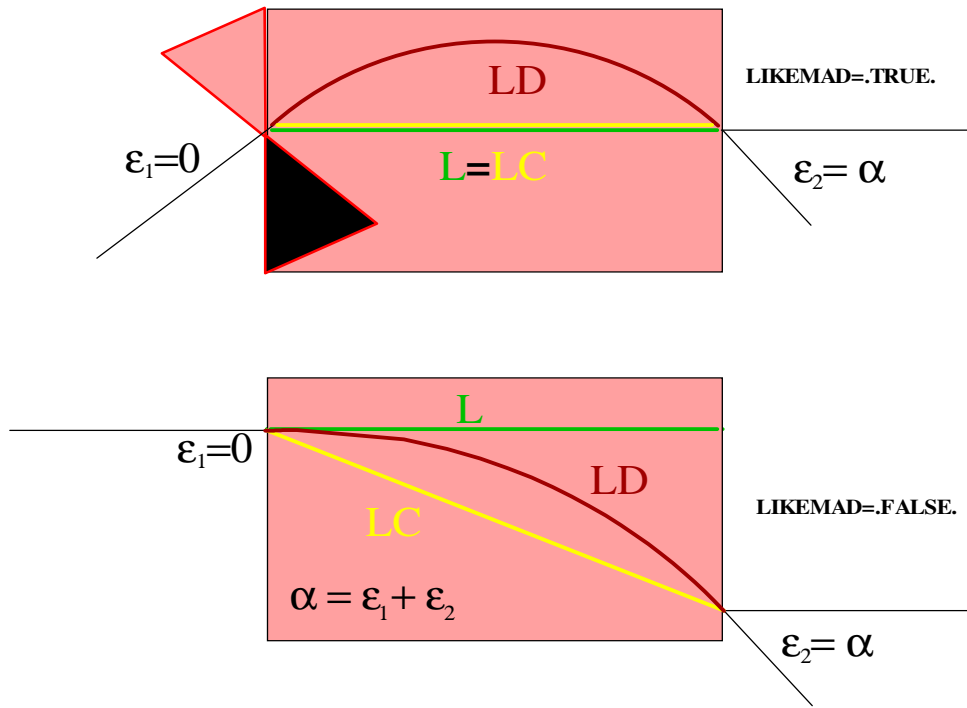


Figure 19: Two Different Bends constructed from the RBEND by PTC

approaches are not equivalent if there are multipole components in the body of the bend simply because  $\rho_d$ -dependent Maxwellian effects must be included in type TEAPOT.

If LIKEMAD=.FALSE., then PTC truly constructs a parallel face bend. One should notice that for this bend the sum of the entrance and exit angles must always be equal to the total bending angle. Moreover the case LIKEMAD=.FALSE. is very interesting since it is a rare case of  $L \neq L_c \neq L_d$ ; it also shows very clearly that  $L$  is truly an internal variable whose meaning is detail-dependent while  $L_c$  and  $L_d$  are layout variables describing the desired geometry of the element. The total map for the magnet is

$$M(L) = R_{xy}^{-1} \circ R_{xz}(\epsilon_1) \circ F_1^{out} \circ F_2^{out} \circ B(L) \circ F_2^{in} \circ F_1^{in} \circ R_{xz}(\epsilon_2) \circ R_{xy} \quad (60)$$

$R_{xz}$  is a dynamical rotation: the famous PROT of Dragt.  $F_1^{in/out}$  is the fringe field due to  $b_1$ , which acts mostly in the vertical plane and  $F_2^{in/out}$  is the quadrupole fringe field from any  $b_2$  or  $a_2$ . Here we see explicitly that in a more exact treatment of a bend, the thin quadrupole trick does not appear.

#### K.4.13 SOLT: Delta-dependent Quadratic Hamiltonian with a Solenoidal Term and Multipole Kicks

This type is very similar to type SOL5 except that the quadratic part, including the erect quadrupole component, is solved using the techniques of Sect. K.4.6. The reader should convince himself that it is not that easy to write a matrix integrator of the type TKTF for a solenoid. We will not discuss this type any further at this stage.

## L Si\_DEF\_ELEMENT.f90

We are now in the core of PTC. This is where ELEMENT and ELEMENTP are defined and managed. While the layout and fibre types are important, they would fall flat on their face if a standard element was not properly defined.

### L.1 Constants and Internal Routines of Si\_DEF\_ELEMENT.f90

There are a few routines that perform hidden operations.

#### L.1.1 ZERO\_ANBN

This routines initializes AN and BN arrays if the syntax EL=N is used, see L.4.1.

#### L.1.2 ALWAYS\_EXACTMIS and ALWAYS\_FRINGE

These two variables can be changed by the user during execution. ALWAYS\_EXACTMIS ensures that exact Euclidean operators are being used in the misalignments. ALWAYS\_FRINGE forces quadrupole fringe fields (Lee-Whiting formulas) to be always turned on.

#### L.1.3 The Logical FEED\_P0C

This logical is used in connection with type WORK. Please look at Sect. L.4.3.

#### L.1.4 BERZ and ETIENNE

What can be simpler: BERZ=.TRUE. and ETIENNE=.NOT.BERZ! Joke aside this is the convenient flag used when calling the TPSA package with the interface INIT (see Sect. I.7). BERZ refers to the LBNL version of Martin Berz's "DA-Package." ETIENNE refers to an experimental version of a similar package. ETIENNE (newda.f90) was written for low order and many variables. Attempts to speed it up failed. So please avoid ETIENNE<sup>32</sup> for the moment.

#### L.1.5 MOD\_N(I,J)

This is the regular FORTRAN MOD(I,J) routine except that it returns J instead of 0 when I is a multiple of J. It is used in the layout tracking routines.

#### L.1.6 RESET31(ELP), TPSAFIT(LNV), and SET\_TPSAFIT

The real allocatable targeted array "TPSAFIT(LNV)" is used to store the changes to polymorphic knobs. SET\_TPSAFIT is used by type POL\_BLOCK to distinguish between the assignments of knobs in FPP versus the setting of the knobs in PTC using the data in TPSAFIT(LNV). It is a global targeted logical. Both TPSAFIT(LNV) and SET\_TPSAFIT are used if USE\_TPSAFIT is true, which it is by default. The user has the flexibility to change that, but it is not advised unless some TPSA and ordinary fitting is used simultaneously.

The subroutine RESET31 removes all parametric polymorphic variables, i.e., KIND=3 from ELP. Everything returns to KIND=1. It is used in the routine KILL\_PARA on a LAYOUT. KILL\_PARA(PSR) removes all parametric knobs from the PSR lattice; see Sect. O.1.3.

#### L.1.7 VERBOSE and GEN

These are two logicals which should not be changed from their defaults. VERBOSE is a printing flag set to false. GEN is a more important flag restricting the computation of the matrix part of KIND7 (type TKTF) to cases when it is absolutely necessary. It avoids unnecessary calls to the COSY-INFINITY like routines of type TKTF. Please do not touch it.

---

<sup>32</sup>The moral of the story is this: if the Taylor series is sparse, it is hard to beat the Berz. Parole d'Etienne!



## L.2 Types whose functionality is defined in Si\_DEF\_ELEMENT.f90

Of course this module defines ELEMENT and ELEMENTP. We will assume that this is known at this stage (please see Sect. C.1). In addition, the following types are crucial in module S\_DEF\_ELEMENT:

1. The type MUL\_BLOCK: this permits the easy assignment and retrieval of multipole components to and from an ELEMENT(P).

```
TYPE MUL_BLOCK
! stuff for setting multipole
  REAL(DP) AN(NMAX),BN(NMAX)
  INTEGER NMUL,NATURAL,ADD
END TYPE MUL_BLOCK
```

2. Type WORK: this type can fetch and assign energy like variables to an ELEMENT(P). The definition is

```
TYPE WORK
  REAL(DP) BETA0,ENERGY,KINETIC,POC,BRHO,GAMMAOI,GAMBET
  REAL(DP) MASS
  LOGICAL RESCALE
END TYPE WORK
```

It is actually defined in the module S\_status, but it is used primarily in the module S\_DEF\_ELEMENT.

3. Finally type POL\_BLOCK: it allows the creation of knobs in an ELEMENTP. It is actually defined in the module S\_status as well, but it is used primarily in the module S\_DEF\_ELEMENT. It also allows the scanning of a family (ELP%NAME) and family members (EL%VORNAME) in a layout. Finally it even permits the setting of the knobs based on the global array TPSAFIT(LNV). It is a powerful type but a little complex. It facilitates the use of polymorphism.

```
TYPE POL_BLOCK
  CHARACTER*16 NAME,VORNAME
  ! STUFF FOR SETTING MAGNET USING GLOBAL ARRAY TPSAFIT
  REAL(DP),DIMENSION(:), POINTER :: TPSAFIT
  LOGICAL, POINTER :: SET_TPSAFIT
  ! STUFF FOR PARAMETER DEPENDENCE
  INTEGER NPARA
  INTEGER IAN(NMAX),IBN(NMAX)
  REAL(DP) SAN(NMAX),SBN(NMAX)
  INTEGER IVOLT, IFREQ,IPHAS
  INTEGER IB_SOL
  REAL(DP) SVOLT, SFREQ,SPHAS
  REAL(DP) SB_SOL
  TYPE(POL_BLOCK1) USER1
  TYPE(POL_BLOCK2) USER2
END TYPE POL_BLOCK
```

## L.3 Copying ELEMENT and ELEMENTP: COPY and EQUAL

There are three routines: one to copy ELEMENT into ELEMENTP, ELEMENTP into ELEMENT, and finally ELEMENT into ELEMENT. The syntax is simple. For example if we want to copy EL into ELP we simply write

```
CALL COPY(EL,ELP)
```

or

```
CALL EQUAL(ELP,EL)
```

It should be pointed out that if ELP is a new variable, it must be constructed using some syntax as explained Sect. L.4.

```
ELP=0
CALL COPY(EL,ELP)
```

In PTC a type ELEMENT is created first and then copied into an equivalent ELEMENTP.

## L.4 The Assignment (=)

We will list the various possible assignments and the routines which overload the (=) assignment.

### L.4.1 EL(P)=INTEGER : ZERO\_EL and ZERO\_ELP

ZERO\_EL (ZERO\_ELP) initializes EL (ELP) with the syntax EL=N with  $N \geq 0$ . This calls the constructor for the magnet. The destructor is invoked with "EL=-1".

The syntax EL=N is permitted to create an ELEMENT with EL%MUL=N, i.e., multipoles are assigned to order  $b_N$  and  $a_N$ . The private routine ZERO\_ANBN is called for this purpose.

### L.4.2 EL(P)=STATE : MAGSTATE and MAGPSTATE

In the module S\_TRACKING (file Sm\_TRACKING.f90, see Sect. P) the layout tracking routines must pass the state to the ELEMENT. This is done with the following structure:

```
EL = K          ! PRESENT STATE PASSED to the routine
CALL TRACK(EL,X)
EL = DEFAULT
```

Here we simplified a bit, but the reader can check in Sm\_TRACKING.f90 and he will see indeed this type of structure (see Sect. P.2). The magnet is put in the present state K (whatever that is) and then it reverts to default.

The routine MAGSTATE is quite trivial:

```
SUBROUTINE MAGSTATE(EL,S)
IMPLICIT NONE
TYPE(ELEMENT), INTENT(INOUT)::EL
TYPE(INTERNAL_STATE), INTENT(IN)::S

IF(S%TOTALPATH) THEN
  EL%TOTALPATH=1
ELSE
  EL%TOTALPATH=0
ENDIF

EL%RADIATION=S%RADIATION
EL%TIME=S%TIME
EL%NOCAVITY=S%NOCAVITY
EL%FRINGE=S%FRINGE

END SUBROUTINE MAGSTATE
```

### L.4.3 The Type(WORK): Design Energy

PTC can deal with different design energies in each magnet. This is because the PATCH in the type FIBRE can also patch energy as well as geometry. One can easily retrieve and change this data using a variable of type WORK. It is quite remarkable that if the EXACT\_MODEL option is used in PTC and the total time is used (TOTALPATH state), then a particle can be accelerated with a single reference energy (real way) or with an updated energy. The results then agree perfectly. If fake models are used, then only the updated energy method is reliable. This is why it is constantly used in standard codes. If relative time is used, then one must use the updated energy method and watch out for thick cavities. The situation is subtle. The

important point is that PTC can deal with an exact solution so that fudges can be calibrated and benchmarked.

To perform this kind of work one must be able to read the design energy of a magnet and change it: this is done with a object of type WORK defined in Sect. L.2.

```

TYPE(WORK) ENERGY_DATA
TYPE(FIBRE), POINTER :: P    ! THIS POINTER IS USED TO LOCATE A PARTICULAR FIBRE
.
.
.
NULLIFY(P)
CALL MOVE_TO(PSR,P,2)
WRITE(6,*) "THE NAME IS ",P%MAG%NAME

ENERGY_DATA=P%MAG
POC_OLD=P%MAG%POC
NEW_ENERGY=ENERGY_DATA%MASS + 1.D-3 * ENERGY_DATA%KINETIC
ENERGY_DATA=0
ENERGY_DATA=NEW_ENERGY
P=ENERGY_DATA

```

In the line “ENERGY\_DATA=P%MAG,” P%MAG communicates its design information to ENERGY\_DATA of type WORK (Here “P” is the fibre). Then the kinetic energy is reduced by a factor of 1000 and stored into the real variable NEW\_ENERGY. Then ENERGY\_DATA is zeroed by assigning the integer 0 to it. Finally the assignment ENERGY\_DATA=NEW\_ENERGY fills all the fields of ENERGY\_DATA on the basis of the new total energy. This assignment *always* adds the real value on the right hand side to the total energy and recomputes the WORK variable. If one does not zero the work variable prior to assigning a new energy, then the right hand side is simply added to the old energy. It is possible to use the momentum POC rather than the energy if the global variable FEED\_POC is set to TRUE. It is false by default.

The new reference energy data is passed to the fibre P with P=ENERGY\_DATA. This is defined much later in module S\_FAMILY. This involves several steps. First the new reference energy is passed to the magnet P%MAG and the polymorphic version P%MAGP. This is done in the routines EL\_WORK and ELP\_WORK. In these routines, the AN and BN arrays as well as B\_SOL are rescaled if applicable. In addition, the rescaling routines of the USER1 and USER2 (user-defined types) are called. Of course these routines may be doing nothing. It should be pointed out that in PTC the cavity used the unscaled voltage, thus nothing happens in relation with RF cavities. Now besides the magnet, P%CHART%ENERGY is set to TRUE in the chart of the associated fibre. This means that this magnet has an energy patch.

The syntax ENERGY\_DATA=P is also acceptable for convenience. The information is extracted from P%MAG. In addition, the rescaling of variables triggered by an energy change can be ignored by a unary (+): P=+ENERGY\_DATA. The unary plus sets the variable ENERGY\_DATA%RESCALE to false— something one can do manually as well.

#### L.4.4 The Type(MUL\_BLOCK): Changing the AN and BN

One may want to change the values of AN and BN for all sorts of reasons. The type MUL\_BLOCK facilitates the task. There are several ways to fill in a MUL\_BLOCK. First we can simply initialize a variable of type MUL\_BLOCK using

```

TYPE(MUL_BLOCK) MUL_DATA
.
.
.
MUL_DATA = P%MAG          ! Overloaded with BL_EL or BL_ELP

```

With this syntax, one copies the P%MAG%AN and P%MAG%BN into the corresponding arrays of MUL\_DATA. The maximum size of the arrays is given by NMAX, defaulted<sup>33</sup> to 20 in PTC. For convenience the syntax MUL\_DATA = P is also acceptable. The data is extracted from P%MAG.

<sup>33</sup>PTC has no limit for EL%NMUL, however objects of type MUL\_BLOCK are limited to NMAX. This could be relaxed but not without some small reprogramming.

One can also set up a MUL\_BLOCK of arbitrary size MUL\_BLOCK%NMUL (no bigger than NMAX) with the syntax:

```
MUL_DATA = NMUL ! Overloaded with BL_0
```

Finally, one can set a magnet using a MUL\_BLOCK:

```
P%MAG = MUL_DATA ! Overloaded with EL_BL or ELP_BL
```

This may involve far more than copying. First the value of MUL\_DATA%NMUL may be larger than P%MAG in which case the dynamical arrays in P%MAG must be enlarged. This is done internally using the routine ADD described in Sect. L.6. Finally for magnet types TKTF (KIND7) and TEAPOT (KIND10), the arrays EL%AN and ELP%BN are not used directly and thus a call to special routines are necessary to recompute the matrices for TKTF and the fields for the exact sector bend TEAPOT.

The syntax

```
P = MUL_DATA ! Overloaded with fibre_bl in file Sm_Tracking.f90
```

is also possible; this is equivalent to **P%MAG=MUL\_DATA; P%MAGP=MUL\_DATA;** .

### Unary + on a MUL\_BLOCK

Rather than overwriting the array of a magnet, one can add to the existing multipole components using a unary +. The syntax, applicable to ELEMENT, ELEMENTP, and FIBRE, is just

```
P%MAG = +MUL_DATA ! UNARYP_BL defines the unary +
```

### L.4.5 Example of Non-Trivial Use of Types MUL\_BLOCK and WORK

In the following example, we color coded the various sections.

- In red, we display the reading of original energy data in the variable energy\_data of type WORK.
- In green, we change energy\_data. Now it corresponds to a particle with a kinetic energy a thousand times smaller.
- In orange, we reset the magnet (EL and ELP) and the fibre from # 3 to #8. On the second example, we reset the magnet only.
- In purple, we adjust the definition of relative time if TOTALPATH is false. This is very interesting. Obviously in a real machine TOTALPATH is always true! It shows again that the more one fakes the physics, the harder it gets to do things correctly.
- In turquoise, we set up the energy patches. In the layout tracking routine, an energy patch at the end of element 3 and element 8 will adjust the momenta and the energy variables on the basis of the subsequent element. See Sect. P.2.
- In dark red, we call a layout routine which copies the ELEMENT of each fibre into the equivalent ELEMENTP. In doing so, the entire line is upgraded for regular real number tracking and for polymorph tracking (only in the second example).
- Finally in blue we track the closed orbit in two segments. We stop in the middle of the insane insertion to check that indeed things are pretty wild in there.

Despite the fact that the value of  $X(5)$ ,  $\delta E/p_0c$ , reaches the extraordinary value of 20.58, the code returns correctly to the closed orbit. Even better, a subsequent computation of the one-turn map at position 1, seems unaffected by the insane insertion. Both the usage of SBEND (type TEAPOT) or RBEND (type STREX) return the results of Dragt's paper. Now, as Richard Talman would say, this is *exact*.

**Predictably, non-exact magnets go haywire. Do not try this with EXACT\_MODEL=.FALSE.!**

```
energy_data=p%mag
beta0=energy_data/beta0
p0c_old=energy_data/p0c
```

```

energy_data%energy=0.d0
energy_data=energy_data%mass+1.d-3*energy_data%kinetic
p0c_new=energy_data%p0c

do i=3,8

p=energy_data

  if(.not.default%totalpath )then
    p%PATCH%time=.true.
    p%PATCH%b_t=p%mag%P%ld*(1.d0/beta0-1.d0/p%mag%P%beta0)
  endif

p=>p%next
enddo

x(:)=0.d0
CALL TRACK(PSR,x,1,6,DEFAULT+TIME)
WRITE(6,*) X
CALL TRACK(PSR,x,6,PSR%n+1,DEFAULT+TIME)
WRITE(6,*) X

```

The results are

```

1.465684947001223E-016  1.776356839400250E-015  0.000000000000000E+000
0.000000000000000E+000  20.5837135951167      3.425792982625353E-010
2.482054817439483E-016 -3.976603433443070E-017  0.000000000000000E+000
0.000000000000000E+000  0.000000000000000E+000  3.426300576592212E-009

```

Now just as an example, let us do it the longer and more dangerous way.

```

energy_data=p%mag
beta0=energy_data%beta0
p0c_old=energy_data%p0c

energy_data%energy=0.d0
energy_data=energy_data%mass+1.d-3*energy_data%kinetic
p0c_new=energy_data%p0c
beta0_new=energy_data%beta0

do i=3,8

p%mag%beta0=beta0_new
p%mag%p0c=p0c_new
mul=p%mag
if(p%mag%nmul>0) then
  do k=1,p%mag%nmul
    mul%bn(k)=mul%bn(k)*p0c_old/p0c_new
    mul%an(k)=mul%an(k)*p0c_old/p0c_new
  enddo
p%mag=mul
endif

  if(.not.default%totalpath )then
    p%PATCH%time=.true.
    p%PATCH%b_t=p%mag%P%ld*(1.d0/beta0-1.d0/p%mag%P%beta0)
  endif

p%PATCH%energy=.true.

p=>p%next
enddo

call EL_TO_ELP(psr)

x(:)=0.d0
CALL TRACK(PSR,x,1,6,DEFAULT+TIME)
WRITE(6,*) X
CALL TRACK(PSR,x,6,PSR%n+1,DEFAULT+TIME)
WRITE(6,*) X

```

The above example is more dangerous because we assume explicitly that the only type of magnets are those with  $AN$  and  $BN$  arrays. For example, if a user specified magnet is present, it will be ignored completely. On the other hand the assignment “FIBRE=WORK,” as used in the first example (  $p=energy\_data$  ), will invoke the appropriate scaling rules implemented by the user through  $scale\_user1$  or  $scale\_user2$ . Again, PTC emphasizes the object-oriented nature of that flow and provides methods which hide the inside of an element. However the user can violate the inner guts of a magnet if he so desires.

**It is a good time to remember the importance of the flow once more. The algorithms based on TPSA/Polymorphic/DA maps are all single particle algorithms acting on the flow properly extended. Thus the flow through a magnet must be made into an object; this is why we have the FIBRE. Algorithms cannot be visitor functions on the magnet but on the flow. Analytical calculations (based on maps or Hamiltonians) need the inner details of the magnets and must thus be visitor functions of any private type. The failure to see the importance of the flow leads automatically to a flawed class structure.**

#### L.4.6 Setting the Knobs Using a POL\_BLOCK: Routines BLPOL\_0 and ELP\_POL

This new type is critical for managing knobs easily. In this section we describe its interaction with the ELEMENTP. In Sect. O we will see how it can be used to scan entire layouts and turn into polymorphic knobs a family of magnets as well as individual magnets. Perhaps as we describe here the interaction of a POL\_BLOCK with a single ELEMENTP, the reader will imagine easily the kind of scanning possible on the layout.

Before we start we must remind the reader the properties of polymorphs deeply buried into the FPP package. A real polymorph is defined as (in definition.f90)

```

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

```

Denoting a polymorph as  $P$ , the important points here are the variables  $P\%I$  and  $P\%S$ . If  $P\%KIND = 3$ , i.e., we have a knob, then FPP will always set  $P$  to the following object:

$$\begin{aligned}
 P &= r + s x_i \\
 \text{where } r &= P\%R, \quad s = P\%S, \quad i = P\%I
 \end{aligned}
 \tag{61}$$

Here the variable  $x_i$  is the  $i^{\text{th}}$  variable of the TPSA package being used (Berz’s normally). The variable  $P\%S$  is a compromise designed to be used in generic tracking codes such as PTC. For example if there is a special relation between the various multipole components, it would not be included in PTC per se. By contrast special codes which solve Maxwell’s equations for each magnet may be polymorphic from top to bottom. In such codes the variable  $P\%S$  would be useless and even discouraged. In PTC we tolerate its existence and use it if necessary.

Now, let us see how to set knobs on an example. In this section, we set the knobs of magnets whose position in the layout is known.

```

CALL INIT(DEFAULT+DELTA,2,1,BERZ,ND2,NPARA)
CALL ALLOC(NORMAL); CALL ALLOC(Y);
NULLIFY(P)
CALL MOVE_TO(PSR,P,2)

P%MAGP%VORNAME="JELLO"
P%MAG%VORNAME="JELLO"
POLB=NPARA
POLB%NAME=P%MAGP%NAME
POLB%VORNAME=P%MAG%VORNAME
POLB%IBN(2)=1
PSR=POLB

```

```

X(:)=0.DO
Y=NPARA
Y=X
CALL TRACK(PSR,Y,1,+DEFAULT)
NORMAL=Y
CALL PRINT(NORMAL%DHDJ%V(1),6)

```

In the above code, the TPSA package is initialized to order 2 and with 1 knob. Then the POL\_BLOCK variable is initialized with NPARA. This is important; it will allow PTC to tell FPP the exact location of the knobs. FPP has a knowledge of phase space which, in the case of delta being a parameter (internal state DEFAULT+DELTA), conflicts with PTC. In that case, for PTC the knobs start at 6 while for FPP knobs start at 5. This is why the quantity NPARA is crucial since it is not always equal to ND2.

```

ETALL 1, NO = 2, NV = 6, INA = 272
*****
I COEFFICIENT ORDER EXPONENTS
NO = 2 NV = 6
0 0.2541028124202658 0 0 0 0 0 0
1 -0.9281703999632901 0 0 0 0 1 0
1 0.1507923216684006 0 0 0 0 0 1
-3 0.0000000000000000 0 0 0 0 0 0

```

The normal and skew sextupoles are declared as the first and second knob respectively. Finally the call **PSR=POLB** communicates this information to the layout PSR.

The reader will notice that we set the first name of the second element to JELLO. This is the only QD in the lattice with VORNAME=JELLO. In addition, we set the VORNAME of POL\_BLOCK also to "JELLO." PTC will notice that POLB%VORNAME ≠ ' ' and thus will try a perfect match. Therefore this particular run will produce the dependence of tune as a function of the quadrupole strength of the first QD and not of the full family.

To get the full family, one simply leaves the field POL\_BLOCK%VORNAME blank.

```

P%MAGP%VORNAME="JELLO"
P%MAGP%VORNAME="JELLO"
POLB=NPARA
POLB%NAME=P%MAGP%NAME
POLB%IBN(2)=1
PSR=POLB

```

```

X(:)=0.DO
Y=NPARA
Y=X
CALL TRACK(PSR,Y,1,+DEFAULT)
NORMAL=Y
CALL PRINT(NORMAL%DHDJ%V(1),6)

```

The result is then

```

ETALL 1, NO = 2, NV = 6, INA = 272
*****
I COEFFICIENT ORDER EXPONENTS
NO = 2 NV = 6
0 0.2541028124202658 0 0 0 0 0 0
1 -0.9281703999632901 0 0 0 0 1 0
1 1.507923216684005 0 0 0 0 0 1
-3 0.0000000000000000 0 0 0 0 0 0

```

Predictably, due to the periodicity of the lattice, this first order result is ten times larger. Remember that PTC ignores lower case letters and blanks. But for safety, using capitals with no blanks is certainly recommended.

Let us return to our example, and use the normal sextupole component of this single QD to kill the horizontal chromaticity.

```

NULLIFY(P)
CALL MOVE_TO(PSR,P,2) P%MAGP%VORNAME="JELLO"
P%MAG%VORNAME="JELLO"
POLB=NPARA
POLB%NAME=P%MAGP%NAME
POLB%VORNAME=P%MAG%VORNAME
POLB%IBN(3)=1
PSR=POLB

CALL TRACK(PSR,Y,1,+DEFAULT)
NORMAL=Y CALL PRINT(NORMAL%DHDJ%V(1),6)
TPSAFIT(:)=0.DO
TPSAFIT(1)=- (NORMAL%DHDJ%V(1).SUB.'000010')/(NORMAL%DHDJ%V(1).SUB.'000011')
WRITE(6,*) " TPSAFIT(1) = ", TPSAFIT(1)
SET_TPSAFIT=.TRUE.          ! FOR ASSIGNING TPSAFIT
PSR=POLB                    ! ASSIGNING

CALL KILL(NORMAL); CALL KILL(Y);

```

TPSAFIT is a targeted global array. Each type POL\_BLOCK points to it by default. The user could change this but, in the absence of good reasons, one may as well use TPSAFIT. The above red line is a simple extraction of the BN(3) necessary to kill the chromaticity. This is exact for an ideal machine. Obviously in a non-ideal machine it would be part of a fitting do-loop which would include a call to FIND\_ORBIT. (See Sect. R.2)

Now we follow this code, with the following lines:

```

CALL INIT(DEFAULT+DELTA,2,0,BERZ,ND2,NPARA)
CALL ALLOC(NORMAL); CALL ALLOC(Y);

Y=NPARA
Y=X          ! MAKES Y = CLOSED ORBIT + IDENTITY MAP
CALL TRACK(PSR,Y,1,DEFAULT)
NORMAL=Y
CALL PRINT(NORMAL%DHDJ%V(1),6)

```

First, in dark red, the global targeted logical SET\_TPSAFIT is set to true. This means that now the array TPSAFIT will be used to correct the strength of the magnets on the basis of the information already contained in the variable POLB. Then again POLB is assigned to the magnet QD with VORNAME='JELLO', but this time TPSAFIT is loaded in. The chromaticity, computed once more, is now nearly zero. The result of this code is

```

ETALL  1, NO =  3, NV =  6, INA = 273
*****
  I  COEFFICIENT          ORDER  EXPONENTS
  NO =  3      NV =  6
0  0.2541028124202658      0  0  0  0  0  0
1 -0.9281703999632901      0  0  0  0  1  0
1  0.1255822091885237E-15  0  0  0  0  0  1
2  0.7695358029965638E-01  2  0  0  0  0  0
2  0.8834874115176436E-17  1  1  0  0  0  0
2  0.7695358029965638E-01  0  2  0  0  0  0
2  0.2675634631832561      0  0  2  0  0  0

```



```

2 0.2675634631832561      0 0 0 2 0 0
2 1.045593753641231      0 0 0 0 2 0
2 0.7340570869703674      0 0 0 0 1 1
2 -0.5356864274082178E-31 0 0 0 0 0 2
-11 0.0000000000000000      0 0 0 0 0 0
TPSAFIT(1) = 1.26443898770064
Berz's Package
      NO      ND      ND2      NP      NDPT      NV
      2      2      4      1      0      5
ETALL 1, NO = 2, NV = 5, INA = 272
*****
I COEFFICIENT      ORDER  EXPONENTS
NO = 2      NV = 5
0 0.2541028124202654      0 0 0 0 0
1 0.3887344610677632E-14      0 0 0 0 1
-2 0.0000000000000000      0 0 0 0 0

```

**N.B. You cannot feed different POL\_BLOCKS in succession to the same ELEMENTP. If PTC detects such an attempt, pray that it stops you. It should. You can however wipe out all the knobs from a layout with the KILL\_PARA(LAYOUT) routine and start again with a different set of TPSA knobs.**

#### L.4.7 EL=X(6) : Subroutine MIS\_, MIS\_P, and FIBRE\_MIS for Misalignments

We have already seen this assignment in our discussion of the layout in the PSR example. This routine allocates the translation and rotation arrays of EL and fills them up. It uses the syntax:

```
EL=X
```

which MIS\_ implements as

```

DO I=1,3
  EL%D(I)=X(I)
  EL%R(I)=X(3+I)
ENDDO

```

The reader should notice that the logical variable EL%MIS is not changed by this routine. Thus the misalignment is still inactive. In addition the ELEMENT logical EL%EXACTMIS is also FALSE by default. A similar routine, MIS\_P, handles the polymorphic element ELP with the same syntax: ELP=X.

Of course, this is very dangerous since one could have different misalignments in EL and ELP. For this reason, it is safer to assign the element direction on a fibre. Thus if P is a fibre, the assignment

```
P=X
```

will, through the subroutine FIBRE\_MIS of module S\_FAMILY perform

```

      .
      P%MAG =X
      P%MAGP =X
      P%MAG%MIS = .TRUE.
      P%MAGP%MIS = .TRUE.
      .
      CALL FACTORIZE_ROTATION(S1,S2%CHART%L,S2%CHART%ALPHA,S2%CHART%D_IN, &
        & S2%CHART%ANG_IN,S2%CHART%D_OUT,S2%CHART%ANG_OUT)
      CALL ADJUST_INTERNAL(S2)

```

The call to FACTORIZE\_ROTATION has been explained in Sect. D. ADJUST\_INTERNAL adjusts the MAGNET\_CHART of each element: the magnet is located in absolute space. Finally a user may call the routine MISALIGN\_FIBRE(P,X) rather than the equal sign (=) that it overloads.

## L.5 The SETFAMILY Interface: Pointing from ELEMENT to Magnet Types

This is a routine which is never called by a “standard user.” If a user wishes to add a new type of magnet, then this routine must be extended. We show part of the polymorphic version SETFAMILYP. In particular it is interesting to look at the two thick matrix methods. Here is the part of SETFAMILYP dealing with them:

```

CASE(KIND6)
  IF(EL%P%EXACT.AND.EL%P%B0/=0.DO) THEN
    WRITE(6,*) " EXACT BEND OPTION NOT SUPPORTED FOR KIND ", EL%KIND
    IPAUSE=MYPAUSE(777)
  ENDIF
  IF(.NOT.ASSOCIATED(EL%T6)) THEN
    ALLOCATE(EL%T6)
    EL%T6=0
  ELSE
    EL%T6=-1
    EL%T6=0
  ENDIF
  EL%T6%P=>EL%P
  EL%T6%L=>EL%L
  IF(EL%P%NMUL==0) THEN
    WRITE(6,*) "ERROR ON T6: SLOW THICK "
    IPAUSE=MYPAUSE(0)
  ENDIF
  EL%T6%AN=>EL%AN
  EL%T6%BN=>EL%BN
  EL%T6%FINT=>EL%FINT
  EL%T6%HGAP=>EL%HGAP
  EL%T6%H1=>EL%H1
  EL%T6%H2=>EL%H2
  NULLIFY(EL%T6%MATX);ALLOCATE(EL%T6%MATX(2,3));
  NULLIFY(EL%T6%MATY);ALLOCATE(EL%T6%MATY(2,3));
  NULLIFY(EL%T6%LX);ALLOCATE(EL%T6%LX(6));
  NULLIFY(EL%T6%LY);ALLOCATE(EL%T6%LY(3));
CASE(KIND7)
  IF(EL%P%EXACT.AND.EL%P%B0/=0.DO) THEN
    WRITE(6,*) " EXACT BEND OPTION NOT SUPPORTED FOR KIND ", EL%KIND
    IPAUSE=MYPAUSE(777)
  ENDIF
  ! IF(.NOT.ASSOCIATED(EL%T7))ALLOCATE(EL%T7)
  IF(.NOT.ASSOCIATED(EL%T7)) THEN
    ALLOCATE(EL%T7)
    EL%T7=0
  ELSE
    EL%T7=-1
    EL%T7=0
  ENDIF
  EL%T7%P=>EL%P
  EL%T7%L=>EL%L
  IF(EL%P%NMUL==0) THEN
    WRITE(6,*) "ERROR ON T7: FAST THICK "
    IPAUSE=MYPAUSE(0)
  ENDIF
  EL%T7%AN=>EL%AN
  EL%T7%BN=>EL%BN
  EL%T7%FINT=>EL%FINT
  EL%T7%HGAP=>EL%HGAP
  EL%T7%H1=>EL%H1
  EL%T7%H2=>EL%H2
  NULLIFY(EL%T7%MATX);ALLOCATE(EL%T7%MATX(2,3));
  NULLIFY(EL%T7%MATY);ALLOCATE(EL%T7%MATY(2,3));
  NULLIFY(EL%T7%LX);ALLOCATE(EL%T7%LX(3));
  NULLIFY(EL%T7%RMATX);ALLOCATE(EL%T7%RMATX(2,3));
  NULLIFY(EL%T7%RMATY);ALLOCATE(EL%T7%RMATY(2,3));
  NULLIFY(EL%T7%RLX);ALLOCATE(EL%T7%RLX(3));
  IF(GEN) CALL GETMAT7(EL%T7)

```

The reader will remember (see Sect. K.4.6) that for the slow element KTK the code must recompute the matrix (EL%T6%MATX and EL%T6%MATY) and the quadratic polynomial (EL%T6%LX and EL%T6%LY) for the path length all the time! This means that the polymorphic variables associated to these pointers should be constructed (ALLOC of FPP) and destroyed (KILL of FPP) inside the tracking routines for the magnet EL%T6. The reader can check that this is the case by looking at INTKTKD and INTKTKS of Sh\_DEF\_KIND.f90.

This situation is to be contrasted with type TKTF and type TEAPOT (see Sects. K.4.7 and Sect. K.4.9). For TKTF the matrix of an element is computed once and for all in SETFAMILY by the call GETMAT7(EL%T7). Therefore the polymorphic variables are constructed using ALLOC(EL%T7). In the tracking routines INTTKT7D and INTTKT7S one can check that calls to ALLOC or KILL are performed only

if the length `EL%T7%L` or the quadrupole strength `EL%T7%BN(2)` are knobs. In that case the matrix is computed with parametric dependence. Then, upon exiting, it is restored to its original state. In the case of TEAPOT, the exact sector bend, the  $a_n$  and  $b_n$  are not used directly in the tracking for speed reasons. Maxwell's equations are solved and the field is computed once and for all in terms of  $a_n$  and  $b_n$  defined by Equation (56). Thus PTC recomputes the field only if polymorphic knobs are needed as in the case of TKTF.

Finally, it is interesting to look at `KINDUSER1`. Here `SETFAMILY` does the minimal generic amount of work:

```

CASE(KINDUSER1)
  if(.not.ASSOCIATED(EL%U1)) THEN
    ALLOCATE(EL%U1)
    EL%U1=0
  ELSE
    EL%U1=-1
    EL%U1=0
  ENDIF
  EL%U1%P=>EL%P
  EL%U1%L=>EL%L
  IF(EL%P%NMUL==0) CALL ZERO_ANBN(EL,1)
  EL%U1%AN=>EL%AN
  EL%U1%BN=>EL%BN
  CALL POINTERS_USER1(EL%U1)
  CALL ALLOC(EL%U1)      ! In Polymorphic version only

```

The arrays `AN` and `BN` should be present even if not used at all. User defined fields are allocated in `POINTERS_USER1` while the polymorphs are constructed by `CALL ALLOC(EL%U1)`.

## L.6 Adding Multipole Components: ADD

When fitting or simulating errors it is common to extend the multipole content of a magnet. This is a little tricky in PTC because the arrays are dynamically allocated to order `EL%NMUL`. For example, if we want to put an octupole inside QD of the PSR ring, the following logic would provoke a system exception/crash or an aberrant behavior:

```

QD%MAG%NMUL=4
QD%MAG%BN(4)=1.d0

```

The array `QD%MAG%BN(4)` was only defined as `QD%MAG%BN(1:2)`. Therefore, PTC provides the routine interface `ADD` for `ELEMENT` and `ELEMENTP`. The correct syntax is thus

```
CALL ADD(QD%MAG,4,1,1.d0)
```

Generally the input is

$$\text{ADD}(\text{EL}, \pm N, i, \Delta)$$

$$\left. \begin{array}{l} \text{EL}\% \text{BN}(N) \\ \text{EL}\% \text{AN}(N) \end{array} \right\} = \begin{cases} i * \text{EL}\% \text{BN}(N) + \Delta & \text{if } N > 0 \\ i * \text{EL}\% \text{AN}(-N) + \Delta & \text{if } N < 0 \end{cases}$$

The routine `ADD` takes care of the allocation and deallocation details for the user including resetting pointers. This includes automatically the user defined elements `KINDUSER1` and `KINDUSER2`. The call can also be done on the fibre itself:

```
CALL ADD(QD,4,1,1.d0)
```

in which case `MAG` and `MAGP` are changed.

## M S<sub>j</sub>\_ELEMENTS.f90

This module is very simple after all the hard work put into the basic modules. It creates the interface for the magnet calls TRACK(EL,X), TRACK(ELP,Y), and TRACK(ELP,YS). The reader should look at the module: it contains three routines: TRACKR, TRACKP, and TRACKS. For convenience, TRACKR is presented here:

```
SUBROUTINE TRACKR(EL,X)
  IMPLICIT NONE
  INTEGER IPAUSE, MYPAUSE
  REAL(DP),INTENT(INOUT):: X(6)
  TYPE(ELEMENT),INTENT(INOUT):: EL

  SELECT CASE(EL%KIND)
  CASE(KIND0)
  CASE(KIND1)
    CALL TRACK(EL%D0,X)
  CASE(KIND2)
    CALL TRACK(EL%K2,X)
  CASE(KIND3)
    CALL TRACK(EL%K3,X)
  CASE(KIND4)
    CALL TRACK(EL%C4,X)
  CASE(KIND5)
    CALL TRACK(EL%S5,X)
  CASE(KIND6)
    CALL TRACK(EL%T6,X)
  CASE(KIND7)
    CALL TRACK(EL%T7,X)
  CASE(KIND8)
    CALL TRACK(EL%S8,X)
  CASE(KIND9)
    CALL TRACK(EL%S9,X)
  CASE(KIND10)
    CALL TRACK(EL%TP10,X)
  CASE(KIND11:KIND14)
    CALL TRACK(EL%MON14,X)
  CASE(KIND15)
    CALL TRACK(EL%SEP15,X)
  CASE(KIND16)
    CALL TRACK(EL%K16,X)
  CASE(KIND17)
    CALL TRACK(EL%S17,X)
  CASE(KINDFITTED)
    CALL TRACK(EL%BEND,X)
  CASE(KINDUSER1)
    CALL TRACK(EL%U1,X)
  CASE(KINDUSER2)
    CALL TRACK(EL%U2,X)
  CASE DEFAULT
    WRITE(6,*) EL%KIND," NOT SUPPORTED "
    IPAUSE=MYPAUSE(0)
  END SELECT
END SUBROUTINE TRACKR
```

This routine is just a “multiple goto” to the appropriate magnet routine in module S\_DEF\_KIND. It is quite trivial to see how one would add a new magnet.

## N Sk\_LINK\_LIST.f90

In this section we take a look at the routines manipulating the fundamental type FIBRE and the linked list LAYOUT built upon the fibre. We also list the routines of the module which include some FLAT file capability. We explain why printing the flat file of a fibre bundle is not as easy as a that of a standard lattice.

### N.1 The fundamental types FIBRE and LAYOUT

We have already outlined the major aspects of the linked list used in PTC; these objects are defined in the fibre\_bundle module in the file Sk\_LINK\_LIST.f90. For the record, here are again the actual types. First we start with type FIBRE:

```
TYPE FIBRE
  ! BELOW ARE THE DATA CARRIED BY THE NODE
  INTEGER, POINTER :: DIR
  REAL(DP), POINTER :: POC, BETA0
  TYPE(PATCH), POINTER :: PATCH
  TYPE(CHART), POINTER :: CHART
  TYPE (ELEMENT), POINTER :: MAG
  TYPE (ELEMENTP), POINTER :: MAGP
  ! END OF DATA
  ! POINTER TO THE MAGNETS ON EACH SIDE OF THIS NODE
  TYPE (FIBRE), POINTER :: PREVIOUS
  TYPE (FIBRE), POINTER :: NEXT
  ! POINTING TO PARENT LAYOUT AND PARENT FIBRE DATA
  TYPE (LAYOUT), POINTER :: PARENT_LAYOUT
  TYPE (FIBRE), POINTER :: PARENT_PATCH
  TYPE (FIBRE), POINTER :: PARENT_CHART
  TYPE (FIBRE), POINTER :: PARENT_MAG
END TYPE FIBRE
```

A fibre is the actual node of the list. It is recursively defined with pointers to the adjacent fibres. Mathematically the most important variable of the fibre is the chart. This is a complete reversal from standard accelerator physics. In standard accelerator physics, the connection between the coordinates on the magnet is deduced **solely** from the so-called “geometry of the magnet.” Thus there is no need for a chart. In addition, standard accelerator physics, suffering from acute Courant-Snyderitis, assumes a smooth connection between the outside space and the internal coordinates of the magnets. It does not do this on physical grounds, but because it tries to impose a smooth Hamiltonian structure on the “s” dependent flow. This violates completely the object-orientedness of the magnet-flow when it is realizable.

As we said before, this is reflected in the CLASSIC classes. There is no fibre and when a discontinuous patch is needed, it is introduced as an *ad hoc* element in the beam line.

Here, in PTC, as well as in the original C++ classes that Bengtsson dreamt up in collaboration with Forest, the geometrical nature of the fibre reigns supreme. The next step is to introduce a magnet, namely EL and/or ELP. The propagator of the full fibre, if well-defined, inherits properties from the chart itself. In other words a magnet exists first as a piece of material junk. It can be rotated, translated and drawn. The chart provides the connection between this magnet/junk and the external three dimensional space. Obviously this exists independently of the existence of single particle propagators associated to EL and/or ELP. It is a remarkable mathematical feature that these propagators, under certain conditions, inherit the transformational properties of the chart. Of course PTC is set up to take advantage of this.

In type FIBRE, there are a few other quantities of interest. In addition to CHART, there is of course PATCH. Patch contains the various patches necessary, if the previous fibre was not smoothly joined to the present one. In recirculators, the distinction between a fibre and a magnet is glaring. Indeed the magnet may be traversed many times, but the patches needed to enter the following fibre may differ turn after turn. In Figure 20, the particle passes several times through the “common bend.” Each time the magnet must be the same obviously. However the patches to the following fibre must differ. If one thinks of the fibre as the “s” variables and the patches as s-dependent transformations bringing us to the correct local coordinates, it should be clear how the FIBRE of PTC completely solves recirculation, dog-bones, common rings and other oddities that gave CLASSIC and MAD9 stomach burns.

The next type is the layout itself which is a linked list based on type FIBRE.

TYPE LAYOUT

```

CHARACTER(120), POINTER :: NAME ! IDENTIFICATION
INTEGER, POINTER :: INDEX, CHARGE ! IDENTIFICATION, CHARGE SIGN
LOGICAL, POINTER :: CLOSED
INTEGER, POINTER :: N ! TOTAL ELEMENT IN THE CHAIN
INTEGER, POINTER :: NTHIN ! NUMBER IF THIN LENSES IN COLLECTION (FOR SPEED ESTIMATES)
REAL(DP), POINTER :: THIN ! PARAMETER USED FOR AUTOMATIC CUTTING INTO THIN LENS
!POINTERS OF LINK LAYOUT
INTEGER, POINTER :: LASTPOS ! POSITION OF LAST VISITED
TYPE (FIBRE), POINTER :: LAST ! LAST VISITED
!
TYPE (FIBRE), POINTER :: END
TYPE (FIBRE), POINTER :: START
TYPE (FIBRE), POINTER :: START_GROUND ! STORE THE GROUNDED VALUE OF START DURING CIRCULAR SCANNING
TYPE (FIBRE), POINTER :: END_GROUND ! STORE THE GROUNDED VALUE OF END DURING CIRCULAR SCANNING
END TYPE LAYOUT

```

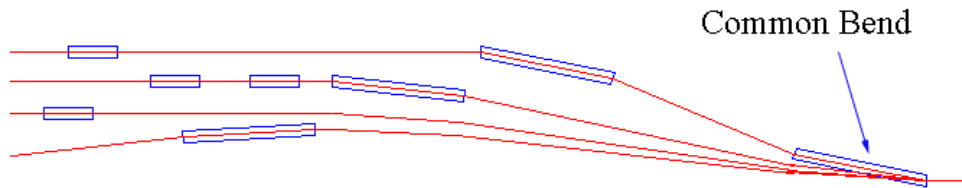


Figure 20: Line Switch at a Common Bend

The type layout contains variables of greater and lesser importance. As explained before, the variable CLOSED controls the nature of the linked list: it is either terminated at both ends for a single pass system or, for a ring, it is circular. In the file Sk\_LINK\_LIST.f90 the maintenance routines, which control the behavior of the list, assume a terminated link list. Therefore PTC provides two routines to toggle between a terminated and circular state. These are LINE\_L and RING\_L:

```

SUBROUTINE LINE_L(L,DONEIT)
IMPLICIT NONE
TYPE (LAYOUT) L
LOGICAL DONEIT
DONEIT=.FALSE.
IF(L%CLOSED) THEN
  IF(ASSOCIATED(L%END%NEXT)) THEN
    L%END%NEXT=>L%START_GROUND
    DONEIT=.TRUE.
  ENDIF
  IF(ASSOCIATED(L%START%PREVIOUS)) THEN
    L%START%PREVIOUS=>L%END_GROUND
  ENDIF
ENDIF
END SUBROUTINE LINE_L

SUBROUTINE RING_L(L,DOIT)
IMPLICIT NONE TYPE (LAYOUT) L
LOGICAL DOIT
IF(L%CLOSED.AND.DOIT) THEN
  IF(.NOT.(ASSOCIATED(L%END%NEXT))) THEN
    L%START_GROUND=>L%END%NEXT ! SAVING GROUNDED POINTER
    L%END%NEXT=>L%START
  ENDIF
ENDIF

```

```

        IF(.NOT.(ASSOCIATED(L%START%PREVIOUS))) THEN
            L%END_GROUND=>L%START%PREVIOUS ! SAVING GROUNDED POINTER
            L%START%PREVIOUS=>L%END
        ENDIF
    ENDIF
END SUBROUTINE RING_L

```

These two routines implement the “cutting and re-establishment” of the violet link in the figure of Sect. A.2.1. They are called with a logical which is local to the calling routine. There are a lot of maintenance routines needed, and of course the number of such routines will increase. But, just for pedagogical purposes, here is a routine which appends the copy of an existing FIBRE to a LAYOUT.

```

SUBROUTINE APPEND_FIBRE( L, EL )
IMPLICIT NONE
    TYPE (FIBRE), INTENT(IN) :: EL
    TYPE (FIBRE), POINTER :: CURRENT
    TYPE (LAYOUT), TARGET, INTENT(INOUT):: L
    LOGICAL DONEIT
    CALL LINE_L(L,DONEIT)
    L%N=L%N+1
    CALL ALLOC_FIBRE(CURRENT)
    CALL COPY(EL%MAGP,CURRENT%MAG)
    CALL COPY(CURRENT%MAG,CURRENT%MAGP)
    CALL COPY(EL%MAG,CURRENT%MAG)
    CALL COPY(EL%CHART,CURRENT%CHART)
    CALL COPY(EL%PATCH,CURRENT%PATCH)
    CURRENT%DIR=EL%DIR
    CURRENT%POC=EL%POC
    CURRENT%BETAO=EL%BETAO

    CURRENT%PARENT_LAYOUT=>L
    IF(L%N==1) CURRENT%NEXT=> L%START
    CURRENT % PREVIOUS => L % END ! POINT IT TO NEXT FIBRE
    IF(L%N>1) THEN
        L % END % NEXT => CURRENT !
    ENDIF

    L % END => CURRENT
    IF(L%N==1) L%START=> CURRENT

    L%LASTPOS=L%N ; L%LAST=>CURRENT;
    CALL RING_L(L,DONEIT)
END SUBROUTINE APPEND_FIBRE

```

The routine APPEND\_FIBRE is typically used in creating a beam line in a standard lattice. An element (a fibre in PTC) is taken from the list of ideal elements and cloned: this is done with the procedure interface “COPY” in PTC. The ELEMENT EL%MAG, the ELEMENTP EL%MAGP, the chart EL%CHART and the patch EL%PATCH are all cloned in variable CURRENT and appended at the end of the layout.

Of course PTC can do much more. In particular in recirculators, where certain beam lines are re-used more than once, fibres must not be cloned but pointed at. The following subroutine APPEND\_POINT is very useful in recirculators. In fact we used it in the examples Sects. B.1 and B.2.

```

SUBROUTINE APPEND_POINT( L, EL )
IMPLICIT NONE
    TYPE (FIBRE), POINTER :: EL
    TYPE (FIBRE), POINTER :: CURRENT
    TYPE (LAYOUT), TARGET:: L
    TYPE(FIBRE), POINTER :: P
    LOGICAL DONEIT
    NULLIFY(P);
    CALL LINE_L(L,DONEIT)
    L%N=L%N+1

```

```

CALL ALLOCATE_FIBRE(CURRENT);

! FINDING THE VERY ORIGINAL FIBRE RECURSIVELY
P=>EL;DO WHILE(ASSOCIATED(P)); CURRENT%PARENT_MAG=>P ;P=>P%PARENT_MAG; ENDDO;
P=>EL;DO WHILE(ASSOCIATED(P)); CURRENT%PARENT_PATCH=>P ;P=>P%PARENT_PATCH; ENDDO;
P=>EL;DO WHILE(ASSOCIATED(P)); CURRENT%PARENT_CHART=>P ;P=>P%PARENT_CHART; ENDDO;
! END OF FINDING THE VERY ORIGINAL FIBRE
CURRENT%PARENT_LAYOUT=>EL%PARENT_LAYOUT
CURRENT%MAG=>EL%MAG
CURRENT%MAGP=>EL%MAGP
CURRENT%CHART=>EL%CHART
CURRENT%PATCH=>EL%PATCH
ALLOCATE(CURRENT%DIR);ALLOCATE(CURRENT%POC);ALLOCATE(CURRENT%BETAO);
CURRENT%DIR=EL%DIR
CURRENT%POC=EL%POC
CURRENT%BETAO=EL%BETAO
IF(L%N==1) CURRENT%NEXT=> L%START
CURRENT % PREVIOUS => L % END ! POINT IT TO NEXT FIBRE
IF(L%N>1) THEN
    L % END % NEXT => CURRENT !
ENDIF

L % END => CURRENT
IF(L%N==1) L%START=> CURRENT

L%LASTPOS=L%N ;
L%LAST=>CURRENT;
CALL RING_L(L,DONEIT)

END SUBROUTINE APPEND_POINT

```

The reader will notice the crucial differences. A fibre EL is passed to the procedure, presumably from one of the beam lines of the recirculator defined in a traditional way. The recirculator itself is denoted by L. Rather than cloning the fibre EL, the content of the new fibre points to the content of the existing fibre EL. For example, if a particle revisits the same linac 4 times, then each time it enters its first quadrupole, it actually enters the very same element of PTC, just as it does in the real machine. Of course since PTC can handle energies and deltas  $X(5)$  without fudges, this element perform its role without problems even if the particle has gained a tremendous amount of energy.

The reader should try to see why the syntax `CURRENT=>EL` is disastrous and fails to achieve the desired result. (Remember that EL is presumably from an existing beam line).

The `PARENT_MAG`, `PARENT_CHART` and `PARENT_PATCH` pointers are there to indicate that the FIBRE contains data which originated somewhere else. It is not essential for tracking a strange beam line; however without them, it would not be possible to copy an oddity such a recirculator. It would not be possible for example to produce a flat file. How can that be? One can imagine a sequence of persons one visits at certain street addresses to hand in a pamphlet. The street address, the “s” variable, is the fibre. The person at a given address is the data inside the fibre, for example the magnet. Handing in the pamphlet represents the tracking operation. So, if one sees that the person at address # 5 looks identical to that of address # 19, one can certainly hand in the pamphlet. Now suppose you instruct the person at number # 5 to slap the face of any peddler (this represents a mispowering!), then do you have to worry about # 19? The answer is obviously no. If the person at # 19 is the same person, then he will simply follow the instruction you gave him when you “mispowered” him when talking to him at address # 5. If on the other hand # 19 is a twin of his, then he will stay put unless specifically instructed to do the same. This is how PTC handles any complex oddities during tracking. This is why the FIBRE structure is so powerful. The real electron does not need to know whether or not # 5 and # 19 are the same individual and neither does PTC in simulation mode. But of course PTC, especially if linked to MAD-X, becomes like the trinity of Brahma, Vishnu and Shiva: it creates, duplicates and annihilates lattices. So it must therefore know whether or not # 5 and # 19 are the same object or simply twins. The PARENT pointers are a minor first step in this direction and they were tested in FLAT file routines found in the module `S_FIBRE_BUNDLE`.



## N.2 The various routines of S\_FIBRE\_BUNDLE

At present there are four basic types of routines in this module. First we see routines that manipulate fibres and layouts in a rather fundamental way. Whenever a link list is created, it must be accompanied by a manipulation routine. Without these PTC would simply not function. This is the big difference between a linked list and a simple allocatable array. The routines of the second type concern the printing of a flat file. These routines, for the moment, work for a recirculator; they are not intended for a double Siamese ring configuration. Nothing is fundamentally impossible here; this is not the primary function of PTC at this time. Clearly if and when MAD-X supports fully Siamese, recirculators and other monstrosities, then MAD-X will need C-routines of the type described here of far greater power than the meagre collection in PTC. The third type, with a single representative, is a switching routine. This routine allows a magnet to change kind. Again one can hardly imagine MAD-X without an arsenal of these routines. Here in PTC it is almost useless since users could generate these things in FORTRAN90 as they need them.

The final two routines are patching routines interfacing the routine FIND\_PATCH of the module S\_FRAME. These are extremely important routines. Beyond MAD-X, these routines would be central to a version of PTC (or MAD-X) tied to a CAD program. This is the logical way to design follow-the-terrain lattices where the optics is sometimes simpler than the civil engineering work of guiding the beam line through a tight physical space.

```
INTERFACE KILL
  MODULE PROCEDURE KILL_LAYOUT
  MODULE PROCEDURE DEALLOC_FIBRE
INTERFACE ALLOC
  MODULE PROCEDURE SET_UP
INTERFACE APPEND
  MODULE PROCEDURE APPEND_FIBRE
INTERFACE MOVE_TO
  MODULE PROCEDURE MOVE_TO_P
  MODULE PROCEDURE MOVE_TO_NAME
  MODULE PROCEDURE MOVE_TO_NAME2
  MODULE PROCEDURE MOVE_FROM_TO_NAME
INTERFACE FIND_PATCH
  MODULE PROCEDURE FIND_PATCH_P
  MODULE PROCEDURE FIND_PATCH_O
INTERFACE ASSIGNMENT (=)
MODULE PROCEDURE NULL_ITO
! CREATION AND DESTRUCTION ROUTINES
SUBROUTINE KILL_LAYOUT( L ) ! DESTROYS A LAYOUT
SUBROUTINE APPEND_FIBRE( L, EL ) ! STANDARD APPEND THAT CLONES EVERYTHING
SUBROUTINE FIND_POS( L, C, I ) ! FINDS THE LOCATION "I" OF THE FIBRE C IN LAYOUT L
SUBROUTINE MOVE_TO_P( L, CURRENT, I ) ! MOVES CURRENT TO THE K^TH POSITION
SUBROUTINE MOVE_FROM_TO_NAME( L, C1, POSC1, CURRENT, NAME, POS ) ! MOVES FROM (C1, POSC1) TO CURRENT CALLED "NAME" (POSC1<=0 THEN FINDS POSC1)
SUBROUTINE MOVE_TO_NAME( L, CURRENT, NAME, POS ) ! MOVES TO NEXT ONE IN LIST CALLED NAME
SUBROUTINE MOVE_TO_FLAT( L, CURRENT, NAME, POS ) ! FIND IN A SIMPLE FLAT FILE (LIKE MOVE_TO_NAME BUT STARTS WITH ONESELF AND SCAN COMPLETELY)
SUBROUTINE MOVE_TO_NAME2( L, CURRENT, NAME, VORNAME, POS ) ! SAME AS MOVE_TO_NAME BUT MATCHES NAME AND VORNAME
SUBROUTINE SET_UP( L ) ! SETS UP A LAYOUT: GIVES A UNIQUE NEGATIVE INDEX
SUBROUTINE DE_SET_UP( L ) ! DEALLOCATES LAYOUT CONTENT
SUBROUTINE NULL_ITO( L, I ) ! NULLIFIES LAYOUT CONTENT
SUBROUTINE LINE_L( L, DONEIT ) ! MAKES INTO LINE TEMPORARILY
SUBROUTINE RING_L( L, DOIT ) ! BRINGS BACK TO RING IF NEEDED
SUBROUTINE APPEND_MAD_LIKE( L, EL ) ! USED IN MAD-LIKE INPUT
SUBROUTINE APPEND_POINT( L, EL ) ! APPOINTS WITHOUT CLONING
SUBROUTINE APPEND_EMPTY( L ) ! CREATES AN EMPTY FIBRE TO BE FILLED LATER
SUBROUTINE NULL_FIBRE( CURRENT ) ! NULLIFIES FIBRE CONTENT
SUBROUTINE ALLOCATE_FIBRE( CURRENT ) ! ALLOCATES AND NULLIFIES CURRENT'S CONTENT
SUBROUTINE ALLOCATE_DATA_FIBRE( CURRENT ) ! ALLOCATES POINTERS IN FIBRE
SUBROUTINE ALLOC_FIBRE( C ) ! DOES THE FULL ALLOCATION OF FIBRE AND INITIALIZATION OF INTERNAL VARIABLES
SUBROUTINE DEALLOC_FIBRE( C ) ! DESTROYS INTERNAL DATA IF IT IS NOT POINTING (I.E. NOT A PARENT)
SUBROUTINE APPEND_FLAT( L, EL, NAME ) ! POINTS UNLESS CALLED "NAME" IN WHICH CASE IT CLONES
! SWITCHING ROUTINES
SUBROUTINE SWITCH_TO_KIND7( EL ) ! SWITCH TO KIND7
! PATCHING FIBRES
SUBROUTINE FIND_PATCH_P( EL1, EL2, D, ANG, DIR, ENERGY_PATCH ) ! COMPUTES PATCHES
SUBROUTINE FIND_PATCH_O( EL1, EL2, NEXT, ENERGY_PATCH ) ! COMPUTES PATCHES
! FLAT FILES
SUBROUTINE PRINT_FLAT( L, FILEN )
SUBROUTINE READ_FLAT( L, FILEN )
SUBROUTINE READ_FULL_FIBRE( L, A, MF )
FUNCTION ANALYSE_FIBRE( EL ) SUBROUTINE PRINT_P_FIBRE( L, EL, A, MF )
SUBROUTINE PRINT_BASIC( L, MF )
SUBROUTINE READ_BASIC( L, MF )
SUBROUTINE PRINT_MAG( EL, MF )
SUBROUTINE READ_MAG( EL, MF )
SUBROUTINE PRINT_MAG_CHART( EL, MF )
SUBROUTINE READ_MAG_CHART( EL, MF )
SUBROUTINE PRINT_PATCH( EL, MF )
SUBROUTINE READ_PATCH( EL, MF )
```

```
SUBROUTINE PRINT_CHART( EL,MIS, MF )  
SUBROUTINE READ_CHART( EL, MIS,MF )
```

## O S\_FAMILY.f90

The prototype versions of PTC as well as its pre-prototype Small\_Code used allocatable arrays instead of linked lists. These types were easier to manage but less flexible. The layout was then defined in the module S\_FAMILY within the file S\_FAMILY.f90. These codes were not capable of handling fibre bundles and thus were only experimental prototypes on which the magnet physics and the Taylor polymorphism could be tested.

With the transformation of PTC into a fully fledged fibre bundle structure and the resulting complexity of managing our linked list, the FIBRE and the LAYOUT were put into the module S\_FIBRE\_BUNDLE of Sk\_LINK\_LIST.f90. Thus the module S\_FAMILY now contains a few layout operations ( standard surveys, translation and rotations of entire Layouts) as well as a few fibre operations (misalignments, scans for polymorphic knobs, etc.).

### O.1 More on POL\_BLOCK

Here we discuss the interaction of a POL\_BLOCK with the full layout.

#### O.1.1 Assigning Polymorphs to a Layout with SETPOL\_L: LAYOUT=POL\_BLOCK

We revisit here the topic of Sect. L.4.6. The routine responsible for the assignment definition LAYOUT=POL\_BLOCK is SETPOL.

The following piece of code, based on our PSR lattice, does the same work in two different ways. In the **red lines**, we simply scan for all the magnets with the same name as magnet #3, which happens to be QD. Now in the section in the **blue coding**, we assign the name of magnet #3 to POL, we then equate POL to PSR. PTC will scan for the members of the same family (same EL%NAME) and will power the polymorphic knobs accordingly.

```
CALL INIT(DEFAULT,5,2,BERZ,ND2,NPARA)
CALL ALLOC(NORMAL)
CALL ALLOC(Y)
POLB=NPARA ;POLB%IBN(3)=1 ;POLB%IAN(3)=2 ;
CALL MOVE_TO(PSR,P,2)

C=>PSR%START
DO I=1,PSR%N
  IF(C%MAGP%NAME==P%MAGP%NAME.AND.(.NOT.ASSOCIATED(C%PARENT_MAG))) C%MAGP=POLB
C=>C%NEXT
ENDDO

X(:)=0.DO ;Y=NPARA; Y=X ;CALL TRACK(PSR,Y,1,-DEFAULT) ;NORMAL=Y;
CALL PRINT(NORMAL%DHDJ%V(1),6)
```

The red coding can be replaced by the following blue coding:

```
POLB%NAME=P%MAGP%NAME
PSR=POLB
```

When a POL\_BLOCK is assigned to a fibre (C=POLB) or equivalently to an ELEMENTP (C%MAGP=POLB), PTC checks the NAME and the VORNAME of the POL\_BLOCK. If the fields are blank, then the assignment proceeds. If there is a NAME but no VORNAME, then it checks only the NAME. This is used when one wants the dependence on a family. Finally if both the NAME and the VORNAME are present, then only a perfect match results in a polymorphic assignment.

Of course the assignment PSR=POLB performs a complete scan of the lattice. The routine for the scan checks the status of the PARENT\_MAG pointer and thus assigns knobs only to the primitive fibre which contains the original data.

Finally, it is possible to use POLB to perform the reverse operation. Rather than investigate the polymorphic dependence, one can power the magnets as a function of the polymorphic description in POLB. Again consider a variant of the example we just displayed:

```

CALL INIT(DEFAULT,3,2,BERZ,ND2,NPARA)
CALL ALLOC(NORMAL);CALL ALLOC(Y);

POLB=NPARA ;POLB%IBN(3)=1 ;POLB%IAN(3)=2 ;
CALL MOVE_TO(PSR,P,2)
POLB%NAME=P%MAGP%NAME
PSR=POLB
X(:)=0.DO ;Y=NPARA; Y=X ;CALL TRACK(PSR,Y,1,+DEFAULT) ;NORMAL=Y
CALL PRINT(NORMAL%DHDJ%V(1),6)
TPSAFIT=0.DO
TPSAFIT(1)=- (NORMAL%DHDJ%V(1).SUB.'00001000')/(NORMAL%DHDJ%V(1).SUB.'00001010')
WRITE(6,*) TPSAFIT(1)

```

```

SET_TPSAFIT=.TRUE.
PSR=POLB ; CALL KILL_PARA(PSR);

```

```

CALL INIT(DEFAULT,2,0,BERZ,ND2,NPARA)
CALL ALLOC(NORMAL);CALL ALLOC(Y);
X(:)=0.DO ;Y=NPARA; Y=X ;CALL TRACK(PSR,Y,1,DEFAULT) ;NORMAL=Y
CALL PRINT(NORMAL%DHDJ%V(1),6)

```

Here the interesting lines are in red. The variable SET\_TPSAFIT, which is a global targeted logical, is set to true. The constructor command POLB=NPARA automatically forces the pointer POLB%SET\_TPSAFIT to point to this global logical. It also forces the pointer POLB%TPSAFIT to point to the global arrays TPSAFIT(LNV). Thus, the arrays TPSAFIT can be fed to the magnet on the basis of the polymorphic information of POLB. In the above example, the horizontal chromaticity is computed and then corrected. The result of the code is:

```

ETALL 1, NO = 3, NV = 8, INA = 299
*****
I COEFFICIENT ORDER EXPONENTS
NO = 3 NV = 8
0 0.2541028124202658 0 0 0 0 0 0 0 0
1 -0.9281703999632887 0 0 0 0 1 0 0 0
1 0.5670943332284326E-15 0 0 0 0 0 0 1 0
2 0.7695358029965586E-01 2 0 0 0 0 0 0 0
2 0.7695358029965586E-01 0 2 0 0 0 0 0 0
2 0.2675634631832551 0 0 2 0 0 0 0 0
2 0.2675634631832551 0 0 0 2 0 0 0 0
2 1.045593753641225 0 0 0 0 2 0 0 0
2 7.340570869703631 0 0 0 0 1 0 1 0
2 -0.1525438237398347E-29 0 0 0 0 0 0 2 0
2 -0.7712950084242795E-27 0 0 0 0 0 0 0 2
-11 0.0000000000000000 0 0 0 0 0 0 0 0
TPSAFIT(1) = 0.126443898770064
Berz's Package
NO ND ND2 NP NDPT NV
2 3 6 0 5 6
ETALL 1, NO = 2, NV = 6, INA = 298
*****
I COEFFICIENT ORDER EXPONENTS
NO = 2 NV = 6
0 0.2541028124202655 0 0 0 0 0 0
1 -0.3975693351829396E-15 0 0 0 0 1 0
-2 0.0000000000000000 0 0 0 0 0 0

```

**N.B. As said in Sect. L.4.6, you cannot feed different POL\_BLOCKS in succession to the same ELEMENTP. If PTC detects such an attempt, pray that it stops you. It should. For example, if you rerun the above example with SET\_TPSAFIT=.TRUE. commented out, you should get the message**

YOU CANNOT USE A POL\_BLOCK AGAIN ON SAME ELEMENTP QD  
PAUSE: 144

as the second PSR=POLB is illegal. Of course it is possible, using the command KILL\_PARA(PSR), to reset all the knobs and start again.

### O.1.2 Why is the Polymorph ELP%L not in POL\_BLOCK?

The careful reader may have noticed that the variable L of ELEMENTP is a polymorph. But, upon a careful examination of type POL\_BLOCK, it seems that we are making the user's life difficult by ignoring it totally.

Welcome to PTC's worst variable. The variable L is truly a pointer to something related to the inner private parts of the various elements. Basically it is the length over which integration is performed. Standard accelerator theory, intent on using models where everything is properly confused so as to fit snugly into schemes à la Guignard or Courant-Snyder style formalisms, does not make a big fuss about L, LC, and LD as PTC does. If the reader wants to see an example where all these variables are different, he should look at Sect. Q.5.8 or Figure 19.

So, why is L a difficult variable? For example, imagine a ring has been built, with a properly defined fibre: CHART, PATCH and ELEMENT(P) are defined and the ring closes. Now, suppose we decide to shrink a quadrupole by 1cm at both ends. How is PTC reacting to this if it is done through the polymorph L? Incorrectly, is the correct answer! In order for the L to be physically polymorphic, we need to use an object PATCH as in MAGNET\_PATCH to provide automatic patching on the factory bench. For speed reasons and also because this is pushing theology to its outer limit, we refrained from giving PTC this extra layer of automatic realism. **We will certainly regret it one day; but it can be added rather easily.**

Therefore, to emphasize the lack of true polymorphism of the variable ELEMENTP%L, it is kept out of POL\_BLOCK completely for the time being.

### O.1.3 Removing Parameters: KILL\_PARA

To ensure a complete removal of the knobs one uses the following call:

```
CALL KILL_PARA(PSR)
```

It is always prudent to terminate a routine using parametric dependence with a call to KILL\_PARA.

## O.2 Routines extended from EL(P) to FIBRE

There are few routines that are applicable to either the ELEMENT or/and the ELEMENTP. It is convenient to provide a FIBRE interface. We list these routines now.

### O.2.1 The Interface ADD: ADDP\_ANBN

The reader is invited to look at Sect. L.6 where it is discussed in detail.

### O.2.2 FIBRE\_POL: FIBRE=POL\_BLOCK

There is a routine which allows the syntax FIBRE=POL\_BLOCK. The reader probably thinks that this is the routine called during the layout operation of Sect. O.1.1; indeed one would just need to traverse the linked list associated to the LAYOUT and use this function. This is not the case. **FIBRE=POL\_BLOCK does not check the parentage of magnet.** It is there for the user to write his own more flexible and perhaps more dangerous scanning routines. Of course in standard lattices, all these things amount to the same thing.

### O.2.3 FIBRE\_BL: FIBRE=MUL\_BLOCK

This is discussed in detail in Sect. L.4.5.

### O.2.4 FIBRE\_WORK: FIBRE=WORK

This is also discussed in detail in Sect. L.4.5.

### O.2.5 MISALIGN\_FIBRE: FIBRE=X(6)

This subroutine applies misalignments to the fibre. We remind the reader that while it is possible to apply misalignments individually to ELEMENT and ELEMENTP, it makes little sense in practice. It can be called directly as “CALL MISALIGN\_FIBRE(FIBRE,X)” or with the (=) sign interface. Here X(1:3) contains the displacements and X(4:6) the angles of rotation.

## O.3 Copying all ELPs into ELs and Vice Versa

There are all sorts of reasons why one would like to copy all the ELEMENTs of an existing layout into their polymorphic versions or vice versa. Typically certain kinds of twiddling or fitting will use only one type of element. For example, a TPSA fit using polymorphic knobs would act first on the ELEMENTP. Obviously, if the fit is successful, the ELEMENTPs should be copied on the ELEMENTs. This is done with the syntax

```
CALL ELP_TO_EL(PSR)
```

Suppose instead that one has written an interactive fitting routine and decides that the fit is going nowhere. Then, the reverse call is possible

```
CALL EL_TO_ELP(PSR)
```

restoring all the ELEMENTPs to their pristine states presumably preserved in the ELEMENTs.

## O.4 Copying Layouts: COPY and EQUAL

It is possible to copy an entire layout into an new virgin linked list. This is done with the command:

```
CALL COPY(PSR,PSR2)
```

For the people whose brain works in reverse Polish notation, the reversed syntax of EQUAL can be used: COPY(PSR,PSR2) = EQUAL(PSR2,PSR).

With this command, a complete copy of the fibre structure is created: magnet, polymorphic magnet, and the existing chart are all faithfully duplicated. For the moment, this command works only on a standard layout.

## O.5 Standard Surveys

PTC is capable of executing a MAD-like standard survey. In that case the internal geometry of the magnet located in type MAGNET\_CHART is used to deduce the geometry of the fibre located in type CHART of the fibre. Again, in PTC, these things need not be the same at all.

### O.5.1 Full Standard Survey

The call to SURVEY(PSR) will do a standard MAD-like survey of the layout PSR. By default the first magnet is located at the origin of space. The three directions of space correspond to the local entrance chart of the first magnet.

### O.5.2 Partial Standard Survey

It is also possible to do a partial survey using the command

```
CALL SURVEY(PSR,3,14)
```

This will perform a survey from element 3 to 14. This generally will reproduce the result of the original survey unless some magnets were changed. In particular one may want to change the variable EL%TILTD in a vertical bump.

Obviously, if global coordinates are computed, the user should make sure that the ring closes and that all vertical bumps are correctly patched. Vertical bumps are particularly annoying since rotations do not commute in three dimensional space.

## O.6 Moving a Layout

There are two built-in routines for moving the lattice around space. These are useful if several beam lines are present.

### O.6.1 Rotating a Layout

A layout PSR can be rotated around a point OMEGA(3) by a set of angles ALPHA(3) operating in the standard PTC order using the command:

```
OMEGA=0.D0; ALPHA=0.D0; ALPHA (2)=0.1D0;  
CALL ROTATE(PSR,OMEGA,ALPHA)
```

### O.6.2 Moving a Layout

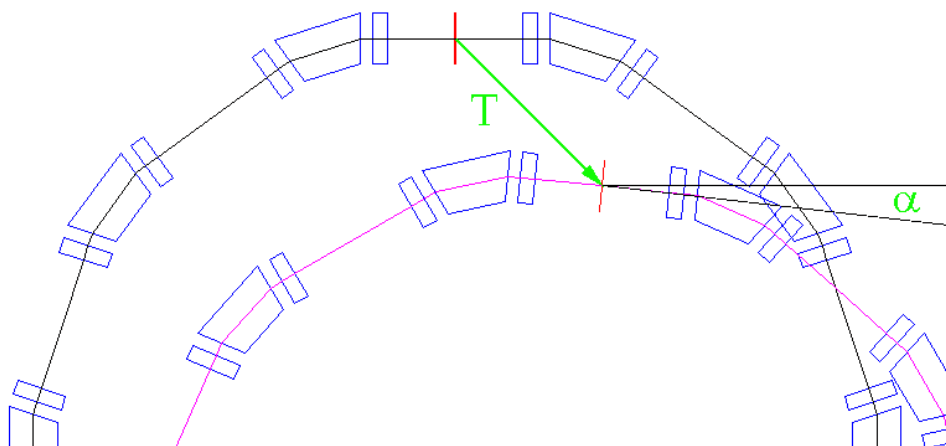


Figure 21: Rotating a Layout

One can also translate a layout with the syntax:

```
D=0.D0; D(1)=-5.D0; D(3)=5.D0;  
CALL TRANS(PSR,D)
```

The result of the rotation of Sect. O.6.1 followed by the above translation is displayed in Figure 21— an actual run of PTC on the PSR lattice.

## O.7 Routines of S\_FAMILY

The routines and their interfaces are listed here. There is not much to add to the previous discussion.

```
INTERFACE EL_TO_ELP  
  MODULE PROCEDURE EL_TO_ELP_L  
INTERFACE ELP_TO_EL  
  MODULE PROCEDURE ELP_TO_EL_L  
INTERFACE SURVEY  
  MODULE PROCEDURE SURVEY_ONE  
  MODULE PROCEDURE SURVEY_EXIST_PLANAR_L_NEW  
  MODULE PROCEDURE SURVEY_EXIST_PLANAR_IJ_NEW  
INTERFACE KILL_PARA  
  MODULE PROCEDURE KILL_PARA_L  
INTERFACE ADD  
  MODULE PROCEDURE ADDP_ANBN  
INTERFACE ASSIGNMENT (=)  
  MODULE PROCEDURE SETPOL_L  
  MODULE PROCEDURE FIBRE_WORK  
  MODULE PROCEDURE FIBRE_MIS  
  MODULE PROCEDURE FIBRE_POL  
  MODULE PROCEDURE FIBRE_BL  
  MODULE PROCEDURE BL_FIBRE  
  MODULE PROCEDURE WORK_FIBRE  
INTERFACE EQUAL  
  MODULE PROCEDURE COPY_LAYOUT  
INTERFACE COPY  
  MODULE PROCEDURE COPY_LAYOUT_I  
  MODULE PROCEDURE COPY_LAYOUT_IJ  INTERFACE TRANS
```

```

MODULE PROCEDURE TRANS_D
INTERFACE TRANSLATE
MODULE PROCEDURE TRANS_D
INTERFACE ROTATE
MODULE PROCEDURE ROT_A
SUBROUTINE ADDP_ANBN(EL,NM,F,V) ! EXTENDS THE ADD ROUTINES FROM THE ELEMENT(P) TO THE FIBRE
SUBROUTINE FIBRE_WORK(S2,S1) ! CHANGES THE ENERGY OF THE FIBRE AND TURNS THE ENERGY PATCH ON
SUBROUTINE WORK_FIBRE(S2,S1) ! SUCKS THE ENERGY OUT OF A FIBRE BY LOOKING AT ELEMENT
SUBROUTINE FIBRE_MIS(S2,S1) ! MISALIGNS FULL FIBRE; FILLS IN CHART AND MAGNET_CHART
SUBROUTINE ADJUST_INTERNAL(S2) ! ADJUSTS FRAMES OF MAGNET_CHART ON THE BASIS OF THE MISALIGNMENTS
SUBROUTINE COPY_INTERNAL(S2) ! PUTS THE FRAMES OF THE CHART INTO THOSE OF THE MAGNET_CHART
SUBROUTINE TRANS_D(R,D) ! TRANSLATES A LAYOUT
SUBROUTINE ROT_A(R,OMEGA,A) ! ROTATES A LAYOUT AROUND OMEGA BY A(3) IN STANDARD PTC ORDER
SUBROUTINE ROTATE_FIBRE(EL1,ELO) ! PUTS EL1 AT THE END OF ELO FOR SURVEY PURPOSES. ELO AND EL1 MUST HAVE SAME EL%DIR (STANDARD SURVEY)
SUBROUTINE FIBRE_BL(S2,S1) ! PUTS A NEW MULTIPOLE BLOCK INTO FIBRE. EXTENDS ELEMENT(P) ROUTINES TO FIBRES
SUBROUTINE BL_FIBRE(S2,S1) ! SUCKS THE MULTIPOLE OUT LOOKING AT ELEMENT
SUBROUTINE SURVEY_EXIST_PLANAR_IJ_NEW(PPLAN,I1,I2) ! STANDARD SURVEY FROM FIBRE #I1 TO #I2
SUBROUTINE SURVEY_EXIST_PLANAR_L_NEW(PPLAN) ! CALLS ABOVE ROUTINE FROM FIBRE #1 TO #PLAN%N : STANDARD SURVEY
SUBROUTINE SURVEY_ONE(C) ! SURVEYS A SINGLE ELEMENT FILLS IN CHART AND MAGNET_CHART; LOCATES ORIGIN AT THE ENTRANCE OR EXIT
SUBROUTINE COPY_LAYOUT(R2,R1) ! COPY STANDARD LAYOUT ONLY
SUBROUTINE COPY_LAYOUT_IJ(R1,I,J,R2) ! COPY PIECES OF A STANDARD LAYOUT FROM FIBRE #I TO #J
SUBROUTINE COPY_CHART(R1,R2) ! JUST COPIES THE CHARTS AND THE PATCHES IN IDENTICAL LAYOUTS
SUBROUTINE COPY_LAYOUT_I(R1,R2) ! COPIES IN THE COPY ORDER RATHER THAN THE LAYOUT ORDER
SUBROUTINE KILL_PARA_L(R) ! RESETS ALL THE PARAMETERS IN A LAYOUT : REMOVE POLYMORPHIC KNOBS
SUBROUTINE FIBRE_POL(S2,S1) ! SET POLYMORPH IN A FIBRE UNCONDITIONALLY
SUBROUTINE SETPOL_L(R,B) ! SET POLYMORPH IN A FULL LAYOUT ONLY IF THE MAGNET IS A PRIMITIVE PARENT
SUBROUTINE EL_TO_ELP_L(R) ! COPY ALL PRIMITIVES ELEMENT INTO ELEMENTP
SUBROUTINE ELP_TO_EL_L(R) ! COPY ALL PRIMITIVES ELEMENTP INTO ELEMENT

```



## P Sm\_TRACKING.f90

This file contains the module S\_TRACKING. This module contains the fibre tracking routines and the layout tracking routines built from them.

### P.1 TRACK for a Layout

In PTC, this is just a do-loop over the fibres.

```
SUBROUTINE TRACK_LAYOUT_FLAG_P(R,X,I1,I2,K) ! TRACKS POLYMORPHS FROM I1 TO I2 IN STATE K
  IMPLICIT NONE
  TYPE(LAYOUT),INTENT(INOUT):: R ;TYPE(REAL_8), INTENT(INOUT):: X(6);
  INTEGER, INTENT(IN):: I1,I2; TYPE(INTERNAL_STATE) K;
  INTEGER J;

  TYPE (FIBRE), POINTER :: C

  CALL MOVE_TO(R,C,MOD_N(I1,R%N))

  J=I1

  DO WHILE(J<I2.AND.ASSOCIATED(C))

    CALL TRACK(C,X,K,R%CHARGE)

    C=>C%NEXT
    J=J+1
  ENDDO

END SUBROUTINE TRACK_LAYOUT_FLAG_P
```

### P.2 TRACK for a Fibre

What follows is the routine used in tracking polymorphs with suitable comments. There is not much to add here except, perhaps, that one should notice that energy patching is done on-the-fly in PTC. In the frontal energy patch, PTC checks the previous fibre. PTC gives “priority” to the exit patch. By this we mean that if two fibres follow one another, than patching is performed at the end of the first fibre rather than at the front of the second fibre.

```
SUBROUTINE TRACK_FIBRE_P(C,X,K,CHARGE)
  IMPLICIT NONE
  TYPE(FIBRE),TARGET,INTENT(INOUT):: C
  TYPE(REAL_8), INTENT(INOUT):: X(6)
  INTEGER, TARGET, INTENT(IN) :: CHARGE
  TYPE(INTERNAL_STATE), INTENT(IN) :: K
  LOGICAL OU,PATCH,PATCHT,PATCHG,PATCHE
  TYPE (FIBRE), POINTER :: CN
  REAL(DP), POINTER :: PO,BO

  ! NEW STUFF WITH KIND=3: KNOB OF FPP IS SET TO TRUE IF NECESSARY
  IF(K%PARA_IN ) KNOB=.TRUE.
  ! END NEW STUFF WITH KIND=3

  ! DIRECTIONAL VARIABLE AND CHARGE IS PASSED TO THE ELEMENT
  C%MAGP%P%DIR=>C%DIR
  C%MAGP%P%CHARGE=>CHARGE
  !

  ! PASSING THE STATE K TO THE ELEMENT
  C%MAGP=K
  !FRONTAL PATCH
  IF(ASSOCIATED(C%PATCH)) THEN
    PATCHT=C%PATCH%TIME ;PATCHE=C%PATCH%ENERGY ;PATCHG=C%PATCH%PATCH;
  ELSE
    PATCHT=.FALSE. ; PATCHE=.FALSE. ;PATCHG=.FALSE. ;
  ENDIF
  ! ENERGY PATCH
```

```

IF (PACHE) THEN
  NULLIFY (PO); NULLIFY (BO);
  CN=>C%PREVIOUS
  IF (ASSOCIATED(CN)) THEN ! ASSOCIATED
    IF (.NOT.CN%PATCH%ENERGY) THEN ! NO NEED TO PATCH IF PATCHED BEFORE
      PO=>CN%MAGP%P%POC
      BO=>CN%MAGP%P%BETA0

      X(2)=X(2)*PO/C%MAGP%P%POC
      X(4)=X(4)*PO/C%MAGP%P%POC
      IF (C%MAGP%P%TIME) THEN
        X(5)=SQRT(ONE+TWO*X(5)/BO+X(5)**2) !X(5) = 1+DP/POC_OLD
        X(5)=X(5)*PO/C%MAGP%P%POC-ONE !X(5) = DP/POC_NEW
        X(5)=(TWO*X(5)+X(5)**2)/(SQRT(ONE/C%MAGP%P%BETA0**2+TWO*X(5)+X(5)**2)+ONE/C%MAGP%P%BETA0)
      ELSE
        X(5)=(ONE+X(5))*PO/C%MAGP%P%POC-ONE
      ENDIF
    ENDIF ! NO NEED TO PATCH
  ENDIF ! ASSOCIATED

ENDIF

! POSITION PATCH
IF (PATCHG) THEN
  PATCH=ALWAYS_EXACT_PATCHING.OR.C%MAGP%P%EXACT
  CN=>C%PREVIOUS
  IF (ASSOCIATED(CN)) THEN
    X(1)=CN%DIR*C%DIR*X(1); X(2)=CN%DIR*C%DIR*X(2);
  ENDIF
  CALL ROT_YZ(C%PATCH%A_ANG(1), X, C%MAGP%P%BETA0, PATCH, C%MAGP%P%TIME)
  CALL ROT_XZ(C%PATCH%A_ANG(2), X, C%MAGP%P%BETA0, PATCH, C%MAGP%P%TIME)
  CALL ROT_XY(C%PATCH%A_ANG(3), X, PATCH)
  CALL TRANS(C%PATCH%A_D, X, C%MAGP%P%BETA0, PATCH, C%MAGP%P%TIME)
ENDIF

! TIME PATCH
IF (PATCHT) THEN
  X(6)=X(6)-C%PATCH%A_T
ENDIF

! MISALIGNMENTS AT THE ENTRANCE
IF (C%MAGP%MIS) THEN
  OU = K%EXACTMIS.OR.C%MAGP%EXACTMIS
  CALL MIS_FIB(C, X, OU, .TRUE.)
ENDIF

! *****
! THE ACTUAL MAGNET PROPAGATOR AS IT WOULD APPEAR IN A STANDARD CODE

CALL TRACK(C%MAGP, X)

! *****

! MISALIGNMENTS AT THE EXIT
IF (C%MAGP%MIS) THEN
  CALL MIS_FIB(C, X, OU, .FALSE.)
ENDIF

!EXIT PATCH
! TIME PATCH
IF (PATCHT) THEN
  X(6)=X(6)-C%PATCH%B_T
ENDIF

! POSITION PATCH
IF (PATCHG) THEN
  CN=>C%NEXT
  IF (ASSOCIATED(CN)) THEN
    X(1)=CN%DIR*C%DIR*X(1); X(2)=CN%DIR*C%DIR*X(2);
  ENDIF
  CALL ROT_YZ(C%PATCH%B_ANG(1), X, C%MAGP%P%BETA0, PATCH, C%MAGP%P%TIME)
  CALL ROT_XZ(C%PATCH%B_ANG(2), X, C%MAGP%P%BETA0, PATCH, C%MAGP%P%TIME)

```

```

CALL ROT_XY(C%PATCH%B_ANG(3),X,.TRUE.)
CALL TRANS(C%PATCH%B_D,X,C%MAGP%P%BETA0,PATCH,C%MAGP%P%TIME)
ENDIF

! ENERGY PATCH
IF(PATCHE) THEN
NULLIFY(P0);NULLIFY(B0);
CN=>C%NEXT
IF(.NOT.ASSOCIATED(CN)) CN=>C
PO=>CN%MAGP%P%POC
BO=>CN%MAGP%P%BETA0      X(2)=X(2)*C%MAGP%P%POC/PO
X(4)=X(4)*C%MAGP%P%POC/PO
IF(C%MAGP%P%TIME)THEN
X(5)=SQRT(ONE+TWO*X(5)/C%MAGP%P%BETA0+X(5)**2)  !X(5) = 1+DP/POC_OLD
X(5)=X(5)*C%MAGP%P%POC/PO-ONE  !X(5) = DP/POC_NEW
X(5)=(TWO*X(5)+X(5)**2)/(SQRT(ONE/BO**2+TWO*X(5)+X(5)**2)+ONE/BO)
ELSE
X(5)=(ONE+X(5))*C%MAGP%P%POC/PO-ONE
ENDIF
ENDIF

! ELEMENT IS RESTAURED TO THE DEFAULT STATE
C%MAGP=DEFAULT
! DIRECTIONAL VARIABLE AND CHARGE ARE ELIMINATED
NULLIFY(C%MAGP%P%DIR)
NULLIFY(C%MAGP%P%CHARGE)

! KNOB IS RETURNED TO THE PTC DEFAULT
! NEW STUFF WITH KIND=3
KNOB=.FALSE.
! END NEW STUFF WITH KIND=3

END SUBROUTINE TRACK_FIBRE_P

```

### P.3 The MIS\_FIB Routines: Misaligning a Magnet

These routines use the arrays computed in Rotation\_mis based on the compressed magnet idea and the associated operators refactorized in the standard PTC order. The routines are quite transparent; here is the routine MIS\_FIBR, which acts on double precision numbers (ELEMENT):

```

SUBROUTINE MIS_FIBR(C,X,OU,ENTERING)
! MISALIGNS REAL FIBRES IN PTC ORDER FOR FORWARD AND BACKWARD FIBRES
TYPE(FIBRE),INTENT(INOUT):: C
REAL(DP), INTENT(INOUT):: X(6)
INTEGER J
LOGICAL,INTENT(IN):: OU,ENTERING

IF(C%DIR==1) THEN  ! FORWARD PROPAGATION
IF(ENTERING) THEN
CALL ROT_YZ(C%CHART%ANG_IN(1),X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)  ! ROTATIONS
CALL ROT_XZ(C%CHART%ANG_IN(2),X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)  ! ROTATIONS
CALL TRANS(C%CHART%D_IN,X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)  ! TRANSLATION
ELSE
CALL ROT_YZ(C%CHART%ANG_OUT(1),X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)  ! ROTATIONS
CALL ROT_XZ(C%CHART%ANG_OUT(2),X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)  ! ROTATIONS
CALL ROT_XY(C%CHART%ANG_OUT(3),X,OU)
CALL TRANS(C%CHART%D_OUT,X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)  ! TRANSLATION
ENDIF
ELSE
IF(ENTERING) THEN  ! BACKWARD PROPAGATION
C%CHART%D_OUT(1)=-C%CHART%D_OUT(1)
C%CHART%D_OUT(2)=-C%CHART%D_OUT(2)
C%CHART%ANG_OUT(3)=-C%CHART%ANG_OUT(3)
CALL TRANS(C%CHART%D_OUT,X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)  ! TRANSLATION
CALL ROT_XY(C%CHART%ANG_OUT(3),X,OU)
ENDIF
ENDIF

```

```

CALL ROT_XZ(C%CHART%ANG_OUT(2),X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)
CALL ROT_YZ(C%CHART%ANG_OUT(1),X,C%MAG%P%BETA0,OU,C%MAG%P%TIME) ! ROTATIONS
C%CHART%D_OUT(1)=-C%CHART%D_OUT(1)
C%CHART%D_OUT(2)=-C%CHART%D_OUT(2)
C%CHART%ANG_OUT(3)=-C%CHART%ANG_OUT(3)

ELSE

C%CHART%D_IN(1)=-C%CHART%D_IN(1)
C%CHART%D_IN(2)=-C%CHART%D_IN(2)
C%CHART%ANG_IN(3)=-C%CHART%ANG_IN(3)
CALL TRANS(C%CHART%D_IN,X,C%MAG%P%BETA0,OU,C%MAG%P%TIME) ! TRANSLATION
CALL ROT_XY(C%CHART%ANG_IN(3),X,OU)
CALL ROT_XZ(C%CHART%ANG_IN(2),X,C%MAG%P%BETA0,OU,C%MAG%P%TIME)
CALL ROT_YZ(C%CHART%ANG_IN(1),X,C%MAG%P%BETA0,OU,C%MAG%P%TIME) ! ROTATIONS
C%CHART%D_IN(1)=-C%CHART%D_IN(1)
C%CHART%D_IN(2)=-C%CHART%D_IN(2)
C%CHART%ANG_IN(3)=-C%CHART%ANG_IN(3)

ENDIF
ENDIF
END SUBROUTINE MIS_FIBR

```

The first part of MIS\_FIBR is just the plain dynamical rendition of the SO(3) operators. The second part of the routine corresponds to reverse propagation ( $C\%DIR=-1$ ). Normally one would expect that the angles and the translations simply reverse sign. **This is not the case because in PTC we adopted the accelerator physics convention of leaving the drifts alone by keeping integration lengths positive. In doing this, the vector potential  $A_z$  for example changes sign although  $(A_x, A_y)$  stays invariant. We may regret this choice! What else is new? ^\_^**

The reader uncertain of the origin of all these conventions should go back to first principles, namely, by deriving the “s”-dependent Hamiltonian  $-p_z$  in a straight magnet from the original time dependent one. By comparing the cases of forward propagation ( $p_z > 0$ ) and backward propagation ( $p_z < 0$ ), the above assertions will start to make sense.

## P.4 The Variables ALWAYS\_EXACT\_PATCHING

The logical variable ALWAYS\_EXACT\_PATCHING is joined with the logical  $C\%MAG\%P\%EXACT$  using the operator .OR. If ALWAYS\_EXACT\_PATCHING is true, then the patching is always done exactly. This is the safe situation. If it is false, then the logical  $C\%MAG\%P\%EXACT$  will determine the method used for patching.

## P.5 The Routines of S\_TRACKING

There is not much to say; here are the routines:

```

INTERFACE TRACK
MODULE PROCEDURE TRACK_LAYOUT_FLAG_R
MODULE PROCEDURE TRACK_LAYOUT_FLAG_P
MODULE PROCEDURE TRACK_LAYOUT_FLAG_S
MODULE PROCEDURE TRACK_LAYOUT_FLAG_R1
MODULE PROCEDURE TRACK_LAYOUT_FLAG_P1
MODULE PROCEDURE TRACK_LAYOUT_FLAG_S1
MODULE PROCEDURE TRACK_FIBRE_R
MODULE PROCEDURE TRACK_FIBRE_P
MODULE PROCEDURE TRACK_FIBRE_S
INTERFACE MIS_FIBR
MODULE PROCEDURE MIS_FIBR
MODULE PROCEDURE MIS_FIBP
MODULE PROCEDURE MIS_FIBS
SUBROUTINE TRACK_LAYOUT_FLAG_R1(R,X,II1,K) ! TRACKS DOUBLE PRECISION FROM II1 TO THE END OR BACK TO II1 IF CLOSED
SUBROUTINE TRACK_LAYOUT_FLAG_P1(R,X,II1,K) ! TRACKS POLYMORPHS FROM II1 TO THE END OR BACK TO II1 IF CLOSED
SUBROUTINE TRACK_LAYOUT_FLAG_S1(R,X,II1,K) ! TRACKS ENVELOPE FROM II1 TO THE END OR BACK TO II1 IF CLOSED
SUBROUTINE TRACK_LAYOUT_FLAG_R(R,X,I1,I2,K) ! TRACKS DOUBLE FROM I1 TO I2 IN STATE K
SUBROUTINE TRACK_LAYOUT_FLAG_P(R,X,I1,I2,K) ! TRACKS POLYMORPHS FROM I1 TO I2 IN STATE K
SUBROUTINE TRACK_LAYOUT_FLAG_S(R,X,I1,I2,K) ! TRACKS ENVELOPES FROM I1 TO I2 IN STATE K
SUBROUTINE TRACK_FIBRE_R(C,X,EXACTMIS,STATE,DIR,CHARGE,MIS) ! TRACKS DOUBLE PRECISION FROM II1 TO THE END OR BACK TO II1 IF CLOSED
SUBROUTINE TRACK_FIBRE_P(C,X,EXACTMIS,STATE,DIR,CHARGE,MIS) ! TRACKS DOUBLE PRECISION FROM II1 TO THE END OR BACK TO II1 IF CLOSED
SUBROUTINE TRACK_FIBRE_S(C,X,EXACTMIS,STATE,DIR,CHARGE,MIS) ! TRACKS DOUBLE PRECISION FROM II1 TO THE END OR BACK TO II1 IF CLOSED
SUBROUTINE MIS_FIBR(C,X,OU,ENTERING)
SUBROUTINE MIS_FIBP(C,X,OU,ENTERING) ! MISALIGNS POLYMORPHIC FIBRES IN PTC ORDER FOR FORWARD AND BACKWARD FIBRES
SUBROUTINE MIS_FIBS(C,Y,OU,ENTERING) ! MISALIGNS ENVELOPE FIBRES IN PTC ORDER FOR FORWARD AND BACKWARD FIBRES

```

## Q Sn\_MAD\_LIKE.f90

This is based on a suggestion of Hiroshi Nishimura of LBNL. Since FORTRAN90 supports operator overloading and procedure interface, and since the user of PTC is assumed to have a compiler at hand, why not use these tools to create an input language? The advantage is that one can avoid writing a complex parser. Here we describe this input in detail.

**N.B. Of course the integration of PTC in MAD-X will take advantage of the MAD input parser. Presumably it will be extended to take full advantage of PTC's novel features.**

### Q.1 Example of the PSR Revisited

Before we start the description, let us look again at the PSR. Here we put all the calls to the MAD-like interface in a routine called INPUT\_PSR. The main program is simply:

```
PROGRAM RUN_PSR
USE S_TRACKING
IMPLICIT NONE
INTEGER ND2,NPARA
TYPE(REAL_8) Y(6)
TYPE(NORMALFORM) NORMAL
TYPE(LAYOUT) PSR
REAL(DP) X(6)

CALL INPUT_PSR(PSR,2.72D0,-1.92D0);CALL SURVEY(PSR);

X=0.d0; CALL FIND_ORBIT(PSR,X,1,DEFAULT) ! DEFAULT IS A STATE

CALL INIT(DEFAULT,3,0,BERZ,ND2,NPARA)
CALL ALLOC(Y);CALL ALLOC(NORMAL); ! ALLOCATE VARIABLES
Y = NPARA
Y = X

CALL TRACK(PSR,Y,1,DEFAULT)

NORMAL = Y
WRITE(6,*) NORMAL%TUNE
CALL DAPRINT(NORMAL%DHDJ%V(1),6);CALL DAPRINT(NORMAL%DHDJ%V(2),6);

CALL KILL(Y);CALL KILL(NORMAL);

END PROGRAM RUN_PSR
```

It is followed by the subroutine INPUT\_PSR.

```
SUBROUTINE INPUT_PSR(PSR,KFO,KDO)
USE MAD_LIKE
IMPLICIT NONE
REAL(DP) KFO,KDO
TYPE(LAYOUT) PSR
TYPE(LAYOUT) CELL,MYRING
TYPE(FIBRE) D1,QD,QF,D2,B
REAL(DP) KF,KD,ANG,BRHO
INTEGER INTEGRATION_METHOD

CALL MAKE_STATES(.FALSE.)
EXACT_MODEL = .TRUE.
DEFAULT = DEFAULT + NOCAVITY + EXACTMIS
```

```

CALL UPDATE_STATES
MADLENGTH = .FALSE.

ANG = (TWOPI * 36.D0 / 360.D0)
BRHO = 1.2D0 * (2.54948D0 / ANG)

INTEGRATION_METHOD = 6
CALL SET_MAD(BRHO = BRHO, METHOD = INTEGRATION_METHOD, STEP = 10)
MADKIND2 = KIND2

KF = KFO/BRHO; KD = KDO/BRHO;

D1 = DRIFT("D1", 2.28646D+00); D2 = DRIFT("D2", 0.45D+00);
QF = QUADRUPOLE("QF", 0.5D0, KF); QD = QUADRUPOLE("QD", 0.5D0, KD);
B = RBEND("B", 2.54948D0, ANG)

CELL = D1 + QD + D2 + B + D2 + QF + D1; MYRING = 10 * CELL;

PSR = .RING.MYRING

CALL CLEAN_UP

END SUBROUTINE

```

Let us concentrate on the subroutine INPUT\_PSR. First we recognize a bunch of fibres. They are the fundamental building blocks of a beam line. They will help us construct the ideal PSR, called MYRING here. It suffices to change QF and automatically all the QFs change in MYRING. Actually this is a very bad way to say it: there is only one QF and every occurrence of it in MYRING points to it. The routine which appends fibres (APPEND\_MAD\_LIKE) is similar in spirit to the routine APPEND\_POINT discussed in Sect. N in the context of recirculators and Siamese rings.

After the ideal object is created, we need to create a real layout, that is to say, one where QF is cloned. This is done by the commands PSR=.RING.MYRING for a ring and with PSR=.LINE.MYRING for a single pass system.

We will now list these simple operations.

## Q.2 Operations on types LAYOUT and FIBRE

The goal of the MAD-like module is to create a layout which represents the ideal layout to be tracked. It does so by defining operations on FIBREs and LAYOUTs. In what follows EL represent a FIBRE and BL a layout.

### Q.2.1 EL+EL

The following operation creates a layout QF\_TOTAL of 2 elements:

```
QF_TOTAL = QF_ENT + QF_EXI
```

In this case we took two “quadrupoles” and join them like Siamese twins. Actually such an operation is discouraged. Of course the addition of elements is used all the time in creating layouts.

### Q.2.2 EL+BL and BL+EL (BL stands for a layout)

Using the layout created in Sect. Q.2.1, we can imagine the list HALF\_CELL:

```
HALF_CELL = QF_TOTAL + D1;
```

The following equivalent syntax

```
HALF_CELL = QF_ENT + QF_EXI + D1;
```

also uses a temporary layout since it is equivalent to the syntax

```
HALF_CELL = (QF_ENT + QF_EXI) + D1;
```

### Q.2.3 BL+BL

This is the simple addition of two layouts.

### Q.2.4 N\*EL and N\*BL

This allows for the repetition of several elements. Thus the layout

```
TWO_HALF_CELL = QF_ENT + QF_EXI + D1 + QF_ENT + QF_EXI + D1;
```

can be obtained using

```
TWO_HALF_CELL = 2 * (QF_ENT + QF_EXI + D1);
```

It is also possible to multiply a single element.

### Q.2.5 -BL

It is possible to reverse a layout using the minus sign. This works as a unary operator

```
CELL = (QF + D1 + QD)
CELLI = -CELL
TOTAL_CELL = CELL + CELLI
```

or equivalently as a binary operator

```
CELL = (QF + D1 + QD)
TOTAL_CELL = CELL - CELL
```

## Q.3 Am I a Dumb or Smart User?

The first thing to understand about the MAD-like input is that it is a “dumb” user interface. Those familiar with the code TRACYII will know that this was perhaps the first code to support the concept of dumb-user and smart-user interfaces. In a dumb user interface, we do not provide all that is possible for a magnet. TRACYII was based on the belief that a dumb user interface should be built on the foundation of a smart user interface. In this way complex situations could always be handled. This was so successful that, in the 2 years of the PEPB design, Robin and Bengtsson recompiled TRACYII no more than 2 or 3 times. PTC has only this MAD-like input as a dumb-user interface. However, whatever improvements are made to the dumb side of things, we adhere to the TRACYII idea that the core of the code is never to be compromised by a desire for a “Joe-Six-Packs” interface.

In the case of TRACYII, this was realized by separating the lattice input file (dumb user) from the command input file (smart user). This idea, originally from Nishimura, was turned into an uncompromising product by Bengtsson. In PTC the same can be achieved by stripping all the core routines from any dumb user idiosyncracies. One example common to TRACYII and PTC is the absence of quadrupoles in the core. The reader never saw the word quadrupoles in our list of magnets: we have combined function magnets with arbitrarily high multipoles in them. Thus a routine to handle quadrupoles cannot exist in the core, but it can certainly exist in the dumb interface, where it invokes the more general element.

Here we will use the RBEND command to show the limitations of a dumb user interface. Consider a rectangular bend of angle ANG and arc length L, then in the MAD-LIKE interface this object could be created simply by the command

```
CALL MAKE_STATES(.FALSE.)
EXACT_MODEL = MYCHOICE
DEFAULT = DEFAULT + NOCAVITY + EXACTMIS
CALL UPDATE_STATES
MADLENGTH = .FALSE.
```

```

      .
      .
      .
B = RBEND("B",L,ANG)

```

This would create a rectangular bend B whose layout angle is exactly ANG. If MYCHOICE is false, then the origin of phase space will also be sent unto itself under the action of B. This is the result of traditional codes.

But what if MYCHOICE is true? If the integration is good enough or if the global DRIFT\_KICK is false, then for all practical purposes, the origin would also be invariant. But consider a worse case, namely, let us introduce a quadrupole component into this rectangular bend. The MAD-like interface allows a “dumb-user” way:

```

k = 0.001d0
B = RBEND("B", L,ANG).Q.K

```

The situation here is again trivial in the fake expanded Hamiltonian. However for MYCHOICE=true, things are far more complex. Here PTC will simply use ANG as the nominal layout angle and will compute the layout B0 associated with it. It will then put this B0 in BN(1). So far, so good, this is simply the standard case. However the addition of K puts K into BN(2). Clearly the origin of phase space is not preserved under B.

What does the user want? What should we put in PTC as a default? This could be argued ad infinitum since it becomes an important issue for small machines when details are sufficiently important that the “dumb user” should elevate himself to the level of “smart user” or simply get out of the kitchen. There is no way, in the absence of a concrete problem, that one can program a stupid-proof code. In any event, we have neither the patience nor the intelligence to do it.

As an example let us suppose that in the particular problem under consideration, we would like the angle of the layout and the angle of the trajectory to be identical when a “design K” is thrown in. Certainly the “smart user” can program a new routine called “FIXBEND” which will scan the lattice to refit BN(1) to achieve the desired result. Then, he and his colleague can revert to the “dumb user” status until someone complains.

One must understand that the core of PTC is like a real machine. If a quadrupole coil is added along the body of a rectangular bend in a real ring, to achieve some focusing, the magnet will not miraculously re-adjust its bending field to ensure closure. Someone would have to write some control algorithm to adjust the main B-field. PTC behaves the same way. The more exact the model, the more painful it becomes for the dumb user; but it is more realistic and thus easier for the slightly more informed user. The code simply behaves like a real machine whenever possible.

Obviously if one wants to try a smarter interface, or link the core routine with a true parser, or with a Mathematica-like interface (as in SAD), be our guest. It is certainly a major amount of work, beyond the skills of the authors, to create a powerful fool-proof interface. Actually, as mentioned already, PTC will be integrated into MAD-X; the details of this and what it really means is still an issue not fully settled. MAD-X will probably have a flexible and powerful dumb-user interface.

## Q.4 The Rules

### Q.4.1 Making the States and the Logicals: MAD and MADLENGTH

As in a regular run of PTC, the states must have been created. Thus at a very minimum one must have called the routine MAKE\_STATES. The following logicals, defaulted to false (i.e. to PTC internal definition), should be set to the desired value prior to the creation of elements:

```

MAD = .TRUE.
MADLENGTH = .TRUE.

```

The logical MAD forces the multipole definition of MAD; MADLENGTH refers to the Cartesian length in a rectangular bend being used rather than the arc length. This has been discussed in some detail in Sect. I item 11 & 12.



### Q.4.2 The Subroutine SET\_MAD and MADKIND2 (MADTHICK)

This subroutine must be executed before defining any element. There are two interfaces. The first one is

```
CALL SET_MAD(ENERGY,KINETIC,POC,BRHO,BETA,NOISY,METHOD,STEP)
MADKIND2 = KIND2 ! Or equivalently MADTHICK=DRIFT_KICK_DRIFT
```

Here ENERGY is the total energy, KINETIC is the kinetic energy, and POC is the momentum, all in GeV. BRHO is  $p_0/q$ , BETA is  $v_0/c$ , NOISY is true if one wants the code to send a lot of garbage to the screen. METHOD is the default order of the method for the “thick” elements (2,4, or 6) and STEP is the default number of integration steps.

If all the argument are in the call statement, then PTC looks for a negative input for either ENERGY, KINETIC,POC, BRHO or BETA. The code will then compute the remaining variables. The variables are private. However the preferred method to call SET\_MAD uses the OPTIONAL construct of FORTRAN90. So, for example,

```
CALL SET_MAD(BRHO=BRHO,NOISY=.FALSE.,METHOD=INTEGRATION_METHOD,STEP=10)
```

One selects a single variable amongst the energy-like variables and use the keyword assignment KEYWORD=VALUE. Their numerical value can be retrieved using

```
CALL GET_ENERGY(ENERGY,GET_ENERGY,BRHO,BETA,POC)
```

The routine GET\_ENERGY is a passive routine which fetches the private variables set by SET\_MAD. Generally in PTC we do not recommend messing around the lattice with the MAD-like routine. The philosophy is like that of TRACY. In the MAD-like input one concentrates on the ideal lattice. Strange things such as changes in the reference energy of magnets should be handled by a lucidly written user routine. Keep any code in the MAD-like phase as simple as possible. Remember that generally a magnet at this point is really a template which is cloned when the layout is created, like a factory prototype.

Finally MADKIND2 = KIND2 (or MADTHICK=DRIFT\_KICK\_DRIFT) tells PTC what kind of integration method to use. This is supplemented with the global logical variable DRIFT\_KICK which affects type TEAPOT (see Sect. K.4.9) and STREX (see Sect. K.4.12).

### Q.4.3 Logicals and Integer Flags

The following global constants can be modified anytime during an input:

- FIBRE\_DIR controls the direction of propagation of the fibre. It is defaulted to one: forward propagation. At this point the MAD-like input and the standard survey does not handle a mixture of propagation direction. So please beware. However there is no impediment in PTC structures to mix forward and backward propagators as Sect. B.1 exemplifies.
- FIBRE\_FLIP is a logical defaulted to true. If true, the various strengths of a reversed fibre are reversed so as to ensure that the magnet has the same propagation properties as the forward fibre. For example a planar clockwise lattice can be turned into a counter-clockwise lattice if FIBRE\_FLIP is true. If false, then typically a stable bend becomes unstable, as a particle enters from the exit face and therefore bends outwards.
- MAD=.true. forces the input of the multipoles following the MAD definition, defaulted false.
- MADLENGTH=.true. forces the Cartesian length for the rectangular bend, defaulted false. Notice that this has no effect on survey since EL%P%LC and EL%P%LD are properly computed.
- MADKIND2=kind2, kind6, or kind7. This chooses the type of integrator for the general magnet, defaulted to kind2. The nomenclature is confusing and is a left-over of Small\_Code which is the PTC prototype. In reality the MAD-like input will jump from KIND2, KIND6, KIND7, KIND10 and KIND16 pretty much on its own on the basis of the internal logic of these types. See Sect. Q.4.4 for some explanations.
- MADKIND3N=kind3 or kind8. Kind3 is always safe. Kind8 is the normal “SMI” of SixTrack. One should be very careful using this. Certain operators we will describe later are not compatible with kind8. Defaulted to kind3.

- MADKIND3S=kind3 or kind9. Kind3 is always safe. Kind9 is the skew “SMI” of SixTrack. One should be very careful using this. Certain operators we will describe later are not compatible with kind9. Defaulted to kind3.
- EXACT\_MODEL forces the code to use exact square root Hamiltonian.
- METD (2,4,6) is the method of integration. It is defaulted to 2. It is set by SET\_MAD and can be locally changed by a simple assignment such as METD=4 for example. It is also changed by a call to THIN\_LENS once a layout is produced. Kind6 supports only METD=2. Some of the magnet types also support the user defined METHOD\_i, i=1,2,3,4,F. These five methods must be defined by the user after a MAKE\_STATE call. There are no defaults for them. PTC will throw an exception or crash if there is an attempt to use them prior to definition.
- NSTD this is the number of integration steps. It can be changed to any number greater or equal to one. It is also modified by a call to THIN\_LENS.
- LIKEMAD is a private logical controlling the appearance of wedges in type STREX. See Figure 19 for a pictorial explanation.

#### Q.4.4 More on MADKINDs

PTC started as Small\_Code where a lot of the integrators were tested. The nomenclature of the MAD-like input, particularly the abuse of the word “KIND2” to mean anything DRIFT-KICK-DRIFT was fine in Small\_code but is confusing in PTC. If a user prefers a more readable code, he may want to use the following names, defined through assignments and pointing operations in module S\_STATUS:

```
DRIFT_KICK_DRIFT = KIND2
MATRIX_KICK_MATRIX = KIND7
KICK_SIXTRACK_KICK = KIND6
MADTHICK => MADKIND2
MADTHIN_NORMAL => MADKIND3N
MADTHIN_SKEW => MADKIND3S
```

This change of nomenclature is useful simply because “MADKIND2” and “KIND2” in the MAD-like language of PTC do not refer to KIND2 anymore. So here are the rules.

First if the EXACT\_MODEL is false, then PTC acts in the old way: KIND2 means KIND2, KIND6 means KIND6 and KIND7 means KIND7.

Things gets more complex if EXACT\_MODEL is true. If a straight element is generated ( $EL\%P\%B0=0$ ), like a quadrupole, then MADTHICK=DRIFT\_KICK\_DRIFT will use STREX (KIND16). KICK\_SIXTRACK\_KICK will use KIND7 and KICK\_SIXTRACK\_KICK will use KIND6.

If one the other hand we are dealing with a BEND, either RBEND or SBEND, then things get more messy. DRIFT\_KICK\_DRIFT will direct the element to either KIND10 (TEAPOT) if SBEND or KIND16 (STREX) if RBEND. The split used in this case will be controlled by the global parameter DRIFT\_KICK. However if either KICK\_SIXTRACK\_KICK or KICK\_SIXTRACK\_KICK is used for MADTHICK, then we have a problem because PTC has no integration methods for a split where the expanded Hamiltonian is solved exactly when EXACT\_MODEL is true and  $EL\%P\%B0/=0$ . One could in the case of a sector bend generate such a split, but it is not supported by PTC. In the case of the true RBEND, it is not possible because there are no analytic solutions to our knowledge in the presence of a quadrupole component. So what does PTC do? In this case PTC switches from KIND6 or KIND7 to KIND10 or KIND16 automatically. In addition, since the user intended to have the body as exact as possible, PTC chooses the integration split using the constant bend solution (driftkick=.FALSE.) rather than the drift-kick split.

Users can overwrite these defaults of course. In addition, MAD-X will probably have an assortment of default and global commands providing more “automatic flexibility” than PTC as well as PTC’s hands-on capability to override any annoying default.

#### Q.4.5 Cleaning Up

Then at the end, we recommend killing the fibres used in the MAD-LIKE input.

```

PSR=.RING.PSR
.
.
CALL CLEAN_UP

```

This clean up must happen after the true layout has been created. PTC keeps a linked list (MAD\_LIST) containing all the fibres created by commands such as “D = DRIFT(“D”,L = 1.d0).” Thus, after these fibres have been cloned into the true layout, one can exterminate them.

## Q.5 The Elements

We now describe the calls to create a fibre. **All real quantities must be double precision!**

### Q.5.1 The Marker

This is the simplest element of all. For example an element with name SYM is created with

```
SYM = MARKER("SYM")
```

### Q.5.2 The Drift

```
D = DRIFT("D",L)
```

The drift is affected by the EXACT\_MODEL variable. The variable  $L$  is a keyword of the optional construct. Thus one can also write

```
D = DRIFT("D",L = 0.5d0)
```

### Q.5.3 The Monitors and the Instruments

This is a simple element which is essentially a drift of length  $L$  with a measure of the positions  $x$  and  $y$  at the center. The syntax is quite obvious:

```

MON1 = MONITOR("MON1",L)
MON2 = HMONITOR("HMON",L)
MON3 = VMONITOR("VMON",L)
CAM = INSTRUMENT("CAMERA",L)

```

PTC reproduces MAD’s full monitors, horizontal monitors, vertical monitors and instruments. There are all the same except that their KIND are different: KIND11, KIND12, KIND13 and KIND14 respectively.

### Q.5.4 The Quadrupole and Tilting

This element is created with the command:

```
QF = QUADRUPOLE("QF",L,K1)! L and K1 are keywords of the optional construct.
```

This element can also be tilted into a skew quadrupole. The command is

```
QF = QUADRUPOLE("QF",L,K1,TILT);
```

This particular command creates a “natural tilt” turning the normal quadrupole into a skew quadrupole.

Since we use overloading the reader may have realized that TILT is not a string as in regular tracking parsers but an object of type TILTING briefly mentioned in Sect. I.1. Thus it is possible to define operators which acts on TILT. Consider the operator “.IS.” (which stands for “to be” as in third person singular!):

```
QF = QUADRUPOLE("QF",L,K1,TILT.IS.0.1d0);
```

This literally means that the tilt is of 0.1d0 radian. In all cases the value of TILTD is affected. For straight elements this is equivalent to a tilt of the multipole components.

**This is a design tilt. It is not an error tilt. Beware.**

### Q.5.5 The Solenoid

This element is supported mainly in the expanded Hamiltonian framework. If EXACT\_MODEL is true, only the drift part will be computed exactly using the usually DRIFT routines in S\_DEF\_KIND. The syntax of the MAD-like command is simply:

```
SOL1 = SOLENOID("SOL1",L,KS,TILT);
```

In PTC, one can produce a solenoid with multipole components. These things must be added on the resulting fibre directly, using for example,

```
SOL1 = SOLENOID("SOL1",L,KS,TILT)
```

Multipoles can be added using the ADD command of Sect. L.6. The full solenoid with multipoles supports methods 2,4, and 6 of the Yoshida integrators. User defined methods are not implemented for the solenoid.

### Q.5.6 Other Straight Elements

We have of course the sextupole and the octupole available through a MAD-like command:

```
SF = SEXTUPOLE("SF",L,K2)! L and K2 are keywords of the optional construct.
```

or

```
SF = SEXTUPOLE("SF",L,K2,TILT)
```

For octupoles, the syntax is identical

```
OF = OCTUPOLE("OF",L,K3)! L and K3 are keywords of the optional construct.
```

or

```
OF = OCTUPOLE("OF",L,K3,TILT)
```

In both cases, the TILT.IS.ANGLE syntax is supported.

### Q.5.7 More Straight Elements: HKICKER, VKICKER, and KICKER

The syntax is simply

```
HKICK = HKICKER("HKICK",L,KICK)! L and KICK are keywords of the optional construct.
```

```
VKICK = VKICKER("VKICK",L,KICK)! L and KICK are keywords of the optional construct.
```

The tilt option is fully supported. **The MAD convention applies for KICK: a positive KICK means a positive kick for  $p_x$  and thus a negative EL%BN(1).** Finally, the general kicker is obtained with the call

```
KICK = KICKER("HKICK",L,HKICK,VKICK)! L, HKICK, and VKICK are keywords
! of the optional construct.
```

### Q.5.8 The Rectangular Bend

The rectangular bend is created with the command

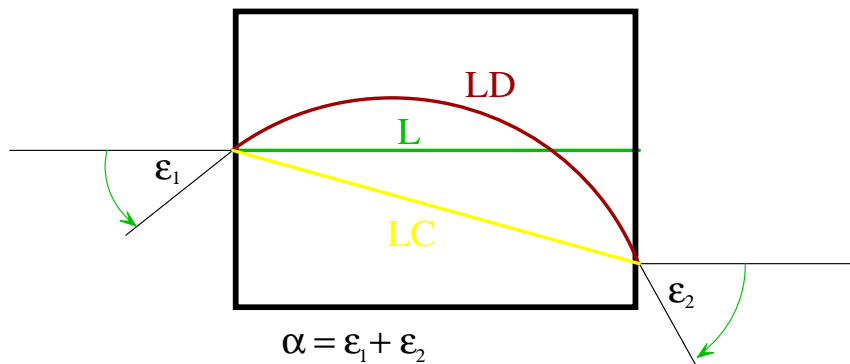


Figure 22: The True RBEND of PTC

```
B = RBEND("B",L,ANGLE)! L and ANGLE are keywords of the optional construct.
      or
B = RBEND("B",L,ANGLE,TILT)
```

It is also possible to call RBEND with the syntax

```
B = RBEND("B",L,ANGLE,E1)! L, ANGLE, and E1 are keywords of the optional construct.
      or
B = RBEND("B",L,ANGLE,E1,TILT)
```

```
B = RBEND("B",L,ANGLE,E2)! L, ANGLE, and E1 are keywords of the optional construct.
      or B = RBEND("B",L,ANGLE,E2,TILT)
```

In this case the entrance angle E1 of the bend is following the MAD convention that E1 is zero for ideal parallel face bend, that is to say:

$$\varepsilon_1 = \alpha + E1 \text{ and } \varepsilon_2 = \alpha + E2 \quad (62)$$

where  $\alpha$  is the total angle and  $\varepsilon$  is defined in Figure 22.

The total bending angle is ANGLE and the exit angle is ANGLE- $\varepsilon_1$ . In PTC, unlike MAD, RBEND really means a rectangular Cartesian bend in the EXACT\_MODEL option. Therefore  $\varepsilon_1$ , the exit angle, must be ANGLE- $\varepsilon_1$ . This bend is very interesting because it is the only element of PTC for which EL%L, EL%P%LD, and EL%P%LC are all different!

As seen in Figure 22, this is a perfect example of the generality needed to cover an arbitrary magnet. Here the variable EL%L is tied, as we said, to the inner field details of the magnet. Since the magnet is a rectangular object, it must be integrated in Cartesian coordinates. This has nothing to do with the end purpose of the magnet. This length L is really a “private” variable of the magnet similar to a bizarre B-field or other strange properties that arbitrary complex magnets might have. On the other hand, EL%P%LD and EL%P%LC are once more the variables indicating the coordinate needed to patch to the outside world.

The RBEND, obviously, supports the option EXACT\_MODEL.

In that context, we remind the reader of the option LIKEMAD=.TRUE. displayed in Figure 19. This is the case of a rectangular bend with arbitrary entrance and exit angles. This is not possible in a true rectangular geometry. Therefore MAD-like wedges are applied. The MAD-like input for this object is

```
B = RBEND("B",L,ANGLE,E1,E2)! L, ANGLE, E1, and E2 are keywords of the optional construct.
      or
B = RBEND("B",L,ANGLE,E1,E2,TILT)
```

PTC sees in this case an arbitrary E1 and E2; therefore it uses the wedges. In fact, even if one inputs an E2 matching the condition of the “true bend”, PTC will still use the wedges if the two keywords are present in the call statement. This is the clue telling PTC to use MAD wedges *on top of the true rectangular bend*.

### Q.5.9 The Sector Bend

```
B = SBEND("B",L,ANGLE)! L and ANGLE are keywords of the optional construct.
      or
B = SBEND("B",L,ANGLE,TILT)
```

The sector bend calls the general bend routines. If EXACT\_MODEL is true, then the exact sector bend (KIND10) is used. As in the case of RBEND, it is possible to have up to two extra angles, in which case wedges are added. The syntax is as before:

```
B = SBEND("B",L,ANGLE,E1,E2)! L, ANGLE, E1, and E2 are keywords of the optional construct.
      or
B = SBEND("B",L,ANGLE,E1,E2,TILT)
```

In this case, the angles E1 and E2 are just  $\varepsilon_1$  and  $\varepsilon_2$ .

### Q.5.10 The General Bend

This bend is only implemented for the approximate expanded Hamiltonian using the famous “quadrupole” trick to create the edge angles. E1 and E2 are defined here as in the SBEND.

```
B = GBEND("B",L,ANGLE,E1,E2)! L, ANGLE, E1, and E2 are keywords of the optional construct.
```

or

```
B = GBEND("B",L,ANGLE,E1,E2,TILT)
```

GBEND does not support the option EXACT\_MODEL in PTC because one must specify in PTC the nature of the body. Unlike in MAD, where bodies are always sector bends, in PTC one must make a choice between RBEND or SBEND. No default is provided. Obviously, in a fancy code such as MAD-X, a flexible default will probably be provided for backward compatibility (SBEND most likely).

### Q.5.11 The RF Cavity

The RF cavity input is

```
CAV = RFCAVITY("CAV",L,VOLT,LAG,HARMON,REV_FREQ)! L, VOLT, LAG, HARMON, and REV_FREQ  
! are keywords of the optional construct.
```

These are the MAD-input definitions. Harmon is the harmonic number, LAG is the phase, REV\_FREQ is the ideal rotational frequency for one turn around the machine. VOLT is the voltage. **The voltage is in megavolts even though PTC uses GeV for energy-like variables.** The variable LAG is the opposite of the variable PHAS inside PTC. Finally, FREQ of PTC is simply HARMON\*REV\_FREQ.

### Q.5.12 The Single Lens or SixTrack’s SMI

This is the single thin lens element of SixTrack which is supported in PTC. The call is simply, for example,

```
SF = SMI("SF",K,3)
```

or

```
SF = SINGLE_LENS("SF",K,3)
```

A skew element takes a negative entry for the multipole order:

```
SF_SKEW = SMI("SF_S",K,-3)
```

These elements can be tilted in the usual manner. They are strictly multipoles of a given kind. Therefore the addition of another multipole component to them should send an exception to the screen. For example, the following syntax, explained below in Sect. Q.6.1, is strictly forbidden.

```
SF_SKEW = SMI("SF_S",K,-3).S.2.d0! FORBIDDEN ADDITION OF A NORMAL COMPONENT TO A SKEW "SMI"
```

### Q.5.13 The Thin Multipole Block

Of course one can take a thick element and add a multipole block easily using the syntax described in Sect. L.4.4. In addition a thick element with L=0.D0 becomes a true thin element. However, again for the purpose of reading old “legacy” lattices, we support a more direct input namely through the routine

```
TYPE(MUL_BLOCK) BL
```

.

.

```
BL = 3
```

```
BL%BN(3) = 3.684068976279142D-001
```

```
B3 = multipole_block("B3",BL)
```

The tilt option is also possible, as well as the addition of other multipoles, since this is a full block of  $a_n$  and  $b_n$ 's.

## Q.6 Operators Acting On MAD-Like Input

In this section we describe a few useful operators.

### Q.6.1 Adding Multipole Components

Consider the case of a combined function sector bend. Obviously the command

```
B = SBEND("B",L,ANGLE)
```

creates a regular sector bend. Now, if as in the ALS, this magnet has a design quadrupole strength KF, then how do we put it in easily? Of course, one could put on a smart user hat, and use the routine ADD of Sect. L.6. One could even dump our routines and write something more adequate to one's taste. Please be our guest if such a thing is needed. But, if one puts on the dumb user hat, here is the operator provided.

```
B = SBEND("B",L,ANGLE).Q.KF
```

This routine is additive. Thus the following insane call would remove the quadrupole component from a quadrupole:

```
Q = QUADRUPOLE("B",L,KF).Q.(-KF)
```

PTC provides this particular operator up to order 20-pole (icosapole ?) in general and up to order dodecapole with special nomenclature. The special nomenclature is based on the mixture of Latin and Greek names used in accelerator physics. The letter S as in .SD. stands for the skew version. Here are the special names:

1. Bend: .D. and .SD.
2. Quadrupole: .Q. and .SQ.
3. Sextupole: .S. and .SS.
4. Octupole: .O. and .SO.
5. Decapole: .DE. and .SDE.
6. Dodecapole: .DO. and .SDO.

In general the Roman numeral is used for the name of the operator corresponding to an n-pole. For example, the octupole has 8 poles and thus the generic operator is .VIII. As before the letter S modifies this operator into the skew version. This number is twice the order of the vector potential  $A_z$  appearing in the Hamiltonian.

**Watch out for MADKIND3N=KIND8 or MADKIND3S=KIND9. These are the so-called SMI of SixTrack. They are single multipoles.**

For example the following syntax is acceptable:

```
L=0.d0; MADKIND3N=KIND3;  
OF = OCTUPOLE("OF",L,KO).Q.KF;
```

However the following one is a no-no: L=0.d0; MADKIND3N=KIND8;

```
OF = OCTUPOLE("OF",L,KO).Q.KF;
```

This will attempt to put a quadrupole component in an object which is strictly a thin normal octupole.

### Q.7 The Final Step: the Creation of a Layout

The final goal of this exercise is to create a layout (PSR) which can be used by PTC for tracking, map production, map analysis, etc. This was shown in the example of Sect. Q.1

```
CELL = D1 + QD + D2 + B + D2 + QF + D1  
MYRING = 10 * CELL  
PSR = .RING.MYRING
```

where CELL, RING, and MYRING are also of type LAYOUT. Therefore, after having created all these objects to one's satisfaction, the assignment `PSR=.RING.MYRING` is of crucial importance. It creates PSR of type Layout which is the quintessential object needed by PTC. PSR is different from the other layouts in the sense that it is made of clones: each fibre is an individual fibre and not something pointing elsewhere. The layouts used during the construction (CELL, RING, and MYRING) are trackable but are made of fibres pointing to the single fibre created by the Mad-like commands. They are all "single-pass" by default, i.e., `MYRING%CLOSED=.FALSE.`, so beware if you use them for fitting the bare lattice.

Finally, we remind the reader that if PSR had been a single pass object, it would have been created by `PSR=.LINE.MYRING`.

## R So\_FITTING.f90: Non-core Routines

This module contains examples of fitting routines using polymorphism as well as some routines to re-adjust the number of integration steps and the method of integration. Finally we find here the fixed point finders. Apart from the fixed point finders, which are probably in a final usable state, the other routines will evolve. We should also say that all the routines belong to the analysis part of a tracking code, and that is why they are not in the core of PTC.

### R.1 Changing the Integration Method of a Magnet

Of all the various kinds of magnets, KIND2, KIND10 (TEAPOT) and KIND16 (STREX) rely the most on the actual integration method used, especially if the DRIFTKICK variable of KIND10 and KIND16 is set to true. Indeed these magnets should always be handled and modified using the “Talman” philosophy of modelling. As we have said before, it is customary in accelerator physics to regard the method of integration as part of the model. In the case of Drift-Kick-Drift integrators, it is imperative to do so. Following this line of thought, one tries to minimize the number of thin lenses (steps of integration) while retaining the general properties of the lattice. It is more an art than a science, but so is most of accelerator modelling.

The subroutine THINLENS helps in this regard. The syntax is simply

```
THI=-1
CALL THINLENS(PSR,THI)
WRITE(6,*) THI
```

If the quantity THI is less than zero, then THINLENS will interactive, using a screen dialogue. If it is a number greater than zero, it will select that particular value of THI and store it in the variable PSR%THIN after it has properly modified the lattice. So what is this input THI? THI corresponds approximately to the quadrupole kick of each integration step. Inside THINLENS, the following hand waving formula is used

```
GG=XL*(RHOI**2+DABS(QUAD)) ! RHOI=EL%P%BO and QUAD = sqrt(EL%BN(2)**2+EL%AN(2)**2)
GG=GG/THI
NTE=IDINT(GG)
```

The approximate integrated focusing strength is first evaluated and then divided by THI. Thus GG becomes the number of thin lens kicks for this element. In PTC there are three methods of integration for KIND2/KIND10/KIND16: Yoshida 2,4, and 6. Each of these methods have a certain number of multipole kicks per steps. PTC switches automatically from one method to the next on the basis of the private array LIMIT(2)=(3,14) of THINLENS. If NTE is less than 3, then the second order method is used. If NTE is between 3 and 13, then the fourth order Yoshida-Ruth is used. Finally for NTE greater or equal to 14, then sixth order Yoshida is used.

In summary, if no magnet kind is specified, PTC will re-slice all magnets of KIND2,KIND10 and KIND16 according to the above rule.

However this routine can be called on a specific kind: KIND2, KIND6, KIND7, KIND10 or KIND16: CALL THINLENS(PSR,THI,THISKIND). Of course the criterion for the number of thin lenses should be different in all cases. For drift-kick integrators, one must refit known properties and perhaps look at lattice functions, chromaticities and short term dynamic aperture. This is really the Teapot code problem. For KIND6, the slow thick element, it is really just a matter of multipole errors.

Obviously, we hope to move these features into MAD-X where the re-slicing can be done most effectively thanks to the various powerful matching modules of MAD. Undoubtedly, while MAD-X will not be more powerful than PTC model-wise, it will have defaults and reslicing routines of a far greater sophistication than those in the present PTC. In addition, a smart user of MAD-X will be able to re-slice and re-fit a vast array of variables to create his “thin lens” lattice using whatever powerful command MAD-X inherits from MAD-8 or are simply added to this new version of MAD. This should be a killer code which will put an end to the ugly exercise of transferring thick lattices from MAD (or other design codes) into particle/TPSA pushers such as SixTrack, DESPOT or even TRACYII.

### R.2 Fixed Point Routines: Polymorphi Delendi Sunt

Fixed points routines are extremely important in a tracking code such as PTC or any other code of the integrator variety. This is because we do not have a special “reference” orbit like in matrix codes. It is in the nature of properly written integrators to look for the correct closed orbit.



Closed orbit routines lie somewhere between analysis and tracking. In fact they are the first step of a normal form: having found the closed orbit, all the normal form procedures are done around that orbit. But of course it would be ridiculous to consider them part of the FPP package since FPP is more general than PTC and could be used by anyone.

The real(dp) `DEPS_TRACKING` is a FORTRAN constant set to  $1^{-10}$  and it is used in the fixed point routines. In COSY-INFINITY style computations of both FPP and PTC (kind6 and kind7), in generating function tracking of FPP (type `genfield`), in single Lie exponent searcher (type `ONELIEEXPO` of FPP) and in fixed point routines, we use a two step process. First the code looks at the actual value of a merit function. When this merit function drops below a certain value (such as `DEPS_TRACKING`), then the code continues iterating until the merit function either settles or goes up. At this point, machine precision has been reached. PTC has four built-in fixed point routines for convenience. All of them are called with the interface `FIND_ORBIT`.

1. First it has a basic fixed point searcher: one which uses TPSA and one which does not.
  - (a) `FIND_ORBIT_LAYOUT`: `CALL FIND_ORBIT(RING,X,LOC,STATE)`: this routine finds the fixed point X for the internal state "STATE." RING is a layout and LOC is the position in the layout where the fixed point is wanted.
  - (b) `FIND_ORBIT_LAYOUT_NODA`: `CALL FIND_ORBIT(RING,X,LOC,STATE,EPS)`: this routine has the same functionality as that of item (1a). However it uses numerical differentiation rather than TPSA in the Newton search. This is done with the parameter EPS.
2. `FIND_ORBIT_M_LAYOUT`: `CALL FIND_ORBIT(RING,Y,LOC,STATE)`: this is the same as the above TPSA routine except that the map is stored in the polymorph Y(6).
3. `FIND_ENV_LAYOUT`: `CALL FIND_ORBIT(RING,YS,X,LOC)`: this finds the linear beam envelope map in the presence of stochastic radiation.

Three of these routines (items 1a, 2, and 3) use FPP and TPSA and therefore they should not be called in the middle of a TPSA calculation. To paraphrase Cato the Elder: *Polymorphi Delendi Sunt*. All external polymorphs must be destroyed by the kill routine before calling these three routines.

In all these routines the variable `STATE` is optional. If ignored, PTC will try a search in the `DEFAULT` state whatever it may be. The applicable `KEYWORDS` are `STATE` and `EPS` (see Sect. R.2.2 for example).

### R.2.1 `FIND_ORBIT_LAYOUT(RING,X,LOC,STATE)`

This is the simplest routine. It was used in Sect. A.2.1 as `FIND_ORBIT(PSR,X,1,DEFAULT)` in the first example. This routine looks at `DEFAULT%NOCAVITY`. It performs a linear Newton search.

- If it is true then it finds the fixed point for the value of delta contained in X(5).
- If it is false, then it finds a six-dimensional fixed point. In that case the routine checks for the presence of a cavity; if none are found it throws an exception.

### R.2.2 `FIND_ORBIT_LAYOUT_NODA(RING,X,LOC,STATE,EPS)`

Same functionality as the routine in Sect. R.2.1. We recommend a typical value of about 1.d-8 for EPS. A typical call can also use the keywords:

```
CALL FIND_ORBIT(PSR,X,1,STATE=DEFAULT+FRINGE,EPS=1.d-8)
```

### R.2.3 `FIND_ORBIT_M_LAYOUT(RING,Y,LOC,STATE)`

This routine is basically identical to the previous one. The only difference is that it returns the fixed point inside the polymorph Y. In that case, the TPSA package is initialized with order 1 and no parameters. Thus Y contains the linear map around the fixed point. Of course the map Y should enter the routine in a newly allocated or killed state.

## R.2.4 FIND\_ENV\_LAYOUT(RING,YS,X,LOC,STATE)

The purpose of this routine is to find the beam envelope in the case of radiation. It replaces the old synchrotron integral technique. It is explained in Sect. C.4. This routine tracks the state SSS

```
sss=(STATE-nocavity0-only_4d0-delta0)+radiation0
```

if the input state is incompatible with a beam envelope calculation. The STATE variable can be omitted, in that case the DEFAULT state is used for STATE. This state, sss, should contain everything which is compatible with a beam envelope calculation.

One notices that the beam envelope in Equation (14) is a collection of quadratic moments. Quadratic moments are dual to quadratic polynomials; thus FPP uses an equivalent quadratic polynomial to normalize the beam envelope. However PTC first computes  $M$  and  $B$  using a linear TPSA calculation. Then, inside FIND\_ENV\_LAYOUT, the package is changed to second degree. This second order beam envelope is normalized directly using an object of type BEAMENVELOPE (see Sect. C.4.2).

Therefore upon exit, the TPSA package reverts to first order. The map part of YS is set to the identity around the closed orbit. The fields YS(i)%SIGMA0(j) are set to the equilibrium beam sizes. This beam envelope is ready to be tracked around the ring for a radiative lattice function calculation using a linear TPSA!

As we said before, parameter dependence is a bit tricky. This routine **does not** support parameter dependence. We discuss parameter dependence in the next section.

## R.2.5 Parameter Dependence: FIND\_ENVELOPE(RING,YS,A,FIX,LOC,STATE)

The theory of beam envelope is essentially a linear theory but valid in a nonlinear environment. It is not an exact nonlinear theory. The map YS(6)%V computed in a normal beam envelope run is the usual radiative deterministic map. This is obviously an approximation. We are saying that the average of the first moment  $\langle x_i \rangle$  is the same as the deterministic trajectory. Secondly, having “found” the average trajectory, we compute the map for second moments around this trajectory. The end result is then normalized using the type BEAMENVELOPE. The situation is similar to that of spin calculations in accelerators. The standard field affects the spin but the spin does not affect the usual trajectory.

Because we do not have a full nonlinear<sup>34</sup> moment theory, we cannot locate the parameter (knobs) dependent second order moments from a final beam envelope map. However we can compute the deterministic map as a function of the parameters. Through standard normal form techniques, we can extract the parameter dependent fixed closed orbit. This is done in FIND\_ENVELOPE. Let us look at the actual code:

```
SUBROUTINE FIND_ENVELOPE_L(RING,YS,A1,FIX,LOC,STATE)
  IMPLICIT NONE
  TYPE(LAYOUT),INTENT(INOUT)::RING
  TYPE(REAL_8),INTENT(INOUT)::A1(6)
  TYPE(ENV_8),INTENT(INOUT)::YS(6)
  REAL(DP),INTENT(INOUT)::FIX(6)
  INTEGER,INTENT(IN)::LOC
  TYPE(INTERNAL_STATE) STATE
  TYPE(REAL_8) Y(6)
  TYPE(DAMAP) ID
  TYPE(NORMALFORM) NORMAL

  CALL ALLOC(Y);CALL ALLOC(ID);CALL ALLOC(NORMAL);

  Y=6 ; Y=FIX ;
  CALL TRACK(RING,Y,LOC,STATE) !
  NORMAL= Y ;
```

---

<sup>34</sup>Actually we have such a theory, but it is a slow messy theory involving a very large operator. In fact it is an operator dual, in the deterministic case, to the Lie operator of perturbation theory. The Lie operator is almost lower triangular (nice); the moment operator is thus worse than upper triangular (awful). In other words low moments are affected by high order moments unlike linear tunes which do not depend on tune shifts with amplitude.

```

Y=NORMAL%A1+FIX
A1=Y
YS=Y
CALL TRACK(RING,YS,LOC,STATE)

```

```

Y=YS
ID=Y      ID=(NORMAL%A1**(-1))*ID
Y=ID+FIX
YS=Y

```

```
CALL KILL(NORMAL);CALL KILL(ID);CALL KILL(Y);
```

```
END SUBROUTINE FIND_ENVELOPE
```

The blue part shows the computation of the map around the closed orbit FIX. Then, in the green section, the initial map  $YS(6)\%V$  is initialized as

$$M = A_1 + FIX \quad (63)$$

where  $A_1$  is the transformation which subtracts the parameter dependent closed orbit from a ray. It is then tracked around for one-turn. In the red part, the map  $YS(6)\%V$  is brought back around the parameter dependent fixed point. This is needed because, if one simply tracks  $A_1$ , then the initial coordinates are around the parameter dependent fixed point but at the exit they are expressed around the ordinary fixed point.

A call to FIND\_ENVELOPE can be followed by a call to the regular tracking routine TRACK:

```

TYPE(BEAMENVELOPE) ENV      .
      .
      .
ENV=YS

YS=Y
YS=ENV%SIJ0

CALL TRACK(ALS,YS,1,10,+STATE)
ENV%SIJ0=YS

```

The equilibrium envelope is computed by  $ENV=YS$  and it is then set to the array  $YS(6)\%E(6)$  with the assignment  $YS=ENV\%SIJ0$ . It is then tracked. Before exiting the tracking routine, the tracked envelope is evaluated and put into  $YS(6)\%SIGMA(6)$  using  $ENV\%SIJ0=YS$ . The actual evaluation of  $YS(6)\%SIGMA(6)$  is done inside TRACK.

## References

- [1] C. Iselin, The MAD User's guide, the most up-to-date version can be found at CERN and is available on the internet.
- [2] Code by K. Oide, Japanese documentation can be found on KEK internet site.
- [3] A. J. Dragt, Part. Accel. **12**, 205 (1982).
- [4] L. Schachinger and R. Talman, Part. Accel. **22**, 35 (1987), the authors checked TEAPOT against the PSR lattice paper of Dragt.
- [5] E. Forest, *Beam Dynamics: A New Attitude and Framework* (Harwood Academic Publishers, Amsterdam, The Netherlands, 1997).
- [6] L. Michelotti, *Intermediate Classical Dynamics with Applications to Beam Physics, Wiley Series in Beam Physics and Accelerator Technology* (Wiley, New York, 1995).
- [7] V. K. Decyk and C. D. Norton and B. K. Szymanski, Introduction to Object-Oriented Concepts using Fortran90, to be published.
- [8] E. Forest and K. Hirata, Technical Report No. 92-12, KEK (unpublished).
- [9] A. W. Chao, J. Appl. Phy. **50**, 595 (1979).

## A Postface by Etienne Forest

No form of Nature is inferior to Art; for the arts merely imitate natural forms. . . . all arts do the inferior things for the sake of the superior . . .

Marcus Aurelius, Meditations.



Georges Seurat: Art, Technique and Nature

*In this postface, I describe mainly the attitude taken by myself in developing PTC. PTC is perhaps a bad code for several reasons. The most obvious is simply the lack of mathematical, physical and programming skills of the author, mainly myself. It could be also related to using the wrong language or combination of languages which, in turn, is also a lack of skills on my part. But most fundamentally the particular frame of mind I adopted in developing PTC is what is worth discussing and perhaps reject; if one adopts it and combines it with a maximum level of skills, then there are no two ways of producing PTC. In a sense it is the most revolutionary part of this work and perhaps the most controversial as well in certain circles.*

*So my attitude has been to consider the project more as an artistic expression in the Western sense of the word rather than just a mere implementation of known algorithms. While the development of physics since Pythagoras is certainly related to the belief that Nature obeys laws which are mathematically beautiful, we lose track of these things in our everyday life. This underlying belief does not prevent us from generating ugly algorithms and dirty experiments when they are required. In developing PTC, I come back to the source so to speak, and religiously seek the creation of an object which artistically represent what I can see in Nature. Greek logic and esthetics perhaps were predestined to produce the David of Michelangelo. While I am certainly not a Michelangelo, I can see similar forces which, if we let ourselves guided by them, will result in a product literally fashioned by our own inner sense of beauty. It is the principle which guided me; others I hope can follow it with greater skills and grace.*

The alleged purpose of this paper was to describe PTC— yet another computer code simulating particle tracking. What are the bells and whistles? What are its new and interesting algorithms? What are the new models which are perhaps embedded in it? Does it have some neat graphics? What would be a short but accurate answer to all these questions? I can give it immediately: **Nothing**.

PTC has more in common with the above painting of Seurat than it has with TRANSPORT. This is not an outrageous answer, but the plain truth. It is best to understand PTC as a work of art using techniques in the realm of mathematics, computer science and physics to approximate a certain rendition of reality on

a canvas which happens to be made of silicon chips. On the one hand, it cannot surpass the reality which it tries to imitate. On the other hand, as any artwork, it goes beyond the object which it imperfectly imitates because the artwork is by itself a part of Nature. PTC has no algorithm to compute the beta function, or the synchrotron integral, or the phase advance, or the tune shift with amplitude. Indeed if one scans the core of the code, none of these things will be found. They are not there. There are no synchrotron integral modules. There are no “HARMON” modules. There are no coupled formalisms. And there are no multi-summations over Fourier modes of one’s favorite Hamiltonian. Nothing: it is just as placidly banal as the above painting. And yet, if one adds to PTC a few lines of code, it suddenly computes all of the above more faithfully and more correctly than any of the typical implementation of these algorithms.

As I said PTC is about Art. In the painting of Seurat, the scene is banal: a few nineteenth century Parisians enjoying an afternoon at the local park. PTC is banal: a computer realization of the physical system; the creation of an object on the computer which behaves, through careful syntactic composition, like a certain idealization of the true physical object. The issue which has interested me since 1987 and particularly when I became conscious of Object Oriented Programming is this: On the computer silicon canvas can we create objects which will act more or less like their physical counterpart? Or are we condemned to implement on the computer only algorithms relevant to the objects of the physical world and then link them cleverly by some manager as MAD has been doing since its creation?

As pointed out by Marcus Aurelius, Art fails to reproduce Nature fully. And yet the artwork may, by a clever and paradoxical use of its own limitations, go in some direction further than one would have anticipated. We can use the limitation of the medium to our own advantage by the use of techniques applicable only to the restricted space in which we decide to work: the canvas of the painter or the computer canvas of the programmer/physicist. In the case of Seurat, it is pointillism, the usage of tiny dots of primary colors to generate secondary colors. The artist added to the actual depiction an element which is not really present in the natural scene; but in doing so generates in the human mind emotions and pleasures which are certainly part of Nature itself. Art does indeed inferior things for the sake of the superior as well said by Marcus Aurelius.

I need to talk about my own canvas: the computer. The human brain is capable of more emotions and representations than a Seurat can ever put on his canvas. In accelerator physics, I can see with pure reasoning the passage from Newton-Maxwell to an “s”-theory, i.e., a local representation good only sometimes, like the two dimensional figures of Seurat. I can also imagine how an electron would suddenly see other electrons and other collective effects that invalidates our “s”-picture. In fact I can even imagine suddenly switching back to the Newtonian time domain and interacting with billions of particles. I can imagine things of fantastic complexity. Unfortunately, the computer like Seurat’s canvas has some limitations and therefore it is an illusion, as long as complex calculations are involved, to ignore these limitations.

So we must address the issue of the computer. Firstly, it is a relatively finite system, and for this reason, it is reasonable as Seurat did to limit ourselves. PTC is centered around single particle propagators and so are all these other tracking codes: MAD8/9, SAD, etc. Anyway, I do not want the reader to be too bold here and lose track of my line of reasoning. So let us concentrate on the travel and tribulation of a single particle through some accelerator.

Secondly, we interact with the computer through a language. In that sense, PTC is more like a piece of literature than a painting. We cannot ignore the central issue of algorithmic languages versus object-oriented languages. Although FORTRAN90/95, the language of PTC, can barely be considered object oriented, the issue of object-orientedness cannot be ignored, it is *the central artistic theme* of PTC.

My goal was to create on the silicon canvas a representation of the accelerator which, under the assumption of single particle dynamics, will seem to behave and breathe like a real accelerator. In PTC, for example, if a magnet is misaligned, then it is automatically correctly misaligned in all beam lines that share this magnet. *Remarkably, in PTC, this is not achieved by programming logical links between the various objects, rather it is achieved by implementing the correct mathematical structures— fibre bundles— and using the tools provided for us by the computer scientists to ensure a faithful representation on the silicon canvas. The objects are ours, the computer techniques are theirs, but they are part of the same artistic composition. One cannot ignore the other.* In addition, as we shall see, if some user’s algorithm uses PTC extended definition of the ray to compute the equivalent of the “synchrotron integrals,” then it will be correctly computed under any possible mispowering and misaligning of the elements. PTC is a faithful representation of a part of nature, just as Seurat’s painting is a faithful representation of some aspect of a scene. In addition, just as pointillism adds to the natural setting a seemingly unnatural element, PTC adds properties to the ray being tracked which do not exist in nature. In the case of PTC, thanks to a polymorphic type first dreamt up by Bengtsson, the electron carries with itself a potential Taylor Series whose variable space is nearly infinite.

Thanks to this feature, PTC can, from a composition point of view, be restricted entirely to the creation of types (objects) whose role is to elevate the flow through a magnet to that of a mathematical object. Once this is done, a user can write an algorithm that will compute all the nasty objects of perturbative theory even though PTC seems to have none of them. This is why PTC is object oriented in a satisfying way: the objects of PTC are representations of some natural objects. As Seurat did with colors, I extended the meaning of phase space so that all of perturbative quantities would come out for free: PTC's electron and proton potentially carry Power Series on their back just as the natural ones carry spin.

The usage of C++ and object-orientedness in MAD9 and other CLASSIC-style programs did not attempt to create objects germane to single particle dynamics. Rather, these codes emphasized the same good old algorithms of the good old procedural programming. It is not surprising because the standard global "s"-dependent theory of accelerator physics lends itself, in the human brain, to algorithmic manipulations. One can imagine "turning" the crank on the Hamiltonian expressed around the closed orbit and pushing calculation à la Guignard to the tenth order; but how many amongst you would even trust someone claiming that his first order calculations are valid under *any tracking conditions* of his code? What happens to synchrotron integrals in MAD when things are misaligned, mispowered and solenoids are all over the place? Who knows really?

My views have been, at least since the C++ business got underway, that the flow through the magnet must be elevated to the status of a mathematical object. And then, it must find its counterpart on the silicon canvas, whether painted in C++ or any other language. Polymorphism, Bengtsson's pointillism, will take care of the rest. This is achieved by a local "s"-dependent theory which is shaped around individual magnets. The global system is then patched together. The mathematicians gave us the tools to manipulate this object: the fibre bundle. PTC simply creates a restricted fibre bundle on the computer, one which is relevant to particle accelerators. This structure is incompatible with standard Courant-Snyder theory and other similar constructs like Sand's integrals. There is nothing I can do about that except to provide an equivalent theory which is compatible with the silicon brain. And that I did in case you have not noticed.

In PTC, if one metaphorically replaces the magnet by a person, we can say that the person appears whole but of course incomplete as do the flat characters of Seurat's painting. In algorithmic programming and in the C++ implementations such as MAD9/CLASSIC, the person is dissected on the table: the bowels here, the organs there, the head over here, and so on and so forth. These are the "algorithms": an eating algorithm, a procreation algorithm, a speaking algorithm, etc. These algorithms, as in a FORTRAN77 code, loom over the design. The designers of CLASSIC, in particular the SLAC people, kept repeating "what are the important algorithms and what classes do we need for the various algorithms?" Bengtsson and I did not need to listen one second further to conclude that such a project had to fail. The resulting codes, despite C++ and object orientedness, are to PTC what "the cubic period of Picasso" is to Seurat: abstract art. What better proof than the fact that the accelerator group in the CERN SL division have launched the MAD-X project, a rewrite of MAD8, instead of further developing MAD9. What else can be said? I certainly had nothing to do with this decision. In fact I believe that if the CLASSIC people had really been object oriented, today I would be its greatest advocate. By objects, I mean objects derived from physics, from nature, and not from computer science. For example, besides the SLAC insistence on "algorithms," nothing stuck more in my mind from that fateful FNAL pre-CLASSIC meeting, than CERN's presentation on the influence of C++ and object-orientedness on MAD's parser. *Excuse me, important detail perhaps, but what does this have to do with accelerator physics objects? So dangerous a misconception (algorithmically driven design), followed by irrelevance (MAD parser) and cultish belief that in C++ things will be miraculously well designed, led me to believe that this empire had collapsed before it had risen.* I am sorry but an electron does not need our computation algorithms for the tenth order momentum compaction to find its way around a real accelerator and therefore, a true object oriented silicon rendition of the accelerator should not either. Focusing primarily on algorithms is incompatible with good object oriented design.

It is true that if one ignores the perturbative algorithms, then we are seemingly incapable of computing anything but the ray itself. But this is only true if we ignore TPSA and the Hamiltonian-free techniques and revert to the antiquated Courant-Snyder approach. I wrote a book where I explained these techniques, where I explained our pointillism: TPSA, Taylor polymorphism, Hamiltonian-free perturbation theory, the magnet-object realized by a proper implementation of the Euclidean group acting on the magnet propagator[5] (flow), etc.. It was not my goal to be the actual artist; I believe that the C++ proponents would have done much better than myself, had they spent sufficient time studying the theory before jumping into C++ constructs. PTC and FORTRAN90 are not necessarily the best choice. C++ might still be a better choice or perhaps some other language or even a mixture of languages: I do not know. I know however from pure reasoning that the natural system, the mathematical theory describing it, the various renditions of this theory, and

the computer— the medium used to create the art work, all impose on us extremely strict constraints. The creation of classes in C++ to control algorithms as in CLASSIC, where one focuses on single functions *in vitro*, should not be the guiding design principle of the accelerator physics classes. This is true whether or not we decide later to include a lot of external algorithms (through visitor classes) or through MAD8 style modules. The combination of the blind belief in the C++ language and a reverence for the archaic Courant-Snyder theory will lead to only overly complex programs. The silicon canvas is not the human brain. Equivalent theories in our brain are not equivalent on the computer. This should be quite easy to comprehend.

So PTC is my little work of art, for my own satisfaction. I created it due to an accident of circumstances; anybody can have it. Personally, after 10 years of CLASSIC and MAD9, I have no interest in getting involved in this debate again. The next time you hear that some beautiful C++ code gives wrong synchrotron integrals when the cavities or the bends are misaligned or mispowered, or tilted, as I have heard already, then you will know what I am talking about.

Etienne Forest

### **Acknowledgments**

Besides the two individuals whose names appear on this paper and Aimin Xiao who collaborated on the very first prototype, I would like to thank Johan Bengtsson (of parts unknown) for convincing me that, at least in C++, one could go ahead and make a reasonable job of polymorphism and fibre bundles. In addition, I acknowledge the influence of Martin Berz on this work. In practice, without his work, pretty much all of PTC falls apart. More recently, I want to thank David Robin at ALS (LBNL) and Ross Schlueter (LBNL) for providing real work by which these tools could be tested. Particularly I am not sure there would be a complex polymorph without the PEEMB project on which I worked a little as an invited guest at ALS. Also I was fortunate to try the first version of the fibre bundle PTC on an incomplete but very complex recirculator lattice provided to me by Carol Johnstone at FNAL; I am grateful to her. I am also grateful to David Sagan of Cornell for many suggestions and his integration of FPP/PTC within his code BMAD. I am grateful to Alexander Zholentz and again David Robin for letting me test these ideas on a simpler but more complete recirculator. I am not good enough to generate remotely useful code being left to myself; so the immediate help of these people was of tremendous practical importance. Last but not least, I do not forget Werner Herr's advice concerning the usefulness of an early introduction of linked lists in PTC: the sooner, the better.



# Overview of the “Fully Polymorphic Package” FPP

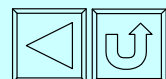
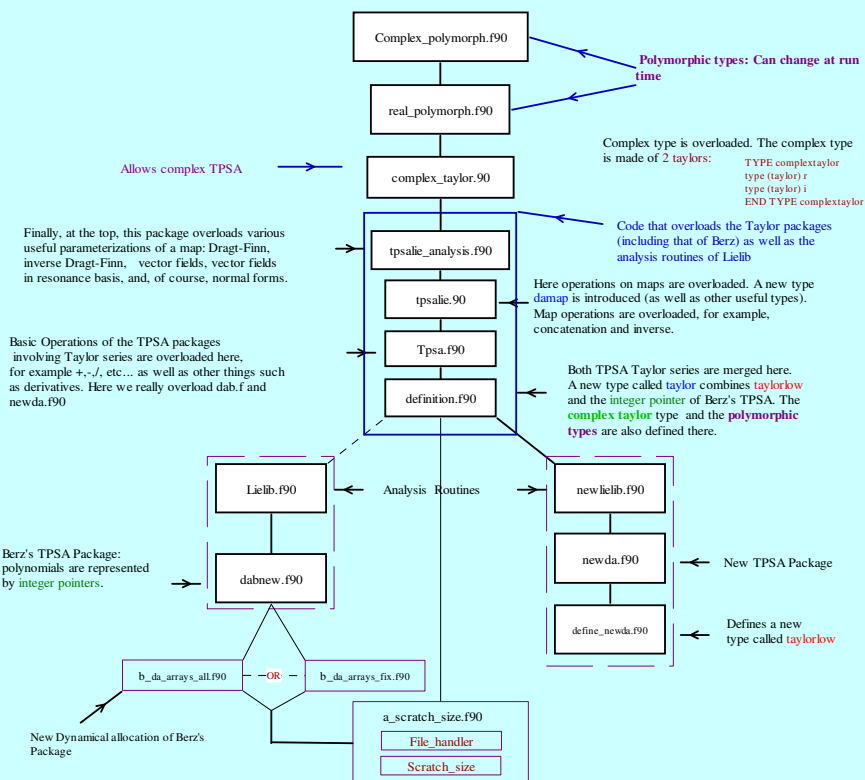
Etienne Forest, KEK  
With the help of  
Frank Schmit (CERN),  
Aimin Xiao (DESY)  
and  
David Robin  
(LBNL)

New Stuff: Polymorphism supports kind=0,1,2 and now 3. Kind=3 is a knob.

Best seen with Internet Explorer 4.0 or above. Sorry!  
Set your screen to 1024x768 pixels



## Structure of FPP

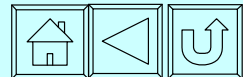


# Tpsalie\_analysis

The Tpsalie\_analysis module **was** the most upper level module prior to the development of the polymorphic packages. It overloads the most advanced functions of the LieLib library.

Now we strongly recommend that the module **polymorphic\_complex\_taylor** be included at the top whether or not you deal with complex entities.

Let us go to an example



## Example: Taylor Expansion of sin(x)

```

program sine
  use polymorphic_complex_taylor
  implicit none
  real*8 x
  integer NO,NV
  type (taylor) xt

  x=0.d0
  call double_sine(x)
  write(6,*)x

  no=5
  nv=1
  call init(NO,NV,.true.)

  call alloc(xt)
  call var(xt,0.d0,1)
  call tpsa_sine(xt)
  call daprint(xt,6)

  call kill(xt)

end program sine
  
```

**Degree of polynomials** → `no=5`

**Number of variables** → `nv=1`

**Initializes TPSA** → `call init(NO,NV,.true.)`

**xt = 0.d0 + x<sub>1</sub> ; Initializes the variable xt. The real part is 0.d0 and the TPSA part is the monomial x<sub>1</sub>.**

**Garbage Collection: Allocates a local TPSA and destroys it**

```

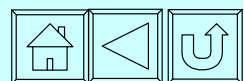
subroutine double_sine(x0)
  implicit none
  real*8 x0,x
      x=sin(x0)
      x0=x
  return
end subroutine double_sine

subroutine tpsa_sine(x0)
  use polymorphic_complex_taylor
  implicit none
  type (taylor) x0,x
  call alloc(x)
      x=sin(x0)
      x0=x
  call kill(x)
  return
end subroutine tpsa_sine
  
```

`call alloc(xt)` → **Garbage Collection: Allocates a local TPSA and destroys it**

`call kill(xt)` → **Garbage Collection: Allocates a local TPSA and destroys it**

`call kill(x)` → **Garbage Collection: Allocates a local TPSA and destroys it**



Output

# Results of Example

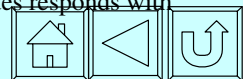
```

0.0000000000000000E+000 ←
Old Berzio ←
ETALL , NO = 5, NV = 1, INA = 20
*****
ORDER   COEFFICIENT   EXPONENTS
NO = 5   NV = 1
1  1.0000000000000000    1
3  -.16666666666666667   3
5  .8333333333333333E-02  5
-3 .00000000000000000    0

```

x=0.d0  
call double\_sine(x)  
write(6,\*)x  
call init(NO,NV,.true.)  
call daprint(xt,6)

There are three important steps in this program. First the real program is called and the sine function is evaluated on the argument 0.d0  
 Then the overloaded package is TPSA (actually tpsa.f90 only) is initiated. The value .true. Is passed to indicate our desire to use dabnew.f, the old TPSA of Berz. The overloading codes responds with **Old Berzio** to indicate that all is OK!



# Results of Example...

```

ETALL , NO = 5, NV = 1, INA = 20
ORDER   COEFFICIENT   EXPONENTS
NO = 5   NV = 1
1  1.0000000000000000    1
3  -.16666666666666667   3
5  .8333333333333333E-02  5
-3 .00000000000000000    0

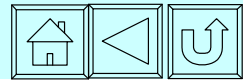
```

This first line indicates the name (**ETALL**), the degree (**5**), the number of variables (**1**), and the actual pointer integer used by Berz (**20**) for this particular polynomial.

As explained before, the variable xt is initializes as  $xt = 0.d0 + x_1$ ; the program then evaluates  $\text{sine}(xt) = x_1 - (1/3!) x_1^3 + (1/5!) x_1^5$

Last line is a termination marker.

Only one column of exponents since there is only one variable (nv=1) in this run.

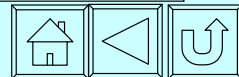
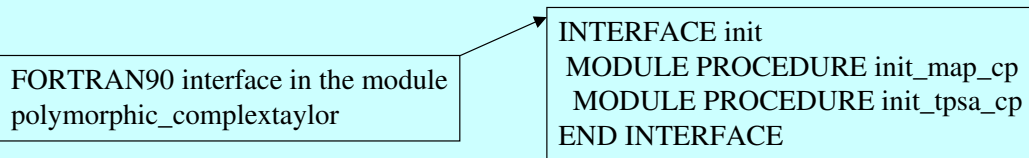


# Initialization of Overloaded TPSA: Init

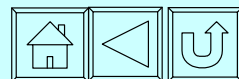
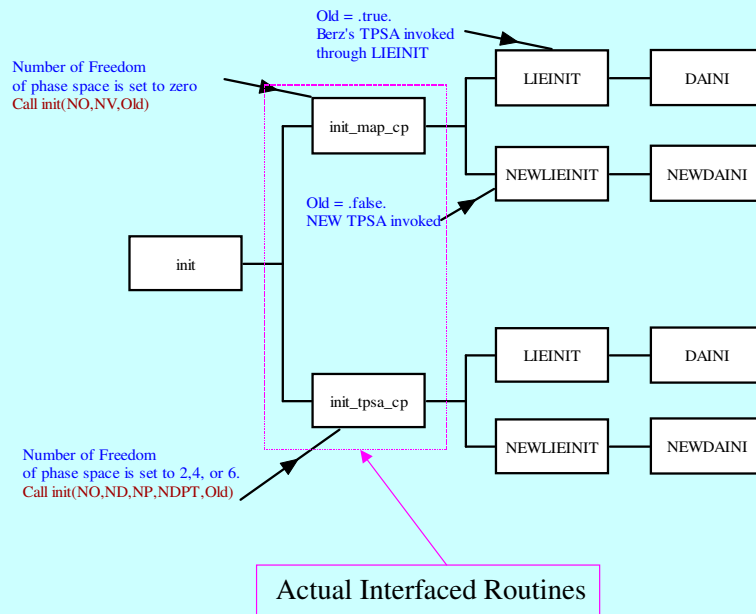
We did not want to rewrite the old LieLib package. This is a perhaps mistake in the long run but given its practical importance we decided to leave it as is. One drawback is that one must write a new LieLib each time a TPSA package is written ([see tree](#)).

Nevertheless it is not necessary to initialize the overloaded routines of LIELIB. In the  $\sin(x)$  function example, only the TPSA packages were activated and LIELIB was in effect unusable. LieLib was initially written to handle symplectic maps, which are of even dimension. Therefore initializing LieLib forces the parameter NV, the number of TPSA variables, to be greater or equal to 2.

Here we will show how one can use the function `init` to initialize the TPSA with or without LieLib.



## Structure of Init in `polymorphic_complextaylor`



# Potential Calls to Init:1

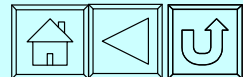
There are two possible calls to init as indicated on the previous page:

- 1) Call `init(NO,NV,Old)`
- 2) Call `init(NO,ND,NP,NDPT,Old)`

The first call enables the TPSA package only. This maybe of some interest when the calculation has nothing to do with beam dynamics. Being in accelerator physics, I do not use this option much. For example, in the sine function example, `NV` was set to 1.

The parameter `Old` refers to the package being used. At the moment, having only two packages, we have:

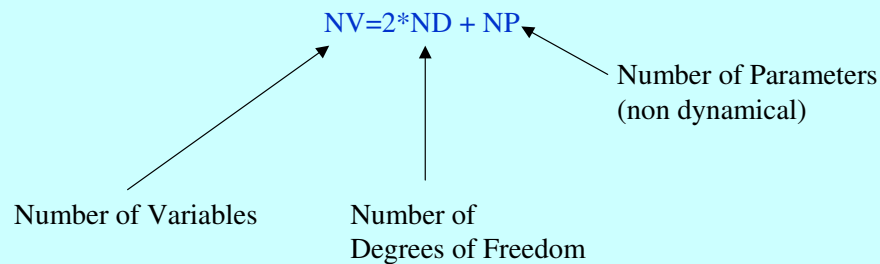
- `Old=.true.`       $\longrightarrow$       Berz's TPSA package (LBNL/CERN) version
- `Old=.false.`       $\longrightarrow$       New TPSA package



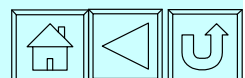
# Potential Calls to Init:2

The call `init(NO,ND,NP,NDPT,Old)` initializes the tools of LieLib, which are overloaded in `tpsalie` and `tpsalie_analysis`.

The order of the TPSA package is then:



The parameter `NDPT` is presently an input. It is related to the normal form process. It might be removed in future version. We will discuss its importance later. The main difference between `tpsalia` alone and `tpsalia + tpsalie_analysis` resides in the type `damap` and the physical assumption behind that `damap`.



# Types in TPSALIE: DAMAP

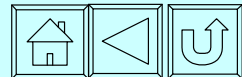
The type `damap` is defined as follows:

```
TYPE damap
type (taylor) v(ndim2)
END TYPE damap
```

The constant `ndim2` is equal to 6. This reflects the maximum size of the phase space dimension we normally considers in accelerator physics. However, when the subroutine `init(NO,ND,NP,NDPT,Old)` is called, only  $2*ND$  polynomials are allocated for each `damaps`.

In mathematical terms, the `type damap` represents a mapping from  $\mathbb{R}^{NV}$  to  $\mathbb{R}^{2*ND}$  where each  $2*ND$  (2,4,or 6) entry is a polynomial in `NV` variables.

It should said that these maps are truly “damap” (differential algebraic map), in other words, when they are concatenated using the overloaded (`*`), the constant part is set aside. To treat them as “TPSA maps”, one must used the new operator (`.o.`). In that case, constant parts are substituted in the polynomials.



## Example Program: Create a Rotation

```
program MAKE_A_MAP
use polymorphic_complex_taylor
implicit none
real*8 TWOPI !defined now in definition.f90 for convenience
integer NO,ND,NP,NDPT
type (damap) M, IDENTITY
type (pbfield) h

NO=5
ND=1
NP=0
NDPT=0
CALL INIT(NO,ND,NP,NDPT,.TRUE.)

call alloc(M)
call alloc(IDENTITY)
call alloc(H)

IDENTITY=1

H=-0.20d0*(TWOPI/2.D0)*(IDENTITY.V(1)**2+IDENTITY.V(2)**2)
M=TEXP(H,IDENTITY)

CALL DAPRINT(M,6)

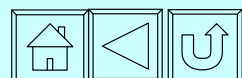
IDENTITY=M**5

CALL DAPRINT(IDENTITY,6)
call kill(H)
call kill(M)
call kill(IDENTITY)
end program MAKE_A_MAP
```

Poisson Bracket Operator will be used to create a rotation of tune 1/5.  
The `pbfield` type and the function `Texp` are explained momentarily.

The (`=`) sign has been overloaded to create easily an identity map.

The (`**`) has been overloaded on maps to produce power of concatenation (even) negative powers.



# Results of Rotation Program

Old Berzio

```

ETALL 1, NO = 5, NV = 2, INA = 44
*****
ORDER  COEFFICIENT  EXPONENTS
NO = 5  NV = 2
1 .3090169943749475  1 0
1 .9510565162951535  0 1
-2 .0000000000000000  0 0

ETALL 2, NO = 5, NV = 2, INA = 45
*****
ORDER  COEFFICIENT  EXPONENTS
NO = 5  NV = 2
1 -.9510565162951535  1 0
1 .3090169943749475  0 1
-2 .0000000000000000  0 0
    
```

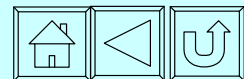
The map M

```

ETALL 1, NO = 5, NV = 2, INA = 46
*****
ORDER  COEFFICIENT  EXPONENTS
NO = 5  NV = 2
1 1.0000000000000000  1 0
1 -.4440892098500626E-15  0 1
-2 .0000000000000000  0 0

ETALL 2, NO = 5, NV = 2, INA = 47
*****
ORDER  COEFFICIENT  EXPONENTS
NO = 5  NV = 2
1 .4440892098500626E-15  1 0
1 1.0000000000000000  0 1
-2 .0000000000000000  0 0
    
```

The map M\*\*5



## Types in TPSALIE: Pbfield and Vecfield

The type Pbfield

```

TYPE pbfield
type (taylor) h
Integer ifac
END TYPE pbfield
    
```

The type Vecfield

```

TYPE vecfield
type (taylor) v(ndim2)
Integer ifac
END TYPE vecfield
    
```

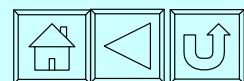
The **pbfield** type is a Poisson bracket operator. The exponential of such an object can act on a **dmap** and produce a **dmap**. The result is a symplectic map. The syntax was shown in the previous example, i.e.,  $M = \text{TEXP}(H, \text{IDENTITY})$ . In the notation of Dragt, we have  $M = \exp(:H:) \text{IDENTITY}$ .

The integer ifac is usually set to zero. However if the vector field is factorized (something we discuss later), this flag is set to +1 or -1. (See **WARNING further down**)

The **vecfield** type is a vector field operator. The exponential of such an object can also act on a **dmap** and produce a **dmap**. The syntax is identical to that of the **Pbfield**, i.e.,  $M = \text{TEXP}(H, \text{IDENTITY})$ . However this operator really does:

$$M = \exp(\vec{H} \cdot \vec{V}) \text{IDENTITY}$$

H is truly a a collection of **ND2** polynomials as hinted above.



# Pbfield and Vecfield: Continue

It is possible to convert between a **vecfield** object and **pbfield** object. For example, if we have:

Type (vecfield) F

Type (pbfield) H

We see here the Poisson bracket operator for a rotation followed by the equivalent vector field gotten by the equation  $F=H$ .

**The reverse equation  $H=F$  is self-consistent only if F is a Hamiltonian vector field.**

This is H

```
ETALL ,NO = 5,NV = 2,INA = 48
*****
```

ORDER	COEFFICIENT	EXPONENTS
	NO = 5 NV = 2	
2	-.6283185307179586	2 0
2	-.6283185307179586	0 2
-2	.0000000000000000	0 0

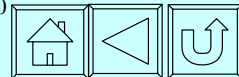
This is F

```
ETALL 1,NO = 5,NV = 2,INA = 49
*****
```

ORDER	COEFFICIENT	EXPONENTS
	NO = 5 NV = 2	
1	1.256637061435917	0 1
-1	.0000000000000000	0 0

```
ETALL 2,NO = 5,NV = 2,INA = 50
*****
```

ORDER	COEFFICIENT	EXPONENTS
	NO = 5 NV = 2	
1	-1.256637061435917	1 0
-1	.0000000000000000	0 0

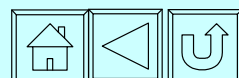


## Conclusion on TPSALIE

We have looked at the main functionality of the TPSALIE package, namely the introduction of a map of **ND2** dimension. This map can be concatenated with a similar map, it can be acted upon by a Lie operator (pbfield or vecfield), etc...

Other aspects of TPSALIE will be explained in our detailed description of the routines.

Now we are ready for a short glance at the analysis package, the TPSALIE\_ANALYSIS. This is truly the package containing the physics relevant to an accelerator.

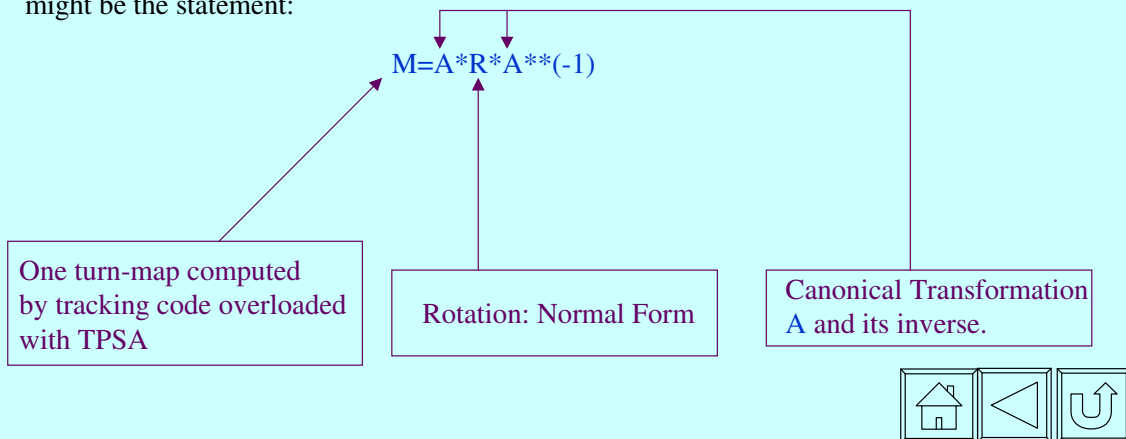




# The TPSALIE\_ANALYSIS Package

. This package is truly devoted to analysis of damaps. Its most important type is the **normalform**.

Normal forms are central concepts in the generalization of the Courant-Snyder theory to maps in general and also to time dependent vector fields, such as Hamiltonians. Since TPSA allows us to compute one-turn maps easily, the first thing we can do is to ask about its normal form. A normal form arises naturally when we investigate questions of stability in a periodic system. For example in a linear one-degree-of-freedom (**ND=1**) system a normal might be the statement:



## Useful Parameterization of Lie Maps

### DRAGT-FINN

```

TYPE DRAGTFINN
REAL(8) constant(ndim2)
type (damap) Linear
type (vecfield) nonlinear
type (pbfield) pb
END TYPE DRAGTFINN
    
```

This is the so-called Dragt-Finn representation of a Lie map. The map is represented as follows:

$$M = M_{\text{Linear}} \exp(\vec{F}_2 \cdot \vec{V}) \cdots \exp(\vec{F}_{\text{NO}} \cdot \vec{V})$$

Suppose we define a Drag-Finn map **df** and a damap **dm**:

```

Type (dragtfinn) df
Type (damap) dm
    
```

Then, through the overloading of the assignment (=), the line **df=dm** will produce the Dragt-Finn representation of **dm**.

The linear  $M_{\text{linear}}$  part will be stored **df.linear**.

The nonlinear part will be stored into **df.nonlinear** and the equivalent Poisson bracket operator into **df.pb**

The constant part will contain the constant part of the original map **dm**.

**NB:** The operations **damap=Texp(df.nonlinear,damap)** and **damap=Texp(df.pb,damap)** are now permitted. In fact the **vecfield** and **pbfield** types know what they are through **ifac**.

# Useful Parameterization of Lie Maps

## REVERSE-DRAGT-FINN, ONE-LIE-EXPONENT

```

TYPE REVERSEDRAAGTFINN
REAL(8) constant(NDIM2)
type (damap) Linear
type (vecfield) nonlinear
type (pbfield) pb
END TYPE REVERSEDRAAGTFINN
    
```

$$M = \exp(:\vec{F}_{NO} \cdot \vec{V} :) \cdots \exp(:\vec{F}_2 \cdot \vec{V} :) M_{Linear}$$

Everything is the same as with the Dragt-Finn representation except that the order is reversed.

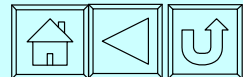
Another parametrization is the One-Lie-Exponent parameterization. The type definition is:

```

TYPE ONELIEEXPONENT
REAL(8) CONSTANT(NDIM2),EPS
type (vecfield) vector
type (pbfield) pb
END TYPE ONELIEEXPONENT
    
```

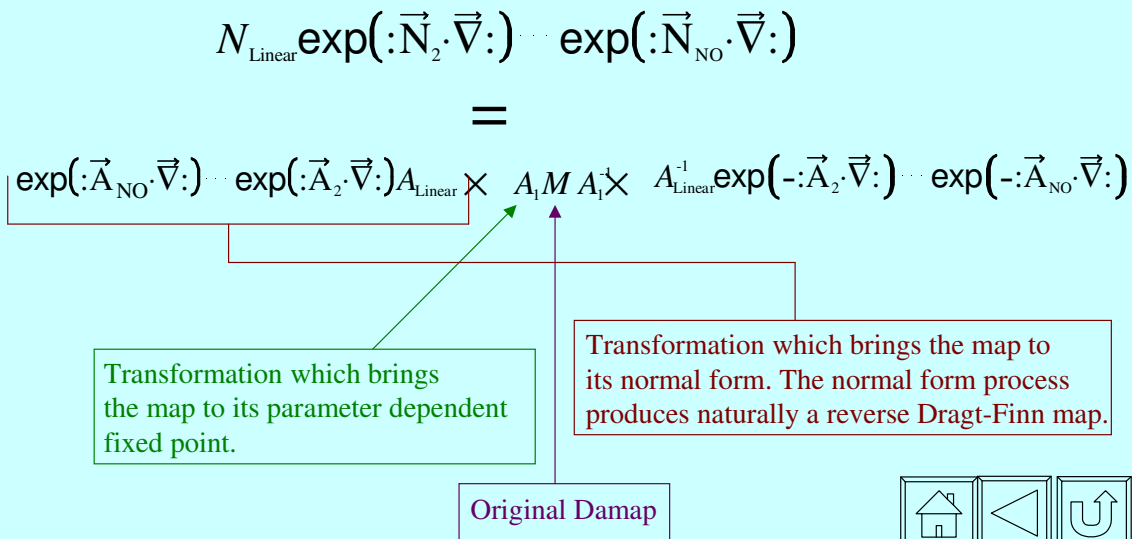
$$M = \exp(:\vec{F} \cdot \vec{V} :)$$

The vector field is computed by an iterative procedure which fails if  $M$  is far from the identity.



## The Mother of all Types: Normal Form

The Normal form is central to accelerator theory. It generalizes concepts such as the Courant-Snyder theory to nonlinear systems, coupled systems, and radiative systems. It also applies to hyperbolic systems as they are found near unstable fixed points (resonance extraction for example).



# The Normal Form Type:1

```

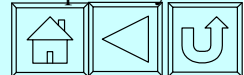
TYPE normalform
type (damap) A_T ← Total A for convenience
type (damap) A1 ← A1
type (reversedragtfinn) A ← exp(: $\vec{A}_{NO} \cdot \vec{V}$ :) ... exp(: $\vec{A}_2 \cdot \vec{V}$ :) ALinear

type (dragtfinn) NORMAL ← NLinear exp(: $\vec{N}_2 \cdot \vec{V}$ :) ... exp(: $\vec{N}_{NO} \cdot \vec{V}$ :)
type (damap) DHDJ
real(8) TUNE(NDIM),DAMPING(NDIM)
integer nord,jtune,
integer NRES,M(NDIM,NRESO),PLANE(NDIM)
logical AUTO
END TYPE normalform
    
```

Not discussed; on its way out.

The parameter `nord` is usually set to `NO` by default. It must be between `1` and `NO`. It indicates that the map is brought to its parameter dependent fixed point to order `nord`. Now is a good time to discuss the parameter `NDPT` of the subroutine `INIT`.

Role of `NDPT`: In the absence of a cavity and radiation, the longitudinal motion is drift-like in nature. In that case the energy, which is canonically conjugate to time, is a **constant** of the motion as well as a **canonical variable**. Because the longitudinal plane is normally the 3<sup>rd</sup> degree of freedom, the energy will be the 5<sup>th</sup> or 6<sup>th</sup> variable. The value of `NDPT` is precisely equal to 5 or 6, i.e., depending on where the energy is.



# The Normal Form Type:2

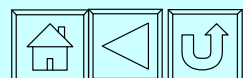
When `NDPT` is either 5 or 6, the normal form is called a *coasting beam* normal form. The transverse planes are reduced to rotations while the longitudinal plane becomes drift-like. The time of flight (or path length) increases proportionally to the energy deviation. The coefficient of proportionality is related to the **momentum compaction**.

The damap `DHDJ` of the normal form contain the tunes and momentum compaction **including the nonlinear parts**. They are written in Cartesian basis in the first `ND` entries of the map and in resonance basis in the last `ND` entries. The resonance basis will be discuss soon.

However, if resonances can be left in the map, this is achieved by filing up the variables `NRES` and `M(NDIM,NRESO)`. For example, a  $3Q_x$  resonance can be left in the normal form `N` by choosing:

```

NRES=1 ← One Resonance only
M(1,1)=3 }
M(2,1)=0 } ←  $\vec{m}=(3,0,0) \Rightarrow \vec{m} \cdot \vec{Q}=3Q_x$ 
M(3,1)=0 }
    
```



# The Normal Form Type:3

For convenience the tunes and damping are stored in the arrays:

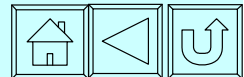
`TUNE(NDIM),DAMPING(NDIM)`

These are linear parts only; in the **coasting beam** normalization (`NDPT=5` or `6`) the momentum compaction is stored in `TUNE(3)`. The array `DAMPING` contains the damping coefficients when the map is not symplectic (**radiation**).

The parameter `JTUNE` is used in the radiative case. When the linear part of the map is nonsymplectic (i.e. damping is present), it is possible to remove all nonlinear terms by normal form. In this case `JTUNE` is set to `one`. However **this is not desirable** in accelerator physics, thus the default for this option is `JTUNE=0`.

Finally the logical `AUTO` is set to `.true.` if the normal form should be done without user interaction. Otherwise it should be set to `.false.` This affects the linear part of the normal form, the calculation of  $A_{Linear}$ .

We now go to a simple example of a normal form!



## Example: the Pendulum

```

program pendulum
use polymorphic_complex,taylor
type (pbfield) h
type (taylor) ht,x,px
type (damap) M ,A
type (NORMALFORM) N
integer NO,ND,NP,NDPT
REAL *8 K,DT,OMEGA,TWOPI

NO=4
ND=1
NP=0
NDPT=0
K=1.D0
DT=0.01D0
TWOPI=DATAN(1.D0)*8.D0
OMEGA=1.D0

Call INIT(NO,ND,NP,NDPT,.true.)

CALL ALLOC(M)
CALL ALLOC(A)
CALL ALLOC(N)
CALL ALLOC(H)
CALL ALLOC(HT)
CALL ALLOC(X)
CALL ALLOC(PX)

M=1
X=M.V(1)
PX=M.V(2)

HT=PX**2/2.D0+2.D0*(TWOPI*OMEGA)**2*DSIN(X/2.D0)**2
H=-DT*HT

M=TEXP(H,M)
N.AUTO=.TRUE.
N=M
A=N.A

N.DHDJ.V(1)=N.DHDJ.V(1)/DT
N.DHDJ.V(2)=N.DHDJ.V(2)/DT

CALL DAPRINT(N.DHDJ,6)

N.DHDJ=N.DHDJ*A**(-1) (See result)
CALL DAPRINT(N.DHDJ.V(1),6)

CALL KILL(M)
CALL KILL(A)
CALL KILL(N)
CALL KILL(H)
CALL KILL(HT)
CALL KILL(X)
CALL KILL(PX)
end program pendulum
    
```

**Hamiltonian** →  $H = -DT * HT$

**Lie operator for time=DT** →  $M = \text{TEXP}(H, M)$

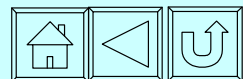
**Creates the map** →  $N = M$

**Put A in a damap** →  $A = N.A$

**Normalizes the map** →  $N = M$

**M is set identity** →  $M = 1$

**Tune shift per DT, i.e., 1/period =  $T^{-1}$**  →  $N.DHDJ.V(1) = N.DHDJ.V(1)/DT$   
 $N.DHDJ.V(2) = N.DHDJ.V(2)/DT$



# Pendulum: Theory

The Hamiltonian is given by:

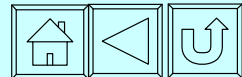
$$H = \frac{P_x^2}{2} + 2(2\pi\omega)^2 \sin^2(x/2)$$

The period T of oscillation can be expressed in terms of the initial displacement **d** of the pendulum.

$$T = \frac{1}{\pi\omega} \int_0^d \frac{dx}{\sqrt{\sin^2(d/2) - \sin^2(x/2)}}$$

This period can be re-expressed so that it is easily expandable in power of **d**.

$$T = \frac{2}{\pi\omega} \int_0^1 \frac{dx}{\sqrt{1-x^2} \sqrt{1 - \sin^2(d/2)x^2}} = \frac{1}{\omega} \left\{ 1 + \frac{d^2}{16} + \dots \right\}$$



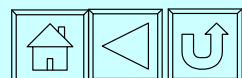
# Pendulum: Theory....

The tune Q is just T<sup>-1</sup>. Thus we have  $T^{-1} = \frac{1}{\omega} \left\{ 1 - \frac{d^2}{16} + \dots \right\}$ .

This is to be compared with the result of the [program pendulum](#).

```
ETALL 1, NO = 4, NV = 2, INA = 59
*****
```

```
ORDER    COEFFICIENT    EXPONENTS
NO = 4    NV = 2
0 1.000000000000001 0 0
2 -.624999999999998E-01 2 0
2 -.1583143494411528E-02 0 2
4 .3255208333333329E-02 4 0
4 -.1816038638913219E-17 3 1
4 -.9894646840072043E-04 2 2
4 -.2626164852475825E-19 1 3
4 -.1253171661948771E-05 0 4
-8 .000000000000000 0 0
```

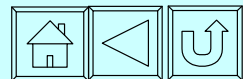


## Conclusion

In this document we have presented an overview of the original package prior to polymorphism. The actual code which does the overloading is easy to understand. In fact, just a knowledge of FORTRAN77 and a little intuition is enough to understand all the work that was done and even **modify it**.

Obviously the easiest package to understand is **TPSA.f90**. This package overloads the operation of Taylor Series and nothing else. One subtlety is the creation of scratch variables; for example a Berz-TPSA polynomial is just an integer pointing to a big array in Berz's package. The reader can see that creating a "scratch integer" would not be of much help. Therefore a routine called **ASS** assigns scratch TPSA polynomials to the overloading package. **(The Scratch variable scheme had to be modified to permit polymorphism)**

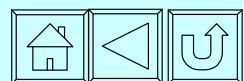
The rest of the package is easy to understand from a programming standpoint: just more of the same. However from a physics standpoint it makes sense only to those who have invested time in map methods and their relevance to tracking codes. We could go into the **theory of normal forms** in great detail here. We hope that the examples, plus access to the relevant references, will shed some light.



## New Things

### Complex TPSA and Real Polymorphism

- **Complex\_Taylor.f90** : the complex equivalent of **TPSA.f90**
- **Real\_polymorph.f90** : a type which can change at run time.
- **Complex Polymorph**



## Example of Complex Type

```

program test_complex
Use polymorphic_complex_taylor !As usual include the upper most library
implicit none
integer no
type (complex_taylor) c
type (taylor) a
double complex i

no=4
call init(no,3,.true.)

call alloc(a)
call alloc(c)

a=1.d0
a=a+(5.d0.mono.'10')
a=sin(a)

call daprint(a,6)

i=dcmplx(0.d0,1.d0)

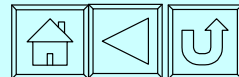
c=1.d0
c=c+(5.d0.mono.'10')
c=(exp(i*c)-exp(-i*c))/2/i ! That should be a the sine

call daprint(c,6)

end program test_complex

```

What is INIT?



## A Simple Example

```

PROGRAM TEST_POLYMORPHISM
USE POLYMORPHIC_COMPLEX_TAYLOR
IMPLICIT NONE
INTEGER NO, LNSTEP
TYPE (REAL_8) L,K,X(2),DL

CALL ALLOC(L)
CALL ALLOC(K)
CALL ALLOC(X,2)
CALL ALLOC(DL)

```

For security, variables are first initialized in a sexless state: not real, not TPSA

```

NSTEP=1000
NO=4
CALL INIT(NO,1,2,0,.TRUE.)

```

What is INIT?

The TPSA packages are initialized using INIT.

```

X(1)=0.D0
X(2)=0.D0
L=2.0
K=1.5
DL=L/NSTEP

```

These lines initialize the polymorph as real\*8

```

DO I=1,NSTEP
X(1)=X(1)+(DL/2.D0)*X(2)/DSQRT(1.D0-X(2)**2)
X(2)=X(2)-DL*K*X(1)
X(1)=X(1)+(DL/2.D0)*X(2)/DSQRT(1.D0-X(2)**2)
ENDDO

```

Integrate a quadrupole in canonical variables in one degree of freedom

```

CALL PRINT(X(1),6)
CALL PRINT(X(2),6)

```

Prints the results, which should be real\*8!

X=2

This array of polymorph is set to integer. This operation would be almost meaningless for a real array. In fact, when a polymorph is to an integer, it is ready to transform itself into TPSA as well as any polymorph it encounters.

```

X(1)=0.D0
X(2)=0.D0

```

In fact, X=2 means that X(1) will be a polynomial of the first TPSA variable while X(2) will be a polynomial of the second TPSA variable.

```

DO I=1,NSTEP
X(1)=X(1)+(DL/2.D0)*X(2)/DSQRT(1.D0-X(2)**2)
X(2)=X(2)-DL*K*X(1)
X(1)=X(1)+(DL/2.D0)*X(2)/DSQRT(1.D0-X(2)**2)
ENDDO

```

```

CALL PRINT(X(1),6)
CALL PRINT(X(2),6)

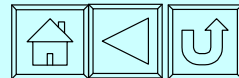
```

```

END PROGRAM TEST_POLYMORPHISM

```

Thus X(1)=0.d0 means X(1)= 0.d0+x<sub>1</sub> while X(2)=0.d0 implies X(2)= 0.d0+x<sub>2</sub>.



## Revisited Simple Example: Polymorphism using kind=0

```

program test_polymorphism
Use polymorphic_complex_taylor
implicit none
integer no,i,nstep
type (real_8) L,K,x(2),DL

CALL ALLOC(L)
CALL ALLOC(K)
CALL ALLOC(X,2)
CALL ALLOC(DL)

nstep=1000
no=4
call init(no,1,2,0,.true.)

X(1)=0.D0
X(2)=0.D0
L=2.0
K=1.5
DL=L/NSSTEP

do i=1,NSSTEP
X(1)=X(1)+(DL/2.D0)*X(2)/DSQRT(1.D0-X(2)**2)
X(2)=X(2)-DL*K*X(1)
X(1)=X(1)+(DL/2.D0)*X(2)/DSQRT(1.D0-X(2)**2)
ENDDO

CALL PRINT(X(1),6)
CALL PRINT(X(2),6)

CALL RESET(L)
CALL RESET(K)
CALL RESET(X,2)
write(6,*) " Control Integers for X(1) and X(2) "
read(5,*) X(1)%i , X(2)%i
write(6,*) " Control Integers for K and L "
read(5,*) K%i , L%i

X(1)=0.D0
X(2)=0.D0
L=2.0
K=1.5
DL=L/NSSTEP

do i=1,NSSTEP
X(1)=X(1)+(DL/2.D0)*X(2)/DSQRT(1.D0-X(2)**2)
X(2)=X(2)-DL*K*X(1)
X(1)=X(1)+(DL/2.D0)*X(2)/DSQRT(1.D0-X(2)**2)
ENDDO

CALL PRINT(X(1),6)
CALL PRINT(X(2),6)

End program test_polymorphism

```

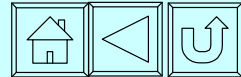
A variable which is reset becomes temporarily kind=0. Then, depending on the control integers, it will turn into either a taylor series or a real upon assigning a real number to it.

The other way to produce parameter dependence is a knob (kind=3).

Variables are reset.

The variable future TPSA status is set here.

The variables are initialized.



## Revisited Simple Example: Polymorphism using kind=3

```

program test_polymorphism
Use polymorphic_complex_taylor
implicit none
integer no,i,nstep
type (real_8) L,K,x(2),DL

CALL ALLOC(L)
CALL ALLOC(K)
CALL ALLOC(X,2)
CALL ALLOC(DL)

nstep=1000
no=4
call init(no,1,2,0,.true.)

X(1)=0.D0
X(2)=0.D0
L=2.0
K=1.5
DL=L/NSSTEP

do i=1,NSSTEP
X(1)=X(1)+(DL/2.D0)*X(2)/DSQRT(1.D0-X(2)**2)
X(2)=X(2)-DL*K*X(1)
X(1)=X(1)+(DL/2.D0)*X(2)/DSQRT(1.D0-X(2)**2)
ENDDO

CALL PRINT(X(1),6)
CALL PRINT(X(2),6)

K%kind=3 !CALL RESET(L)
L%kind=3 !CALL RESET(K)
CALL RESET(X,2)
write(6,*) " Control Integers for X(1) and X(2) "
read(5,*) X(1)%i , X(2)%i
write(6,*) " Control Integers for knobs K and L "
read(5,*) K%i , L%i

X(1)=0.D0
X(2)=0.D0
!L=2.0
!K=1.5
DL=L/NSSTEP

do i=1,NSSTEP
X(1)=X(1)+(DL/2.D0)*X(2)/DSQRT(1.D0-X(2)**2)
X(2)=X(2)-DL*K*X(1)
X(1)=X(1)+(DL/2.D0)*X(2)/DSQRT(1.D0-X(2)**2)
ENDDO

CALL PRINT(X(1),6)
CALL PRINT(X(2),6)

End program test_polymorphism

```

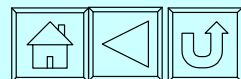
A variable is set kind=3.

Kind=3 act as knobs and cannot themselves change unless the global logical `setknob` is set to true (defaulted to false). A knob is taken into account if the logical `knob` is true (defaulted to false). Knobs should always be used in conjunction with TPSA calculations since even if `knob=.false.`, FPP calls the TPSA package for knobs.

Only X(2) is reset. Notice that it cannot be a knob.

The variable future TPSA status is set here.

The variables are not re-initialized anymore for kind=3.





# Analyzing the resulting map

```

program test_polymorphism
Use polymorphic_complex_taylor
implicit none
integer no,i,nstep
type (real_8) L,K,x(2),DL
type (damap) the_quad
type (normalform) N

CALL ALLOC(L)
CALL ALLOC(K)
CALL ALLOC(X,2)
CALL ALLOC(DL)

nstep=1000
no=4
call init(no,1,2,0,.true.)

write(6,*) " Control Integers for X(1) and X(2) "
read(5,*) X(1)%i , X(2)%i
write(6,*) " Control Integers for K and L "
read(5,*) K%i , L%i

X(1)=0.D0
X(2)=0.D0
L=2.0
K=1.5
DL=L/NSTEP

do i=1,NSTEP
X(1)=X(1)+(DL/2.D0)*X(2)/DSQRT(1.D0-X(2)**2)
X(2)=X(2)-DL*K*X(1)
X(1)=X(1)+(DL/2.D0)*X(2)/DSQRT(1.D0-X(2)**2)
ENDDO

CALL PRINT(X(1),6)
CALL PRINT(X(2),6)
call alloc(the_quad)
call alloc(N)
the_quad=x
n%auto=.true.
N=the_quad
call daprint(N%hdj%v(1),6)

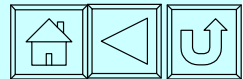
End program test_polymorphism

```

What is a damap?

What is a normal form?

← Creates a damap and a normal form.  
← Copies array of polymorphs into a damap.  
← Normalizes map; the result in N.



# The Definition File: Constants

This file contains comments, definition of constants and arrays, and finally definition of the basic types.

```

module definition
use define_newda
use scratch_size
implicit none
logical :: newread=.false., newprint = .false., first_time = .true.
logical :: print77=.true., read77 = .true.
logical :: no_ndum_check = .false.
logical :: setknob = .false., knob=.true.
double complex i_
double precision RAD_TO_DEG_, DEG_TO_RAD_,pil,pim,twopi
parameter (twopi=4.d0*1.57079632679489661923132169163975d0 )
parameter (pil=1.57079632679489661923132169163975d0-0.4d0 )
parameter (pim=1.57079632679489661923132169163975d0+0.4d0 )
parameter (RAD_TO_DEG_=57.2957795130823208767981548141052D0)
parameter (DEG_TO_RAD_=0.0174532925199432957692369076848861d0)
integer master,lnv
parameter (lnv=100)

```

← This imports the definition of a taylorlow.  
← This imports the size of scratch layers  
← Newread and newprint are a format used to accommodate a large number of variable for taylorlow. They are set to false.  
← Old printing format used mainly by the code Sixtrack90 of Schmidt. They are set to false.  
← Controls the behavior of knobs, i.e., polymorphs kind=3  
← Number of variables in polynomials.  
LNV=100 Maximum size of Newda. Berz's Package is set at nvmax=40.

The rest of these parameters are self evident except perhaps **pil** and **pim**. These are used in the polymorphic implementation of **datan2** and **datan2d**. Essentially **datan2** cannot be extended easily to Taylor series. Instead we must used **datan** away from  $\pm 90^\circ$  and use **dacos** in the vicinity of  $\pm 90^\circ$  ; **pil** and **pim** define the meaning of vicinity.

# The Definition File :Fundamental Types

```

TYPE taylor
INTEGER i      ! integer is a pointer in old da-package of Berz
type (taylorlow) j ! Taylorlow is the newda Taylor series
END TYPE taylor

TYPE complextaylor
type (taylor) r
type (taylor) i
END TYPE complextaylor

! this is a real polymorphic type
TYPE real_8
type (taylor) t
real*8 r
logical alloc
integer kind
integer i
double precision s ! scaling factor : useful in big tracking codes
END TYPE real_8

! this is a real polymorphic type
TYPE double_complex
type (complextaylor) t
double complex r
logical alloc
integer kind
integer i,j
double precision s ! scaling factor : useful in big tracking codes
END TYPE double_complex

!Radiation
TYPE ENV_8
type (REAL_8) v
type (REAL_8) e(ndim2)
END TYPE ENV_8

```

These are the four fundamental types void of physics and the polymorphic ENV\_8 used in a tracking code.

The type Taylor is just an ordinary taylor series. It can be created by the old DA-package of Berz or by newda. Technically more tpsa packages could be brought in.

The type complextaylor contains a real and imaginary part. Standard operations are permitted. In

**A linked list of scratch variables is now available by setting old\_scheme=false**

# The Definition File :Scratch variables

```

integer ndum,ndumt ,ndummax
parameter (ndum=5,ndumt=6,ndummax=100) ! ndum is used only for dummy maps now
integer, parameter :: ndumuser(ndumt)=(/72,45,22,22,22,22/)
! scratch variables
INTEGER iassdoluser(ndumt)
integer DUMMY,temp
integer DUMuser(ndumt,ndummax)
integer iass0user(ndumt)
integer ndim2,NDIM
parameter( ndim2 = 6)
PARAMETER (NDIM=NDIM2/2)
integer mmmmmm1,mmmmmm2,mmmmmm3,mmmmmm4
parameter (mmmmmm1=1,mmmmmm2=2,mmmmmm3=3,mmmmmm4=4)
type (taylorlow) DUMluser(ndumt,ndummax)
type (taylorlow) DUMMY1,temp1 !,DUMl(ndum)

```

Tells what layer of scratch variables is being used

Stuff in italic is now imported from a\_scratch\_size.f90

Number of scratch variables in each layer

Integers used to identify the state of a polymorph (kind) and perform the appropriate operations.

Global scratch variables represent perhaps the most subtle and annoying point of this package. I will first give the reason for their existence. Consider the following line

$a=(a+b)+(c+d)$  where a,b,c, and d are taylor types.

The compiler will need to put (a+b) in a scratch memory location  $m_1$  and then (c+d) in a scratch memory location called  $m_2$ . When the variables are real\*8, then compiler knows exactly what kind of memory needs to be allocated and does it automatically. In the case of taylor series or any other fundamental types such as complextaylor or polymorphs, the compiler is incapable of allocating by itself the proper memory. In addition if we allocate memory, we must also destroy it. This requires a severe problem of garbage collection which one cannot do easily in FORTRAN90. In addition, allocation and deallocation of dynamical variables (as used by the type taylorlow) slows down computation.

The solution adopted here is to use global scratch variables. There are six layers of such variables as indicated by the constant ndumt. This could be change.

# More on scratch variables


Consider the expression

$$\begin{array}{c}
 a=(a+b)+(c+d)+(f+e) \\
 \underbrace{\quad\quad}_S1 \quad \underbrace{\quad\quad}_S2 \quad \underbrace{\quad\quad}_S3 \\
 \underbrace{\quad\quad\quad}_S4 \\
 a= \quad \underbrace{\quad\quad\quad}_S5
 \end{array}$$

The reader will notice that the compiler may require 5 scratch variables if it is not too clever and only 3 if clever. Unfortunately, as a programmer, we have no way to overload the parentheses “(” and “)”. This means that we must for safety use 5 scratch variables. Since we can, of course, overload the “=” sign, we know that the assignment `a=(a+b)+(c+d)+(f+e)` did not use more than 5 scratch variables.

In summary, when our package uses one of the layers of scratch variables, it checks within an assignment if the number used, in this case 5, exceeds the number of the layer `ndumuser(i)` in the  $i^{\text{th}}$  layer. If so a severe warning is sent.

Finally, why many layers? Several layers are needed if new types are defined in terms of older ones containing taylor types. This is also true if the user insists on using functions rather than subroutines. We discourage this practice but if necessary, one must mimic the kind of footwork when one accesses a deeper layer of scratch variables. If there are not enough layers, a severe warning is sent and the user must increase the variable `ndumt`.

We now show an example of the use of a new layer in the case of a function of type `complextaylor` 

## Simple example of scratch layer

```

function tBst4(V0,w,a,h)
implicit none
type (complextaylor) tBst4
real*8, INTENT (IN) :: v0,a,h
type (complextaylor), INTENT (IN) :: w
integer localmaster

```

```
localmaster=master
```

The identity of level prior to the function being called is stored

```
call ass(tBst4)
```

The `complextaylor` `tBst4` is assigned to a scratch variable of the level `master+1`. If `master+1` exceeds the maximum number of levels `ndumt`, then the code terminates and a warning message is issued.

```
tBst4 = V0*i_ *w/a
```

```
master=localmaster
```

The prior level is restored

```
end function tBst4
```

If the maximum number of scratch variable is breached, the user can modify his code or add another layer. To do this a few changes need to be do through out the package. This is explained in the next slide.

## Adding a 7<sup>th</sup> Layer and Warnings

**A linked list of scratch variables is now available by setting `old_scheme=false`**

To achieve this it suffices to change `ndumt` and add one more parameter in `ndumuser`.  
The only changes are in `definition.f90`  
All other upgrades are automatic.

```
parameter (ndum=20,ndumt=7,ndummax=100)
integer, parameter :: ndumuser(ndumt)=(/20,45,22,22,22,22,20/)
```

It is a simple task. It is true that one could have written a complex garbage collection schemes, where instead of layers, one simply rotates through a large number of scratch variables. The advantage of layer is in our ability to check using the overloaded equal sign on the adequacy of each layer. The `check_snake` routine is called on routines overloading the assignment operator (`=`).

### **N.B. There are two things which are obviously dangerous.**

1) A recursive function. FORTRAN90 allows it. It should work but I have not tried it. It should be written with the same care as the example [tBst4](#).

2) An expression inside the call of a function or subroutine. For example, `call my_sub(a+a,b)`. Normally subroutines can be written without invoking a new layer of scratch variables unlike functions. In other words the care used in the example [tBst4](#) is not necessary. This makes overloading a real easy task. This is because the values `passed` or `returned` by a subroutine do not sit in an algebraic expression. In a call such as `“call my_sub(a+a,b)”` the variable `a+a` is an expression. Inside the subroutine the same layer of scratch variables will be used with the possibility that the intermediate variable `a+a` will **be overwritten without** the user’s knowledge. **Please avoid** this kind of syntax.