Revision: July 2, 2019

The Bmad Reference Manual

David Sagan

Overview

Bmad (Otherwise known as "Baby MAD" or "Better MAD" or just plain "Be MAD!") is a subroutine library for relativistic charged–particle and X-Ray simulations in accelerators and storage rings. *Bmad* has been developed at Cornell University's Laboratory for Elementary Particle Physics and has been in use since 1996.

Prior to the development of *Bmad*, simulation programs at Cornell were written almost from scratch to perform calculations that were beyond the capability of existing, generally available software. This practice was inefficient, leading to much duplication of effort. Since the development of simulation programs was time consuming, needed calculations where not being done. As a response, the *Bmad* subroutine library, using an object oriented approach and written in Fortran 2008, were developed. The aim of the *Bmad* project was to:

- Cut down on the time needed to develop programs.
- Cut down on programming errors.
- Provide a simple mechanism for lattice function calculations from within control system programs.
- Provide a flexible and powerful lattice input format.
- Standardize sharing of lattice information between programs.

Bmad can be used to study both single and multi–particle beam dynamics as well as X-rays. Over the years, *Bmad* modules have been developed for simulating a wide variety of phenomena including intra beam scattering (IBS), coherent synchrotron radiation (CSR), Wakefields, Touschek scattering, higher order mode (HOM) resonances, etc., etc. *Bmad* has various tracking algorithms including Runge–Kutta and symplectic (Lie algebraic) integration. Wake fields, and radiation excitation and damping can be simulated. *Bmad* has routines for calculating transfer matrices, emittances, Twiss parameters, dispersion, coupling, etc. The elements that *Bmad* knows about include quadrupoles, RF cavities (both storage ring and LINAC accelerating types), solenoids, dipole bends, Bragg crystals etc. In addition, elements can be defined to control the attributes of other elements. This can be used to simulate the "girder" which physically support components in the accelerator or to easily simulate the action of control room "knobs" that gang together, say, the current going through a set of quadrupoles.

One current area of development for *Bmad* is X-ray simulation. To that end, new element classes have been defined including a `mirror` element and a `crystal` element for simulations of crystal diffraction. The ultimate aim is to develop a environment where *Bmad* can be used for simulations starting from electron generation from a cathode, to X-ray generation in Wigglers and other elements, to X-ray tracking through to the experimental end stations.

To be able to extend *Bmad* easily, *Bmad* has been developed in a modular, object oriented, fashion to maximize flexibility. As just one example, each individual element can be assigned a particular tracking method in order to maximize speed or accuracy and the tracking methods can be assigned via the lattice file or at run time in a program.

Introduction

As a consequence of *Bmad* being a software library, this manual serves two masters: The programmer who wants to develop applications and needs to know about the inner workings of *Bmad*, and the user who simply needs to know about the *Bmad* standard input format and about the physics behind the various calculations that *Bmad* performs.

To this end, this manual is divided into three parts. The first two parts are for both the user and programmer while the third part is meant just for programmers.

Part I
: Part I discusses the *Bmad* lattice input standard. The *Bmad* lattice input standard was developed using the *MAD* [Grote96, Iselin94]. lattice input standard as a starting point but, as *Bmad* evolved, *Bmad*'s syntax has evolved with it.

Part II
: part II gives the conventions used by *Bmad*— coordinate systems, magnetic field expansions, etc. — along with some of the physics behind the calculations. By necessity, the physics documentation is brief and the reader is assumed to be familiar with high energy accelerator physics formalism.

Part III
: Part III gives the nitty–gritty details of the *Bmad* subroutines and the structures upon which they are based.

More information, including the most up–to–date version of this manual, can be found at the *Bmad* web site[Bmad]. Errors and omissions are a fact of life for any reference work and comments from you, dear reader, are therefore most welcome. Please send any missives (or chocolates, or any other kind of sustenance) to:

`David Sagan <dcs16@cornell.edu>`

The *Bmad* manual is organized as reference guide and so does not do a good job of instructing the beginner as to how to use *Bmad*. For that there is an introduction and tutorial on *Bmad* and *Tao* (§1.6) concepts that can be downloaded from the *Bmad* web page. Go to either the *Bmad* or *Tao* manual pages and there will be a link for the tutorial.

It is my pleasure to express appreciation to people who have contributed to this effort, and without whom, *Bmad* would only be a shadow of what it is today: To David Rubin for his support all these years, to Étienne Forest (aka Patrice Nishikawa) for use of his remarkable PTC/FPP library (not to mention his patience in explaining everything to me), to Desmond Barber for very useful discussions on how to simulate spin, to Mark Palmer, Matt Rendina, and Attilio De Falco for all their work maintaining the build system and for porting *Bmad* to different platforms, to Frank Schmidt and CERN for permission to use the *MAD* tracking code. to Hans Grote and CERN for granting permission to adapt two figures from the *MAD* manual for use in this one, to Martin Berz for his DA package, and to Dan Abell, Ivan Bazarov, Moritz Beckmann, Scott Berg, Joel Brock, Sarah Buchan, Avishek Chatterjee, Jing Yee Chee, Joseph Choi, Robert Cope, Jim Crittenden, Gerry Dugan, Christie Chiu, Michael Ehrlichman, Ken Finkelstein, Mike Forster, Thomas Gläßle Richard Helms, Georg Hoffstaetter, Chris Mayes, Karthik Narayan, Katsunobu Oide, Tia Plautz, Matt Randazzo, Michael Saelim, Jim Shanks, Jeff Smith, Jeremy Urban, Mark Woodley, and Demin Zhou for their help.

Contents

List of Figures

List of Tables

Part I

Language Reference

Chapter 1

Bmad Concepts and Organization

This chapter is an overview of some of the nomenclature used by *Bmad*. Presented are the basic concepts, such as `element`, `branch`, and `lattice`, that *Bmad* uses to describe such things as LINACs, storage rings, X-ray beam lines, etc.

1.1 Lattice Elements

The basic building block *Bmad* uses to describe a machine is the `lattice element`. An element can be a physical thing that particles travel "through" like a bending magnet, a quadrupole or a Bragg crystal, or something like a `marker` element (§3.28) that is used to mark a particular point in the machine. Besides physical elements, there are `controller` elements (Table 3.3) that can be used for parameter control of other elements.

Chapter §3 lists the complete set of different element types that *Bmad* knows about.

In a lattice `branch` (§1.2), The ordered array of elements are assigned a number (the element index) starting from zero. The zeroth `beginning_ele` (§3.4) element, which is always named `BEGINNING`, is automatically included in every branch and is used as a marker for the beginning of the branch. Additionally, every branch will, by default, have a final marker element (§3.28) named `END`.

1.2 Lattice Branches

The next level up from a `lattice element` is the `lattice branch`. A `lattice branch` contains an ordered sequence of lattice elements that a particle will travel through. A branch can represent a LINAC, X-Ray beam line, storage ring or anything else that can be represented as a simple ordered list of elements.

Chapter §6 shows how a `branch` is defined in a lattice file with `line`, `list`, and `use` statements.

A `lattice` (§1.3), has an array of `branches`. Each `branch` in this array is assigned an index starting from 0. Additionally, each `branch` is assigned a name which is the `line` that defines the branch (§6.6).

1.3 Lattice

an array of `branches` that can be interconnected together to describe an entire machine complex. A `lattice` can include such things as transfer lines, dump lines, x-ray beam lines, colliding beam storage rings, etc. All of which are connected together to form a coherent whole. In addition, a lattice may contain `controller elements` (Table 3.3) which can simulate such things as magnet power supplies and lattice element mechanical support structures.

Branches can be interconnected using `fork` and `photon_fork` elements (§3.19). This is used to simulate forking beam lines such as a connections to a transfer line, dump line, or an X-ray beam line. The `branch` from which other `branches` fork is called a `root` branch.

A lattice may contain multiple `root branches`. For example, a pair of intersecting storage rings will generally have two `root` branches, one for each ring. The `use` statement (§6.6) in a lattice file will list the `root branches` of a lattice. To connect together lattice elements that are physically shared between branches, for example, the interaction region in colliding beam machines, `multipass` lines (§7.2) can be used.

The root branches of a lattice are defined by the `use` (§6.6) statement. To further define such things as dump lines, x-ray beam lines, transfer lines, etc., that branch off from a root branch, a forking element is used. `Fork` elements can define where the particle beam can branch off, say to a beam dump. `photon_fork` elements can define the source point for X-ray beams. Example:

```
erl: line = (..., dump, ...)              ! Define the root branch
use, erl
dump: fork, to = d_line                    ! Define the fork point
d_line: line = (..., q3d, ...)             ! Define the branch line
```

Like the root branch *Bmad* always automatically creates an element with `element index` 0 at the beginning of each branch called `beginning`. The longitudinal `s` position of an element in a branch is determined by the distance from the beginning of the branch.

Branches are named after the line that defines the `branch`. In the above example, the branch line would be named `d_line`. The root branch, by default, is called after the name in the `use` statement (§6.6).

The "branch qualified" name of an element is of the form

```
branch_name>>element_name
```

where `branch_name` is the name of the branch and `element_name` is the "regular" name of the element. Example:

```
root>>q10w
xline>>cryst3
```

When parsing a lattice file, branches are not formed until the lattice is expanded (§2.23). Therefore, an `expand_lattice` statement is required before branch qualified names can be used in statements. See §2.7 for more details.

1.4 Lord and Slave Elements

A real machine is more than a collection of independent lattice elements. For example, the field strength in a string of elements may be tied together via a common power supply, or the fields of different elements may overlap.

Bmad tries to capture these interdependencies using what are referred to as `lord` and `slave` elements. The `lord` elements may be divided into two classes. In one class are the `controller` elements. These

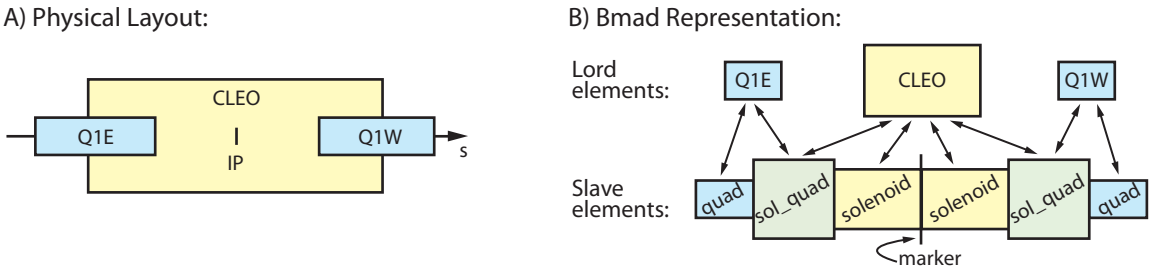


Figure 1.1: Superposition Example. A) Interaction region layout with quadrupoles overlapping a solenoid. B) The Bmad lattice representation has a list of split elements to track through and the undivided "lord" elements. Pointers (double headed arrows), keep track of the correspondence between the lords and their slaves.

are `overlay` (§3.36), `group` (§3.21), and `girder` (§3.20) elements that control the attributes of other elements which are their slaves.

The other class of `lord` elements embody the separation of the physical element from the track that a particle takes when it passes through the element. There are two types

An example will make this clear. `Superposition` (§7.1) is the ability to overlap lattice elements spatially. Fig. 1.1 shows an example which is a greatly simplified version of the IR region of Cornell's CESR storage ring when CESR was an e+/e– collider. As shown in Fig. 1.1A, two quadrupoles named `q1w` and `q1e` are partially inside and partially outside the interaction region solenoid named `cleo`. In the lattice file, the IR region layout is defined to be

```
cesr: line = (... q1e, dft1, ip, dft1, q1w ...)
cleo: solenoid, l = 3.51, superimpose, ref = ip
```

The line named `cesr` ignores the solenoid and just contains the interaction point marker element named `ip` which is surrounded by two drifts named `dft1` which are, in turn, surrounded by the `q1w` and `q1e` quadrupoles. The solenoid is added to the layout on the second line by using superposition. The "ref = ip" indicates that the solenoid is placed relative to `ip`. The default, which is used here, is to place the center of the superimposed `cleo` element at the center of the `ip` reference element. The representation of the lattice in *Bmad* will contain two branch `sections` ("sections" is explained more fully later): One section, called the `tracking section`, contains the elements that are needed for tracking particles. In the current example, as shown in Fig. 1.1B, the first IR element in the tracking section is a quadrupole that represents the part of `q1e` outside of the solenoid. The next element is a combination solenoid/quadrupole, called a `sol_quad`, that represents the part of `q1e` inside `cleo`, etc. The other branch section that Bmad creates is called the `lord section` This section contain the undivided "physical" `super_lord` elements (§7.1) which, in this case are `q1e`, `q1w`, and `cleo`. Pointers are created between the lords and their `super_slave` elements in the tracking section so that changes in parameters of the lord elements can be transferred to their corresponding slaves.

`super_lord`s are used when there are overlapping fields between elements, the other case where there is a separation between the physical element and the particle track comes when a particle passes through the same physical element multiple times such as in an Energy Recovery Linac or where different beams pass through the same element such as in an interaction region. In this case, `multipass_lords` representing the physical element and `multipass_slaves` representing the track can be constructed (§7.2). Superposition and multipass can be combined in situations where there are overlapping fields in elements where the particle passes through

Each lattice element is assigned a `slave_status` indicating what kind of slave it is and a `lord_status` indicating what kind of lord it is. Normally a user does not have to worry about this since these status

attributes are handled automatically by *Bmad*. The possible `lord_status` settings are:

girder_lord
A `girder_lord` element is a `girder` element (§3.20).

multipass_lord
`multipass_lord` elements are created when multipass lines are present (§7.2).

overlay_lord
An `overlay_lord` is an `overlay` element (§3.36).

group_lord
A `group_lord` is a `group` element (§3.21).

super_lord
A `super_lord` element is created when elements are superimposed on top of other elements (§7.1).

not_a_lord
This element does not control anything.

Any element whose `lord_status` is something other than `not_a_lord` is called a `lord` element. In the `tracking part` of the branch, `lord_status` will always be `not_a_lord`. In the `lord section` of the branch, under normal circumstances, there will never be any `not_a_lord` elements.

Lord elements are divided into two classes. A `major` lord represents a physical element which the slave elements are a part of. `super_lord`s and `multipass_lord`s are `major` lords. As a consequence, a `major` lord is a lord that controls nearly all of the attributes of its slaves. The other lords — `girder_lord`s, `group_lord`s and `overlay_lord`s — are called `minor` lords. These lords only control some subset of a slaves attributes.

The possible `slave_status` settings are

multipass_slave
A `multipass_slave` element is the slave of a `multipass_lord` (§7.2).

slice_slave
A `slice_slave` element represents a longitudinal slice of another element. Slice elements are not part of the lattice but rather are created on-the-fly when, for example, a program needs to track part way through an element.

super_slave
A `super_slave` element is an element in the tracking part of the branch that has one or more `super_lord` lords (§7.1).

minor_slave
`minor_slave` elements are elements that are not `slice_slave`s and are only controlled by `minor` lords (`overlay_lord`s, `group_lord`s, or `girder_lord`s).

free
A `free` element is an element with no lords.

For historical reasons, each `branch` in a lattice has a `tracking section` and a `lord section` and the `tracking section` is always the first (lower) part of the element array and the `lord section` inhabits the second (upper) part of the array. All the `lord` elements are put in the `lord section` of branch 0 and all the other `lord sections` of all the other branches are empty.

As a side note, Étienne Forest's PTC code (§1.5) uses separate structures to separate the physical element, which PTC calls an `element` from the particle track which PTC call a `fibre`. [Actually, PTC has two structures for the physical element, `element` and `elementp`. The latter being the "polymorph" version.] This `element` and `fibre` combination corresponds to *Bmad* `multipass_lord` and `multipass_slave` elements. PTC does not handle overlapping fields as *Bmad* does with `superposition` (§7.1).

1.5 PTC: Polymorphic Tracking Code

Étienne Forest[Forest98] has written what is actually two software libraries: FPP and PTC. FPP stands for "Fully Polymorphic Package." What this library does is implement Taylor maps (aka Truncated Power Series Algebra or TPSA) and Lie algebraic operations. Thus in FPP you can define a Hamiltonian and then generate the Taylor map for this Hamiltonian. FPP is very general. It can work with an arbitrary number of dimensions. FPP, however, is a purely mathematical package in the sense that it knows nothing about accelerator physics. That is, it does not know about bends, quadrupoles or any other kind of element, it has no conception of a lattice (a string of elements), it doesn't know anything about Twiss parameters, etc. This is where PTC (Polymorphic Tracking Code) comes in. PTC implements particle tracking and uses FPP as the engine to do the Lie algebraic calculations. For the purposes of this manual, PTC and FPP are generally considered one package and the combined PTC/FPP will be referred to as simply "PTC". For programmers, interface documentation can be found in chapter §34.

Bmad interfaces to PTC in two ways: One way, called "single element" mode, uses PTC on a per element basis. In this case, the method used for tracking a given element can be selected on an element-by-element basis so non-PTC tracking methods can be mixed with PTC tracking methods to optimize speed and accuracy. [PTC tends to be accurate but slow.] The advantage of single element mode is the flexibility it affords. The disadvantage is that it precludes using PTC's analysis tools which rely on the entire lattice being tracked via PTC. Such tools include normal form analysis beam envelope tracking, etc.

The alternative to single element mode is "whole lattice" mode where a series of PTC `layouts` (equivalent to a *Bmad* branch) are created from a *Bmad* lattice. Whether single element or whole lattice mode (or both) is used is determined by the program being run.

1.6 Tao: Tool for Accelerator Optics Program

The strength of *Bmad* is that, as a subroutine library, it provides a flexible framework from which sophisticated simulation programs may easily be developed. The weakness of *Bmad* comes from its strength: *Bmad* cannot be used straight out of the box. Someone must put the pieces together into a program. To partially remedy this problem, the *Tao* program[Tao] has been developed at Cornell. *Tao*, which uses *Bmad* as its simulation engine, is a general purpose program for simulating particle beams in accelerators and storage rings. Thus *Bmad* combined with *Tao* represents the best of both worlds: The flexibility of a software library with the ease of use of a program.

Chapter 2

Lattice Files

A lattice (§1) defines the sequence of elements that a particle will travel through along with the attributes (length, strength, orientation, etc.) of the elements. A lattice file (or files) is a file that is used to describe an accelerator or storage ring.

2.1 Bmad Lattice File Format

The syntax that a *Bmad* standard lattice file must conform to is modeled after the lattice input format of the *MAD* program. Essentially, a *Bmad* lattice file is similar to a *MAD* lattice file except that a *Bmad* file has no "action" commands (action commands tell the program to calculate the Twiss parameters, do tracking, etc.). Since *Bmad* is a software library, interacting with the user to determine what actions a program should take is left to the program and is not part of *Bmad* (although the *Bmad* library provides the routines to perform many of the standard calculations). A program is not required to use the *Bmad* parser routines but, if it does, the following chapters describe how to construct a valid lattice file.

2.2 MAD, SAD, and XSIF Lattice Files

Besides being able to parse *Bmad* lattice files, *Bmad* has software to parse XSIF[Tenen01] lattice files. See §12.2 for more details.

While *Bmad* cannot directly read in *MAD* [Grote96] or `SAD`[SAD] files, translation between *MAD* and *Bmad* lattice files is possible using the `Universal Accelerator Parser` as discussed in Chapter §12.

2.3 File Example and Syntax

The following (rather silly) example shows some of the features of a *Bmad* lattice file:

```
! This is a comment
parameter[E_TOT] = 5e9                    ! Parameter definition
pa1 = sin(3.47 * pi / c_light)                  ! Constant definition
bend1: sbend, type = "arc bend", l = 2.3,       ! An element definition
    g = 2*pa1, tracking_method = bmad_standard
bend2: bend1, l = 3.4                           ! Another element def
bend2[g] = 105 - exp(2.3) / 37.5                ! Redefining an attribute
ln1: line = (ele1, ele2, ele3)                  ! A line definition
ln2: line = (ln1, ele4, ele5)                   ! Lines can contain lines
arg_ln(a, b): line = (ele1, a, ele2, b)         ! A line with arguments.
use, ln2                                        ! Which line to use for the lattice
```

A *Bmad* lattice file consists of a sequence of statements. An exclamation mark (!) denotes a comment and the exclamation mark and everything after the exclamation mark on a line are ignored. *Bmad* is generally case insensitive. Most names are converted to uppercase. Exceptions where a name is not converted include file names and atomic formulas for materials used in crystal diffraction.

Normally a statement occupies a single line in the file. Several statements may be placed on the same line by inserting a semicolon (";") between them. A long statement can occupy multiple lines by putting an ampersand ("&") at the end of each line of the statement except for the last line. Additionally, lines that end with an "implicit continuation character" are automatically continued to the next line. The implicit continuation characters are

```
,   (   {   [   =
```

Notice that this is *not* like `C/C++`. Thus the following is bad syntax

```
wall = {
  section = {s = 0.45     ! BAD SYNTAX. NO CONTINUATION CHARACTER HERE.
  }                       ! BAD SYNTAX. NO CONTINUATION CHARACTER HERE.
}
```

Correct is:

```
wall = {
  section = {s = 0.45} }
```

or even:

```
wall = {
  section = {s = 0.45} &
}
```

Names of constants, elements, lines, etc. are limited to 40 characters. The first character must be a letter (`A` — `Z`). The other characters may be a letter, a digit (`0` — `9`) or an underscore (`_`). Other characters may appear but should be avoided since they are used by Bmad for various purposes. For example, the backslash (\) character is used to by Bmad when forming the names of superposition slaves (§7.1) and dots (`.`) are used by Bmad when creating names of `tagged` elements (§6.7). Also use of special characters may make the lattice files less portable to non-Bmad programs.

The following example constructs a linear lattice with two elements:

```
parameter[geometry] = open
parameter[e_tot] =2.7389062E9
parameter[particle] = POSITRON
beginning[beta_a] = 14.5011548
beginning[alpha_a] = -0.53828197
```

```
beginning[beta_b] = 31.3178048
beginning[alpha_b] = 0.25761815
q: quadrupole, l = 0.6, b1_gradient = 9.011
d: drift, l = 2.5
t: line = (q, d)
use, t
```

here `parameter[geometry]` (§8.1) is set to `open` which specifies that the lattice is not circular. In this case, the beginning Twiss parameters need to be specified and this is done by the `beginning` statements (§8.4). A quadrupole named `q` and a drift element named `d` are specified and the entire lattice consists of element `q` followed by element `d`.

2.4 Digested Files

Normally the *Bmad* parser routine will create what is called a "digested file" after it has parsed a lattice file so that when a program is run and the same lattice file is to be read in again, to save time, the digested file can be used to load in the lattice information. This digested file is in binary format and is not human readable. The digested file will contain the transfer maps for all the elements. Using a digested file can save considerable time if some of the elements in the lattice need to have Taylor maps computed. (this occurs typically with map–type wigglers).

Bmad creates the digested file in the same area as the lattice file. If *Bmad* is not able to create a digested file (typically because it does not have write permission in the directory), an error message will be generated but otherwise program operation will be normal.

Digested files contain the names of the lattice files used to create them. If a lattice file has been modified since the digested file has been created then the lattice files will be reread and a new digested file will be generated.

Note: If any of the random number functions (§2.13) are used in the process of creating the lattice, the digested file will be ignored. In this case, each time the lattice is read into a program, different random numbers will be generated for expressions that use such random numbers.

Digested files can also be used for easy transport of lattices between programs or between sessions of a program. For example, using one program you might read in a lattice, make some adjustments (say to model shifts in magnet positions) and then write out a digested version of the lattice. This adjusted lattice can now be read in by another program.

2.5 Element Sequence Definition

A `line` defines a sequence of elements. `lines` may contain other `lines` and so a hierarchy may be established. One line is selected, via a `use` statement, that defines the lattice. For example:

```
l3: line = (l1, l2)   ! Concatenate two lines
l1: line = (a, b, c)  ! Line with 3 elements
l2: line = (a, z)     ! Another line
use, l3               ! Use l3 as the lattice definition.
```

In this case the lattice would be

```
(a, b, c, a, z)
```

`Lines` can be defined in any order. See Chapter 6 for more details.

The `superimpose` construct allows elements to be placed in a lattice at a definite longitudinal position. What happens is that after a lattice is expanded, there is a reshuffling of the elements to accommodate any new superimpose elements. See §7.1 for more details.

2.6 Lattice Elements

The syntax for defining a lattice element roughly follows the *MAD* [Grote96] program:

```
ele_name: keyword [, attributes]
```

where `ele_name` is the element name, `keyword` is the type of element, and `attributes` is a list of the elements attributes. Chapter 3 gives a list of elements types with their attributes. `Overlay` and `group` type elements have a slightly different syntax:

```
ele_name: keyword = { list }, master-attribute [= value] [, attributes]
```

and `Girder` elements have the syntax

```
ele_name: keyword = { list } [, attributes]
```

For example:

```
q01w: quadrupole, type = "A String", l = 0.6, tilt = pi/2
h10e: overlay = { b08e, b10e }, var = {hkick}
```

2.7 Lattice Element Names

A valid element name may be up to 40 characters in length. The first character of the name must be a letter [A-Z]. After that, the rest of the name can contain only letters, digits [0-9], underscore "_", period ".", backslash "\", or a hash mark "#". It is best to avoid these last three symbols since *Bmad* uses them to denote "relationships". Periods are used for tagging (§6.7), and backslash and hash marks are used for to compose names for superposition (§7.1) and multipass (§7.2) slave elements.

There is a short list of names that cannot be used as an element name. These reserved names are:

```
beam
particle_start
beginning
debug_marker
end
no_digested
parameter
parser_debug
print
root
title
use
write_digested
```

Where appropriate, for example when setting element attributes (§2.8), the wild cards **''*''** and **''%''** can be used to select multiple elements. The **''*''** character will match any number of characters (including zero) while **''%''** maches to any single character. Additionally, matching can be restricted to a certain element class using the syntax:

```
class::element_name
```

where `class` is a class name. For example:

```
m*              ! Match to all elements whose name begins with "m".
a%c             ! Match to "abc" but not to "ac" or "azzc".
quadrupole::*w  ! Match to all quadrupoles whose name ends in "w"
```

After lattice expansion (§2.23), the general syntax to specify a set of elements is:

```
{class::}{branch_id>>}element_id{##N}
```

where `{...}` marks an optional component, `class` is a class name, `branch_id` is a branch name or index (§1.2), `element_id` is and element name or element index (§6.2), and `##N` indicates that the N^{th} matching element is to be used. Examples:

```
quad::x_br>>q*        ! All quadrupoles of branch "x_br" whose name begins with "q".
2>>45                 ! element #45 of branch #2.
q01##3                ! The 3rd element in each branch named q01.
```

`Note:` It is advised to avoid setting the parameters of differing elements that share the same name to differing values since this can lead problems later on. For example, consider this in a lattice file named, say, `lat.bmad`:

```
q1: quadrupole, ...
a: line = (..., q1, ...)
b: line = (..., q1, ...)
c: line = (a, b)  ! There are two q1 elements.
                  ! One from A-line and one from B-line.
use, c
expand_lattice    ! Expand the lattice
q1##1[k1] = 0.1   ! Set first element whose name is q1
q1##2[k1] = 0.2   ! Set second element whose name is q1
```

Now if later on someone wants to study just the `B` line that person could try to do this by creating a second file with just two lines:

```
call, file = lat.bmad
use, b
```

Normally this would work but in this case the lattice is invalid since there is only one `q1` element in line `B`. A more flexible solution would be to use unique names for the two `q1` elements.

Multiple elements in a lattice may share the same name. When multiple branches are present, to differentiate elements that appear in different branches, the "branch qualified" element name may be used. The branch qualified element name is of the form

```
branch_name>>element_name
```

where `branch_name` is the name of the branch and `element_name` is the "regular" name of the element. Example:

```
root>>q10w
x_branch>>crystal3
```

For `branch` lines (§1.2), the full "branch qualified" name of an element is of the form

```
branch_name>>element_name
```

where `branch_name` is the name of the branch and `element_name` is the "regular" name of the element. Example:

```
root>>q10w
xline>>cryst3
```

Using the full name is only needed to distinguish elements that have the same regular name in separate branches. When parsing a lattice file, branches are not formed until the lattice is expanded (§2.23). Therefore an `expand_lattice` statement is required before full names can be used in statements.

After lattice expansion (§2.23), when setting element attributes (§2.8, a comma deliminated list of names can be used. Each item in a list is either the name of an element or an element range. An element range has the syntax:

```
key::ele1:ele2
```

where

```
key      = Optional key name ("quadrupole", "sbend", etc.)
ele1     = Starting element of the range.
ele2     = Ending element of the range.
```

For example:

```
3,15:17          ! Elements with index 3, 15, 16, and 17 in branch 0.
2>>45:51         ! Elements 45 through 51 of branch 2.
q1:q5            ! Elements between q1 and q5.
sbend::q1:q5     ! All sbend elements between q1 and q5.
```

2.8 Lattice Element Attributes

Any lattice element has various attributes like its name, its length, its strength, etc. The values of element attributes can be specified when the element is defined. For example:

```
b01w: sbend, l = 6.0, rho = 89.0 ! Define an element with attributes.
```

After an element's definition, an individual attribute may be referred to using the syntax

```
class::element_name[attribute_name]
```

Element attributes can be set or used in an algebraic expression:

```
b01w[roll] = 6.5                  ! Set an attribute value.
b01w[L] = 6.5                     ! Change an attribute value.
b01w[L] = b01w[rho] / 12          ! OK to reset an attribute value.
my_const = b01w[rho] / b01w[L]    ! Use of attribute values in an expression.
```

Notice that there can be no space between the element name and the `[` opening bracket.

Chapter Chapter 3 lists the attributes appropriate for each element class.

When setting an attribute value, if more than one element has the `element_name` then *all* such elements will be set. When setting an attribute value, if `element_name` is the name of a type of element, all elements of that type will be set. For example

```
q_arc[k1] = 0.234                     ! Set all elements named Q_ARC.
rfcavity::*[voltage] = 3.7            ! Set all RFcavity elements.
```

To set an attribute for multiple element at one time, The wild cards "`*`", and "`%`" can be used in element names (§2.7). Examples:

```
*[tracking_method] = bmad_standard  ! Matches all elements.
quadrupole::Q*[k1] = 0.234    ! Matches all quadrupoles with names beginning with Q.
Q%1[k1] = 0.234               ! Matches to "Q01" but not "Q001".
```

Unlike when there are no wild cards used in a name, it is not an error if a name with wild cards does not match to any element. Note: A name with wild cards will never match to the `BEGINNING` element (§6.6).

After lattice expansion (§2.23), the attributes of specific elements may be set using the syntax as discussed in Section §2.7. Example:

```
expand_lattice              ! Expand the lattice.
97[x_offset] = 0.0023       ! Set x_offset attribute of 97th element
b2>>si_cryst##2[tilt] = 0.1 ! Tilt the 2nd instance of "si_cryst" in branch "b2"
5:32[x_limit] = 0.3         ! Sets elements with indexs 5 through 32 in branch 0.
```

2.9 Custom Element Attributes

Real scalar and vector custom element attributes may be defined for any class of element and real scaler parameters can be defined for the lattice as a whole. Custom element attributes are useful with programs that need to associate "extra" information with particular lattice elements or the lattice itself and it is desired that this extra information be settable from within a lattice file. For example, a program might need an error tolerance for the strength of quadrupoles.

Adding custom attributes will not disrupt programs that are not designed to use the custom attributes. Currently, up to 40 named custom attributes may be defined for any given element class. The syntax for defining custom attributes is:

```
parameter[custom_attributeN] = {class_name::}attribute_name
```

Where "`N`" is an integer between 1 and 40 and "`attribute_name`" is the name of the attribute. To restrict the custom attribute to a particular element class, the element class can be prefixed to the attribute name. To define a global parameter for the lattice, use `parameter`" as the class name. Examples:

```
parameter[custom_attribute1] = quadrupole::error_k1
parameter[custom_attribute1] = mag_id
parameter[custom_attribute1] = sextupole::error_k2
parameter[custom_attribute2] = color
parameter[custom_attribute2] = parameter::quad_mag_moment
```

The first line in the example assigns, for the first custom attribute group (`custom_attribute1`), a name of `error_k1` to all quadrupoles. The second line in the example assigns to the first custom attribute group the name `mag_id` to all element classes except quadrupoles since that class of element already has an assigned name. The third line assigns, for the first custom attribute group, a name of `error_k2` to all sextupoles overriding the `mad_id` name. The fourth line in the above example assigns, for the second custom attribute group, a name of `color` to all element classes. Finally, the last line defines a global parameter called `quad_mag_moment`.

Once a custom attribute has been defined it may be set for an element of the correct class. Example:

```
parameter[custom_attribute2] = lcavity::rms_phase_err
parameter[custom_attribute3] = parameter::cost
...
parameter[cost] = 140000000
l2a: lcavity, rms_phase_err = 0.0034, ...
```

Notice that defining the name for a custom attribute must come before its use.

Custom attributes that are assigned to an individual element class, like `error_k1` above, are called "`class-specific`" attributes. Custom attributes, like `mag_id` above, that are assigned to all element classes, are called "`common`" attributes. For a given custom attribute group, The setting of a `class-specific` attribute will take precedence over the setting of a `common` attribute. Thus, in the above example, the fact that `quadrupole::error_k1` comes before `mag_id` and `sextupole::error_k2` appears after does not affect anything. Once a `common` attribute is defined for a given custom attribute group, it cannot be changed. Similarly, once a `class-specific` attribute is defined for a given class for a given custom attribute group it cannot be changed. Trying to redefine a given custom attribute using a new name that is the same as the old name is not considered an error. For example, the following is OK:

```
parameter[custom_attribute2] = color
parameter[custom_attribute2] = color   ! OK since the same name is used.
```

Custom attributes are global in a program and not lattice-specific. That is, if a program reads in two different lattices the custom attribute settings of both lattices will be combined.

For someone creating a program, section §27.19 describes how to make the appropriate associations.

Note: If custom string information needs to be associated with an element, the `type`, `alias` and `descrip` element components (§4.3) are available.

Besides the named custom attributes described above, there is a three dimensional vector, called `r_custom`, associated with each element that can be used to store numbers. For example:

```
qq: quadrupole, r_custom(-2,1,5) = 34.5, r_custom(-3) = 77.9
```

Negative indices are accepted and if only one or two indices are present, the others are assumed to be zero. Thus `r_custom(-3)` is equivalent to `r_custom(-3,0,0)`.

Note: When there is a superposition (§7.1), the `super_slave` elements that are formed do *not* have any custom attributes assigned to them even when their `super_lord` elements have custom attributes. This is done since the *Bmad* bookkeeping routines are not able to handle the situation where a `super_slave` element has multiple `super_lord` elements and thus the custom attributes from the different `super_lord` elements have to be combined. Proper handling of this situation is left to any custom code that a program implements to handle custom attributes.

2.10 Variable Types

There are five types of variables in *Bmad*: reals, integers, switches, logicals (booleans), and strings. Acceptable logical values are

```
true    false
t       f
```

For example

```
rf1[is_on] = False
```

String literals can be quoted using double quotes (") or single quotes ('). If there are no blanks or commas within a string, the quotes can be omitted. For example:

```
Q00W: Quad, type = "My Type", alias = Who_knows, &
                                  descrip = "Only the shadow knows"
```

Unlike most everything else, strings are not converted to uppercase.

Switches are variables that take discrete values. For example:

```
parameter[particle] = positron
q01w: quad, tracking_method = bmad_standard
```

The name "switch" can refer to the variable (for example, `tracking_method`) or to a value that it can take (for example, `bmad_standard`). The name "method" is used interchangeably with switch.

2.11 Units and Constants

Bmad uses SI (Système International) units as shown in Table 2.1. Note that *MAD* uses different units. For example, *MAD*'s unit of Particle Energy is GeV not eV.

Note: For compatibility with *MAD*, the `beam, energy = xxx` command (§8.3) uses GeV and the `emass` and `pmass` constants (see below this section) also use GeV. It is recommended that the use of these constructs be avoided.

Quantity	*Units*
Angles	radians
Betatron Phase	radians
Current	Amps
Frequency	Hz
Kick	radians
Length	meters
Magnetic Field	Tesla
Particle Energy	eV
RF Phase Angles	radians/2π
Voltage	Volts

Table 2.1: Physical units used by *Bmad*.

Bmad defines commonly used physical and mathematical constants shown in Table 2.2. All symbols use straight SI units except for `emass` and `pmass` which are provided for compatibility with *MAD* and *should be avoided*.

As an alternative, the `mass_of`, and `anomalous_moment_of` functions (§2.13) may be used in place of the defined constants for mass and anomalous magnetic moment.

Note: The standard definition of the magnetic moment g-factor for spin 1/2 fundamental particles is

$$\mu = g \frac{q}{2\,m} \mathbf{S} \tag{2.1}$$

where μ is the magnetic moment, q is the particle charge, and m is the mass. The anomalous moment a is then defined to be

$$a = \frac{g-2}{2} \tag{2.2}$$

For nuclei and other composite baryonic particles, it is conventional to define the g-factor using

$$\mu = g \frac{e}{2\,m_p} \mathbf{S} \tag{2.3}$$

where m_p is the mass of the proton. This is inconvenient for calculations since an equation like Eq. (19.2) would not work for all particles. To get around this, the g-factors used by Bmad are always derived from Eq. (2.1) (think of this as an "effective" g-factor).

Symbol	*Value*	*Units*	*Name*
pi	3.14159265359		
twopi	2 * pi		
fourpi	4 * pi		
e_log	2.718281828		
sqrt_2	1.4142135623731		
degrad	180 / pi		From rad to deg
degrees	pi / 180		From deg to rad
raddeg	pi / 180		From deg to rad
anom_moment_deuteron	-0.1429872724		Deuteron anomalous magnetic moment*
anom_moment_electron	0.001159652193		Electron anomalous magnetic moment
anom_moment_muon	0.0011659208		muon anomalous magnetic moment
anom_moment_proton	1.79285		proton anomalous magnetic moment
anom_moment_he3	-4.184153686		He^3 anomalous magnetic moment*
fine_struct_const	0.00729735257		Fine structure constant
m_deuteron	$1.875612928 \cdot 10^9$	eV	Deuteron mass
m_electron	$0.5109989461 \cdot 10^6$	eV	Electron mass
m_muon	$105.6583715 \cdot 10^6$	eV	Muon mass
m_pion_0	$134.9766 \cdot 10^6$	eV	π^0 mass
m_pion_charged	$139.57018 \cdot 10^6$	eV	π^+, π^- mass
m_proton	$0.9382720813 \cdot 10^9$	eV	Proton mass
c_light	$2.99792458 \cdot 10^8$	m/sec	Speed of light
r_e	$2.8179403227 \cdot 10^{-15}$	m	Electron radius
r_p	$1.5346980 \cdot 10^{-18}$	m	Proton radius
e_charge	$1.6021766208 \cdot 10^{-19}$	Coul	Electron charge
h_planck	$4.13566733 \cdot 10^{-15}$	eV*sec	Planck's constant
h_bar_planck	$6.58211899 \cdot 10^{-16}$	eV*sec	Planck / 2π
emass	$0.5109989461 \cdot 10^{-3}$	GeV	Electron mass (please avoid using)
pmass	0.9382720813	GeV	Proton mass (please avoid using)

* Effective anomalous moments. See the discussion after Eq. (2.1).

Table 2.2: Physical and mathematical constants recognized by *Bmad*.

2.12 Arithmetic Expressions

Arithmetic expressions can be used in a place where a real value is required. The standard operators are defined:

$a + b$	Addition
$a - b$	Subtraction
$a * b$	Multiplication
a / b	Division
$a \wedge b$	Exponentiation

Bmad also has a set of intrinsic functions. A list of these is given in §2.13.

Literal constants can be entered with or without a decimal point. An exponent is marked with the letter E. For example

```
1, 10.35, 5E3, 314.159E-2
```

Symbolic constants can be defined using the syntax

```
constant_name = expression
```

Alternatively, to be compatible with *MAD*, using ":=" instead of "=" is accepted

```
constant_name := expression
```

Examples:

```
my_const = sqrt(10.3) * pi^3
abc      := my_const * 23
```

Unlike *MAD*, *Bmad* uses immediate substitution so that all constants in an expression must have been previously defined. For example, the following is *not* valid:

```
abc      = my_const * 23      ! No: my_const needs to be defined first.
my_const = sqrt(10.3) * pi^3
```

here the value of `my_const` is not known when the line "`abc` = ..." is parsed. Once defined, symbolic constants cannot be redefined. For example:

```
my_const = 1
my_const = 2  ! No: my_const cannot be redefined.
```

`group` (§3.21) and `overlay` (§3.36) controller elements are an exception to the immediate evaluation rule. Since controller elements may control elements that do not exist until `lattice expansion` (§2.23), the arithmetic expressions associated with controller elements are not evaluated until lattice expansion. Example:

```
s_20W: sextupole, l = 0.27
sk: overlay = {s_20W[a1]:-2*s_20W[L]}, var = {k1}, k1 = 0.2
s_20W[L] = 0.34
s_30E: s_20W
...
expand_lattice
```

Here the expression of overlay `sk` is evaluated, when the lattice is expanded, to be `-0.68 = -2*0.34`. This uses uses the length of element `s_20W` at the point when the lattice is expanded and not at the point when `sk` was defined. Additionally, the element `s_30E`, which inherits the attributes of `s_20W`, inherits a value of zero for `a1` (skew multipole moment) since inheritance uses immediate evaluation just like the setting of constants.

Element attributes can be used after they have been defined but not before. Example:

```
sa: sextupole, l = 0.3, k2 = 0.01 * sa[L]   ! Good
sb: sextupole, k2 = 0.01 * sb[L], l = 0.3   ! BAD SET OF K2. L IS DEFINED AFTER.
```

In this example, the `k2` attribute of element `sa` is correctly set since `k2` is defined after `l`. On the other hand, `k2` of element `sb` will have a value of zero since `l` of `sb` defaults to zero before it is set.

One potential pitfall with immediate substitution is that when an element attribute changes, it does not affect prior evaluations. Example:

```
s1: sextupole, k2 = 2.3
aa = s1[k2]              ! aa = 2.3
s1[k2] = 1.7             ! value of aa does not change
```

Here the value of constant `aa` will remain fixed at 2.3 no matter how the value of `s1[k2]` is altered after `aa` is defined.

Another potential pitfall is when using dependent element attributes (§4.1). For example:

```
b01w: sbend, l = 0.5, angle = 0.02
a_const = b01w[g]    ! No: bend g has not yet been computed!
```

Here the bend strength `g` (§3.5) will eventually be computed to be 0.04 (= angle / l) but that computation does not happen until lattice expansion (§2.23). In this case, the value of `a_const` will be the default value of `g` which is zero. As a rule of thumb, never rely on dependent attributes having their correct value.

2.13 Intrinsic functions

The following intrinsic functions are recognized by *Bmad*:

`sqrt`(x)	Square Root
`log`(x)	Logarithm
`exp`(x)	Exponential
`sin`(x)	Sine
`cos`(x)	Cosine
`tan`(x)	Tangent
`asin`(x)	Arc sine
`acos`(x)	Arc cosine
`atan`(x)	Arc Tangent
`atan2`(y, x)	Arc Tangent of y/x
`abs`(x)	Absolute Value
`factorial`(n)	Factorial
`ran`()	Random number between 0 and 1
`ran_gauss`()	Gaussian distributed random number
`int`(x)	Nearest integer with magnitude less then x
`nint`(x)	Nearest integer to x
`floor`(x)	Nearest integer less than x
`ceiling`(x)	Nearest integer greater than x
`mass_of`(A)	Mass of particle A
`charge_of`(A)	Charge, in units of the elementary charge, of particle A
`anomalous_moment_of`(A)	Anomalous magnetic moment of particle A
`species`(A)	Species ID of A

`ran_gauss` is a Gaussian distributed random number with unit RMS. Both `ran` and `ran_gauss` use a seeded random number generator. To choose the seed set

```
parameter[ran_seed] = <Integer>
```

A `value` of zero will set the seed using the system clock so that different sequences of random numbers will be generated each time a program is run. The default behavior if `parameter[ran_seed]` is not present is to use the system clock for the seed.

If an element is used multiple times in a lattice, and if `ran` or `gauss_ran` is used to set an attribute value of this element, then to have all instances of the element have different attribute values the setting of the attribute must be after the lattice has been expanded (§2.23). For example:

```
a: quad, ...
a[x_offset] = 0.001*ran_gauss()
my_line: line = (a, a)
use, my_line
```

Here, because *Bmad* does immediate evaluation, the `x_offset` values for `a` gets set in line 2 and so both copies of `a` in the lattice get the same value. This is probably not what is wanted. On the other hand if the attribute is set after lattice expansion:

```
a: quad, ...
my_line: line = (a, a)
use, my_line
expand_lattice
a[x_offset] = 0.001*ran_gauss()
```

Here the two `a` elements in the lattice get different values for `x_offset`.

The `mass_of`, `charge_of`, and `anomalous_moment_of` functions give the mass of, charge of (in units of the elementary charge), and anomalous moment of, a particle. Example:

```
parameter[particle] = deuteron
am = anomalous_moment_of(parameter[particle])^2
my_particle = species(He++)      ! my_particle now represents He++
chg1 = charge_of(my_particle)    ! chg = charge of He++
chg2 = charge_of(He++)           ! Same as previous line
chg3 = charge_of(species(He++))  ! Same as previous line
```

The `species` function is needed in the definition of `my_particle` so that *Bmad* knows that the string "He++" represents a type of particle. Inside functions like `mass_of`, the use of `species` is optional since, in this case, *Bmad* can correctly parse the argument.

2.14 Statement Order

With some exceptions, statements in a lattice file can be in any order. For example, the lines (§6.2) specified in a `use` statement (§6.6) can come after the `use` statement. And `group` (§3.21) and `overlay` (§3.36) controller elements may be defined before the slave elements whose parameters they control are defined.

The exceptions to this rule are:

- If there is an `expand_lattice` statement (§2.22), everything necessary for lattice expansion must come before. In particular, all `lines` (§6.2), `lists` (§6.5), and `use` (§6.6) statements necessary for lattice expansion must come before.
- Immediate evaluation of arithmetic expressions (§2.12) mandates that values be defined before use.
- A lattice element must be defined before any of its parameters are set. Example:
  ```
  pp[z_offset] = 0.1    ! WRONG! PP HAS NOT BEEN DEFINED YET!
  pp: patch             ! Here PP is defined
  ```
  In this example, the `z_offset` of the element `pp` is set before `pp` has been defined. This is an error. As a corollary to this rule, element parameters that are set using wild card characters will only affect those parameters that have been already defined. For example:
  ```
  crystal::*[b_param] = 0.2
  c5: crystal
  ```
  In this example, the `b_param` of all `crystal` elements is set to `0.2` *except* for `c5` and all other crystal elements that are defined after the set.

2.15 Print Statement

The `print` statement prints a message at the terminal when the lattice file is parsed by a program. Syntax:

```
print <string>
```

Where `<string>` is the string to be printed. Variable values can be printed by using **back-tick** characters. For example:

```
print Remember! Q01 quad strength of `q01[k1]` not yet optimized!
print Optimization is as easy as 2 + 2 = `2+2`.
```

will result in the following being printed:

```
Message in Lattice File: Remember! Q01 quad strength of 0.4526 not yet optimized!
Message in Lattice File: Optimization is as easy as 2 + 2 = 4.
```

The `print` statement is useful to remind someone using the lattice of important details.

2.16 Title Statement

The `title` statement sets a title string which can be used by a program. For consistency with *MAD* there are two possible syntaxes

```
title, <String>
```

or the statement can be split into two lines

```
title
<String>
```

For example

```
title
"This is a title"
```

2.17 Call Statement

It is frequently convenient to separate the lattice definition into several files. Typically there might be a file (or files) that define the layout of the lattice (something that doesn't change often) and a file (or files) that define magnet strengths (something that changes more often). The `call` is used to read in separated lattice files. The syntax is

```
call, filename = <String>
```

Example:

```
call, filename = "../layout/my_layout.bmad"      ! Relative pathname
call, filename = "/nfs/cesr/lat/my_layout.bmad"  ! Absolute pathname
call, filename = "$LATDIR/my_layout.bmad"        ! Absolute pathname
```

Bmad will read the called file until a `return` or `end_file` statement is encountered or the end of the file is reached.

For filenames that have a relative pathname, the called file will be searched for relative to the directory of the calling file. Thus, in the above example, if the file containing the call statements is in the directory `/path/to/lat_dir`, the first call will open the file:

```
/path/to/lat_dir/../layout/my_layout.bmad
```

To call a file relative to the current working directory, use the environment variable `PWD`. Example:

```
call, filename = $PWD/here.bmad
```

Where a called file is searched for may be modified by using a `use_local_lat_file` statement. See Section §2.20 for more details.

An XSIF (§2.1) lattice file may be called from within a *Bmad* lattice file by prepending `"xsif::"` to the file name. Example:

```
call, filename = "xsif::my_lattice.xsif"
```

This statement must be the first statement in the *Bmad* lattice file except for any comments or debugging statements (§2.25). The XSIF lattice file must define a complete lattice and cannot contain any *Bmad* specific statements. The call to the XSIF file automatically expands the lattice (§2.23) and any additional statements in the *Bmad* lattice file operate on the expanded lattice.

2.18 Inline Call

Any lattice elements will have a set of attributes that need to be defined. As a convenience, it is possible to segregate an element attribute or attributes into a separate file and then "call" this file using an "inline call". The inline call has three forms. In an element definition, the inline call has the form

```
<ele_name>: <ele_type>, ..., call::<file_name>, ...
```

or

```
<ele_name>: <ele_type>, ..., <attribute_name> = call::<file_name>, ...
```

where `<attribute_name>` is the name of the attribute and `<file_name>` is the name of the where the attribute structure is given. The third form of the inline call occurs when an element attribute is redefined and has the form

```
<ele_name>[<attribute_name>] = call::<file_name>
```

Example:

```
c: crystal, call::my_curvature.bmad, surface = call::my_surface.bmad, ...
```

Inline calls can be used to call binary files (useful for speeding up lattice parsing). See §2.19 for more details.

2.19 Binary Format

It is possible to store certain element attributes in a binary format file which is useful for speeding up lattice parsing. These binary files must have a `.bin` suffix. The attributes that support binary files are:

```
cartesian_map       ! §4.15.2
cylindrical_map     ! §4.15.3
grid_field          ! §4.15.4
taylor_field        ! §4.15.5
```

The syntax for calling a binary file is the same as an inline call (§2.18). Example:

```
qq: quadrupole, grid_field = call::my_grid.bin, ...
```

To create a binary file or files, first create a lattice with the attributes defined with plain text. Next read the lattice into any program that can create *Bmad* lattice files (for example, the *Tao* program (§1.6) can do this) and have the program generate a lattice file.

2.20 Use_local_lat_file Statement

It is sometimes convenient to override where *Bmad* looks for called files (see §2.17). For example, suppose it is desired to temporarily override the settings in a called file without modifying the called file itself. In this case, the `use_local_lat_file` statement can be used. When this statement is encountered in a lattice file, the local directory (that is, the directory from which the program is run) is searched first for the called file and if a file of the correct name is found, that file is used.

An example will make this clear. Suppose lattice file `/A/lat.bmad` contains the call:

```
call, filename = "/B/sub.bmad"
```

Now suppose that you want to use `lat.bmad` with a modified `sub.bmad` but you do not want to modify `/A/lat.bmad` or `/B/sub.bmad`. The solution is to create two new files. One file, call it `new.bmad`, which can be situated in any directory, has two lines in it:

```
use_local_lat_file
call, filename = "/A/lat.bmad"
```

The second new file is the modified `sub.bmad` and it must be in the directory from which the program is run.

2.21 Return and End_File Statements

`Return` and `end_file` have identical effect and tell *Bmad* to ignore anything beyond the `return` or `end_file` statement in the file.

2.22 Expand_Lattice Statement

Normally, lattice expansion happens automatically at the end of the parsing of the lattice file but an explicit `expand_lattice` statement in a lattice file will cause immediate expansion. See §2.23 for details. If a second `expand_lattice` statement is encountered after a first `expand_lattice` statement has caused lattice expansion, the second `expand_lattice` does nothing and is ignored.

2.23 Lattice Expansion

At some point in parsing a lattice file, the ordered sequence (or sequences if there are multiple branches) of elements that form a lattice must be constructed. This process is called `lattice expansion` since the element sequence can be built up from sub–sequences (§6). Normally, lattice expansion happens automatically at the end of the parsing of the lattice file (or files) but an explicit `expand_lattice` statement in a lattice file will cause immediate expansion. The reason why lattice expansion may be necessary before the end of the file is due to the fact that some operations need to be done after lattice expansion. This includes:

- The `ran` and `ran_gauss` functions, when used with elements that show up multiple times in a lattice, generally need to be used after lattice expansion. See §2.13.
- Some dependent parameters may be set as if they are independent parameters but only if done before lattice expansion. See §4.1.
- Setting the `phi0_multipass` attribute for an `Lcavity` or `RFcavity` multipass slave may only be done after lattice expansion (§7.2).
- Setting individual element attributes for tagged elements can only be done after lattice expansion (§6.7).

Notice that all `lines` (§6.2), `lists` (§6.5), and `use` (§6.6) statements necessary for lattice expansion must come before an `expand_lattice` statement.

Lattice expansion is only done once so it is an error if multiple `expand_lattice` statements are present.

The steps used for lattice expansion are:

1. Instantiate all of the lines listed in the last `use` statement (§6.6). If an instantiated line has `fork` or `photon_fork` (§3.19) elements, instantiate the lines connected to the fork elements if the `fork` or `photon_fork` is connected to a new branch. Instantiation of a given line involves:
   (a) Line expansion (§6) where the element sequence is constructed from the line and sub-lines.
   (b) Adding any superpositions (§7.1).
2. Form multipass lords and mark the appropriate multipass slaves (§7.2).
3. Add girder control elements (§3.20).

4. Add group (§3.21) and overlay (§3.36) control elements.

A lattice file where all the statements are post lattice expansion valid is called a "`secondary lattice file`". To promote flexibility, *Bmad* has methods for parsing lattices in a two step process: First, a "primary" lattice file that defines the basic lattice is read. After the primary lattice has been parsed and lattice expansion has been done, the second step is to read in one or more secondary lattice files. Such secondary lattice files can be used, for example, to set such things as element misalignments. The point here is that there are no calls (§2.17) of the secondary files in the primary file so the primary lattice file does not have to get modified when different secondary files are to be used.

2.24 Slice_Lattice Statement

The `slice_lattice` statement is used to remove elements from the lattice. The `slice_lattice` is useful when analysis of only part of the lattice is desired and the analysis of the entire lattice can take a significant amount of time.

The `slice_lattice` statement must come after lattice expansion (§2.23). The syntax of the `slice_lattice` statement is

```
slice_lattice <element-list>
```

where `<element-list>` is a list of elements (§2.7). For example

```
expand_lattice           ! Lattice expansion must happen first
slice_lattice q1##2:357
```

In this example, all elements outside of the range from element `q1##2` (the second instance of `q1` in the lattice) to element with index number 357 are discarded. The exception is that lord elements (§1.4) of any elements that remain are also not discarded and the `beginning` element at the start of any branch is also retained.

For any lattice branch where elements are removed, the Twiss parameters and reference energy is computed, and the Twiss parameters and reference energy at the entrance end of the first element that is not removed is transfered to the beginning branch element. The branch `geometry` is also set to `open`.

For a lattice branch with a closed geometry, the Twiss parameters are computed with the RF on. That is, the reference energy at the beginning of the sliced lattice branch will probably be non-zero. This will affect the Twiss calculation. If this is not what is wanted, the RF can be turned off before the `slice_lattice` command which will ensure the reference energy is zero at the beginning of the lattice branch. Example:

```
expand_lattice
rfcavity::*[is_on] = False    ! Turn RF off
slice_lattice q1##2:357       ! Slice the lattice
! This shows how to reset the RF and geometry if needed.
rfcavity::*[is_on] = True     ! Turn RF back on
parameter[geometry] = closed  ! Change the geometry.
... etc ...
```

2.25 Debugging Statements

There are a few statements which can help in debugging the *Bmad* lattice parser itself. That is, these statements are generally only used by programmers. These statements are:

```
debug_marker
no_digested
no_superimpose
parser_debug
write_digested
```

The `debug_marker` statement is used for marking a place in the lattice file where program execution is to be halted. This only works when running a program in conjunction with a program debugging tool.

The `no_digested` statement if present, will prevent *Bmad* from creating a digested file (§2.4. That is, the lattice file will always be parsed when a program is run. The `write_digested` statement will cancel a `no_digested` statement.

The `no_superimpose` statement is used to suppress superpositions (§7.1). This is useful for debugging purposes.

The `parser_debug` statement will cause information about the lattice to be printed out at the terminal. The syntax is

```
parser_debug <switches>
```

Valid `<switches>` are

```
particle_start      ! Print the particle_start information.
const               ! Print table of constants defined in the lattice file.
ele <n1> <n2> ...   ! Print full info on selected elements.
lattice             ! Print a list of lattice element information.
lord                ! Print full information on all lord elements.
seq                 ! Print sequence information.
slave               ! Print full information on all slave elements.
time                ! Print timing information.
```

Here $< n1 >$, $< n2 >$, etc. are the index of the selected elements in the lattice. Example

```
parser_debug var lat ele 34 78
```

Chapter 3

Elements

A lattice is made up of a collection of elements — quadrupoles, bends, etc. This chapter discusses the various types of elements available in *Bmad*.

Element	*Section*	*Element*	*Section*
AB_Multipole	3.1	Match	3.30
AC_Kicker	3.2	Monitor	3.23
BeamBeam	3.3	Multipole	3.32
Beginning_Ele	3.4	Null_Ele	3.34
Converter	3.9	Octupole	3.35
Custom	3.10	Patch	3.37
Drift	3.13	Photon_Fork	3.19
E_Gun	3.14	Pipe	3.23
Ecollimator	3.7	Quadrupole	3.39
ElSeparator	3.15	Rbend	3.5
EM_Field	3.16	Rcollimator	3.7
Fiducial	3.17	RFcavity	3.40
Floor_Shift	3.18	Sad_Mult	3.41
Fork	3.19	Sbend	3.5
HKicker	3.24	Sextupole	3.43
Hybrid	3.22	Sol_Quad	3.45
Instrument	3.23	Solenoid	3.44
Kicker	3.25	Taylor	3.46
Lcavity	3.26	Undulator	3.47
Marker	3.28	VKicker	3.24
Mask	3.29	Wiggler	3.47

Table 3.1: Table of element types suitable for use with charged particles.

Most element types available in *MAD* are provided in *Bmad*. Additionally, *Bmad* provides a number of element types that are not available in *MAD*. A word of caution: In some cases where both *MAD* and *Bmad* provide the same element type, there will be an overlap of the attributes available but the two sets of attributes will not be the same. The list of element types known to *Bmad* is shown in Table 3.1, 3.2, and 3.3. Table 3.1 lists the elements suitable for use with charged particles, Table 3.2 which lists the

elements suitable for use with photons, and finally Table 3.3 lists the `controller` element types that can be used for parameter control of other elements. Note that some element types are suitable for both particle and photon use.

Element	*Section*	*Element*	*Section*
Beginning_Ele	3.4	Marker	3.28
Capillary	3.6	Mask	3.29
Crystal	3.8	Match	3.30
Custom	3.10	Monitor	3.23
Detector	3.11	Mirror	3.31
Diffraction_Plate	3.12	Multilayer_Mirror	3.33
Drift	3.13	Patch	3.37
Ecollimator	3.7	Photon_Fork	3.19
Fiducial	3.17	Photon_Init	3.38
Floor_Shift	3.18	Pipe	3.23
Fork	3.19	Rcollimator	3.7
Instrument	3.23	Sample	3.42
Lens	3.27		

Table 3.2: Table of element types suitable for use with photons.

Element	*Section*	*Element*	*Section*
Group	3.21	Overlay	3.36
Girder	3.20		

Table 3.3: Table of controller elements.

For a listing of element attributes for each type of element, see Chapter §13.

3.1 AB_Multipole

An `ab_multipole` is a thin magnetic multipole lens up to 21^{st} order. The basic difference between this and a `multipole` (§3.32 is the input format. See section §15.1 for how the multipole coefficients are defined.

General `ab_multipole` attributes are:

Attribute Class	§	*Attribute Class*	§
a*n*, b*n* multipoles	4.14	Length	4.12
Aperture limits	4.8	Offsets & tilt	4.6
Chamber wall	4.11	Reference energy	4.5
Custom Attributes	2.9	Superposition	7.1
Description strings	4.3	Tracking & transfer map	5
Is_on	4.13		

See §13.1 for a full list of element attributes.

The length `l` is a fictitious length that is used for synchrotron radiation computations and affects the

longitudinal position of the next element but does not affect any tracking or transfer map calculations. The `x_pitch` and `y_pitch` attributes are not used in tracking.

When an `ab_multipole` is superimposed (§7.1) on a lattice, it is treated as a zero length element and in this case it is an error for the length of the `ab_multipole` to be set to a nonzero value.

Unlike a `multipole`, an `ab_multipole` will *not* affect the reference orbit if there is a dipole component.

Example:
```
abc: ab_multipole, a2 = 0.034e-2, b3 = 5.7, a11 = 5.6e6/2
```

3.2 AC_Kicker

An `ac_kicker` element simulates a time dependent kicker element.

General `ac_kicker` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Is_on	4.13
Chamber wall	4.11	Length	4.12
Custom Attributes	2.9	Mag & Elec multipoles	4.14
Description strings	4.3	Offsets, pitches & tilt	4.6
Field Maps	4.15	Reference energy	4.5
Fringe Fields	4.20	Superposition	7.1
Hkick & Vkick	4.7	Symplectify	5.6
Integration settings	5.4	Tracking & transfer map	5

See §13.2 for a full list of element attributes.

Attributes specific to a `ac_kicker` element are:
```
t_offset        = <Real>                          ! Time offset of field waveform.
ref_time_offset = <Logic>                         ! Relative to the ref time? (Default: True).
interpolation   = <switch>                        ! spline (default) or linear.
amp_vs_time = {(<t1>, <A1>), (<t2>, <A2>), ...}   ! Field amplitude vs Time.
frequencies = {(<f1>, <A1>, <phi1>), (<f2>, <A2>, <phi2>), ...} ! Freq. components.
```
Note: The units of the phases `phi` with the `frequencies` attribute are `radians/2pi`.

An `ac_kicker` element is like a `kicker` element except that the field varies in time. The field is calculated in two steps:

1. Calculate the field the same as for a `kicker` element (§3.25).
2. Scale the field using the function $A(\delta t)$. If `ref_time_offset` is set to `True` (the default), $\delta t = t - t_0 - t_{ref}$ where t is the time, t_0 is the value of the `t_offset` attribute, and t_{ref} is the reference time at the entrance end of the element. If `ref_time_offset` is set to `False`, $\delta t = t - t_0$. Note: Unlike RF cavity elements, the time t here is always the absolute time (§22.1) and not the relative time.

There are two ways to specify the dimensionless time variation $A(\delta t)$ of the field. One way is to specify points on the $A(\delta t)$ curve using the `amp_vs_time` attribute. Example:
```
mk: ac_kicker, l = 0.3, scale_multipoles = F, b1 = 0.27, t_offset = 3.6e-8,
        amp_vs_time = {(-1.2e-6, 0.02), ... }
```

The element in this example is an AC quadrupole kicker. The times (in seconds) must be in ascending order and no two times may be the same. The method used to interpolate between the time points is determined by the setting of the `interpolate` parameter which may be one of

```
linear      ! Linear interpolation.
spline      ! Cubic spline interpolation (default).
```

For times before of the range specified by `amp_vs_time`, the amplitude is taken to be the amplitude of the first point of the range. Similarly, for times after the the range specified by `amp_vs_time`, the amplitude is taken to be the amplitude of the last point of the range.

The second way to specify $A(\delta t)$ is to specify the frequencies in the $A(t)$ spectrum using the `frequencies` attribute:

$$A(t) = \sum_i A_i \cos(2\pi(f_i \delta t + \phi_i)) \tag{3.1}$$

Example:

```
mk: ac_kicker, l = 0.3, field_calc = fieldmap, cartesian_map = {...},
        frequencies = {(3.4e6, 0.34, 0.12), ...}
```

When using a frequency spectrum, the `interpolate` attribute is ignored. Note: The units of the phases `phi` with the `frequencies` attribute are `radians/2pi`.

Note: The calculated field will only obey Maxwell's equations in the limit that the time variation of the field is "slow":

$$\omega \ll \frac{c}{r} \tag{3.2}$$

where ω is the characteristic frequency of the field variation, c is the speed of light, and r is the characteristic size of the `ac_kicker` element. That is, the fields at opposite ends of the element must be able to "communicate" (which happens at the speed of light) in a time scale short compared to the time scale of the change in the field.

3.3 BeamBeam

A `beambeam` element simulates an interaction with an opposing ("strong") beam traveling in the opposite direction. The strong beam is assumed to be Gaussian in shape. In the `bmad_standard` calculation the beam–beam kick is computed using the Bassetti–Erskine complex error function formula[Talman87]

General `beambeam` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Is_on	4.13
Chamber wall	4.11	Offsets, pitches & tilt	4.6
Custom Attributes	2.9	Reference energy	4.5
Description strings	4.3	Superposition	7.1
Is_on	4.13	Tracking & transfer map	5

See §13.3 for a full list of element attributes.

Attributes specific to a `beambeam` element are:

```
sig_x   = <Real>     ! Horizontal strong beam sigma at the center
sig_y   = <Real>     ! Vertical strong beam sigma at the center
sig_z   = <Real>     ! Strong beam length
charge  = <Real>     ! Strong beam charge. Default = -1
```

```
n_slice = <Integer>  ! Number of strong beam slices
beta_a  = <Real>     ! $a$-mode beta Twiss parameter
alpha_a = <Real>     ! $a$-mode alpha Twiss parameter
beta_b  = <Real>     ! $b$-mode beta Twiss parameter
alpha_b = <Real>     ! $b$-mode alpha Twiss parameter
bbi_constant         ! See below. Dependent attribute (§4.1).
```

The number of particles in the strong beam is set by

```
charge * parameter[n_part]
```

`parameter[n_part]` is the nominal number of particles of the strong beam. `parameter[n_part]` is a global setting (§8.1) and is used for all `beambeam` elements. To vary the number of particles in an individual `beambeam` element, the `charge` attribute is used. The default is `charge` = -1 which indicates that the strong beam has the opposite charge of the weak beam.

`sig_z` are the strong beam's longitudinal sigma. The strong beam is divided up into `n_slice` equal charge (not equal thickness) slices. Propagation through the strong beam involves a kick at the charge center of each slice with drifts in between the kicks. The kicks are calculated using the standard Bassetti–Erskine complex error function formula[Talman87]. Even though the strong beam can have a finite `sig_z`, the length of the element is always considered to be zero. This is achieved by adding drifts at either end of any tracking so that the longitudinal starting point and ending point are identical. The longitudinal s–position of the `BeamBeam` element is at the center of the strong bunch. For example, with `n_slice` = 2 the calculation would proceed as follows:

1. Start with the reference particle at the center of the strong bunch.
2. Propagate (drift) backwards to the center of the first slice.
3. Apply the beam–beam kick due to the first slice.
4. Propagate (drift) forwards to the center of the second slice.
5. Apply the beam–beam kick due to the second slice.
6. Propagate (drift) backwards to end up with the reference particle at the center of the strong bunch.

`sig_x`, `sig_y` are the transverse sigmas of the strong beam at s_0 which is the s-position where the `beambeam` element is located. For calculating the sigmas of any given slice, `sig_x` and `sig_y` are extrapolated using the Twiss parameters at s_0. The Twiss parameters at s_0 are set by `beta_a`, `beta_b`, `alpha_a`, and `alpha_b`. If `beta_a` is zero (the default), the a-mode Twiss parameters as calculated from the lattice is used. Similarly, if `beta_b` is zero (the default), the b-mode Twiss parameters as calculated from the lattice is used.

`x_offset`, `y_offset`, and `z_offset` are used to offset the `beambeam` element. Note that in *MAD* the attributes used to offset the strong beam are called `xma` and `yma`. `x_pitch` and `y_pitch` gives the beam–beam interaction a crossing angle. This is the full crossing angle, not the half-angle.

The `bbi_constant` is a measure of the beam–beam interaction strength. It is a dependent variable and is calculated from the equation

$$C_{bbi} = N\, m_e\, r_e / (2\,\pi\,\gamma\,(\sigma_x + \sigma_y)) \tag{3.3}$$

In the linear region, near $x = y = 0$, the beam–beam kick is approximately

$$\begin{aligned} k_x &= -4\,\pi\,x\,C_{bbi}/\sigma_x \\ k_y &= -4\,\pi\,y\,C_{bbi}/\sigma_y \end{aligned} \tag{3.4}$$

The beam–beam tune shift is

$$dQ_x = C_{bbi}\,\beta_x/\sigma_x$$
$$dQ_y = C_{bbi}\,\beta_y/\sigma_y \tag{3.5}$$

Example:

```
parameter[n_part] = 1.34e10     ! Used for all beambeam eles
bbi: beambeam, sig_x = 3e-3, sig_y = 3e-4, x_offset = 0.05
```

3.4 Beginning_Ele

An `beginning_ele` element, named `BEGINNING`, is placed at the beginning of every branch (§1.2) of a lattice to mark the start of the branch. The `beginning_ele` always has element index 0 (§1). The creation of this `beginning_ele` element is automatic and it is not permitted for a lattice file to define any other `beginning_ele` elements.

`beginning_ele` attributes are generally set using either `parameter` (§8.1) or `beginning` (§8.4) statements. In particular the initial energy may be set (§4.5).

3.5 Bends: Rbend and Sbend

`Rbend`s and `sbend`s are dipole bends. The difference is that `rbend` elements use a Cartesian ("rectangular") coordinate system to describe the shape of the magnet while `sbend` elements use a polar ("sector") coordinate system.

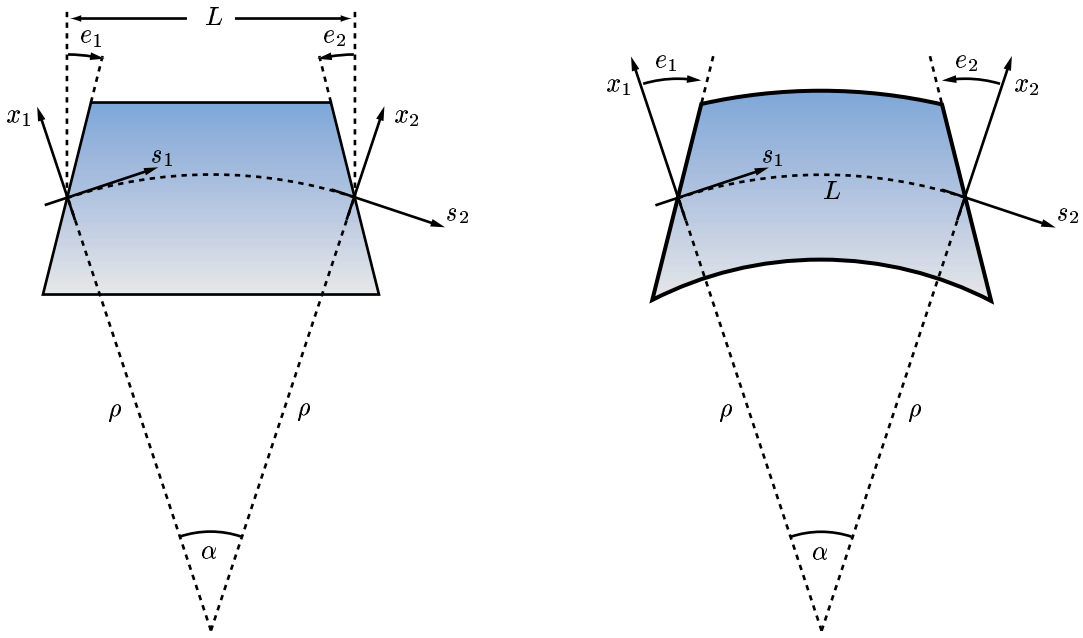
For any given `sbend` element it is possible to construct an equivalent `rbend` element that has the same shape and vice versa. Given this, and to simplify internal bookkeeping, all `rbend` elements are converted to `sbend` elements when a lattice is read in to a program. In order to preserve the information as to whether a bend element was originally specified as an `sbend` or an `rbend` in the lattice file, all bend elements have a `sub_key` parameter which is appropriately set when the lattice is parsed. This `sub_key` parameter does not affect tracking and is only used if a new lattice file is generated by the program.

General `rbend` and `sbend` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Mag & Elec multipoles	4.14
Chamber wall	4.11	Offsets, pitches & tilt	4.6
Custom Attributes	2.9	Overlapping Fields	4.17
Description strings	4.3	Reference energy	4.5
Fringe Fields	4.20	Superposition	7.1
Hkick & Vkick	4.7	Symplectify	5.6
Is_on	4.13	Field Maps	4.15
Integration settings	5.4	Tracking & transfer map	5
Length	4.12		

See §13.4 for a full list of element attributes.

Attributes specific to `rbend` and `sbend` elements are:



(a) rbend (b) sbend

Figure 3.1: Coordinate systems for (a) `rbend` and (b) `sbend` elements. For the bends drawn as viewed from "above" (viewed from positive y), `g`, `angle`, `rho`, `e1` and `e2` are all positive.

```
angle              = <Real>     ! Design bend angle. A settable dependent var (§4.1).
b_field            = <Real>     ! Design field strength (= P_0 g / q) (§4.1).
b_field_err        = <Real>     ! Field strength error (§4.1).
b1_gradient        = <Real>     ! Quadrupole field strength (§4.1).
b2_gradient        = <Real>     ! Sextupole field strength (§4.1).
e1, e2             = <Real>     ! Face angles.
exact_multipoles   = <Switch>   ! Curved coordinate correction? off is default.
fint, fintx        = <Real>     ! Face field integrals.
g                  = <Real>     ! Design bend strength (= 1/rho).
g_err              = <Real>     ! Bend strength error (§4.1).
h1, h2             = <Real>     ! Face curvature.
hgap, hgapx        = <Real>     ! Pole half gap.
k1                 = <Real>     ! Quadrupole strength.
k2                 = <Real>     ! Sextupole strength (§4.1).
l                  = <Real>     ! "Length" of bend. See below.
l_arc              = <Real>     ! Arc length. For rbends only.
l_chord                         ! Chord length. Dependent attribute. See §4.12.
n_ref_pass         = 0 or 1     ! Multipass reference turn (§7.2).
ptc_field_geometry = <Switch>   ! See below.
rho                = <Real>     ! Design bend radius. A settable dependent var (§4.1).
roll               = <Real>     ! See 4.6.
```

angle

The total design bend angle. A positive `angle` represents a bend towards negative x values (see Fig. 14.2).

e1, e2

The rotation angle of the entrance pole face is `e1` and at the exit face it is `e2`. Zero `e1` and `e2` for an `rbend` gives a rectangular magnet (Fig. 3.1a). Zero `e1` and `e2` for an `sbend` gives a wedge shaped magnet (Fig. 3.1b). An `sbend` with an `e1` = `e2` = `angle`/2 is equivalent to an `rbend` with `e1` = `e2` = 0 (see above). This formula holds for both positive and negative angles.

Note: The correspondence between `e1` and `e2` and the corresponding parameters used in the SAD program [SAD] is:

```
e1(Bmad) =  e1(SAD) * angle + ae1(SAD)
e2(Bmad) =  e2(SAD) * angle + ae2(SAD)
```

exact_multipoles

The `exact_multipoles` switch can be set to one of:

```
off                   ! Default
vertically_pure
horizontally_pure
```

This switch determines if the multipole fields, both magnetic and electric, and including the `k1` and `k2` components, are corrected for the finite curvature of the reference orbit in a bend. See §15.3 for a discussion of what `vertically` pure versus `horizontally` pure means. Setting `exact_multipoles` to `vertically_pure` means that the individual a_n and b_n multipole components are used with the vertically pure solutions

$$\mathbf{B} = \sum_{n=0}^{\infty} \left[\frac{a_n}{n+1} \nabla\phi_n^r + \frac{b_n}{n+1} \nabla\phi_n^i \right] \mathbf{E} \qquad = \sum_{n=0}^{\infty} \left[\frac{a_{en}}{n+1} \nabla\phi_n^i + \frac{b_{en}}{n+1} \nabla\phi_n^r \right]$$

and if `exact_multipoles` is set to `horizontally_pure` the horizontally pure solutions ψ_n^r and ψ_n^i are used instead of the vertically pure solutions ϕ_n^r and ϕ_n^i.

To use exact multipoles with PTC based tracking (§5), the PTC exact model tracking must be turned on. That is, in the lattice file set:

```
parameter[ptc_exact_model] = T
```

With exact model tracking, PTC always assumes that multipole coefficients correspond to `horizontally_pure`. In this case, *Bmad* will convert `vertically_pure` to `horizontally_pure` as needed when passing multipole coefficients to PTC. Note that in the case where PTC is doing exact model tracking (§5.4) but the `exact_multipoles` switch is set to `off`, PTC will still be treating the multipoles as `horizontally_pure` even though *Bmad* tracking will be treating them as straight line multipoles. Note: If the bend has an associated electric field, PTC will always be doing exact modeling.

fint, fintx, hgap, hgapx

The field integrals for the entrance and exit pole faces are give by `fint` and `fintx` respectively

$$F_{int} = \int_{pole} ds \, \frac{B_y(s) \left(B_{y0} - B_y(s) \right)}{2 \, H_{gap} \, B_{y0}^2} \tag{3.6}$$

with a similar equation for `fintx`. In the equation B_{y0} is the field in the interior of the dipole and H_{gap} is the pole half gap. The parameters `hgap` and `hgapx` are the half gaps at the entrance and exit faces. If `fint` or `fintx` is given without a value then a value of 0.5 is used. If `fint` or `fintx` is not present, the default value of 0 is used. Note: *MAD* does not have the `fintx` and `hgapx` attributes. *MAD* just assumes that the values are the same for the entrance and exit faces. For compatibility with *MAD*, if `fint` is given but `fintx` is not, then `fintx` is set equal to `fint`. Similarly, `hgapx` will be set to `hgap` if `hgapx` is not given.

Note: The SAD program uses `fb1+f1` for the entrance fringe and `fb2+f1` for the exit fringe. The correspondence between the two is

```
fint  * hgap  = (fb1 + f1) / 12
fintx * hgapx = (fb2 + f1) / 12
```

`fint` and `hgap` can be related to the Enge function which is sometimes used to model the fringe field. The Enge function is of the form

$$B_y(s) = \frac{B_{y0}}{1 + \exp[P(s)]} \tag{3.7}$$

where

$$P(s) = C_0 + C_1\, s + C_2\, s^2 + C_3\, s^3 + \ldots \tag{3.8}$$

The C_0 term simply shifts where the edge of the bend is. If all the C_n are zero except for C_0 and C_1 then

$$C_1 = \frac{1}{2\, H_{gap}\, F_{int}} \tag{3.9}$$

g, g_err, rho
The design bending radius which determines the reference coordinate system is `rho` (see §14.1.1). `g` = 1/`rho` is the curvature function and is proportional to the design dipole magnetic field. The true field strength is given by `g` + `g_err` so changing `g_err` leaves the design orbit unchanged but varies a particle's orbit.

h1, h2
The attributes `h1` and `h2` are the curvature of the entrance and exit pole faces. They are present for compatibility with MAD but are not yet implemented in terms of tracking and other calculations.

k1, b1_gradient
The normalized and unnormalized quadrupole strength.

k2, b2_gradient
The normalized and unnormalized sextupole strength.

l, l_arc, l_chord
For compatibility with MAD, for an `rbend`, `l` is the chord length and not the arc length as it is for an `sbend`. However, after reading in a lattice, *Bmad* will internally convert all `rbend`s into `sbend`s, additionally, the `l_chord` attribute will be set to the input `l`, and `l` will be set to the true path length (see above). Alternatively for an `rbend`, instead of setting `l`, the `l_arc` attribute can be set to the true arc length.

ref_tilt
The `ref_tilt` attribute rotates a bend about the longitudinal axis at the entrance face of the bend. `ref_tilt` = 0 bends the bends the element in the $-y$ direction. See Fig. 14.7. It is important to understand that `ref_tilt`, unlike the `tilt` attribute of other elements, bends both the reference orbit along with the physical element. Note that the MAD `tilt` attribute for bends is equivalent to the *Bmad* `ref_tilt`. Bends in *Bmad* do not have a `tilt` attribute.

The difference between `rbend` and `sbend` elements is the way the `l`, `e1`, and `e2` attributes are interpreted. To ease the bookkeeping burden, after reading in a lattice, *Bmad* will internally convert all `rbend`s into `sbend`s. This is done using the following transformation on `rbend`s:

```
l_chord(internal) = l(input)
l(internal) = 2 * asin(l_chord * g / 2) / g
e1(internal) = e1(input) + theta / 2
e2(internal) = e2(input) + theta / 2
```

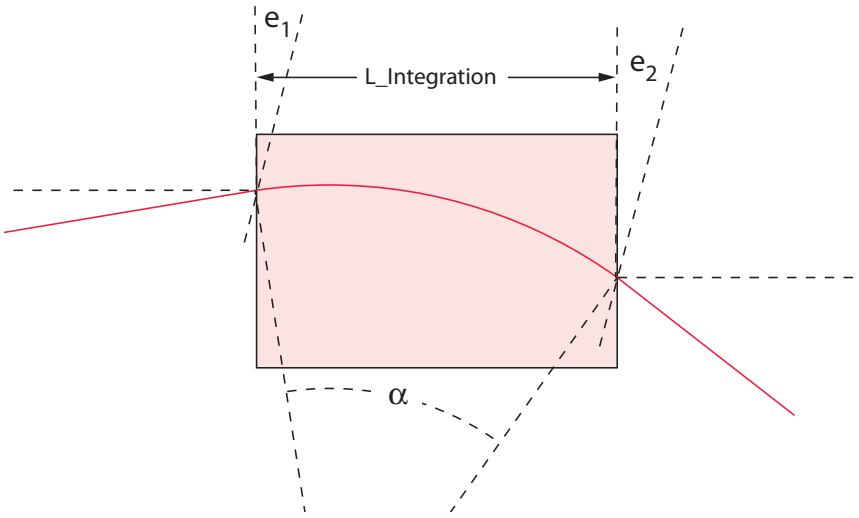


Figure 3.2: Coordinate system when `ptc_field_geometry` is set to `true_rbend`.

The attributes `g`, `angle`, and `l` are mutually dependent. If any two are specified for an element *Bmad* will calculate the appropriate value for the third. After reading in a lattice, `angle` is considered a dependent variable (§4.1).

Since internally all `rbend`s are converted to `sbend`s, if one wants to vary the `g` attribute of a bend and still keep the bend rectangular, an overlay (§3.36) can be constructed to maintain the proper face angles. For example:

```
l_ch = 0.54
g_in = 1.52
l_coef = asin(l_ch * g_in / 2) / g_in
my_bend: rbend, l = l_ch, g = g_in
my_overlay: overlay = {my_bend, my_bend[e1]:l_coef, my_bend[e2]:l_coef},
              var = {g}, g = g_in
```

Notice that `l_coef` is just `arc_length/2`.

The `n_ref_pass` attribute are only used when a bend is part of a `multipass` line and is used to set the reference geometry of the bend. See section §7.2 for more details.

In the local coordinate system (§14.1.1), looking from "above" (bend viewed from positive y), and with `ref_tilt` = 0, a positive `angle` represents a particle rotating clockwise. In this case. `g` will also be positive. For counterclockwise rotation, both `angle` and `g` will be negative but the length `l` is always positive. Also, looking from above, a positive `e1` represents a clockwise rotation of the entrance face and a positive `e2` represents a counterclockwise rotation of the exit face. This is true irregardless of the sign of `angle` and `g`. Also it is always the case that the pole faces will be parallel when

```
e1 + e2 = angle
```

Example bend specification:

```
b03w: sbend, l = 0.6, k1 = 0.003, fint  ! gives fint = fintx = 0.5
```

`ptc_field_geometry` determines how PTC integrates through a bend if PTC is being used for tracking. Possible values for `ptc_field_geometry` are:

```
sector      ! Default
straight
true_rbend  ! Only valid for rbend elements
```

For sector tracking, the tracking coordinate reference frame is with respect to the arc of the reference trajectory. For straight tracking the tracking coordinate reference frame is with respect to the chord line. For a bend where the number of integration steps is large enough, and where there are no other fields besides the basic dipole field, the results are the same. When there are quadrupole or higher order fields, the fields are expanded about the tracking reference frame. Since Maxwell's equations must be satisfied, the higher order fields will differ when tracking with sector vs straight the difference in the fields will scale with the inverse of the bending radius 1/rho. The above discussion is true for ptc_exact_model set to True, for ptc_exact_model set to False, a simplified sector tracking model is used in all cases.

The true_rbend tracking of ptc_field_geometry is used only with rbend elements and the entrance and exit faces must be parallel as shown in Fig. 3.2. That is

```
e1 + e2 = 0
```

In this case, the tracking geometry is parallel, to the bend face as shown in the figure. This can be an advantage in some situations but Étienne Forest discourages use of true_rbend due to complications of how to handle the reference frames. In particular, how to handle the reference frames is problematic when you have more than one of them in a row.

3.6 Capillary

A capillary element is a glass tube that is used to focus x-ray beams.

General capillary attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Offsets, Pitches & Tilt	4.6
Capillary Wall	4.11	Reference energy	4.5
Custom Attributes	2.9	Tracking & transfer map	5
Description strings	4.3		

See §13.5 for a full list of element attributes.

Attributes specific to a capillary element are:

```
critical_angle_factor = <Real>     ! Critical angle * Energy (rad * eV)
```

The critical angle above which photons striking the capillary surface are refracted into the capillary material scales as 1/Energy. The constant of critical angle * energy is given by the critical_angle_factor.

The inside wall of a capillary is defined using the same syntax used to define the chamber wall for other elements (§4.11).

The length of the capillary is a dependent variable and is given by the value of s of the last wall cross-section (§4.11.4).

3.7 Collimators: Ecollimator and Rcollimator

An ecollimator is a drift with elliptic collimation. An rcollimator is a drift with rectangular collimation.

Alternatively, for defining a collimator with an arbitrary shape, a mask element (§3.29) may be used.

General `ecollimator` and `rcollimator` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Offsets, Pitches & Tilt	4.6
Chamber wall	4.11	Overlapping Fields	4.17
Custom Attributes	2.9	Reference energy	4.5
Description strings	4.3	Superposition	7.1
Hkick & Vkick	4.7	Symplectify	5.6
Integration settings	5.4	Field Maps	4.15
Length	4.12	Tracking & transfer map	5

Note: Collimators are the exception to the rule that the aperture is independent of any `tilts`. See §4.8 for more details. Additionally, the default setting of `offset_moves_aperture` is `True` for collimators (§4.8.1).

Example:

```
d21: ecollimator, l = 4.5, x_limit = 0.09/2, y_limit = 0.05/2
```

3.8 Crystal

A `crystal` element represents a crystal used for photon diffraction.

General `crystal` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Surface Properties	4.10
Custom Attributes	2.9	Symplectify	5.6
Description strings	4.3	Offsets, Pitches & Tilt	4.6
Reference energy	4.5	Tracking & transfer map	5

See §13.8 for a full list of element attributes.

Attributes specific to a `crystal` element are:

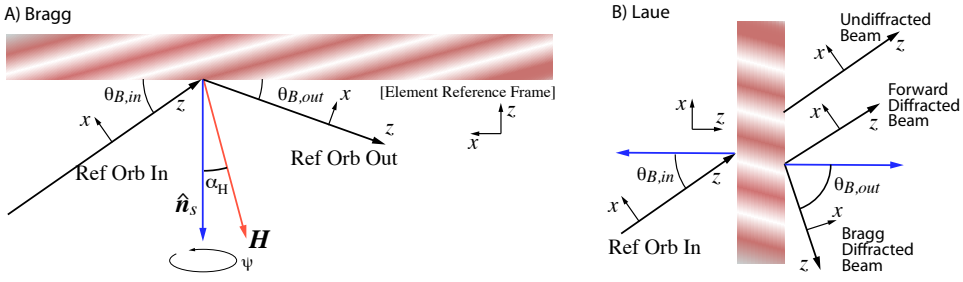


Figure 3.3: Crystal element geometry. A) Geometry for Bragg diffraction. The geometry shown is for `ref_tilt` = 0 (reference trajectory in the x-z plane). The angle α_H (`alpha_angle`) is the angle of the $\mathbf{H}$ vector with respect to the surface normal $\hat{\mathbf{n}}$. For ψ (`psi_angle`) zero, the incoming reference orbit, the outgoing reference orbit, $\hat{\mathbf{n}}$, and $\mathbf{H}$ are all coplanar. B) Geometry for Laue diffraction. In this case there are three outgoing beams: The Bragg diffracted beam, the forward diffracted beam, and the undiffracted beam.

```
b_param            = <Real>       ! b parameter
crystal_type       = <String>     ! Crystal material (§4.9) and reflection plane.
psi_angle          = <Real>       ! Rotation of H-vector about the surface normal.
thickness          = <Real>       ! Thickness of crystal for Laue diffraction.
ref_orbit_follows  = <which_beam> ! Reference orbit aligned with what outgoing beam?
graze_angle_in     = <Real>       ! Angle between incoming ref orbit and surface.
graze_angle_out    = <Real>       ! Angle between outgoing ref orbit and surface.
```

Dependent variables (§4.1) specific to a `crystal` element are:

```
alpha_angle                ! Angle of H-vector with respect to the surface normal.
bragg_angle                ! Nominal Bragg angle at the reference wave length.
bragg_angle_in             ! Incoming grazing angle for Bragg diffraction.
bragg_angle_out            ! Outgoing grazing angle for Bragg diffraction.
d_spacing                  ! Lattice plane spacing.
darwin_width_pi            ! Darwin width for pi polarized light (radians).
darwin_width_sigma         ! Darwin width for sigma polarized light (radians).
dbragg_angle_de            ! Variation of the Bragg angle with energy (radians/eV).
l                          ! Length of reference orbit.
pendellosung_period_pi     ! Pendellosung period for pi polarized light.
pendellosung_period_sigma  ! Pendellosung period for sigma polarized light.
ref_wavelength             ! Reference wavelength (§4.5). Dependent attribute (§4.1).
ref_cap_gamma              ! Γ at the reference wavelength.
tilt_corr                  ! Tilt correction due to a finite psi_angle.
v_unitcell                 ! Unit cell volume.
```

The `crystal_type` attribute defines the crystal material and diffraction lattice plane. The syntax is `"ZZZ(ijk)"` where `ZZZ` is the material name and `ijk` are the Miller indices for the diffraction plane. For example,

```
b_cryst1: crystal, crystal_type = "Si(111)", b_param = -1, ...
```

The atomic formula is case sensitive so, for example, `"SI(111)"` is not acceptable. The list of known crystal materials is given in §4.9. Given the `crystal_type`, the spacing between lattice planes (`d_spacing`), the unit cell volume (`v_unitcell`), and the structure factor[Bater64] values can be computed.

The `b_param` is the standard b asymmetry factor

$$b = \frac{\sin(\alpha_H + \theta_B)}{\sin(\alpha_H - \theta_B)} \tag{3.10}$$

where θ_B is the Bragg angle (`bragg_angle`)

$$\theta_B = \sin^{-1}\left(\frac{\lambda}{2\,d}\right) \tag{3.11}$$

and α_H (`alpha_angle`) is the angle of the reciprocal lattice **H** vector with respect to the surface normal as shown in Fig. 3.3A. If `b_param` is set to -1 then there is Bragg reflection and `alpha_H` is zero. If `b_param` is set to 1 then there is Laue diffraction again with `alpha_H` zero. With the orientation shown in Fig. 3.3A, `alpha_H` is positive.

The `thickness` parameter is used with Laue diffraction only.

The `ref_orbit_follows` parameter sets how the outgoing reference orbit is constructed. This is only relevant with Laue diffraction. The possible settings of this parameter are:

```
bragg_diffracted
forward_diffracted
undiffracted
```

The geometry of this situation is shown in Fig. 3.3B. The reference orbit for the `undiffracted` beam is just a straight line extension of the incoming reference trajectory. This trajectory is that trajectory that photons whose energy is far from the Bragg condition (that is, far from the reference energy) will follow. The `forward_diffracted` reference orbit is parallel to the `undiffracted` trajectory and is the trajectory of the forward diffracted photons whose energy is the reference energy and whose incoming orbit is on the incoming reference trajectory. Finally, the `bragg_diffracted` reference orbit is the backward diffracted orbit.

Note: Changing the setting of `ref_orbit_follows` will change the reference orbit downstream of the crystal which, in turn, will change the placement all downstream elements.

The value of the element reference orbit length `l` is calculated by *Bmad*. `L` will be zero for Bragg diffraction. For Laue diffraction, `l` will depend upon the crystal `thickness` and the setting of `ref_orbit_follows`.

If `psi_angle` is zero, the incoming reference orbit, the outgoing reference orbit, $\widehat{\mathbf{n}}$ and $\mathbf{H}$ are all coplanar. A non-zero `psi_angle` Rotates the $\mathbf{H}$ vector around the $+\widehat{\mathbf{x}}$ axis of the `Element Reference Frame` (See Fig. 3.3A).

To keep the outgoing reference trajectory independent of the value of `psi_angle`, the crystal will be automatically tilted by the appropriate "tilt correction" `tilt_corr`. The calculation of `tilt_corr` is outlined in §23.4.2. `tilt_corr` will be zero if `psi_angle` is zero.

The reference trajectory for a Bragg `crystal` is that of a zero length bend (§14.2.3) and hence the length (`l`) parameter of a crystal is fixed at zero. If the `graze_angle_in` and `graze_angle_out` angles are zero (the default), the orientation of the reference trajectory with respect to the crystal surface is specified by the incoming Bragg angle `bragg_angle_in` ($\theta_{g,in}$) and outgoing Bragg angle `bragg_angle_out` ($\theta_{g,out}$) as shown in Fig. 3.3A. These angles are computed from the photon reference energy and the other crystal parameters such that a photon with the reference energy traveling along the reference trajectory will be in the center of the Darwin curve (§23.4). It is sometimes convenient to be able to specify the angles that the reference trajectory makes with respect to the crystal independent of the Bragg angles. To do this, set `graze_angle_in` and `graze_angle_out` to the desired angles.

Notice that due to refraction at the surface, the computed `bragg_angle` from Eq. (3.11) will deviate slightly from the average of `bragg_angle_in` and `bragg_angle_out`.

The reference trajectory in the global coordinate system (§14.2) is determined by the value of the `ref_tilt` parameter along with the value of `bragg_angle_in` + `bragg_angle_out`. These bragg angles take into account refraction so that the reference trajectory downstream of the crystal will be properly centered with respect to the reference photon. A positive `bragg_angle_in` + `bragg_angle_out` bends the reference trajectory in the same direction as a positive `g` for a bend element. The

A `crystal` may be offset and pitched (4.6). The incoming local reference coordinates are used for these misalignments.

When a crystal is bent (§4.10), the $\mathbf{H}$ vector is assumed follow the surface curvature. That is, it is assumed that the lattice planes are curved by the bending.

Example:

```
crystal_ele: crystal, crystal_type = 'Si(111)', b_param = -1
```

The `darwin_width_sigma` and `darwin_width_pi` parameters are the computed Darwin width, in radians, for sigma and pi polarized light respectively. Here the Darwin width $d\theta_D$ is defined as the width at the $\eta = \pm 1$ points (cf. Batterman[Bater64] Eq (32))

$$d\theta_D = \frac{2\,\Gamma\,|P|\,\mathrm{Re}\big([F_H\,F_{\overline{H}}]^{1/2}\big)}{|b|^{1/2}\,\sin\theta_{tot}} \tag{3.12}$$

where

θ_{tot} = bragg_angle_in + bragg_angle_out

The pendellosung_period_sigma and pendellosung_period_pi are the pendellosung periods for Laue diffraction. If the crystal is set up for Bragg diffraction then the values for these parameters will be set to zero.

The dbragg_angle_de parameter is the variation in Bragg angle with respect to the photon energy and is given by the formula

$$\frac{d\theta_B}{dE} = -\frac{\lambda}{2\,d\,E\,\cos(\theta_B)} \tag{3.13}$$

See Section §11.5 for an example lattice that can be used to simulate a Rowland circle spectrometer.

3.9 Converter

A converter element represents a target onto which particles are slammed in order to generate particles of a different type. For example, a tungsten plate which is bombarded with electrons to generate positrons.

Note: At this point in time, converter elements are experimental and act like marker elements with bmad_standard tracking.

General custom attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Is_on	4.13
Chamber wall	4.11	Length	4.12
Custom Attributes	2.9	Offsets, pitches & tilt	4.6
Description strings	4.3	Reference energy	4.5
Integration settings	5.4	Superposition	7.1
		Tracking & transfer map	5

3.10 Custom

A custom element is an element whose properties are defined outside of the standard *Bmad* subroutine library. That is, to use a custom element, some programmer must write the appropriate custom routines which are then linked with the *Bmad* subroutines into a program. *Bmad* will call the custom routines at the appropriate time to do tracking, transfer matrix calculations, etc. See the programmer who wrote the custom routines for more details! See §33.2 on how to write custom routines.

General custom attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Is_on	4.13
Chamber wall	4.11	Length	4.12
Custom Attributes	2.9	Offsets, pitches & tilt	4.6
Description strings	4.3	Reference energy	4.5
Field Maps	4.15	Superposition	7.1
Fringe fields	4.20	Symplectify	5.6
Integration settings	5.4	Tracking & transfer map	5

See §13.9 for a full list of element attributes.

As an alternative to defining a custom element, standard elements can be "customized" by setting one or more of the following attributes to `custom`:

```
tracking_method       §5.1
mat6_calc_method      §5.2
field_calc            §5.4
aperture_type         §4.8
```

As with a custom element, setting one of these attributes to `custom` necessitates the use of custom code to implement the corresponding calculation.

Attributes specific to a `custom` element are

```
val1, ..., val12 = <Real>  ! Custom values
delta_e_ref      = <Real>  ! Change in energy.
```

`delta_e_ref` is the energy gain of the *reference* particle between the starting edge of the element and the ending edge.

Example:

```
c1: custom, l = 3, val4 = 5.6, val12 = 0.9, descrip = 'params.dat'
```

In this example the `descrip` string is being used to specify a file that contains parameters for the element.

3.11 Detector

A `detector` element is used to detect particles and X-rays. A `detector` is modeled as a grid of pixels which detect particles and x-rays impinging upon them.

General `detector` element attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Offsets, pitches & tilt	4.6
Chamber Wall	4.11	Reference energy	4.5
Custom Attributes	2.9	Superposition	7.1
Description strings	4.3	Tracking & transfer map	5
Detector Geometry	4.10.1		

See §13.10 for a full list of element attributes.

The detector pixels are are arranged in a rectangular grid (§4.10.1).

The `aperture_type` (§4.8) parameter of a `detector` will default to `auto` which will set the aperture limits to define a rectangular aperture that just cover the clear area of the plate.

Example:

```
det: detector, surface = {grid =
        {ix_bounds = (-4,5), iy_bounds = (-10,10), dr = (0.01, 0.01)}}
```

This example defines a detector with 1 cm x 1 cm pixels.

3.12 Diffraction_Plate

A `diffraction_plate` element is a flat surface oriented, more or less, transversely to a x-ray beam through which photon can travel. A `diffraction_plate` can be used, for example, to model a Fresnel

zone plate or Young's double slits. A `diffraction_plate` element is used in places where diffraction effects must be taken into account. This is in contrast to setting an aperture attribute (§4.8 for other elements where diffraction effects are ignored.

A `diffraction_plate` element is similar to a `mask` (§3.29) element except that with a `mask` element coherent effects are ignored. Additionally, a `mask` element can be used with charged particles while a `diffraction_plate` cannot.

General `diffraction_plate` element attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Offsets, pitches & tilt	4.6
Custom Attributes	2.9	Mask geometry	4.11
Description strings	4.3	Reference energy	4.5
Is_on	4.13	Tracking & transfer map	5

See §13.11 for a full list of element attributes.

Attributes specific to a `diffraction_plate` element are:

```
mode               = <Type>   ! Reflection or transmission
field_scale_factor = <Real>   ! Factor to scale the photon field
ref_wavelength                ! Reference wavelength (§4.5). Dependent attribute (§4.1).
```

The `mode` switch sets whether X-rays are transmitted through the `diffraction_plate` or or reflected. Possible values for the `mode` switch are:

```
reflection
transmission        ! Default
```

The geometry of the plate, that is, where the openings (in transmission mode) or reflection regions are, is defined using the "wall" attribute. See (§4.11) for more details.

In transmission mode, a `diffraction_plate` is nominally orientated transversely to the beam. Like all other elements, the `diffraction_plate` can be reoriented using the element's offsets, pitches and tilt attributes (§4.6).

The `aperture_type` (§4.8) parameter of a `diffraction_plate` will default to `auto` which will set the aperture limits to define a rectangular aperture that just cover the clear area of the plate.

The `field_scale_factor`, if set to a non-zero value (zero is the default) will be used to scale the field of photons as they pass through the `diffraction_plate` element:

```
field -> field * field_scale_factor
```

Scaling is useful since the electric field of photons traveling through a `diffraction_plate` are renormalized (see Eqs. (23.10) and (23.11)). This can lead to large variation of the photon field and can, for example, make visual interpretation of plots of field verses longitudinal position difficult to interpret. `field_scale_factor` can be used to keep the field more or less constant.

A `diffraction_plate` that is "turned off" (`is_on` attribute set to False), does not diffract at all and transmits through all the light incident on it.

Example:

```
fresnel: diffraction_plate, wall = {...}
```

3.13 Drift

A `drift` element is a space free and clear of any fields.

General drift attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Offsets, pitches & tilt	4.6
Custom Attributes	2.9	Reference energy	4.5
Description strings	4.3	Symplectify	5.6
Length	4.12	Tracking & transfer map	5

See §13.12 for a full list of element attributes.

Example:

```
d21: drift, l = 4.5
```

Note: If a chamber wall (§4.11) is needed for a field free space, use a pipe element instead of a drift [a wall for a drift is not allowed due to the way drifts are treated with superposition. That is, drifts "disappear" when superimposed upon. (§7.1)].

3.14 E_Gun

An e_gun element represents an electron gun and encompasses a region starting from the cathode were the electrons are generated. General e_gun attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Length	4.12
Chamber wall	4.11	Mag & Elec multipoles	4.14
Custom attributes	2.9	Offsets, pitches & tilt	4.6
Description strings	4.3	Overlapping Fields	4.17
Field autoscaling	4.18	Reference energy	4.5
Hkick & Vkick	4.7	Symplectify	5.6
Integration settings	5.4	Field Maps	4.15
Is_on	4.13	Tracking & transfer map	5

See §13.15 for a full list of element attributes.

The attributes specific to an e_gun are

```
gradient       = <Real>    ! Gradient.
gradient_err   = <Real>    ! Gradient error.
phi0           = <Real>    ! Phase (rad/2π) of the reference particle with
                           !   respect to the RF. phi0 = 0 is on crest.
phi0_err       = <Real>    ! Phase error (rad/2π)
rf_frequency   = <Real>    ! Frequency of the RF field.
voltage        = <Real>    ! Voltage. Dependent attribute (§4.1).
voltage_err    = <Real>    ! Voltage error. Dependent attribute (§4.1).
```

The voltage is simply related to the gradient via the element length l:

```
voltage = gradient * l
```

If the voltage is set to a non-zero value, the length l must also be non-zero to keep the gradient finite. A particle with the charge as the reference particle will have a positive energy gain if the voltage and gradient are positive and vice versa.

field_autoscale The voltage and gradient are scaled by field_autoscale and, if there is a finite rf_frequency, the phase of the frequency is shifted by phi0_autoscale as discussed in Section §4.18. Autoscaling can be toggled on/off by using the autoscale_phase and autoscale_amplitude toggles.

An e_gun may either be DC if the rf_frequency component is zero of AC if not. For an AC e_gun, the phase of the e_gun, The phase $\phi_{\rm ref}$ is

$$\phi_{\rm ref} = {\rm phi0} + {\rm phi0\_err} + {\rm phi0\_autoscale}$$

Electrons generated at the cathode can have zero initial momentum and this presents a special problem (§4.5). As a result, the use of e_gun elements are restricted and they can only be used in a "linear" (non-recirculating) lattice branch. Only one e_gun can be present in a lattice branch and, if it is present, it must be, except for possibly marker or null_ele elements, the first element in any branch.

Note: In order to be able to avoid problems with a zero reference momentum at the beginning of the e_gun, the reference momentum and energy associated with an e_gun element is calculated as outlined in Section §4.5. Additionally, the reference momentum at the exit end of the e_gun, that is p0c, must be non-zero. Thus, for example, if p0c is zero at the start of the lattice, the e_gun voltage must be non-zero.

Additionally, in order to be able to avoid problems with a zero reference momentum at the beginning of the e_gun, absolute time tracking (§22.1) is always used in an e_gun element independent of the setting of parameter[absolute_time_tracking] (§8.1).

Note: The default tracking_method (§5.1) setting for an e_gun is time_runge_kutta and the default mat6_calc_method is tracking.

In this example the field of an e_gun is given by a grid of field values (§4.15.4):

```
apex: e_gun, l = 0.23, field_calc = fieldmap, rf_frequency = 187e6,
              grid_field = call::apex_gun_grid.bmad
```

with the file apex_gun_grid.bmad being:

```
{
  m = 0, harmonic = 1,
  master_scale = voltage,
  geometry = rotationally_symmetric_rz,
  r0 = (0, 0),
  dr = (0.001, 0.001),
  pt(0,0) = ( (0, 0), (0, 0), (1, 0),  (0, 0), (0, 0), (0, 0)),
  pt(0,1) = ( (0, 0), (0, 0), (0.99, 0),  (0, 0), (0, 0), (0, 0)),
  ... }
```

3.15 ELseparator

An elseparator is an electrostatic separator.

General elseparator attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Mag & Elec multipoles	4.14
Chamber wall	4.11	Offsets, pitches & tilt	4.6
Custom Attributes	2.9	Overlapping Fields	4.17
Description strings	4.3	Reference energy	4.5
Fringe Fields	4.20	Superposition	7.1
Hkick & Vkick	4.7	Symplectify	5.6
Integration settings	5.4	Field Maps	4.15
Is_on	4.13	Tracking & transfer map	5
Length	4.12		

See §13.13 for a full list of element attributes.

Attributes specific to an elseparator element are:

```
gap = <Real> ! Distance between electrodes
voltage      ! Voltage between electrodes. This is a settable dependent variable (§4.1).
e_field      ! Electric field. This is a settable dependent variable (§4.1).
```

For an elseparator, the kick for a positively charged particle, with the magnitude of the charge that is the same as the reference particle (set by parameter[particle] §8.1), is determined by hkick and vkick. The kick for a negatively charged particle is opposite this. The gap for an Elseparator is used to compute the voltage for a given kick

```
e_field (V/m) = sqrt(hkick^2 + vkick^2) * P0 * c_light / L
voltage (V) = e_field * gap
```

Example:

```
h_sep: elsep, l = 4.5, hkick = 0.003, gap = 0.11
```

3.16 EM_Field

An em_field element can contain general electro-magnetic (EM) fields. Both AC and DC fields are accommodated. General em_field attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Is_on	4.13
Chamber wall	4.11	Length	4.12
Custom Attributes	2.9	Offsets, pitches & tilt	4.6
Description strings	4.3	Reference energy	4.5
Field Maps	4.15	Superposition	7.1
Hkick & Vkick	4.7	Symplectify	5.6
Integration settings	5.4	Tracking & transfer map	5

See §13.14 for a full list of element attributes.

Attributes specific to an em_field element are:

```
constant_ref_energy = <Logical> ! Does the element have a constant reference energy? Default = True.
```

If the constant_ref_energy logical is set to True (the default), the reference energy (§14.4.1) at the exit end of the element is set equal to the entrance end reference energy. This is the same behavior for most other elements. If the constant_ref_energy logical is set to False, the reference energy at the exit end is calculated like it is in a lcavity or e_gun element.

Note: em_field elements will be created when elements are superimposed (§7.1) and there is no other suitable element class.

3.17 Fiducial

A `fiducial` element is used to fix the position and orientation of the reference orbit within the global coordinate system at the location of the `fiducial` element. A `fiducial` element will affect the global floor coordinates (§14.2) of elements both upstream and downstream of the fiducial element.

Other elements that are used to shift the lattice in the global coordinate frame are `floor_shift` (§3.18) and `patch` (§3.37).

General `fiducial` element attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Reference energy	4.5
Custom Attributes	2.9	Superposition	7.1
Description strings	4.3	Tracking & transfer map	5

See §13.16 for a full list of element attributes.

Attributes specific to a `fiducial` elements are:

```
origin_ele        = <Name>     ! Reference element.
origin_ele_ref_pt = <location> ! Reference pt on reference ele.
dx_origin         = <Real>     ! x-position offset
dy_origin         = <Real>     ! y-position offset
dz_origin         = <Real>     ! z-position offset
dtheta_origin     = <Real>     ! orientation angle offset.
dphi_origin       = <Real>     ! orientation angle offset.
dpsi_origin       = <Real>     ! orientation angle offset.
```

For tracking purposes, the `fiducial` element is considered to be a zero length marker. That is, the transfer map through a `fiducial` element is the unit map.

A `fiducial` element sets the global floor coordinates (§14.2) of itself and of the elements, both upstream and downstream, around it. This can be thought of as a two step process. The first step is to determine the global coordinates of the `fiducial` element itself, and the second step is to shift the coordinates of the elements around it. That is, shifting the position of a `fiducial` element shifts the lattice elements around it as one solid body.

The floor coordinates of the `fiducial` element are determined starting with an `origin_ele` element. If `origin_ele` is not specified, the origin of the global coordinates (§14.2 is used. If the `origin_ele` has a finite length, the reference point may be chosen using the `origin_ele_ref_pt` attribute which may be set to one of

```
entrance_end
center          ! Default
exit_end
```

Once the origin reference position is determined, the reference position of the `fiducial` element is calculated using the offset attributes

```
[dx_origin,  dy_origin,  dz_origin]
[dtheta_origin,  dphi_origin,  dpsi_origin]
```

The transformation between origin and fiducial positions is given in §14.2.4.

Once the position of the `fiducial` element is calculated, all elements of the lattice branch the `fiducial` element is contained in, *both* the upstream and downstream elements, are shifted so that everything

is consistent. That is, the `fiducial` element orients the entire lattice branch. The exception here is that if there are `flexible patch` elements (§3.37) in the lattice branch, the `fiducial` element will only determine the positions up to the `flexible patch` element.

Example: A lattice branch with elements 0 through 103 has a `fiducial` element at position 34 and a `flexible patch` at position 67. In this case the `fiducial` element will determine the reference orbit for elements 0 through 66.

Rules:

- If an `origin_ele` is specified, the position of this element must to calculated before the position of the `fiducial` element is calculated (§14.1.1). This means, the `origin_ele` must be in a prior lattice branch from the branch the `fiducial` element is in or the `origin_ele` in the same branch as the `fiducial` element but is positioned upstream from the `fiducial` element and there is a `flexible patch` in between the two elements.
- If a `fiducial` element affects the position of element 0 in the lattice branch (that is, there are no flexible `patch` elements in between), any positioning of element 0 via `beginning` or `line parameter` statements (§8.4) are ignored.
- `Fiducial` elements must not over constrain the lattice geometry. For example, two `fiducial` elements may not appear in the same lattice branch unless separated by a `flexible patch`.

  Another example is that if there are no flexible `patch` elements in the lattice, and if branch `A` has a `branch` element connecting to branch `B`, the geometry of branch `A` will be calculated first and the geometry of branch `B` can then be calculated from the known coordinates of the `fork` element. If branch `B` contains a `fiducial` element then this is an error since the coordinate calculation never backtracks to recalculate the coordinates of the elements of a branch once the calculation has finished with that branch.

Example:

```
f1: fiducial, origin_ele = mark1, x_offset = 0.04
```

See §11.4 for an example where a fiducial element is used to position the second ring in a dual ring colliding beam machine.

3.18 Floor_Shift

A `floor_shift` element shifts the reference orbit in the global coordinate system without affecting particle tracking. That is, in terms of tracking, a `floor_shift` element is equivalent to a `marker` (§3.28) element.

Also see `patch` (§3.37) and `fiducial` (§3.17) elements.

General `floor_shift` element attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Reference energy	4.5
Custom Attributes	2.9	Superposition	7.1
Description strings	4.3	Tracking & transfer map	5
Length	4.12		

See §13.17 for a full list of element attributes.

Attributes specific to a `floor_shift` elements are:

```
  l                = <Real>     ! Length
  x_offset         = <Real>     ! x offset from origin point.
  y_offset         = <Real>     ! y offset from origin point.
  z_offset         = <Real>     ! z offset from origin point.
  x_pitch          = <Real>     ! rotation of the reference coords.
  y_pitch          = <Real>     ! rotation of the reference coords.
  tilt             = <Real>     ! rotation of the reference coords.
  origin_ele       = <Name>     ! Reference element.
  origin_ele_ref_pt = <location> ! Reference pt on the reference ele.
```

The `floor_shift` element sets the reference orbit at the exit end of the `floor_shift` element as follows: Start with the reference orbit at the `origin_ele` reference point (see below). This coordinate system is shifted using the offset, pitch and tilt parameters of the `floor_shift` element. The shifted coordinate system is used as the coordinate system at the exit end of the `floor_shift` element. The reference position transformation through a `floor_shift` element is given in Section §14.2.4. In this respect, the `floor_shift` element is similar to the `fiducial` element. The difference being that the `fiducial` element affects the global floor coordinates of elements both upstream and downstream of the `fiducial` element while a `floor_shift` element only affects the floor position of elements downstream from it.

Like a `fiducial` element, the transfer map through a `floor_shift` element will be the unit map. That is, the phase space coordinates of a particle will not change when tracking through a `floor_shift` element.

The `l` attribute can be used to adjust the longitudinal s position.

The `floor_shift` element can be used, for example, to restore the correct global geometry when a section of the lattice is represented by, say, a `taylor` type element.

If an `origin_ele` is not specified, the default `origin_ele` is the lattice element before the `floor_shift` element. If an `origin_ele` is specified, *Bmad* needs to be able to calculate the position of this element before the position of the `fiducial` element is calculated. See the discussion of the `origin_ele` for `fiducial` elements (§3.17). Notice that if the `origin_ele` is specified, and is different from the element upstream from the `floor_shift` element, the coordinates at the exit end of the `floor_shift` element is independent of the coordinates of the upstream element.

If the `origin_ele` has a finite length, the reference point may be chosen using the `origin_ele_ref_pt` attribute which may be set to one of

```
  entrance_end
  center
  exit_end         ! Default
```

PTC does not have an analogous element for the `Floor_shift` element. When converting to PTC, a `floor_shift` element will be treated as a `marker` element.

Example:

```
  floor: floor_shift, z_offset = 3.2
```

This offsets the element after the `floor_shift` 3.2 meters from the previous element.

3.19 Fork and Photon_Fork

A `fork` or `photon_fork` element marks the start of an alternative `branch` for the beam (or X-rays or other particles generated by the beam) to follow.

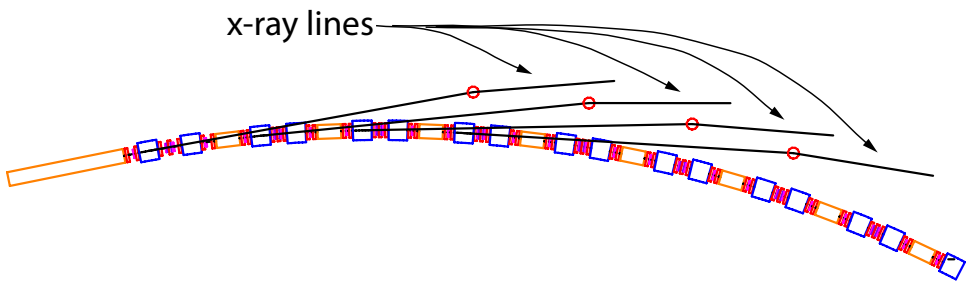


Figure 3.4: Example use of `photon_fork` elements showing four X-ray lines (branches) attached to a machine.

Collectively `fork` and `photon_fork` elements are called forking elements. An example geometry is shown in Fig. 3.4. The `branch` containing a forking element is called the "`base branch`". The `branch` that the forking element points to is called the "`target branch`".

The only difference between `fork` and `photon_fork` is that the default particle type for the target branch forked from a `fork` element is the same particle type as the base branch. The default particle type for the target branch from a `photon_fork` element is a photon. The actual particle associated with a branch can be set by setting the `particle` attribute of the forking element.

General `fork` and `photon_fork` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Length	4.12
Chamber wall	4.11	Reference energy	4.5
Custom Attributes	2.9	Superposition	7.1
Description strings	4.3	Tracking & transfer map	5
Is_on	4.13		

See §13.18 for a full list of element attributes.

Attributes specific to `fork` and `photon_fork` elements are:

```
direction    = <+/- 1>      ! Particles are entering or leaving?
to_line      = <LineName>   ! What line to fork to.
to_element   = <ElementID>  ! What element to attach to in the line being forked to.
new_branch   = <T/F>        ! Make a new branch from the to_line? Default = True.
```

Branch lines can themselves have forking elements. A branch line always starts out tangential to the line it is branching from. A `patch` element (§3.37) can be used to reorient the reference orbit as needed. Example:

```
from_line: line = (... A, PB, B, ...)  ! Defines base branch
pb: photon_fork, to_line = x_line
x_line: line = (X_PATCH, X1, X2, ...)          ! Defines target branch
x_patch: patch, x_offset = 0.01
use, from_line
```

In this example, a photon generated at the fork element `PB` with $x = 0$ with respect to the `from_line` reference orbit through `PB` will, when transferred to the `x_line`, and propagated through `X_PATCH`, have an initial value for x of -0.01.

Forking elements have zero length and, like `marker` elements, the position of a particle tracked through a forking element does not change.

Forking elements do not have orientational attributes like `x_pitch` and `tilt` (4.6). If the orientation of the target branch needs to be modified, this can be accomplished using a `patch` element at the beginning of the line.

The `is_on` attribute, while provided for use by a program, is ignored by *Bmad* proper.

If the reference orbit needs to be shifted when forking from one ring to another ring, a patch can be placed in a separate "transfer" line to isolate it from the branches defining the rings. Example:

```
ring1: line = (... A, F1, B, ...)      ! First ring
x_line: line = (X_F1, X_PATCH, X_F2)   ! "Transfer" line
ring2: line = (... C, F2, D, ...)      ! Second ring
use, ring1

f1: fork, to_line = x_line
f2: fork, to_line = x_line, direction = -1
x_patch: patch, x_offset = ...
x_f1: fork, to_line = ring1, to_element = f1, direction = -1
x_f2: fork, to_line = ring2, to_element = f2
```

Here the `fork` `F1` in `ring1` forks to `x_line` which in turn forks to `ring2`.

The above example also illustrates how to connect machines for particles going in the reverse direction. In this case, rather than using a single `fork` element to connect lines, pairs of `fork` elements are used. `Ring2` has a `fork` element `f2` that points back through `x_line` and then to `ring1` via the `x_f1` fork. Notice that both `f2` and `x_f2` have their `direction` attribute set to -1 to indicate that the fork is appropriate for particles propagating in the $-s$ direction. Additionally, since `f2` has `direction` set to -1, it will, by default, connect to the downstream end of the `x_line`. The default setting of `direction` is 1.

It is important to note that the setting of `direction` does not change the placement of elements in the forked line. That is, the global position (§14.2) of any element is unaffected by the setting of `direction`. To shift the global position of a forked line, `patch` elements must be used. In fact, the `direction` parameter is merely an indicator to a program on how to treat particle propagation. The `direction` parameter is not used in any calculation done by *Bmad*.

The `to_element` attribute for a forking element is used to designate the element of the target branch that the forking element connects to. To keep things conceptually simple, the `to_element` must be a "marker-like" element which has zero length and unit transfer matrix. Possible `to_element` types are:

```
beginning_ele
fiducial
fork and photon_fork
marker
```

When the `to_element` is not specified, the default is to connect to the beginning of the target branch if `direction` is 1 and to connect to the end of the target branch if `direction` is -1. In this case, there is never a problem connecting to the beginning of the target branch since all branches have a `beginning_ele` element at the beginning. When connecting to the end of the target branch the last element in the target branch must be a marker-like element. Note that, by default, a marker element is placed at the end of all branches (§6.1)

The reference energy of a target branch line, needs to be set using line parameter statements (§8.4). The default reference particle type of a branch line will be a `photon` is using a `photon_fork` or will be the same type of particle as the base branch if a `fork` element is used.

Example showing an injection line branching to a ring which, in turn, branches to two x-ray lines:

```
inj: line = (..., br_ele, ...)          ! Define the injection line
use, inj                                ! Injection line is the root
```

```
br_ele: fork, to_line = ring              ! Fork element to ring
ring: line = (..., x_br, ..., x_br, ...)  ! Define the ring
x_br: photon_fork, to_line = x_line       ! Fork element to x-ray line
x_line: line = (...)                      ! Define the x-ray line
x_line[E_tot] = 1e3
```

The `new_branch` attribute is, by default, `True` which means that the lattice branch created out of the `to_line` line is distinct from other lattice branches of the same name. Thus, in the above example, the two lattice branches made from the `x_line` will be distinct. If `new_branch` is set to `False`, a new lattice branch will not be created if a lattice branch created from the same line already exists. This is useful, for example, when a chicane line branches off from the main line and then branches back to it.

When a lattice is expanded (§2.23), the branches defined by the `use` statement (§6.6) are searched for fork elements that branch to new target branches. If found, the appropriate branches are instantiated and the process repeated until there are no more branches to be instantiated. This process does *not* go in reverse. That is the lines defined in a lattice file are not searched for fork elements that have target instantiated branches. For example, if, in the above example, the use statement was:

```
use, x_line
```

then only the `x_line` would be instantiated and the lines `inj` and `ring` would be ignored.

How to analyze a lattice with multiple branches can be somewhat complex and will vary from program to program. For example, some programs will simply ignore everything except the root branch. Hopefully any program documentation will clarify the matter.

3.20 Girder

A `girder` is a support structure that orients the elements that are attached to it in space. A girder can be used to simulate any rigid support structure and there are no restrictions on how the lattice elements that are supported are oriented with respect to one another. Thus, for example, optical tables can be simulated.

General `girder` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Custom Attributes	2.9	Length	4.12
Description strings	4.3	Offsets, pitches & tilt	4.6
Is_on	4.13		

See §13.19 for a full list of element attributes.

Attributes specific to a `girder` are:

```
girder = {<List>}   ! List of elements on the Girder
origin_ele        = <Name>     ! Reference element.
origin_ele_ref_pt = <location> ! Reference pt on reference ele.
dx_origin         = <Real>     ! x-position offset
dy_origin         = <Real>     ! y-position offset
dz_origin         = <Real>     ! z-position offset
dtheta_origin     = <Real>     ! orientation angle offset.
dphi_origin       = <Real>     ! orientation angle offset.
dpsi_origin       = <Real>     ! orientation angle offset.
l                 ! Girder "Length" (4.12). Dependent attribute (§4.1).
```

Attributes specific to a `floor_shift` elements are:

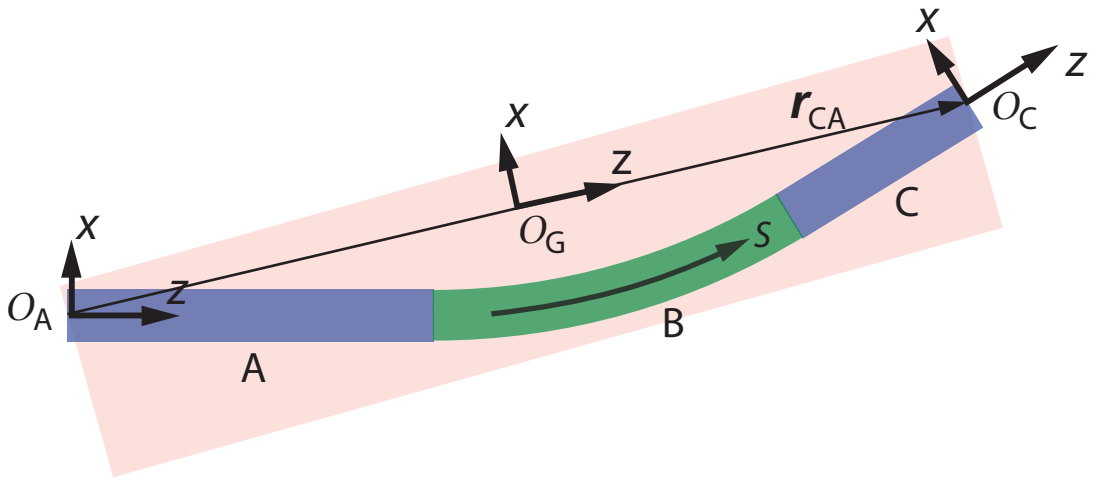


Figure 3.5: Girder supporting three elements labeled `A`, `B`, and `C`. $\mathcal{O}_A$ is the reference frame at the upstream end of element `A` (§14.1.2), $\mathcal{O}_C$ is the reference frame at the downstream end of element `C`, and $\mathcal{O}_G$ is the default `origin` reference frame of the girder. r_{CA} is the vector from $\mathcal{O}_A$ to $\mathcal{O}_C$. The length `l` of the girder is the difference in s between points $\mathcal{O}_C$ and $\mathcal{O}_A$.

A simple example of a girder is shown in Fig. 3.5. Here a girder supports three elements labeled `A`, `B`, and `C` where `B` is a bend so the geometry is nonlinear. Such a girder may specified in the lattice file like:

```
g1: girder = {A, B, C}
```

The `girder` statement can take one of two forms:

```
<element_name>: GIRDER = {<ele1>, <ele2>, ..., <eleN>}, ...
```

or

```
<element_name>: GIRDER = {<ele_start>:<ele_end>}, ...
```

With the first form, a `girder` element will be created for each section of the lattice where there is a "consecutive" sequence of "slave" elements `<ele1>` through `<eleN>`. This section of the lattice from `<ele1>` through `<eleN>` is called the "girder support region". "Consecutive" here means there are no other elements in the girder support region except for possibly `drift` and/or `marker` elements. `Drift` elements cannot be controlled by a girder but may appear in the girder slave list. If a `drift` does appear in the slave list, only `marker` elements, but not `drift` elements will be ignored when determining if elements are consecutive. Note: If a drift-like element is desired to be supported by a `girder`, use a `pipe` element instead. `Marker` elements present in a girder support region, but not mentioned in the girder slave list, are simply ignored.

The second form of a `girder` statement specifies the first and last elements in the sequence of elements to be supported. Everything in between except `drift` elements will be supported by the `girder`.

Wild card characters (§2.8) can be used in any element name in the girder slave list. Additionally, beam line names (§6.2) can be used. In this case, any `drift` elements within a beam line will be ignored.

A lattice element may have at most one `girder` supporting it. However, a `girder` can be supported by another `girder` which in turn can be supported by a third `girder`, etc. Girders that support other girders must be defined in the lattice file after the supported girders are defined. Example:

```
g1: girder = {A, B, C}
g2: girder = {g1}      ! g2 must come after g1!
```

A `girder` may not directly support `multipass_slave` (§7.2) or `super_slave` (§7.1) elements. Rather, a `girder` may support the corresponding lord elements.

The reference frame from which the girder's offset, pitch, and tilt attributes (§4.6) are measured is constructed as follows: A reference frame, called the "`origin`" reference frame may be defined using the attributes `origin_ele` and `origin_ele_ref_pt` which constructs the girder's `origin` frame to be coincident with the reference frame of another element. Example:

```
g2: girder = {...}, origin_ele = Q, origin_ele_ref_pt = entrance_end
```

In this example, girder `g2` has an `origin` reference frame coincident with the entrance end frame of an element named `Q`. Valid values for `origin_ele_ref_pt` are

```
entrance_end
center        ! Default
exit_end
```

For `crystal`, `mirror`, and `multilayer_mirror` elements, setting `origin_ele_ref_pt` to `center` results in the reference frame being the frame of the surface (cf. Fig. 4.6).

To specify that the global coordinates (§14.2) are to be used for a girder set `origin_ele` to `global_coordinates`. Typically this is the same as using the `beginning` element (§3.4) as the `origin_ele` except when the `beginning` element is offset or reoriented (§8.4).

If `origin_ele` is not given, the default `origin` frame is used. The default `origin` frame is constructed as follows: Let $\mathcal{O}_A$ be the reference frame of the upstream end of the first element in the list of supported elements. In this example it is the upstream end of element `A` as shown in the figure. Let $\mathcal{O}_C$ be the downstream end of the last element in the list of supported elements. In this example this is the downstream end of element `C`. The origin of the `girder`'s reference frame, marked $\mathcal{O}_G$ in the figure, will be half way along the vector r_{CA} from the origin of $\mathcal{O}_A$ to the origin of $\mathcal{O}_B$. The orientation of $\mathcal{O}_G$ is constructed by rotating the $\mathcal{O}_A$ coordinate system along an axis in $\mathcal{O}_A$'s x-y plane such that $\mathcal{O}_A$'s z axis ends up parallel with r_{CA}. In the example above, the rotation axis will be along $\mathcal{O}_A$'s y-axis.

Once the `origin` reference frame is established, the reference frame of the girder can be offset from the `origin` frame using the parameters

```
dx_origin     dtheta_origin
dy_origin     dphi_origin
dz_origin     dpsi_origin
```

The orientation of the `girder`'s reference frame from the `origin` frame is given in §14.2.4. Example:

```
g3: girder = { ... }, dx_origin = 0.03
```

This offsets girder `g3`'s reference frame 3 cm horizontally from the default `origin` frame. If no offsets are given, the `origin` frame is the same as the girder's reference frame.

The length `l` of a girder, which is not used in any calculations, is a dependent attribute computed by *Bmad* and set equal to the s path length between points $\mathcal{O}_C$ and $\mathcal{O}_A$.

The physical orientation of the girder with respect to it's reference frame is, like other elements, determined by the offset, pitch and tilt orientation attributes as outlined in §4.6 and §14.2.4. When a girder is shifted in space, the elements it supports are also shifted. In this case, the orientation attributes (`x_offset`, `y_pitch`, etc.) give the orientation of the element with respect to the `girder`. The orientation with respect to the local reference coordinates is given by `x_offset_tot`, which are computed from the orientation attributes of the element and the `girder`. An example will make this clear:

```
q1: quad, l = 2
q2: quad, l = 4, x_offset = 0.02, x_pitch = 0.01
d: drift, l = 8
g4: girder = {q1, q2}, x_pitch = 0.002, x_offset = 0.03
this_line: line = (q1, d, q2)
use, this_line
```

In this example, g4 supports quadrupoles q1 and q2. Since the supported elements are colinear, the computation is greatly simplified. The reference frame of g4, which is the default origin frame, is at $s = 7$ meters which is half way between the start of q1 at at $s = 0$ meters and the end of q2) which is at $s = 14$. The reference frames of q1 and q2 are at their centers so the s positions of the reference frames is

```
Element        S_ref   dS_from_g4
q1              1.0     -6.0
g4              7.0      0.0
q2             12.0      5.0
```

Using a small angle approximation to simplify the calculation, the x_pitch of g4 produces an offset at the center of q2 of $0.01 = 0.002 * 5$. This, added to the offsets of g4 and q2, give the total offset of q2 to be $0.06 = 0.01 + 0.03 + 0.02$. The total x_pitch of q2 is $0.022 = 0.02 + 0.001$.

A girder that has its is_on attribute set to False is considered to be unsifted with respect to it's reference frame.

3.21 Group

Group elements are a type of control element (§1.4) used to make variations in the attributes of other elements (called "slave" attributes) during execution of a program. For example, to simulate the action of a control room knob that changes the beam tune in a storage ring, a group element can be used to vary the strength of selected quads in a specified manner. Also see overlay (§3.36) The difference between group and overlay elements is that overlay elements set the values of the attributes directly while group elements make delta changes to attribute values.

General group attributes are:

Attribute Class	Section	*Attribute Class*	Section
Custom Attributes	2.9	Description strings	4.3

See §13.20 for a full list of element attributes.

Attributes specific to a Group element are:

```
var = {<var1>, <var2>, ...}   ! List of variables.
```

This lists the variables that control the slave attributes.

There are two types of group elements: Expression based and spline based. The general syntax for a expression based group element is

```
name: GROUP = {ele1[attrib1]:exp1, ele2[attrib2]:exp2, ...},
              VAR = {var1, var2, ...}, var1 = init_val1, old_var1 = init_val_old1, ...
```

where ele1[attrib1], ele2[attrib2], etc. specify the slave attributes and exp1, exp2, etc. are the arithmetical expressions, that are functions of var1, var2, etc., and are used to determine a value for the slave attributes.

The general syntax for a spline based group element is

```
name: GROUP = {ele1[attrib1]:{y_knot_points1}, ele2[attrib2]:{y_knot_points2}, ...},
              VAR = {var1}, X_KNOT = {x_knot_points},
              var1 = init_val1, old_var1 = init_val_old1, ...
```

See Section §4.4 for a detailed description of this syntax.

Example of a expression based group element:

```
gr1: group = {q[k1]:a+b^2}, var = {a, b}, a = 1, old_a = 2
gr1[old_b] = 2
```

There are two numbers associated with each variable in a group: One number is the value of the variable (also called the "present" value) and the other number is the "old" value. To refer to these old values prepend the string "old_" to the variable name. Thus, in the above example, the old variable values have names `old_a` and `old_b` and these old values can be set in the same manner as the present values.

Example of a `spline` based `group` element:

```
gr2: group = {beginning[e_tot]]:{4e6,...}},
    var = {time}, x_knot = {...}
```

Here the function used to translate from the `group`'s variables to the slave attribute values is a spline interpolation based upon the knot points specified (§4.4).

A `group` element is like an `overlay` element in that a `group` element controls the attribute values of other "slave" elements. The difference is that the value of a slave attribute that is controlled by (one or more) `overlay` elements is uniquely determined by the controlling `overlay` elements. A `group` element, on the other hand, is used to make changes in value. An example will make this clear:

```
gr: group = {q1[k1]:0.1*a^2}, var = {a}, a = 2, old_a = 1
q, quad, k1 = 0.5
```

When a program reads the lattice file, initially the value of `q[k1]` will be 0.5 as set in the definition of `q`. Later, during lattice expansion (§2.23), the group elements are added to the lattice. When the group element `gr` is added, the fact that `old_a` and `a` are different causes the value of `q[k1]` to be modified. The delta value is

```
delta = 0.1*a^2 - 0.1*old_a^2
      = 0.3
```

And this is added to the existing value of 0.5 so that the value of `q[k1]` becomes 0.8. After the value of `q[k1]` has been updated, the value of `old_a` is automatically update to be the present value of `a` so that the value of `q[k1]` will not be further modified.

In general, deltas used to modify slave attributes are computed as the difference between the arithmetic expression evaluated with the present variable values and the arithmetic expression evaluated with the old variable values.

Notice that in a lattice file the value of a slave attribute after the lattice is read in is independent of whether the group is defined before or after elements whose attributes are controlled by the group. This is true since the effect of a `group` element happens when the lattice is expanded, not when parser reads the `group` definition. On the other hand, after the lattice has been read in, if a program varies both a group variable and a slave attribute, the value of the slave attribute will be dependent upon the order of which is modified first. For example, consider a lattice containing:

```
gr: group = {q[k1]:a^2}, var = {a}
q, quad
```

Now if a program first sets `gr[a]` to 0.3 and then sets `q[k1]` to 0.5, the result is that `q[k1]` will have a value of 0.5. That is, the value of `q[k1]` will be independent of `gr[a]`. If the setting is reversed so that `q[k1]` is set first, the value of `q[k1]` will be 0.59. Since the result is order dependent, trying to "simultaneously" vary the attributes of both group variables and slave attributes can lead to unpredictable results. For example, consider lattice "optimization" where a program varies a set of lattice parameters to achieve certain goals (for example, minimum beta at some point in the lattice, etc.). If the list of parameters to be varied contains both group variables and slave attributes, the actual changes to slave attributes may be different from what the user expects when the program varies its list of parameters.

Different `group` elements may control the same slave attribute and a `group` element may control other `group`, `overlay` or `girder` element attributes. However, It does not make sense, and it is not permitted,

for a group element to control the same attribute as an overlay element or for a group element to control a dependent attribute (§4.1). To setup a group element to control the same slave attribute as an overlay, define an intermediate overlay. For example:

```
ov: overlay = {qk1  q2[k1], ...}, var = {a}
q, quad
gr: group = {ov_q[k1]:a^2}, var = {a}  ! New
ov_q: overlay = {q}, var = {k1}        ! New
```

In this example, the overlay ov controls the attribute q[k1] so it is not permitted for q[k1] to be a slave of a group element. To have group control of q[k1], two elements are introduced: the group gr is setup controlling ov_q[k1] and overlay ov_q is an overlay that controls q[k1]. Notice that trying to control ov directly by a group element will not work since ov controls multiple elements.

A group can be used to control an elements position and length using one of the following attributes:

```
accordion_edge  ! Element grows or shrinks symmetrically
start_edge      ! Varies element's upstream edge s-position
end_edge        ! Varies element's downstream edge s-position
s_position      ! Varies element's overall s-position. Constant length.
```

With accordion_edge, start_edge, end_edge, and symmetric_edge the longitudinal position of an elements edges are varied. This is done by appropriate control of the element's length and the lengths of the elements to either side. In all cases the total length of the lattice is kept invariant.

As an example, consider accordion_edge which varies the edges of an element so that the center of the element is fixed but the length varies:

```
gr: group = {Z[accordion_edge]}, var = {offset}
```

A change of, say, 0.1 gr's offset variable moves both edges of element Z by 0.1 meters so that the length of Z changes by 0.2 meters but the center of Z is constant. To keep the total lattice length invariant, the lengths of the elements to either side are decreased by 0.1 meters to keep the total lattice length constant.

```
q10: quad, l = ...
q11: quad, l = ...
d1: drift, l = ...
d2: drift, l = ...
this_line: line = (... d1, q10, d2, q11, ...)
gr2: group = {q10[start_edge]}, var = {a}, a = 0.1
```

The effect of gr2[a] will be to lengthen the length of q10 and shorten the length of d1.

A lattice file may contain lines and lattice elements that are not part of the actual lattice when the lattice is constructed. Group elements where *none* of its slave elements are part of the finished lattice are ignored and are also not part of the finished lattice. It is not permitted to have group elements that have some slave elements that are part of the finished lattice and some slave elements that are not.

If the arithmetical expression used for an group contains an element attribute, care must be taken if that element attribute is changed. This is discussed in §2.12 and §4.4.

3.22 Hybrid

A hybrid element is an element that is formed by a program by concatenating other element together. Hybrid elements are used to reduce the number of elements in a lattice to speed up a simulation. In terms of tracking a hybrid element is essentially the same as a taylor element.

3.23 Instrument, Monitor, and Pipe

Essentially *Bmad* treats `instrument`, `monitor`, and `pipe` elements like a `drift`. There is a difference, however, when superimposing elements (§7.1). For example, a `quadrupole` superimposed on top of a `drift` results in a free `quadrupole` element in the tracking part of the lattice and no lord elements are created. On the other hand, a `quadrupole` superimposed on top of a `monitor` results in a `quadrupole` element in the tracking part of the lattice and this `quadrupole` element will have two lords: A `quadrupole` superposition lord and a `monitor` superposition lord.

General `instrument`, `monitor`, and `pipe` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Is_on	4.13
Chamber wall	4.11	Length	4.12
Custom Attributes	2.9	Offsets, pitches & tilt	4.6
Description strings	4.3	Reference energy	4.5
Hkick & Vkick	4.7	Superposition	7.1
Instrumental variables	4.21	Symplectify	5.6
Integration settings	5.4	Tracking & transfer map	5

See §13.22 for a full list of element attributes.

The `offset`, `pitch`, and `tilt` attributes are not used by any *Bmad* routines. If these attributes are used by a program they are typically used to simulate such things as measurement offsets. The `is_on` attribute is also not used by *Bmad* proper. Example:

```
d21: instrum, l = 4.5
```

3.24 Kickers: Hkicker and Vkicker

An `hkicker` gives a beam a horizontal kick and a `vkicker` gives a beam a vertical kick. Also see the `kicker` (§3.25) element.

General `hkicker vkicker` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Is_on	4.13
Chamber wall	4.11	Length	4.12
Custom Attributes	2.9	Mag & Elec multipoles	4.14
Description strings	4.3	Offsets, pitches & tilt	4.6
Field Maps	4.15	Reference energy	4.5
Fringe Fields	4.20	Superposition	7.1
Hkick & Vkick	4.7	Symplectify	5.6
Integration settings	5.4	Tracking & transfer map	5

See §13.24 for a full list of element attributes.

Note that `hkicker` and `vkicker` elements use the `kick` attribute while a `kicker` uses the `hkick` and `vkick` attributes. Example:

```
h_kick: hkicker, l = 4.5, kick = 0.003
```

3.25 Kicker

A `kicker` can deflect a beam in both planes. Note that a `kicker` uses the `hkick` and `vkick` attributes while `hkicker` and `vkicker` elements use the `kick` attribute. In addition, a `kicker` can apply a displacement to a particle using the `h_displace` and `v_displace` attributes.

General `kicker` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Mag & Elec multipoles	4.14	Length	4.12
Aperture limits	4.8	Offsets, pitches & tilt	4.6
Chamber wall	4.11	Overlapping Fields	4.17
Custom Attributes	2.9	Reference energy	4.5
Description strings	4.3	Superposition	7.1
Fringe Fields	4.20	Symplectify	5.6
Hkick & Vkick	4.7	Field Maps	4.15
Integration settings	5.4	Tracking & transfer map	5
Is_on	4.13		

See §13.23 for a full list of element attributes.

Example:

```
a_kick: kicker, l = 4.5, hkick = 0.003
```

3.26 Lcavity

An `lcavity` is a LINAC accelerating cavity. The main difference between an `rfcavity` and an `lcavity` is that, unlike an `rfcavity`, the reference energy (§14.4.2) through an `lcavity` is not constant.

General `lcavity` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Offsets, pitches & tilt	4.6
Chamber wall	4.11	Overlapping Fields	4.17
Custom Attributes	2.9	Reference energy	4.5
Description strings	4.3	RF Couplers	4.16
Field autoscaling	4.18	Superposition	7.1
Fringe Fields	4.20	Symplectify	5.6
Hkick & Vkick	4.7	Field Maps	4.15
Integration settings	5.4	Tracking & transfer map	5
Is_on	4.13	Wakes	4.19
Length	4.12		

See §13.25 for a full list of element attributes.

The attributes specific to an `lcavity` are

```
cavity_type     = <Switch>  ! Type of cavity.
gradient        = <Real>    ! Accelerating gradient (V/m).
gradient_err    = <Real>    ! Accelerating gradient error (V/m).
```

```
phi0             = <Real>    ! Phase (rad/2π) of the reference particle with
                             !   respect to the RF. phi0 = 0 is on crest.
phi0_multipass   = <Real>    ! Phase with respect to a multipass lord (rad/2π).
phi0_err         = <Real>    ! Phase error (rad/2π)
e_loss           = <Real>    ! Loss parameter for short range wake fields (V/Coul).
rf_frequency     = <Real>    ! Rf frequency (Hz).
voltage                      ! Cavity voltage. Dependent attribute (§4.1).
l_hard_edge      = <Real>    ! Active region length. Dependent attribute (§4.1).
n_cell           = <Int>     ! Number of cavity cells. Default is 1.
longitudinal_mode = <Int>    ! Longitudinal mode. Default is 1. May be 0 or 1.
```

The dependent variable `voltage` attribute can be used in place of `gradient` as discussed in §4.1. `voltage` is a dependent attribute and is defined to be

```
voltage = gradient * L
```

The energy kick felt by a particle, assuming no phase slippage, is

$$\texttt{dE = gradient\_tot * L * cos(2}\pi\texttt{ * (}\phi_t\texttt{ + }\phi_{\texttt{ref}}\texttt{))}$$

where the total gradient is

```
gradient_tot = (gradient + gradient_err) * field_autoscale
```

ϕ_t is the part of the phase due to when the particle arrives at the cavity and depends upon whether `absolute time tracking` or `relative time tracking` is being used as discussed in §22.1. The phase ϕ_{ref} is

$$\phi_{\texttt{ref}}\texttt{ = phi0 + phi0\_multipass + phi0\_err + phi0\_autoscale}$$

`phi0_multipass` is only to be used with multipass to shift the phase of the cavity from pass to pass. See §7.2.

`phi0_autoscale` and `field_autoscale` are calculated by *Bmad*'s auto-scale module. See Section §4.18 for more details. Autoscaling can be toggled on/off by using the `autoscale_phase` and `autoscale_amplitude` toggles.

The energy change of the reference particle is just the energy change for a particle with $z = 0$ and no phase or gradient errors. Thus

$$\texttt{dE(reference) = gradient * L * cos(2}\pi\texttt{ * }\phi_{\texttt{ref}}\texttt{)}$$

The energy kick for a *Bmad* `lcavity` is consistent with MAD. Note: The MAD8 documentation for an `lcavity` has a wrong sign. Essentially the MAD8 documentation gives

$$\texttt{dE = gradient * L * cos(2}\pi\texttt{ * (}\phi_{\texttt{ref}}\texttt{ - phi(z))) ! WRONG}$$

This is incorrect.

When short-range wake fields are being simulated, with `bmad_com%sr_wakes_on = True` (§9.3), the `e_loss` attribute can be used to modify the gradient in order to maintain a constant average energy gain. That is, `e_loss` can be used to simulate the effect of a feedback circuit that attempts to maintain the average energy of the bunch after the element constant. The energy kick is then

```
dE(with wake) = dE + e_loss * n_part * e_charge
```

`n_part` is set using the `parameter` statement (§8.1) and represents the number of particles in a bunch. `e_charge` is the charge on an electron (Table 2.2). Notice that the `e_loss` term is independent of the sign of the charge of the particle.

The `cavity_type` is the type of cavity being simulated. Possible settings are:

```
ptc_standard
standing_wave    ! Default
traveling_wave
```

The cavity_type switch is ignored if a field map is used. With the standing_wave setting, the transverse trajectory through an lcavity is modeled using equations developed by Rosenzweig and Serafini[Rosen94] modified to give the correct phase-space area at non ultra-relativistic energies. See Section §22.10 for more details. Note: The transfer matrix for an lcavity with finite gradient is never symplectic. See §14.4.2. In addition, couplers (§4.16) and HOM wakes (§4.19) can be modeled.

When an element's tracking_method is set to runge_kutta (§5.1), the "active region" over which there is a finite field is n_cell half-wave pillbox resonators where each pillbox has length $\lambda/2$ (§15.8). The default setting for n_cell is 1. The dependent parameter l_hard_edge is set to the length of the active region. The active region should have a length less than the length of the element. If the length of the element is not equal to the active region, the active region is centered in the element and the regions to either side are treated as field free.

When an element's tracking_method is set to runge_kutta, the fields used with field_calc set to bmad_standard is described in Section (§15.8). With cavity_type set to standing_wave, the longitudinal mode is set by the longitudinal_mode parameter. The possible values are 0 or 1 and the default setting is 1.

Note: When an element's tracking_method is set to bmad_standard, settings for n_cell, and longitudinal_mode are ignored.

Example:

```
lwf: lcavity, l = 2.3, rf_frequency = 500e6, voltage = 20e6,
       tracking_method = runge_kutta, n_cell = 3, longitudinal_mode = 1
```

Note: The default bmad_standard *tracking for* lcavity *elements when the velocity β is significantly different from 1 can only be considered as a rough approximation. Indeed, the only accurate way to simulate a cavity in this situation is by integrating through the actual field [Cf. Runge Kutta tracking (§5.1)]*

3.27 Lens

A lens is an element for concentrating or dispersing light rays.

This element is under development...

3.28 Marker

A marker is a zero length element meant to mark a position.

General marker attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Is_on	4.13
Chamber wall	4.11	Offsets & tilt	4.6
Custom Attributes	2.9	Reference energy	4.5
Description strings	4.3	Superposition	7.1
Instrumental variables	4.21	Tracking & transfer map	5

See §13.27 for a full list of element attributes.

Attributes specific to a marker element are:

```
x_ray_line_len = <Real>
```

`x_ray_line_len` is the length of an associated x-ray synchrotron light line measured from the marker element. This is used for machine geometry calculations and is irrelevant for lattice computations.

The `x_offset`, `y_offset` and `tilt` attributes are not used by any *Bmad* routines. Typically, if these attributes are used by a program, they are used to simulate things like BPM offsets. The `is_on` attribute is also not used by *Bmad* proper.

Example:

```
mm: mark, type = "BPM"
```

3.29 Mask

A `mask` element defines an aperture where the mask area can essentially have an arbitrary shape.

For X-ray tracking, a `mask` element is similar to a `diffraction_plate` (§3.12) element except that with a `diffraction_plate` element, coherent effects are taken into account while, with a `mask` element, coherent effects are ignored. Also a `mask` element can be used with charged particles while a `diffraction_plate` cannot.

General `mask` element attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Offsets, pitches & tilt	4.6
Custom Attributes	2.9	Reference energy	4.5
Description strings	4.3	Superposition	7.1
Is_on	4.13	Tracking & transfer map	5
Mask geometry	4.11		

See §13.28 for a full list of element attributes.

Notice that, unlike a `rcollimator` or a `ecollimator`, a `mask` element has zero length.

Attributes specific to a `mask` element are:

```
mode               = <Type>   ! Reflection or transmission.
field_scale_factor = <Real>   ! Factor to scale the photon field
ref_wavelength                ! Reference wavelength (§4.5). Dependent attribute (§4.1).
```

Note: These attributes are only pertinent for photon tracking. Charged particle tracking assumes transmission mode and does not use `field_scale_factor` and `ref_wavelength` attributes.

The `mode` switch, which is only used for photon tracking, sets whether X-rays are transmitted through the `mask` or or reflected. Possible values for the `mode` switch are:

```
reflection
transmission        ! Default
```

The geometry of the mask, that is, where the openings (in transmission mode) or reflection regions are, is defined using the "wall" attribute. See (§4.11) for more details.

In transmission mode, a `mask` is nominally orientated transversely to the beam. Like all other elements, the `mask` can be reoriented using the element's offsets, pitches and tilt attributes (§4.6).

The `aperture_type` (§4.8) parameter of a `mask` will default to `auto` which will set the aperture limits to define a rectangular aperture that just cover the clear area of the mask.

The `field_scale_factor`, if set to a non-zero value (zero is the default) will be used to scale the field of photons as they pass through the `mask` element:

```
field -> field * field_scale_factor
```

Scaling is useful since the electric field of photons traveling through a `mask` are renormalized (see Eqs. (23.10) and (23.11)). This can lead to large variation of the photon field and can, for example, make visual interpretation of plots of field verses longitudinal position difficult to interpret. `field_scale_factor` can be used to keep the field more or less constant.

A `mask` that is "turned off" (`is_on` attribute set to False), does not mask at all and transmits everything.

Example:

```
scrapper: mask, wall = {...}
```

3.30 Match

A `match` element is used to match the Twiss parameters between two points.

General `match` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Length	4.12
Custom Attributes	2.9	Reference energy	4.5
Description strings	4.3	Superposition	7.1
Is_on	4.13		

See §13.29 for a full list of element attributes.

Attributes specific to a `match` element are:

```
beta_a0, beta_b0   = <Real>                          ! Beginning betas
beta_a1, beta_b1   = <Real>                          ! Ending betas
alpha_a0, alpha_b0 = <Real>                          ! Beginning alphas
alpha_a1, alpha_b1 = <Real>                          ! Ending alphas
eta_x0, eta_y0     = <Real>                          ! Beginning etas
eta_x1, eta_y1     = <Real>                          ! Ending etas
etap_x0, etap_y0   = <Real>                          ! Beginning eta'
etap_x1, etap_y1   = <Real>                          ! Ending eta'
c11_mat0, c12_mat0, c21_mat0, c22_mat0 = <Real>      ! Beginning coupling.
c11_mat1, c12_mat1, c21_mat1, c22_mat1 = <Real>      ! Ending coupling.
dphi_a, dphi_b     = <Real>                          ! Phase advances
x0, px0, y0, py0, z0, pz0 = <Real>                   ! Beginning coordinates
x1, px1, y1, py1, z1, pz1 = <Real>                   ! Ending coordinates
delta_time  = <Real>                                 ! Change in time.
match_end = <Logic>                                  ! See below. Default is False.
match_end_orbit = <Logic>                            ! See below. Default is False.
```

The transfer map for a `match` element is a linear transformation with a "kick":

$$r_1 = \mathbf{M}\, r_0 + \mathbf{V} \tag{3.14}$$

where r_1 is the output coordinates, and r_0 are the input coordinates. The matrix $\mathbf{M}$ is the linear part of the map and the vector $\mathbf{V}$ is the zeroth order part of the map.

Nomenclature: The parameters `beta_a0`, `alpha_a0`, etc. of the `match` element are called the beginning (upstream) "design" Twiss parameters. The parameters `beta_a1`, `alpha_a1`, etc. of the `match`

element are called the ending (downstream) "design" Twiss parameters. Similarly, `c11_mat0`, etc. are the beginning components of the **C** coupling matrix (§20.1).

The matrix **M** is calculated such that if (and only if) the actual (computed) Twiss coupling parameters at the beginning `match` element are equal to the beginning design Twiss and coupling parameters, then the computed Twiss and coupling parameters at the end of the `match` element will be the end design Twiss and coupling parameters and the phase advances (in radians) will be `dphi_a` and `dphi_b` (§20.1).

Note: If all input betas and alphas (`beta_a0`, etc.) are zero, and `match_end` is False, then **M** will be the unit map.

The kick term **V** is constructed so that if a particle has coordinates `(x0, px0, y0, py0, z0, pz0)` before the `match` element, the coordinates just after the element will be `(x1, px1, y1, py1, z1, pz1)`. With this, **V** will be:

$$\mathbf{V} = \begin{pmatrix} \text{x1} \\ \text{px1} \\ \text{y1} \\ \text{py1} \\ \text{z1} \\ \text{pz1} \end{pmatrix} - \mathbf{M} \begin{pmatrix} \text{x0} \\ \text{px0} \\ \text{y0} \\ \text{py0} \\ \text{z0} \\ \text{pz0} \end{pmatrix} \tag{3.15}$$

The `delta_time` parameter adds a constant to the particle's time. This will also affect the z phase space coordinate through Eq. (14.28) and the transfer map though the element. If `delta_time` is zero, the transfer map through the element will be the **M** matrix as discussed above. With a finite `delta_time`, the transfer map will be different from **M**. The order of operations, is the effect of `delta_time` is applied first and the linear transformation above is applied afterwards. Since using `match_end` or `match_end_orbit` with a finite `delta_time` can be confusing, such a situation is not allowed. Use two separate `patch` elements if needed.

The length attribute `l` is not used in the transfer matrix calculation. The length `l` is used to compute the time it takes to go through a match element.

Example:

```
mm: match, beta_a0 = 12.5, beta_b0 = 3.4, eta_x0 = 1.0, ...
```

match_end, match_end_input

The default value of `match_end` is False. The `match_end` attribute is used for appropriately setting the beginning design Twiss parameters from within a program. If the `match_end` attribute is set to True, the beginning design Twiss parameters are set to be equal to the actual Twiss parameters from the exit end of the previous element. In this case, the actual Twiss parameters at the end of the match element will be the design Twiss parameters. The `match_end` attribute may only be set to True with `open` lattices (§8.1) since, for a `closed` lattice, it is not possible to calculate the Twiss parameters at the previous element independently of the design end Twiss parameters at the `match` element.

When running a program, if a `match` element initially has it's `match_end` attribute set to True, the *Bmad* bookkeeping routines will ensure that the `match` element's beginning design Twiss parameters are appropriately set as explained above. If `match_end` is now toggled to False by the program, the beginning Twiss attribute values, and hence the transfer matrix for the `match` element, will be frozen. Thereafter, variation of any parameter in the lattice that affects the calculated Twiss parameters through the `match` element will not affect the `match` element's transfer matrix.

When a lattice is read in, the `match_end_input` attribute is set by *Bmad* to be equal to `match_end`. This attribute is not settable by the user and is simply present to tell the user what the setting of `match_end` was in the lattice file.

match_end_orbit, match_end_orbit_input

The match_end_orbit attribute is similar in operation to the match_end attribute. The default value of match_end_orbit is False. When running a program, if match_end_orbit is set to True, when any particle is tracked through the match element, the match element's starting coordinate parameters, (x0, px0, y0, py0, z0, pz0), will be set to the particle's coordinates at the exit end of the previous element. That is, the particle will always have coordinates equal to (x1, px1, y1, py1, z1, pz1) at the end of the match element. If match_end_orbit is now toggled to False by the program, the ending coordinate parameters, and hence the **V** vector, will become fixed. As with the match_end attribute, the match_end_orbit attribute may only be used with open lattices (§8.1).

When a lattice is read in, the match_end_orbit_input attribute is set by *Bmad* to be equal to match_end_orbit. This attribute is not settable by the user and simply present to tell the user what the setting of match_end_orbit was in the lattice file.

A match element that is "turned off" (is_on attribute set to False), is considered to be like a marker element. That is, the orbit and twiss parameters are unchanged when tracking through a match element that is turned off.

3.31 Mirror

A mirror reflects photons.

General mirror attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Reference energy	4.5
Custom Attributes	2.9	Superposition	7.1
Description strings	4.3	Surface Properties	4.10
Offsets, pitches & tilt	4.6	Tracking & transfer map	5

See §13.30 for a full list of element attributes.

Attributes specific to a mirror element are:

```
graze_angle     = <Real>   ! Angle between incoming beam and mirror surface.
critical_angle  = <Real>   ! Critical angle.
ref_wavelength             ! Reference wavelength (§4.5). Dependent attribute (§4.1).
```

The reference trajectory for a mirror is that of a zero length bend (§14.2.3) and hence the length (l) parameter of a mirror is fixed at zero. The reference trajectory is determined by the values of the graze_angle and ref_tilt parameters. A positive graze_angle bends the reference trajectory in the same direction as a positive g for a bend element.

A mirror may be offset and pitched (4.6). The incoming local reference coordinates are used for these misalignments.

3.32 Multipole

A multipole is a thin magnetic multipole lens up to 21^{st} order. The basic difference between this and an ab_multipole is the input format. See section §15.1 for how the multipole coefficients are defined.

General multipole attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Reference energy	4.5
Custom Attributes	2.9	Is_on	4.13
Chamber wall	4.11	Offsets, pitches & tilt	4.6
Description strings	4.3	Tracking & transfer map	5
K*n*L, T*n* multipoles	4.14		

See §13.32 for a full list of element attributes.

The length l is a fictitious length that is used for synchrotron radiation computations and affects the longitudinal position of the next element but does not affect any tracking or transfer map calculations.

When an multipole is superimposed (§7.1) on a lattice, it is treated as a zero length element and in this case it is an error for the length of the multipole to be set to a nonzero value.

Like a *MAD* multipole, a *Bmad* multipole will affect the reference orbit if there is a dipole component. Example:

```
m1: multipole, k1l = 0.034e-2, t1, k3l = 4.5, t3 = 0.31*pi
```

3.33 Multilayer_mirror

A multilayer_mirror is a substrate upon which multiple layers of alternating substances have been deposited. The idea is similar to crystal diffraction: light reflected at each interface constructively interferes with light reflected from other interfaces. The amplified reflection offsets losses due to absorption.

General crystal attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Symplectify	5.6
Custom Attributes	2.9	Offsets, pitches & tilt	4.6
Description strings	4.3	Superposition	7.1
Reference energy	4.5	Tracking & transfer map	5
Surface Properties	4.10		

The attributes specific to a multilayer_mirror are

```
material_type    = <String>  ! Materials in each layer.
d1_thickness     = <Real>    ! Thickness of layer 1
d2_thickness     = <Real>    ! Thickness of layer 2
n_cell           = <Integer> ! Number of cells (= Number of layers / 2)
ref_wavelength               ! Reference wavelength (§4.5). Dependent attribute (§4.1).
```

See §13.31 for a full list of element attributes.

Dependent attributes (§4.1) are

```
graze_angle      ! Angle between incoming beam and mirror surface.
v1_unitcell      ! Unit cell volume for layer 1
v2_unitcell      ! Unit cell volume for layer 2
```

A multilayer_mirror is constructed of a number of "cells". The number of cells is set by n_cell. Each cell consists of two layers of dielectric material. The materials used is given by the material_type attribute. The format for this is

```
material_type = "<material_1>:<material_2>"
```

where <material_1> and <material_2> are the material names for the first and second layers of the cell respectively. The first layer is the bottom layer and the second layer is the top layer of the cell. Material names are case sensitive. So "FE" cannot be used in place of "Fe" A list of materials is given in §4.9 and can include crystal materials or elemental materials.

Example:

```
mm: multilayer_mirror, material_type = 'W:BORON_CARBIDE', n_cell = 100, &
          d1_thickness = 1e-9, d2_thickness = 1.5e-9
```

3.34 Null_Ele

A null_ele is a special type of element. It is like a marker but it has the property that when the lattice is expanded (§6.2) all null_ele elements are removed. The primary use of a null_ele is in computer generated lattices where it can be used to serve as a reference point for element superpositions (§7.1). Another use is to split an element using superposition while avoiding having to add a marker element to the lattice. Example:

```
N: null_ele, superimpose, ref = quadrupole::*
```

This will split all quadrupoles in the lattice in two.

Null_ele elements are not generally useful otherwise.

3.35 Octupole

An octupole is a magnetic element with a cubic field dependence with transverse offset (§15.1). The bmad_standard calculation treats an octupole using a kick–drift–kick model.

General octupole attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Mag & Elec multipoles	4.14
Chamber wall	4.11	Offsets, pitches & tilt	4.6
Custom Attributes	2.9	Overlapping Fields	4.17
Description strings	4.3	Reference energy	4.5
Fringe Fields	4.20	Superposition	7.1
Hkick & Vkick	4.7	Symplectify	5.6
Integration settings	5.4	Field Maps	4.15
Is_on	4.13	Tracking & transfer map	5
Length	4.12		

See §13.33 for a full list of element attributes.

Attributes specific to an octupole element are:

```
k3          = <Real>   ! Octupole strength.
b3_gradient = <Real>   ! Field strength. (§4.1).
```

If the tilt attribute is present without a value then a value of $\pi/8$ is used. Example:

```
oct1: octupole, l = 4.5, k3 = 0.003, tilt ! same as tilt = pi/8
```

3.36 Overlay

Overlay elements are a type of control element (§1.4) used to make variations in the attributes of other elements (called "slave" attributes) while a program is running. For example, to simulate the action of a magnet power supply that controls a string of magnets. Also see group (§3.21) The difference between group and overlay elements is that overlay elements set the values of the attributes directly while group elements make delta changes to attribute values.

General overlay attributes are:

Attribute Class	Section	*Attribute Class*	Section
Custom Attributes	2.9 Description strings	4.3	

See §13.34 for a full list of element attributes.

Attributes specific to a Overlay element are:

```
var = {<var1>, <var2>, ...}   ! List of variables.
```

This lists the variables that control the slave attributes.

There are two types of overlay elements: Expression based and spline based. The general syntax for a expression based overlay element is

```
name: OVERLAY = {ele1[attrib1]:coef1, ele2[attrib2]:coef2, ...},
                                                  default_attrib = init_value
```

where ele1[attrib1], ele2[attrib2], etc. specify the slave attributes and exp1, exp2, etc. are the arithmetical expressions, that are functions of var1, var2, etc., and are used to determine a value for the slave attributes.

The general syntax for a spline based overlay element is

```
name: OVERLAY = {ele1[attrib1]:{y_knot_points1}, ele2[attrib2]:{y_knot_points2}, ...},
            VAR = {var1}, X_KNOT = {x_knot_points}, var1 = init_val1, ...
```

See Section §4.4 for a detailed description of this syntax.

An overlay element is used to control the attributes of other elements. For example:

```
over1: overlay = {a_ele, b_ele:2.0}, var = {hkick}, hkick = 0.003
over2: overlay = {b_ele}, var = {hkick}
over2[hkick] = 0.9
a_ele: quad, hkick = 0.05   ! NO: Cannot control slave attributes of overlays
b_ele: rbend, ...
this_line: line = (... a_ele, ... b_ele, ...)
use, this_line
```

In the example the overlay over1 controls the hkick attribute of the "slave" elements a_ele and b_ele. over2 controls the hkick attribute of just b_ele. over1[hick] has a value of 0.003 and over2[hkick] has been assigned a value of 0.9. Thus:

```
a_ele[hkick] = over1[hkick]
             = 0.003
b_ele[hkick] = over2[hkick] + 2 * over1[hkick]
             = 0.906
```

Overlays completely determine the value of the attributes that are controlled by the overlay. in the above example, the hkick of 0.05 assigned directly to a_ele is overwritten by the overlay action of over1.

The default value for an overlay is 0 so for example

```
over3: overlay = {c_ele}, var = {k1}
```

will make `c_ele[k1]` = 0.

As illustrated above, different `overlay` elements may control the same element attribute. And an `overlay` element may control other `overlay`, `group` or `girder` elements. However, It does not make sense for an `overlay` element to control the same attribute as a `group` element or for an `overlay` element to control a `dependent` attribute (§4.1).

A lattice file may contain lines and lattice elements that are not part of the actual lattice when the lattice is constructed. `Overlay` elements where *none* of its slave elements are part of the finished lattice are ignored and are also not part of the finished lattice. It is not permitted to have `overlay` elements that have some slave elements that are part of the finished lattice and some slave elements that are not.

If the arithmetical expression used for an `overlay` contains an element attribute, care must be taken if that element attribute is changed. This is discussed in §2.12 and §4.4.

3.37 Patch

A `patch` element shifts the reference orbit and time. Also see `floor_shift` (§3.18) and `fiducial` (§3.17) elements.

General `patch` element attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Offsets, pitches & tilt	4.6
Chamber wall	4.11	Reference energy	4.5
Custom Attributes	2.9	Superposition	7.1
Description strings	4.3	Tracking & transfer map	5
Length	4.12		

See §13.35 for a full list of element attributes.

Attributes specific to a `patch` elements are:

```
x_offset        = <Real>    ! Exit face offset from Entrance.
```

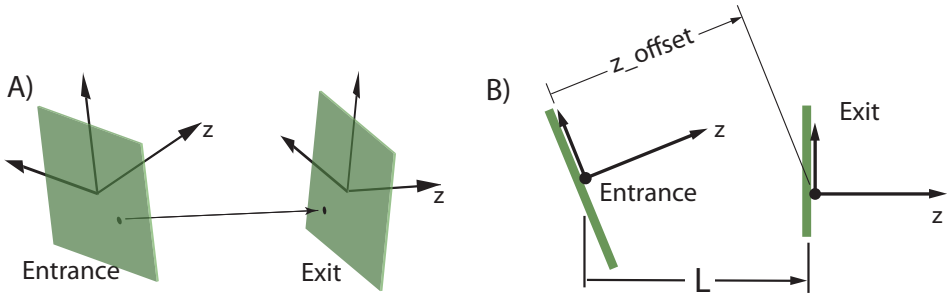


Figure 3.6: A) A `patch` element can align its exit face arbitrarily with respect to its entrance face. B) The reference length of a `patch` element is the longitudinal distance from the entrance origin to the exit origin using the reference coordinates at the exit end. Notice that while the length of the patch is defined with respect to the exit coordinates, the offsets that define the position of the exit end coordinates are defined with respect to the entrance coordinates.

```
y_offset        = <Real>    ! Exit face offset from Entrance.
z_offset        = <Real>    ! Exit face offset from Entrance.
t_offset        = <Real>    ! Reference time offset.
x_pitch         = <Real>    ! Exit face orientation from Entrance.
y_pitch         = <Real>    ! Exit face orientation from Entrance.
tilt            = <Real>    ! Exit face orientation from Entrance.
e_tot_offset    = <Real>    ! Reference energy offset (eV).
e_tot_set       = <Real>    ! Reference energy at exit end (eV).
p0c_set         = <Real>    ! Reference momentum at exit end (eV).
flexible        = <Logic>   ! Default: False.
l               = <Real>    ! Reference length. Dependent attribute (§4.1).
```

A straight line element like a `drift` or a `quadrupole` has the exit face parallel to the entrance face. With a `patch` element, the entrance and exit faces can be arbitrarily oriented with respect to one another as shown in Fig. 3.6A. The length `l` of a `patch` is a dependent (§4.1) parameter and is the longitudinal (z component) distance from the entrance origin to the exit origin using the exit end reference coordinates as shown in Fig. 3.6B. See §14.1.3 for a further discussion on the coordinate system of a `patch`. Notice that while the length of the patch is defined with respect to the exit coordinates, the offsets that define the position of the exit end coordinates are defined with respect to the entrance coordinates.

There are two different ways the orientation of the exit face is determined. Which way is used is determined by the setting of the `flexible` attribute. With the `flexible` attribute set to `False`, the default, The exit face of the `patch` will be determined from the offset, tilt and pitch attributes as described in §14.2.4. This type of `patch` is called "rigid" or "inflexible" since the geometry of the `patch` is solely determined by the `patch`'s attributes and is independent of everything else. Example:

```
pt: patch, z_offset = 3.2   ! Equivalent to a drift
```

With `flexible` set to `True`, the exit face is taken to be the reference frame of the entrance face of the next element in the lattice. In this case, it must be possible to compute the reference coordinates of the next element (or any element downstream of the `patch` as long as the elements in between have zero length) before the reference coordinates of the `patch` are computed. A `flexible patch` will have the its offsets, pitches, and tilt as dependent parameters (§4.1) and these parameters will be computed. Here the `patch` is called "flexible" since the geometry of the patch will depend upon the geometry of the rest of the lattice and, therefore, if the geometry of the rest of the lattice is modified (is "flexed"), the geometry of the `patch` will vary as well. See Section §11.2 for an example.

If a `flexible` patch is within a `multipass` (§7.2) region (as opposed to being just outside a multipass region as in the example in Section §11.2), the offsets, pitches, and tilt will be computed using the geometry of the first pass.

With `bmad_standard` tracking (§5.1) A particle, starting at the upstream face of the `patch`, is propagated in a straight line to the downstream face and the suitable coordinate transformation is made to translate the particle's coordinates from the upstream coordinate frame to the downstream coordinate frame (§22.12). In this case the `patch` element can be thought of as a generalized `drift` element.

If there are magnetic or electric fields within the `patch`, the tracking method through the `patch` must be set to either `runge_kutta` or `custom`. Example:

```
pa2: patch, tracking_method = runge_kutta, field_calc = custom,
            mat6_calc_method = tracking, ...
```

In order to supply a custom field when `runge_kutta` tracking is used, `field_calc` (§5.4) needs to be set to `custom`. In this case, custom code must be supplied for calculating the fields as a function of position (§33.2).

The `e_tot_offset` attribute offsets the reference energy:

```
E_tot_ref(exit) = E_tot_ref(entrance) + E_tot_offset (eV)
```

Setting the `e_tot_offset` attribute will affect a particle's p_x, p_y and p_z coordinates via Eqs. (14.27) and (14.31). Notice that `e_tot_offset` does not affect a particle's actual energy, it just affects the difference between the particle energy and the reference energy.

Alternatively, to set the reference energy, the `E_tot_set` or `p0c_set` attributes can be used to set the reference energy/momentum at the exit end. It is is an error if more than one of `e_tot_offset`, `E_tot_set` and `p0c_set` is nonzero.

`Important`: *Bmad* may apply the energy transformation either before or after the coordinate transformation. This matters when the speed of the reference particle is less than c. For this reason, and due to complications involving PTC, it is recommended to use two patches in a row when both the orbit and energy are to be patched.

The `t_offset` attribute offsets the reference time so that the reference time at the exit end of the patch `t_ref(exit)` is related to the reference time at the beginning of the patch `t_ref(entrance)` via

```
t_ref(exit) = t_ref(entrance) + t_offset + dt_travel_ref
```

where `dt_travel_ref` is the time for the reference particle to travel through the patch. `dt_travel_ref` is defined to be:

```
dt_travel_ref = L / beta_ref
```

Where `L` is the length of the `patch` as shown in Fig. 3.6 and `beta_ref` is the reference velocity/c at the exit end of the element. That is, the reference energy offset is applied *before* the reference particle is tracked through the patch. Since this point can be confusing, it is recommended that a `patch` element be split into two consecutive patches if the `patch` has finite `l` and `E_tot_offset` values.

While a finite `t_offset` will affect the reference time at the end of a patch, a finite `t_offset` will *not* affect the time that is calculated for a particle to reach the end of the patch. On the other hand, a finite `t_offset` will affect a particle's z coordinate via Eqs. (14.28). The change in z, δz will be

$$\delta z = \beta \cdot c \cdot \text{t\_offset} \tag{3.16}$$

where β is the normalized particle speed (which is independent of any energy patch). Another way of looking at this is to note that In a drift, if the particle is on-axis and on-energy, t and t_ref change but z does not change. In a time patch (a patch with only `t_offset` finite), t_ref and z change but t does not.

When a lattice branch contains both normally oriented and reversed elements (§14.1.2), a `patch`, or series of `patches`, which reflects the z direction must be placed in between. See §11.3 for an example. Such a `patch`, (or patches) is called a `reflection patch`. See Section §14.2.6 for more details on how a reflection patch is defined.

Since the geometry of a `patch` element is complicated, interpolation of the chamber wall in the region of a patch follows special rules. See section §4.11.5 for more details.

3.38 Photon_Init

A `photon_init` element is used as a starting element for x-ray tracking. A `photon_init` element can be used to define such things as the initial energy spectrum and angular orientation. As explained below, a `photon_init` element can be a "stand alone" photon source or it can have an associated "physical source" element.

General `photon_init` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Length	4.12
Chamber wall	4.11	Offsets, pitches & tilt	4.6
Custom Attributes	2.9	Reference energy	4.5
Description strings	4.3	Tracking & transfer map	5

See §13.36 for a full list of element attributes.

Attributes specific to an photon_init element are:

```
  ds_slice                 = <Real>
  E_center                 = <Real>    ! Average init photon energy of 1st mode (eV).
  E2_center                = <Real>    ! Average init photon energy of 2nd mode (eV).
  E2_probability           = <Real>    ! Probability of 2nd mode.
  E_center_relative_to_ref = <Logic>   ! E_center relative to reference E?
  e_field_x                = <Real>    ! Polarization. x & y = 0 -> random
  e_field_y                = <Real>
  energy_distribution      = <Switch>  ! Gaussian or uniform
  physical_source          = <String>  ! physical source of x-rays
  ref_wavelength                       ! Reference wavelength (§4.5). Dependent attribute (§4.1).
  sig_x                    = <Real>
  sig_y                    = <Real>
  sig_z                    = <Real>
  sig_vx                   = <Real>
  sig_vy                   = <Real>
  sig_E                    = <Real>    ! Init photon energy width of 1st mode (eV).
  sig_E2                   = <Real>    ! Init photon energy width of 2nd mode (eV).
  spatial_distribution     = <Switch>  ! Gaussian or uniform.
  transverse_sigma_cut     = <Real>
  velocity_distribution    = <Switch>  ! Gaussian, spherical, or uniform.
```

The distribution of photons is bimodal. The first mode is characterized by the parameters E_center, and sig_E, the second mode is characterized by the parameters E2_center and sig_E2. The probability of emitting a photon in the second mode is given by E2_probability.

ds_slice
: Used when there is an associated physical source element. The physical source element is sliced into pieces of thickness ds_slice and each slice is tested to see if photons from the slice can possibly pass through the first aperture. When photons are generated, photons will only be generated from slices where they have a hope of passing through the first aperture. This makes the simulation more efficient. The default value of ds_slice is 0.01 meter.

E_center, E2_center
: Average initial photon energy in eV. If E_center_relative_to_ref is set to True, E_center and E2_center will be relative to the reference energy.

E_center_relative_to_ref
: With a setting of True (the default), E_center and E2_center are taken to be with respect to the reference energy (§14.4.1). That is, if True, the center energy <E> is

```
<E-1st-mode> = E1_center + Reference_Energy
<E-2nd-mode> = E2_center + Reference_Energy
```

If E_center_relative_to_ref is set to False, E_center and E2_center are taken to be the center energy values independent of the reference energy.

`E2_probability`
Probability of emitting a photon from the 2nd mode. A value of 0 (the default) will mean that all photons will be emitted from the 1st mode and a value of 1 will mean that all photons will be emitted from the 2nd mode.

`e_field_x`, `e_field_y`
Electric field component of initial photons in the x and y planes. If both are set to 0 then a random field is chosen with unit intensity $E_x^2 + E_y^2 = 1$.

`energy_distribution`
Sets the type of energy spectrum for emitted photons. If there is an associated physical element then this parameter is ignored and the energy distribution is calculated from the properties of the physical element. Possible settings are:

```
gaussian   ! Default
uniform
```

The `gaussian` setting gives Gaussian distributions for the two modes with width set by `sig_E` and `sig_E2`. The `uniform` setting gives a flat distribution in the range:

```
[-sig_E, sig_E]    ! For the 1st mode
[-sig_E2, sig_E2]  ! For the 2nd mode
```

`physical_source`
Used to specify the "physical" source of the photons. See below for more details

`sig_E`, `sig_E2`
Energy width of the two modes in eV. See `energy_distribution` for more details.

`sig_vx`, `sig_vy`
Width of emitted photons in v_x/c and v_y/c directions. See `velocity_distribution` for more details.

`sig_x`, `sig_y`, `sig_z`
Width of emitted photons in x, y and z directions. See `spatial_distribution` for more details.

`spatial_distribution`
Sets spacial (x, y, z) spectrum of emitted photons. If there is an associated physical element then this parameter is ignored and the energy distribution is calculated from the properties of the physical element. Possible settings are:

```
gaussian    ! Default
uniform
```

The `gaussian` setting gives a Gaussian distribution with width σ where *sigma* is

```
sig_x     ! for x distribution
sig_y     ! for y distribution
sig_z     ! for z distribution
```

The `uniform` setting gives a flat distribution in the range: $[-\sigma, \sigma]$.

`velocity_distribution`
Sets the transverse $(v_x/c, v_y/c)$ velocity spectrum of emitted photons. If there is an associated physical element then this parameter is ignored and the energy distribution is calculated from the properties of the physical element. The longitudinal velocity is always computed to make $v_x^2 + v_y^2 + v_z^2 = c^2$ Possible settings are:

```
gaussian    ! Default
spherical
uniform
```

The `gaussian` setting gives a Gaussian distribution with width σ where *sigma* is

```
sig_vx     for vx/c distribution
sig_vy     for vy/c distribution
```

The `uniform` setting gives a flat distribution in the range: $[-\sigma, \sigma]$. The `spherical` setting gives flat distribution in all directions. With the `spherical` setting is used, and the next downstream element excluding drifts and markers is an element with aperture limits (§4.8), *Bmad* can optimize photon emission to only emitting photons that are very likely to be within the aperture when they hit the downstream element. This cuts down on computation time.

For the purposes of positioning the elements in the lattice around it, a `photon_init` element is considered to have zero length.

`photon_init` elements are used in one of two modes: With or without an associated physical source element specified by the `physical_source` attribute. Without an associated physical source, the `photon_init` element completely specifies the initial photon distribution. With an associated physical source element, the photon distribution is determined by the properties of the physical source but the shape of the energy spectrum can be modified by setting attributes in the `photon_init` element. Example:

```
b05w: sbend, l = 3.2, angle = 0.1
pfork: photon_fork, to_line = c_line, superimpose, ref = b05w, offset = 0.4
bend_line: line = (..., b05w, ...)
use bend_line

c_line: line = (pinit, ...)
c_line[e_tot] = 15e3
pinit: photon_init, physical_source = 'b05w', sig_E = 2.1
```

In this example, the bend `b05w` is a bend producing photons. It is part of the line `bend_line`. `bend_line` also contains a `photon_fork` element named `pfork` which branches to the line `c_line`. `c_line` contains the `photon_init` element `pinit` which references `b03w` as the associated physical source element. When photons are tracked, they are generated in `b05w` and then propagated to the `pfork` fork. After this they are propagated through `c_line`. The `pinit` element acts like a zero length `marker` element when photons propagate through it. That is, the `pinit` element essentially serves to associate `c_line` with `b03w` for the purposes of photon tracking. Also, in this example, `pinit` modifies the photon energy spectrum so that only photons whose energy is within 2.1 eV are generated

It is important to note that in the above example, with the `photon_init` element having an associated physical source, the setting of things like the spatial shape `sig_z`, etc. in the `photon_init` element will be ignored.

See Section §11.5 for an example lattice that can be used to simulate a Rowland circle spectrometer using a `photon_init` element.

3.39 Quadrupole

A `quadrupole` is a magnetic element with a linear field dependence with transverse offset (§15.1).

General `quadrupole` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Mag & Elec multipoles	4.14
Chamber wall	4.11	Offsets, pitches & tilt	4.6
Description strings	4.3	Overlapping Fields	4.17
Fringe Fields	4.20	Reference energy	4.5
Hkick & Vkick	4.7	Superposition	7.1
Integration settings	5.4	Symplectify	5.6
Is_on	4.13	Field Maps	4.15
Length	4.12	Tracking & transfer map	5

See §13.37 for a full list of element attributes.

Attributes specific to a `quadrupole` element are:

```
b1_gradient     = <Real>    ! Field strength. (§4.1).
k1              = <Real>    ! Quadrupole strength.
fq1             = <Real>    ! Soft edge fringe parameter.
fq2             = <Real>    ! Soft edge fringe parameter.
```

If the `tilt` attribute is present without a value then a value of $\pi/4$ is used.

For a quadrupole with zero `tilt` and a positive `k1`, the quadrupole is horizontally focusing and vertically defocusing (§15.1).

The `fq1` and `fq2` parameters are used to specify the quadrupolar "soft" edge fringe. See §16.3 for more details. The `fringe_at` and `fringe_type` settings (§4.20) determine if the fringe field is used in tracking (§4.20).

Example:

```
q03w: quad, l = 0.6, k1 = 0.003, tilt  ! same as tilt = pi/4
```

3.40 RFcavity

An `rfcavity` is an RF cavity without acceleration generally used in a storage ring. The main difference between an `rfcavity` and an `lcavity` is that, unlike an `lcavity`, the reference energy (§14.4.2) through an `rfcavity` is constant.

General `rfcavity` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Offsets, pitches & tilt	4.6
Chamber wall	4.11	Overlapping Fields	4.17
Custom Attributes	2.9	Reference energy	4.5
Description strings	4.3	RF Couplers	4.16
Field autoscaling	4.18	Superposition	7.1
Fringe Fields	4.20	Symplectify	5.6
Hkick & Vkick	4.7	Field Maps	4.15
Integration settings	5.4	Tracking & transfer map	5
Is_on	4.13	Wakes	4.19
Length	4.12		

See §13.38 for a full list of element attributes.

Attributes specific to an `rfcavity` are:
```
rf_frequency    = <Real>    ! Frequency
harmon          = <Real>    ! Harmonic number
voltage         = <Real>    ! Cavity voltage
phi0            = <Real>    ! Cavity phase
phi0_multipass  = <Real>    ! Phase variation with multipass
phi0_autoscale  = <Real>    ! Set by Bmad if autoscaling is turned on.
gradient        = <Real>    ! Accelerating gradient (V/m). Dependent attribute (§4.1).
```
The `phi0` attribute here is identical to the `lag` attribute of *MAD*. The integrated energy kick felt by a particle, assuming no phase slippage, is
```
dE = -e_charge * voltage * sin(2π * (φ_t - φ_ref))
```
where
```
φ_ref = phi0 + phi0_multipass + phi0_autoscale
```

and ϕ_t is the part of the phase due to when the particle arrives at the cavity and depends upon whether `absolute time tracking` or `relative time tracking` is being used as discussed in §22.1.

`phi0_multipass` is only to be used to shift the phase with respect to a `multipass` lord. See §7.2. `e_charge` is the charge on an electron (Table 2.2). Notice that the energy kick is independent of the sign of the charge of the particle

`phi0_autoscale` and `field_autoscale` are calculated by *Bmad*'s auto-scale module. See Section §4.18 for more details. Autoscaling can be toggled on/off by using the `autoscale_phase` and `autoscale_amplitude` toggles.

Note: Zero phase for $\phi_{\rm ref}$ corresponds to the stable fixed point above transition.

If `harmon` is non–zero the `rf_frequency` is calculated by
```
rf_frequency = harmon * c_light * beta0 / L_lattice
```
where `L_lattice` is the total lattice length and `beta0` is the velocity of the reference particle at the start of the lattice. After the lattice has been read in, `rf_frequency` will be the independent variable (§4.1).

Couplers (§4.16) and HOM wakes (§4.19) can be modeled. In addition, if a field map is specified (§4.15), tracking using an integrator is possible.

If a field map is specified (§4.15), tracking using an integrator is possible. A field map is only used for `runge_kutta`, `fixed_step_runge_kutta`, and `symp_lie_bmad` tracking (§5.1). Only the fundamental mode has an analytical formula for the symplectic tracking. In the future, the other modes could be used with `symp_lie_bmad` tracking using a field expansion about the centerline.

The `cavity_type` is the type of cavity being simulated. Possible settings are:
```
ptc_standard
standing_wave    ! Default
traveling_wave
```
The `cavity_type` switch is ignored if a field map is used.

Example:
```
rf1: rfcav, l = 4.5, harmon = 1281, voltage = 5e6
```

3.41 Sad_Mult

A `sad_mult` element is equivalent to a SAD[SAD] `mult` element. This element is a combination solenoid, multipole, bend, and RF cavity.

General sample attributes are:

Attribute Class	Section	*Attribute Class*	Section
a*n*, b*n* multipoles	4.14	Length	4.12
Aperture limits	4.8	Offsets, pitches & tilt	4.6
Chamber wall	4.11	Reference energy	4.5
Custom Attributes	2.9	Superposition	7.1
Description strings	4.3	Tracking & transfer map	5
Fringe Fields	4.20		

See §13.39 for a full list of element attributes.

Attributes specific to an sextupole element are:

```
bs_field        = <Real>    ! Solenoid field. SAD equivalent: BZ.
e1, e2          = <Real>    ! Bend face angles.
eps_step_scale  = <Real>    ! Step size scale. Default = 1. SAD equivalent: EPS.
fq1             = <Real>    ! Quadrupole fringe integral. SAD equivalent: F1.
fq2             = <Real>    ! Quadrupole fringe integral. SAD equivalent: F2.
ks              = <Real>    ! Solenoid strength.
x_offset_mult   = <Real>    ! Mult component offset. SAD equivalent: DX.
y_offset_mult   = <Real>    ! Mult component offset. SAD equivalent: DY.
x_pitch_mult    = <Real>    ! Mult component pitch. SAD equivalent: DPX or CHI1.
y_pitch_mult    = <Real>    ! Mult component pitch. SAD equivalent: DPY or CHI2.
fringe_type     = <Switch>  ! Type of fringe. SAD equivalent: DISFRIN.
fringe_at       = <Switch>  ! Where fringe is applied. SAD equivalent: FRINGE.
```

One difference between SAD and *Bmad* is that SAD defines the solenoid field by what are essentially a set of marker elements so that the solenoid field at a SAD mult element is not explicitly declared in the mult element definition. *Bmad*, on the other hand, requires a sad_mult element to explicitly declare the solenoid parameters.

Another difference between SAD and *Bmad* is that, within a solenoid, the reference trajectory is aligned with the solenoid axis (and not aligned with the axis of the elements within the solenoid region).

The SAD mult element uses normal Kn and skew KSn multipole components. The *Bmad* sad_mult element used normal an and skew bn multipole components. As can be seen from the equations in §15.1, there is a factor of $n!$ between the two representations.

The fq1 and fq2 parameters are used to specify the quadrupolar "soft" edge fringe. See §16.3 for more details.

The fringe_at and fringe_type settings determine if the fringe field is used in tracking. See Sec §4.20 for the translation between these two switches and the fringe and disfrin switches of SAD.

Unlike other elements, the ds_step and num_steps attributes (§5.4) of a sad_mult are dependent attributes (§4.1) and are not directly settable. Rather these attributes are calculated using SAD's own algorithm for setting the step size. To vary the calculated step size for a single sad_mult element, the attribute eps_step_scale may be set. To vary the step size for all sad_mult elements, the global parameter bmad_com[sad_eps_scale] (§9.3) may be set. The default values for these parameters are:

```
eps_step_scale           = 1
bmad_com[sad_eps_scale] = 5e-3
```

SAD conventions to be aware of when comparing SAD to Bmad:

- A SAD rotate or chi3 rotation is opposite to a *Bmad* tilt

- SAD element offsets (`dx`, `dy`, `dz`) are with respect to the entrance end of the element as opposed to *Bmad*'s convention of referencing to the element center.

- The *Bmad* `sad_mult` element does not have any attributes corresponding to the following SAD `MULT` element attributes:

  `angle, harmon, freq, phi, dphi, volt, dvolt`

3.42 Sample

A `sample` element is used to simulate a material sample which is illuminated by x-rays.

General `sample` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Offsets, pitches & tilt	4.6
Chamber wall	4.11	Reference energy	4.5
Custom Attributes	2.9	Surface Properties	4.10
Description strings	4.3	Superposition	7.1
Integration settings	5.4	Tracking & transfer map	5
Length	4.12		

See §13.40 for a full list of element attributes.

This element is in development.

Attributes specific to an `solenoid` element are:

```
mode       = <Switch> ! Reflection or transmission.
material   = <type>   ! Type of material. §4.9
```

The `mode` parameter can be set to:

```
reflection
transmission
```

With `mode` set to `reflection`, photons will be back scattered from the sample surface isotropically. In this case the material properties will not matter. Additionally, a `patch` (§3.37) element will be needed after the `sample` element to properly reorient the reference orbit.

With `mode` set to `transmission`, photons will be transmitted through the sample. In this case `material` will be used to determine the attenuation and phase shift of the photons.

Example:

```
formula409: sample, x_limit = 10e-3, y_limit = 20e-3, mode = reflection
```

3.43 Sextupole

A `sextupole` is a magnetic element with a quadratic field dependence with transverse offset (§15.1).

General `sextupole` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Mag & Elec multipoles	4.14
Chamber wall	4.11	Offsets, pitches & tilt	4.6
Custom Attributes	2.9	Overlapping Fields	4.17
Description strings	4.3	Reference energy	4.5
Fringe Fields	4.20	Superposition	7.1
Hkick & Vkick	4.7	Symplectify	5.6
Integration settings	5.4	Field Maps	4.15
Is_on	4.13	Tracking & transfer map	5
Length	4.12		

See §13.41 for a full list of element attributes.

Attributes specific to an `sextupole` element are:

```
k2          = <Real>   ! Sextupole strength.
b2_gradient = <Real>   ! Field strength. (§4.1).
```

The `bmad_standard` calculation treats a sextupole using a kick–drift–kick model.

If the `tilt` attribute is present without a value then a value of $\pi/6$ is used. Example:

```
q03w: sext, l = 0.6, k2 = 0.3, tilt  ! same as tilt = pi/6
```

3.44 Solenoid

A `solenoid` is an element with a longitudinal magnetic field.

General `solenoid` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Mag & Elec multipoles	4.14
Chamber wall	4.11	Offsets, pitches & tilt	4.6
Custom Attributes	2.9	Overlapping Fields	4.17
Description strings	4.3	Reference energy	4.5
Fringe Fields	4.20	Superposition	7.1
Hkick & Vkick	4.7	Symplectify	5.6
Integration settings	5.4	Field Maps	4.15
Is_on	4.13	Tracking & transfer map	5
Length	4.12		

See §13.43 for a full list of element attributes.

Attributes specific to an `solenoid` element are:

```
ks         = <Real>   ! Solenoid strength.
bs_field   = <Real>   ! Field strength. (§4.1).
```

The `bmad_standard` tracking model (§5.1) uses a "hard edge" model where an impulse kick is applied at the entrance and exit ends of the element due to the fringe fields there.

Example:

```
cleo_sol: solenoid, l = 2.6, ks = 1.5e-9 * parameter[p0c]
```

3.45 Sol_Quad

A sol_quad is a combination solenoid/quadrupole. Alternatively, the sad_mult element can also be used. The advantage of the sad_mult element is that it can simulate a quadrupole field that is canted with respect to the solenoid field.

General sol_quad attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Mag & Elec multipoles	4.14
Chamber wall	4.11	Offsets, pitches & tilt	4.6
Custom Attributes	2.9	Overlapping Fields	4.17
Description strings	4.3	Reference energy	4.5
Fringe Fields	4.20	Superposition	7.1
Hkick & Vkick	4.7	Symplectify	5.6
Integration settings	5.4	Field Maps	4.15
Is_on	4.13	Tracking & transfer map	5
Length	4.12		

See §13.42 for a full list of element attributes.

Attributes specific to a sol_quad element are:

```
k1          = <Real>    ! Quadrupole strength.
ks          = <Real>    ! Solenoid strength.
bs_field    = <Real>    ! Solenoid Field strength.
b1_gradient = <Real>    ! Quadrupole Field strength.
```

Example:

```
sq02: sol_quad, l = 2.6, k1 = 0.632, ks = 1.5e-9*parameter[p0c]
```

3.46 Taylor

A taylor is a Taylor map (§21.1) that maps the input orbital phase space and possibly spin coordinates of a particle at the entrance end of the element to the output orbital and spin coordinates at the exit end of the element. This can be used in place of the *MAD* matrix element.

General taylor attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Offsets & tilt	4.6
Custom Attributes	2.9	Reference energy	4.5
Description strings	4.3	Superposition	7.1
Is_on	4.13	Symplectify	5.6
Length	4.12	Tracking & transfer map	5

See §13.44 for a full list of element attributes.

Attributes specific to a taylor element are:

```
ref_orbit = (<x>, <px>, <y>, <py>, <z>, <pz>)     ! Reference orbit.
```

```
x_ref  = <Real>                                    ! $x$ reference orbit component.
px_ref = <Real>                                    ! $p_x$ reference orbit component.
y_ref  = <Real>                                    ! $y$ reference orbit component.
py_ref = <Real>                                    ! $p_y$ reference orbit component.
z_ref  = <Real>                                    ! $z$ reference orbit component.
pz_ref = <Real>                                    ! $p_z$ reference orbit component.
{<out>: <coef>, <e1> <e2> <e3> <e4> <e5> <e6>}     ! Taylor term. First form.
{<out>: <coef> | <n1> <n2> ...}                    ! Taylor term. Second form.
tt<out><n1><n2>...  = <Coef>                       ! Taylor term. Third form.
delta_ref_time = <Real>                            ! Change in the reference time.
delta_e_ref = <Real>                               ! Change in the reference energy.
```

For historical reasons, there are three different forms that can be used to specify a taylor term. Notice that the first form (above) uses a comma "," to separate the `<coef>` from `<e1>`, while the second form uses a vertical bar "|" to separate `<coef>` from `<n1>`.

The orbital (x, p_x, y, p_y, z, p_z) part of the Taylor map, $\mathcal{M}$, maps input orbital coordinates $\mathbf{r}(\text{in})$ to the output orbital coordinates $\mathbf{r}(\text{out})$

$$\mathbf{r}(\text{out}) = \boldsymbol{\mathcal{M}}(\mathbf{r}(\text{in})) \tag{3.17}$$

Notice that Stern-Gerlach effects are ignored so that the output coordinates are independent of the spin. $\boldsymbol{\mathcal{M}}$ has six components $\mathcal{M}_j$ one for each output coordinate $r_j(\text{out})$

$$r_j(\text{out}) = \mathcal{M}_j(\mathbf{r}(\text{in})) \tag{3.18}$$

Each $\mathcal{M}_j$ is made up of a number of terms

$$\mathcal{M}_j = \sum_{k=1}^{N_j} M_{jk} \tag{3.19}$$

and each term M_{jk} is a polynomial in the input orbital coordinates with respect to the reference orbit.

$$M_{jk}(\mathbf{r}(\text{in})) = C_{jk} \cdot \Pi_{i=1}^{6} \delta r_i^{e_{ijk}} \tag{3.20}$$

where C_{jk} is the coefficient for the term, the e_{ijk} are integer exponents, and $\delta\mathbf{r} = \mathbf{r}(\text{in}) - \mathbf{r}_{\text{ref}}$ with $\mathbf{r}_{\text{ref}}$ with being the reference orbit.

A term in a Taylor map can be specified by one of three forms as shown above. The first form is

```
{<out>: <coef>, <e1> <e2> <e3> <e4> <e5> <e6>}
```

`<Out>` is an integer in the range 1 to 6 corresponding to the index j in Eq. (3.20) (`<out>` = 1 for x, etc.). `<coef>` corresponds to C_{jk} in Eq. (3.20), and `<e1>`, `<e2>`, `<e3>`, `<e4>`, `<e5>`, and `<e6>` correspond to $e_{ijk}, i = 1 \ldots 6$. For example, the Taylor map

$$p_y(\text{out}) = 0.9 \cdot \delta x + 2.73 \cdot \delta y^2(\text{in})\, \delta p_z(\text{in}) + \ldots \tag{3.21}$$

would be written as

```
{4: 0.9: 1 0 0 0 0 0}, {4: 2.73, 0 0 2 0 0 1}, ...
```

The second form for specifying a Taylor term uses the syntax:

```
{<out>: <coef> | <n1> <n2> ...}
```

The set of integers `<n1>`, `<n2>` each must be between 1 and 6 inclusive. The value of the i^{th} exponent e_{ijk} in Eq. (3.20) is equal to the number of integers that are equal to i. For example, the above Taylor map would be written using the second form as

```
{4: 0.9 | 1}, {4: 2.73 | 336}, ...
```

Notice that with the second form, spaces between exponent integers is optional.

The third form is like the second form and has the syntax:

```
tt<out><n1><n2>...  = <Coef>                    ! Taylor term. Third form.
```

For example, the Taylor map above would be written using the third form as:

```
tt41 = 0.9, tt4336 = 2.73, ...
```

The spin (§19.1) part of the transport map $\boldsymbol{\mathcal{Q}}$ (§21.2) gives the spin rotation quaternion $\mathbf{q}$ (§19.2) as a function of input orbital coordinates (the form of the T-BMT equation assures that $\boldsymbol{\mathcal{Q}}$ cannot depend upon the spin coordinates):

$$\mathbf{q} = \boldsymbol{\mathcal{Q}}(\mathbf{r}(\text{in})) \tag{3.22}$$

$\mathbf{q}$ has four components and in analogy to Eq. (3.18) one writes

$$q_j = \mathcal{Q}_j(\mathbf{r}(\text{in})) \tag{3.23}$$

Each $\mathcal{Q}_j$ is made up of a number of terms

$$\mathcal{Q}_j = \sum_{k=1}^{N_j} Q_{jk} \tag{3.24}$$

and each term Q_{jk} is a polynomial in the input orbital coordinates with respect to the reference orbit.

$$Q_{jk}(\mathbf{r}(\text{in})) = C_{jk} \cdot \Pi_{i=1}^{6}\, \delta r_i^{e_{ijk}} \tag{3.25}$$

Rather than using an integer index, the four components of a quaternion are labeled (`S1`, `Sx`, `Sy`, `Sz`). The syntax for the spin part uses the first or second forms as described above. For example

```
{Sx: 0.43 | 13 }
```

is equivalent to the term

$$S_x = 0.43 \cdot \delta x(\text{in})\, \delta y(\text{in}) \tag{3.26}$$

By default, a `taylor` element starts out with the unit phase space map but no default is defined for the spin part of the map. That is, a `taylor` element starts with the following 6 terms

```
{1: 1.0, 1 0 0 0 0 0}
{2: 1.0, 0 1 0 0 0 0}
{3: 1.0, 0 0 1 0 0 0}
{4: 1.0, 0 0 0 1 0 0}
{5: 1.0, 0 0 0 0 1 0}
{6: 1.0, 0 0 0 0 0 1}
```

Which is equivalent to

```
{1: 1.0 | 1}
{2: 1.0 | 2}
{3: 1.0 | 3}
{4: 1.0 | 4}
{5: 1.0 | 5}
{6: 1.0 | 6}
```

The `ref_orbit` attribute specifies the phase space (x, px, y, py, z, pz) reference orbit at the start of the element used to construct the Taylor map. Alternatively, the individual components of the reference orbit may be specified by using the attributes `x_ref`, `px_ref`, `y_ref`, `py_ref`, `z_ref`, or `pz_ref`.

Note: when converting the map from *Bmad* to PTC (§34), the *Bmad*/PTC interface code will convert from Bmad phase space coordinates to PTC phase space coordinates and will convert the map to using the reference orbit as the map zero orbit. This does not affect tracking but will affect map analysis.

A term in a `taylor` element will override any previous term with the same `out` and `e1` through `e6` indexes. For example the term:

```
tt: Taylor, {1: 4.5, 1 0 0 0 0 0}
```

will override the default `{1:  1.0, 1 0 0 0 0 0}` term.

The `l` length attribute of a `taylor` element does not affect phase space coordinates but will affect the longitudinal s position of succeeding elements and will affect the time it takes a particle to track through the element The calculation involves first calculating the change in reference time which is the time a particle with the reference energy would take to transverse the element. Next, Eq. (14.28) is used with the change in the phase space z coordinate to calculate the time a particle takes to traverse the element.

The time a particle takes to track through a `taylor` element can also be controlled by setting the `delta_ref_time` attribute which sets the travel time for the reference particle. `delta_ref_time` is a dependent attribute so that if both `l` and `delta_ref_time` are set, the value of `delta_ref_time` will be modified by *Bmad* to correspond to the setting of `l`.

The `delta_e_ref` attribute can be used to modify the reference energy at the exit end of the `taylor` element. The phase space transport is completely determined by the Taylor map and is independent of `delta_e_ref`. For example, with a unit Taylor map, the phase space coordinates p_x and p_y constant through the element independent of `delta_e_ref`. However, a finite `delta_e_ref` will modify the reference momentum P_0 and hence through Eq. (14.27) will affect the transport downstream of the `Taylor` element. This behavior is in contrast to how `delta_e_ref` is handled in a `patch` element. In a `patch` element, the transformation used when `delta_e_ref` is non-zero is to hold as constant the actual transverse momenta P_x and P_y and then p_x and p_y are modified using Eq. (14.27).

A `taylor` element that is "turned off" (`is_on` attribute set to False), is considered to be like a `marker` element. That is, the orbit and twiss parameters are unchanged when tracking through a `taylor` element that is turned off.

Example `taylor` element definition:

```
tt: Taylor, {4:  2.7, 0 0 2 0 0 1}, {2:  1.9 | 1 1 2},
            {S1: 0.43 | 2 }, ...,
            ref_orbit = (0.01, 0.003, 0.002, 0.001, 0.0, 0.2)
```

Note: When tracking a particle's spin through a map, the quaternion used to rotate the spin is always normalized to one so that the magnitude of the spin will be invariant.

Note: Tracking through a `taylor` elements using `symp_lie_ptc` is the same as tracking with the `taylor` tracking method. That is, the Taylor map is simply evaluated and no effort at symplectification is done. Furthermore, evaluating the Taylor map of a `taylor` element using the `taylor` method is faster than evaluation using `symp_lie_ptc`. Thus the `taylor` tracking method should always be used with `taylor` elements.

3.47 Wiggler and Undulator

A `wiggler` or `undulator` element is basically a periodic array of alternating bends. The difference between `wigglers` and `undulators` is in the x-ray emission spectrum. Charged particle tracking will be the same.

Henceforth, the term "`wiggler`" will denote either a `wiggler` or `undulator`

General `wiggler` attributes are:

Attribute Class	Section	*Attribute Class*	Section
Aperture limits	4.8	Mag & Elec multipoles	4.14
Chamber wall	4.11	Offsets, pitches & tilt	4.6
Custom Attributes	2.9	Overlapping Fields	4.17
Description strings	4.3	Reference energy	4.5
Fringe Fields	4.20	Superposition	7.1
Hkick & Vkick	4.7	Symplectify	5.6
Integration settings	5.4	Field Maps	4.15
Is_on	4.13	Tracking & transfer map	5
Length	4.12		

See §13.45 for a full list of element attributes.

There are three types of wigglers. Wigglers that are described using a magnetic field map are called "`map type`" and are discussed in §3.47.2. Wigglers that are described assuming a periodic field are called "`periodic type`" and are described in §3.47.1. The third type of wiggler has a custom field. The different wiggler types are distinguished by the setting of the element's `field_calc` parameter as discussed in section §5.5.2. For example:

```
wig1: wiggler, l = 1.6, field_calc = fieldmap, ...
```

In this example `wig1` is a map type wiggler.

Attributes specific to wiggler and undulator elements are:

```
b_max      = <Real>  ! Maximum magnetic field (in T) on the wiggler centerline.
l_period   = <Real>  ! Length over which field vector returns to the same orientation.
n_period   = <Real>  ! The number of periods (L / L_period). A settable dependent attribute (§4.1).
l_pole     = <Real>  ! Wiggler pole length. DEPRECATED. USE L_PERIOD INSTEAD.
n_pole     = <Real>  ! The number of poles. DEPRECATED. USE N_PERIOD INSTEAD.
polarity   = <Real>  ! For scaling the field.
k1_pseudo            ! Pseudo K1 value. A Dependent attribute (§4.1).
g_max                ! Maximum bending strength. Dependent attribute.
osc_amplitude        ! Amplitude of the particle oscillations. Dependent attribute.
x_ray_line_len = <Real> ! Deprecated.
```

The `polarity` value is used to scale the magnetic field. By default, `polarity` has a value of 1.0. Example:

```
wig1: wiggler, l = 1.6, polarity = -1, cartesian_map = {...}
```

In this example the wiggler field is defined by a Cartesian map (§4.15.2) and the field is reversed from what it would be with `polarity` set to 1.

`x_ray_line_len` is the length of an associated x-ray synchrotron light line measured from the exit end of the element. This is used for machine geometry calculations and is irrelevant for lattice computations. `x_ray_line_len` is deprecated. Use a `photon_fork` (§3.19) with an attached lattice branch instead.

3.47.1 Periodic Type Wigglers

`Periodic type` wiggers are modeled assuming the field is periodic longitudinally. `Periodic type` wigglers have their `field_calc` parameter set to one of

```
planar_model    ! Default
helical_model
```

For historic purposes, if there is no fieldmap defined for the element (that is, it is not a `map type` wiggler), and `field_calc` is not set, then `field_calc` will default to `planar_model`.

Example:

```
wig2: wiggler, l = 1.6, b_max = 2.1, n_period = 8
```

This defines a periodic type wiggler with `field_type` defaulting to `planar_model`.

For the `planar_model`, wigglers use a simplified model where the wiggler has infinitely wide poles and the magnetic field components are

$$\begin{aligned} B_x &= 0 \\ B_y &= \quad \text{b\_max} \cosh(k_z\, y) \cos(k_z\, z + \phi_z) \\ B_z &= -\text{b\_max} \sinh(k_z\, y) \sin(k_z\, z + \phi_z) \end{aligned} \tag{3.27}$$

where z is the distance from the beginning of the wiggler, the input parameter `b_max` is the maximum field on the centerline, and k is given in terms of the period length (`l_period`) by

$$k_z = \frac{2\pi}{l_{\text{period}}} \tag{3.28}$$

Here the phase ϕ_z is chosen so that B_y is symmetric about the center of the wiggler

$$\phi_z = \frac{-k_z\, L}{2} \tag{3.29}$$

Note: Originally k_z was calculated using `l_pole` — the length of a pole — with the period length being twice the pole length. When the helical model option was introduced this became problematical since the period of a helical wiggler could be either 2 or 4 times the pole length depending upon the geometry. As a result, using the pole length was depracated and instead the period length or number should be used.

The `helical_model` for the field is

$$\begin{aligned} B_x &= -\text{b\_max} \cosh(k_z\, x) \sin(k_z\, z + \phi_z) \\ B_y &= \quad \text{b\_max} \cosh(k_z\, y) \cos(k_z\, z + \phi_z) \\ B_z &= -\text{b\_max} \left[\sinh(k_z\, x) \cos(k_z\, z + \phi_z) + \sinh(k_z\, y) \sin(k_z\, z + \phi_z)\right] \end{aligned} \tag{3.30}$$

With `field_calc` set to `planar_model`, and with `bmad_standard` tracking (§5), the vertical focusing is assumed small so averaged over many periods the horizontal motion looks like a drift and the vertical motion is modeled as a combination focusing quadrupole and focusing octupole giving a kick[Corbett99]

$$\frac{dp_y}{dz} = k1 \left(y + \frac{2}{3}\, k_z^2\, y^3\right) \tag{3.31}$$

where

$$g_{\max} = \frac{e\, B_{\max}}{P_0\, (1 + p_z)} \tag{3.32}$$

$$\text{k1\_pseudo} = \frac{-1}{2}\, g_{\max}^2 \tag{3.33}$$

with `k1_pseudo` (a dependent element attribute) being the linear focusing constant. Notice that the focusing is only in the vertical plane. The subscript `pseudo` is used to remind the user that this is not the same as the `k1` attribute of a quadrupole since the focusing is only in one plane.

With `field_calc` set to `helical_model`, and with `bmad_standard` tracking, the transport in the vertical and horizontal planes is the same as with the transport in the verticalplane with `planar_model` (Eq. (3.31)).

While `bmad_standard` tracking uses an averaged trajectory, the actual trajectory has oscillations that look like

$$x = A \cos(k_z z) \tag{3.34}$$

with the amplitude A given by

$$A = \frac{g_{\max}}{k_z^2} \tag{3.35}$$

The value of A, computed for an on-energy ($p_z = 0$) particle, is calculated and stored in the dependent parameter `osc_amplitude`.

With `field_calc` set to `planar_model` and `bmad_standard` tracking, the phase ϕ_z in Eqs. (3.29) is irrelevant. When the tracking involves Taylor maps and symplectic integration, the choice of phase is such that, with an integer number of periods, a particle that enters the wiggler on-axis will leave the wiggler on-axis provided there is an integer number of periods. Notice that with `field_calc` set to `helical_model` it is not possible to set the phase so that a particle that enters the wiggler on-axis will leave the wiggler on-axis.

When using a tracking through a periodic wiggler with a tracking method that integrates through the magnetic field (§5.4), The magnetic field is approximated using a single wiggler `term` as if the wiggler were a `map type` wiggler. This wiggler model has unphysical end effects and will give results that are different from the results obtained when using the `bmad_standard` tracking method.

Tracking a particle through a wiggler is always done so that if the particle starts on-axis with no momentum offsets, there is no change in the z coordinate even though the actual trajectory through the wiggler does not follow the straight line reference trajectory.

3.47.2 Map Type Wigglers

`Map type` wigglers are modeled using a field map as described in section §4.15. `Map type` wigglers have their `field_calc` parameter set to `fieldmap`. Note: For historic reasons, unlike other types of elements, `field_calc` will default to `fieldmap` if there is a field map present in a `wiggler`.

Unlike `periodic type` wigglers, the `b_max` attribute for a `map type wiggler` is a dependent attribute and is set by *Bmad* to be the maximum field on-axis computed for `polarity` = 1.

Note: There is no `bmad_standard` tracking for a `map_type wiggler`.

3.47.3 Old Wiggler Cartesian Map Syntax

When the wiggler model was first developed, the only type of map that could be used for `map type` wigglers was a Cartesian map (§4.15). The syntax for specifying this Cartesian map was different from what it is currently. The old syntax for a Cartiesian map term was:

$$\text{term(i)} = \{C, k_x, k_y, k_z, \phi_z\} \tag{3.36}$$

Example:

```
wig1: wiggler, l = 1.6,
      term(1) = {0.03, 3.00, 4.00, 5.00, 0.63},
      term(2) = ...
```

The old syntax was limited to using the `cartesian_map y` family (§15.5) with $x_0 = y_0 = 0$. There was also a different normalization convention. The old style `hyper-y` form was

$$\begin{aligned} B_x &= -C \frac{k_x}{k_y} \sin(k_x x) \sinh(k_y y) \cos(k_z z + \phi_z) \\ B_y &= \quad C \quad \cos(k_x x) \cosh(k_y y) \cos(k_z z + \phi_z) \qquad \text{! Old style} \\ B_s &= -C \frac{k_z}{k_y} \cos(k_x x) \sinh(k_y y) \sin(k_z z + \phi_z) \\ &\text{with } k_y^2 = k_x^2 + k_z^2 \ . \end{aligned} \tag{3.37}$$

The old style `hyper-xy` form was

$$\begin{aligned} B_x &= \quad C \frac{k_x}{k_y} \sinh(k_x x) \sinh(k_y y) \cos(k_z z + \phi_z) \\ B_y &= \quad C \quad \cosh(k_x x) \cosh(k_y y) \cos(k_z z + \phi_z) \qquad \text{! Old style} \\ B_s &= -C \frac{k_z}{k_y} \cosh(k_x x) \sinh(k_y y) \sin(k_z z + \phi_z) \\ &\text{with } k_y^2 = k_z^2 - k_x^2 \ , \end{aligned} \tag{3.38}$$

The old style `hyper_x` form was

$$\begin{aligned} B_x &= \quad C \frac{k_x}{k_y} \sinh(k_x x) \sin(k_y y) \cos(k_z z + \phi_z) \\ B_y &= \quad C \quad \cosh(k_x x) \cos(k_y y) \cos(k_z z + \phi_z) \qquad \text{! Old style} \\ B_s &= -C \frac{k_z}{k_y} \cosh(k_x x) \sin(k_y y) \sin(k_z z + \phi_z) \\ &\text{with } k_y^2 = k_x^2 - k_z^2 \ . \end{aligned} \tag{3.39}$$

The correspondence between C in the above equations and A in the new equations is given by comparing Eqs. (3.37), (3.38), and (3.39) with Eqs. (15.35).

When the cartesian_map construct was being developed, an intermediate hybrid syntax was used defined:

$$term(i) = \{A, k_x, k_y, k_z, x_0, y_0, \phi_z, \text{family}\} \tag{3.40}$$

The parameters here directly correspond to the `cartesian_map` forms (see Eqs. (15.32) through (15.37)).

For example, the old style syntax:

```
term(1) = {0.03*4/5, 3.00, 4.00, 5.00, 0.63 }    ! Old style
```

is equivalent to the hybrid syntax:

```
term(2) = {0.03, 3.00, 4.00, 5.00, 0, 0, 0.63, y}  ! Hybrid style
```

Note: When converting from the old or hybrid styles to the new syntax, the `field_calc` parameter must be set to `fieldmap`.

Chapter 4

Element Attributes

For a listing of element attributes for each type of element, see Chapter §13.

4.1 Dependent and Independent Attributes

For convenience, *Bmad* computes the values of some attributes based upon the values of other attributes. Some of these dependent variables are listed in Table 4.1. Also shown in Table 4.1 are the independent variables they are calculated from. In the table `n_part` and `l_lattice` (lattice length) are lattice attributes, not element attributes. The first two are set by the `parameter` statement (See §8.1). `l_lattice` is calculated when the lattice is read in.

Element	*Independent Variables*	*Dependent Variables*
All elements	`ds_step`	`num_steps`
`BeamBeam`	`charge`, `sig_x`, `sig_y`, `e_tot`, `n_part`	`bbi_constant`
`Elseparator`	`hkick`, `vkick`, `gap`, `l`, `e_tot`	`e_field`, `voltage`
`Lcavity`	`gradient`, `l`	`e_loss`, `voltage`
`Rbend`, `Sbend`	`g`, `l`	`rho`, `angle`, `l_chord`
`Wiggler` (map type)	`term(i)`	`b_max`, `k1`, `rho`
`Wiggler` (periodic type)	`b_max`, `e_tot`	`k1`, `rho`

Table 4.1: Partial listing of dependent variables and the independent variables they are calculated from.

For electric and magnetic field strength parameters, the `field_master` parameter (§4.2) can be used to determine if the be normalized or unnormalized, values are dependent or independent.

No attempt should be made to set or vary within a program dependent attributes. It should be remarked that this is not an iron clad rule. If a program properly bypasses *Bmad*'s attribute bookkeeping routine then anything is possible. In a lattice file, before lattice expansion (§2.23), *Bmad* allows the setting of a select group of dependent attributes if the appropriate independent attributes are not set. The list of settable dependent variables is given in Table 4.2. After reading in the lattice *Bmad* will set the appropriate independent variable based upon the value of the dependent variable. `harmon` is the exception in that it will never be set by the bookkeeping routine.

Element	*Dependent Variable Set*	*Independent Variables Not Set*
Lcavity	voltage	gradient
Rbend, Sbend	rho	g
Rbend, Sbend	angle	g, or l
RFcavity	rf_frequency	harmon
Wiggler (periodic type)	n_pole	l_pole

Table 4.2: Dependent variables that can be set in a primary lattice file.

4.2 Field_Master

The field_master attribute of an element sets whether the element's normalized (normalized by the reference energy) field strengths or the unnormalized strengths are the independent variables (§4.1). The setting of field_master also sets whether an element's magnetic multipoles (§4.14) are interpreted as normalized or unnormalized (electric multipoles are always treated as unnormalized).

Table 4.3 shows some normalized and unnormalized field strength attributes. The default value of field_master for an element is False if there are no field values set in the lattice file for that element. If normalized field values are present then the default is also False and if there are unnormalized field values present then the default is True.

For example:

```
Q1: quadrupole, b1_gradient = 0   ! Field strengths are the independent variables
Q1: quadrupole, field_master = T  ! Same as above
Q2: quadrupole        ! Define Q2.
Q2[b1_gradient] = 0   ! Field strengths now the independent variables.
Q2[field_master] = T  ! Same as above.
```

Specifying both normalized and unnormalized strengths for a given element is not permitted. For example:

```
Q3: quadrupole, k1 = 0.6, bl_hkick = 37.5  ! NO. Not VALID.
```

Element	*Normalized*	*Unnormalized*
Sbend, Rbend	g	b_field
Sbend, Rbend	g_err	b_field_err
Solenoid, Sol_quad	ks	bs_field
Quadrupole, Sol_quad, Sbend, Rbend	k1	b1_gradient
Sextupole, Sbend, Rbend	k2	b2_gradient
Octupole	k3	b3_gradient
HKicker, VKicker	kick	bl_kick
Most	hkick	bl_hkick
Most	vkick	bl_vkick

Table 4.3: Example normalized and unnormalized field strength attributes.

4.3 Type, Alias and Descrip Attributes

There are three string labels associated with any element:

```
type    = <String>
alias   = <String>
descrip = <String>
```

Bmad routines do not use these labels except when printing element information. `type` and `alias` can be up to 40 characters in length and `descrip` can be up to 200 characters. The attribute strings can be enclosed in double quotation marks ("). The attribute strings may contain blanks. If the attribute string does not contain a blank then the quotation marks may be omitted. In this case the first comma (,) or the end of the line marks the end of the string. Example:

```
Q00W: Quad, type = "My Type", alias = Who_knows, &
                                  descrip = "Only the shadow knows"
```

4.4 Syntax for Group and Overlay Elements

The syntax for specifying `group` (§3.21) and `overlay` (§3.36) elements are virtually identical and is discussed below. The name "`controller`" will be used to denote either a `group` or `overlay` element.

Controller elements have a set of one or more "`variables`" that are used to control the values of attributes of other elements (called "`slave`" attributes).

There are two types of controllers. `Expression` based controllers and `Spline` based controllers. `Expression` based controllers use mathematical expressions to define how slave attribute values are calculated based upon the values of the controller variables. `Spline` based controllers use a spline fit (in particular, the cubic non-smoothing Akima spline[Akima70]) to a function defined by a set of points (called "`knots`") to determine the relationship between variables and slave attributes. With a spline based controller, the number of variables is restricted to be one.

The general syntax for an expression based `group` element is

```
name: GROUP = {ele1[attrib1]:exp1, ele2[attrib2]:exp2, ...},
        VAR = {var1, var2, ...}, var1 = init_val1,
        old_var1 = old_init_val1, GANG = logical, ...
```

For an expression based `overlay` element the syntax is identical except OVERLAY is substituted for GROUP and there are no old values to set:

```
name: OVERLAY = {ele1[attrib1]:exp1, ele2[attrib2]:exp2, ...},
        VAR = {var1, var2, ...}, var1 = init_val1, GANG = logical, ...
```

`Name` is the name of the controller element, `ele1`, `ele2`, ... are the elements whose attributes are to be controlled, `attrib1`, `attrib2`, etc. are the controlled attributes (called "slave" attributes), `var1`, `var2`, etc. are the control variables, and `exp1`, `exp2`, etc. are the arithmetical expressions that define the relationship between the variables and the slave attributes. Example:

```
gr1: group = {q1[k1]:-tan(a)*b, q2[tilt]:b^2}, var = {a, b}
```

To define an expression based `controller` element, two lists are needed: One list defines the slave attributes along with the arithmetic expressions used for computing the value of the slave attributes. The other list defines the variables within the `controller` element that can be varied. In the above example, the variables are `a` and `b`, and the slave attributes are the `k1` attribute of element `q1` and the `tilt` attribute of element `q2`. The arithmetic expressions used for the control are `-tan(a)*b` and `b^2`.

The general syntax for For a `spline` based `group` element is

```
name: GROUP = {ele1[attrib1]:{y_knot_points1}, ele2[attrib2]:{y_knot_points2}, ...},
            VAR = {var1}, X_KNOT = {x_knot_points},
            var1 = init_val1, old_var1 = init_val_old1, ...
```

and the general syntax for a `spline` based `overlay` element is

```
name: OVERLAY = {ele1[attrib1]:{y_knot_points1}, ele2[attrib2]:{y_knot_points2}, ...},
            VAR = {var1}, X_KNOT = {x_knot_points}, var1 = init_val1, ...
```

Example:

```
ov: overlay = {q1[k1]:{0.234, 0.534}, q2[k1], q3[k1], q4[k1]:{0.375, 0.923}},
      var = time, x_knot = {0.0, 1.0}
```

The array sizes of all knot arrays must be the same. If a `y_knot_points` array is not present for a particular slave attribute, the knot array of the previous slave attribute is used. Thus, in the above example, the y_knot array used for `q2[k1]` and `q3[k1]` are the same as the y_knot array for `q1[k1]`.

If the `gang` attribute is `True` (which is the default) a `controller` will control all elements of a given name. Thus, in the above example, if there are multiple elements named `q1` then `gr1` will control the `k1` attribute of all of them. If `gang` is set to `False`, then a separate controller is created for each element in the lattice of a given name. For example:

```
gr1: group = {q1[k1]:-tan(a)*b, q2[tilt]:b^2}, var = {a, b}, gang = False
```

In this example, suppose there are five `q1` and five `q2` elements in the lattice. In this case, there will be five `gr1` group elements created. The first `gr1` will control the first `q1` and `q2` to appear in the lattice, etc. With `gang` set to `False`, it is an error if the number of instances in the lattice for a given slave name is different from any other slave name. In this example, it would be an error if the number of `q1` elements in the lattice is different from the number of `q2` elements in the lattice.

The syntax for specifying an attribute `attrib` of element `ele` to be controlled is `ele[attrib]`. The attribute part `[attrib]` may be omitted and in this case the name of the attribute will be taken to be the name of the first variable. Example:

```
ov1: overlay = {sex1:-tan(k2)^b}, var = {k2, b}
```

In this example, the controlled attribute of element `sex1` is `k2`. Except in cases where this default attribute syntax is used, the names of the variables are arbitrary and do not have to correspond to the name of any actual attribute.

The arithmetic expressions used to evaluate controlled attribute value changes may be a constant. In this case, the actual expression used is this constant times the first variable. If the expression is omitted entirely, along with the separating ":", the constant will be taken to be unity. Example:

```
gr1: group = {b1, b3:-pi}, var = {angle}
```

This is equivalent to

```
gr1: group = {b1[angle]:angle, b3[angle]:-pi*angle}, var = {angle}
```

Arithmetic expressions may themselves contain element attributes. Example:

```
sk_q20W: overlay = {sex_20W[a1]:-sex_20W[L]}, k1
```

Here the `sk_q20w` overlay controls the `a1` multipole attribute of element `sex_20w` and the length of `sex_20w` is used as a scale factor between the overlay's variable `k1` and the controlled attribute `a1`. The potential problem here is that, to keep the internal bookkeeping simple, the value of `sex_20w[L]` is evaluated once during parsing of the lattice file and never reevaluated (§2.12). If it is desired to use a variable element attribute in an expression, this may be effectively done by defining a control variable to take its place. Thus the above overlay may be recast as:

```
sk_q20W: overlay = {sex_20W[L]:ll, sex_20W[a1]:ll*k1}, var = {k1, ll}
```

Initial values can be assigned to the variables from within the definition of the controller element. Example:

```
ov1: overlay = {...}, var = {a, b}, a = 7, b = 2
```

Here the initial values 7 and 2 are assigned to `a` and `b` respectively. Alternatively, variables can be set after a controller element has been defined. Example:

```
ov1: overlay = {...}, var = {a, b}
ov1[a] = 7
gr1[b] = 2
```

There is an old deprecated syntax. For group elements the syntax was:

```
name: GROUP = {ele1[attrib1]:coef1, ele2[attrib2]:coef2, ...},
                      attrib = init_value  ! DO NOT USE THIS SYNTAX!
```

and for overlay elements the old syntax was identical except that GROUP was replaced by OVERLAY:

```
name: OVERLAY = {ele1[attrib1]:coef1, ele2[attrib2]:coef2, ...},
                      attrib = init_value  ! DO NOT USE THIS SYNTAX!
```

With this old syntax, there is only one variable. Additionally, there are no arithmetic expressions. Rather, attribute changes are linear in the command variable with the constant of proportionality given by a specified coefficient. For example with the old syntax

```
ov1: overlay = {sq1:3.7, sq2[tilt]}, k0 = 2  ! DO NOT USE THIS SYNTAX!
```

is equivalent, in the present syntax, to:

```
ov1: overlay = {sq[k0]:3.7, sq2[tilt]}, var = {k0}, k0 = 2
```

Note: In this old syntax the colon ":" separating the controlled attribute from the linear coefficient may be replaced by a slash "/". For group elements, there was an added wrinkle that, with the old syntax, the variable's name is fixed to be command. For example, with the old syntax

```
gr1: group = {sq1:3.7, sq2[tilt]}, k0 = 2  ! DO NOT USE THIS SYNTAX!
```

is equivalent, in the present syntax, to:

```
gr1: group = {sq[k0]:3.7, sq2[tilt]}, var = {command}, command = 2
```

4.5 Energy and Wavelength Attributes: E_tot, P0C, and Ref_Wavelength

The attributes that define the reference energy and momentum at an element are:

```
e_tot  = <Real>  ! Total energy in eV.
p0c    = <Real>  ! Momentum in eV.
```

The energy and momentum are defined at the exit end of the element. For ultra–relativistic particles, and for photons, these two values are the same (§14.4.2). Except for multipass elements (§7.2), e_tot and p0c are dependent attributes and, except for multipass elements, any setting of e_tot and p0c in the lattice input file is an error. The value of e_tot and p0c for an element is calculated by *Bmad* to be the same as the previous element except for e_gun, lcavity and patch elements. To set the e_tot or p0c at the start of the lattice use the beginning or parameter statements. See §8.1. Since the energy changes from the start to the end of an lcavity or. em_field, an lcavity or em_field has the dependent attributes

```
e_tot_start   and
p0c_start
```

which are just the reference energy and momentum at the start of the element.

The beginning_ele element (§3.4) also has associated e_tot_start and p0c_start attributes as well as e_tot and p0c. Generally, for an beginning_ele, p0c_start and p0c are the same and e_tot_start and e_tot are the same and the values for these attributes are set in the lattice file with the appropriate parameter (§8.1) or beginning (§8.4) statement. The exception occurs when there is an e_gun element in the lattice (§3.14). In this case, the p0c_start and e_tot_start attributes of the beginning_ele are set to the values as set in the lattice file and e_tot is set to

```
e_tot = e_tot_start + voltage
```

and p0c is calculated from e_tot and the mass of the particle being tracked. For example, if the lattice file contained:

```
beginning[p0c] = 0
gun: e_gun, voltage = 0.5e6
injector: line = (gun, ...)
```

Then the following energy values will be set for the beginning `beginning_ele` element:

```
p0c_start   = 0
e_tot_start = mc2
e_tot       = mc2 + 0.5e6
p0c         = Sqrt(e_tot - mc2^2)
```

where `mc2` is the particle rest mass. The reason for using this convoluted convention is to allow the setting, in the lattice file, of a zero reference momentum at the start of the lattice, while avoiding the calculational problems that would occur if the `e_gun` element truly had a starting reference momentum of zero. Specifically, the problem with zero reference momentum is that the phase space momentum would be infinity as can be seen from Eqs. (14.27).

For `multipass` elements, the reference energy is set by specifying one of `e_tot`, `p0c`, or `n_ref_pass` as described in §7.2.

For photons, the reference wavelength, `ref_wavelength` is also a dependent attribute calculated from the reference energy.

4.6 Orientation: Offset, Pitch, Tilt, and Roll Attributes

By default, an element, like a quadrupole, is aligned in space coincident with the reference orbit running through it (§14.1.2). A quadrupole can be displaced in space using the quadrupole's "`orientational`" attributes. For a quadrupole, the orientational attributes only affect the physical element and not the reference orbit. However, the orientational attributes of some other elements, like the `fiducial` element, do affect the reference orbit. To sort all this out, lattice elements can be divided into seven classes:

1. `Straight line elements` (§4.6.1)
   Straight line elements are elements where the reference orbit is a straight line. Examples include `quadrupoles`, and `sextupoles` as well as zero length elements like `markers`.

2. `Dipole bends` (§4.6.2)
   Dipole bends are:
   `sbend & rbend`

3. `Photon reflecting elements` (§4.6.3)
   The reflecting elements are
   `crystal`
   `mirror`
   `multilayer_mirror`
   These elements have a kink in the reference orbit at the nominal element surface.

4. `Reference orbit manipulator elements` (§4.6.4)
   Elements that are used to manipulate the reference orbit are
   `fork & photon_fork`
   `floor_shift`
   `patch`

5. `Fiducial Element` (§4.6.5)

6. `Girder Elements` (§4.6.6)

7. `Control Elements`
   Control elements are elements that control attributes of other elements. The control elements are:

   `group`
   `overlay`
   These elements do not have orientational attributes.

4.6.1 Straight Line Element Orientation

The straight line elements have the following orientational attributes:

```
x_offset = <Real>
y_offset = <Real>
z_offset = <Real>
x_pitch  = <Real>
y_pitch  = <Real>
tilt     = <Real>
```

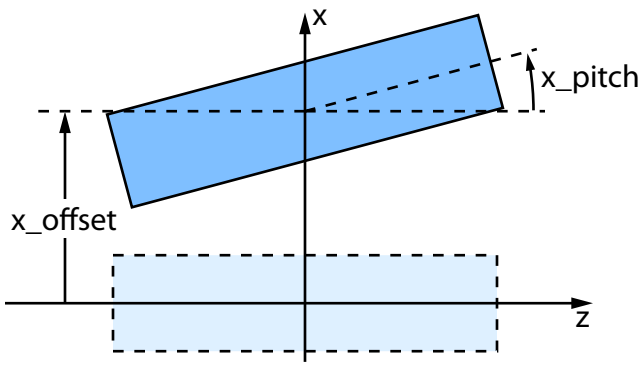


Figure 4.1: Geometry of Pitch and Offset attributes

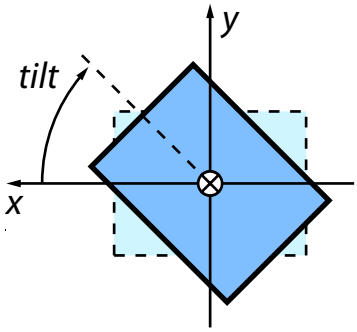


Figure 4.2: Geometry of a Tilt

For straight line elements the orientational attributes only shift the physical element and do not affect the reference orbit.

`x_offset` translates an element in the local x–direction as shown in Fig. 4.1. Similarly, `y_offset` and `z_offset` translate an element along the local y and z–directions respectively.

The `x_pitch` attribute rotates an element about the element's center such that with a positive `x_pitch` the exit face of the element is displaced in the $+x$–direction as shown in figure 4.1. [One way to visualize the effect of an `x_pitch` is to think of the element as an airplane pointing in the $+z$ direction. A positive `x_pitch` would then move the front of the plane in the $+x$–direction.] An`x_pitch` represents a rotation around the positive y-axis.

Similarly, the `y_pitch` attribute rotates an element about the element's center using the negative x–axis as the rotation axis so that, with a positive `y_pitch` the exit face of the element is displaced in the $+y$–direction.

Note: the `x_pitch` and `y_pitch` rotations are about the center of the element which is in contrast to the `dtheta` and `dphi` misalignments of *MAD* which rotate around the entrance point. The sense of the rotation between *Bmad* and MAD is:

```
x_pitch (Bmad) =  dtheta (MAD)
y_pitch (Bmad) = -dphi (MAD)
```

The tilt attribute rotates the element in the (x, y) plane as shown in figure 4.2. The rotation axis is the positive z-axis. For example

```
q1: quad, l = 0.6, x_offset = 0.03, y_pitch = 0.001, tilt
```

Like MAD, *Bmad* allows the use of the `tilt` attribute without a value to designate a skew element. The default tilt is $\pi/(2(n+1))$ where n is the order of the element:

```
sol_quad       n = 1
quadrupole     n = 1
sextupole      n = 2
octupole       n = 3
```

Note that `hkick` and `vkick` attributes are not affected by `tilt` except for `kicker` and `elseparator` elements.

4.6.2 Bend Element Orientation

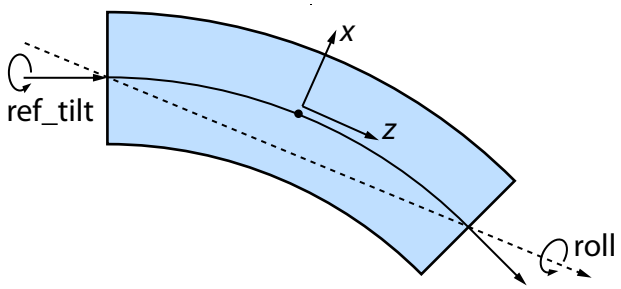


Figure 4.3: Geometry of a Bend. Like straight line elements, offsets and pitches are calculated with respect to the coordinates at the center of the bend. The exception is the `roll` attribute which is a rotation around the axis passing through the entrance and exit points. Shown here is the geometry for a bend with `ref_tilt` = 0. That is, the bend is in the $x - z$ plane.

The orientation attributes for `sbend` and `rbend` elements is

```
x_offset = <Real>
y_offset = <Real>
z_offset = <Real>
x_pitch  = <Real>
y_pitch  = <Real>
ref_tilt = <Real>    ! Shifts and reference orbit rotation axis.
roll     = <Real>
```

The geometry for orienting a bend is shown in Fig. 4.3. Like straight line elements, the offset and pitch attributes are evaluated with respect to the center of the element.

Unlike the straight line elements, bends do not have a `tilt` attribute. Rather they have a `ref_tilt` and a `roll` attribute. The `roll` attribute rotates the bend along an axis that runs through the entrance point and exit point as shown in figure 4.3. A `roll` attribute, like the offset and pitch attributes does not affect the reference orbit. The major effect of a `roll` is to give a vertical kick to the beam. For a bend with positive bend angle, a positive `roll` will move the outside portion ($+x$ side) of the bend upward and the inside portion (-x side) downward. Much like car racetracks which are typically slanted towards the inside of a turn.

The `ref_tilt` attribute of a bend rotates the bend about the z axis at the upstream end of the bend as shown in Fig. 4.3. Unlike `rolls` and `tilts`, `ref_tilt` also shifts the rotation axis of the reference orbit along with the physical element. A `bend` with a `ref_tilt` of $\pi/2$ will bend a beam vertically downward (§14.2). Note that the `ref_tilt` attribute of *Bmad* is the same as the MAD `tilt` attribute.

4.6.3 Photon Reflecting Element Orientation

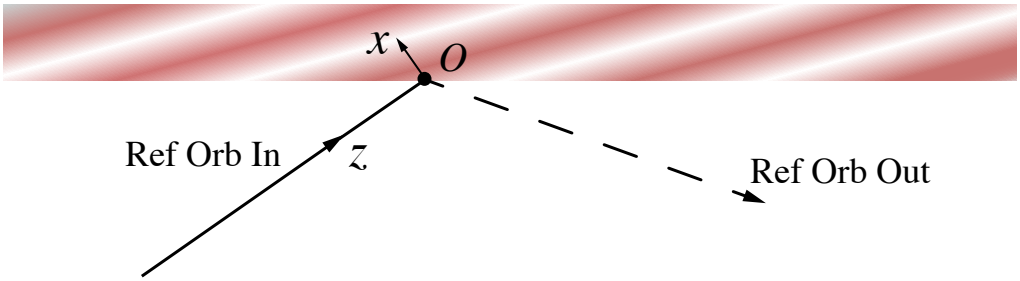


Figure 4.4: Geometry of a photon reflecting element orientation. The reference coordinates used for defining the orientational attribute is the entrance reference coordinates.

Photon reflecting elements have the following orientational attributes:

```
x_offset = <Real>
y_offset = <Real>
z_offset = <Real>
x_pitch  = <Real>
y_pitch  = <Real>
ref_tilt = <Real>    ! Shifts both element and reference orbit.
tilt     = <Real>
```

Roughly, these elements can be viewed as zero length bends except, since there is no center position, the orientational attributes are defined with respect to the entrance coordinates as shown in Fig. 4.4. Like bend elements, the `ref_tilt` attribute rotates both the physical element and the reference coordinates. The `tilt` attribute rotates just the physical element. Thus the total rotation of the physical element about the entrance z axis is the sum `tilt` + `ref_tilt`.

Frequently, it is desired to orient reflecting elements with respect to the element's surface. This can be done using a `girder` element (§3.20) which supports the reflecting element and with the `girder`'s `origin_ele_ref_pt` attribute set to `center`.

4.6.4 Reference Orbit Manipulator Element Orientation

The `fork`, `photon_fork`, `floor_shift`, and `patch` elements use the following attributes to orient their exit edge with respect to their entrance edge:

```
x_offset = <Real>
y_offset = <Real>
z_offset = <Real>
x_pitch  = <Real>
y_pitch  = <Real>
tilt     = <Real>
```

Here "exit" edge for `fork` and `photon_fork` elements is defined to be the start of the line being branched to. [Within the line containing the fork, the `fork` element is considered to have zero length so the exit face in the line containing the fork is coincident with the entrance face.] The placement of the exit edge for these elements defines the reference orbit. Thus, unlike the corresponding attributes for other elements, the orientational attributes here directly control the reference orbit.

4.6.5 Fiducial Element Orientation

The `fiducial` element (§3.20) uses the following attributes to define its position:

```
origin_ele        = <Name>     ! Reference element.
origin_ele_ref_pt = <location> ! Reference pt on reference ele.
dx_origin         = <Real>     ! x-position offset
dy_origin         = <Real>     ! y-position offset
dz_origin         = <Real>     ! z-position offset
dtheta_origin     = <Real>     ! orientation angle offset.
dphi_origin       = <Real>     ! orientation angle offset.
dpsi_origin       = <Real>     ! orientation angle offset.
```

See Section §3.17 for more details.

4.6.6 Girder Orientation

A `girder` (§3.20) element uses the same attributes as a `fiducial` element (§3.17) to orient the reference girder position. In addition, the following attributes are used to move the girder physically from the reference position:

```
x_offset = <Real>
y_offset = <Real>
z_offset = <Real>
x_pitch  = <Real>
y_pitch  = <Real>
tilt     = <Real>
```

Shifting the girder from its reference position shifts all the elements that are supported by the girder. See Section §3.20 for more details.

If an element is supported by a `girder` element (§3.20), the orientational attributes of the element are with respect to the orientation of the `girder`. The computed offsets, pitches and tilt with respect to the local reference coordinates are stored in the dependent attributes

```
x_offset_tot
y_offset_tot
z_offset_tot
x_pitch_tot
y_pitch_tot
tilt_tot
roll_tot
```

A `*_tot` attribute will only be present if the corresponding non `*_tot` attribute is present. For example, only `sbend` and `rbend` elements have a `roll_tot` attribute since only these elements have a `roll` attribute.

If an element is not supported by a `girder`, the values of the `*_tot` attributes will be the same value as the values of the corresponding non `*_tot` attributes.

4.7 Hkick, Vkick, and Kick Attributes

The kick attributes that an element may have are:

```
kick,  bl_kick  = <Real>  ! Used only with a Hkicker or Vkicker
hkick, bl_hkick = <Real>
vkick, bl_vkick = <Real>
```

`kick`, `hkick`, and `vkick` attributes are the integrated kick of an element in radians. `kick` is only used for `hkicker` and `vkicker` elements. All other elements that can kick use `hkick` and `vkick`. The `tilt` attribute will only rotate a kick for `hkicker`, `vkicker`, `elseparator` and `kicker` elements. This rule was implemented so that, for example, the `hkick` attribute for a skew quadrupole would represent a horizontal steering. The `bl_kick`, `bl_hkick`, and `bl_vkick` attributes are the integrated field kick in `meters-Tesla`. Normally these are dependent attributes except if they appear in the lattice file (§4.1).

For an `elseparator` element, the `hkick` and `vkick` are appropriate for a positively charged particle. The kick for a negatively charged particle is opposite this.

4.8 Aperture and Limit Attributes

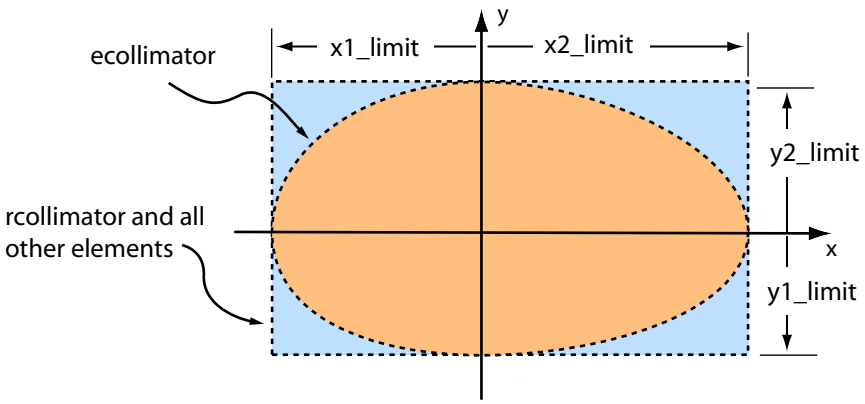


Figure 4.5: Apertures for ecollimator and rcollimator elements. [note: positive z points up, out of the page.] As drawn, all limits `x1_limit`, `x2_limit`, `y1_limit`, `y2_limit` are positive.

The aperture attributes are:

```
x1_limit      = <Real>      ! Horizontal, negative side, aperture limit
x2_limit      = <Real>      ! Horizontal, positive side, aperture limit
y1_limit      = <Real>      ! Vertical, negative side, aperture limit
y2_limit      = <Real>      ! Vertical, positive side, aperture limit
x_limit       = <Real>      ! Alternative to specifying x1_limit and x2_limit
y_limit       = <Real>      ! Alternative to specifying y1_limit and y2_limit
aperture      = <Real>      ! Alternative to specifying x_limit and y_limit
aperture_at   = <Switch>    ! What end aperture is at. (§4.8.2)
aperture_type = <Switch>    ! What type of aperture it is
offset_moves_aperture = <Logical> ! Element offsets affect aperture position (§4.8.1)
```

`x1_limit`, `x2_limit`, `y1_limit`, and `y2_limit` specify the half–width of the aperture of an element as shown in figure 4.5. A zero `x1_limit`, `x2_limit`, `y1_limit`, or `y2_limit` is interpreted as no aperture in the appropriate plane.

For convenience, `x_limit` can be used to set `x1_limit` and `x2_limit` to a common value. Example:

```
s: sextupole, x1_limit = 0.09, x2_limit = 0.09
s: sextupole, x_limit = 0.09   ! Same as above
```

Similarly, `y_limit` can be used to set `y1_limit` and `y2_limit`. The `aperture` attribute can be use to set all four `x1_limit`, `x2_limit`, `y1_limit` and `y2_limit` to a common value. Internally, the *Bmad* code does *not* store `x_limit`, `y_limit`, or `aperture`. This means that using `x_limit`, `y_limit` or aperture in arithmetic expressions is an error:

```
q1: quad, aperture = 0.09
q2: quad, aperture = q1[aperture]   ! THIS IS AN ERROR!
q2: quad, aperture = q1[x1_limit]   ! Correct
```

By default, apertures are assumed to be rectangular except that an `ecollimator` has a elliptical aperture. This can be changed by setting the `aperture_type` attribute. The possible values of this attribute are:

```
auto         ! Default for detector, mask and diffraction_plate elements
custom
elliptical   ! Default for ecollimator elements.
rectangular  ! Default for most elements.
wall3d       ! Vacuum chamber wall (§4.11).
```

The `custom` setting is used in the case where programs have been compiled with custom, non-Bmad, code to handle the aperture calculation. The `auto` setting is used for automatic calculation of a rectangular aperture. For `diffraction_plate` and `mask` elements, the `auto` setting causes the four aperture limits to be set to just cover the clear area of element (§4.11.6). For all other elements, the `auto` setting is only to be used when there is an associated surface grid (§4.10.1) for the element and, in this case, *Bmad* to set the four limits to just cover the surface grid.

The `wall3d` setting uses the vacuum chamber wall as specified by a `wall` attribute (§4.11). Using the `wall` construct allows for complex apertures to be constructed. Note that The wall thickness and material type are not used when calculating if a particle has hit the wall. That is, the wall is considered to be infinitely thin. Also note that a wall must cover the entire length of the element longitudinally. This is done in order to be able to spot errors in specifying the wall geometry.

For rectangular apertures, the limits `x1_limit`, `x2_limit`, `y1_limit`, or `y2_limit` may be negative. For example:

```
s: sextupole, x1_limit = -0.02, x2_limit = 0.09
```

In this case, particles will hit the aperture if their x-coordinate is outside the interval [0.02, 0.09]. That is, particles at the origin will be lost.

To avoid numerical overflow and other errors in tracking, a particle will be considered to have hit an aperture in an element, even if there are no apertures set for that element, if its orbit exceeds 1000 meters. Additionally, there are other situations where a particle will be considered lost. For example, if a particle's trajectory does not intersect the output face in a bend.

Examples:

```
q1, quadrupole, y1_limit = 0.03
q1[y2_limit] = 0.03
q1[y_limit] = 0.03  ! equivalent to the proceeding 2 lines.
q1[aperture_at] = both_ends
```

4.8.1 Apertures and Element Offsets

Normally, whether a particle hits an aperture or not is evaluated independent of any element offsets (§4.6). This is equivalent to the situation where a beam pipe containing an aperture is independent of the placement of the physical element the beam pipe passes through. That is, the beam pipe does not "touch" the physical element. This can be changed by setting the `offset_moves_aperture` attribute to `True`. In this case any offsets or pitches will be considered to have shifted the aperture boundary. The exceptions here is that the default for the following elements is for `offset_moves_aperture` to be `True`:

```
rcollimator,
ecollimator,
```

```
multilayer_mirror,
mirror, and
crystal
```

Even with `offset_moves_aperture` set to `True`, `tilt`s will not affect the aperture calculation. This is done, for example, so that the tilt of a skew quadrupole does not affect the aperture. The exception here is that tilting an `rcollimator` or `ecollimator` element will tilt the aperture. Additionally, when the aperture is at the `surface` (see below), any `tilt` will be used in the calculation.

Example:

```
q1: quad, l = 0.6, x1_limit = 0.045, offset_moves_aperture = T
```

4.8.2 Aperture Placement

By default, for most elements, the aperture is evaluated at the exit face of the element. This can be changed by setting the `aperture_at` attribute. Possible settings for `aperture_at` are:

```
both_ends
continuous
entrance_end
exit_end       ! Default for most elements
no_aperture
surface
wall_transition
```

The `exit_end` setting is the default for most elements except for the following elements who have a default of `surface`:

```
crystal
diffraction_plate
mask
mirror
multilayer_mirror
sample
```

In fact, for the following elements:

```
mirror,
multilayer_mirror
crystal
```

The `surface` setting for `aperture_at` must be used. Additionally, due to the complicated geometry of these elements, to keep things conceptionally simple, the rule is imposed that, for an aperture at the surface, the `offset_moves_aperture` setting must be left in its default state of True. Additionally, For `entrance_end` or `exit_end` apertures, `offset_moves_aperture` must be set to False.

Note: The entrance and exit ends of an element are independent of which direction particles are tracked through an element. Thus if a particle is tracked backwards it enters an element at the "exit end" and exits at the "entrance end". The `continuous` setting indicates that the aperture is continuous along the length of the element. This only matters when particle tracking involves stepping through an element a little bit at a time. For example, as in Runge-Kutta tracking (§5.1). For tracking where a formula is used to transform the particle coordinates at the entrance of an element to the coordinates at the exit end, the aperture is only checked at the end points so, in this situation, a `continuous` aperture is equivalent to the `both_ends` setting.

The `wall_transition` setting is like the `continuous` setting in that the aperture boundary is considered to be continuous along the element's length. However, unlike the `continuous` setting, with the

`wall_transition` setting a particle outside the wall is considered alive and it is only when a particle moves through the wall that it is lost. The `wall_transition` setting is used for things like septum magnets where a particle may be safely outside or inside the wall. Note to programmers: By supplying a custom `wall_hit_handler_custom` routine, scattering of particles through a wall may be simulated.

Examples:

```
q2: quad, aperture_type = elliptical, aperture_at = continuous
q1: quad, l = 0.6, x1_limit = 0.045, offset_moves_aperture = T
```

4.8.3 Apertures and X-Ray Generation

With X-ray simulation apertures can be used by *Bmad* to limit the directions in which photons are generated. This can greatly decrease simulation times. For example, a photon passing through a `diffraction_plate` element will diffract in an arbitrary direction. If a *downstream* element has an aperture set, *Bmad* can restrict the velocity directions so that the photons will fill the downstream aperture and the amount of time wasted tracking photons that ultimately would be collimated is minimal.

4.9 X-Rays Crystal & Compound Materials

For basic crystallographic and X-ray matter interaction cross-sections, *Bmad* uses the XRAYLIB[Schoon11] library. Crystal structure parameters in XRAYLIB are mainly from R. W. G. Wyckoff[Wyckoff65] with some structure parameters coming from NIST. The list of available structures is:

AlphaAlumina	GaP	KCl	Platinum
AlphaQuartz	GaSb	KTP	RbAP
Aluminum	Ge	LaB6	Sapphire
Be	Gold	LaB6_NIST	Si
Beryl	Graphite	LiF	Si_NIST
Copper	InAs	LiNbO3	Si2
CsCl	InP	Muscovite	SiC
CsF	InSb	NaCl	Titanium
Diamond	Iron	PET	TlAP
GaAs	KAP		

These names are case sensitive

Besides the above crystal list, *Bmad* can calculate structure factors for all the elements and the following list of materials. Material properties are from NIST. These names are case sensitive. That is, the NIST materials all use upper case. As noted in the table, several of the materials may be specified using the appropriate chemical formula. For example, liquid water may be referenced using the name `H2O`.

A_150_TISSUE_EQUIVALENT_PLASTIC
ACETONE
ACETYLENE
ADENINE
ADIPOSE_TISSUE_ICRP
AIR_DRY_NEAR_SEA_LEVEL
ALANINE
ALUMINUM_OXIDE, Al2O3
AMBER
LITHIUM_TETRABORATE
LUNG_ICRP
M3_WAX
MAGNESIUM_CARBONATE
MAGNESIUM_FLUORIDE
MAGNESIUM_OXIDE
MAGNESIUM_TETRABORATE
MERCURIC_IODIDE
METHANE

Continued on next page

AMMONIA, NH3
ANILINE
ANTHRACENE
B_100_BONE_EQUIVALENT_PLASTIC
BAKELITE
BARIUM_FLUORIDE
BARIUM_SULFATE
BENZENE, C6H6
BERYLLIUM_OXIDE
BISMUTH_GERMANIUM_OXIDE
BLOOD_ICRP
BONE_COMPACT_ICRU
BONE_CORTICAL_ICRP
BORON_CARBIDE, B4C
BORON_OXIDE, B2O3
BRAIN_ICRP
BUTANE
N_BUTYL_ALCOHOL
C_552_AIR_EQUIVALENT_PLASTIC
CADMIUM_TELLURIDE
CADMIUM_TUNGSTATE
CALCIUM_CARBONATE
CALCIUM_FLUORIDE
CALCIUM_OXIDE
CALCIUM_SULFATE
CALCIUM_TUNGSTATE
CARBON_DIOXIDE
CARBON_TETRACHLORIDE
CELLULOSE_ACETATE_CELLOPHANE
CELLULOSE_ACETATE_BUTYRATE
CELLULOSE_NITRATE
CERIC_SULFATE_DOSIMETER_SOLUTION
CESIUM_FLUORIDE
CESIUM_IODIDE
CHLOROBENZENE
CHLOROFORM
CONCRETE_PORTLAND
CYCLOHEXANE
12_DDIHLOROBENZENE
DICHLORODIETHYL_ETHER
12_DICHLOROETHANE
DIETHYL_ETHER
NN_DIMETHYL_FORMAMIDE
DIMETHYL_SULFOXIDE
ETHANE
ETHYL_ALCOHOL
ETHYL_CELLULOSE
ETHYLENE
EYE_LENS_ICRP
FERRIC_OXIDE
FERROBORIDE
FERROUS_OXIDE
FERROUS_SULFATE_DOSIMETER_SOLUTION
FREON_12
FREON_12B2
FREON_13
FREON_13B1
FREON_13I1
GADOLINIUM_OXYSULFIDE
GALLIUM_ARSENIDE
GEL_IN_PHOTOGRAPHIC_EMULSION
GLASS_PYREX
GLASS_LEAD
METHANOL
MIX_D_WAX
MS20_TISSUE_SUBSTITUTE
MUSCLE_SKELETAL
MUSCLE_STRIATED
MUSCLE_EQUIVALENT_LIQUID_WITH_SUCROSE
MUSCLE_EQUIVALENT_LIQUID_WITHOUT_SUCROSE
NAPHTHALENE
NITROBENZENE
NITROUS_OXIDE
NYLON_DU_PONT_ELVAMIDE_8062
NYLON_TYPE_6_AND_TYPE_6_6
NYLON_TYPE_6_10
NYLON_TYPE_11_RILSAN
OCTANE_LIQUID
PARAFFIN_WAX
N_PENTANE
PHOTOGRAPHIC_EMULSION
PLASTIC_SCINTILLATOR_VINYLTOLUENE_BASED
PLUTONIUM_DIOXIDE
POLYACRYLONITRILE
POLYCARBONATE_MAKROLON_LEXAN
POLYCHLOROSTYRENE
POLYETHYLENE
POLYETHYLENE_TEREPHTHALATE_MYLAR
POLYMETHYL_METHACRALATE_LUCITE_PERSPEX
POLYOXYMETHYLENE
POLYPROPYLENE
POLYSTYRENE
POLYTETRAFLUOROETHYLENE_TEFLON
POLYTRIFLUOROCHLOROETHYLENE
POLYVINYL_ACETATE
POLYVINYL_ALCOHOL
POLYVINYL_BUTYRAL
POLYVINYL_CHLORIDE
POLYVINYLIDENE_CHLORIDE_SARAN
POLYVINYLIDENE_FLUORIDE
POLYVINYL_PYRROLIDONE
POTASSIUM_IODIDE
POTASSIUM_OXIDE
PROPANE
PROPANE_LIQUID
N_PROPYL_ALCOHOL
PYRIDINE
RUBBER_BUTYL
RUBBER_NATURAL
RUBBER_NEOPRENE
SILICON_DIOXIDE
SILVER_BROMIDE
SILVER_CHLORIDE
SILVER_HALIDES_IN_PHOTOGRAPHIC_EMULSION
SILVER_IODIDE
SKIN_ICRP
SODIUM_CARBONATE
SODIUM_IODIDE
SODIUM_MONOXIDE
SODIUM_NITRATE
STILBENE
SUCROSE
TERPHENYL
TESTES_ICRP
TETRACHLOROETHYLENE
THALLIUM_CHLORIDE

Continued on next page

GLASS_PLATE
GLUCOSE
GLUTAMINE
GLYCEROL
GUANINE
GYPSUM_PLASTER_OF_PARIS
N_HEPTANE
N_HEXANE
KAPTON_POLYIMIDE_FILM
LANTHANUM_OXYBROMIDE
LANTHANUM_OXYSULFIDE
LEAD_OXIDE
LITHIUM_AMIDE
LITHIUM_CARBONATE
LITHIUM_FLUORIDE
LITHIUM_HYDRIDE
LITHIUM_IODIDE
LITHIUM_OXIDE
TISSUE_SOFT_ICRP
TISSUE_SOFT_ICRU_FOUR_COMPONENT
TISSUE_EQUIVALENT_GAS_METHANE_BASED
TISSUE_EQUIVALENT_GAS_PROPANE_BASED
TITANIUM_DIOXIDE
TOLUENE
TRICHLOROETHYLENE
TRIETHYL_PHOSPHATE
TUNGSTEN_HEXAFLUORIDE
URANIUM_DICARBIDE
URANIUM_MONOCARBIDE
URANIUM_OXIDE
UREA
VALINE
VITON_FLUOROELASTOMER
WATER_LIQUID, H2O
WATER_VAPOR
XYLENE

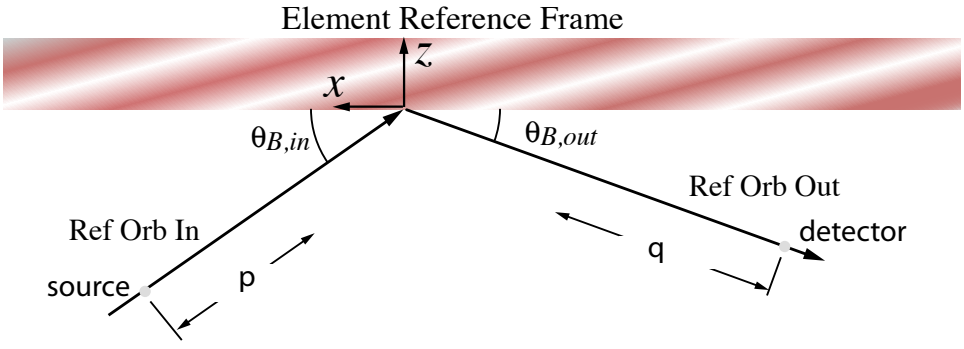


Figure 4.6: Surface curvature geometry. The element reference frame used to describe surface curvature has the z axis pointing towards the interior of the element, and the x axis in the plane defined by the entrance and exit reference orbit.

4.10 Surface Properties for X-Ray elements

The following X-ray elements have a surface which X-rays impinge upon:

`crystal`	§3.8
`detector`	§3.11
`diffraction_plate`	§3.12
`mask`	§3.29
`mirror, and`	§3.31
`multilayer_mirror`	§3.33
`sample`	§3.42

[There is also the `capillary` element but this element specifies its surface differently.]

The coordinate system used for characterizing the curvature of a surface is the element reference frame as shown in Fig. 4.6). This coordinate system has the z axis pointing towards the interior of the element, and the x axis in the plane defined by the entrance and exit reference orbit. In this coordinate system, the surface is an ellipsoid plus a fourth order polynomial in x and y plus a possible "figure error" contribution $z_{\rm fig}$ defined by a surface grid:

$$-z = \frac{1}{g_z}\left[1-\sqrt{1-(g_x\,x)^2-(g_y\,y)^2}\right]+ \frac{1}{g_{sp}}\left[1-\sqrt{1-(g_{sp}\,x)^2-(g_{sp}\,y)^2}\right]+\sum_{2\le i+j\le 6} c_{ij}\,x^i\,y^j - z_{\rm fig} \tag{4.1}$$

$z_{\rm fig}$ is discussed in section §4.10.1 and is only present when the surface grid `type` is set to `Figure_Error`. In Eq. (4.1) the c_{ij} coefficients parameterize the fourth order polynomial, g_{sp} parameterize the spherical curvature, and g_x, g_y, and g_z parameterize ellipsoid curvature. [In principle, the spherical curvature is not needed since the elliptical curvature is more general. In practice, it is sometimes convieneint to be able to specify spherical curvature.] If g_z is zero, the elliptical curvature is ignored. If g_z or g_{sp} is positive, the curvature is concave towards the incoming photon. If negative the curvature is convex. The spherical and ellipsoid parameters are set for an element by setting the element parameters

```
spherical_curvature    = <Real>   ! g_sp
elliptical_curvature_x = <Real>   ! g_x
elliptical_curvature_y = <Real>   ! g_y
elliptical_curvature_z = <Real>   ! g_z
```

The polynomial coefficients c_{ij} are set in the lattice file by setting the element attributes

```
curvature_xM_yN = <Real>
```

where `M` and `N` are integers in the range 0 through 6 with the restriction

```
2 ≤ M + N ≤ 6
```

Example:

```
c2: crystal, spherical_curvature = 1/4.7, curvature_x2_y0 = 0.37, ...
```

in this example, `curvature_x2_y0` corresponds to the c_{20} term in Eq. (4.1). To get the effect of a nonzero $x^0\,y^0$, $x^1\,y^0$, or $x^0\,y^1$ terms (since corresponding `curvature_xN_yM` are not permitted), element offsets and pitches can be used (§4.6).

Some useful formulas: Series expansion for a sphere of radius R:

$$-z = \frac{x^2}{2\,R}+\frac{x^4}{8\,R^3}+\frac{x^6}{16\,R^5}+\frac{y^2}{2\,R}+\frac{y^4}{8\,R^3}+\frac{y^6}{16\,R^5}+\frac{x^2\,y^2}{4\,R^3}+\frac{3\,x^4\,y^2}{16\,R^5}+\frac{3\,x^2\,y^4}{16\,R^5} \tag{4.2}$$

If p is the distance from the source to the crystal, and q is the distance from the crystal to the detector, the radius of the crystal R_s in the sagittal (transverse) plane is given by[Rio98]

$$\frac{1}{p}+\frac{1}{q} = \frac{\sin\theta_{g,in}+\sin\theta_{g,out}}{R_s} \tag{4.3}$$

where $\theta_{g,in}$ and $\theta_{g,out}$ are the entrance and exit graze angles. In the tangential (meridional) plane, the radius R_t needed for foucusing is

$$\frac{\sin^2\theta_{g,in}}{p} + \frac{\sin^2\theta_{g,out}}{q} = \frac{\sin\theta_{g,in} + \sin\theta_{g,out}}{R_t} \tag{4.4}$$

The above formulas assume that the crystal is constructed so that the orientation of the Bragg planes follows the orientation of the surface. Mirrors have similar formulas with $\theta_{g,in} = \theta_{g_out} = \theta$.

Example:
```
t_bragg = 1.3950647
rt = 1  ! Crystal transverse radius
rs = rt*(sin(t_bragg))^2
c: crystal, crystal_type =  'Si(553)', b_param = -1,
      curvature_x0_y2 =  1 / (2 * rs), curvature_x0_y4 = 1 / (8 * rs^3),
      curvature_x2_y0 = 1 / (2 * rt), curvature_x4_y0 = 1 / (8 * rt^3),
```

4.10.1 Surface Grid

A surface can be broken up into a grid of rectangles. This is useful, for example, in breaking up a `detector` element into pixel photo receptors or in simulating a rough serface for `crystal`s and other elements. The general syntax is:
```
surface = {
  grid = {
    type = <type_name>,                ! Off, Segmented, Figure_Error, or H_Misalign
    ix_bounds = (<ix_min>, <ix_max>),  ! Min/max index bounds in x-direction
    iy_bounds = (<iy_min>, <iy_max>),  ! Min/max index bounds in y-direction
    r0 = (<x0>, <y0>),                 ! (x,y) coordinates at grid origin
    dr = (<dx>, <dy>),                 ! width and height of pixels.
    ! Grid points used with type = H_Misalign:
    pt(<i>,<j>) = (<dz_dx>, <dz_dy>, <dz_dx_rms>, <dz_dy_rms>)
    ! Grid points used with type = Figure_Error:
    pt(<i>,<j>) = (<z>, <dz_dx>, <dz_dy>),    ! or
    pt(<i>,<j>) = <z>,

      } }
```
Example:
```
ccd: crystal, surface = {
       grid = {
         type = h_misalign,
         r0 = (0.0, 0.01), dr = (0.005, 0.005),
         ix_bounds = (1, 57), iy_bounds = (-30, 10),
         pt(1,-30) = (0.001, -0.002, 0, 0),
         pt(1,-29) = ...,
       } }
```
The grid is a two dimensional with bounds given by the `ix_bounds` and `iy_bounds` components. These two components must be present. In the above example the grid is 57 pixels in x and 41 pixels in y.

The physical placement of the grid on the element is determined by the `r0` and `dr` components. `r0` is optional and gives the (x, y) coordinates of the center of the pixel with index $(0, 0)$. The `dr` component, which must be present, gives the pixal width and height. Thus the center of the (i, j) pixel is:
```
(x,y) = (r0(1), r0(2)) + (i*dr(1), j*dr(2))
```

There are several types of grids. What type of grid is determined by the `type` component Possible `type` values are:

```
Figure_Error       ! Mesh defines the surface figure error.
H_Misalign         ! Misalignment of crystal H vector
Off                ! Ignore grid
Segmented          ! Surface is a matrix of flat rectangles
```

Figure_Error

With the `type` parameter set to `Figure_Error`, a figure error, $z_{\rm fig}$ is added to the surface curvature as shown in Eq. (4.1). $z_{\rm fig}$ is determined by a spline interpolation of the z values of the grid of points defined by `pt(<i>,<j>)`. For each point `pt(<i>,<j>)`, the z value along with the $dz_d x$ and $dz_d y$ slopes can be specified or only the z value needs to be specified. With the later option the slopes will be computed by taking finite differences of nearest neighbors.

H_Misalign

A setting of the `type` attribute to `H_Misalign` is used with crystals only. With `H_Misalign`, the grid defines misalignment of the H vector which is the normal to the diffracting planes of the crystal (§23.4). When using `H_Misalign`, each `pt(i,j)` component gives the misalignment of H for the corresponding pixel. For an individual photon, the misalignt of H will be

```
dz_dx_tot = <dz_dx> + r1 * <dz_dx_rms>
dz_dy_tot = <dz_dy> + r2 * <dz_dy_rms>
```

where `dz_dx_tot` and `dz_dy_tot` are the rotational misalignment (§4.6) used in the calculation, the quantities in brackets `<...>` are components of `pt`, and `r1` and `r2` are Gaussian distributed random numbers with unit rms. These random numbers are regenerated for each photon. Note: `pt` is only used with `H_Misalign`.

Off

When the `type` component is set to `Off`, the grid will not be used.

Segmented

When the `type` component is set to `Segmented`, the crystal surface is modeled as a grid of flat "rectangles" (the actual shape is very close but not quite rectangular). Using a segmented surface only makes sense when the surface is curved (see Eq. (4.1)). There is one rectangle for each pixel. Each rectangle has an extent in the (x, y) transverse dimensions equal to the extent of the corresponding pixel. Eq. (4.1) is used to calculate the z coordinate of the vertices of a given rectangle and then these z values are adjusted so that

```
1) The rectangle is flat (the verticies all lie on a plane).
2) The rectangle contacts the unsegmented surface (Eq. (4.1)) at two diagonally opposite vertices
3) The other two diagonaly opposite vertices will be as close as possible in the
   least squares sense from the unsegmented surface.
```

Note: The `pt` component is not used here.

4.11 Walls: Vacuum Chamber, Capillary and Mask

The `wall` attribute for an element is used to define:

```
vacuum chamber wall
capillary element (§3.6) inside wall
diffraction_plate (§3.12) geometry
```

The topics of the following subsections are:

4.11.1 Wall Syntax

The syntax of the `wall` attribute is:

```
wall = {
  superimpose = <T/F>,                ! Chamber wall only
  thickness = <real>                  ! Default thickness.
  opaque_material = <material_type>   ! Default opaque material.
  clear_material = <material_type>    ! Default clear material.
  section = {
    type = <section_type>,            ! Chamber Mask, and Diffraction_plate only
    s = <longitudinal_position>,      ! Relative to beginning of element.
    r0 = (<x0>, <y0>),                ! section (x,y) origin
    absolute_vertices = <T/F>,        ! Vertex nums: abs or relative to r0? Default = F.
    material = <material_type>,       ! Mask and Diffraction_plate only.
    thickness = <real>,               ! Mask and Diffraction_plate only.
    dr_ds = <value>,                  ! Capillary and Chamber only
    v(1) = {<x>, <y>, <radius_x>, <radius_y>, <tilt>},
    v(2) = { ... },
    ...},
  section = {
    s = <longitudinal_position>,
    v(1) = {... },
    ... },
  ... }
```

A `wall` begins with "`wall = {`" and ends with a "`}`". In between are a number of individual cross-section structures. Each individual cross-section begins with "`section = {`" and ends with a "`}`". The `s` parameter of a cross-section gives the longitudinal position of the cross-section. Example:

```
this_cap: capillary,
  wall = {
    section = { ! cross-section with top/bottom symmetry
      s = 0, v(1) =  {0.02, 0.00},
      v(2) = {0.00, 0.02, 0.02}, v(3) = {-0.01, 0.01} },
    section = {  ! Cross-section that is a tilted ellipse.
      s = 0.34,
      v(1) = {0.003, -0.001, 0.015, 0.008, 0.2*pi} } }
```

In this example an element called `this_cap` is a `capillary` whose wall is defined by two cross-sections.

4.11.2 Wall Sections

The wall is defined by a number of cross-sectional slices. For Fig. 4.7A shows the geometry for `capillary` or vacuum chamber walls. Each cross-section is defined by a longitudinal position s relative to the beginning of the element and a number of vertices. The verticies are defined with respect to the local

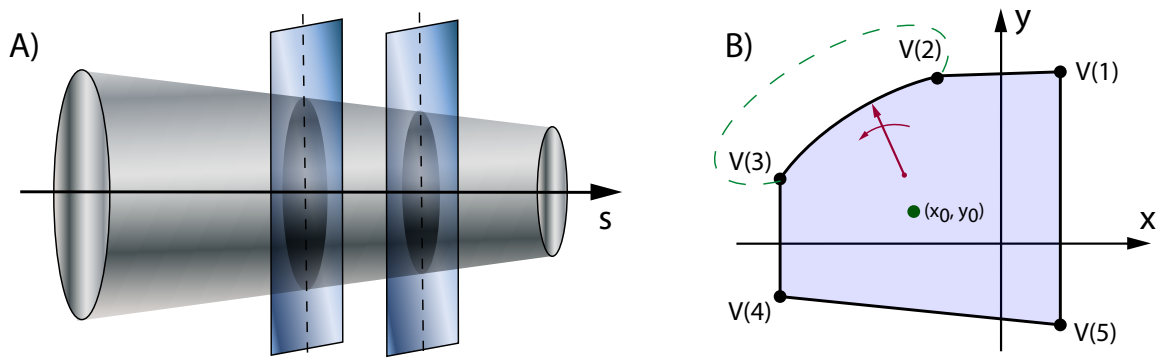


Figure 4.7: A) The inside wall of a capillary or the vacuum chamber wall of a non-capillary element is defined by a number of cross-sectional slices. B) Each cross-section is made up of a number of vertices. The segments between the vertices can be either a line segment, the arc of a circle, or a section of an ellipse.

sector origin r_0 except if `absolute_vertices` is set to `True` in which case the vertex numbers are taken as absolute. The arc between each vertex may be either a straight line, an arc of a circle, or a section of an ellipse. For a capillary it is mandatory that a cross-section be convex. That is, given any two points within the cross-section, all points on the line segment connecting them must be within the cross-section.

The `v(<j>)` within a cross-section define the vertices for each cross-section. The vertices are defined with respect to the section origin given by `r0`. Each `v(<j>)` has five parameters. It is mandatory to specify the first two parameters `<x>` and `<y>`. Specifying the rest, `<radius_x>`, `<radius_y>`, and `<tilt>`, is optional. The default values, if not specified, is zero. The point (`<x>`, `<y>`) defines the position of the vertex. The parameters `<radius_x>`, `<radius_y>`, and `<tilt>` define the shape of the segment of the cross-section between the given vertex and the preceding one.

```
<radius_x>  = 0, <radius_y>  = 0   --> Straight line segment.
<radius_x> != 0, <radius_y>  = 0   --> Circular arc with radius = radius_x
<radius_x>  = 0, <radius_y> != 0   --> Illegal!
<radius_x> != 0, <radius_y> != 0   --> Ellipse section.
```

When an ellipse is specified, `<radius_x>`, and `<radius_y>` are the half width and half height of the semi-major axes and the `<tilt>` parameter gives the tilt of the ellipse. `<radius_x>` and `<radius_y>` must not be negative.

In the example above, for the first cross-section, `v(2)` specifies a non-zero `<radius_x>` and, by default, `<radius_y>` is zero. Thus the segment of the cross-section between `v(1)` and `v(2)` is circular in nature with a radius of 0.02. Since `v(3)` does not specify `<radius_x>` nor `<radius_y>`, the cross-section between `v(2)` and `v(3)` is a straight line segment.

The vertex points must be arranged in a "counter clockwise manner". For vertices `<v(i)>` and `<v(i+1)>` connected by a line segment this translates to

$$0 < \theta_{i+1} - \theta_i \pmod{2\pi} < \pi \tag{4.5}$$

where (r_n, θ_n) are the polar coordinates of the n^{th} vertex. For vertices connected by an arc, "counter clockwise manner" means that the line segment with one end at the center of the arc and the other end traversing the arc from `<v(i)>` to `<v(i+1)>` rotates in counter clockwise as shown in Fig. 4.7B.

The red line segment with one end at the center of the arc and the other end traversing the arc from, in this case, $V(2)$ to $V(3)$, rotates in counter clockwise manner. In general, there are two solutions for constructing such an arc. For positive radii, the solution chosen is the one whose center is closest to

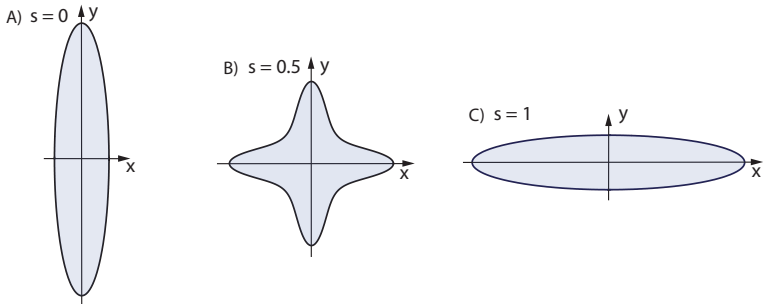


Figure 4.8: Example where convex cross-sections do not produce a convex volume. Cross-sections (A) and (C) are ellipses with a 5 to 1 aspect ratio. Half way in between, linear interpolation produces a convex cross-section as shown in (B).

the section origin (x_0, y_0). If the radii are negative, the center point will be the point farthest from the origin (the dashed line between $V(2)$ and $V(3)$ in the figure).

A restriction on cross-sections is that the section origin (x_0, y_0) must be in the interior of any cross-section and that for any cross-section a line drawn from the origin at any given angle θ will intersect the cross-section at exactly one point as shown in Fig. 4.7B. This is an important point in the construction of the wall between cross-sections as explained below.

The last vertex specified, call it `<v(n)>`, should not have the same `<x>`, `<y>` values as the first vertex `<v(1)>`. That is, there will be a segment of the cross-section connecting `<v(n)>` to `<v(1)>`. The geometry of this segment is determined by the parameters of `<v(1)>`.

If there is mirror symmetry about the x or y axis for a cross-section, the "mirrored" vertices, on the "negative" side of the mirror plane, do not have to be specified. Thus if all the vertex points of a cross-section are in the first quadrant, that is, all `<x>` and `<y>` are zero or positive, mirror symmetry about both the x and y axes is assumed. If all the `<y>` values are zero or positive and some `<x>` values are positive and some are negative, mirror symmetry about the x axis is assumed. Finally, if all the `<x>` values are zero or positive but some `<y>` values are positive and some are negative, symmetry about the y axis is assumed. For example, for the first in the above example, since all the `<y>` values are non-negative and there are positive and negative `<x>` values, symmetry about the x axis is assumed.

The one exception to the above rule that (`<x>`, `<y>`) is the vertex center is when a single vertex `v(1)` is specified for a cross-section with a non-zero `<radius_x>`. In this case, (`<x>`, `<y>`) are taken to be the center of the circle or ellipse. For example, if a single vertex is specified for a cross-section as:

```
section = {s = 0.3, v(1) = {0.03, -0.01, 0.15, 0.08, 0.2}}
```

the cross-section will be an ellipse with center at $(0.03, -0.01)$ with a tilt of 0.2 and axes radii of 0.15 and 0.08. If a cross-section has a single vertex and `<radius_x>` is not specified, the cross-section is a rectangle. For example

```
section = {s = 0.3, v(1) = {0.03, 0.01}}
```

4.11.3 Interpolation Between Sections

For `capillary` and vacuum chamber walls, the wall between cross-sections, is defined by interpolation. At a given s position, the r, θ coordinate system in the transverse x, y plane is defined with respect to an origin $\mathbf{r}_O(s)$ given by a linear interpolaion of the origins of the cross-sections to either side of the given s position. Let s_1 denote the position of the cross-section just before s and s_2 denote the position of the

cross-section just after s. Let $\mathbf{r}_{01}$ be the (x_0, y_0) origin defined for the cross section at s_1 and $\mathbf{r}_{02}$ be the (x_0, y_0) origin defined for the cross section at s_2. Then

$$\mathbf{r}_O(s) = (1 - \widetilde{s})\,\mathbf{r}_{01} + \widetilde{s}\,\mathbf{r}_{02} \tag{4.6}$$

where

$$\widetilde{s} \equiv \frac{s - s_1}{s_2 - s_1} \tag{4.7}$$

Let $r_{c1}(\theta)$ and $r_{c2}(\theta)$ be the radiusus of the wall as a function of θ for the cross-sections at $s = s_1$ and $s = s_2$ respectively. The wall $r_c(\theta, s)$ at any point s between s_1 and s_2 is then defined by the equation

$$r_c(\theta, s) = p_1(\widetilde{s})\, r_{c1}(\theta) + p_2(\widetilde{s})\, r_{c2}(\theta) \tag{4.8}$$

where p_1 and p_2 are cubic polynomials parameterized by

$$\begin{aligned} p_1 &= 1 - \widetilde{s} + a_1\,\widetilde{s} + a_2\,\widetilde{s}^2 + a_3\,\widetilde{s}^3 \\ p_2 &= \widetilde{s} + b_1\,\widetilde{s} + b_2\,\widetilde{s}^2 + b_3\,\widetilde{s}^3 \end{aligned} \tag{4.9}$$

If $a_i = b_i = 0$ for all $i = 1, 2, 3$, the interpolation is linear and this is the default if either of the parameters `dr_ds1` and `dr_ds2` are not given in the wall definition. These parameters are the slopes of the wall with respect to s at the end points

$$\text{dr\_ds1} \equiv \left.\frac{d\overline{r}}{ds}\right|_{s=s_1}, \qquad \text{dr\_ds2} \equiv \left.\frac{d\overline{r}}{ds}\right|_{s=s_2} \tag{4.10}$$

where $\overline{r}$ is the average r averaged over all θ. When *both* `dr_ds1` and `dr_ds2` are specified, the a_i and b_i are calculated so that the slopes of the wall match the values of `dr_ds1` and `dr_ds2` along with the constraints.

$$\begin{aligned} &p_1(0) = 1\,, \qquad p_1(1) = 0 \\ &p_2(0) = 0\,, \qquad p_2(1) = 1 \\ &M \equiv a_1^2 + a_2^2 + a_3^2 + b_1^2 + b_2^2 + b_3^2 \text{ is a minimum} \end{aligned} \tag{4.11}$$

The last constraint ensures a "smooth" transition between the two cross-sections.

To refer to a cross-section parameters after an element has been defined, the following syntax is used:

```
ele_name[wall%section(n)%v(j)%x]   ! x value of j^th vertex of n^th cross-section
```

4.11.4 Capillary Wall

For a `capillary`, `s` must be zero for the first cross-section and the length of the capillary is given by the value of `s` of the last cross-section.

For a `capillary`, in order for *Bmad* to quickly track photons, *Bmad* assumes that the volume between the cross-sections is convex. The volume will be convex if each cross-section $r_c(\theta, s)$ at any given s is convex. Note that it is *not* sufficient for $r_c(\theta, s)$ to be convex at the specified cross-sections as shown in Fig. 4.8. Also note that it is perfectly fine for the total capillary volume to not be convex.

4.11.5 Vacuum Chamber Wall

The vacuum chamber wall is independent of the element apertures (§4.8). Unless a program is specifically constructed, the presence of a vacuum chamber wall will not affect particle tracking.

The vacuum chamber wall defined for an element may be shorter or longer than the element. The vacuum chamber wall for a particular lattice branch is the sum of all the chamber walls of the individual elements. That is, the chamber wall at any given point is determined by interpolation of the nearest sections upstream and downstream to the point. Thus a given lattice element need not contain a `wall` component for the chamber wall to be well defined at the element.

The exception to the above rule is when a `section` has its `type` component set to either:

```
wall_start
wall_end
```

`wall_start` and `wall_end` sections must come in pairs. The next section after a `wall_end` section (if this section is not the last section in the lattice) must be a `wall_start` section. If a section has a `type` of `wall_start`, the region between that section and the previous section (which must be a `wall_end` section) will be considered to have no wall. If the `wall_start` section is the first section of the lattice branch, the region of no wall will start at the beginning of the branch. Similarly, if a section has a `type` of `wall_end`, the region between that section and the next section (or the end of the lattice branch if there is no next seciton) will not have a wall.

The chamber walls of any two elements may not overlap. The exception is when the `superimpose` attribute for a wall of an element is set to True. In this case, any other wall cross-sections from any other elements that overlap the superimposed wall are discarded. Superposition of a wall is useful, for example, in introducing mask regions into the wall.

If a branch has a closed geometry (§8.1), wall sections that extend beyound the ends of the branch are "wrapped" around.

If a particle is past the last wall cross-section or before the first wall cross-section, The following rules are used: If the branch has a `closed geometry`, the wall will be interpolated between the last and first cross-sections. If the branch has an `open` geometry, the wall is taken to have a constant cross-section in these regions.

The chamber wall is defined with respect to the local coordinate system (§14.1.1). That is, in a bend a wall that has a constant cross section is a section of a torus.

`Patch` elements (§3.37) complicate the wall geometry since the coordinate system at the end of the `patch` may be arbitrarily located relative to the beginning of the patch. To avoid confusion as to what coordinate system a wall section belongs to, `patch` elements are not allowed to define a wall. The wall through a patch is determined by the closest wall sections of neighboring elements.

Each section has a `type` attribute. This attribute is not used for `capillary` elements. For a vacuum chamber wall, the `type` attribute is used to dscribe a "crotch" geometry where two pipes merge into one pipe. The possible values for the `type` attribute are:

```
normal     ! default
leg1
leg2
trunk1
trunk2
trunk
```

The geometry of a crotch is shown in Fig. 4.9A. Two pipes, called "leg1" and "leg2", merge into one pipe called the "trunk" pipe. The trunk pipe can be either upstream or downstream of the leg pipes. To describe this situation, five sections are needed: One section in each leg pipe which need to have their `type` attribute set to `leg1` and `leg2`, and three sections in the trunk with one having a a `type` attribute of `trunk1`, another having a `type` attribute of `trunk2` and the third haveing a `type` attribute of `trunk`. There can be no sections between the leg sections and the trunk sections.

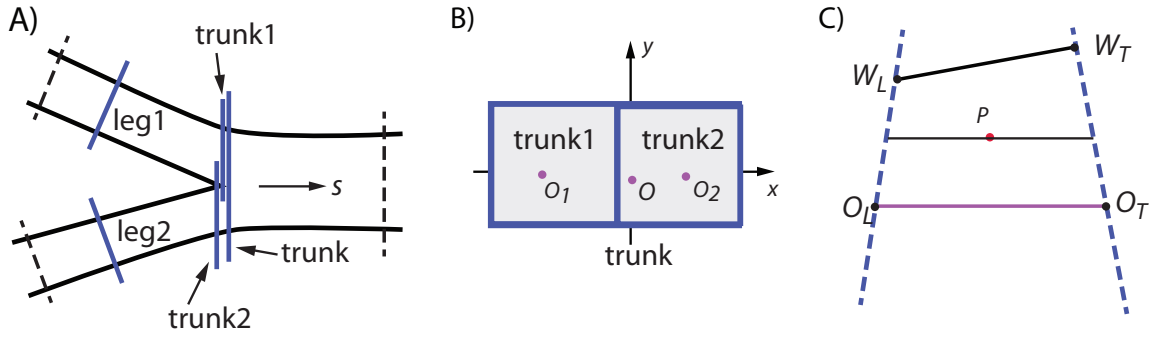


Figure 4.9: A) Crotch geometry: Two pipes labeled "leg1" and "leg2" merge into a single pipe called the "trunk" pipe. Five wall sections are used to define the crotch geometry (solid lines). Dashed lines represent sections not involved in defining the crotch. For purposes of illustration, the three trunk sections are displaced longitudinally but in reality must have the same longitudinal coordinate. B) Example layout of the trunk1, trunk2 and trunk wall sections. O_1, O_2 and O are the x_0, y_0 origins of the sections.

All three trunk sections must be associated with the same element and have the same `s` value. In the list of sections of the element containing the trunk elements, the `trunk1` and `trunk2` sections must be listed first if the leg pipes are upstream of the trunk pipe (the situation shown in the figure) and must be listed last if the leg pipes are downstream. That is, the `trunk1` and `trunk2` sections are "between" the leg sections and the `trunk` section. It does not matter if `trunk1` is before or after `trunk2`.

The `trunk1` and `trunk2` sections must not overlap and the `trunk` section must be constructed so that its area is the union of the areas of `trunk1` and `trunk2`. An example is illustrated in Fig. 4.9B. Here the `trunk1` and `trunk2` sections are squares with origins labeled O_1 and O_2 in the figure. By necessity, these origins must be different since each must lie within the boundaries of their respective areas. The `trunk` section is a rectagle encomposing the two squares and has an origin labeled O.

Between `leg1` and `trunk1` sections the wall is interpolated using these two section. Similarly for the region between `leg2` and `trunk2` sections. Away from these regions interpolation is done as outlined in §4.11.3. However, these two regions need a different interpolation scheme since, `leg1` and `trunk1`, as well as `leg2` and `trunk2` sections do not have to be parallel to each other.

4.11.6 Mask Wall For Diffraction Plate and Mask Elements

The `wall` of a `diffraction_plate` or `mask` element specifies what areas of the element will transmit or reflect particle and what areas will not. The areas where there is transmission or reflection are called "clear" areas and everything else is called "opaque".

A `wall` is comprised a a ordered list of `sections` as discussed in §4.11.2. Each section of the wall must have its `type` attribute set to one of:

```
clear
opaque
```

A section is called "clear" or "opaque" depending upon the setting of its `type` attribute. Do not confuse "clear section" with "clear area".

A clear area is defined by one or more consecutive wall sections. The first section that defines a clear area must be a clear section. All the other sections associated with a clear area must be opaque sections.

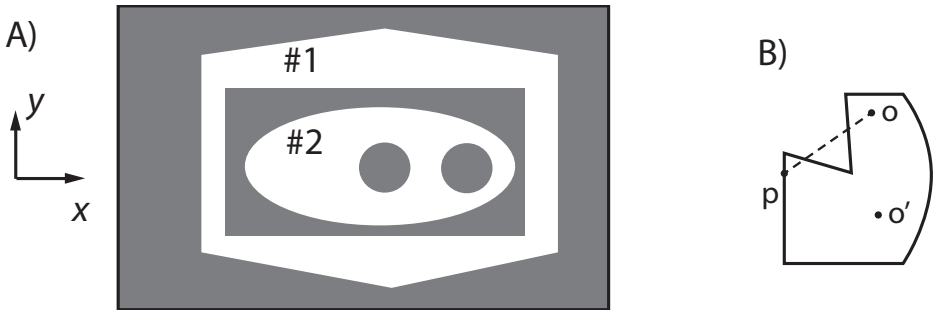


Figure 4.10: A) The diffraction plate or mask surface is divided into "clear" (white) and "opaque" (black) areas. In this example there are two clear sections labeled #1 and #2. B) All wall sections must be star shaped with respect to the section's origin. In this example, The section is *not* star shaped since a line drawn from the origin point o to the point p on the boundary intersects the boundary twice in between. In this case the section can be made star shaped by moving the origin to o'.

That is, a clear area starts with a clear section and any preceeding opaque sections up to the next clear section or the end of the section list. As a consequence of the above rules, the fist section of the wall must be a clear section and the number of clear areas is equal to the number of clear sections.

The default behavior is that a photon will be transmitted if it is within any clear area. A photon is considered to be within a given clear area if its (x, y) coordinates put in within the corresponding clear section but not within any `opaque` section of the clear area. `Opaque` sections only affect the clear area they are associated with. See the example below.

Any clear section can be given a `material` and `thickness`. Available materials are listed in §4.9. A photon transversing a clear area with a defined material will be attenuated and have a phase shift. Note that `material` and `thickness` properties are not to be assigned to `opaque` sections.

To enable *Bmad* to quickly calculate whether a photon has landed on a clear or opaque section, All sections, both clear and opaque, must be "star shaped" with respect to the (x_0, y_0) origin used by the section. That is, a line drawn from the section origin to any point on the section boundary must not pass through any boundary points of the section in between. This is illustrated in Fig. 4.10B where the section is not star shaped since a line drawn from the origin o to the point p on the boundary passes through two boundary points in between. In this case the section can trivially be made star shaped by moving the origin to point o'. If it is not possible to make a section star shaped by moving the origin, the section must be divided into multiple sections.

An example geometry is shown in Fig. 4.10A. In the figure, there are two openings labeled #1, and #2. A wall that constructs this geometry is:

```
z_plate: diffraction_plate, wall = {
  thickness = <real>
  opaque_material = <material_type>
  clear_material = <material_type>
  section = {           ! Clear area # 1
    type = clear,
    v(1) = {0.04, 0}, v(2) = {0.04, 0.022},
    v(3) = {0, 0.03},
  section = {
    type = opaque,
    v(1) = {0.032, 0.016},
  section = {          ! Clear area # 2
```

```
    type = clear,
    v(1) = {0, 0, 0.03, 0.013},
  section = {
    type = opaque,
    v(1) = {0, 0, 0.005},
  section = {
    type = opaque,
    r0 = (0.02, 0.00),
    v(1) = {0, 0, 0.005} }
```

Clear area #1 has a clear section and one opaque section. These sections rely on the four fold symmetry of the sections so that only points in the first quadrant need be specified. Clear area #2 has one clear section in the shape of an ellipse with two opaque circles. Notice that the `opaque` section of clear area #1 does not affect the clear area #2 even though it (completely) overlaps clear area #2.

Sections may overlap and a opaque section does not have to be wholly within the corresponding clear section. If a photon is within multiple clear areas then, for the purposes of calculation, it is considered to be within the first possible clear area in the list.

4.12 Length Attributes

The length attributes are

```
l       = <Real>  !
l_chord = <Real>  ! Chord length of a bend. Dependent attribute.
```

The length `l` is the path length of the reference particle. The one exception is for an `rbend`, the length `l` set in the lattice file is the chord length (§3.5). internally, *Bmad* converts all `rbend`s to `sbend`s and stores the chord length under the `l_chord` attribute. Example:

```
b: rbend, l = 0.6   ! For rbends, l will be converted to l_chord
```

For a `girder` element the length `l` is a dependent attribute and is set by *Bmad* to be the difference in longitudinal position s of the downstream end of the last element supported relative to the upstream end of the first element.

For `wiggler`s, the length `l` is not the same as the path length for a particle with the reference energy starting on the reference orbit. See §14.1.1.

For `patch` elements the `l` length is, by definition, equal to `z_offset`. For `patch` elements, `l` is a dependent attribute and will be automatically set to `z_offset` by *Bmad*.

The length of a `capillary` element is a dependent variable and is given by the value of `s` of the last wall cross-section (§4.11.4).

The length of a crystal is zero for Bragg diffraction and is a dependent attribute dependent upon the crystal thickness for Laue diffraction. See §3.8 for more details.

4.13 Is_on Attribute

The `is_on` attribute

```
is_on = <Logical>
```

is used to turn an element off. Turning an element off essentially converts it into a drift. Example

```
q1: quad, l = 0.6, k1 = 0.95
q1[is_on] = False
```

`is_on` does not affect any apertures that are set. Additionally, `is_on` does not affect the reference orbit. Therefore, turning off an `lcavity` will not affect the reference energy.

The following elements cannot be "turned off:"

```
beginning_ele
capillary
crystal
drift
fiducial
floor_shift
patch
group
null_ele
overlay
hybrid
mirror
multilayer_mirror
photon_init
sample
```

4.14 Multipole Attributes: Magnetic and Electric

Multipole formulas for are given in §15.1 and §15.2. Note that the setting of `field_master` (§4.2) will determine if multipoles are interpreted as normalized or unnormalized.

A `multipole` (§3.32) element specifies its magnetic multipole components using an Amplitude (`KnL`) and a tilt (`Tn`)

```
KnL = <Real>
Tn  = <Real>  ! Default is $pi$/(2n + 2)
```

Where `n` is an integer in the range from 0 (dipole component) through 21. If `Tn` is given without a value, a default of $pi/(2n + 2)$ will be used producing a skew field. Example:

```
m: multipole, k1l = 0.32, t1  ! Skew quadrupole of strength 0.32
```

Following `MAD`, a non-zero dipole (`K0L` component will affect the reference orbit (just like a normal dipole will). This is not true for any other element.

An `ab_multipole` (§3.1) specifies magnetic multipoles using normal (`Bn`) and skew (`An`) components:

```
An = <Real>
Bn = <Real>
```

Here `n` ranges from 0 (dipole component) through 21. Example:

```
q1: ab_multipole, b2 = 0.12, a20 = 1e7, field_master = T
```

Elements like `quadrupoles` and `sextupoles` can have assigned to them both magnetic and electric multipole fields. In this case, the magnetic fields are specified using the same convention as the `ab_multipole`. For such non-multipole elements, the magnetic multipole strength is scaled by a factor $F\, r_0^{n_{\rm ref}}/r_0^n$ (cf. Eq. (15.17)) where F is the strength of the element (for example F is $K1 \cdot L$ for a quadrupole), and r_0 is the "measurement radius" and is set by the `r0_mag` attribute. The default value of r_0. This behavior may be turned off by setting the `scale_multipoles` attribute. Example:

```
q1: quadrupole, b2 = 0.12, a20 = 1e7, scale_multipoles = F
```

Alternatively, a value of zero (the default) for `r0_mag` is equivalent to setting `scale_multipoles` to False.

Electric multipoles are specified using normal (`Bn_elec`) and skew (`An_elec`) components.

```
An_elec = <Real>
Bn_elec = <Real>
```

Here `n` ranges from 0 (dipole component) through 21. Like the magnetic multipoles, a measurement radius `r0_elec` can be used to scale the multipoles as explained in §15.2. Example:

```
q1: quadrupole, l = 1.2, b2_elec = 1e6, r0_elec = 0.034
```

See §15.2 for how electric multipoles are defined. Notice that Electric multipoles are never scalled by the element's field strength as they are with magnetic multipoles. If the value of `r0_elec` is zero (the default) the multipoles will not be scalled.

Unlike magnetic multipoles, there are no factors of the reference momentum nor the element length in the definition for electric multipoles. That is, electric multipole values represent the field and not the normalized integrated field. Thus an electric multipole associated with a zero length element will have no effect on tracking. This being the case, *Bmad* does not allow electric multipole values to be specified for `multipole` and `ab_multipole` elements. Indeed, in the limit of zero element length at constant integrated electric field strength, the equations of motion are singular since, unlike the magnetic case, the infinite fringe fields give rise to infinite energy shifts.

The magnetic and electric multipole kick can be toggled on or off using the `multipoles_on` attribute. Example:

```
call, file = 'lattice.bmad'              ! Read in a lattice file
quadrupole::*[multipoles_on] = False     ! But I want the multipoles off.
q1[k1] = 0.3                             ! k1 attribute not affected.
```

`multipoles_on` only effect multipoles specified by `An`, `Bn`, `An_elec`, or `Bn_elec`. Other multipoles, like the `k2` multipole of a `sextupole`, are not affected. The exception is `multipole` and `ab_multipole` elements do not have the `mulipoles_on` attribute. Rather they can be toggles on/off using the `is_on` attribute.

4.15 Field Maps

There are two general ways to specify complicated electro-magnetic field configurations that cannot be simply modeled using multipoles. One way is to use `custom` fields. Specifying a custom field is done by using custom code and linking this code with *Bmad* into a program. That is, custom fields are defined outside of the *Bmad* software (§5.4).

The other way to specify a complicated field is to use a "`field map`". There are four types of field maps:

```
cartesian_map       ! §4.15.2
cylindrical_map     ! §4.15.3
grid_field          ! §4.15.4
taylor_field        ! §4.15.5
```

Essentially, `cylindrical_map` and `cartesian_map` define fields using a set of functions with user defined coefficients with the functions formulated to obey Maxwell's equations. The `grid_field` type defines the field on a grid of points and interpolation is used to evaluate the field inbetween the points. Finally, the `taylor_field` type defines a set of Taylor maps. Each map defines the field in the transverse (x, y) plane at constant z. Interpolation is used to evaluate the field in between the planes.

The `cylindrical_map` and `grid_field` types can be used with both RF and DC fields. The other two types can only be used with DC fields. RF fields may only be used with the following element classes:

```
e_gun         ! §3.14
em_field      ! §3.16
lcavity       ! §3.26
rfcavity      ! §3.40
```

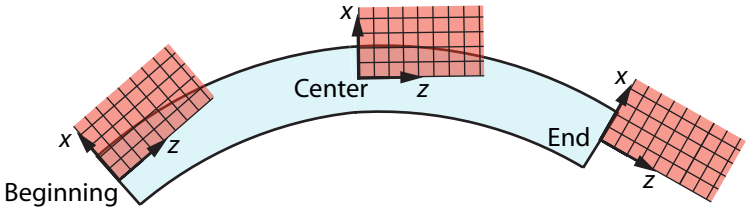


Figure 4.11: When used with a bend element, by default, field map coordinates will be Cartesian and not curved like the reference orbit. The orientation of the field map coordinates is determined by the setting of `ele_anchor_pt`. To use curvilinear coordinates instead, `curved_ref_frame` must be set to `True` [Available in `grid_field` and `taylor_field` only].

An element may specify multiple fields of a given type and/or may define multiple fields of different types. In both these cases, the field in the element is taken to be the sum of the individual fields. For example:

```
sb: sbend, field_calc = fieldmap, cylindrical_map = {...},  cylindrical_map = {...}
```

In this example an element has two `cylindrical_map` fields and the total field is the sum of the fields of each one. Separating fields like this can be useful, for example, to decouple the specification of electric from magnetic fields, or to decouple the specification of AC and DC fields.

fields may stored in a `binary` format (§2.19). For example:

```
qq: quadrupole, grid_field = call::my_grid.bin, ...
```

The field of one element can overlap onto other elements. This is explained in Sec. §4.17.

Field maps are used with integration type tracking methods (§5.4). It is important to note that field maps are *ignored* by `bmad_standard` tracking. Additionaly, `grid_field` field maps cannot be used with `symp_lie_ptc`.

Field maps may extend longitudinally beyound the ends of an element. See Sec (§4.17) for more details.

In a lattice file, once a field map is defined for an element, components of the field map may be redefined using the notation

```
ele_name[field_map_name(index)%component_name] = value
```

where `ele_name` is the name of the element, `field_map_name` is the name of the type of field map, `index` is the index of the field map which is "1" for the first field map defined for an element, etc., `component_name` is the name of the component, and `value` is the value to set to. Example:

```
qq, quadrupole, grid_field = {field_scale = 0.5, ...}, ...
qq[grid_field(1)%field_scale] = 0.7  ! Change field_scale value
```

4.15.1 Field Map Common attributes

This section explains some of the attributes that are common to the `field map` types. Not all attributes are used in all field map types. See the documentation on the individual types for a list of the attributes pertainent to that type.

curved_ref_frame

For bends, the coordinates of the field are, by default, Cartesian and do not follow the curved bend coordinates. The orientation of the field map coordinates with respect to the bend is determined by the placement of the anchor point (specified by `ele_anchor_pt`) as shown in Fig. 4.11. In this case, when tracking a particle, *Bmad* will convert particle coordinates (which are expressed in the

bend's curvilinear coordinate system defined by the reference orbit) to the Cartesian coordinates of the field map and will rotate the computed field from the field map coordinates back to the particle coordinates.

For `grid_field` and `taylor_field` types only, this default behavior can be changed by setting the `curved_ref_frame` component of the field map to `True`. In this case, the field grid coordinates will follow curved bend coordinates. The `curved_ref_frame` parameter is only pertinent for bend elements (`sbend`s, `rbend`s). The setting of `curved_ref_frame` is ignored for non-bend elements.

ele_anchor_pt

The `ele_anchor_pt`, along with `r0`, determines the origin of the field with respect to the lattice element. Possible settings are:

```
beginning   ! Beginning of element (default).
center      ! Center of element.
end         ! Exit end of element.
```

Example:

```
rfc0: rfcavity, taylor_field = {ele_anchor_pt = center, ...}, ...
```

field_type

The `field_type` attribute sets the type of field described. Possible settings for `field_type` are:

```
electric      ! Pure electric field. For DC fields only.
magnetic      ! Pure magnetic field. For DC fields only.
mixed         ! Mixed EM fields. Used for grid_field only.
```

Example:

```
bb: sbend, cartesian_map = {field_type = electric, ...}, ...
```

The `cylindrical_map` type does not have a `field_type` since it has explicit arrays for the electric and magnetic fields.

field_scale

The `field_scale` attribute is used to scale the overall field magnitude. The default value is 1. A value of -1 will reverse the field. If the `master_parameter` is defained, it is multiplied with the `field_scale` to give the overall scale. Example:

```
qq, quadrupole, grid_field = {field_scale = 0.5, ...}, ...
qq[grid_field(1)%field_scale] = 0.7  ! Change field_scale value after element def.
```

harmonic

The `harmonic` attibute, along with `rf_frequency` element attribute, sets the oscillation frequency of the field map. The `harmonic` attribute is only used with `cylindrical_map` and `grid_field` types. The default value of `harmonic` is 0. The `harmonic` number needs to be 0 for DC fields. Example:

```
lc1: lcavity, rf_frequency = 500e6, grid_field = {harmonic = 2, ...}, ...
```

Notice that `rf_frequency` is set outside of any field map and is common to all field maps.

master_parameter

The `master_parameter` defines a "master" element attribute for scaling the field. Example:

```
qq: quadrupole, taylor_field = {master_parameter = 'K1', ...}, k1 = ...
```

This example defines the `master_parameter` for the `taylor_field` to be the quadrupole strength `k1`. By using the same master parameter for a set of field map instances within a given lattice element, the sum field of the set can be controled by a single attribute. The `master_parameter` must be set to a valid element attribute. If the name is blank (""), no master parameter is used. The `master_parameter`, if defined, is multiplied with the `field_scale` to give the value used to scale the fields. The default `master_parameter` is blank ("") except for `wiggler` elements where, for historical reasons, the default is `polarity`.

Certain parameters have an associated "error" parameter. If the `master_parameter` is one of these parameters, the value used to scale the field is the sum of the `master_parameter` and the associated error parameter. The elements that have associated error parameters are:

```
Parameter     Assoc. Error Param    Element Type
---------     ------------------    --------------
Voltage       Voltage_err           E_gun, LCavity
Gradient      Gradient_err          E_gun, LCavity
Phi0          Phi0_err              E_gun, LCavity
G             G_err                 Sbend, Rbend
B_field       B_field_err           Sbemd, Rbend
```

phi0_fieldmap

For AC fields, `phi0_fieldmap` is the phase of the field map field relative to the fundamental mode. The phase `phi0_fieldmap` is relative to the fundamental frequency and not the frequency of the field map mode. That is, the "zero crossing" point of the field map is shifted by a time `phi0_fieldmap`$/f_0$ where f_0 is the fundamental mode frequency.

r0

The `r0` attribute is the $(x0, y0, z0)$ vector specifying the offset of the origin point that defines the field relative to the `anchor` point defined by `ele_anchor_pt`. The origin position of the field (`r_origin`) is determined by

```
r_origin = r0 + r_anchor
```

where `r_anchor` is determined by the setting of `ele_anchor_pt`. In the reference coordinates (§14.1.1) with respect to the element `r_anchor` is:

```
ele_anchor_pt       r_anchor
-------------       ---------
beginning           (0, 0, 0)      ! Default
center              (0, 0, L/2)
end                 (0, 0, L)
```

with `L` being the length of the element.

Example:

```
rfc0: rfcavity, taylor_field = {r0 = (-0.23, ...), ...}, ...
```

4.15.2 Cartesian_Map Field Map

The `cartesian_map` field map is only used for DC fields. Each term of a `cartesian_map` is a solution of Laplace's equation in cartesian coordinates. as described in Sec. §15.5.

The lattice file syntax for the `cartesian_map` type is:

```
cartesian_map = {
  field_type       = <String>,           ! Type of field: Default = Magnetic.
  field_scale      = <Real>,             ! Scale factor for the E & B fields.
  master_parameter = <Name>,             ! Master scaling parameter for E & B fields.
  ele_anchor_pt    = <Position>,         ! Anchor position: Beginning (default), Center, or End.
  r0               = (<x0>, <y0>, <z0>), ! Anchor offset. Default is 0.
  term = {<A>, <k_x>, <k_y>, <k_z>, <x_0>, <y_0>, <phi_z>, <family>},
  term = {....}  }
```

The possible settings of `<family>` are explained in Sec. §15.5. Example:

```
q01: quadrupole, l = 0.6, field_calc = fieldmap,
       cartesian_map = {
         term = {0.03, 3.00, 4.00, 5.00, 0, 0, 0.63, y},
         term = {...}, ...    }
```

See Sec. §4.15.1 for an explanation of the attributes that are common with other field map types.

To use with PTC dependent tracking methods (§5.4) there are a number of restrictions:

- There can be only one `cartesian_map` field map and there cannot be any other field maps of any kind.
- `cartesian_map` may not be used with a bend.
- Only magnetic fields may be used.
- The transverse terms in `r0` must be zero.

4.15.3 Cylindrical_Map Field Map

The `cylindrical_map` field map is used for both DC and AC fields. Each term of a `cylindrical_map` is a solution of Laplace's equation in cylindrical coordinates. as described in Sec. §15.6.

The lattice file syntax for the `cylindrical_map` type is:

```
cylindrical_map = {
  field_scale      = <Real>,              ! Scale factor for the E & B fields.
  master_parameter = <Name>,              ! Master scaling parameter for E & B fields.
  ele_anchor_pt    = <Position>,          ! Anchor position: Beginning (default), Center, or End.
  m                = <Integer>,           ! Azimuthal mode number
  harmonic         = <Integer>,           ! RF frequency harmonic number
  phi0_fieldmap    = <Real>,              ! Phase of oscillations.
  theta0_azimuth   = <Real>,              ! Azimuthal orientation.
  r0               = (<x0>, <y0>, <z0>), ! Anchor offset. Default is 0.
  dz        = <Real>,                     ! Distance between sampled field points.
  e_coef_re = (<Real>, <Real>, ....),     ! Real part of E.
  e_coef_im = (<Real>, <Real>, ....),     ! Imaginary part of E.
  b_coef_re = (<Real>, <Real>, ....),     ! Real part of B.
  b_coef_im = (<Real>, <Real>, ....),     ! Imaginary part of B.
}
```

See Sec. §4.15.1 for an explanation of the attributes that are common with other field map types.

For DC fields, the `e` coefficients specify the electric fields and the `b` coefficients specify the magnetic fields. For AC fields, the `e` coefficients specify modes that have finite longitudinal electric fields while the modes associated with the `b` coefficients do not.

To specify the RF frequency, specify the `rf_frequency` *element* attribute along with the `harmonic` attribute. See the discussion of the `harmonic` attribute in Sec. §4.15.1.

The basic equations used for the `cylindrical_map` decomposition of the fields are given in Section §15.6. A lattice element may have multiple `cylindriacl_map` components with each `cylindrical_map` being associated with a particular azimuthal mode m.

`e_re` and `e_im` give the real an imaginary part of e and `b_re` and `b_im` give the real and imaginary part of b. All of these vectors must be present and have the same length. The exception is with an $m = 0$ mode either the e or b arrays can be omitted and will default to zero. The number of terms N for the e or b vectors must be a power of 2 and all modes must have the same number of terms. The n^{th} element in the e or b arrays, with n running from 0 to $N-1$, is associated with a wavelength k_n

$$k_n = \begin{cases} \frac{2\,\pi\,n}{N\,dz} & 0 \le n < \frac{N}{2} \\ \frac{2\,\pi\,(n-N)}{N\,dz} & \frac{N}{2} \le n \le N-1 \end{cases} \tag{4.12}$$

This convention produces less high frequency components then the convention of using $k_n = 2\,\pi\,n/Ndz$.

The longitudinal length of the field is

$$L_{\text{field}} = \frac{N-1}{dz} \tag{4.13}$$

this may be different from the length `l` specified for the element.

For AC fields, the time t in Eq. (15.47) is computed depending upon whether `absolute time tracking` or `relative time tracking` is being used as discussed in §22.1. For `rfcavity` elements, the phase factor ϕ_{0j} in Eq. (15.47) is computed by

ϕ_{0j} `= harmonic(j) * [0.25 - (phi0 + phi0_multipass + phi0_err + phi0_autoscale + phi0_fieldmap(j))]`

where `phi0_fieldmap(j)` and `harmonic(j)` are specific to the jth grid field while the other factors are element parameters and so will be the same for all grid field maps of a given element. For non `rfcavity` elements the phase is

ϕ_{0j} `= harmonic(j) * [phi0 + phi0_multipass + phi0_err + phi0_autoscale + phi0_fieldmap(j)]`

where `phi0_fieldmap(j)` and `harmonic(j)` are specific to the jth cylindrical field map while the other factors are element parameters and so will be the same for all cylindrical field maps of a given element.

Example:

```
m1: lcavity, rf_frequency = 1e6, voltage = 2e6, cylindrical_map = {
  m = 2,                   harmonic = 3,
  r0 = (0, 0, 0.001),      dz = 0.1,
  theta0_azimuth = 0.3,    field_scale = 0.7,
  ele_anchor_pt = center,  master_parameter = voltage,
  e_coef_re = (...),       e_coef_im = (...),
  b_coef_re = (...),       b_coef_im = (...)}, field_calc = fieldmap
```

Note: When using PTC based tracking (§5), the following restrictions apply:

- The fields must be DC.
- all the `e_coef` and `b_coef` arrays must have the same length.
- `r0(1)` and `r0(2)` (the transverse offsets) must be zero.
- The element containing the map cannot be an `sbend` or `rbend`.
- May not be combined with other field map types.

4.15.4 Grid_Field Field Map

A `grid_field` is grid of field points specified using the syntax:

```
grid_field = {
  geometry         = <String>,    ! Geometry of the grid.
  field_type       = <String>,    ! Type of field: Default = Mixed.
  field_scale      = <Real>,      ! Scale factor for the E & B fields.
  phi0_fieldmap    = <Real>,      ! Phase of oscillations.
  harmonic         = <Integer>,   ! RF frequency harmonic number
  master_parameter = <Name>,      ! Master scaling parameter for E & B fields.
  curved_ref_frame = <Logical>,   ! Use a curved reference frame with bends?
  r0   = (...),                   ! Grid origin. Syntax is geometry dependent.
  dr   = (...),                   ! Grid spacing. Syntax is geometry dependent.
  ele_anchor_pt = <Position>      ! BEGINNING, CENTER, or END
  pt(<Integer>, ...) = ( ... ), ! Field points. Syntax is geometry dependent.
  ... } } }
```

See Sec. §4.15.1 for an explanation of the attributes that are common with other field map types.

To specify the RF frequency, specify the `rf_frequency` *element* attribute along with the `harmonic` attribute. See the discussion of the `harmonic` attribute in Sec. §4.15.1.

For `field_type` set to `electric` or `magnetic`, the field is DC. That is, For `field_type` set to `electric` or `magnetic`, the value of `harmonic` must be 0. For `field_type` set to `mixed`, the field may be DC or AC.

For AC fields, the individual field components are complex. the syntax for specifying a complex number is:

```
(<Re>, <Im>)
```

Example:

```
pt(0, 0, -7) = ((0.34, -4.3), (2.37, 9.34), ...)  ! Complex field
pt(0, 0, -7) = (0.12, -0.33, ...)                 ! Imaginary components are zero
```

The actual fields $\mathbf{E}$ and $\mathbf{B}$ are computed from the complex fields $\mathbf{E}_c$ and $\mathbf{B}_c$ via

$$\mathbf{E} = \Re\Big[\mathbf{E}_c \exp\left(-2\,\pi\,i\,(\phi_t + \phi_{\rm ref})\right)\Big] \tag{4.14}$$

with a similar equation for $\mathbf{B}$. ϕ_t is the part of the phase due to when the particle arrives at the cavity and depends upon whether `absolute time tracking` or `relative time tracking` is being used as discussed in §22.1. The phase $\phi_{\rm ref}$ for the j^{th} grid field in an `rfcavity` element is

$\phi_{\texttt{ref,j}}$ `= harmonic(j) * [0.25 - (phi0 + phi0_multipass + phi0_err +`
`phi0_autoscale + phi0_fieldmap(j))]`

where `phi0_fieldmap(j)` and `harmonic(j)` are specific to the jth grid field while the other factors are element parameters and so will be the same for all grid field maps of a given element. For non `rfcavity` elements the phase is

$\phi_{\texttt{ref,j}}$ `= harmonic(j) * [phi0 + phi0_multipass + phi0_err +`
`phi0_autoscale + phi0_fieldmap(j)]`

The `geometry` switch sets the type of the grid and must come before any `pt` is given. The possible settings of `geometry` are:

```
rotationally_symmetric_rz
xyz
```

The `rotationally_symmetric_rz` setting for `geometry` is for fields that are rotationally symmetric around the z axis. The format for this type of `grid_field` is

```
grid_field = {
  geometry = rotationally_symmetric_rz,
  r0   = (<x0>, <y0>, <z0>),  ! Grid origin
  dr   = (<dr>, <dz>),        ! Grid spacing
  pt(<ir>, <iz>) = (<E_r>, <E_phi>, <E_z>) ! For field_type = Electric
  pt(<ir>, <iz>) = (<B_r>, <B_phi>, <B_z>) ! For field_type = Magnetic
  pt(<ir>, <iz>) = (<E_r>, <E_phi>, <E_z>, <B_r>, <B_phi>, <B_z>)
                                           ! For field_type = Mixed.
  ... }
```

where `<iz>` can be negative but `<ir>` must be non-negative.

The `xyz` setting for `geometry` can be used for all rectangular field grids. The format for this type of `grid_field` is

```
grid_field = {
  geometry = xyz,
  r0   = (<x0>, <y0>, <z0>),    ! Grid origin
```

```
  dr   = (<dx>, <dy>, <dz>),    ! Grid spacing
  pt(<ix>, <iy>, <iz>) = (<E_x>, <E_y>, <E_z>),  ! For field_type = Electric
  pt(<ix>, <iy>, <iz>) = (<B_x>, <B_y>, <B_z>),  ! For field_type = Magnetic
  pt(<ix>, <iy>, <iz>) = (<E_x>, <E_y>, <E_z>, <B_x>, <B_y>, <B_z>),
                                                 ! For field_type = Mixed.
  ... }
```

where `<ix>`, `<iy>`, and `<iz>` can be negative.

[For clarity sake, the following discusses the `xyz` case. Extension to other cases is straight forward.] There is no restriction on the bounds of the indexes `(ix, iy, iz)` of the `pt(ix, iy, iz)` array. A point (ix, iy, iz) corresponds in space to the point (x, y, z):

```
(x, y, z) = dr * (ix, iy, iz) + r0 + r_anchor
```

where `z` is measured from the beginning of the element and `r_anchor` is determined by the setting of `ele_anchor_pt`:

```
ele_anchor_pt       r_anchor
-------------       ---------
beginning           (0, 0, 0)      ! Default
center              (0, 0, L/2)
end                 (0, 0, L)
```

with `L` being the length of the element.

Example:

```
apex: e_gun, l = 0.23, field_calc = fieldmap, rf_frequency = 187e6,
                                grid_field = call::apex_gun_grid.bmad
```

with the file `apex_gun_grid.bmad` being:

```
{
  geometry = rotationally_symmetric_rz,
  harmonic = 1,
  master_parameter = voltage,
  r0 = (0, 0),
  dr = (0.001, 0.001),
  pt(0,0) = ( (0, 0), (0, 0), (1, 0),  (0, 0), (0, 0), (0, 0)),
  pt(0,1) = ( (0, 0), (0, 0), (0.99, 0),  (0, 0), (0, 0), (0, 0)),
  ... }
```

It is considered an error if the field of the grid is evaluated for a point that is transversely outside of the grid. That is, a grid must extend transversely to the aperture or at least beyound the trajectory of any particle. [Actually, to prevent problems when the aperture is set at the grid boundary, if the distance between the particle and the grid boundary is within 1/2 of the spacing between grid points, no error is generated and the field will calcuated using extrapolation.] On the other hand, it is acceptable to evaluate the grid field at a point that is longitudinally outside of the grid. In this case, the field is assumed to go to zero. This is done by effectively adding to the grid two planes of zero field longitudinally to either side of the grid. So a particle traveling ouside of the grid longitudinally will see the field drop to zero within one longitudinal grid spacing length.

4.15.5 Taylor_Field Field Map

The `taylor_field` field map is only used for DC fields. A `taylor_field` is comprised of a set of two dimensional Taylor maps.

The syntax for describing a `taylor_field` is:

```
taylor_field = {
  field_type       = <String>,           ! Type of field: Default = Magnetic.
  field_scale      = <Real>,             ! Scale factor for the E & B fields.
  master_parameter = <Name>,             ! Master scaling parameter for E & B fields.
  curved_ref_frame   = <Logical>,        ! Use curved coords with bends?
  canonical_tracking = <logical>         ! Use (px, py) instead of (x', y') when tracking?
  ele_anchor_pt    = <Position>,         ! Anchor position: Beginning (default), Center, or End.
  r0               = (<x0>, <y0>, <z0>), ! Anchor offset. Default is 0.
  dz               = <Real>,             ! Distance between sampled field points.
  plane(<i>) = {<term1>, <term2>, <term3>, ....}, ! Taylor map at constant z.
  plane(<j>) = {....}  }
```

See Sec. §4.15.1 for an explanation of the attributes that are common with other field map types.

Each `plane(<i>)` component specifies a Taylor map at constant z (§15.7). The index `<i>` for the different planes must be in consecutive order. The plane with <i> = 0 corresponds to the $z = 0$ origin. There is no restriction for the starting `<i>` index of the first plane. That is, there does not have to be a plane with index `<i>` = 0 (in this case, the origin needs to be outside of the element).

The individual Taylor series terms follow the same syntax as the Taylor terms in a `taylor` element (§3.46):

```
{<out>: <coef>, <e1> <e2>}           ! EM Taylor term. First form.
{<out>: <coef> | <n1> <n2> ...}      ! EM Taylor term. Second form.
```

except that 1) `<out>` must be one of

```
Bx, By, or Bz
```

2) there are only two exponents `<e1>` and `<e2>` corresponding to `x`, and `y` respectively for the first form, and 3) the integers `<n1>`, `<n2>`, etc., are restricted to being 1 or 2.

To use with PTC dependent tracking methods (§5.4) there are a number of restrictions:

- There can be only one `taylor_field` field map and there cannot be any other field maps of any kind.
- The number of planes must be odd. If you want to be able to track with PTC to the center of the element, the number of planes must be of the form $4\,n + 1$ where n is an integer.
- Only magnetic fields may be used.
- The plane locations must be symmetric with respect to the center of the element. That is, the first plane and the last plane must be equidistant from the element center.
- The element may not be superimposed upon (§7.1). [PTC tracks from plane to plane so there is no good way to cut an element at an arbitrary position.]
- In a bend with `curved_ref_frame` = False, The setting of `ele_anchor_pt` must be `center`.

If the edges of the `taylor_field`, which are defined by the first and last planes, is different than the edges of the element, *Bmad* and PTC will do tracking differently. With *Bmad*, the tracking will start at one edge of the element and will end at the other end. That is, *Bmad* will ignore the field outside of the element (but this can be modified using field overlap (§4.17)). PTC, on the other hand, will start at an element end and then track from the element end backwards to the first plane like in a drift. It will then track to the ending plane and finally track backwards from the ending plane to the ending element edge line in a drift. That is, PTC does not ignore the field outside of the element.

When using PTC based tracking (§5), the `canonical_tracking` parameter can be used to select whether the transverse phase space coordinates used in tracking is the canonical (x, p_x, y, p_y) (§14.4.2) coordinates or the non-canonical (x, x', y, y') coordinates. The default is `False`. The difference comes at the edges of element if the field is not zero. With canonical coordinates, there is a kick at the edge while with the non-canonical there is not. What is best depends upon the problem. For example, with a solenoid magnet where the field is constant right up to the edge, canonical tracking will be needed since the edge kick is significant. However, the edge kick is not well defined (that is, certain assumptions are built into the edge kick calculation. For example, with a solenoid, cylindrical symmetry is generally assumed). Thus for some arbitrary field, the assumptions used by PTC may by incorrect for the problem at hand. Thus in many cases, especially for single pass machines, the non-canonical tracking is a better choice.

Example:

```
t1: sbend, k1 = 2, taylor_field = {
  field_type = electric,   ele_anchor_pt = end,
  dz = 1.2,                r0 = (0, 0, 2.0),
  field_scale = 1.3,       curved_ref_frame = False,
  master_parameter = k1,
  plane(-2) = {{Bx: 0.3| 1}, {Bx: 1.3, 1 2}, {By: 0.7, 3 1}, ...},
  plane(-1) = {{Bx: 0.6|}, {By: 0.4|112}, {By: 0.7, 4 0}, ...},
  ...  }, field_calc = fieldmap
```

4.16 RF Couplers

For `lcavity` and `rfcavity` elements, the attributes that characterize the dipole transverse kick due to a coupler port are:

```
coupler_at       = <Switch> ! What end the coupler is at
coupler_strength = <Real>   ! Normalized strength
coupler_angle    = <Real>   ! Polarization angle (rad/2π)
coupler_phase    = <Real>   ! Phase angle with respect to the RF (rad/2π)
```

The possible `coupler_at` settings are:

```
entrance_end
exit_end  ! default
both_ends
```

The kick due to the coupler is

```
dP_x = amp * cos(phase) * cos(angle)
dP_y = amp * cos(phase) * sin(angle)
dE   = amp * (cos(angle) * x + sin(angle) * y) * sin(phase) * twopi * rf_frequency / c_light
```

where `dP_x` and `dP_y` are the transverse momentum kicks, `dE` is an energy kick, and

```
amp   = gradient * coupler_strength
phase = twopi * (phase_particle + phase_ref + coupler_phase)         ! For lcavity §3.26
      = pi/2 + twopi * (phase_particle - phase_ref + coupler_phase)  ! For rfcavity §3.26
angle = twopi * coupler_angle
```

The energy kick is needed to keep things symplectic.

Example:

```
rf1: lcav, l = 4.5, gradient = 1.2e6, coupler_at = both_ends,
                                                  coupler_strength = 0.037
```

4.17 Field Extending Beyond Element Boundary

The `field_overlaps` element attribute can be used to indicate that the electric or magnetic fields of one element overlap another element. The syntax is:

```
<overlapping_ele>: ... field_overlaps = {<overlapped_ele1>, <overlapped_ele2>, ...}
```

The {} braces are optional if there is only one overlapped element.

Example:

```
b1: sbend, l = 2.3, field_overlaps = {q1, s2}, ...
inj_line: line = (..., s2, b1, mark3, q1, ...)
```

In this example, the field of element `b1` extends beyond the ends of `b1` and overlaps elements `q1` and `s2`. There is no limit to the number of elements that are overlapped by any given element and overlapped elements do not have to be next to the overlapping element in the line. If there are multiple elements whose name matches the name of a overlapped element, the element closest to the overlapping element is chosen. Thus in the above example, if there are multiple elements named `q1`, the closest `q1` to `b1` is designated as the overlapped element.

There can be multipole `field_overlaps = ...` constructs for an overlapping element. Thus the following is equivalent to the above example:

```
b1: sbend, l = 2.3, field_overlaps = q1, field_overlaps = s2
```

Note: When the field overlaps elements that are superimposed (§7.1), the overlapped elements must be the `super_lord` elements and never the slaved elements.

The field, when `field_calc` (§5.4) is set to `bmad_standard`, never extends beyond the element boundary and so a `bmad_standard` field will never overlap another element.

4.18 Automatic Scaling of Accelerating Fields

Elements that have accelerating fields are:

```
e_gun       ! §3.14
em_field    ! §3.16
lcavity     ! §3.26
rfcavity    ! §3.40
```

[Notice that `rfcavity` elements by definition, have a constant reference energy while with all the other elements the entrance end reference energy will, in general, be different from the exit end reference energy.]

The problem that arises with accelerating fields is how to set the overall amplitude (and phase if the fields are oscillating) of the field so that a particle, starting on the reference orbit and starting with the reference energy, has the desired energy gain at the exit end of the element where the "desired" is set by the `voltage` or `gradient` attribute of the element plus a `phi0` phase attribute for AC fields.

The scaling problem is not present when `bmad_standard` tracking (§5.1) is used since `bmad_standard` tracking uses an integrated formula that is designed to give the proper acceleration. Rather it is a problem for Runge-Kutta and other methods integration methods.

The problem becomes even more complicated at non-ultra relativistic energies where the particle velocity is not a constant. In this case, the proper amplitude and/or phase settings will depend upon what the incoming energy of the reference particle is.

To help with the scaling problem, *Bmad* has the capability to automatically scale an accelerating field's amplitude and/or phase. The two lattice element parameters that turn on/off auto scaling are (§8.1):

```
autoscale_phase      = <Logical>  ! Automatic phase scaling.
autoscale_amplitude  = <Logical>  ! Automatic amplitude scaling.
```

The default value is True for both parameters. Example:

```
rf2: rfcavity, autoscale_phase = F
```

Scaling takes place during program execution when a lattice is initially created (that is, when the lattice file is parsed) and when parameters in the lattice that would change the scaling are varied. The element parameters varied when autoscaling is done are:

```
field_autoscale       ! Amplitude scale
phi0_autoscale        ! phase scale
```

For an `rfcavity` element, the `field_autoscale` parameter is set so that when the phase is adjusted for maximum acceleration, the voltage gain of a particle on the zero orbit is equal to the value of the element's `voltage` parameter and then phi0_autoscale is shifted by approximately 90 degrees so that with `phi0` equal to zero a particle on the zero orbit will not see any energy gain through the cavity.

If no autoscaling is done, the default setting of `field_autoscale` is 1 and the default setting of `phi0_autoscale` is 0.

For an `rfcavity`, the autoscaling is normally done around a phase of `phi0` = 0 which is appropriate for a ring above transition since `phi0` = 0 is the stable fixed point. Below transition, the stable fixed point is at `phi0` = 0.5. In this case, the `bmad_com` global parameter `rf_phase_below_transition_ref` (§9.2) should be set to True so that autoscaling is done around `phi0` = 0.5. For an ultrarelativistic reference particle with speed very nearly c there is no difference in the autoscaling between `phi0` = 0 and `phi0` = 0.5. However, when there are velocity variations of the particle within the cavity there will be differences between autoscaling at the two phases.

Notice that if autoscaling is on, the `field_autoscale` is adjusted so that a particle on the zero orbit will see a field appropriate for the setting of the element's `voltage` setting. If autoscaling is off, the field is independent of the element's `voltage` setting.

`field_autoscale` and `phi0_autoscale` are not needed and therefore ignored when `bmad_standard` tracking is done.

4.19 Wakefields

Wake fields can be specified for many elements. The attributes that characterize the wakes are:

```
sr_wake_file     = <String>   ! Short range wake field definition file. (§4.19.1)
lr_wake_file     = <String>   ! Long range wake field definition file. (§4.19.2)
lr_freq_spread   = <Real>     ! RMS fractional frequency spread of the LR wake fields.
lr_self_wake_on  = <Logical>  ! Apply longitudinal long range self-wake? Default = True.
```

The `lr_freq_spread` attribute is used to randomly spread out the long range mode frequencies among different cavities. The spread is Gaussian in shape with an RMS of `lr_freq_spread` * F where F is the frequency of a mode.

The `lr_self_wake_on` attribute can be used to turn off the longitudinal long-range self-wake which is the longitudinal kick on the particles of a bunch due to the wake generated by these same particles. [The transverse self wake is always zero.] The default setting of `lr_self_wake_on` is `True`. Turning off the self-wake, for example, can be done to avoid double counting if both long-range and short-range wakes are defined.

Example:

```
abc: lcavity, lr_wake_file = 'lr.wake', lr_freq_spread = 0.0023, lr_self_wake_on = F
```

The formulas used to compute the wake field are given in §15.9. *Bmad* has two modes for tracking particles. One mode tracks individual particles one at a time. The other mode tracks bunches of particles. Which mode is used for a given program is decided by the program. The wake field is ignored when tracking individual particles and only used when tracking bunches.

4.19.1 Short-Range Wakes

The input file name for the short–range wake fields is specified using the `sr_wake_file` attribute. The file gives both monopole longitudinal and dipole transverse wakes. An example input file is:

```
! Pseudo Wake modes:
!                        Amp       Damp          K      Phase  Polar-    Transverse_
! Longitudinal:       [V/C/m]     [1/m]       [1/m]     [rad]  ization   Dependence
! Transverse:      [V/C/m^2]     [1/m]       [1/m]     [rad]

&short_range_modes
  longitudinal(1) = 3.23e14     1.23e3     3.62e3     0.123
  longitudinal(2) = 6.95e13     5.02e2     1.90e3    -1.503
  .. etc ..
  transverse(1) =   4.23e14     2.23e3     5.62e3     0.789    X  linear_trailing
  transverse(2) =   8.40e13     5.94e2     1.92e3     1.455
   .. etc ..
  z_max = 1.3e-3
/
```

Wakes are specified via a set of "pseudo" modes (§15.9.1). The magnitude of `z_max` should be set to the maximum z value at which the pseudo mode fit is valid. *Bmad* will check the distance between particles does not exceed `z_max`. If it does, *Bmad* will report an error.

The `polarization` parameter is used to specify the wake polarization. Possible settings for this parameter are:

```
none    ! Default
x_axis
y_axis
```

The `polarization` name may be abbreviated. For example, if the `polarization` is set to `x_axis`, there is no vertical kick from the pseudo mode.

The `transverse_dependence` parameter sets whether the wake kick is linear in the offset of the leading or trailing particle or is independent of the transverse offset. Possible settings of this parameter are:

```
none              ! Default for longitudinal modes
linear_leading    ! Default for transverse modes
linear_trailing
```

The `transverse_dependence` parameter may be abbreviated. Note: Due to the way the wake file is parsed, if `transverse_dependence` is specified for a particular mode, `polarization` must also be specified.

For `longitudinal` modes: If the `transverse_dependence` is `none` (the default), then the `polarization` must also be `none` (other combinations do not make sense). If the `transverse_dependence` is *not* `none` for a `longitudinal` mode, then the `polarization` must be set to `x_axis` or `y_axis`.

Note: In a beam chamber with circular symmetry, the linear terms in the `longitudinal` wake are zero and the transverse wake has no terms independent of the transverse offsets nor terms that depend upon the trailing particle offset.

4.19.2 Long-Range Wakes

Equations for long-range wakes is given in Sec. §15.9.2.

The input file name for the long–range wake fields is specified using the `lr_wake_file` attribute. The file gives the wake modes by specifying the frequency (in Hz), R/Q (in Ω/meter^{2m}), Q, and m (order number), and optionally the polarization angle (in radians/2pi) for each cavity mode. The input uses Fortran90 namelist syntax: The data begins with the string `&long_range_modes` and ends with a slash `/`. Everything outside this is ignored. Each mode is labeled `lr(i)` where `i` is the mode index. An example input file is:

```
              Freq      R/Q      Q    m   Polar   b_sin  b_cos a_sin  a_cos  t_ref
                       [Ohm/              Angle
              [Hz]      m^(2m)]          [Rad/2pi]
  &long_range_modes
    lr(1) = 1.650e9    0.76    7.0e4  1    "unpol"
    lr(2) = 1.699e9   11.21    5.0e4  1    "0.15"
    lr(3) =    0       0.57    1.1e6  0    "unpol"
  /
```

[Note: The quotation marks are needed with some compilers and not with others.] A frequency of zero is used to designate wakes that are part of the fundamental accelerating mode. *Bmad* needs to know if a wake is part of the fundamental mode due to timing issues as discussed in §22.1.

If the polarization angle is set to "`unpolarized`" the mode is taken to be unpolarized. [Note: Technically the unpolarized mode is actually two polarized normal modes. The axes of these two normal modes can be chosen arbitrary as long as they are at right angles to each other.]

Two element attributes that affect the long-range wake are `lr_freq_spread` and `lr_self_wake_on` as explained in §4.19.

After the long–range modes have been defined they can be referenced or redefined using the notation

```
lr(n)%freq      ! Frequency
lr(n)%r_over_q  ! R/Q
lr(n)%q         ! Q
lr(n)%angle     ! Polarization Angle
```

Example:

```
lcav[lr(2)%freq] = 1.1 * lcav[lr(2)%freq] ! Raise frequency by 10%
```

Example:

```
rf1: lcav, l = 4.5, gradient = 1.2e6, sr_wake_file = "sr1.dat"
```

4.20 Fringe Fields

Lattice elements can have fringe fields at the element edges. Whether *Bmad* tries to model the fringe fields using the models described below first depends upon what kind of tracking is done. Fringe effects are *not* applied when an element's `tracking_method` is set to:

```
custom
linear
mad
```

Additionally, no fringe effects will be used if the `tracking_method` is `runge_kutta` or `time_runge_kutta`, and the element's `field_calc` (§5.4) is not `bmad_standard`. This is done since it is assumed, in this case, that the body of the field includes any fringe fields.

4.20.1 Turning On/Off Fringe Effects

Whether fringe fields are ignored or not is determined by the setting of the `fringe_at` element parameter. The possible settings are

```
no_end
both_ends             ! Default
entrance_end
exit_end
```

This is particularly useful in vetoing the fringe effect in the interior of split elements. If there is no fringe at a particular boundary, the setting of attributes like `fringe_type` (see below) are ignored.

When a particle's spin is being tracked through the fringe field at an element's edge, the `spin_fringe_on` logical attribute of the element determines how the tracking is handled. Example:

```
q: quad, spin_fringe_on = T, fringe_at = exit_end
```

Here, there is no fringe effect at the entrance of the element and the fringe at the exit end of the element will affect the spin. The default setting of `spin_fringe_on` is `True`.

4.20.2 Fringe Types

Bmad and PTC have several fringe field models for the magnetic field. Which fringe model is used is set by two element attributes:

```
fringe_type
ptc_fringe_geometry
```

For elements that have a multipole type fringe field (dipole, quadrupole, etc., as opposed to solenoid or RF fringes), the `fringe_type` switch is used to select how a fringe field is simulated. For everything except for `rbend` and `sbend` elements, `fringe_type` may be set to one of:

```
none              ! Default
soft_edge_only
hard_edge_only
full
```

The `none` setting ignores any fringe fields. The fringe field kick is divided into two pieces. The first piece is called the `hard edge` fringe kick and is the kick in the limit that the longitudinal extent of the fringe is zero. The second piece is the `soft edge` fringe kick which is the fringe kick with the fringe having a finite longitudinal extent minus the hard edge fringe kick. That is

```
fringe kick = hard fringe kick + soft fringe kick
```

The advantage of separating the fringe kick in this way is that the hard fringe can be used without having to know anything about the longitudinal extent of the fringe. In many cases, this is a good enough approximation. Note that using the soft fringe without the hard fringe is not physical but can be useful in understanding how the soft edge component affects tracking. See §16 for details.

For `rbend` and `sbend` elements, the possible `fringe_type` settings are:

```
none
soft_edge_only      !          SAD equivalent: fringe = 1, disfrin = 1
hard_edge_only      !          SAD equivalent: fringe = 0, disfrin = 0
full
linear_edge
basic_bend          ! Default. SAD equivalent: fringe = 0, disfrin = 1
sad_full            !          SAD equivalent: fringe = 1, disfrin = 0
```

The `basic_bend` setting, which is the default, is essentially the basic vertical focusing effect that is present when there is a finite `e1` or `e2` face angle. With `bmad_standard` tracking, `basic_bend` also includes second order terms (§16.2). The `linear_edge` setting ignores these second order terms. In some cases, for instance in a chicane, `basic_bend` is not good enough. With `fringe_type` set to `full`, higher order effects are taken into account.

PTC does not have a `linear_edge` fringe model. With PTC tracking, `basic_bend` tracking is used if `linear_edge` is chosen.

Additionally, for use with PTC, the `ptc_fringe_geometry` switch can be used to define the symmetry of the fringe fields. Possible settings are:

```
x_invariant
multipole_symmetry
```

The difference between `x_invariant` and `multipole_symmetry` is that with `multipole_symmetry` the fringe field for an n^{th} order multipole is assumed to have the same rotational symmetry as the multipole. With this assumption, the fringe field has $n + 1^{st}$ order terms. With `x_invariant`, the fringe field is calculated assuming that there is translational invariance along the horizontal x axis. This differs from `multipole_symmetry` by adding terms of increasing order consistent with the translational invariance. See Étienne Forest's book[Forest98] for more details. Which setting of `ptc_fringe_geometry` is appropriate depends upon how the dipole under consideration is constructed. The two settings of `ptc_fringe_geometry` represent two points of a continuum of possible fringe field geometries.

Additionally, when using PTC tracking (§1.5), the `parameter[ptc_max_fringe_order]` (§8.1) determines the maximum order of the calculated fringe fields.

Example:

```
b1: rbend, angle = pi/4, g = 0.3, fringe_type = full
```

In a bend element, the `fringe_type` setting only pertains to the dipole fringe. For higher order multipole fields in a bend, the `higher_order_fringe_type` selects the fringe type. Possible settings are the same as for the non-bend `fringe_type` elements

```
none              ! Default
soft_edge_only
hard_edge_only
full
```

The `soft_edge_only`, `hard_edge_only` and `sad_full` settings of `fringe_type` emulate the fringe field tracking used in the SAD program[SAD]. The `soft_edge_only` setting only uses the linear part of the fringe, `hard_edge_only` ignores the linear part of the fringe, and `sad_full` uses the full fringe. For an `sbend` or `rbend` element, these SAD fringe fields are in addition to the fringe fields that occurs with a finite `e1` or `e2` face angle.

For `quadrupole` and `sad_mult` elements, the translation between the `fringe_at` and `fringe_type` settings and the `fringe` and `disfrin` switches of SAD is:

		fringe_at:			
		no_end	entrance_end	exit_end	both_ends*
fringe_type:	none	[0, $\neq 0$]	[0, $\neq 0$]	[0, $\neq 0$]	[0, $\neq 0$]
	soft_edge_only	[0, $\neq 0$]	[1, $\neq 0$]	[2, $\neq 0$]	[3, $\neq 0$]
	hard_edge_only*	[0, $\neq 0$]	No SAD Equiv	No SAD Equiv	[0, $= 0$]
	full	[0, $\neq 0$]	[1, $= 0$]	[2, $= 0$]	[3, $= 0$]
*Default value.		[fringe, disfrin]			

Each entry is the table is of the form [`fringe`, `disfrin`]. The `soft_edge_only` fringe kick is a kick that is linear in the transverse (x, p_x, y, p_y) coordinates and comes from the finite width of the quadrupolar

fringe field. The width of the quadrupolar fringe field is characterized by the `f1` and `f2` attributes. The `sad_nonlinear_only` fringe kick comes from the nonlinear part of the quadrupolar field plus the fringes of the other multipoles.

The soft fringe quadrupole parameters `fq1` and `fq2` (§16.3) are related to the corresponding SAD parameters `f1` and `f2` via

```
f1 = -sign(fq1) * sqrt(24 * |fq1|)
f2 = fq2
```

In the SAD documentation, the soft edge is called the "linear" fringe.

For programmers who deal with PTC directly: The translation between `ptc_fringe_geometry` on the *Bmad* side and `bendfringe` on the PTC side is:

ptc_fringe_geometry	*bendfringe*
x_invariant	True
multipole_symmetry	False

4.21 Instrumental Measurement Attributes

`instrument`, `monitor`, `detector`, and `marker` elements have special attributes to describe errors associated with orbit, betatron phase, dispersion and coupling measurements. These attributes are:

Attribute	*Symbol* (See: §24.2)	
`tilt`	θ_t	See §4.6
`x_offset`	$x_{\rm err}$	See §4.6
`y_offset`	$y_{\rm err}$	See §4.6
`x_gain_err`	$dg_{x,\rm err}$	Horizontal gain error
`y_gain_err`	$dg_{y,\rm err}$	Vertical gain error
`crunch`	$\psi_{\rm err}$	Crunch angle
`tilt_calib`	$\theta_{\rm err}$	tilt angle calibration
`x_offset_calib`	$x_{\rm cal}$	Horizontal offset calibration
`y_offset_calib`	$y_{\rm cal}$	Vertical offset calibration
`x_gain_calib`	$dg_{x,\rm cal}$	Horizontal gain calibration
`y_gain_calib`	$dg_{y,\rm cal}$	Vertical gain calibration
`crunch_calib`	$\psi_{\rm cal}$	Crunch angle calibration
`noise`	n_f	Noise factor
`de_eta_meas`	dE/E	Percent change in energy
`n_sample`	N_s	Number of sampling points
`osc_amplitude`	$A_{\rm osc}$	Oscillation amplitude

A program can use these quantities to calculate "measured" values from the "laboratory" values. Here, "laboratory" means as calculated from some model lattice. See §24.2 for the conversion formulas.

Chapter 5

Tracking, Spin, and Transfer Matrix Calculation Methods

Bmad allows for a number of methods that can be use to "track" a particle through a lattice element. Here "track" can mean one of three things:

```
1) Calculate a particle's phase space coordinates at the exit
   end of the element given the coordinates at the entrance end.
2) Calculate the linear transfer map (Jacobian) through an element
   about a given reference orbit.
3) Calculate the a particle's spin orientation at the exit end
   of the element given the coordinates at the beginning.
```

The different tracking methods that are available have different advantages and disadvantages in terms of speed, symplecticity, etc. What tracking method is used, is selected on an element–by–element basis using the attributes:

```
tracking_method      = <Switch>   ! phase space tracking method.
mat6_calc_method     = <Switch>   ! 6x6 transfer matrix calculation.
spin_tracking_method = <Switch>   ! Spin tracking method.
```

Example:

```
q2: quadrupole, tracking_method = symp_lie_ptc
q2[tracking_method] = symp_lie_ptc
quadrupole::*[tracking_method] = symp_lie_ptc
```

The first two lines of this example have exactly the same effect in terms of setting the `tracking_method`. The third line shows how to set the `tracking_method` for an entire class of elements.

These switches are discussed in more detail in the following sections.

5.1 Particle Tracking Methods

The `tracking_method` attribute of an element sets the algorithm that is used for single particle tracking through that element. Table 5.1 gives which methods are available for each type of element.

A note on terminology: Adaptive step size control used with the `Runge_Kutta` integrator means that instead of taking fixed step sizes the integrator chooses the proper step size so that the error in the tracking is below the maximum allowable error set by `rel_tol` and `abs_tol` tolerances. The advantage

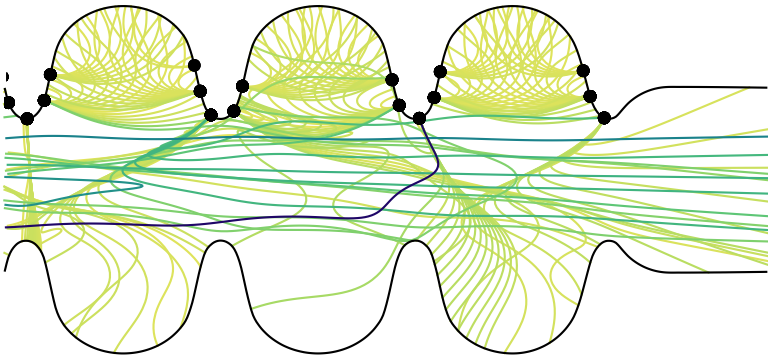


Figure 5.1: Dark current tracking. Example of where a time based tracker (`time_runge_kutta`) is useful for simulating particles that can reverse their longitudinal velocity. Here the tracks drawn are from a simulation of "dark current" electrons generated at the walls of an RF cavity due to the large electromagnetic fields.

of step size control is that the integrator uses a smaller step size when needed (the fields are rapidly varying), but makes larger steps when it can. The disadvantage is that a step is more computationally intensive since the error in a step is estimated by repeating a step using two mini steps. If the fields are rather uniform and you know what a good step size is you can save time by using a fixed step size.

`Bmad_Standard` Uses formulas for tracking. The formulas generally use the paraxial approximation. The emphasis here is on speed.

`Custom` This method will call a routine `track1_custom` which must be supplied by the programmer implementing the custom tracking. The default `track1_custom` supplied with the *Bmad* release will print an error message and stop the program if it is called which probably indicates a program linking problem. See `s:custom.ele` for more details.

`fixed_step_runge_kutta` The `fixed_step_runge_kutta` method is similar to `runge_kutta` tracking except that `fixed_step_runge_kutta` does not use adaptive step size control but instead takes steps of fixed size using the setting of `ds_step` or `num_steps` for the element being tracked through (§5.4). Generally, using adaptive step control will be much more efficient so it is recommended that `fixed_step_runge_kutta` *not* be used unless there is a compelling reason not to. For example, one reason to use `fixed_step_runge_kutta` would be to test how much adaptive step control contributes to non-symplecticity. Note: If `ds_step` is used to set the step size, and if the element length is not an integer multiple of `ds_step`, the last step will be shortened appropriately to track to the end of the element.

`fixed_step_time_runge_kutta` The `fixed_step_time_runge_kutta` method is similar to `time_runge_kutta` tracking except that `fixed_step_time_runge_kutta` does not use adaptive step size control but instead takes steps of fixed size using the setting of `ds_step` or `num_steps` for the element being tracked through (§5.4). Generally, using adaptive step control will be much more efficient so it is recommended that `fixed_step_time_runge_kutta` *not* be used unless there is a compelling reason not to. For example, one reason to use `fixed_step_time_runge_kutta` would be to test how much adaptive step control contributes to non-symplecticity. Note: If `ds_step` is used to set the step size, and if the element length is not an integer multiple of `ds_step`, the last step will be shortened appropriately to track to the end of the element.

`Linear` Linear just uses the 0th order vector with the 1st order 6x6 transfer matrix for an element. Very simple. Depending upon how the transfer matrix was generated this may or may not be symplectic.

Note: a `linear` tracking method may not be used with `mat6_calc_method` set to `tracking` since this would give a circular dependency.

`MAD` This uses the MAD 2nd order transfer map. This method is not able to handle element misalignments or kicks, and becomes inaccurate as the particle energy deviates from the reference energy. MAD tracking is generally only used for testing purposes. Note: Thanks to CERN and Frank Schmidt for permission to use the MAD tracking code within *Bmad*.

`runge_kutta` This uses a 4^{th} order Runge Kutta integration algorithm with adaptive step size control. This is essentially the `ODEINT` subroutine from Numerical Recipes[Press92]. This may be slow but it should be accurate. This method is non-symplectic. `Warning:` When using `custom` fields, if the fields do not obey Maxwell's equation, there is the possibility of the `runge_kutta` tracking halting mid way through an element. See section §5.4 for more details.

`Symp_Lie_Bmad` Symplectic tracking using a Hamiltonian with Lie operation techniques. This is similar to `Symp_Lie_PTC` (see below) except this uses a *Bmad* routine. By bypassing some of the generality inherent in PTC (§1.5), `Symp_Lie_Bmad` achieves about a factor of 10 improvement in speed over `Symp_Lie_PTC`.

`Symp_Lie_PTC` Symplectic tracking using a Hamiltonian with Lie operator techniques. This uses Étienne Forest's PTC (§1.5) software for the calculation. This method is symplectic but can be slow. Exceptions: The tracking is not symplectic when tracking through and element with an associated electric field and when tracking through a `taylor` element.

`Symp_Map` This uses a partially inverted, implicit Taylor map. The calculation uses Étienne Forest's PTC software (§1.5). Since the map is implicit, a Newton search method must be used. This will slow things down from the Taylor method but this is guaranteed symplectic. Note: Due to memory limitations in PTC, the number of elements using symp_map is limited to be of order 50.

`Taylor` This uses a Taylor map generated from the PTC (§1.5) package. Generating the map may take time but once you have it it should be very fast. One possible problem with using a Taylor map is that you have to worry about the accuracy if you do tracking at points that are far from the expansion point about which the map was made. This method is non-symplectic away from the expansion point. Whether the Taylor map is generated taking into account the offset an element has is governed by the `taylor_map_includes_offsets` attribute (§5.7).

The order of a Taylor map is set by the `parameter[taylor_order]` parameter (§8.1).

`Time_Runge_Kutta` This method uses time as the independent variable instead of the longitudinal z position. The advantage of this method is that it can handle particles which reverse direction longitudinally. One use for this method is "dark current" tracking where, as illustrated in Fig. 5.1, low energy particles generated at the vacuum chamber walls can be found traveling in all directions. Notice that `time_runge_kutta` is different from using `absolute time tracking` as explained in §22.1.

Element Class	Bmad_Standard	Custom	Linear	MAD	Fixed_Step_Runge_Kutta Runge_Kutta	Symp_Lie_Bmad	Symp_Lie_PTC	Symp_Map	Taylor	Fixed_Step_Time_Runge_Kutta Time_Runge_Kutta
ab_multipole	D	X	X				X	X	X	
ac_kicker	D	X	X		X					X
beambeam	D	X	X							
capillary	D	X								
crystal	D	X								
custom		D	X		X					X
drift	D	X	X	X	X		X	X	X	X
e_gun		X								D
ecollimator	D	X	X		X		X	X	X	X
elseparator	D	X	X	X	X		X	X	X	X
em_field		X			D		X	X	X	X
floor_shift	D	X								
hkicker	D	X	X		X		X	X	X	X
instrument	D	X	X		X		X	X	X	X
kicker	D	X	X		X		X	X	X	X
lcavity	D	X	X		X		X	X	X	X
marker	D	X	X				X	X	X	
match	D	X							X	
monitor	D	X	X		X		X	X	X	X
mirror	D	X								
multipole	D	X	X				X	X	X	
multilayer	D	X								
octupole	D	X	X		X		X	X	X	X
patch	D	X			X		X		X	
quadrupole	D	X	X	X	X	X	X	X	X	X
rbend	D	X	X	X			X	X	X	
rcollimator	D	X	X		X		X	X	X	X
rfcavity	D	X	X	X	X		X	X	X	X
sad_mult	D	X	X				X	X	X	
sample	D	X								
sbend	D	X	X	X	X		X	X	X	X
sextupole	D	X	X	X	X		X	X	X	X
solenoid	D	X	X	X	X	X	X	X	X	X
sol_quad	D	X	X		X	X	X	X	X	X
taylor		X	X				X		D	
vkicker	D	X	X		X		X	X	X	X
wiggler (map type)		X	X		X	X	X	X	X	X
wiggler (periodic type)	D	X	X		X[a]	X[a]	X[a]	X[a]	X[a]	

[a] See §3.47.1 for more details

Table 5.1: Table of valid **tracking_method** switches for a given element class. "D" denotes the default method. "X" denotes a valid method.

5.2 Linear Transfer Map Methods

The `mat6_calc_method` attribute sets how the 6x6 Jacobian transfer matrix for a given element is computed. Table 5.2 gives which methods are available for each type of element.

In addition to the `mat6_calc_method` switch, two element attributes that can affect the way the transfer matrix is calculated are `symplectify` and `taylor_map_includes_offsets`. These are discussed in sections §5.6 and §5.7 respectively.

For methods that do not necessarily produce a symplectic matrix the `symplectify` attribute of an element can be set to `True` to solve the problem. See §21.3.

Symplectic integration is like ordinary integration of a function f(x) but what is integrated here is a Taylor map. Truncating the map to 0^{th} order gives the particle trajectory and truncating to 1^{st} order gives the transfer matrix (Jacobian). The order at which a Taylor series is truncated at is set by `taylor_order` (see §8.1. Like ordinary integration there are various formulas that one can use to do symplectic integration. In *Bmad* (or more precisely in PTC (§1.5)) you can use one of 3 methods. This is set by `integrator_order`. `integrator_order` = n where n is allowed by PTC to be 2, 4, or 6. With an integration order of n the error in an integration step scales as dz^n where dz is step size. The step size dz is set by the length of the element and the value of `ds_step`. Remember, as in ordinary integration, higher integration order does not necessarily imply higher accuracy.

`Bmad_Standard` Uses formulas for the calculation. The formulas generally use the paraxial approximation. The emphasis here is on speed.

`Custom` This method will call a routine `make_mat6_custom` which must be supplied by the programmer implementing the custom transfer matrix calculation. The default `make_mat6_custom` supplied with the *Bmad* release will print an error message and stop the program if it is called which probably indicates a program linking problem. See `s:custom.ele` for more details.

`MAD` This uses the MAD 2nd transfer map. This method is not able to handle element misalignments or kicks, and becomes inaccurate as the particle energy deviates from the reference energy. MAD tracking is generally only used for testing purposes. Thanks must be given to CERN and Frank Schmidt for permission to use the MAD tracking code within *Bmad*.

`Static` This prevents the transfer matrix from being recomputed. Using `Static` in the input file is generally not a good idea since it prevents the matrix from being computed in the first place. Typically `Static` is used internally in a program to prevent recomputation.

`Symp_Lie_Bmad` A symplectic calculation using a Hamiltonian with Lie operator techniques. This is similar to `Symp_Lie_PTC` (see below) except this uses a *Bmad* routine. By bypassing some of the generality inherent in PTC, `Symp_Lie_Bmad` achieves about a factor of 10 improvement in speed over `Symp_Lie_PTC`. However, `Symp_Lie_Bmad` cannot generate maps above first order.

`Symp_Lie_PTC` Symplectic integration using a Hamiltonian and Lie operators. This uses the PTC (§1.5) software for the calculation. This method is symplectic but can be slow. Exceptions: The tracking is not symplectic when tracking through and element with an associated electric field and when tracking through a `taylor` element.

`Symp_Map` This uses a partially inverted, implicit Taylor map. The calculation uses Étienne Forest's PTC software (§1.5). Since the map is implicit, a Newton search method must be used. This will slow things down from the Taylor method but this is guaranteed symplectic. Note: Due to memory limitations in PTC, the number of elements using symp_map is limited to be of order 50.

`Taylor` This uses a Taylor map generated from Étienne's PTC package. Generating the map may take time but once you have it it should be very fast. One possible problem with using a Taylor map is that you have to worry about the accuracy if you do a calculation at points that are far from the expansion point about which the map was made. This method is non-symplectic away from the expansion point. Whether the Taylor map is generated taking into account the offset an element has is governed by the `taylor_map_includes_offsets` attribute (§5.7). `bmad_standard` and `taylor` tracking methods are identical. Note: Taylor maps for `match`, and `patch` elements are limited to first order.

The order of a Taylor map is set by the `parameter[taylor_order]` parameter (§8.1).

`Tracking` This uses the tracking method set by `tracking_method` to track 6 particles around the central orbit. This method is susceptible to inaccuracies caused by nonlinearities. Furthermore this method is almost surely slow. While non–symplectic, the advantage of this method is that it is directly related to any tracking results. Note: a `linear` tracking method may not be used with `mat6_calc_method` set to `tracking` since this would give a circular dependency. The two parameters that affect this calculation are `bmad_com%d_orb(6)` (§9.2) which sets the six deltas used for displacing the initial particle coordinates from the reference orbit and `bmad_com%mat6_track_symmetric` which sets whether symmetric or asymmetric differences are used.

	Bmad_Standard	Custom	MAD	Static	Symp_Lie_Bmad	Symp_Lie_PTC	Taylor	Tracking
ab_multipole	D	X		X		X	X	X
ac_kicker		X		X		X	X	D
beambeam	D	X		X				X
capillary	D	X		X				X
crystal	D	X		X				X
custom		D		X				X
drift	D	X	X	X		X	X	X
e_gun		X		X				D
ecollimator	D	X		X		X	X	X
elseparator	D	X	X	X		X	X	X
em_field		X		X		X	X	D
floor_shift	D	X		X				X
hkicker	D	X		X		X	X	X
instrument	D	X		X		X	X	X
kicker	D	X		X		X	X	X
lcavity	D	X		X		X	X	X
marker	D	X		X		X	X	X
match	D	X		X				X
mirror	D	X		X				X
monitor	D	X		X		X	X	X
multipole	D	X		X		X	X	X
multilayer_mirror	D	X		X				X
octupole	D	X		X		X	X	X
patch	D	X		X		X	X	X
quadrupole	D	X	X	X	X	X	X	X
rbend	D	X	X	X		X	X	X
rcollimator	D	X		X		X	X	X
rfcavity	D	X	X	X		X	X	X
sad_mult	D	X		X		X	X	X
sample	D	X		X				X
sbend	D	X	X	X		X	X	X
sextupole	D	X	X	X		X	X	X
solenoid	D	X	X	X	X	X	X	X
sol_quad	D	X	X	X	X	X	X	X
taylor		X		X		X	D	
vkicker	D	X		X		X	X	X
wiggler	D	X		X	X[a]	X[a]	X[a]	X

[a] See §3.47.1 for more details

Table 5.2: Table of available `mat6_calc_method` switches for a given element class. "D" denotes the default method. "X" denotes an available method.

5.3 Spin Tracking Methods

The `spin_tracking_method` attribute of an elements sets the algorithm that is used for tracking a particle's spin (§19.1) through that element. Table 5.3 gives which methods are available for each type of element.

Possible `spin_tracking_method` settings are:

`Custom`
: This method will call a routine `track1_spin_custom` which must be supplied by the programmer implementing the custom spin tracking calculation. See `s:custom.ele` for more details.

`Tracking`
: How spin is tracked here will depend also on the setting of `tracking_method`. If `tracking_method` is set to `runge_kutta` or `time_runge_kutta` the spin will be tracked along with the phase space particle coordinates using the local fields. For `tracking_method` set to `symp_lie_ptc`, the spin tracking will use `PTC`. For all other `tracking_method`s, the spin will be tracked using the "`bmad_standard`" spin tracking method which involves Romberg integration of the spin rotation matrix.

  The `runge_kutta` and `time_runge_kutta` spin tracking uses the same fourth order integrator as is used for the orbital coordinates to track the spin rotation vector.

`Symp_Lie_PTC`
: Symplectic integration using a Hamiltonian and Lie operators. This uses Étienne's PTC software for the calculation. This method is symplectic but can be slow.

Since speed may be an issue, *Bmad* has an global parameter called `spin_tracking_on` which is part of the `bmad_com` instance (§9.3) that determines whether spin is tracked or not. Note: There is also another `bmad_com` parameter called `spin_baier_katkov_flipping_on` which can influence spin tracking.

The `spin_fringe_on` element attribute (§4.20.1) can be used to toggle whether the fringe fields of an element affect the spin.

Example:

```
q: quadrupole, spin_tracking_method = symp_lie_ptc
```

	Custom	Symp_Lie_PTC	Tracking
ab_multipole	X		D
ac_kicker	X		D
beambeam	X		
capillary			
crystal			
custom	D		X
drift	X	X	D
e_gun	X		D
ecollimator	X	X	D
elseparator	X	X	D
em_field	X		D
hkicker	X	X	D
instrument	X	X	D
kicker	X	X	D
lcavity	X	X	D
marker	X	X	D
match	X		
mirror			
monitor	X	X	D
multipole	X	X	D
multilayer			
octupole	X	X	D
patch	X	X	
quadrupole	X	X	D
rbend	X	X	D
rcollimator	X	X	D
rfcavity	X	X	D
sad_mult	X		
sample			
sbend	X	X	D
sextupole	X	X	D
solenoid	X	X	D
sol_quad	X	X	D
taylor			
vkicker	X	X	D
wiggler	X	X	D

Table 5.3: Table of available `spin_tracking_method` switches for a given element class. "D" denotes the default method. "X" denotes an available method.

5.4 Integration Methods

"Integration methods" are tracking methods that involve integrating through an element's magnetic and electric fields. Integration methods are split into two classes: Those that can track Taylor maps and those that simply track a particle's position. The Taylor map methods are

```
symp_lie_bmad   ! Only to first order
symp_lie_ptc    ! Uses PTC
taylor          ! Uses PTC
```

See section §21.1 for more information on Taylor maps and symplectic integration. The latter two methods involve using the PTC library (§1.5).

The methods that do not involve Taylor maps are

```
fixed_step_runge_kutta
fixed_step_time_runge_kutta
runge_kutta
time_runge_kutta
```

there are a number of element attributes that can affect the calculation. They are

```
ds_step             = <Real>     ! Integration step length (§5.5.1)
num_steps           = <Integer>  ! Number of integration steps. (§5.5.1)
integrator_order    = <Integer>  ! Integrator order (§5.5.3)
field_calc          = <Switch>   ! How the field is calculated (§5.5.2)
```

Example:

```
q1: quadrupole, l = 0.6, tracking_method = bmad_standard,
      mat6_calc_method = symp_lie_ptc, ds_step = 0.2, field_calc = custom
```

5.5 CSR and Space Charge Methods

When doing beam tracking through an element (§17), Coherent Synchrotron Radiation (CSR) and Space Charge (SC) effects can be included by setting the appropriate method switches in that element. These switches are:

```
csr_method          = <Switch>   ! Coherent Synchrotron Radiation
space_charge_method = <Switch>   ! Space charge method
```

Note: For CSR or SC effects to be included in tracking the `bmad_com` logical `csr_and_space_charge_on` must be set to `True` (§9.2).

The possible settings for `csr_method` are

```
off             ! No CSR. Default.
1_dim           ! One dimensional calculation (§17.4.1).
```

The possible settings of `space_charge_method` are

```
off             ! No SC. Default.
slice           ! SC using slices (§17.4.2).
fft_3d          ! SC using a 3D grid (§17.4.3).
```

Example:

```
q1: quadrupole, l = 0.6, csr_method = 1_dim, space_charge_method = slice, ...
```

Note: There is also high energy space charge calculation that can be used with single particle tracking and is discussed in §17.5.

5.5.1 ds_step and num_steps Parameters

One way to create a transfer map through an element is to divide the element up into slices and then to propagate the transfer map slice by slice. There are several ways to do this integration. The `runge_kutta` type methods integrate the equations of motion to give the 0^{th} order Taylor map which just represents a particle's orbit. Symplectic integration using Lie algebraic techniques, on the other hand, can generate Taylor maps to any order. The `ds_step` attribute determines the slice thickness. Alternatively, `num_steps` attribute can be used in place of `ds_step` to specify the number of slices. This is applicable to `symp_lie_bmad` and `symp_lie_ptc` integration.

The `runge_kutta` and `time_runge_kutta` tracking uses adaptive step control independent of `ds_step`. These methods use three `bmad_com` parameters §9.3) namely:

```
bmad_com[rel_tol_adaptive_tracking]
bmad_com[abs_to_adaptive_tracking]
bmad_com[max_num_runge_kutta_step]
```

The estimated error of the integration is then bounded by

```
error < abs_tol + |orbit| * rel_tol
```

lowering the error bounds makes for greater accuracy (as long as round-off doesn't hurt) but for slower tracking.

5.5.2 Field_calc Parameter

The `runge_kutta` type tracking methods all use as input the electric and magnetic fields of an element. How the EM fields are calculated is determined by the `field_calc` attribute for an element. For all lattice elements, except `wigglers` and `undulators`, possible values for `field_calc` are:

```
bmad_standard     ! This is the default except for custom elements
custom            ! Default for custom elements.
fieldmap
```

For `wigglers` and `undulators`, possible values for `field_calc` are:

```
planar_model
helical_model
custom
fieldmap
```

For historical reasons, the default setting for `field_calc` for `wigglers` and `undulators` is `planar_model` except if there is a field map present (§4.15) in which case the default is `fieldmap`. Note that with `bmad_standard` tracking, the setting of `field_calc` is ignored except in the case of `wigglers` and `undulators` where `field_calc` must be set to either `planar_model` or `helical_model`.

`Custom` means that the field calculations are done outside of the *Bmad* software. A program doing `custom` field calculations will need the appropriate custom routine (§33.2). Elements that set `field_calc` to `fieldmap` need to have a field map defined (§4.15).

`Warning:` When tracking a particle through a custom field using `runge_kutta`, it is important that the field obey Maxwell's equations. Fields that do not obey Maxwell's Equations may cause the `runge_kutta` adaptive step size control algorithm to take smaller and smaller steps until the step size becomes so small the tracking will stop. What happens is that the step size control algorithm takes a step and then takes two half steps over the same region and from this estimates the error in the calculation. If the error is larger than the allowed tolerance the control algorithm shortens the step and tries again. A field that does not obey Maxwell's equations can fool the control algorithm into thinking that the error is always larger than the allowed tolerance for any finite step size. A typical situation is where the field has an unphysical step across some boundary.

5.5.3 PTC Integration

The `integrator_order` element attribute is the order of the integration formula for `Symp_Lie_PTC`. Possible values are

```
integrator_order = 2 (default), 4, or 6
```

Essentially, an integration order of n means that the error in an integration step scales as dz^{n+1} where dz is the slice thickness. For a given number of steps a higher order will give more accurate results but a higher order integrator will take more time per step. It turns out that for wigglers, after adjusting `ds_step` for a given accuracy, the order 2 integrator is the fastest. This is not surprising given the highly nonlinear nature of a wiggler. Note that `symp_lie_bmad` always uses an order 2 integrator independent of the setting of `integrator_order`.

When tracking uses the `PTC` library (§1.5), there are two global parameters that can be set in the lattice file that affect the calculation. These are:

```
parameter[ptc_exact_model]        = <Logical>  ! "exact" tracking? Default: False
parameter[ptc_exact_misalignment] = <Logical>  ! "exactly" misalign elements?
                                               !    Default: True
```

The default for `ptc_exact_model` is `False` and the default for `ptc_exact_misalignment` is `True`.

The `ptc_exact_model` parameter sets whether PTC uses an "exact" model for tracking. Essentially this means that the paraxial approximation (§22.3) is made for `ptc_exact_model` set to `False` and is not made if set to `True`. This can be important, for example, for bend tracking when the bend radius is small.

In PTC, exact modeling can be set on an element-by-element basis. Currently *Bmad* does not support specifying element-by-element setting of exact modeling. However, PTC does not have a non-exact tracking option for elements that have an electric field. In this case, PTC tracking will always be exact independent of the setting of `ptc_exact_model`. Additionally, for elements with an electric field, tracking will not be symplectic.

The `ptc_exact_misalignment` parameter determines whether misalignments are handled exactly or whether approximations are made that will speed up the calculation.

In addition to the above parameters, how the Hamiltonian is split when tracking with `PTC` can be set for individual elements using the `ptc_integration_type` parameter. Possible settings of this parameter are

```
drift_kick    ! See Eq.~(125) of [Forest06]
matrix_kick   ! See Eq.~(132) of [Forest06]. Default
ripken_kick   ! See Eq.~(130) of [Forest06]
```

Example:

```
q2: quad, l = 0.6, k1 = 0.34, ptc_integration_type = drift_kick
```

A discussion of the different types of integration schemes is given by Forest[Forest06]. The equation that shows the appropriate splitting of the Hamiltonian for each integration type is referenced in the above list. The `ripken_kick` type is for benchmarking with the `SixTrack` program and is not otherwise generally useful. The difference between `drift_kick` and `matrix_kick` is that with `drift_kick` the quadrupolar part of the magnetic multipole is is included in the applied kick between drifts while in the `matrix_kick` method the quadrupolar component is used for the "matrix" tracking between kicks. With the `matrix_kick` method the tune of a machine tends to be insensitive to how many integration steps (set by `ds_step` or `n_steps`) are used.

PTC does not implement `matrix_kick` tracking for elements with an electric field. In this case, the setting of `ptc_integration_type` is ignored and tracking will be `drift_kick`. Thus, if an electric field is introduced into an element, more integration steps may be required to get the correct tune.

5.6 Symplectify Attribute

The `symplectify` attribute

```
symplectify = <Logical>
```

is used to make the transfer matrix for an element symplectic. The linear transport matrix may be non–symplectic for a number of reasons. For example, the linear matrix that comes from expanding a Taylor Map around any point that is not the origin of the map is generally not symplectic. The transfer matrix for an element can be symplectified by setting the `symplectify` attribute to True. See section §21.3 for details on how a matrix is symplectified. The default value of `symplectify`, if it is not present, is `False`. If it is present without a value then it defaults to true. Examples:

```
s1: sextupole, l = 0.34                       ! symplectify = False
s1: sextupole, symplectify = True, l = 0.34   ! symplectify = True
s1: sextupole, symplectify, l = 0.34          ! symplectify = True
```

Note that for elements like an `lcavity` where the reference momentum at the downstream end of the element is different from the upstream end, the transfer matrix is never symplectic. In this case, "symplectification" involves first transforming the transfer matrix so that the reference momentum is the same upstream and downstream, then performing symplectification, and finally back transforming the reference momentum to their original values.

5.7 taylor_map_include_offsets Attribute

The `taylor_map_includes_offsets` attribute sets whether the Taylor map generated for an element includes the affect due to the elements (mis)orientation in space. That is, the affect of any pitches, offsets or tilt (§4.6). The default is `True` which means that the Taylor map will include such effects.

How `taylor_map_includes_offsets` is set will not affect the results of tracking or the Jacobian matrix calculation. What is affected is the speed of the calculations. With `taylor_map_includes_offsets` set to `True` the Taylor map will have to be recalculated each time an element is reoriented in space. On the other hand, with `taylor_map_includes_offsets` set to `False` each tracking and Jacobian matrix calculation will include the extra computation involving the effect of the orientation. Thus if an element's orientation is fixed it is faster to set `taylor_map_includes_offsets` to `True` and if the orientation is varying it is faster to set `taylor_map_includes_offsets` to `False`.

If the global parameter `bmad_com%conserve_taylor_maps` (§9.3) is set to True (the default), then, if an element is offset within a program, and if `taylor_map_include_offsets` is set to True for that element, *Bmad* will toggle `taylor_map_include_offsets` to False to conserve the map.

Chapter 6

Beam Lines and Replacement Lists

This chapter describes how to define the ordered list of elements that make up a lattice branch (§1.2). In a lattice, branches may be connected together using `fork` or `photon fork` elements (`s:fork`), or by using `multipass` (§7.2).

6.1 Branch Construction Overview

A lattice branch is defined in a lattice file using what are called `beam lines` (§6.2) and `replacement lists` (§6.5). The `beam lines` are divided into two types - lines with (§6.4) and lines without (§6.2) `replacement arguments`. This essentially corresponds to the *MAD* definition of lines and lists. There can be multiple `beam lines` and `replacement lists` defined in a lattice file and lines and lists can be nested inside other lines and lists.

Since lines can be nexted within other lines, The same element name may be repeated multiple times in a brach. To distinguish between mutiple elements of the same name, lines and lists may be `tagged` (§6.7) to produce unique element names.

There will also be a marker element named `END` automatically placed at the end of the lattice. This end marker will not be automatically placed in the lattice if a marker named `end` is defined in the lattice file at the end of the lattice. Additionally, a `parameter[no_end_marker]` statement (§8.1) can be used to suppress the insertion of the end marker.

6.2 Beam Lines and Lattice Expansion

A `beam line` without arguments has the format

```
label: line = (member1, member2, ...)
```

where `member1`, `member2`, etc. are either elements, other `beam lines` or `replacement lists`, or sublines enclosed in parentheses. Example:

```
line1: line = (a, b, c)
line2: line = (d, line1, e)
use, line2
```

The `use` statement is explained in Section §6.6. This example shows how a `beam line` member can refer to another `beam line`. This is helpful if the same sequence of elements appears repeatedly in the lattice.

The process of constructing the ordered sequences of elements that comprise the branches of the lattice is called `lattice expansion`. In the example above, when `line2` is expanded to form the lattice (in this case there is only one branch so `lattice` and `branch` can be considered synonymous), the definition of `line1` will be inserted in to produce the following lattice:

```
beginning, d, a, b, c, e, end
```

The `beginning` and `end` marker elements are automatically inserted at the beginning and end of the lattice. The `beginning` element will always exist but insertion of the `end` element can be supressed by inserting into the lattice:

```
parameter[no_end_marker] = T    ! See: §8.1
```

Lattice expansion occurs at the end when a lattice file has been parsed or if an `expand_lattice` statement (§2.23) is present.

Each element is assigned an `element index` number starting from 0 for the `beginning` element, 1 for the next element, etc.

In the expanded lattice, any `null_Ele` type elements (§3.34) will be discarded. For example, if element `b` in the above example is a `null_Ele` then the actual expanded lattice will be:

```
beginning, d, a, c, e, end
```

A member that is a line or list can be "reflected" (elements taken in reverse order) if a negative sign is put in front of it. For example:

```
line1: line = (a, b, c)
line2: line = (d, -line1, e)
```

`line2` when expanded gives

```
d, c, b, a, e
```

Reflecting a subline will also reflect any sublines of the subline. For example:

```
line0: line = (y, z)
line1: line = (line0, b, c)
line2: line = (d, -line1, e)
```

`line2` when expanded gives

```
d, c, b, z, y, e
```

A repetition count, which is an integer followed by an asterisk, means that the member is repeated. For example

```
line1: line = (a, b, c)
line2: line = (d, 2*line1, e)
```

`line2` when expanded gives

```
d, a, b, c, a, b, c, e
```

Repetition count can be combined with reflection. For example

```
line1: line = (a, b, c)
line2: line = (d, -2*line1, e)
```

`line2` when expanded gives

```
d, c, b, a, c, b, a, e
```

Instead of the name of a line, subline members can also be given as an explicit list using parentheses. For example, the previous example could be rewritten as

```
line2: line = (d, -2*(a, b, c), e)
```

Lines can be defined in any order in the lattice file so a subline does not have to come before a line that references it. Additionally, element definitions can come before or after any lines that reference them.

A line can have the `multipass` attribute. This is covered in §7.2.

6.3 Element Reversal

An element is `reversed` if particles traveling through it enter at the "exit" end and leave at the "entrance" end. Being able to reverse elements is useful, for example, in describing the interaction region of a pair of rings where particles of one ring are going in the opposite direction relative to the particles in the other ring.

Elment reversal is indicated by using a double negative sign "–" prefix. The double negative sign prefix can be applied to individual elements or to a line. If it is applied to a line, the line is both reflected (same as if a single negative sign is used) and each element is reflected. For example:

```
line1: line = (a, b, --c)
line2: line = (--line1)
line3: line = (c, --b, --a)
```

In this example, `line2` and `line3` are identical. Notice that the reversal of a reversed element makes the element unreversed.

Another example involving element reversal is given in Section §11.6.

Reversed elements, unlike other elements, have their local z-axis pointing in the opposite direction to the local s-axis (§14.1.2). This means that there must be a `reflection patch` (§14.2.6) between reversed and unreversed elements. See §11.3 for an example. Since this complicates matters, it is generally only useful to employ element reversal in cases where there are multiple intersectiong lines with particle beams going in opposite directions through some elements (for example, colliding beam interaction regions). In this case, element reversal is typically used with `multipass` (§7.2).

Where reversed elements are not needed, it is simple to define elements that are effectively reversed. For example:

```
b00: bend, angle = 0.023, e1 = ...
b00_rev: b00, angle = -b00[angle], e1 = -b00[e2], e2 = -b00[e1]
```

and `b00_rev` serves as a reversed version of `b00`.

Internally, *Bmad* associates an `orientation` attribute with each element. This attribute is set to -1 for reversed elements and 1 for unreversed elements. This attribute. If a program can print out the attributes for an element, checking the `orientation` attribute will show if an element is reversed or not.

6.4 Beam Lines with Replaceable Arguments

`Beam lines` can have an argument list using the following syntax

```
line_name(dummy_arg1, dummy_arg2, ...): LINE = (member1, member2, ...)
```

The dummy arguments are replaced by the actual arguments when the line is used elsewhere. For example:

```
line1(DA1, DA2): line = (a, DA2, b, DA1)
line2: line = (h, line1(y, z), g)
```

When `line2` is expanded the actual arguments of `line1`, in this case `(y, z)`, replaces the dummy arguments `(DA1, DA2)` to give for `line2`

```
h, a, z, b, y, g
```

Unlike *MAD*, `beam line` actual arguments can only be elements or `beam lines`. Thus the following is not allowed

```
line2: line = (h, line1(2*y, z), g)   ! NO: 2*y NOT allowed as an argument.
```

6.5 Replacement Lists

When a lattice is expanded, all the lattice members that correspond to a name of a `replacement list` are replaced successively, by the members in the `replacement list`. The general syntax is

```
label: LIST = (member1, member2, ...)
```

For example:

```
list1: list = (a, b, c)
line1: line = (z1, list1, z2, list1, z3, list1, z4, list1)
use, line1
```

When the lattice is expanded the first instance of `list1` in `line1` is replaced by `a` (which is the first element of `list1`), the second instance of `list1` is replaced by `b`, etc. If there are more instances of `list1` in the lattice then members of `list1`, the replacement starts at the beginning of `list1` after the last member of `list1` is used. In this case the lattice would be:

```
z1, a, z2, b, z3, c, z4, a
```

Unlike *MAD*, members of a `replacement list` can only be simple elements without reflection or repetition count and not other lines or lists. For example the following is not allowed:

```
list1: list = (2*a, b)  ! NO: No repetition count allowed.
```

6.6 Use Statement

The particular line or lines that defines the root branches (§1.3) to be used in the lattice is selected by the `use` statement. The general syntax is

```
use, line1, line2 ...
```

For example, `line1` may correspond to one ring and `line2` may correspond to the other ring of a dual ring colliding beam machine. In this case, `multipass` (§7.2) will be needed to describe the common elements of the two rings. Example

```
use, e_ring, p_ring
```

would pick the lines `e_ring` and `p_ring` for analysis. These will be the `root` branches.

`use` statements can come anywhere in the lattice, even before the definition of the lines they refer to. Additionally, there can be multiple `use` statements. The last `use` statement in the file defines which `line` to use.

The total number of branches in the lattice is equal to the number of lines that appear on the `use` statement plus the number of `fork` and `photon_fork` elements that branch to a new branch.

To set such things as the geometry of a branch, beginning Twiss parameters, etc., see Section `s:beginning`.

6.7 Line and List Tags

When a lattice has repeating lines, it can be desirable to differentiate between repeated elements. This can be done by tagging lines with a `tag`. An example will make this clear:

```
line1: line = (a, b)
line2: line = (line1, line1)
use, line2
```

When expanded the lattice would be:

```
a, b, a, b
```

The first and third elements have the same name "a" and the second and fourth elements have the same name "b". Using tags the lattice elements can be given unique names. lines or lists are tagged using brackets `[...]`. The general syntax is:

```
line_name[tag_name]                              ! Syntax for lines
list_name[tag_name]                              ! Syntax for lists
replacement_line[tag_name](arg1, arg2, ...)      ! Syntax for replacement lines.
```

Thus to differentiate the lattice elements in the above example `line2` needs to be modified using tags:

```
line1: line = (a, b)
line2: line = (line1[t1], line1[t2])
use, line2
```

In this case the lattice elements will have names of the form:

```
tag_name.element_name
```

In this particular example, the lattice with tagging will be:

```
t1.a, t1.b, t2.a, t2.b
```

Of course with this simple example one could have just as easily not used tags:

```
t1.a: a;   t2.a: a
t1.b: b;   t2.b: b
line1: line = (t1.a, t1.b, t2.a, t2.b)
use, line2
```

But in more complicated situations tagging can make for compact lattice files.

When lines are nested, the name of an element is formed by concatenating the tags together with dots in between in the form:

```
tag_name1.tag_name2. ... tag_name_n.element_name
```

An example will make this clear:

```
list1 = (g, h)
line1(y, z) = (a, b)
line2: line = (line1[t1](a, b))
line3: line = (line2, list1[hh])
line4: line = (line3[z1], line3[z2])
use, line4
```

The lattice elements in this case are:

```
z1.t1.a, z1.t1.b, z1.hh.g, z2.t1.a, z2.t1.b, z1.hh.h
```

To modify a particular tagged element the lattice must be expanded first (§2.23). For example:

```
line1: line = (a, b)
line2: line = (line1[t1], line1[t2])
use, line2
expand_lattice
t1.b[k1] = 1.37
b[k1] = 0.63       ! This statement does not have any effect
```

After the lattice has been expanded there is no connection between the original `a` and `b` elements and the elements in the lattice like `t1.b`. Thus the last line in the example where the `k1` attribute of`b` is modified do not have any effect on the lattice elements.

Chapter 7

Superposition, and Multipass

This chapter covers two concepts: `superposition` (§7.1) and `multipass` (§7.2) `Superposition` is used when elements overlap spatially. `Multipass` is used when an element is "shared" between branches such as the interaction region shared by two storage rings, or when a beam goes through the same physical element in a branch multiple times as in an energy recovery linac.

In both cases, `lord` and `slave` elements (§1.4) are constructed by *Bmad* to hold the necessary information. In both cases, the `lord` elements will represent the "physical" element while the `slave` elements will embody the "beam path".

7.1 Superposition

In practice the field at a particular point in the lattice may be due to more than one physical element. One example of this is a quadrupole magnet inside a larger solenoid magnet as shown in Fig. 7.1A. *Bmad* has a mechanism to handle this using what is called "superposition". A simple example shows how this works (also see section §1.4):

```
Q: quad, l = 4
D: drift, l = 12
```

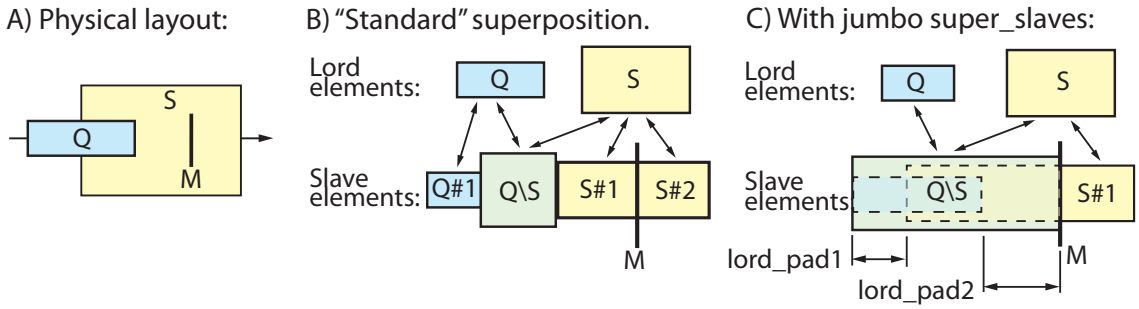


Figure 7.1: Superposition example. A) The physical layout involves a quadrupole partially inside a solenoid. B) The standard superposition procedure involves creating `super_slave` elements whose edges are at the boundaries where the physical elements overlap. C) When jumbo `super_slaves` are created, the `super_slaves` span the entire space where elements overlap.

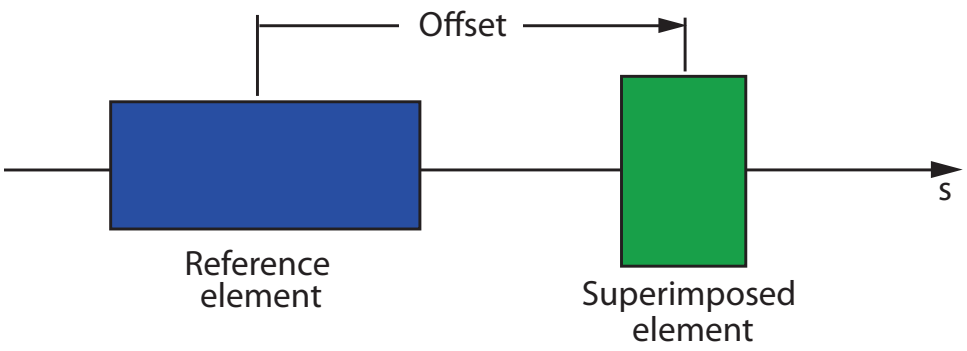


Figure 7.2: The superposition offset is the distance from the origin point of the reference element to the origin point of the element being superimposed.

```
S: solenoid, l = 8, superimpose, ref = Q, ele_origin = beginning
M: marker, superimpose, ref = S, offset = 1
lat: line = (Q, D)
use, lat
```

The `superimpose` attribute of element `S` superimposes `S` over the lattice `(Q, D)`. The placement of `S` is such that the beginning of `S` is coincident with the center of `Q` (this is is explained in more detail below). Additionally, a marker `M` is superimposed at a distance of +1 meter from the center of `S`. The tracking part of the lattice (§1.4) looks like:

```
       Element    Key          Length  Total
1)     Q#1        Quadrupole   2         2
2)     Q\S        Sol_quad     2         4
3)     S#1        Solenoid     3         7
4)     M          Marker       0
4)     S#2        Solenoid     3        10
5)     D#2        Drift        4        14
```

What *Bmad* has done is to split the original elements `(Q, D)` at the edges of `S` and then `S` was split where `M` is inserted. The first element in the lattice, `Q#1`, is the part of `Q` that is outside of `S`. Since this is only part of `Q`, *Bmad* has put a `#1` in the name so that there will be no confusion. (`#` has no special meaning other than the fact that *Bmad* uses it for mangling names). The next element, `Q\S`, is the part of `Q` that is inside `S`. `Q\S` is a combination solenoid/quadrupole element as one would expect. `S#1` is the part of `S` that is outside `Q` but before `M`. This element is just a solenoid. Next comes `M`, `S#1`, and finally `D#2` is the rest of the drift outside `S`.

In the above example, `Q` and `S` will be `super_lord` elements (`s:lord.slave`) and four elements in the tracking part of the lattice will be `super_slave` elements. This is illustrated in Fig. 7.1B.

Notice that the name chosen for the `sol_quad` element `Q\S` is dependent upon what is being superimposed upon what. If `Q` had been superimposed upon `S` then the name would have been `S\Q`.

When *Bmad* sets the element class for elements created from superpositions, *Bmad* will set the class of the element to something other than an `em_field` element (§3.16) if possible. If no other possibilities exist, *Bmad* will use `em_field`. For example, a `quadrupole` superimposed with a `solenoid` will produce a `sol_quad` element but a `solenoid` superimposed with a `rfcavity` element will produce an `em_field` element since there is no other class of element that can simultaneously handle solenoid and RF fields.

With the lattice broken up like this *Bmad* has constructed something that can be easily analyzed. However, the original elements `Q` and `S` still exist within the lord section of the lattice. *Bmad* has

bookkeeping routines so that if a change is made to the `Q` or `S` elements then these changes can get propagated to the corresponding slaves. It does not matter which element is superimposed. Thus, in the above example, `S` could have been put in the Beam Line (with a drift before it) and `Q` could then have been superimposed on top and the result would have been the same (except that the split elements could have different names).

If an element has zero length (for example, a `marker` element), is superimposed, or is superimposed upon, then the element will remain in the tracking part of the lattice and there will be no corresponding lord element. See Fig. 7.1.

Superimpose syntax:

```
Q: quad, superimpose, ...        ! Superimpose element Q/
Q: quad, superimpose = T, ...    ! Same as above.
Q: quad, ...                     ! First define element Q ...
Q[superimpose] = T               !   ... and then superimpose.
Q[superimpose] = F               ! Turn off superposition.
```

The placement of a superimposed element is illustrated in Fig. 7.2. The placement of a superimposed element is determined by three factors: An origin point on the superimposed element, an origin point on the reference element, and an offset between the points. The attributes that determine these three quantities are:

```
create_jumbo_slave = <Logical>     ! See §7.1.2
ref          = <lattice_element>
offset       = <length>            ! default = 0
ele_origin   = <origin_location>   ! Origin pt on element.
ref_origin   = <origin_location>   ! Origin pt on ref element.
```

`ref` sets the reference element. If `ref` is not present then the start of the lattice is used (more precisely, the start of branch 0 (§1.2)). The location of the origin points are determined by the setting of `ele_origin` and `ref_origin`. The possible settings for these parameters are

```
beginning       ! Beginning (upstream) edge of element
center          ! Center of element. Default.
end             ! End (downstream) edge of element
```

`center` is the default setting. `offset` is the longitudinal offset between the origin points. The default offset is zero.

Note: There is an old syntax, deprecated but still supported for now, where the origin points were specified by the appearance of:

```
ele_beginning         ! Old syntax. Do not use.
ele_center            ! Old syntax. Do not use.
ele_end               ! Old syntax. Do not use.
ref_beginning         ! Old syntax. Do not use.
ref_center            ! Old syntax. Do not use.
ref_end               ! Old syntax. Do not use.
```

For example, "ele_origin = beginning" in the old syntax would be "ele_beginning".

The element begin superimposed may be any type of element except `drift`, `group`, `overlay`, and `girder` control elements. The reference element used to position a superimposed element may be a `group` or `overlay` element as long as the `group` or `overlay` controls the attributes of exactly one element. In this case, the controlled element is used as the reference element.

A superimposed element that extends beyond either end of the lattice will be wrapped around so part of the element will be at the beginning of the lattice and part of the element will be at the end. For consistency's sake, this is done even if the `geometry` is set to `open` (for example, it is sometimes convenient to treat a circular lattice as linear). Example:

```
d: drift, l = 10
q: quad, l = 2, superimpose
machine: line = (d)
use, machine
```

The lattice will have three elements in the tracking section:

```
      Element   Key           Length
3)    Q#2       Quadrupole    1
2)    D#1       Drift         8
1)    Q#1       Quadrupole    1
```

The lord section of the lattice will have the element `Q`.

To superimpose a zero length element "`S`" next to a zero length element "`Z`", and to make sure that `S` will be on the correct side of `Z`, set the `ref_origin` appropriately. For example:

```
S1: marker, superimpose, ref = Z, ref_origin = beginning
S2: marker, superimpose, ref = Z, ref_origin = end
Z: marker
```

This will place `S1` upstream and `S2` downstream of `Z`. If `ref_origin` is not present or set to `center`, the ordering of the elements will be arbitrary.

If a superposition uses a reference element, and there are N elements in the lattice with the reference element name, there will be N superpositions. For example, the following will split in two all the quadrupoles in a lattice:

```
M: null_ele, superimpose, ref = quadrupole::*
```

A `null_ele` (§3.34) element is used here so that there is no intervening element between split quadrupole halves as there would be if a `marker` element was used.

When a superposition is made that overlaps a drift the drift, not being a "real" element, vanishes. That is, it does not get put in the lord section of the lattice. Note that if aperture limits (§4.8) have been assigned to a drift, the aperture limits can "disappear" when the superposition is done. Explicitly, if the exit end of a drift has been assigned aperture limits, the limits will disappear if the superimposed element overlays the exit end of the drift. A similar situation applies to the entrance end of a drift. If this is not desired, use a `pipe` element instead.

When the attributes of a super_slave are computed from the attributes of its super_lords, some types of attributes may be "missing". For example, it is, in general, not possible to set appropriate aperture attributes (§4.8) of a super_slave if the lords of the slave have differing aperture settings. When doing calculations, *Bmad* will use the corresponding attributes stored in the lord elements to correctly calculate things.

When superposition is done in a line where there is `element reversal` (§6.3), the calculation of the placement of a superimposed element is also "reversed" to make the relative placement of elements independent of any element reversal. An example will make this clear:

```
d1: drift, l = 1
d2: d1
q1: quad, l = 0.1, superimpose, ref = d1, offset = 0.2,
          ref_origin = beginning, ele_origin = beginning
q2: q1, ref = d2
p: patch, x_pitch = pi  ! Needed to separate reversed and unreversed.
this_line: line = (d1, p, --d2)
use, this_line
```

Since the reference element of the `q2` superposition, that is `d2`, is a reversed element, `q2` will be reversed and the sense of `offset`, `ref_origin`, and `ele_origin` will be reversed so that the position of `q2` with respect to `d2` will be the mirror image of the position of `q1` with respect to `d1`. The tracking part of the lattice will be:

```
Element:           d1#1    q1  d1#2   d2#2    q2   d2#1
Length:             0.2   0.1   0.7    0.7   0.1    0.3
Reversed element?:   No    No    No    Yes   Yes    Yes
```

Notice that, unlike `element reversal`, the rules for how the location of a superimposed element is caculated is not affected by the use of `line reflection` (§6.2). This is done since line reflection mearly changes the order of the elements of the lattice while element reversal physically flips elements front to back.

7.1.1 Superposition and Sub-Lines

Sometimes it is convenient to do simulations with only part of a lattice. The rule for how superpositions are handled in this case is illustrated in the following example. Consider a lattice file which defines a `line` called `full` which is defined by two sublines called `sub1` and `sub2`:

```
sub1: line = ..., ele1, ...
sub2: line = ...
full: line = sub1, sub2
m1: marker, superimpose, ref = ele1, offset = 3.7
use, full
```

Now suppose you want to do a simulation using only the `sub2` line. Rather than edit the original file, one way to do this would be to create a second file which overrides the used line:

```
call, file = 'full.bmad'
use, sub2
```

where `full.bmad` is the name of the original file. What happens to the superposition of `m1` in this case? Since `m1` uses a reference element, `ele1`, that is not in `sub1`, *Bmad* will ignore the superposition. Even though *Bmad* will ignore the superposition of `m1` here, *Bmad* will check that `ele1` has been defined. If `ele1` has not been defined, *Bmad* will assume that there is a typographic error and issue an error message.

Notice that in this case it is important for the superposition to have an explicit reference element since without an explicit reference element the superposition is referenced to the beginning of the lattice. Thus, in the above example, if the superposition were written like:

```
m1: marker, superimpose, offset = 11.3
```

then when the `full` line is used, the superposition of `m1` is referenced to the beginning of `full` (which is the same as the beginning of `sub1`) but when the `sub2` line is used, the superposition of `m1` is referenced to the beginning of `sub2` which is not the same as the beginning of `full`.

7.1.2 Jumbo super_slaves

The problem with the way `super_slave` elements are created as discussed above is that edge effects will not be dealt with properly when elements with non-zero fields are misaligned. When this is important, especially at low energy, a possible remedy is to instruct *Bmad* to construct "`jumbo`" super_slave elements. The general idea is to create one large `super_slave` for any set of overlapping elements. Returning to the superposition example at the start of Section §7.1, If the superposition of solenoid `S` is modified to be

```
S: solenoid, l = 8, superimpose, ref = Q, ele_origin = beginning,
             create_jumbo_slave = T
```

The result is shown in Fig. 7.1C. The tracking part of the lattice will be

```
        Element   Key         Length  Total
  1)    Q\S       Sol_quad     2        4
  2)    M         Marker       0
  3)    S#2       Solenoid     3       10
  4)    D#2       Drift        4       14
```

`Q` and part of `S` have been combined into a jumbo `super_slave` named `Q\S`. Since the `super_lord` elements of a jumbo `super_slave` may not completely span the slave two attributes of each lord will be set to show the position of the lord within the slave. These two attributes are

```
  lord_pad1    ! offset at upstream end
  lord_pad2    ! offset at downstream end
```

`lord_pad1` is the distance between the upstream edge of the jumbo `super_slave` and a `super_lord`. `lord_pad2` is the distance between the downstream edge of a `super_lord` and the downstream edge of the jumbo `super_slave`. With the present example, the lords have the following padding:

```
          lord_pad1    lord_pad2
  Q           0            3
  S           2            0
```

The following rule holds for all super lords with and without jumbo slaves:

```
  Sum of all slave lengths = lord length + lord_pad1 + lord_pad2
```

One major drawback of jumbo `super_slave` elements is that the `tracking_method` (§5.1) will, by necessity, have to be `runge_kutta`, or `time_runge_kutta` and the `mat6_calc_method` (§5.2) will be set to `tracking`.

Notice that the problem with edge effects for non-jumbo `super_slave` elements only occurs when elements with nonzero fields are superimposed on top of one another. Thus, for example, one does not need to use jumbo elements when superimposing a `marker` element.

Another possible way to handle overlapping fields is to use the `field_overlaps` element attribute as discussed in §4.17.

7.1.3 Changing Element Lengths when there is Superposition

When a program is running, if `group` (§3.21) or `overlay` (§3.36) elements are used to vary the length of elements that are involved in superimposition, the results are different from what would have resulted if instead the lengths of the elements where changed in the lattice file. There are two reasons for this. First, once the lattice file has been parsed, lattices can be "mangled" by adding or removing elements in a myriad of ways. This means that it is not possible to devise a general algorithm for adjusting superimposed element lengths that mirrors what the effect of changing the lengths in the lattice file.

Second, even if a lattice has not been mangled, an algorithm for varying lengths that is based on the superimpose information in the lattice file could lead to unexpected results. To see this consider the first example in Section §7.1. If the length of `S` is varied in the lattice file, the upstream edge of `S` will remain fixed at the center of `Q` which means that the length of the `super_slave` element `Q#1` will be invariant. On the other hand, if element `S` is defined by

```
  S: solenoid, l = 8, superimpose, offset = 6
```

This new definition of `S` produces produce exactly the same lattice as before. However, now varying the length of `S` will result in the center of `S` remaining fixed and the length of `Q#1` will not be invariant with changes of the length of `S`. This variation in behavior could be very confusing since, while running a program, one could not tell by inspection of the element positions what should happen if a length were changed.

To avoid confusion, *Bmad* uses a simple algorithm for varying the lengths of elements involved in superposition: The rule is that the length of the most downstream `super_slave` is varied. With the first example in Section §7.1, the `group G` varying the length of `Q` defined by:

```
G: group = {Q}, var = {l}
```

would vary the length of `Q\S` which would result in an equal variation of the length of `S`. To keep the length of `S` invariant while varying `Q` the individual `super_slave` lengths can be varied. Example:

```
G2: group = {Q#1, S#1:-1}, var = {l}
```

The definition of `G2` must be placed in the lattice file after the superpositions so that the super slaves referred to by `G2` have been created.

In the above example there is another, cleaner, way of achieving the same result by varying the downstream edge of `Q`:

```
G3: group = {Q}, var = {end_edge}
```

7.2 Multipass

Some lattices have the beam recirculating through the same element multiple times. For example, an Energy Recovery Linac (ERL) will circulate the beam back through the LINAC part to retrieve the energy in the beam. In *Bmad*, this situation can simulated by designating the LINAC section as `multipass`. A simple example shows how this works.

```
RF1: lcavity
linac: line[multipass] = (RF1, ...)
erl: line = (linac, ..., linac)
use, erl
expand_lattice
RF1\2[phi0_multipass] = 0.5
```

The line called `linac` is designated as `multipass`. This `linac` line appears twice in the line `erl` and `erl` is the root line for lattice expansion. The lattice constructed from `erl` will have two `RF1` elements in the tracking part of the lattice:

```
RF1\1, ..., RF1\2, ...
```

Since the two elements are derived from a `multipass` line, they are given unique names by adding a `\n` suffix. These types of elements are known as `multipass_slave` elements. In addition, to the `multipass_slave` elements, there is a `multipass_lord` element (that doesn't get tracked through) called `RF1` in the lord part of the lattice (§1.4). Changes to attributes of the lord `RF1` element will be passed to the slave elements by *Bmad*'s bookkeeping routines. Assuming that the phase of `RF1\1` gives acceleration, to make `RF1\2` decelerate the `phi0_multipass` attribute of `RF1\2` is set to 0.5. This is the one attribute that *Bmad*'s bookkeeping routines will not touch when transferring attribute values from `RF1` to its slaves. Notice that the `phi0_multipass` attribute had to be set after `expand_lattice` (§2.23) is used to expand the lattice. This is true since *Bmad* does immediate evaluation and `RF1\2` does not exist before the lattice is expanded. `Phi0_multipass` is useful with relative time tracking §22.1. However, `phi0_multipass` is "unphysical" and is just a convienient way to shift the phase pass-to-pass through a given cavity. To "correctly" simulate the recirculating beam, absolute time tracking should be used and the length of the lattice from a cavity back to itself needs to be properly adjusted to get the desired phase advance. See the discussion in section §22.1.

Multiple elements of the same name in a multipass line are considered physically distinct. Example:

```
m_line: line[multipass] = (A, A, B)
u_line: line = (m_line, m_line)
use, u_line
```

In this example the tracking part of the lattice is

```
A\1, A\1, B\1, A\2, A\2, B\2
```

In the control section of the lattice there will be two multipass lords called `A` and one called `B`. [That is, *Bmad* considers the lattice to have three physically distinct elements.] The first `A` lord controls the 1^{st} and 4^{th} elements in the tracking part of the lattice and the second `A` lord controls the 2^{nd} and 5^{th} elements. If `m_line` was *not* marked `multipass`, the tracking part of the lattice would have four `A` and two `B` elements and there would be no lord elements.

Sublines contained in a multipass line that are themselves not marked multipass act the same as if the elements of the subline where substituted directly in place of the subline in the containing line. For example:

```
a_line: line = (A)
m_line: line[multipass] = (a_line, a_line, B)
u_line: line = (m_line, m_line)
use, u_line
```

In this example, `a_line`, which is a subline of the multipass `m_line`, is *not* designated `multipass` and the result is the same as the previous example where `m_line` was defined to be `(A, A, B)`. That is, there will be three physical elements represented by three multipass lords.

Multipass lines do not have to be at the same "level" in terms of nesting of lines within lines. Additionally, multipass can be used with line reversal (§6.3). Example:

```
m_line: line[multipass] = (A, B)
m2_line: line = (m_line)
P: patch, ...
arc: line = (..., P)
u_line: line = (m_line, arc, --m2_line)
use, u_line
```

Here the tracking part of the lattice is

```
A\1, B\1, ..., B\2 (r), A\2 (r)
```

The "(r)" here just denotes that the element is reversed and is not part of the name. The lattice will have a multipass lord `A` that controls the two `A\n` elements and similarly with `B`. This lattice represents the case where a particle goes through the m_line in the "forward" direction, gets turned around in the `arc` line, and then passes back through `m_line` in the reverse direction. While it is possible to use reflection "−" (§6.2) instead of reversal "−−" (§6.3), reflection here does not make physical sense. Needed here is a reversing patch `P` (§3.37) between reversed and unreversed elements.

The procedure for how to group lattice elements into multipass slave groups which represent the same physical element is done by grouping all lattice elements that have the same multipass "`signature`". For any given element in the lattice, this element has some line it came from. Call this line L_0 and denote by n_0 the index where the element under consideration is in L_0 line (the first element in L_0 has index 1, etc.). The L_0 line in turn was contained in some other line. Call this line L_1 and denote by n_1 the position of L_0 in L_1. This chain of lines L_0, L_1, ..., L_n ends at some point and the last (top) line L_n will be one of the root lines listed in the `use` statement (§6.6). For any given element in the lattice, starting with L_0 and proceeding upwards through the chain, let L_m be the first line in the chain that is marked as `multipass`. If no such line exists for a given element, L_m is taken to be the top line L_n. The `signature` of the element is the list of (line, position) pairs (L_0, n_0), ..., (L_m, n_m) and two elements have the same signature if this list is the same for both. Elements that have the same signature represent the same physical element and are slaved together. For example, using the example above, the first element of the lattice, `A\1`, has the chain of (line, position):

```
(L0, n0) = (m_line, 1)
(L1, n1) = (u_line, 1)
```

The last element in the lattice, (`A\2`), has the chain

```
(L0, n0) = (m_line, 1)
(L1, n1) = (m2_line, 1)
(L2, n2) = (u_line, 3)
```

The signature for both elements is just the single (line, position) pair `(m_line, 1)`. Since the signatures are identical, the two elements will be slaved together.

As a final example, consider the case where a subline of a multipass line is also marked `multipass`:

```
a_line: line[multipass] = (A)
m_line: line[multipass] = (a_line, a_line, B)
u_line: line = (m_line, m_line)
use, u_line
```

Here `a_line` is marked as `multipass` so the `A` element that is contained in it has the signature `(a_line, 1)`. In this case the tracking part of the lattice will be:

```
A\1, A\2, B\1, A\3, A\4, B\2
```

There will be two lord elements representing the two physically distinct elements `A` and `B`. The `A` lord element will will control the four `A\n` elements in the tracking part of the lattice. The `B` lord will control the two `B\n` elements in the tracking part of the lattice.

To simplify the constructed lattice, if the set of lattice elements to slave together only contains one element, a multipass lord is not constructed. For example:

```
m_line: line[multipass] = (A, A, B)
u_line: line = (m_line)
use, u_line
```

In this example no multipass lords are constructed and the lattice is simply

```
A, A, B
```

It is important to note that the global coordinates (§14.2) of the slaves of a given multipass lord are not constrained by *Bmad* to be the same. It is up to the lattice designer to make sure that the physical positions of the slaves makes sense (that is, are the same).

7.2.1 The Reference Energy in a Multipass Line

If there are `lcavity` elements in the lattice then the reference energy at a given element may differ from pass to pass. In this case, the normalized strength (k1, kick, etc.) for magnetic and electric elements will not be the same from pass to pass. To avoid an ambiguity, all magnetic and electric elements that are used in a multipass line must have their magnetic or electric field strength set as the independent attribute (§4.1), *or* a reference energy (§4.5) must be defined. A reference energy is defined in a multipass element by setting `e_tot` or `p0c`, or by setting `n_ref_pass` to 1. Exception: For `em_field`, `lcavity`, and `custom` elements where the reference energy may change, `e_tot_start` and `p0c_start` are used in place of `e_tot` and `p0c`.

Setting `n_ref_pass` to 1 means that the reference energy in the lord element is set using the reference energy as computed for the first pass slave elements. Previously, `n_ref_pass` could be set to a number N greater than 1 to indicate that the reference energy would be computed for the lord using the reference energy of N^{th} pass slaves. However, the bookkeeping proved to be too complicated so now this is not allowed. Currently, the only permitted values of `n_ref_pass` are 0 and 1 with 0 indicating that the lord reference energy is set directly in the lattice file. Note: If `e_tot` or `p0c` is set, `n_ref_pass` will default to 0 and it is an error to set it to 1. If neither `e_tot` nor `p0c` is set, `n_ref_pass` will default to 1 and it is an error to set it to 0.

An example of an ERL lattice with multipass can be found in Section §11.2.

Chapter 8

Lattice File Global Parameters

This chapter deals with statements that can be used to set "global" parameter values. That is, parameter values that are associated with the lattice as a whole and not simply associated with a single element.

8.1 Parameter Statements

`Parameter` statements are used to set a number of global variables. If multiple branches are present (§1.2), these variables pertain to the `root` branch. The variables that can be set by `parameter` are

```
  parameter[absolute_time_tracking]   = <Logical>  ! Absolute time used for RF clock?
  parameter[custom_attributeN]        = <string>   ! Defining custom attributes (§2.9).
  parameter[default_tracking_species] = <Switch>   ! Default type of tracked particle.
                                                   !    Default is ref_particle.
  parameter[e_tot]                    = <Real>     ! Reference total Energy.
                                                   !      Default: 1000 * rest_energy.
  parameter[electric_dipole_moment]   = <Real>     ! Particle electric dipole moment.
  parameter[live_branch]              = <Logical>  ! Is branch fit for tracking?
  parameter[geometry]                 = <Switch>   ! Open or closed
  parameter[lattice]                  = <String>   ! Lattice name.
  parameter[machine]                  = <String>   ! Machine name.
  parameter[n_part]                   = <Real>     ! Number of particles in a bunch.
  parameter[no_end_marker]            = <Logical>  ! Default: False.
  parameter[p0c]                      = <Real>     ! Reference momentum.
  parameter[particle]                 = <Switch>   ! Reference species: positron, proton, etc.
  parameter[photon_type]              = <Switch>   ! Incoherent or coherent photons?
  parameter[ptc_exact_model]          = <Logical>  ! PTC to do "exact" tracking?
  parameter[ptc_exact_misalignment]   = <Logical>  ! PTC to "exactly" misalign elements?
  parameter[ptc_max_fringe_order]     = <Integer>  ! Max fringe order.
                                                   !    Default: 2 => Quadrupole.
  parameter[ran_seed]                 = <Integer>  ! Random number generator init.
  parameter[taylor_order]             = <Integer>  ! Default: 3
  parameter[use_hard_edge_drifts]     = <Logical>
  parameter[high_energy_space_charge_on] = <logical>
```

Examples

```
  parameter[lattice]      = "L9A19C501.FD93S_4S_15KG"
```

```
parameter[geometry]      = closed
parameter[taylor_order] = 5
parameter[E_tot]         = 5.6e9    ! eV
```

parameter[absolute_time_tracking]
The absolute_time_tracking switch sets whether the clock for the lcavity and rfcavity elements is tied to the reference particle or to uses the absolute time (§22.1). A value of False (the default) mandates relative time and a value of True mandates absolute time. The exception is that for an e_gun element (§3.14), absolute time tracking is always used in order to be able to avoid problems with a zero reference momentum at the beginning of the element.

parameter[custom_attributeN]
Here N is an integer between 1 and 40. For more information on defining custom attributes, see §2.9. Name of the machine the lattice simulates. Example: "LHC".

parameter[live_branch]
Setting live_branch to False (default is True) indicates to a program that no tracking or other analysis of the root branch should be done. This can be useful if the lattice has multiple branches and analysis of the root branch is not necessary. Other branches can also be marked as alive/dead using line parameter statements (§8.4). Note that the *Bmad* library itself ignores the setting of live_branch and it is up to the program being run to decide if this parameter is ignored or not. In particular, the *Tao* program (§1.6) *will* respect the setting of live_branch.

parameter[default_tracking_species]
The parameter[default_tracking_species] switch establishes the default type of particles to be tracked. Possible setting include all the settings of parameter[particle]. In addition, this switch can be set to:

```
ref_particle     ! default
anti_ref_particle
```

By default, default_tracking_species is set to ref_particle so that the particle being tracked is the same as the reference particle set by param[particle]. In the case, for example, where there are particles going one way and antiparticles going the another, default_tracking_species can be used to switch between tracking the particles or antiparticles.

parameter[e_tot], parameter[p0c]
The parameter[e_tot] and parameter[p0c] are the reference total energy and momentum at the start of the lattice. Each element in a lattice has an individual reference e_tot and p0c attributes which are dependent parameters. The reference energy and momentum will only change between LCavity or Patch elements. The starting reference energy, if not set, will be set to 1000 time the particle rest energy. Note: beginning[e_tot] and beginning[p0c] (§8.4) are equivalent to parameter[e_tot] and parameter[p0c].

parameter[electric_dipole_moment]
The electric_dipole_moment sets the electric dipole moment value η for use when tracking with spin (§19.1).

parameter[geometry]
Valid geometry settings are

```
closed  ! Default w/o LCavity element present.
open    ! Default if LCavity elements present.
```

A machine with a closed geometry is something like a storage ring where the particle beam recirculates through the machine. A machine with an open geometry is something like a linac. In this case, if the reference particle is not a photon, the initial Twiss parameters need to be specified

in the lattice file using the `beginning` statement (§8.4). If the `geometry` is not specified, `closed` is the default. The exception is that if there is an `Lcavity` element present or the reference particle is a photon, `open` will be the default.

Notice that by specifying a `closed` geometry it *does* not mean that the downstream end of the last element of the lattice has the same global coordinates (§14.2) as the global coordinates at the beginning. Setting the geometry to `closed` simply signals to a program to compute closed orbits and periodic Twiss parameters as opposed to calculating orbits and Twiss parameters based upon initial orbit and Twiss parameters at the beginning of the lattice. And indeed, it is sometimes convenient to treat lattices as closed even though there is no closure in the global coordinate sense. For example, when a machine has a number of repeating "super-periods", it may be convenient to only use one period in a simulation. Since *Bmad* ignores closure in the global coorindate sense, it is up to the user to ensure that a lattice is closed in the global sense if that is desired.

Note: `geometry` used to be called `lattice_type`, `closed` used to be called `circular_lattice` and `open` used to be called `linear_lattice`.

high_energy_space_charge_on
Toggle to turn on or off the ultra-relativistic space charge effect in particle tracking (§17.5). Computationally, this is separate from the lower energy space charge and CSR calculation (§17.4). Default is `False`. Notice that including the high energy space charge can be done on a branch-by-branch basis (§8.4).

parameter[lattice]
Used to set the lattice name. The `lattice` name is stored by *Bmad* for use by a program but it does not otherwise effect any *Bmad* routines.

parameter[n_part]
The `parameter[n_part]` is the number of particle in a bunch. it is used with `BeamBeam` elements and is used to calculate the change in energy through an `Lcavity`. See §3.26 for more details.

parameter[no_end_marker]
The `parameter[no_end_marker]` is use to suppress the automatic inclusion of a marker named `END` at the end of the lattice (§6.1).

parameter[p0c]
See `parameter[e_tot]`.

parameter[particle]
The `parameter[particle]` switch sets the reference species. The possible settings for this attribute can be divided into four groups. One group are are fundamental particles. These are:

```
electron,  positron,
muon,      antimuon,
proton,    antiproton,
photon,
pion+,      pion0,      pion-
deuteron
```

Names for the fundamental particles are *not* case sensitive.

Another group are atoms. The general syntax for atoms is:

```
{#nnn}AA{ccc}
```

The curly brackets {...} denote optional prefixes and suffixes. `AA` here is the atomic symbol, `#nnn` is the number of nucleons, and `ccc` is the charge. Examples:

```
parameter[particle] = #12C+3       ! Triply charged carbon-12
parameter[particle] = He--          ! Doubly charged He.
```

If the number of nucleons is given, the appropriate weight for that isotope is used. If the number of nucleons is not present, the mass is an average weighted by the isotopic abundances of the element. The charge may be given by using the appropriate number of plus (+) or minus (-) signs or by using a plus or minus sign followed by a number. Thus "---" is equivalent to "-3". Names here are case sensitive. "@M" must be used and not "@m" for specifying the mass.

Another group of particles are the "known" molecules. The syntax for these are:

```
BBB{@Mmmmm}{ccc}
```

`@Mmmmm` is the mass in AMU, `ccc` is the charge, and `BBB` is the molecular formula. The mass may to specified to hundredths of an AMU. The known molecules are:

```
CO        CO2
D2        D2O
OH        O2
H2        H2O       HF
N2        NH2       NH3
CH2       CH3       CH4
C2H3      C2H4      C2H5
```

Like with atoms, if the mass is not specified, the average isotopic mass is used. Examples:

```
C2H3@M28.4+     ! Singly charged C2H3 with mass of 28.4
CH2             ! Neutral CH2
```

Like the atomic formulas, molecular formulas are case sensitive.

The last group of particle are particles where only the mass and charge are specified. The syntax for these are:

```
@Mmmmm{ccc}
```

The setting of the reference particle is used, for example, to determine the direction of the field in a magnet and given the normalized field strength (EG: `k1` for a quadrupole). Generally, the particles that by default are tracked through a lattice are the same as the reference particle. This default behavior can be altered by setting `parameter[default_tracking_species]`.

parameter[photon_type]

The `photon_type` switch is used to set the type of photons that are used in tracking. Possible settings are:

```
incoherent    ! Default
coherent
```

The general rule is use incoherent tracking except when there is a `diffraction_plate` element in the lattice.

parameter[ptc_exact_model]

The `ptc_exact_model` and `ptc_exact_misalign` switches affect tracking using the `PTC` library. See §5.4 for more details.

parameter[ptc_max_fringe_order]

When using PTC tracking (§1.5), the `parameter[ptc_max_fringe_order]` determines the maximum order of the calculated fringe fields. The default is 2 which means that fringe fields due to a quadrupolar field. These fields are 3^{rd} order in the transverse coordinates.

parameter[ran_seed]

For more information on `parameter[ran_seed]` see §2.13.

parameter[taylor_order]

The Taylor order (§21.1) is set by `parameter[taylor_order]` and is the maximum order for a Taylor map.

parameter[use_hard_edge_drifts]
The `use_hard_edge_drifts` switch determines if a "hard edge" model of certain elements is used. For example, if `runge_kutta` tracking is used for an `rfcavity` or `lcavity` using a standing wave model then the cavity length should be a multiple of the RF wavelength/2. To achieve this, during tracking, the cavity length is appropriately modified and drifts are inserted at either ends of the cavity to keep the total length constant. Default is `True`.

8.2 Particle_Start Statements

`particle_start` statements are used, among other things to set the starting coordinates for particle tracking. If multiple branches are present (§1.2), these variables pertain to the `root` branch.

```
particle_start[x]                = <Real>   ! Horizontal position.
particle_start[px]               = <Real>   ! Horizontal momentum.
particle_start[y]                = <Real>   ! Vertical position.
particle_start[py]               = <Real>   ! Vertical momentum.
particle_start[z]                = <Real>   ! Longitudinal position.
particle_start[pz]               = <Real>   ! Momentum deviation. Only for non-photons.
particle_start[direction]        = +/-1     ! Longitudinal direction of travel.
particle_start[E_photon]         = <Real>   ! Energy (eV). Only used for photons.
particle_start[emittance_a]      = <Real>   ! A-mode emittance
particle_start[emittance_b]      = <Real>   ! B-mode emittance
particle_start[emittance_z]      = <Real>   ! Z-mode emittance
particle_start[sig_z]            = <Real>   ! Beam sigma in z-direction
particle_start[sig_pz]           = <Real>   ! Beam Sigma pz
particle_start[field_x]          = <Real>   ! Photon beam field along x-axis
particle_start[field_y]          = <Real>   ! Photon beam field along y-axis
particle_start[phase_x]          = <Real>   ! Photon beam phase along x-axis
particle_start[phase_y]          = <Real>   ! Photon beam phase along y-axis
particle_start[t]                = <Real>   ! Absolute time
particle_start[spin_x]           = <Real>   ! Spin polarization x-coordinate
particle_start[spin_y]           = <Real>   ! Spin polarization y-coordinate
particle_start[spin_z]           = <Real>   ! Spin polarization z-coordinate
```

Normally the absolute time, set by `particle_start[t]`, is a dependent parameter set by solving Eq. (14.28) for t. The exception is when the initial velocity is zero. (This can happen if there is an `e_gun` (§3.14) element in the lattice). In this case, z must be zero and t is an independent parameter that can be set.

The longitudinal direction of travel is set by `particle_start[direction]`. This can be set to +1 (travel in the +s direction) or -1 for the reverse. +1 is the default. Generally `particle_start[direction]` should not be set to -1 since most programs will not be constructed to handle this situation. To track a particle in the reverse direction see §11.6.

For particles with spin, the spin can be specified using Cartiesian coordinates with `spin_x`, `spin_y`, and `spin_z`.

For photons, `px`, `py`, and `pz` are the normalized velocity components (Cf. Eq. (14.38)). For photons `pz` is a dependent parameter which will be set so that Eq. (14.39) is obeyed.

Note: `particle_start` used to be called `beam_start`. Since this was confusing (beam initialization parameters are stored in a separate `beam_init_struct` structure (§10.1)), the name was changed. Currently the use of the `beam_start` name is deprecated but still supported for backwards compatability.

Example

```
particle_start[y] = 2 * particle_start[x]
```

8.3 Beam Statement

The `beam` statement is provided for compatibility with *MAD*. The syntax is

```
beam, energy = GeV, pc = GeV, particle = <Switch>, n_part = <Real>
```

For example

```
beam, energy = 5.6  ! Note: GeV to be compatible with MAD
beam, particle = electron, n_part = 1.6e10
```

Setting the reference energy using the `energy` attribute is the same as using `parameter[e_tot]`. Similarly, setting `pc` is equivalent to setting `parameter[p0c]`. Valid `particle` switches are the same as `parameter[particle]`.

8.4 Beginning and Line Parameter Statements

For non–circular lattices, the `beginning` statement can be used to set the Twiss parameters and beam energy at the beginning of the first lattice branch.

```
beginning[alpha_a]  = <Real>  ! "a" mode alpha
beginning[alpha_b]  = <Real>  ! "b" mode alpha
beginning[beta_a]   = <Real>  ! "a" mode beta
beginning[beta_b]   = <Real>  ! "b" mode beta
beginning[cmat_ij]  = <Real>  ! C coupling matrix. i, j = {``1'', or ``2''}
beginning[e_tot]    = <Real>  ! Reference total energy in eV.
beginning[eta_x]    = <Real>  ! x-axis dispersion
beginning[eta_y]    = <Real>  ! y-axis dispersion
beginning[etap_x]   = <Real>  ! x-axis dispersion derivative.
beginning[etap_y]   = <Real>  ! y-axis dispersion derivative.
beginning[p0c]      = <Real>  ! Reference momentum in eV.
beginning[phi_a]    = <Real>  ! "a" mode phase.
beginning[phi_b]    = <Real>  ! "b" mode phase.
beginning[ref_time] = <Real>  ! Starting reference time.
beginning[s]        = <Real>  ! Longitudinal starting position.
```

The `gamma_a`, `gamma_b`, and `gamma_c` (the coupling gamma factor) will be kept consistent with the values set. If not set the default values are all zero. `beginning[e_tot]` and `parameter[e_tot]` are equivalent and one or the other may be set but not both. Similarly, `beginning[p0c]` and `parameter[p0c]` are equivalent.

For any lattice the `beginning` statement can be used to set the starting floor position of the first lattice branch (see §14.2). The syntax is

```
beginning[x_position]     = <Real>  ! X position
beginning[y_position]     = <Real>  ! Y position
beginning[z_position]     = <Real>  ! Z position
beginning[theta_position] = <Real>  ! Angle on floor
beginning[phi_position]   = <Real>  ! Angle of attack
beginning[psi_position]   = <Real>  ! Roll angle
```

If the floor position is not specified, the default is to place beginning element at the origin with all angles set to zero.

The `beginning` statement is useful in situations where only parameters for the first branch need be specified. If this is not the case, the parameters for any branch can be specified using a statement of the form

```
line_name[parameter] = <Value>
```

This construct is called a `line parameter` statement Here `line_name` is the name of a line and `parameter` is the name of a parameter. The parameters that can be set here are the same parameters that can be set with the `beginning` statement with the additional parameters from the `parameter` statement:

```
default_tracking_species
geometry
high_energy_space_charge_on
live_branch
particle
```

Example:

```
x_ray_fork: fork, to_line = x_ray
x_ray = (...)
x_ray[E_tot] = 100
```

Rules:

1. The floor position of a line can only be set if the line is used for a `root branch`.
2. Line parameters statements must come after the associated line. This rule is similar to the rule that element attribute redefinitions must come after the definition of the element.

Chapter 9

Parameter Structures

9.1 What is a Structure?

A "structure" is a collection of parameters. *Bmad* has various structures which can be used for various tasks. For example, the `beam_init_struct` structure (§10.1) is used to set parameters used to initialize particle beams.

A given program may give the user access to some of these structures so, in order to allow intelligent parameter setting, this chapter gives an in-depth description of the most common ones.

Each structure has a **''structure name''** (also called a **''type name''**) which identifies the list of parameters (also called "components") in the structure. Associated with a structure there will be an **''instance''** of this structure and this instance will have an **''instance name''** which is what the user uses to set parameters. It is possible to have multiple instances of a structure. For example, in the situation where a program is simulating multiple particle beams, there could be multiple `beam_init_struct` (§10.1) instances with one for each beam.

Bmad defines uses some structures to hold global parameters. That is, parameters that shared by all code. Each of these structures has a single associated instance. These are:

Structure	Instance
bmad_common_stuct	bmad_com
csr_parameter_stuct	csr_param

All other structures will have instance names that are program specific. That is, see the program documentation for the instance name(s) used.

To set a particular component of an instance use the syntax

```
instance_name%parameter_name = value
```

Example:

```
bmad_com%max_aperture_limit = 10
```

this sets the `max_aperture_limit` parameter of `bmad_com`.

9.2 Bmad_Common_Struct

The bmad_common_struct structure contains a set of global parameters. There is only one global instance (§9) of this structure and this instance has the name bmad_com. The components of this structure along with the default values are:

```
type bmad_common_struct
  real(rp) max_aperture_limit = 1e3           ! Max Aperture.
  real(rp) d_orb(6)           = 1e-5          ! for the make_mat6_tracking routine.
  real(rp) default_ds_step    = 0.2           ! Integration step size.
  real(rp) significant_length = 1e-10         ! meter
  real(rp) rel_tol_tracking = 1e-8            ! Closed orbit relative tolerance.
  real(rp) abs_tol_tracking = 1e-10           ! Closed orbit absolute tolerance.
  real(rp) rel_tol_adaptive_tracking = 1e-8   ! Runge-Kutta tracking relative tolerance.
  real(rp) abs_tol_adaptive_tracking = 1e-10  ! Runge-Kutta tracking absolute tolerance.
  real(rp) init_ds_adaptive_tracking = 1e-3   ! Initial step size.
  real(rp) min_ds_adaptive_tracking = 0       ! Minimum step size to use.
  real(rp) fatal_ds_adaptive_tracking = 1e-8  ! Threshold for loosing particles.
  real(rp) autoscale_amp_abs_tol = 0.1_rp     ! Autoscale absolute amplitude tolerance (eV).
  real(rp) autoscale_amp_rel_tol = 1d-6       ! Autoscale relative amplitude tolerance
  real(rp) autoscale_phase_tol = 1d-5         ! Autoscale phase tolerance.
  real(rp) electric_dipole_moment = 0         ! Particle's EDM. Call set_ptc to transfer value to PTC
  real(rp) ptc_cut_factor = 0.006             ! Cut factor for PTC tracking
  real(rp) sad_eps_scale = 5.0d-3             ! Used in sad_mult step length calc.
  real(rp) sad_amp_max = 5.0d-2               ! Used in sad_mult step length calc.
  integer sad_n_div_max = 1000                ! Used in sad_mult step length calc.
  integer taylor_order = 3                    ! 3rd order is default
  integer default_integ_order = 2             ! PTC integration order
  integer ptc_max_fringe_order = 2            ! PTC max fringe order (2 => Quadrupole !).
  integer max_num_runge_kutta_step = 10000    ! Max num RK steps before particle is lost.
  integer space_charge_mesh_size = (32,32,64) ! Mesh size with fft_3d space charge calc.
  logical rf_phase_below_transition_ref = F   ! Autoscale around phase phi0 = 0.5
  logical use_hard_edge_drifts = T            ! Insert drifts when tracking through cavity?
  logical sr_wakes_on = T                     ! Short range wake fields?
  logical lr_wakes_on = T                     ! Long range wake fields
  logical mat6_track_symmetric = T            ! symmetric offsets
  logical auto_bookkeeper = T                 ! Automatic bookkeeping?
  logical csr_and_space_charge_on = F         ! CSR and space charge (separate from HE SC).
  logical spin_tracking_on = F                ! spin tracking?
  logical spin_sokolov_ternov_flipping_on = F ! Spin flipping during radiation emission?
  logical radiation_damping_on = F            ! Damping toggle.
  logical radiation_fluctuations_on = F       ! Fluctuations toggle.
  logical conserve_taylor_maps = T            ! Enable bookkeeper to set
                                              ! ele%taylor_map_includes_offsets = F?
  logical absolute_time_tracking_default = F  ! Default for lat%absolute_time_tracking
  logical aperture_limit_on = T               ! Use aperture limits in tracking.
  logical debug = F                           ! Used for code debugging.
end type
```

Note: bmad_com parameters may always be set in a lattice file as discussed in Section §9.3. However, thought must be given to setting bmad_com parameters in a lattice file since that will affect every program that uses the lattice.

Parameter description:

`%abs_tol_adaptive_tracking`
: Absolute tolerance to use in adaptive tracking. This is used in `runge-kutta` and `time_runge_kutta` tracking (§5.4).

`%abs_tol_tracking`
: Absolute tolerance to use in tracking. Specifically, Tolerance to use when finding the closed orbit.

`%absolute_time_tracking_default`
: Default setting to be applied to a lattice if `absolute_time_tracking` (§8.1) is not specified in a lattice file. Additionally, if an element that is not associated with a lattice is tracked, `%absolute_time_tracking_default` will be used to determine whether absolute time tracking is used.

  To change between absolute and relative time tracking (§22.1) after lattice file parsing, the `%absolute_time_trackin` component of a `lat_struct` (§32.12) can be appropriately set.

`%aperture_limit_on]`
: Aperture limits may be set for elements in the lattice (§4.8). Setting `aperture_limit_on` to `False` will disable all set apertures. `True` is the default.

`%auto_bookkeeper`
: Toggles automatic or intelligent bookkeeping. See section §28.6 for more details.

`%autoscale_amp_abs_tol`
: Used when *Bmad* autoscales (§4.18) an elements field amplitude. This parameter sets the absolute tolerance for the autoscale amplitude parameter.

`%autoscale_amp_rel_tol`
: Used when *Bmad* autoscales (§4.18) an elements field amplitude. This parameter sets the relative tolerance for the autoscale amplitude parameter. Used when *Bmad* autoscales (§4.18) an elements AC phase. This parameter sets the absolute tolerance for the autoscale parameter.

`%autoscale_phase_tol`

`%init_ds_adaptive_tracking`
: Initial step to use for adaptive tracking. This is used in `runge-kutta` and `time_runge_kutta` tracking (§5.4).

`%conserve_taylor_maps`
: Toggle to determine if the Taylor map for an element include any element "misalignments". See Section §5.7 for more details.

`%csr_and_space_charge_on`
: Turn on or off the coherent synchrotron radiation and space charge calculations. (§17.4). The space charge calculation here is not to be confused with the high energy space charge calculation (§17.5)

`%d_orb`
: Sets the orbit displacement used in the routine that calculates the transfer matrix through an element via tracking. That is, when the `mat6_calc_method` (§5.2) is set to `tracking`. `%d_orb` needs to be large enough to avoid significant round-off errors but not so large that nonlinearities will affect the results. The default value is 10^{-5}. Also see `%mat6_track_symmetric`.

`%debug`
Used for communication between program units for debugging purposes.

`%default_ds_step`
Step size for tracking code §5 that uses a fixed step size. For example, `symp_lie_ptc` tracking.

`%default_integ_order`
Order of the integrator used by Étienne Forest's PTC code (§25.2). The order of the PTC integrator is like the order of a Newton-Cotes method. Higher order means the error term involves a higher order derivative of the field.

`%electric_dipole_moment`
The electric dipole moment value used in tracking a particle's spin (§19.1).

`%fatal_ds_adaptive_tracking`
This is used in `runge-kutta` and `time_runge_kutta` tracking (§5.4). If the step size falls below the value set for `%fatal_ds_adaptive_tracking`, a particle is considered lost. This prevents a program from "hanging" due to taking a large number of extremely small steps. The most common cause of small step size is an "unphysical" magnetic or electric field.

`%lr_wakes_on`
Toggle for turning on or off long-range higher order mode wake field effects.

`%mat6_track_symmetric`
Toggle to turn off whether the transfer matrix from tracking routine (twiss_from_tracking) tracks 12 particles at both plus and minus `%d_orb` values or only tracks 7 particles to save time but is less accurate..

`%max_aperture_limit`
Sets the maximum amplitude a particle can have during tracking. If this amplitude is exceeded, the particle is lost even if there is no element aperture set. Having a maximum aperture limit helps prevent numerical overflow in the tracking calculations.

`%max_num_runge_kutta_step`
The maximum number of steps to take through an element with `runge_kutta` or `time_runge_kutta` tracking. The default value is 10,000. If the number of steps reaches this value, the particle being tracked is marked as lost and a warning message is issued. Under "normal" circumstances, a particle will take far fewer steps to track through an element. If a particle is not through an element after 10,000 steps, it generally indicates that there is a problem with how the field is defined. That is, the field does not obey Maxwell's Equations. Especially: discontinuities in the field can cause problems.

`%min_ds_adaptive_tracking`
This is used in `runge-kutta` and `time_runge_kutta` tracking (§5.4). Minimum step size to use for adaptive tracking. If To be useful, `%min_ds_adaptive_tracking` must be set larger than the value of `%fatal_ds_adaptive_tracking`. In this case, particles are never lost due to taking too small a step.

ptc_max_fringe_order
Maximum order for computing fringe field effects in PTC.

`%rf_phase_below_transition_ref`
Used when *Bmad* autoscales (§4.18) an `rfcavity` and when *Bmad* calculates the reference time through a cavity (which affects calculation of phase space z via Eq. (14.28)). If True, the reference phase will be taken to be at `phi0` = 0.5 which is appropriate for a ring below transition. Default is False in which case autoscaling will be around the phase `phi0` = 0.

`%radiation_damping_on`
: Toggle to turn on or off effects due to radiation damping in particle tracking.

`%radiation_fluctuations_on`
: Toggle to turn on or off effects due to radiation fluctuations in particle tracking.

`%rel_tol_adaptive_tracking`
: Relative tolerance to use in adaptive tracking. This is used in `runge_kutta` and `time_runge_kutta` tracking (§5.4).

`%rel_tol_tracking`
: Relative tolerance to use in tracking. Specifically, Tolerance to use when finding the closed orbit.

`%significant_length`
: Sets the scale to decide if two length values are significantly different. For example, The superposition code will not create any super_slave elements that have a length less then this.

`%space_charge_mesh_size`
: The `%space_charge_mesh_size` sets the size of the grid used when an element's `space_charge_method` is set to `fft_3d` (§5.5). The value of this parameter is a 3-element array (n_x, n_y, n_z) giving the mesh size in the x, y, and z directions respectively. Default values are $(32, 32, 64)$.

`%sr_wakes_on`
: Toggle for turning on or off short-range higher order mode wake field effects.

`%spin_sokolov_ternov_flipping_on`
: This determines if the Sokolov-Ternov effect is included in a simulation. The Sokolov-Ternov effect[Barber99] is the self-polarization of charged particle beams due to asymmetric flipping of a particle's spin when the particle is bent in a magnetic field. Also, spin flipping will *not* be done if spin tracking is off or both radiation damping and excitation are off.

`%spin_tracking_on`
: Determines if spin tracking is performed or not.

`%taylor_order`
: Cutoff Taylor order of maps produced by `sym_lie_ptc`.

9.3 Bmad_Com

The parameters of the `bmad_com` instance of the `bmad_common_struct` structure (§9.2) can be set in the lattice file using the syntax

```
bmad_com[parm-name] = value
```

where `parm-name` is the name of a component of `bmad_common_struct`. For example:

```
bmad_com[rel_tol_tracking] = 1e-7
```

Be aware that setting a `bmad_com` parameter value in a lattice file will affect all computations of a program even if the program reads in additional lattice files. That is, setting of `bmad_com` components is "sticky" and persists even when other lattice files are read in. There are two exceptions: A program is always free to override settings of `bmad_com` parameters. Additionally, a second lattice file can also override the setting made in a prior lattice file.

9.4 CSR_Parameter_Struct

The Coherent Synchrotron Radiation (CSR) calculation is discussed in Section §17.4.

Besides the parameters discussed below, the `csr_and_space_charge_on` parameter of `bmad_com` (§9.3) must be set True to enable the CSR calculation. Additionally, tracking with `CSR` will only be done through elements where the parameter `csr_method` (§5.4) has been set to `1_dim`. This is done so that the computationally intensive `CSR` calculation can be restricted to places where the `CSR` effect is significant.

The CSR parameter structure has a `type name` of `csr_parameter_struct` and an `instance name` of `csr_param`. This structure has components

```
type csr_parameter_struct
  real(rp) ds_track_step = 0        ! Tracking step size
  real(rp) beam_chamber_height = 0  ! Used in shielding calculation.
  real(rp) sigma_cutoff = 0.1       ! Cutoff for the lsc calc. If a bin sigma
                                    !  is < cutoff * sigma_ave then ignore.
  integer n_bin = 0                 ! Number of bins used
  integer particle_bin_span = 2     ! Longitudinal particle length / dz_bin
  integer n_shield_images = 0       ! Chamber wall shielding. 0 = no shielding.
  integer sc_min_in_bin = 10        ! Min number of particle needed to compute sigmas.
  logical small_angle_approx = T    ! Use lcsr small angle approximation?
  logical print_taylor_warning = T  ! Print Taylor element warning?
  logical write_csr_wake = F        ! Write a CSR wake file?
end type
```

The values for the various quantities shown above are their default values.

ds_track_step
: `ds_track_step` is the nominal longitudinal distance traveled by the bunch between CSR kicks. The actual distance between kicks within a lattice element is adjusted so that there is an integer number of steps from steps from the element entrance to the element exit. This parameter must be set to something positive otherwise an error will result. Larger values will speed up the calculation at the expense of accuracy.

beam_chamber_height
: `beam_chamber_height` is the height of the beam chamber in meters. This parameter is used when shielding is taken into account. See also the description of the parameter `n_shield_images`.

sigma_cutoff
: `sigma_cutoff` is used in the longitudinal space charge (LSC) calculation and is used to prevent bins with only a few particles in them to give a large contribution to the kick when the computed transverse sigmas are abnormally low.

n_bin
: `n_bin` is the number of bins used. The bind width is dynamically adjusted at each kick point so that the bins will span the bunch length. This parameter must be set to something positive. Larger values will slow the calculation while smaller values will lead to inaccuracies and loss of resolution. `n_bin` should also not be set so large that the average number of particles in a bin is too small. "Typical" values are in the range 100 — 1000.

particle_bin_span
: `particle_bin_span` is the width of a particle's triangular density distribution (cf. §17.4) in multiples of the bin width. A larger span will give better smoothing of the computed particle density with an attendant loss in resolution.

n_shield_images
: `n_shield_images` is the number of shielding current layers used in the shielding calculation. A value of zero results in no shielding. See also the description of the parameter `beam_chamber_height`. The proper setting of this parameter depends upon how strong the shielding is. Larger values give better accuracy at the expense of computation speed. "Typical" values are in the range 0 — 5.

sc_min_in_bin
: the `sc_min_in_bin` parameter sets the minimum number of particle in a bin needed to compute the transverse beam sigmas for that bin. If the number of particles is less than this number, the beam sigmas are taken to be equal to the beam sigmas of a nearby bin where there are enough particle to compute the sigma. The beam sigmas are needed for the CS calculation but not need for the CSR calculation.

write_csr_wake
: If True (default is False), an output file called `csr_wake.dat` is created that contains a table of the CSR wake at each track step (the track step size is set by `ds_track_step`). If tracking is done through multiple lattice elements, the wake tables for the elements are appended to the file. This file is useful for visualization of the wake.

Note: `Taylor` map elements (§3.46) that have a finite length cannot be subdivided for the CSR calculation. *Bmad* will ignore any `taylor` elements present in the lattice but will print a warning that it is doing so. To suppress the warning, `print_taylor_warning` should be set to False.

9.5 Opti_DE_Param_Struct

The Differential Evolution (`DE`) optimizer is used in nonlinear optimization problems. This optimizer is based upon the work of Storn and Price[Storn96]. There are a number of parameters that can be varied to vary how the optimizer works. These parameters are are contained in a structure named `opti_de_param_struct`. the instance name is `opti_de_param`. This structure has components

```
                              Default
type opti_de_param_struct
  real(rp) CR             = 0.8    ! Crossover Probability.
  real(rp) F              = 0.8    !
  real(rp) l_best         = 0.0    ! Percentage of best solution used.
  logical  binomial_cross = False  ! IE: Default = Exponential.
  logical  use_2nd_diff   = False  ! use F * (x_4 - x_5) term
  logical  randomize_F    = False  !
  logical  minimize_merit = True   ! F => maximize the Merit func.
end type
```

The "perturbed vector" is v = x_1 + l_best * (x_best - x_1) + F * (x_2 - x_3) + F * (x_4 - x_5) The last term F * (x_4 - x_5) is only used if `use_2nd_diff` = T.

The crossover can be either "Exponential" or "Binary". Exponential crossover is what is described in the paper. With Exponential crossover the crossover parameters from a contiguous block and the average number of crossover parameters is approximately average crossovers ∼ min(D, CR / (1 - CR)) where D is the total number of parameters. With Binary crossover the probability of crossover of a parameter is uncorrelated with the probability of crossover of any other parameter and the average number of crossovers is average crossovers = D * CR

`randomize_F` = True means that the F that is used for a given generation is randomly chosen to be within the range [0, 2*F] with average F.

9.6 Dynamic Aperture Simulations: Aperture_Param_Struct

The dynamic_aperture_struct is used for dynamic aperture calculations. This structure has components:

```
type aperture_param_struct
  real(rp) :: min_angle = 0
  real(rp) :: max_angle = pi
  integer :: n_angle   = 9
  integer :: n_turn = 100              ! Number of turns a particle must survive
  real(rp) :: x_init = 1e-3            ! Initial estimate for horizontal aperture
  real(rp) :: y_init = 1e-3            ! Initial estimate for vertical aperture
  real(rp) :: accuracy = 1e-5          ! Resolution of bracketed aperture.
end type
```

Chapter 10

Beam Initialization

Some *Bmad* based programs track beams of particles instead of tracking individual particles one-by-one. This can be useful for several reasons. For example, tracking beams is useful when inter-bunch or intra-bunch effects are to be simulated. Also tracking beams can simplify the bookkeeping a program needs to do to calculate such quantities such as the bunch size.

A *Bmad* based program has two standard ways to specify the initial distribution of a beam. One is using a `beam_init_struct` `structure` (§9.1) which holds parameters (for example, the beam emittances) from which a distribution of particles can be constructed. The `beam_init_struct` structure is explained in Section §10.1. The other way is to specify the initial beam distribution via a file that has the individual particle positions. This is covered in Section §10.2.

10.1 Beam_Init_Struct Structure

The `beam_init_struct` `structure` (§9.1) holds parameters which are used to initialize a beam. The parameters of this structure are:

```
type beam_init_struct
  character(200) :: file_name = ''          ! Initialization file name.
  character distribution_type(3)            ! "ELLIPSE", "KV", "GRID", "" (default).
  type (ellipse_beam_init_struct) ellipse(3) ! For ellipse beam distribution
  type (kv_beam_init_struct) KV             ! For KV beam distribution
  type (grid_beam_init_struct) grid(3)      ! For grid beam distribution
  logical use_particle_start_for_center = F  ! Use particle_start instead of center?
  character random_engine           ! "pseudo" (default) or "quasi".
  character random_gauss_converter  ! "exact" (default) or "quick".
  real center(6) = 0                ! Bench center offset.
  real center_jitter(6) = 0.0       ! Bunch center rms jitter
  real emit_jitter(2)   = 0.0       ! %RMS a and b-mode emittance jitter
  real sig_z_jitter     = 0.0       ! bunch length RMS jitter
  real sig_pz_jitter     = 0.0      ! pz energy spread RMS jitter
  real random_sigma_cutoff = -1     ! -1 => no cutoff used.
  integer n_particle = 0            ! Number of simulated particles per bunch.
  logical renorm_center = T         ! Renormalize centroid?
  logical renorm_sigma = T          ! Renormalize sigma?
  real(rp) spin(3)                  ! Spin (x, y, z)
```

```
    real a_norm_emit                  ! a-mode normalized emittance (= γ ϵ)
    real b_norm_emit                  ! b-mode normalized emittance (= γ ϵ)
    real a_emit                       ! a-mode emittance (= γ ϵ)
    real b_emit                       ! b-mode emittance (= γ ϵ)
    real dpz_dz = 0                   ! Correlation of pz with longitudinal position.
    real dt_bunch                     ! Time between bunches.
    real sig_z                        ! Z sigma in m.
    real sig_pz                       ! pz sigma.
    real bunch_charge                 ! Charge in a bunch.
    integer n_bunch = 0               ! Number of bunches.
    character species                 ! Species. Default is reference particle.
    logical full_6D_coupling_calc = F  ! Use 6x6 1-turn matrix to match distribution?
    logical use_t_coords = F        ! If true, the distributions will be
                                    !   calculated using time coordinates
    logical use_z_as_t   = F        ! Only used if  use_t_coords = T
                                    !   If True,  particles will be distributed in t
                                    !   If False, particles will be distributed in s
  end type
```

Note: The `z` coordinate value given to particles of a bunch is with respect to the nominal center of the bunch. Therefore, if there are multiple bunches, and there is an RF cavity whose frequency is not commensurate with the spacing between bunches, absolute time tracking (§22.1) must be used.

%file_name

`%file_name` sets the name of the file to be read in containing the particle coordinates. Input from a file is triggered if not-blank. The format of the file is discussed in Section §10.2.

%a_emit, %b_emit, %a_norm_emit, %b_norm_emit

Normalized and unnormalized emittances. Either `a_norm_emit` or `a_emit` may be set but not both. similarly, either `b_norm_emit` or `b_emit` may be set but not both.

%bunch_charge

The `%bunch_charge` paramter sets the charge of a bunch. If reading from a file, the bunch charge will be set to the value of `%bunch_charge` except if `%bunch_charge` has a value of zero in which case the bunch charge as specified in the file is used.

%center(6)

The `%center` parameter is used to offset the center position of a bunch when the `%use_particle_start_for_center` is set to False. See the description of `%use_particle_start_for_center` below for more details.

%center_jitter, %emit_jitter, %sig_z_jitter, %sig_pz_jitter

These components can be used to provide a bunch-to-bunch random variation in the emittance and bunch center. Note: The old, deprecated, name for `%sig_pz_jitter` is `%sig_e_jitter`. This deprecated name is currently accepted.

%distribution_type(3)

The `%distribution_type(:)` array determines what algorithms are used to generate the particle distribution for a bunch. Note: If `%file_name` is not blank, the beam distribution will be read from the appropriate file and `%distribution_type` will be ignored.

`%distributeion_type(1)` sets the distribution type for the (x, p_x) 2D phase space, etc. Possibilities for `%distributeion_type(:)` are:

```
  "", or "RAN_GAUSS"  ! Random distribution (default).
  "ELLIPSE"           ! Ellipse distribution (§17.1.1)
```

```
"KV"                ! Kapchinsky-Vladimirsky distribution (§17.1.2)
"GRID"              ! Uniform distribution.
```

Since the Kapchinsky-Vladimirsky distribution is for a 4D phase space, if the Kapchinsky-Vladimirsky distribution is used, `"KV"` must appear exactly twice in the `%distributeion_type(:)` array.

Unlike all other distribution types, the `GRID` distribution is independent of the Twiss parameters at the point of generation. For the non-`GRID` distributions, the distributions are adjusted if there is local x-y coupling (§20.1). For lattices with a closed geometry, if `full_6D_coupling_calc` is set to `True`, the full 6-dimensional coupling matrix is used. If `False`, which is the default, The 4-dimensional $\mathbf{V}$ matrix of Eq. (20.5) is used.

Note: The total number particles generated is the product of the individual distributions. For example:

```
type (beam_init_struct) bi
bi%distribution_type = ELLIPSE", "ELLIPSE", "GRID"
bi%ellipse(1)%n_ellipse = 4
bi%ellipse(1)%part_per_ellipse = 8
bi%ellipse(2)%n_ellipse = 3
bi%ellipse(2)%part_per_ellipse = 100
bi%grid(3)%n_x = 20
bi%grid(3)%n_px = 30
```

The total number of particles per bunch will be $32 \times 300 \times 600$. The exception is that when `RAN_GAUSS` is mixed with other distributions, the random distribution is overlaid with the other distributions instead of multiplying. For example:

```
type (beam_init_struct) bi
bi%distribution_type = RAN_GAUSS", "ELLIPSE", "GRID"
bi%ellipse(2)%n_ellipse = 3
bi%ellipse(2)%part_per_ellipse = 100
bi%grid(3)%n_x = 20
bi%grid(3)%n_px = 30
```

Here the number of particle is 300×600. Notice that when `RAN_GAUSS` is mixed with other distributions, the value of `beam_init%n_particle` is ignored.

%full_6D_coupling_calc

If set `True`, coupling between the transverse and longitudinal modes is taken into account when calculating the beam distribution. The default `False` decouples the transverse and longitudinal calculations.

%dPz_dz

Correlation between p_z and z phase space coordinates.

%dt_bunch

Time between bunches

%ellipse(3)

The `%ellipse(:)` array sets the parameters for the `ellipse` distribution (§17.1.1). Each component of this array looks like

```
type ellipse_beam_init_struct
  integer part_per_ellipse  ! number of particles per ellipse.
  integer n_ellipse         ! number of ellipses.
  real(rp) sigma_cutoff     ! sigma cutoff of the representation.
end type
```

%grid(3)

The `%grid` component of the `beam_init_struct` sets the parameters for a uniformly spaced grid of particles. The components of `%grid` are:

```
type grid_beam_init_struct
  integer n_x        ! number of columns.
  integer n_px       ! number of rows.
  real(rp) x_min     ! Lower x limit.
  real(rp) x_max     ! Upper x limit.
  real(rp) px_min    ! Lower px limit.
  real(rp) px_max    ! Upper px limit.
end type
```

%KV

The %kv component of the beam_init_struct sets the parameters for the Kapchinsky-Vladimirsky distribution (§17.1.2). The components of %KV are:

```
type kv_beam_init_struct
  integer part_per_phi(2)    ! number of particles per angle variable.
  integer n_I2               ! number of I2
  real(rp) A                 ! A = I1/e
end type
```

%n_bunch

The number of bunches in the beam is set by n_bunch. If reading the distribution from a file, if %n_bunch is zero, the number of bunches created is the number of defined in the file and if %n_bunch is not zero, the number created is %n_bunch. It is an error if %n_bunch is greater than the number of bunches defined in the file. If not reading from a file, if %n_bunch is zero, one bunch is created.

%n_particle

Number of particles generated when the %distribution_type is "RAN_GAUSS". Ignored for other distribution types. When reading the distribution from a file, if %n_particle is zero, the number of particles in a bunch will be the number of particles defined in the file. If %n_particle is non-zero when reading from a file, the number of particles in a bunch will be %n_particle. It is an error if %n_particle is non-zero and the number of particles defined in the file is less than %n_particle.

%random_engine

This component sets the algorithm to use in generating a uniform distribution of random numbers in the interval [0, 1]. "pseudo" is a pseudo random number generator and "quasi" is a quasi random generator. "quasi random" is a misnomer in that the distribution generated is fairly uniform.

%random_gauss_converter, %random_sigma_cutoff

To generate Gaussian random numbers, a conversion algorithm from the flat distribution generated according to %random_engine is needed. %random_gauss_converter selects the algorithm. The "exact" conversion uses an exact conversion. The "quick" method is somewhat faster than the "exact" method but not as accurate. With either conversion method, if %random_sigma_cutoff is set to a positive number, this limits the maximum sigma generated.

%renorm_center, %renorm_sigma

If set to True, these components will ensure that the actual beam center and sigmas will correspond to the input values. Otherwise, there will be fluctuations due to the finite number of particles generated.

%sig_pz, %sig_z

Longitudinal sigmas. %sig_pz is the fractional energy spread dE/E. This, along with %dPz_dz determine the longitudinal profile. Note: The old, deprecated, name for %sig_pz is %sig_e. This deprecated name is currently accepted.

%species
Name of the species tracked. If not set then the default tracking particle type is used.

%spin
Particle spin in Cartesian (x, y, z) coordinates.

%use_particle_start_for_center
If `%use_particle_start_for_center` is set to `True` (default is `False`), the center of the bunch is determined by (`particle_start[x]`, `particle_start[px]`, ..., `particle_start[pz]`) (§8.2) setting in the lattice file rather than the setting of `%center` in the `beam_init_struct`. In this case, `%center` is a dependent parameter and will be set to the value of `particle_start`.

%use_t_coords, %use_z_as_t
If `use_t_coords` is true, then the distributions are taken as describing particles in t-coordinates (§14.4.3). Furthermore, if `use_z_as_t` is true, then the z coordinates from the distribution will be taken as describing the time coordinates. For example, particles may originate at a cathode at the same s, but different times. If false, then the z coordinate from the distribution describes particles at the same time but different s positions, and each particle gets `%location=inside$`. In this case, the bunch will need to be tracked with a tracking method that can handle inside particles, such as `time_runge_kutta`. All particles are finally converted to proper s-coordinate distributions for Bmad to use.

10.2 File Based Beam Initialization

A beam initialization file specifies the coordinates of all the particles in a beam. If a *Bmad* based program uses a `beam_init_struct` (§10.2) for inputting initialization parameters, the file name for file based beam initialization can be set using the `%file_name` component of the structure. Also the bunch charge, bunch number, and number of particles per bunch can be set in the `beam_init_struct`. Additionally, the bunch centroid can be offset by setting `beam_init%center` and `beam_init%center_jitter`.

There are two formats for the beam initialization file: ASCII and binary. Currently, the binary file format is undergoing revision and will not be discussed here. The ASCII file format is:

```
<ix_ele>          ! Lattice element index. This is ignored.
<n_bunch>         ! Number of bunches.
<n_particle>      ! Number of particles per bunch to use
[bunch loop: ib = 1 to n_bunch]
  BEGIN_BUNCH     ! Marker to mark the beginning of a bunch specification block.
  <species_name>  ! Species of particle
  <bunch_charge>  ! Charge of bunch. 0 => Use <macro_charge>.
  <z_center>      ! z position at center of bunch.
  <t_center>      ! t position at center of bunch.
  [particle loop: Stop when END_BUNCH marker found]
    <x> <px> <y> <py> <z> <pz> <macro_charge> <state> <spin_x> <spin_y> <spin_z>
  [end particle loop]
  END_BUNCH       ! Marker to mark the end of the bunch specification block
[end bunch loop]
```

Example:

```
0       ! ix_ele
1       ! n_bunch
25000   ! n_particle
BEGIN_BUNCH
```

```
  POSITRON
  3.2E-9   ! bunch_charge
  0.0      ! z_center
  0.0      ! t_center
 -6.5E-3  9.6E-3 -1.9E-2  8.8E-3  2.2E-2 -2.4E-2  1.2E-13  1 1.0 0.0 0.0
  8.5E-3  5.5E-3  4.0E-2 -1.9E-2 -4.9E-3  2.1E-2  1.2E-13  1 1.0 0.0 0.0
  1.1E-2 -1.9E-2 -2.5E-2  1.0E-2 -1.8E-2 -7.1E-3  1.2E-13  1 1.0 0.0 0.0
 -3.4E-2 -2.7E-3 -4.1E-3  1.3E-2  1.3E-2  1.0E-2  1.2E-13  1 1.0 0.0 0.0
  6.8E-3 -4.5E-3  2.5E-3  1.4E-2 -2.3E-3  7.3E-2  1.2E-13  1 1.0 0.0 0.0
  1.2E-2 -9.8E-3  1.7E-3  6.4E-3 -9.8E-3 -7.2E-2  1.2E-13  1 1.0 0.0 0.0
  1.1E-2 -3.5E-4  1.2E-2  1.8E-2  5.4E-3  1.4E-2  1.2E-13  1 1.0 0.0 0.0
     ... etc. ...
END_BUNCH
```

The first line of the file gives <ix_ele>, the index of the lattice element at which the distribution was created. This is ignored when the file is Read.

The second line gives <n_bunch>, the number of bunches. This can be overridden by a non-zero setting of beam_init%n_bunch.

The third line gives <n_particle> the number of particles in a bunch. The actual number rows specifying particle coordinates may be more then <n_particle>. In this case, particles will be discarded so that the beam has <n_particle> particles. The setting of <n_particle> can be overridden by a non-zero setting of beam_init%n_particle.

After this, there are <n_bunch> blocks of data, one for each bunch. Each one of these blocks starts with a BEGIN_BUNCH line to mark the beginning of the block and ends with a END_BUNCH marker line. In between, the first four lines give the <species_name> name, <bunch_charge>, <z_center>, and <t_center> values. The <species_name> name may be one of:

```
positron  ! default
electron
proton
antiproton
muon
antimuon
photon
```

The lines following the <t_center> line specify particle coordinates. One line for each particle. Only the first six numbers, which are the phase space coordinates, need to be specified for each particle. If the <macro_charge> column is not present, or is zero, it defaults to <bunch_charge>/<n_particle>.

The <state> parameter indicates whether a particle is alive or dead. Values are

```
1     ! Alive
2-7   ! Dead
```

The particle spin is specified by x, y and z components.

Each particle has an associated <macro_charge>. If <bunch_charge> is set to a non-zero value, the charge of all the particles will be scaled by a factor to make the total macro charge equal to <bunch_charge>. The macro charge is ignored in tracking. The charge of the particle used in tracking is the charge as calculated for the particle species. On the other hand, the macro charge is used to calculate such things as the total charge in a particular region or the field produced by a particle. That is, the macro charge acts as a weighting factor for a particle when the particle's field or the particle's effect on other particles is calculated.

When the particle coordinates are read in the centroid will be shifted by the setting of beam_init%center (unless beam_init%use_particle_start_for_center is set True) and beam_init%center_jitter.

Chapter 11

Lattice Examples

This chapter gives some examples of how lattice files can be constructed to describe various machine geometries.

11.1 Example: Injection Line

An injection line is illustrated in Fig. 11.1. In this example, The path of an injected particle after it leaves the last element `X` of the injection line (dashed blue line) partially goes through the field of the dipole `BND` in the storage ring. One way to simulate this is:

```
INJ_L: line = (..., X, P, BND2, BR)
RING_L: line = (..., BND, M, ...)
P: patch, x_offset = -0.5, x_pitch = 0.15, z_offset = 0.3
BND: sbend, l = 6.2, g = 1/52
BND2: BND, l = 4.7, tracking_method = runge_kutta,
        field_calc = fieldmap, grid_field = {...}
BR: fork, to_line = RING_L, to_element = M
M: marker
use, INJ_L
```

In order to properly track particles through the fringe field of the dipole `BND`, a partial section of `BND`, called `BND2`, is placed in the injection line `INJ_L`. The `tracking_method` for `BND2` is set to `runge_kutta` since the default `bmad_standard` tracking is not able to handle these fringe fields. Additionally, the

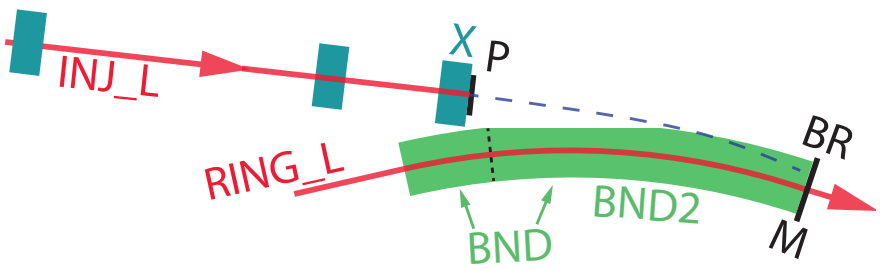


Figure 11.1: Injection line into a dipole magnet.

field_calc parameter of BND2 is set to field map so that the actual field profile of this particular magnet can be used in the tracking. The field is specified in the grid_field parameter (§4.15).

After traversing element X in the injection line, the particle goes through the patch P which offsets the reference trajectory so that following element, BND2, is properly positioned. The beginning of BND2 is marked by a black dashed line in the figure. At the end of BND2 the fork element BR connects INJ_L with the marker M in RING_L.

11.2 Example: Energy Recovery Linac

An Energy Recovery Linac (ERL) is illustrated in Fig. 11.2A. The ERL starts with an injection line that feeds a linac which accelerates the beam to some energy. The beam then transverses a return arc which reinjects the bunches into the linac. The length of the return arc is such that, on the second pass, the beam is decelerated giving its energy back to the RF cavities. Finally, the decelerated beam is steered through a dump line where, at the end, an absorber stops the beam.

A lattice file for modeling this ERL:

```
parameter[geometry] = open
parameter[absolute_time_tracking] = T

BEND_L1: sbend, angle = -25*pi/180, l = 0.2, ...
BEND_L2: BEND_L1

A_PATCH: patch, flexible = T
D_PATCH: patch, x_offset = 0.034, x_pitch = asin(0.32)
INJECT: line = (...)
LINAC: line[multipass] = (BEND_L1, ..., BEND_L2)
ARC: line = (..., BEND_A7)
DUMP: line = (...)

ERL: line = (INJECT, LINAC, ARC, A_PATCH, LINAC, D_PATCH, DUMP)
```

Fig. 11.2B shows the injector and arc merging into the beginning of the linac. The first element of the linac is a bend named BEND_L1. The bending angle for BEND_L1 has been set at the appropriate value for injection from the injector. To get the correct geometry for injection from the arc, a patch element, named A_PATCH, is placed in the ERL line between the arc and the linac. A_PATCH is a flexible patch which means that the exit edge of A_PATCH will automatically be aligned with the entrance edge of the next element which is BEND_L1.

Note that this use of a flexible patch works since the orientation of BEND_L1 has been determined before the orientation of A_PATCH is determined. The orientation of elements is determined in order starting from the first element in the line (the exception to this rule is if there is a floor_position element) and the orientation of BEND_L1 is thus determined right after the injector section on the first pass through the linac.

Fig. 11.2C shows the end of the linac splitting off into the dump and arc sections. The D_PATCH is used to orient the reference trajectory so that the dump is correctly positioned. Here it is not possible to make the D_PATCH flexible since the position of the dump is unknown when the orientation of the D_PATCH is calculated. However, the D_PATCH could be made flexible if a floor_position element is used in the dump line (*Bmad* will work both forward and backwards from a floor_position element so that a floor_position element may be placed anywhere in the dump line).

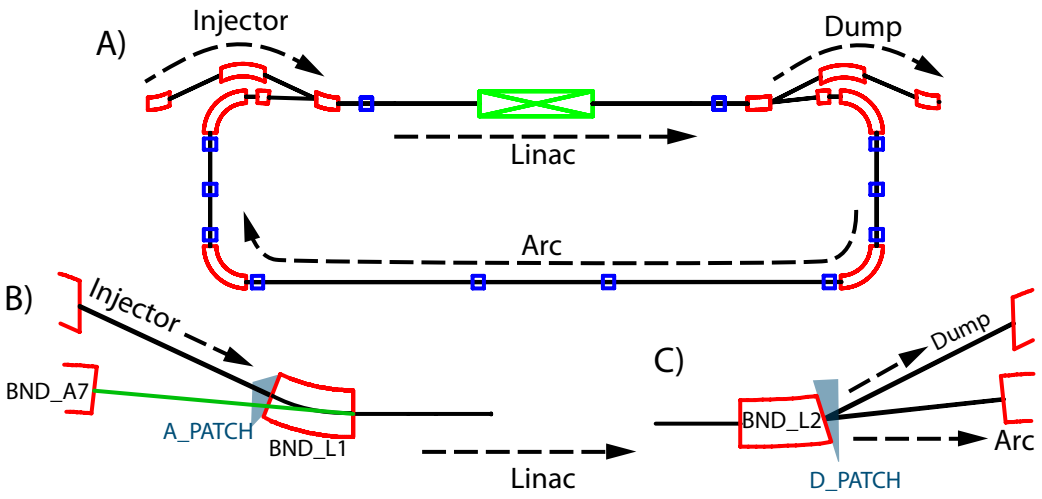


Figure 11.2: Example Energy Recovery Linac. A) The ERL consists of an injection line, accelerating linac, return arc, deceleration linac, and finally a beam dump. B) Close up of the section where the end of the injector and the end of the arc inject into the beginning of the linac. C) Close up of the end of the linac which injects into the dump and the beginning of the arc.

11.3 Example: Patch Between reversed and non-reversed elements

Between normal and reversed elements there must be a reflection `patch` element (§3.37). This is illustrated in Fig. 11.3. The basic lattice is

```
D: drift, l = 2
g_design = pi/12
B: sbend, l = 2, g = g_design, g_err = -2*g_design
P: patch, x_pitch = pi
A_line: line = (D, --B)     ! Illegal. Do not use!
B_line: line = (D, P, --B)  ! Correct
```

Line `A_line` represents the situation shown in Fig. 11.3A. With no patch between the drift `D` and the reversed bend `B`, a particle leaving `D` at `D`'s downstream end will find itself outside of both `D` and `B`. Clearly this is an unphysical situation. Sanity is restored in line `B_line` shown in Fig. 11.3B. In this instance, the patch `P` rotates the reference coordinates around the y-axis leaving the y-axis invariant the

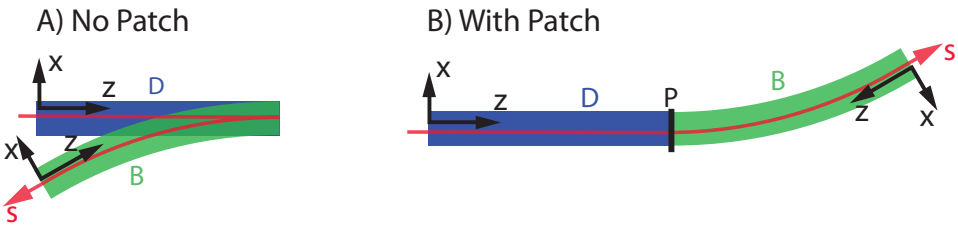


Figure 11.3: Drift element `D` is followed by a reversed drift element `B`. The view is from $+y$ onto the x-z plane. A) If no patch is present then the geometry does not make physical sense. B) With a patch in between, a sane geometry can be obtained.

bend of `B` is in the x-z plane. There are other patch parameter values that could be used to produce a reflection patch (§14.2.6). For example, Setting the patch's `y_pitch` to `pi` would produce a reflection patch.

Since bend `B` is reversed, A particle moving downstream within `B` is going the opposite direction from the normal direction. If `g_err` were zero in this instance, a downstream moving particle would feel a force that will rotate the particle in a clockwise manner opposite from the counterclockwise direction of the bend. To counter this, `g_err` is set so the total bending field `g_tot = g + g_err` is opposite the design field. That is, `g_err` is set so that `g_tot = -g`.

11.4 Example: Colliding Beam Storage Rings

The idealized layout of a pair of storage rings used for colliding counter rotating beams of electrons and gold is shown in Fig. 11.4. Rings `A` and `B` intersect at two interaction regions labeled `ir1` and `ir2` where the beams collide. The basic lattice description is:

```
ir: line[multipass] = (...)
pa_in; patch, ... ;  pa_out; patch, ...
pb_in; patch, ... ;  pb_out; patch, ...
m: marker
fid: fiducial, origin_ele = m
...
A: line = (arc_a, pa_in, ir, m, pa_out)
A[particle] = electron
B_rev: line = (arc_b, pb_in, ir, fid, pb_out)
B: line = (--B_rev)
B[particle] = Au+79
use, A, B
```

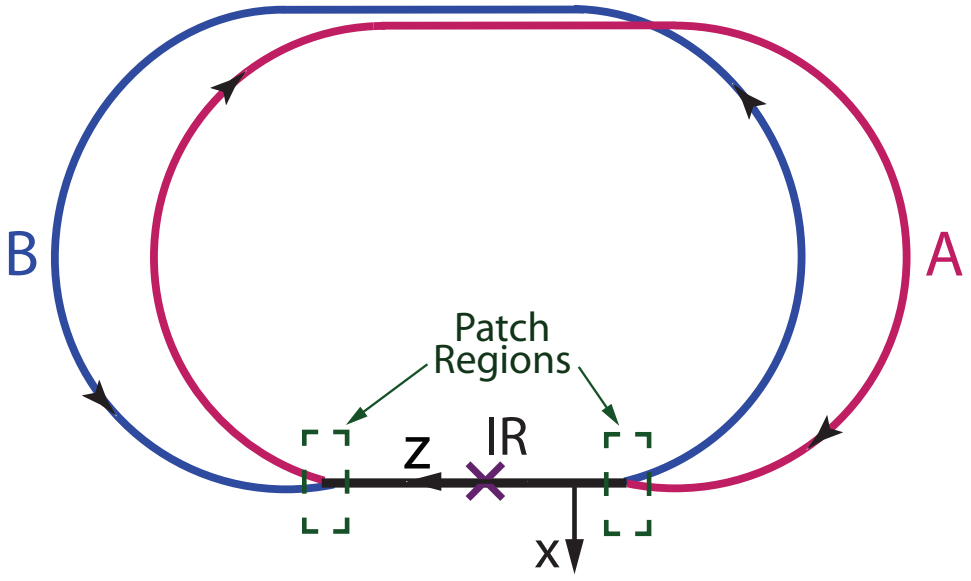


Figure 11.4: Dual ring colliding beam machine. The beam in the `A` ring rotates clockwise and in the `B` ring counterclockwise.

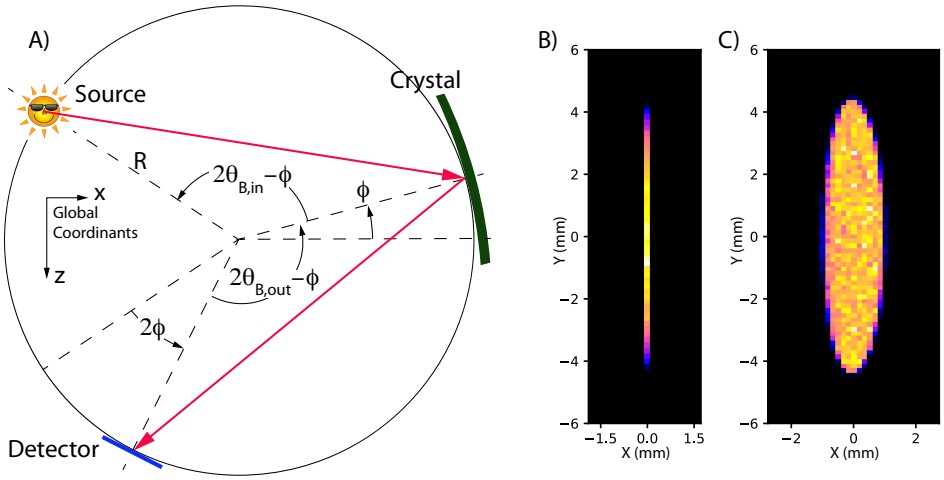


Figure 11.5: Rowland circle spectrometer: A) X-rays scattered from a sample (labeled source in the figure) illuminates a crystal with X-rays. Some of the X-rays are reflected from the crystal onto the detector. Note: For clarity's sake the center of the global coordinate system as shown is shifted from the true center at the Source element. B) The detector image when the radius of curvature of the bent crystal is "perfect". That is, twice the radius of the Rowland circle. C) The detector image when the radius of curvature of the crystal is shifted 1% from perfect.

Lines `ir` is the interaction region line which is declared `multipass` since they are shared by the two rings. Line `A` represents ring `A`. In ring `A` where the electron beam which, by definition, travels in the same direction as increasing s, rotates clockwise. Line `B_rev` is a "reversed" line of ring `B` and, like `a`, represents a beam rotating clockwise. Line `B`, which represents ring `B`, is the reverse of `B_rev` and here the gold beam rotates counterclockwise. In this construction, all elements of `B` are reversed. While this is not mandatory (only the interaction regions must be reversed in `B`), having all of `B` reversed simplifies the geometry since this means that the local coordinate systems of both lines `A` and `b` will be "aligned" with the x-axis pointing to the outside of the ring and the y-axis pointing up, out of the page. Having non-aligned coordinate systems is possible but potentially very confusing.

The two rings are physically aligned using a marker `m` in `A` and a `fiducial` element `fid` in `B` that aligns with `m`. Each ring has two rigid `patch` elements, `pa_in` and `pa_out` for the `A` ring, and `pb_in` and `bp_out` for the `B` ring, on either side of the interaction region. The dashed, green rectangles in the figure show the regions where the patches are.

The finished lattice will have two branches, The first branch (with index 0) will be derived from line `A` (and hence will be named "A") and the second branch (with index 1) will be derived from line `B` (and hence will be named "B"). The multipass lords representing the physical IR elements will be in the "lord section" of branch 0.

11.5 Example: Rowland Circle X-Ray Spectrometer

This example shows how *Bmad* can be used to simulate X-rays. In this case, the present example is taken from a case study where simulations were done in order to understand how imperfections in a Rowland circle spectrometer would affect measurements.

A Rowland circle spectrometer is illustrated in Fig. 11.5A. The `source` was a sample that is illuminated

with X-rays. Some of the X-rays scatter from the `sample` and are reflected from the `crystal` to the `detector`. To properly focus the X-rays onto the `detector`, the `source`, `crystal` and `detector` lie on a circle, called the Rowland circle. The `crystal` is bent and the radius of curvature of the crystal is $2R$ where R is the radius of the Rowland circle.

The angle from the `source` to the Rowland circle center to the crystal is $2\theta_{\text{B,in}} - \phi$ where $\theta_{\text{B,in}}$ is the entrance Bragg angle for photons whose energy matches the given reference energy and ϕ is an angle that will be varied when doing an energy scan of the scattered X-ray spectrum. Similarly, the angle from the `crystal` to the Rowland circle center to the `detector` is $2\theta_{\text{B,out}} - \phi$ where $\theta_{\text{B,out}}$ is the exit Bragg angle at the given reference energy.

The lattice for this simulation is:

```
beginning[e_tot] = 8.955e3    ! Reference photon energy
parameter[particle] = photon

phi = 0
err = 0
r_rowland = 0.5               ! Rowland circle radius

source: photon_init, sig_x = 5e-5, sig_y = 5e-5, spatial_distribution = uniform,
        E_center_relative_to_ref = T, sig_E = 2, energy_distribution = gaussian,
        velocity_distribution = spherical

drift1: drift

cryst: crystal, crystal_type = "Si(553)", b_param = -1, aperture = 0.050,
 spherical_curvature = (1+err) / (2 * r_rowland), aperture_type = elliptical

drift2: drift

det: detector, surface = {grid = {ix_bounds = (-97, 97),
                                  iy_bounds = (-243, 243), dr = (172e-6, 172e-6)}}

daves_line: line = (source, drift1, cryst, drift2, det)
use, daves_line

!------------------

expand_lattice ! Calculates the Bragg angles needed below.

theta_in  = cryst[bragg_angle_in]  ! 78.2759 * pi / 180
theta_out = cryst[bragg_angle_out] ! 78.2759 * pi / 180

cryst[graze_angle_in]  = theta_in - phi/2
cryst[graze_angle_out] = theta_out - phi/2
drift1[L] = 2 * r_rowland * sin(theta_in-phi/2)
drift2[L] = 2 * r_rowland * sin(theta_out-phi/2)

beginning[theta_position] = theta_in + phi/2
det[x_pitch] = pi/2 - theta_out + phi/2
```

The reference photon energy is 8.995 KeV and the Rowland circle radius is 0.5 m. The simulation uses a `photon_init` element (§3.38) for the `source` having a Gaussian energy spread with a sigma of 2 eV.

The initial velocity distribution of the photons, set by the `velocity_distribution` parameter, is taken to be uniform in all directions ("`spherical`" distribution). Since the element that is downstream from the source (which is the crystal element) has a defined aperture, *Bmad* is able to use this to not generate photons that will be lost at the crystal. This reduces the simulation time.

The crystal is `Silicon` 553 crystal which is symmetrically cut (`b_param` = -1) so that in this example the entrance Bragg angle is equal to the exit Bragg angle. The detector has a segmented surface with pixels spaced 172μm apart. Along the x-axis, which is the coordinate along the detector surface in the plane of Fig. 11.5A, the pixel index is in the range $[-97, 97]$. Along the y-axis, which is the out of plane coordinate, the pixel index is in the range $[-243, 243]$.

The `expand_lattice` command (§2.22) is used to command *Bmad* to construct the lattice which includes calculating the Bragg angles. After lattice expansion, the variables `theta_in` and `theta_out` are set to the Bragg angle entrance and exit Bragg angles respectively. The entrance and exit graze angles of the crystal, which are used to determine the reference trajectory (§14.2.3, can be set to `theta_in - phi` and `theta_out - phi` respectively. Note that if these graze angles had not been explicitly set, the graze angles would be automatically set to the Bragg angles which is not what is wanted when doing an energy scan with finite `phi`.

In the actual experimental setup that this example is modeled on, the source and Rowland circle were fixed in the global coordinate system (§14.2) while the crystal and detector move with changing `phi` (see Fig. 11.5A). To mimic this, the `beginning[theta_position]` (§8.4) is set to give the desired orientation of the beginning reference trajectory within the global coordinate system. This does not affect photon tracking since changing the initial orientation of the reference trajectory just shifts all the lattice elements as one rigid body. Additionally, the detector orientation is fixed so that the detector surface normal always points towards the Rowland circle center. To get the correct orientation for the detector, the detector's `x_pitch` attribute, which rotates the detector (§4.6), is set appropriately.

The effect of varying the `crystal` curvature is shown in Fig. 11.5B and Fig. 11.5C. A *Bmad* based program called `Lux` was used for the simulation. The `Lux` program generates a set of photons and records the statistics at the detector. In Fig. 11.5B the `crystal` is correctly bent with the parameter `err` in the lattice set to zero. This produces a well focused spot on the `detector`. In Fig. 11.5C the `crystal` curvature is shifted by 1% by setting `err` equal to 0.01. This error degrades the focusing and leads to a spot that is enlarged along the x-axis.

11.6 Example: Backward Tracking Through a Lattice

By creating a reversed lattice, one can essentially track particles backwards. For example, assume that you have a lattice file called `original_lattice.bmad` which defines a line called `original_line`. To create a reversed lattice, create a new file with the following:

```
call, file = original_lattice.bmad
reversed_line: line = (--original_line)
parameter[default_tracking_species] = antiparticle(parameter[particle])
use, reversed_line
```

The "`--`" reverses the line and reverses the elements (§6.3). Tracking through `reversed_line` is equivalent to tracking backwards through `original_line`.

The default for the type of particle tracked is set by `parameter[default_tracking_species]` (§8.1). [A *Bmad* based program can always override this default but it will be assumed here that this is not the case.] In this case the default species to use for tracking is set to the antiparticle of the reference particle species. If the `original_line` lattice had just static magnetic fields and no electric fields, by tracking with the anti-particle in the reversed lattice, the anti-particle will follow the same path (but

backward) as the particle in the original lattice. For this to work, the anti-particle must be started with the appropriate phase space coordinates. If (x, p_x, y, p_y, z, p_z) is the phase space coordinates of the particle at the end of the original lattice, the anti-particle must be initialized with phase space coordinates of $(x, -p_x, y, -p_y, \text{immaterial}, p_z)$.

Chapter 12

MAD/XSIF/SAD/PTC Lattice Conversion

12.1 MAD Conversion

12.1.1 Convert MAD to Bmad Via UAP

Conversion of lattice files from *MAD* to *Bmad* format can be done using the `Universal Accelerator Parser` (§12.4). Due to differences in language definitions, the conversions must be done with some care. The following differences should be noted:

- *Bmad*, unlike *MAD*, does not have any "action" commands. An action command is a command that makes a calculation. Examples include *MAD*'s `SURVEY` and `TWISS` commands.
- In *Bmad* all variables must be defined. In *MAD* undefined variables will default to 0.
- In *Bmad* all variables must be defined before being used (§2.12) while *MAD* does not have this constraint.
- *Bmad*, unlike *MAD*, does not allow variable values to be redefined.

12.1.2 Convert Bmad to MAD

Besides using the `Universal Accelerator Parser` for conversion from *Bmad* to *MAD*, there is a *Bmad* conversion routine called write_lattice_in_foreign_format. The advantage of this routine is that since *MAD* does not have a `wiggler` or a `sol_quad` element, this conversion routine can make an "equivalent" substitution. For a `sol_quad`, the equivalent substitution will be a drift-matrix-drift series of elements. For a `wiggler`, a series of bend and drift elements will be used (the program can also use a drift-matrix-drift model here but that is not as accurate). The bends and drifts for the `wiggler` model are constructed so that the global geometry of the lattice does not change. Additionally the bends and drifts are constructed to most nearly match the wiggler's

```
Transfer matrix
$I_2$ and $I_3$ synchrotron radiation integrals (§18.2)
```

Note that the resulting model will not have the vertical cubic nonlinearity that the actual wiggler has.

The `bmad_to_mad_or_xsif` routine is embeded in the program `util_programs/bmad_to_mad_or_xsif`.

12.2 XSIF Conversion

XSIF[Tenen01], developed at SLAC, stands for "Extended Standard Input Format." XSIF is essentially a subset of the *MAD* [Grote96] input format.

Bmad has software to directly parse XSIF files so XSIF files may be used in place of *Bmad* lattice files. With some restrictions, an XSIF lattice file may be called from within a *Bmad* lattice file. See Section §2.17 for details.

Since XSIF does not have a `parameter[geometry]` statement (§8.1), the type of the lattice (whether circular or linear) is determined by the presence or absence of any `lcavity` elements in the XSIF file. This is independent of whether `lcavity` elements are actually used in the lattice.

Note: One point that is not covered in the XSIF documentation is that for a `MATRIX` element, unlike *MAD*, the R*ii* terms (the diagonal terms of the linear matrix) are not unity by default. Thus
```
m: matrix
```
in an XSIF file will give a matrix with all elements being zero.

To convert between XSIF and *Bmad* the `Universal Accelerator Parser` can be used (§12.4). Additionally, the `bmad_to_mad_or_xsif` routine in *Bmad* can convert from *Bmad* to XSIF (cf. §12.1).

12.3 SAD Conversion

Conversion from `SAD`[SAD] to *Bmad* is accomplished using the Python script
```
util_programs/sad_to_bmad/sad_to_bmad.py
```
Currently, a converter from *Bmad* to SAD is planned but has not been implemented.

Currently, the following restrictions on SAD lattices apply:

- SAD must elements cannot have an associated RF field
- Misalignments in a `sol` element with `geo` = 1 cannot be handled.

12.4 Translation Using the Universal Accelerator Parser

The `Accelerator Markup Language` (`AML`) / `Universal Accelerator Parser` (`UAP`) project[AML] is a collaborative effort with the aim of 1) creating a lattice format (the `AML` part) that can be used to fully describe accelerators and storage rings, and 2) producing software (the `UAP` part) that can parse `AML` lattice files. A side benefit of this project is that the `UAP` code has been extended to be able to translate between `AML`, *Bmad*, *MAD*-8, *MAD*-X, and `XSIF`.

The program `translate_driver` which comes with the `UAP` code can be used for conversions. To get help with how to run this program use the command
```
path-to-uap-dir/bin/translate_driver -help
```
Example:
```
path-to-uap-dir/bin/translate_driver -constants_first -bmad xxx.madx
```
This will convert a *MAD*-X file `xxx.madx` to *Bmad* format.

Chapter 13

List of Element Attributes

Alphabetical list of element attributes for each type of element.

Note for programmers: The program that generates a file of attributes indexed by the internal reference number is:

`util_programs/element_attributes.f90`

13.1 AB_multipole Element Attributes

a0 - a20, b0 - b20	field_master	reference	x_limit [m]
alias	is_on	spin_tracking_method	x_offset [m]
aperture [m]	l [m]	superimpose	x_offset_tot [m]
aperture_at	mat6_calc_method	tilt [rad]	y1_limit [m]
aperture_type	multipoles_on	tilt_tot [rad]	y2_limit [m]
create_jumbo_slave	offset [m]	tracking_method	y_limit [m]
delta_ref_time [sec]	offset_moves_aperture	type	y_offset [m]
descrip	p0c [eV]	wall	y_offset_tot [m]
e_tot [eV]	ptc_integration_type	x1_limit [m]	z_offset [m]
ele_origin	ref_origin	x2_limit [m]	z_offset_tot [m]

13.2 AC_Kicker Element Attributes

a0 - a20, b0 - b20	fringe_at	p0c [eV]	vkick
alias	fringe_type	ptc_integration_type	wall
amp_vs_time	grid_field	r0_elec [m]	x1_limit [m]
aperture [m]	hkick	r0_mag [m]	x2_limit [m]
aperture_at	integrator_order	ref_origin	x_limit [m]
aperture_type	interpolation	ref_time_offset	x_offset [m]
bl_hkick [T*m]	is_on	reference	x_offset_tot [m]
bl_vkick [T*m]	l [m]	scale_multipoles	x_pitch
cartesian_map	l_hard_edge [m]	space_charge_method	x_pitch_tot
create_jumbo_slave	lord_pad1 [m]	spin_fringe_on	y1_limit [m]
csr_method	lord_pad2 [m]	spin_tracking_method	y2_limit [m]
cylindrical_map	lr_freq_spread [Hz]	sr_wake_file	y_limit [m]
delta_ref_time [sec]	lr_self_wake_on	superimpose	y_offset [m]
descrip	lr_wake_file	symplectify	y_offset_tot [m]
ds_step [m]	lr_wake_spline	t_offset [sec]	y_pitch
e_tot [eV]	mat6_calc_method	taylor_field	y_pitch_tot
ele_origin	multipoles_on	taylor_map_includes_offsets	z_offset [m]
field_calc	n_ref_pass	tilt [rad]	z_offset_tot [m]
field_master	num_steps	tilt_tot [rad]	
field_overlaps	offset [m]	tracking_method	
frequencies	offset_moves_aperture	type	

13.3 BeamBeam Element Attributes

alias	delta_ref_time [sec]	reference	x_offset_tot [m]
alpha_a	descrip	sig_x [m]	x_pitch
alpha_b	e_tot [eV]	sig_y [m]	x_pitch_tot
aperture [m]	ele_origin	sig_z [m]	y1_limit [m]
aperture_at	field_calc	spin_tracking_method	y2_limit [m]
aperture_type	is_on	superimpose	y_limit [m]
bbi_constant	l [m]	tilt [rad]	y_offset [m]
beta_a	mat6_calc_method	tilt_tot [rad]	y_offset_tot [m]
beta_b	n_slice	tracking_method	y_pitch
charge	n_slice	type	y_pitch_tot
cmat_11	offset [m]	wall	z_offset [m]
cmat_12	offset_moves_aperture	x1_limit [m]	z_offset_tot [m]
cmat_21	p0c [eV]	x2_limit [m]	
cmat_22	ptc_integration_type	x_limit [m]	
create_jumbo_slave	ref_origin	x_offset [m]	

13.4 Bends: Rbend and Sbend Element Attributes Element Attributes

a0 - a20, b0 - b20	field_overlaps	lr_wake_file	sr_wake_file
alias	fint	lr_wake_spline	superimpose
angle [rad]	fintx	mat6_calc_method	symplectify
aperture [m]	fringe_at	multipoles_on	taylor_field
aperture_at	fringe_type	n_ref_pass	taylor_map_includes_offsets
aperture_type	g [1/m]	num_steps	tracking_method
b1_gradient [T/m]	g_err [1/m]	offset [m]	type
b2_gradient [T/m^2]	grid_field	offset_moves_aperture	vkick
b_field [T]	h1 [1/m]	p0c [eV]	wall
b_field_err [T]	h2 [1/m]	ptc_canonical_coords	x1_limit [m]
bl_hkick [T*m]	hgap [m]	ptc_field_geometry	x2_limit [m]
bl_vkick [T*m]	hgapx [m]	ptc_fringe_geometry	x_limit [m]
cartesian_map	higher_order_fringe_type	ptc_integration_type	x_offset [m]
create_jumbo_slave	hkick	r0_elec [m]	x_offset_tot [m]
csr_method	integrator_order	r0_mag [m]	x_pitch
cylindrical_map	is_on	ref_origin	x_pitch_tot
delta_ref_time [sec]	k1 [1/m^2]	ref_tilt [rad]	y1_limit [m]
descrip	k2 [1/m^3]	ref_tilt_tot [rad]	y2_limit [m]
ds_step [m]	l [m]	reference	y_limit [m]
e1 [rad]	l_chord [m]	rho [m]	y_offset [m]
e2 [rad]	l_hard_edge [m]	roll [rad]	y_offset_tot [m]
e_tot [eV]	l_sagitta [m]	roll_tot [rad]	y_pitch
ele_origin	lord_pad1 [m]	scale_multipoles	y_pitch_tot
exact_multipoles	lord_pad2 [m]	space_charge_method	z_offset [m]
field_calc	lr_freq_spread [Hz]	spin_fringe_on	z_offset_tot [m]
field_master	lr_self_wake_on	spin_tracking_method	

13.5 Capillary Element Attributes

alias	mat6_calc_method	tilt_tot [rad]	y1_limit [m]
aperture [m]	n_slice_spline	tracking_method	y2_limit [m]
aperture_at	offset [m]	type	y_limit [m]
aperture_type	offset_moves_aperture	wall	y_offset [m]
create_jumbo_slave	p0c [eV]	x1_limit [m]	y_offset_tot [m]
critical_angle_factor [rad*eV]	ptc_integration_type	x2_limit [m]	y_pitch
delta_ref_time [sec]	ref_origin	x_limit [m]	y_pitch_tot
descrip	reference	x_offset [m]	z_offset [m]
e_tot [eV]	spin_tracking_method	x_offset_tot [m]	z_offset_tot [m]
ele_origin	superimpose	x_pitch	
l [m]	tilt [rad]	x_pitch_tot	

13.6 Collimators: Ecollimator and Rcollimator Element Attributes Element Attributes

alias	hkick	ptc_integration_type	x2_limit [m]
aperture [m]	integrator_order	ref_origin	x_limit [m]
aperture_at	is_on	reference	x_offset [m]
aperture_type	l [m]	space_charge_method	x_offset_tot [m]
bl_hkick [T*m]	l_hard_edge [m]	spin_fringe_on	x_pitch
bl_vkick [T*m]	lord_pad1 [m]	spin_tracking_method	x_pitch_tot
create_jumbo_slave	lord_pad2 [m]	sr_wake_file	y1_limit [m]
csr_method	lr_freq_spread [Hz]	superimpose	y2_limit [m]
delta_ref_time [sec]	lr_self_wake_on	symplectify	y_limit [m]
descrip	lr_wake_file	taylor_map_includes_offsets	y_offset [m]
ds_step [m]	lr_wake_spline	tilt [rad]	y_offset_tot [m]
e_tot [eV]	mat6_calc_method	tilt_tot [rad]	y_pitch
ele_origin	n_ref_pass	tracking_method	y_pitch_tot
field_calc	num_steps	type	z_offset [m]
field_overlaps	offset [m]	vkick	z_offset_tot [m]
fringe_at	offset_moves_aperture	wall	
fringe_type	p0c [eV]	x1_limit [m]	

13.7 Converter Element Attributes

alias	mat6_calc_method	tracking_method	y2_limit [m]
aperture [m]	offset [m]	type	y_limit [m]
aperture_at	offset_moves_aperture	wall	y_offset [m]
aperture_type	p0c [eV]	x1_limit [m]	y_offset_tot [m]
create_jumbo_slave	ptc_integration_type	x2_limit [m]	y_pitch
delta_ref_time [sec]	ref_origin	x_limit [m]	y_pitch_tot
descrip	reference	x_offset [m]	z_offset [m]
e_tot [eV]	spin_tracking_method	x_offset_tot [m]	z_offset_tot [m]
ele_origin	superimpose	x_pitch	
is_on	tilt [rad]	x_pitch_tot	
l [m]	tilt_tot [rad]	y1_limit [m]	

13.8 Crystal Element Attributes

alias	diffraction_limited	ref_orbit_follows	x1_limit [m]
alpha_angle	e_tot [eV]	ref_origin	x2_limit [m]
aperture [m]	ele_origin	ref_tilt [rad]	x_limit [m]
aperture_at	elliptical_curvature_x [1/m]	ref_tilt_tot [rad]	x_offset [m]
aperture_type	elliptical_curvature_y [1/m]	ref_wavelength [m]	x_offset_tot [m]
b_param	elliptical_curvature_z [1/m]	reference	x_pitch
bragg_angle [rad]	graze_angle_in [rad]	spherical_curvature [1/m]	x_pitch_tot
bragg_angle_in [rad]	graze_angle_out [rad]	spin_tracking_method	y1_limit [m]
bragg_angle_out [rad]	l [m]	superimpose	y2_limit [m]
create_jumbo_slave	mat6_calc_method	surface	y_limit [m]
crystal_type	offset [m]	thickness [m]	y_offset [m]
curvature_x0_y2 [1/m]	offset_moves_aperture	tilt [rad]	y_offset_tot [m]
d_spacing [m]	p0c [eV]	tilt_corr [rad]	y_pitch
darwin_width_pi [rad]	pendellosung_period_pi [m]	tilt_tot [rad]	y_pitch_tot
darwin_width_sigma [rad]	pendellosung_period_sigma [m]	tracking_method	z_offset [m]
dbragg_angle_de [rad/eV]	psi_angle [rad]	type	z_offset_tot [m]
delta_ref_time [sec]	ptc_integration_type	v_unitcell [m^3]	
descrip	ref_cap_gamma	wall	

13.9 Custom Element Attributes

alias	l_hard_edge [m]	sr_wake_file	val9
aperture [m]	lord_pad1 [m]	superimpose	wall
aperture_at	lord_pad2 [m]	symplectify	x1_limit [m]
aperture_type	lr_freq_spread [Hz]	taylor_map_includes_offsets	x2_limit [m]
create_jumbo_slave	lr_self_wake_on	tilt [rad]	x_limit [m]
csr_method	lr_wake_file	tilt_tot [rad]	x_offset [m]
delta_e_ref [eV]	lr_wake_spline	tracking_method	x_offset_tot [m]
delta_ref_time [sec]	mat6_calc_method	type	x_pitch
descrip	n_ref_pass	val1	x_pitch_tot
ds_step [m]	num_steps	val10	y1_limit [m]
e_tot [eV]	offset [m]	val11	y2_limit [m]
e_tot_start [eV]	offset_moves_aperture	val12	y_limit [m]
ele_origin	p0c [eV]	val2	y_offset [m]
field_calc	p0c_start [eV]	val3	y_offset_tot [m]
field_master	ptc_integration_type	val4	y_pitch
field_overlaps	ref_origin	val5	y_pitch_tot
integrator_order	reference	val6	z_offset [m]
is_on	space_charge_method	val7	z_offset_tot [m]
l [m]	spin_tracking_method	val8	

13.10 Detector Element Attributes

alias	elliptical_curvature_z [1/m]	superimpose	x_offset_tot [m]
aperture [m]	is_on	surface	x_pitch
aperture_at	l [m]	tilt [rad]	x_pitch_tot
aperture_type	mat6_calc_method	tilt_calib [rad]	y1_limit [m]
create_jumbo_slave	n_sample	tilt_tot [rad]	y2_limit [m]
crunch [rad]	noise	tracking_method	y_gain_calib [m]
crunch_calib [rad]	offset [m]	type	y_gain_err [m]
curvature_x0_y2 [1/m]	offset_moves_aperture	wall	y_limit [m]
de_eta_meas	osc_amplitude [m]	x1_limit [m]	y_offset [m]
delta_ref_time [sec]	p0c [eV]	x2_limit [m]	y_offset_calib [m]
descrip	ptc_integration_type	x_gain_calib [m]	y_offset_tot [m]
e_tot [eV]	ref_origin	x_gain_err [m]	y_pitch
ele_origin	reference	x_limit [m]	y_pitch_tot
elliptical_curvature_x [1/m]	spherical_curvature [1/m]	x_offset [m]	z_offset [m]
elliptical_curvature_y [1/m]	spin_tracking_method	x_offset_calib [m]	z_offset_tot [m]

13.11 Diffraction_Plate Element Attributes

alias	field_scale_factor	superimpose	x_pitch_tot
aperture [m]	is_on	surface	y1_limit [m]
aperture_at	mat6_calc_method	tilt [rad]	y2_limit [m]
aperture_type	mode	tilt_tot [rad]	y_limit [m]
create_jumbo_slave	offset [m]	tracking_method	y_offset [m]
curvature_x0_y2 [1/m]	offset_moves_aperture	type	y_offset_tot [m]
delta_ref_time [sec]	p0c [eV]	wall	y_pitch
descrip	ptc_integration_type	x1_limit [m]	y_pitch_tot
e_tot [eV]	ref_origin	x2_limit [m]	z_offset [m]
ele_origin	ref_wavelength [m]	x_limit [m]	z_offset_tot [m]
elliptical_curvature_x [1/m]	reference	x_offset [m]	
elliptical_curvature_y [1/m]	spherical_curvature [1/m]	x_offset_tot [m]	
elliptical_curvature_z [1/m]	spin_tracking_method	x_pitch	

13.12 Drift Element Attributes

alias	l [m]	spin_tracking_method	x_offset_tot [m]
aperture [m]	lord_pad1 [m]	superimpose	x_pitch
aperture_at	lord_pad2 [m]	symplectify	x_pitch_tot
aperture_type	mat6_calc_method	taylor_map_includes_offsets	y1_limit [m]
create_jumbo_slave	n_ref_pass	tilt [rad]	y2_limit [m]
csr_method	num_steps	tilt_tot [rad]	y_limit [m]
delta_ref_time [sec]	offset [m]	tracking_method	y_offset [m]
descrip	offset_moves_aperture	type	y_offset_tot [m]
ds_step [m]	p0c [eV]	wall	y_pitch
e_tot [eV]	ptc_integration_type	x1_limit [m]	y_pitch_tot
ele_origin	ref_origin	x2_limit [m]	z_offset [m]
field_calc	reference	x_limit [m]	z_offset_tot [m]
integrator_order	space_charge_method	x_offset [m]	

13.13 ELseparator Element Attributes

a0 - a20, b0 - b20	gap	ptc_canonical_coords	voltage [Volt]
alias	grid_field	ptc_integration_type	wall
aperture [m]	hkick	r0_elec [m]	x1_limit [m]
aperture_at	integrator_order	r0_mag [m]	x2_limit [m]
aperture_type	is_on	ref_origin	x_limit [m]
cartesian_map	l [m]	reference	x_offset [m]
create_jumbo_slave	l_hard_edge [m]	scale_multipoles	x_offset_tot [m]
csr_method	lord_pad1 [m]	space_charge_method	x_pitch
cylindrical_map	lord_pad2 [m]	spin_fringe_on	x_pitch_tot
delta_ref_time [sec]	lr_freq_spread [Hz]	spin_tracking_method	y1_limit [m]
descrip	lr_self_wake_on	sr_wake_file	y2_limit [m]
ds_step [m]	lr_wake_file	superimpose	y_limit [m]
e_field [V/m]	lr_wake_spline	symplectify	y_offset [m]
e_tot [eV]	mat6_calc_method	taylor_field	y_offset_tot [m]
ele_origin	multipoles_on	taylor_map_includes_offsets	y_pitch
field_calc	n_ref_pass	tilt [rad]	y_pitch_tot
field_master	num_steps	tilt_tot [rad]	z_offset [m]
field_overlaps	offset [m]	tracking_method	z_offset_tot [m]
fringe_at	offset_moves_aperture	type	
fringe_type	p0c [eV]	vkick	

13.14 EM_Field Element Attributes

alias	fringe_at	phi0 [rad/2pi]	wall
aperture [m]	fringe_type	phi0_autoscale [rad/2pi]	x1_limit [m]
aperture_at	grid_field	phi0_err [rad/2pi]	x2_limit [m]
aperture_type	integrator_order	ptc_canonical_coords	x_limit [m]
autoscale_amplitude	is_on	ptc_integration_type	x_offset [m]
autoscale_phase	l [m]	ref_origin	x_offset_tot [m]
cartesian_map	l_hard_edge [m]	reference	x_pitch
constant_ref_energy	lord_pad1 [m]	rf_frequency [Hz]	x_pitch_tot
create_jumbo_slave	lord_pad2 [m]	space_charge_method	y1_limit [m]
csr_method	lr_freq_spread [Hz]	spin_fringe_on	y2_limit [m]
cylindrical_map	lr_self_wake_on	spin_tracking_method	y_limit [m]
delta_ref_time [sec]	lr_wake_file	sr_wake_file	y_offset [m]
descrip	lr_wake_spline	superimpose	y_offset_tot [m]
ds_step [m]	mat6_calc_method	symplectify	y_pitch
e_tot [eV]	n_ref_pass	taylor_field	y_pitch_tot
e_tot_start [eV]	num_steps	taylor_map_includes_offsets	z_offset [m]
ele_origin	offset [m]	tilt [rad]	z_offset_tot [m]
field_autoscale	offset_moves_aperture	tilt_tot [rad]	
field_calc	p0c [eV]	tracking_method	
field_overlaps	p0c_start [eV]	type	

13.15 E_Gun Element Attributes

alias	gradient [eV/m]	phi0_autoscale [rad/2pi]	wall
aperture [m]	gradient_err [eV/m]	phi0_err [rad/2pi]	x1_limit [m]
aperture_at	grid_field	ptc_integration_type	x2_limit [m]
aperture_type	integrator_order	ref_origin	x_limit [m]
autoscale_amplitude	is_on	reference	x_offset [m]
autoscale_phase	l [m]	rf_frequency [Hz]	x_offset_tot [m]
cartesian_map	l_hard_edge [m]	space_charge_method	x_pitch
create_jumbo_slave	lord_pad1 [m]	spin_fringe_on	x_pitch_tot
csr_method	lord_pad2 [m]	spin_tracking_method	y1_limit [m]
cylindrical_map	lr_freq_spread [Hz]	sr_wake_file	y2_limit [m]
delta_ref_time [sec]	lr_self_wake_on	superimpose	y_limit [m]
descrip	lr_wake_file	symplectify	y_offset [m]
ds_step [m]	lr_wake_spline	taylor_field	y_offset_tot [m]
e_tot [eV]	mat6_calc_method	taylor_map_includes_offsets	y_pitch
ele_origin	n_ref_pass	tilt [rad]	y_pitch_tot
field_autoscale	num_steps	tilt_tot [rad]	z_offset [m]
field_calc	offset [m]	tracking_method	z_offset_tot [m]
field_overlaps	offset_moves_aperture	type	
fringe_at	p0c [eV]	voltage [Volt]	
fringe_type	phi0 [rad/2pi]	voltage_err [Volt]	

13.16 Fiducial Element Attributes

alias	dx_origin [m]	mat6_calc_method	ref_origin
delta_ref_time [sec]	dy_origin [m]	offset [m]	reference
descrip	dz_origin [m]	origin_ele	spin_tracking_method
dphi_origin [rad]	e_tot [eV]	origin_ele_ref_pt	superimpose
dpsi_origin [rad]	ele_origin	p0c [eV]	tracking_method
dtheta_origin [rad]	l [m]	ptc_integration_type	type

13.17 Floor_Shift Element Attributes

alias	l [m]	spin_tracking_method	x_offset [m]
aperture [m]	mat6_calc_method	superimpose	x_pitch
aperture_at	offset [m]	tilt [rad]	y1_limit [m]
aperture_type	offset_moves_aperture	tracking_method	y2_limit [m]
create_jumbo_slave	origin_ele	type	y_limit [m]
delta_ref_time [sec]	origin_ele_ref_pt	upstream_ele_dir	y_offset [m]
descrip	p0c [eV]	wall	y_pitch
downstream_ele_dir	ptc_integration_type	x1_limit [m]	z_offset [m]
e_tot [eV]	ref_origin	x2_limit [m]	
ele_origin	reference	x_limit [m]	

13.18 Fork and Photon_Fork Element Attributes Element Attributes

alias	ele_origin	p0c [eV]	type
aperture [m]	is_on	ptc_integration_type	x1_limit [m]
aperture_at	ix_to_branch	ref_origin	x2_limit [m]
aperture_type	ix_to_element	reference	x_limit [m]
create_jumbo_slave	l [m]	spin_tracking_method	y1_limit [m]
delta_ref_time [sec]	mat6_calc_method	superimpose	y2_limit [m]
descrip	new_branch	to_element	y_limit [m]
direction	offset [m]	to_line	
e_tot [eV]	offset_moves_aperture	tracking_method	

13.19 Girder Element Attributes

alias	dz_origin [m]	tilt [rad]	y_offset [m]
descrip	is_on	tilt_tot [rad]	y_offset_tot [m]
dphi_origin [rad]	l [m]	type	y_pitch
dpsi_origin [rad]	origin_ele	x_offset [m]	y_pitch_tot
dtheta_origin [rad]	origin_ele_ref_pt	x_offset_tot [m]	z_offset [m]
dx_origin [m]	ref_tilt [rad]	x_pitch	z_offset_tot [m]
dy_origin [m]	ref_tilt_tot [rad]	x_pitch_tot	

13.20 Group Element Attributes

accordion_edge [m]	end_edge [m]	start_edge	x_knot
alias	gang	type	
descrip	s_position [m]	var	

13.21 Hybrid Element Attributes

alias	e_tot [eV]	p0c_start [eV]	x1_limit [m]
aperture [m]	e_tot_start [eV]	ptc_integration_type	x2_limit [m]
aperture_at	ele_origin	ref_origin	x_limit [m]
aperture_type	l [m]	reference	y1_limit [m]
create_jumbo_slave	mat6_calc_method	spin_tracking_method	y2_limit [m]
delta_e_ref [eV]	offset [m]	superimpose	y_limit [m]
delta_ref_time [sec]	offset_moves_aperture	tracking_method	
descrip	p0c [eV]	type	

13.22 Instrument, Monitor, and Pipe Element Attributes Element Attributes

alias	hkick	ptc_integration_type	x_gain_err [m]
aperture [m]	integrator_order	ref_origin	x_limit [m]
aperture_at	is_on	reference	x_offset [m]
aperture_type	l [m]	space_charge_method	x_offset_calib [m]
bl_hkick [T*m]	l_hard_edge [m]	spin_fringe_on	x_offset_tot [m]
bl_vkick [T*m]	lord_pad1 [m]	spin_tracking_method	x_pitch
create_jumbo_slave	lord_pad2 [m]	sr_wake_file	x_pitch_tot
crunch [rad]	lr_freq_spread [Hz]	superimpose	y1_limit [m]
crunch_calib [rad]	lr_self_wake_on	symplectify	y2_limit [m]
csr_method	lr_wake_file	taylor_map_includes_offsets	y_gain_calib [m]
de_eta_meas	lr_wake_spline	tilt [rad]	y_gain_err [m]
delta_ref_time [sec]	mat6_calc_method	tilt_calib [rad]	y_limit [m]
descrip	n_ref_pass	tilt_tot [rad]	y_offset [m]
ds_step [m]	n_sample	tracking_method	y_offset_calib [m]
e_tot [eV]	noise	type	y_offset_tot [m]
ele_origin	num_steps	vkick	y_pitch
field_calc	offset [m]	wall	y_pitch_tot
field_overlaps	offset_moves_aperture	x1_limit [m]	z_offset [m]
fringe_at	osc_amplitude [m]	x2_limit [m]	z_offset_tot [m]
fringe_type	p0c [eV]	x_gain_calib [m]	

13.23 Kicker Element Attributes

a0 - a20, b0 - b20	fringe_type	p0c [eV]	v_displace [m^3]
alias	grid_field	ptc_canonical_coords	vkick
aperture [m]	h_displace [m]	ptc_integration_type	wall
aperture_at	hkick	r0_elec [m]	x1_limit [m]
aperture_type	integrator_order	r0_mag [m]	x2_limit [m]
bl_hkick [T*m]	is_on	ref_origin	x_limit [m]
bl_vkick [T*m]	l [m]	reference	x_offset [m]
cartesian_map	l_hard_edge [m]	scale_multipoles	x_offset_tot [m]
create_jumbo_slave	lord_pad1 [m]	space_charge_method	x_pitch
csr_method	lord_pad2 [m]	spin_fringe_on	x_pitch_tot
cylindrical_map	lr_freq_spread [Hz]	spin_tracking_method	y1_limit [m]
delta_ref_time [sec]	lr_self_wake_on	sr_wake_file	y2_limit [m]
descrip	lr_wake_file	superimpose	y_limit [m]
ds_step [m]	lr_wake_spline	symplectify	y_offset [m]
e_tot [eV]	mat6_calc_method	taylor_field	y_offset_tot [m]
ele_origin	multipoles_on	taylor_map_includes_offsets	y_pitch
field_calc	n_ref_pass	tilt [rad]	y_pitch_tot
field_master	num_steps	tilt_tot [rad]	z_offset [m]
field_overlaps	offset [m]	tracking_method	z_offset_tot [m]
fringe_at	offset_moves_aperture	type	

13.24 Kickers: Hkicker and Vkicker Element Attributes Element Attributes

a0 - a20, b0 - b20	fringe_type	p0c [eV]	x1_limit [m]
alias	grid_field	ptc_canonical_coords	x2_limit [m]
aperture [m]	integrator_order	ptc_integration_type	x_limit [m]
aperture_at	is_on	ref_origin	x_offset [m]
aperture_type	kick	reference	x_offset_tot [m]
bl_kick [T*m]	l [m]	scale_multipoles	x_pitch
cartesian_map	l_hard_edge [m]	space_charge_method	x_pitch_tot
create_jumbo_slave	lord_pad1 [m]	spin_fringe_on	y1_limit [m]
csr_method	lord_pad2 [m]	spin_tracking_method	y2_limit [m]
cylindrical_map	lr_freq_spread [Hz]	sr_wake_file	y_limit [m]
delta_ref_time [sec]	lr_self_wake_on	superimpose	y_offset [m]
descrip	lr_wake_file	symplectify	y_offset_tot [m]
ds_step [m]	lr_wake_spline	taylor_field	y_pitch
e_tot [eV]	mat6_calc_method	taylor_map_includes_offsets	y_pitch_tot
ele_origin	multipoles_on	tilt [rad]	z_offset [m]
field_calc	n_ref_pass	tilt_tot [rad]	z_offset_tot [m]
field_master	num_steps	tracking_method	
field_overlaps	offset [m]	type	
fringe_at	offset_moves_aperture	wall	

13.25 Lcavity Element Attributes

alias	field_autoscale	num_steps	type
aperture [m]	field_calc	offset [m]	vkick
aperture_at	field_master	offset_moves_aperture	voltage [Volt]
aperture_type	field_overlaps	p0c [eV]	voltage_err [Volt]
autoscale_amplitude	fringe_at	p0c_start [eV]	wall
autoscale_phase	fringe_type	phi0 [rad/2pi]	x1_limit [m]
bl_hkick [T*m]	gradient [eV/m]	phi0_autoscale [rad/2pi]	x2_limit [m]
bl_vkick [T*m]	gradient_err [eV/m]	phi0_err [rad/2pi]	x_limit [m]
cartesian_map	grid_field	phi0_multipass [rad/2pi]	x_offset [m]
cavity_type	hkick	ptc_integration_type	x_offset_tot [m]
coupler_angle [rad]	integrator_order	ref_origin	x_pitch
coupler_at	is_on	reference	x_pitch_tot
coupler_phase [rad/2pi]	l [m]	rf_frequency [Hz]	y1_limit [m]
coupler_strength	l_hard_edge [m]	space_charge_method	y2_limit [m]
create_jumbo_slave	longitudinal_mode	spin_fringe_on	y_limit [m]
csr_method	lord_pad1 [m]	spin_tracking_method	y_offset [m]
cylindrical_map	lord_pad2 [m]	sr_wake_file	y_offset_tot [m]
delta_ref_time [sec]	lr_freq_spread [Hz]	superimpose	y_pitch
descrip	lr_self_wake_on	symplectify	y_pitch_tot
ds_step [m]	lr_wake_file	taylor_field	z_offset [m]
e_loss [eV]	lr_wake_spline	taylor_map_includes_offsets	z_offset_tot [m]
e_tot [eV]	mat6_calc_method	tilt [rad]	
e_tot_start [eV]	n_cell	tilt_tot [rad]	
ele_origin	n_ref_pass	tracking_method	

13.26 Lens Element Attributes

alias	mat6_calc_method	tilt_tot [rad]	y1_limit [m]
aperture [m]	offset [m]	tracking_method	y2_limit [m]
aperture_at	offset_moves_aperture	type	y_limit [m]
aperture_type	p0c [eV]	wall	y_offset [m]
create_jumbo_slave	ptc_integration_type	x1_limit [m]	y_offset_tot [m]
delta_ref_time [sec]	radius [m]	x2_limit [m]	y_pitch
descrip	ref_origin	x_limit [m]	y_pitch_tot
e_tot [eV]	reference	x_offset [m]	z_offset [m]
ele_origin	spin_tracking_method	x_offset_tot [m]	z_offset_tot [m]
focal_strength [1/m]	superimpose	x_pitch	
l [m]	tilt [rad]	x_pitch_tot	

13.27 Marker Element Attributes

alias	lr_self_wake_on	superimpose	x_pitch
aperture [m]	lr_wake_file	tilt [rad]	x_pitch_tot
aperture_at	lr_wake_spline	tilt_calib [rad]	x_ray_line_len [m]
aperture_type	mat6_calc_method	tilt_tot [rad]	y1_limit [m]
create_jumbo_slave	n_sample	tracking_method	y2_limit [m]
crunch [rad]	noise	type	y_gain_calib [m]
crunch_calib [rad]	offset [m]	wall	y_gain_err [m]
de_eta_meas	offset_moves_aperture	x1_limit [m]	y_limit [m]
delta_ref_time [sec]	osc_amplitude [m]	x2_limit [m]	y_offset [m]
descrip	p0c [eV]	x_gain_calib [m]	y_offset_calib [m]
e_tot [eV]	ptc_integration_type	x_gain_err [m]	y_offset_tot [m]
ele_origin	ref_origin	x_limit [m]	y_pitch
is_on	reference	x_offset [m]	y_pitch_tot
l [m]	spin_tracking_method	x_offset_calib [m]	z_offset [m]
lr_freq_spread [Hz]	sr_wake_file	x_offset_tot [m]	z_offset_tot [m]

13.28 Mask Element Attributes

alias	mat6_calc_method	tilt [rad]	x_pitch_tot
aperture [m]	mode	tilt_tot [rad]	y1_limit [m]
aperture_at	offset [m]	tracking_method	y2_limit [m]
aperture_type	offset_moves_aperture	type	y_limit [m]
create_jumbo_slave	p0c [eV]	wall	y_offset [m]
delta_ref_time [sec]	ptc_integration_type	x1_limit [m]	y_offset_tot [m]
descrip	ref_origin	x2_limit [m]	y_pitch
e_tot [eV]	ref_wavelength [m]	x_limit [m]	y_pitch_tot
ele_origin	reference	x_offset [m]	z_offset [m]
field_scale_factor	spin_tracking_method	x_offset_tot [m]	z_offset_tot [m]
is_on	superimpose	x_pitch	

13.29 Match Element Attributes

alias	c22_mat0	is_on	reference
alpha_a0	c22_mat1	l [m]	spin_tracking_method
alpha_a1	create_jumbo_slave	mat6_calc_method	superimpose
alpha_b0	delta_ref_time [sec]	match_end	tracking_method
alpha_b1	delta_time [sec]	match_end_input	type
aperture [m]	descrip	match_end_orbit	x0 [m]
aperture_at	dphi_a [rad]	match_end_orbit_input	x1 [m]
aperture_type	dphi_b [rad]	offset [m]	x1_limit [m]
beta_a0	e_tot [eV]	offset_moves_aperture	x2_limit [m]
beta_a1	ele_origin	p0c [eV]	x_limit [m]
beta_b0	eta_x0 [m]	ptc_integration_type	y0 [m]
beta_b1	eta_x1 [m]	px0	y1 [m]
c11_mat0	eta_y0 [m]	px1	y1_limit [m]
c11_mat1	eta_y1 [m]	py0	y2_limit [m]
c12_mat0 [m]	etap_x0	py1	y_limit [m]
c12_mat1 [m]	etap_x1	pz0	z0 [m]
c21_mat0 [1/m]	etap_y0	pz1	z1 [m]
c21_mat1 [1/m]	etap_y1	ref_origin	

13.30 Mirror Element Attributes

alias	elliptical_curvature_y [1/m]	reference	x_offset [m]
aperture [m]	elliptical_curvature_z [1/m]	spherical_curvature [1/m]	x_offset_tot [m]
aperture_at	graze_angle [rad]	spin_tracking_method	x_pitch
aperture_type	l [m]	superimpose	x_pitch_tot
create_jumbo_slave	mat6_calc_method	surface	y1_limit [m]
critical_angle [rad]	offset [m]	tilt [rad]	y2_limit [m]
curvature_x0_y2 [1/m]	offset_moves_aperture	tilt_tot [rad]	y_limit [m]
delta_ref_time [sec]	p0c [eV]	tracking_method	y_offset [m]
descrip	ptc_integration_type	type	y_offset_tot [m]
diffraction_limited	ref_origin	wall	y_pitch
e_tot [eV]	ref_tilt [rad]	x1_limit [m]	y_pitch_tot
ele_origin	ref_tilt_tot [rad]	x2_limit [m]	z_offset [m]
elliptical_curvature_x [1/m]	ref_wavelength [m]	x_limit [m]	z_offset_tot [m]

13.31 Multilayer_Mirror Element Attributes

alias	elliptical_curvature_z [1/m]	spherical_curvature [1/m]	x_offset_tot [m]
aperture [m]	graze_angle [rad]	spin_tracking_method	x_pitch
aperture_at	l [m]	superimpose	x_pitch_tot
aperture_type	mat6_calc_method	surface	y1_limit [m]
create_jumbo_slave	material_type	tilt [rad]	y2_limit [m]
curvature_x0_y2 [1/m]	n_cell	tilt_tot [rad]	y_limit [m]
d1_thickness [m]	offset [m]	tracking_method	y_offset [m]
d2_thickness [m]	offset_moves_aperture	type	y_offset_tot [m]
delta_ref_time [sec]	p0c [eV]	v1_unitcell [m^3]	y_pitch
descrip	ptc_integration_type	v2_unitcell [m^3]	y_pitch_tot
diffraction_limited	ref_origin	wall	z_offset [m]
e_tot [eV]	ref_tilt [rad]	x1_limit [m]	z_offset_tot [m]
ele_origin	ref_tilt_tot [rad]	x2_limit [m]	
elliptical_curvature_x [1/m]	ref_wavelength [m]	x_limit [m]	
elliptical_curvature_y [1/m]	reference	x_offset [m]	

13.32 Multipole Element Attributes

alias	is_on	spin_tracking_method	x_offset [m]
aperture [m]	k0l - k20l, t0 - t20	superimpose	x_offset_tot [m]
aperture_at	l [m]	tilt [rad]	y1_limit [m]
aperture_type	mat6_calc_method	tilt_tot [rad]	y2_limit [m]
create_jumbo_slave	offset [m]	tracking_method	y_limit [m]
delta_ref_time [sec]	offset_moves_aperture	type	y_offset [m]
descrip	p0c [eV]	wall	y_offset_tot [m]
e_tot [eV]	ptc_integration_type	x1_limit [m]	z_offset [m]
ele_origin	ref_origin	x2_limit [m]	z_offset_tot [m]
field_master	reference	x_limit [m]	

13.33 Octupole Element Attributes

a0 - a20, b0 - b20	fringe_at	offset_moves_aperture	type
alias	fringe_type	p0c [eV]	vkick
aperture [m]	grid_field	ptc_canonical_coords	wall
aperture_at	hkick	ptc_integration_type	x1_limit [m]
aperture_type	integrator_order	r0_elec [m]	x2_limit [m]
b3_gradient [T/m^3]	is_on	r0_mag [m]	x_limit [m]
bl_hkick [T*m]	k3 [1/m^4]	ref_origin	x_offset [m]
bl_vkick [T*m]	l [m]	reference	x_offset_tot [m]
cartesian_map	l_hard_edge [m]	scale_multipoles	x_pitch
create_jumbo_slave	lord_pad1 [m]	space_charge_method	x_pitch_tot
csr_method	lord_pad2 [m]	spin_fringe_on	y1_limit [m]
cylindrical_map	lr_freq_spread [Hz]	spin_tracking_method	y2_limit [m]
delta_ref_time [sec]	lr_self_wake_on	sr_wake_file	y_limit [m]
descrip	lr_wake_file	superimpose	y_offset [m]
ds_step [m]	lr_wake_spline	symplectify	y_offset_tot [m]
e_tot [eV]	mat6_calc_method	taylor_field	y_pitch
ele_origin	multipoles_on	taylor_map_includes_offsets	y_pitch_tot
field_calc	n_ref_pass	tilt [rad]	z_offset [m]
field_master	num_steps	tilt_tot [rad]	z_offset_tot [m]
field_overlaps	offset [m]	tracking_method	

13.34 Overlay Element Attributes

alias	gang	var
descrip	type	x_knot

13.35 Patch Element Attributes

alias	ele_origin	ref_origin	x_limit [m]
aperture [m]	field_calc	reference	x_offset [m]
aperture_at	flexible	spin_tracking_method	x_pitch
aperture_type	l [m]	superimpose	y1_limit [m]
create_jumbo_slave	mat6_calc_method	t_offset [sec]	y2_limit [m]
delta_ref_time [sec]	offset [m]	tilt [rad]	y_limit [m]
descrip	offset_moves_aperture	tracking_method	y_offset [m]
downstream_ele_dir	p0c [eV]	type	y_pitch
e_tot [eV]	p0c_set [eV]	upstream_ele_dir	z_offset [m]
e_tot_offset [eV]	p0c_start [eV]	wall	
e_tot_set [eV]	ptc_integration_type	x1_limit [m]	
e_tot_start [eV]	ref_coordinates	x2_limit [m]	

13.36 Photon_Init Element Attributes

alias	energy_distribution	sig_x [m]	x_offset [m]
aperture [m]	l [m]	sig_y [m]	x_offset_tot [m]
aperture_at	mat6_calc_method	sig_z [m]	x_pitch
aperture_type	offset [m]	spatial_distribution	x_pitch_tot
create_jumbo_slave	offset_moves_aperture	spin_tracking_method	y1_limit [m]
delta_ref_time [sec]	p0c [eV]	superimpose	y2_limit [m]
descrip	physical_source	tilt [rad]	y_limit [m]
ds_slice [m]	ptc_integration_type	tilt_tot [rad]	y_offset [m]
e2_center [eV]	ref_origin	tracking_method	y_offset_tot [m]
e2_probability	ref_wavelength [m]	transverse_sigma_cut	y_pitch
e_center [eV]	reference	type	y_pitch_tot
e_center_relative_to_ref	scale_field_to_one	velocity_distribution	z_offset [m]
e_field_x [V/m]	sig_e [eV]	wall	z_offset_tot [m]
e_field_y [V/m]	sig_e2 [eV]	x1_limit [m]	
e_tot [eV]	sig_vx	x2_limit [m]	
ele_origin	sig_vy	x_limit [m]	

13.37 Quadrupole Element Attributes

a0 - a20, b0 - b20	fq2 [m]	offset_moves_aperture	vkick
alias	fringe_at	p0c [eV]	wall
aperture [m]	fringe_type	ptc_canonical_coords	x1_limit [m]
aperture_at	grid_field	ptc_integration_type	x2_limit [m]
aperture_type	hkick	r0_elec [m]	x_limit [m]
b1_gradient [T/m]	integrator_order	r0_mag [m]	x_offset [m]
bl_hkick [T*m]	is_on	ref_origin	x_offset_tot [m]
bl_vkick [T*m]	k1 [1/m^2]	reference	x_pitch
cartesian_map	l [m]	scale_multipoles	x_pitch_tot
create_jumbo_slave	l_hard_edge [m]	space_charge_method	y1_limit [m]
csr_method	lord_pad1 [m]	spin_fringe_on	y2_limit [m]
cylindrical_map	lord_pad2 [m]	spin_tracking_method	y_limit [m]
delta_ref_time [sec]	lr_freq_spread [Hz]	sr_wake_file	y_offset [m]
descrip	lr_self_wake_on	superimpose	y_offset_tot [m]
ds_step [m]	lr_wake_file	symplectify	y_pitch
e_tot [eV]	lr_wake_spline	taylor_field	y_pitch_tot
ele_origin	mat6_calc_method	taylor_map_includes_offsets	z_offset [m]
field_calc	multipoles_on	tilt [rad]	z_offset_tot [m]
field_master	n_ref_pass	tilt_tot [rad]	
field_overlaps	num_steps	tracking_method	
fq1 [m]	offset [m]	type	

13.38 RFcavity Element Attributes

alias	field_autoscale	n_ref_pass	tracking_method
aperture [m]	field_calc	num_steps	type
aperture_at	field_overlaps	offset [m]	vkick
aperture_type	fringe_at	offset_moves_aperture	voltage [Volt]
autoscale_amplitude	fringe_type	p0c [eV]	wall
autoscale_phase	gradient [eV/m]	phi0 [rad/2pi]	x1_limit [m]
bl_hkick [T*m]	grid_field	phi0_autoscale [rad/2pi]	x2_limit [m]
bl_vkick [T*m]	harmon	phi0_multipass [rad/2pi]	x_limit [m]
cartesian_map	hkick	ptc_integration_type	x_offset [m]
cavity_type	integrator_order	ref_origin	x_offset_tot [m]
coupler_angle [rad]	is_on	reference	x_pitch
coupler_at	l [m]	rf_frequency [Hz]	x_pitch_tot
coupler_phase [rad/2pi]	l_hard_edge [m]	space_charge_method	y1_limit [m]
coupler_strength	longitudinal_mode	spin_fringe_on	y2_limit [m]
create_jumbo_slave	lord_pad1 [m]	spin_tracking_method	y_limit [m]
csr_method	lord_pad2 [m]	sr_wake_file	y_offset [m]
cylindrical_map	lr_freq_spread [Hz]	superimpose	y_offset_tot [m]
delta_ref_time [sec]	lr_self_wake_on	symplectify	y_pitch
descrip	lr_wake_file	taylor_field	y_pitch_tot
ds_step [m]	lr_wake_spline	taylor_map_includes_offsets	z_offset [m]
e_tot [eV]	mat6_calc_method	tilt [rad]	z_offset_tot [m]
ele_origin	n_cell	tilt_tot [rad]	

13.39 Sad_Mult Element Attributes

a0 - a20, b0 - b20	fb1 [m]	reference	x_offset_mult [m]
alias	fb2 [m]	rho [m]	x_offset_tot [m]
aperture [m]	fq1 [m]	spin_fringe_on	x_pitch
aperture_at	fq2 [m]	spin_tracking_method	x_pitch_mult
aperture_type	fringe_at	superimpose	x_pitch_tot
b_field [T]	fringe_type	symplectify	y1_limit [m]
bs_field [T]	is_on	taylor_map_includes_offsets	y2_limit [m]
create_jumbo_slave	ks [1/m]	tilt [rad]	y_limit [m]
delta_ref_time [sec]	l [m]	tilt_tot [rad]	y_offset [m]
descrip	mat6_calc_method	tracking_method	y_offset_mult [m]
ds_step [m]	num_steps	type	y_offset_tot [m]
e1 [rad]	offset [m]	wall	y_pitch
e2 [rad]	offset_moves_aperture	x1_limit [m]	y_pitch_mult
e_tot [eV]	p0c [eV]	x2_limit [m]	y_pitch_tot
ele_origin	ptc_integration_type	x_limit [m]	z_offset [m]
eps_step_scale [m]	ref_origin	x_offset [m]	z_offset_tot [m]

13.40 Sample Element Attributes

alias	elliptical_curvature_z [1/m]	spin_tracking_method	x_offset_tot [m]
aperture [m]	l [m]	superimpose	x_pitch
aperture_at	mat6_calc_method	surface	x_pitch_tot
aperture_type	material_type	tilt [rad]	y1_limit [m]
create_jumbo_slave	mode	tilt_tot [rad]	y2_limit [m]
curvature_x0_y2 [1/m]	offset [m]	tracking_method	y_limit [m]
delta_ref_time [sec]	offset_moves_aperture	type	y_offset [m]
descrip	p0c [eV]	wall	y_offset_tot [m]
e_tot [eV]	ptc_integration_type	x1_limit [m]	y_pitch
ele_origin	ref_origin	x2_limit [m]	y_pitch_tot
elliptical_curvature_x [1/m]	reference	x_limit [m]	z_offset [m]
elliptical_curvature_y [1/m]	spherical_curvature [1/m]	x_offset [m]	z_offset_tot [m]

13.41 Sextupole Element Attributes

a0 - a20, b0 - b20	fringe_at	offset_moves_aperture	type
alias	fringe_type	p0c [eV]	vkick
aperture [m]	grid_field	ptc_canonical_coords	wall
aperture_at	hkick	ptc_integration_type	x1_limit [m]
aperture_type	integrator_order	r0_elec [m]	x2_limit [m]
b2_gradient [T/m^2]	is_on	r0_mag [m]	x_limit [m]
bl_hkick [T*m]	k2 [1/m^3]	ref_origin	x_offset [m]
bl_vkick [T*m]	l [m]	reference	x_offset_tot [m]
cartesian_map	l_hard_edge [m]	scale_multipoles	x_pitch
create_jumbo_slave	lord_pad1 [m]	space_charge_method	x_pitch_tot
csr_method	lord_pad2 [m]	spin_fringe_on	y1_limit [m]
cylindrical_map	lr_freq_spread [Hz]	spin_tracking_method	y2_limit [m]
delta_ref_time [sec]	lr_self_wake_on	sr_wake_file	y_limit [m]
descrip	lr_wake_file	superimpose	y_offset [m]
ds_step [m]	lr_wake_spline	symplectify	y_offset_tot [m]
e_tot [eV]	mat6_calc_method	taylor_field	y_pitch
ele_origin	multipoles_on	taylor_map_includes_offsets	y_pitch_tot
field_calc	n_ref_pass	tilt [rad]	z_offset [m]
field_master	num_steps	tilt_tot [rad]	z_offset_tot [m]
field_overlaps	offset [m]	tracking_method	

13.42 Sol_Quad Element Attributes

a0 - a20, b0 - b20	fringe_at	offset_moves_aperture	vkick
alias	fringe_type	p0c [eV]	wall
aperture [m]	grid_field	ptc_canonical_coords	x1_limit [m]
aperture_at	hkick	ptc_integration_type	x2_limit [m]
aperture_type	integrator_order	r0_elec [m]	x_limit [m]
b1_gradient [T/m]	is_on	r0_mag [m]	x_offset [m]
bl_hkick [T*m]	k1 [1/m^2]	ref_origin	x_offset_tot [m]
bl_vkick [T*m]	ks [1/m]	reference	x_pitch
bs_field [T]	l [m]	scale_multipoles	x_pitch_tot
cartesian_map	l_hard_edge [m]	space_charge_method	y1_limit [m]
create_jumbo_slave	lord_pad1 [m]	spin_fringe_on	y2_limit [m]
csr_method	lord_pad2 [m]	spin_tracking_method	y_limit [m]
cylindrical_map	lr_freq_spread [Hz]	sr_wake_file	y_offset [m]
delta_ref_time [sec]	lr_self_wake_on	superimpose	y_offset_tot [m]
descrip	lr_wake_file	symplectify	y_pitch
ds_step [m]	lr_wake_spline	taylor_field	y_pitch_tot
e_tot [eV]	mat6_calc_method	taylor_map_includes_offsets	z_offset [m]
ele_origin	multipoles_on	tilt [rad]	z_offset_tot [m]
field_calc	n_ref_pass	tilt_tot [rad]	
field_master	num_steps	tracking_method	
field_overlaps	offset [m]	type	

13.43 Solenoid Element Attributes

a0 - a20, b0 - b20	fringe_at	offset_moves_aperture	type
alias	fringe_type	p0c [eV]	vkick
aperture [m]	grid_field	ptc_canonical_coords	wall
aperture_at	hkick	ptc_integration_type	x1_limit [m]
aperture_type	integrator_order	r0_elec [m]	x2_limit [m]
bl_hkick [T*m]	is_on	r0_mag [m]	x_limit [m]
bl_vkick [T*m]	ks [1/m]	ref_origin	x_offset [m]
bs_field [T]	l [m]	reference	x_offset_tot [m]
cartesian_map	l_hard_edge [m]	scale_multipoles	x_pitch
create_jumbo_slave	lord_pad1 [m]	space_charge_method	x_pitch_tot
csr_method	lord_pad2 [m]	spin_fringe_on	y1_limit [m]
cylindrical_map	lr_freq_spread [Hz]	spin_tracking_method	y2_limit [m]
delta_ref_time [sec]	lr_self_wake_on	sr_wake_file	y_limit [m]
descrip	lr_wake_file	superimpose	y_offset [m]
ds_step [m]	lr_wake_spline	symplectify	y_offset_tot [m]
e_tot [eV]	mat6_calc_method	taylor_field	y_pitch
ele_origin	multipoles_on	taylor_map_includes_offsets	y_pitch_tot
field_calc	n_ref_pass	tilt [rad]	z_offset [m]
field_master	num_steps	tilt_tot [rad]	z_offset_tot [m]
field_overlaps	offset [m]	tracking_method	

13.44 Taylor Element Attributes

alias	mat6_calc_method	taylor_map_includes_offsets	x_ref [m]
aperture [m]	offset [m]	tilt [rad]	y1_limit [m]
aperture_at	offset_moves_aperture	tilt_tot [rad]	y2_limit [m]
aperture_type	p0c [eV]	tracking_method	y_limit [m]
create_jumbo_slave	ptc_integration_type	tt<out><n1><n2>...	y_offset [m]
delta_e_ref [eV]	px_ref	type	y_offset_tot [m]
delta_ref_time [sec]	py_ref	wall	y_pitch
descrip	pz_ref	x1_limit [m]	y_pitch_tot
e_tot [eV]	ref_orbit	x2_limit [m]	y_ref [m]
ele_origin	ref_origin	x_limit [m]	z_offset [m]
is_on	reference	x_offset [m]	z_offset_tot [m]
l [m]	spin_tracking_method	x_offset_tot [m]	z_ref [m]
lord_pad1 [m]	superimpose	x_pitch	
lord_pad2 [m]	symplectify	x_pitch_tot	

13.45 Wiggler and Undulator Element Attributes Element Attributes

a0 - a20, b0 - b20	g_max [1/m]	osc_amplitude [m]	type
alias	grid_field	p0c [eV]	vkick
aperture [m]	hkick	polarity	wall
aperture_at	integrator_order	ptc_canonical_coords	x1_limit [m]
aperture_type	is_on	ptc_integration_type	x2_limit [m]
b_max [T]	k1_pseudo [1/m^2]	r0_elec [m]	x_limit [m]
bl_hkick [T*m]	l [m]	r0_mag [m]	x_offset [m]
bl_vkick [T*m]	l_hard_edge [m]	ref_origin	x_offset_tot [m]
cartesian_map	l_period [m]	reference	x_pitch
create_jumbo_slave	lord_pad1 [m]	scale_multipoles	x_pitch_tot
csr_method	lord_pad2 [m]	space_charge_method	x_ray_line_len [m]
cylindrical_map	lr_freq_spread [Hz]	spin_fringe_on	y1_limit [m]
delta_ref_time [sec]	lr_self_wake_on	spin_tracking_method	y2_limit [m]
descrip	lr_wake_file	sr_wake_file	y_limit [m]
ds_step [m]	lr_wake_spline	superimpose	y_offset [m]
e_tot [eV]	mat6_calc_method	symplectify	y_offset_tot [m]
ele_origin	multipoles_on	taylor_field	y_pitch
field_calc	n_period	taylor_map_includes_offsets	y_pitch_tot
field_master	n_ref_pass	term	z_offset [m]
field_overlaps	num_steps	tilt [rad]	z_offset_tot [m]
fringe_at	offset [m]	tilt_tot [rad]	
fringe_type	offset_moves_aperture	tracking_method	

Part II

Conventions and Physics

Chapter 14

Coordinates

Bmad uses three coordinate systems as illustrated in Fig. 14.1. First, the `global` (also called "`floor`") coordinates are independent of the accelerator. Thus such things as the building the accelerator is in may be described using `global` coordinates.

It is not convenient to describe the position of the beam using the `global` coordinate system so a "`local`" coordinate system is used (§14.1). This curvilinear coordinate system defines the nominal position of the lattice elements. The relationship between the `local` and `global` coordinate systems is described in §14.2.

The "nominal" position of a lattice element is the position of the element without what are called "misalignments" (that is, position and orientation shifts). Each lattice element has "`element body`" coordinates which are attached to the physical element. That is, the electric and magnetic fields of an element are described with respect to `element` coordinates. If there are no misalignments, the `element` coordinates are aligned with the `local` coordinates. With misalignments, `element` and `local` coordinates will not coincide. The transformation between `local` and `element` coordinates is given in §14.3.

When discussing `local` vs `element` coordinates, it can be less confusing to use the name "`laboratory`" cordinates instead of `local` coordinates.

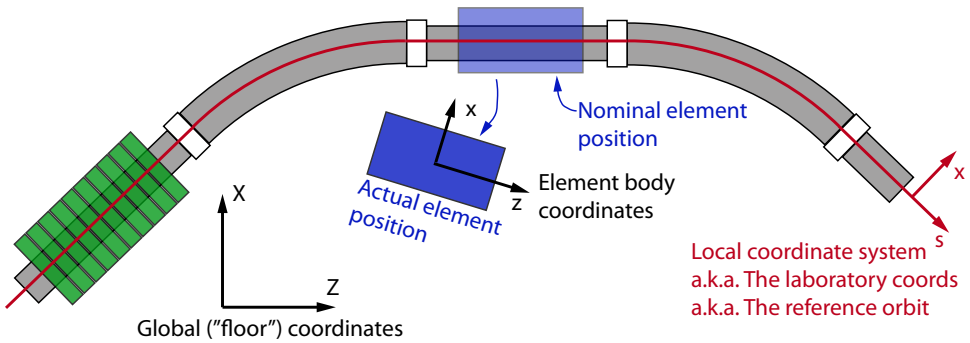


Figure 14.1: The three coordinate systems used by *Bmad*: The `global` (or "`floor`") coordinate system is independent of the accelerator. The `local` curvilinear coordinate system follows the bends of the accelerator. Each lattice element has `element body` coordinates which, if the element is not "misaligned" is the same as the `local` coordinates.

14.1 Local Reference Coordinates

14.1.1 Local Reference Orbit

The `local reference orbit` is the curved path used to define a coordinate system for describing a particle's position as shown in Fig. 14.2. The reference orbit is also used for orientating lattice elements in space. At a given time t, a particle's position can be described by a point on the reference orbit a distance s relative to the reference orbit's zero position plus a transverse offset. This point on the reference orbit is used as the origin of the local (x, y, z) coordinate system with the z–axis tangent to the reference orbit. The z–axis will generally be pointing in the direction of increasing s but, as discussed in detail below, will point counter to s for elements that are `reversed`. The x and y–axes are perpendicular to the reference orbit. As will be shown later, If the lattice has no vertical bends, the y–axis is in the vertical direction and the x–axis is in the horizontal plane. Notice that, by construction, the particle is always at $z = 0$. The coordinate system so constructed is called the **''local coordinate system''** or sometimes the **''laboratory coordinate system''** when there is need to distinguish it from the **''element coordinate system''** (§22.2) which is attached to the physical element.

There is a separate reference orbit for each branch (§1.2) of a lattice. When there are multiple branches, the reference orbit of a branch must not depend upon the configuration of branches later on in the array of branches in the lattice. As a consequence, the reference orbits of the branches can be calculated one at a time starting with the first branch.

Notice that, in a `wiggler`, the reference orbit, which is a straight line, does *not* correspond to the orbit that any actual particle could travel. Typically the physical element is centered with respect to the reference curve. However, by specifying offsets, pitches or a tilt (See §4.6), the physical element may

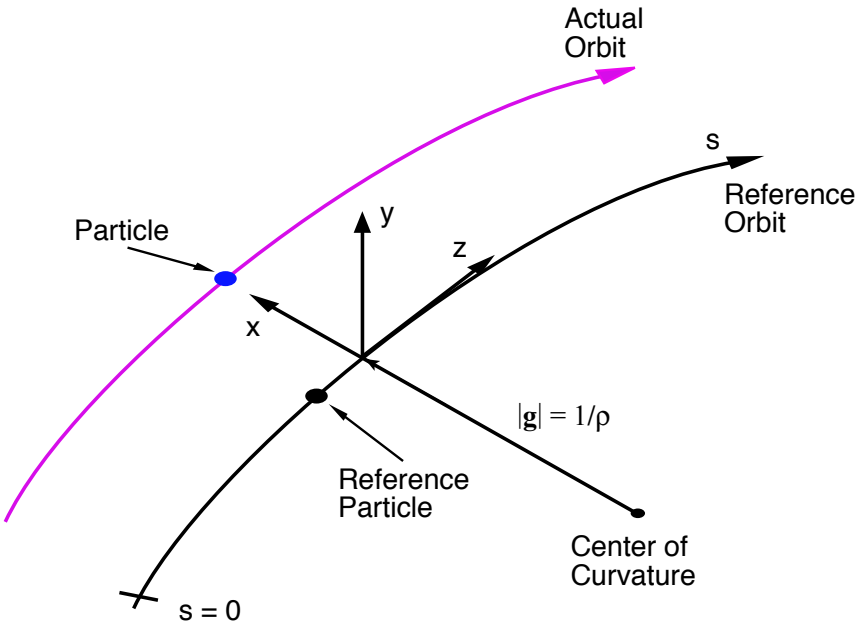


Figure 14.2: The local reference coordinate system. By construction, a particle's z coordinate is zero. This is not to be confused with the phase space z coordinate (§14.4.2). The curvature vector $\mathbf{g}$ lies in the x-y plane and has a magnitude of $1/\rho$ where ρ is the bending radius. The z-axis will normally be parallel to the s-axis but for `reversed` elements it will be antiparallel.

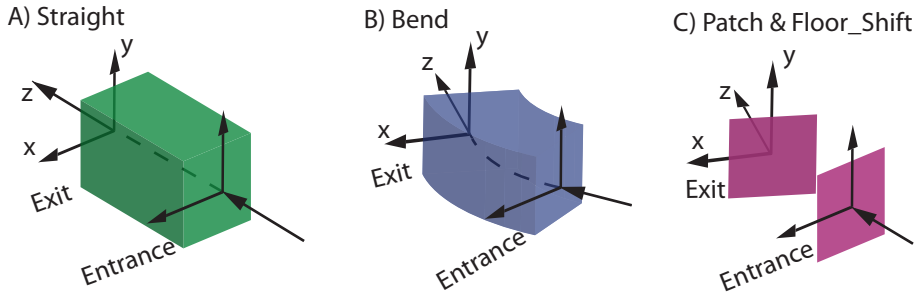


Figure 14.3: Element reference frame LEGO blocks. Shown are the blocks along with the entrance and exit reference frames. The physical element may be displaced in the local coordinate frame using offsets, tilt, and pitches. A) For straight line elements the two frames are colinear. B) For bends elements, the two frames are rotated with respect to each other. C) For `patch` and `floor_shift` elements the exit frame may be arbitrarily positioned with respect to the entrance.

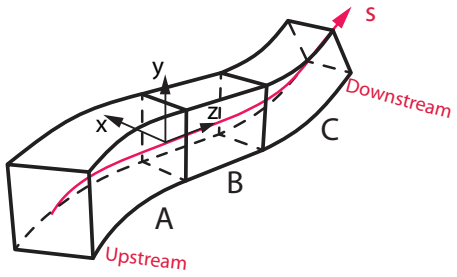


Figure 14.4: Element LEGO block concatenation. A) The reference orbit is constructed by concatenating the LEGO blocks together. B) For normal (non-reversed) elements the z-axis is parallel with the s-axis. C) For `reversed` elements the z-axis is antiparallel with the s-axis. By definition, the "entrance" and "exit" ends of an element are fixed relative to the element (relative to the z-axis) while the "upstream" and "downstream" ends are fixed relative to the s-axis.

be arbitrarily shifted with respect to its reference curve. Shifting a physical magnet with respect to its reference curve generally means that the reference curve does *not* correspond to the orbit that any actual particle could travel.

Do not confuse this reference orbit (which defines the local coordinate system) with the reference orbit about which the transfer maps are calculated (§31.2). The former is fixed by the lattice while the latter can be any arbitrary orbit.

14.1.2 Reference Orbit Construction: Upstream, Downstream, Entrance, and Exit Element Ends

Another way of thinking about the reference orbit is to imagine that each element has assigned to it a local coordinate reference frame assigned to it in the shape of a LEGO block[1] as shown in Fig. 14.3. Every block has an "`entrance`" and an "`exit`" reference frame. For most types of elements, the LEGO block for the element is "straight" as shown in Fig. 14.3A. That is, the reference curve through the block is a straight line segment and the length of the block is the length of the element. For a `bend` (§3.5), the reference curve through the block is a segment of a circular arc as shown in Fig. 14.3B. With a zero

[1] Thanks to Dan Abell for this analogy.

`ref_tilt`, the rotation axis between the entrance and exit frames of a bend is the y-axis (§14.2). For `patch` (§3.37), and `floor_shift` (§3.18) elements, the exit face can can arbitrarily oriented with respect to the entrance end. In this case, the reference orbit between the entrance and exit faces is not defined.

Assuming for the moment that there are no `fiducial` elements present, the construction of the reference orbit starts at the beginning element of a branch. If the branch is a `root` branch (§1.3), The orientation of the beginning element can be set via the appropriate positioning statements (§8.4). If the branch is not a `root` branch, the position of the beginning element is determined by the position of the `fork` or `photon_fork` element from which the branch forks from.

Once the beginning element in a branch is positioned, succeeding element LEGO blocks are concatenated together as illustrated in Fig. 14.4A. When a block is joined, the end of the block that is mated to the previous block is called the "`upstream`" end and the other end which will be mated to the following block is called the "`downstream`" end. Normally, the `entrance` end of a block is used as the `upstream` end the `exit` end is used as the `downstream` end as shown in Fig. 14.4B. However, for `reversed` elements (§6.3), the `upstream` end is the `exit` end of the block and vice versa. To put it another way, the `entrance` and `exit` end of the blocks reference the physical element. Thus, for example, the `e1` edge of a bend (§3.5) is always at the `entrance` face and the `e2` is always at the `exit`face. Also the field of a wiggler is (§3.47) is referenced to $z = 0$ which is at the `entrance` end. The `upstream` and `downstream` ends, on the other hand, are referenced to the reference orbit and the `downstream` end always is at a larger s position relative to the `upstream` end.

In all cases, when a LEGO block is joined to the previous block, the coordinate system at the mating end of the block (the `upstream` end) will be aligned with the mating end of the previous block (the `downstream` end). The situation where one of the blocks is reversed and the other one not, and neither is a `patch` nor `floor_shift` element, does not make physical sense since a particle which is moving downstream, when it comes to the face joining the blocks will have to magically reverse direction in order to travel through the next block. Thus, to have normal and reversed elements in a branch, `reflection patches` must be used in between. Whether it makes physical sense to put a `patch` element next to another element is more complicated. This depends upon whether the other element is reversed or not and whether the `patch` is reversed, whether the `patch` is `reflecting`, and the sign of `z_offset` for the `patch`. The basic criterion is that a particle leaving one block must enter the next. See Section §11.3 for an example.

If there are `fiducial` elements, the local reference frame is constructed beginning at these elements.

14.1.3 Patch Element Local Coordinates

Generally, if a particle is reasonably near the reference orbit, there is a one-to-one mapping between the particle's position and (x, y, s) coordinates. A `patch` (§3.37) elements with a non-zero `x_pitch` or non-zero `y_pitch` breaks the one-to-one mapping. This is illustrated in Fig. 14.5. The `patch` element, shown schematically as an, irregular quadrilateral, is sandwiched between elements `ele_a` and `ele_b`. The local coordinate system with origin at α are the coordinates at the end of `ele_a`. The coordinates at the end of the `patch` has its origin labeled γ. By convention, the length of the patch `L` is taken to be the longitudinal distance from α to γ with the `patch`'s exit coordinates defining the longitudinal direction. The "beginning" point of the `patch` on the reference orbit a distance `L` from point γ is labeled β in the figure.

In the local (x, y, s) coordinate system a particle at α will have some value $s = s_0$. A particle at point β will have the same value $s = s_0$ and a particle at γ will have $s = s_1 = s_0 + L$. A particle at point r_a in Fig. 14.5 illustrates the problem of assigning (x, y, s) coordinates to a given position. If the particle is considered to be within the region of `ele_a`, the particle's s position will be s_{a2} which is greater

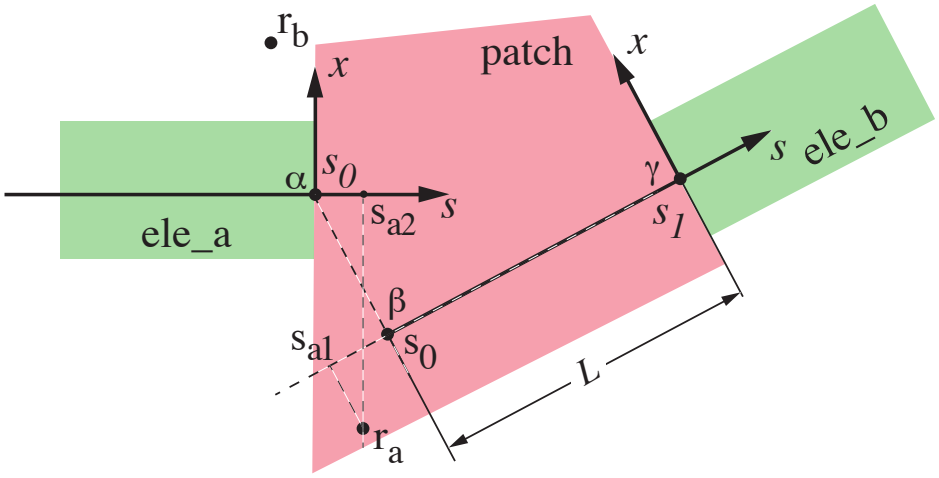


Figure 14.5: The local reference coordinates in a `patch` element. The `patch` element, shown schematically as an irregular quadrilateral, is sandwiched between elements `ele_a` and `ele_b`. `L` is the length of the `patch`. In this example, the `patch` has a finite `x_pitch`.

than the value s_0 at the exit end of the element. This contradicts the expectation that particles within `ele_a` will have $s \leq s_0$. If, on the other hand, the particle is considered to be within the `patch` region, the particle's s position will be s_{a1} which is less than the value s_0 at the entrance to the patch. This contradicts the expectation that a particles within the `patch` will have $s \geq s_0$.

To resolve this problem, *Bmad* considers a particle at position r_a to be within the `patch` region. This means that there is, in theory, no lower limit to the s-position that a particle in the `patch` region can have. This also implies that there is a discontinuity in the s-position of a particle crossing the exit face of `ele1`. Typically, when particles are translated from the exit face of one element to the exit face of the next, this `patch` problem does not appear. It only appears when the track between faces is considered.

Notice that a particle at position r_b in Fig. 14.5 can simultaneously be considered to be in either `ele_a` or the `patch`. While this creates an ambiguity it does not complicate tracking.

14.2 Global Coordinates

The Cartesian `global` coordinate system, also called the 'floor" coordinate system, is the coordinate system "attached to the earth" that is used to describe the local coordinate system. Following the *MAD* convention, the `global` coordinate axis are labeled (X, Y, Z). Conventionally, Y is the "vertical" coordinate and (X, Z) are the "horizontal" coordinates. To describe how the local coordinate system is oriented within the global coordinate system, each point on the s-axis of the local coordinate system is characterized by its (X, Y, Z) position and by three angles $\theta(s)$, $\phi(s)$, and $\psi(s)$ that describe the orientation of the local coordinate axes as shown in Fig. 14.6. These three angles are defined as follows:

$\theta(s)$ **Azimuth angle:** Angle in the (X, Z) plane between the Z–axis and the projection of the z–axis onto the (X, Z) plane. A positive angle of $\theta = \pi/2$ corresponds to the projected z–axis pointing in the positive X direction.

$\phi(s)$ **Pitch (elevation) angle:** Angle between the z–axis and the (X, Z) plane. A positive angle of $\phi = \pi/2$ corresponds to the z–axis pointing in the positive Y direction.

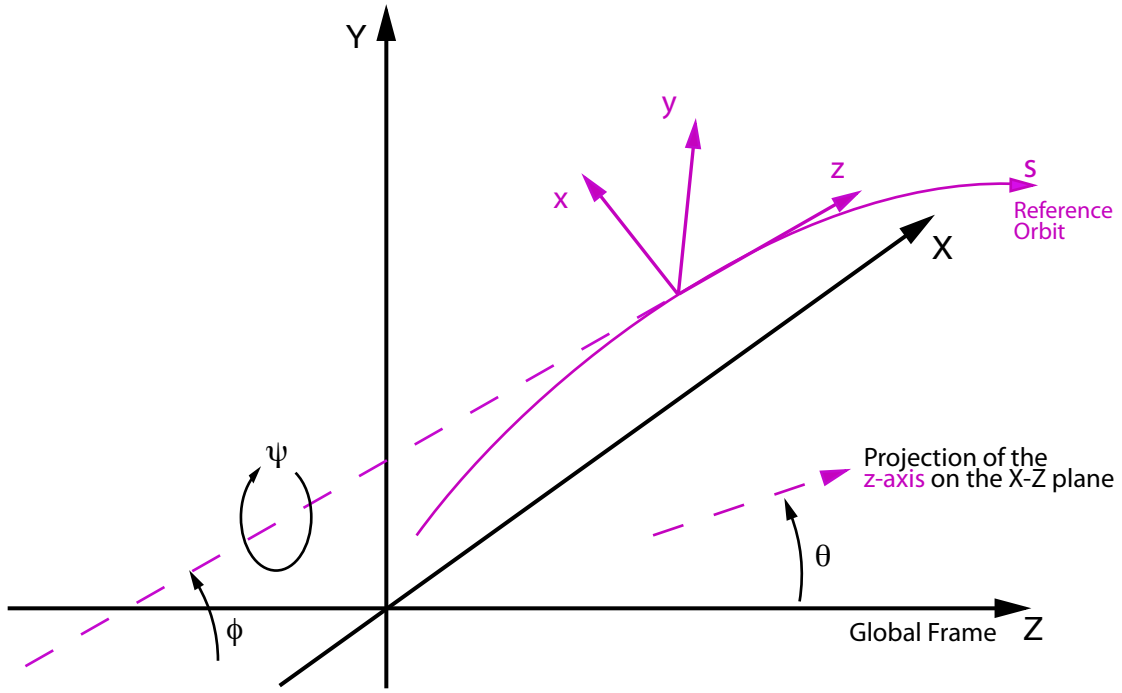


Figure 14.6: The local (reference) coordinate system (purple), which is a function of s along the reference orbit, is described in the global coordinate system (black) by a position $(X(s), Y(s), Z(s))$ and and by angles $\theta(s)$, $\phi(s)$, and $\psi(s)$.

$\psi(s)$ **Roll angle:** Angle of the x–axis with respect to the line formed by the intersection of the (X, Z) plane with the (x, y) plane. A positive ψ forms a right–handed screw with the z–axis.

By default, at $s = 0$, the reference orbit's origin coincides with the (X, Y, Z) origin and the x, y, and z axes correspond to the X, Y, and Z axes respectively. θ decreases as one follows the reference orbit when going through a horizontal bend with a positive bending angle. This corresponds to x pointing radially outward. Without any vertical bends, the Y and y axes will coincide, and ϕ and ψ will both be zero. The `beginning` statement (§8.4) in a lattice file can be use to override these defaults.

Following *MAD*, the global position of an element is characterized by a vector $\mathbf{V}$

$$\mathbf{V} = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \tag{14.1}$$

The orientation of an element is described by a unitary rotation matrix $\mathbf{W}$. The column vectors of $\mathbf{W}$ are the unit vectors spanning the local coordinate axes in the order (x, y, z). $\mathbf{W}$ can be expressed in terms of the orientation angles θ, ϕ, and ψ via the formula

$$\mathbf{W} = \mathbf{R}_y(\theta)\, \mathbf{R}_{-x}(\phi)\, \mathbf{R}_z(\psi) \tag{14.2}$$

where

$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}, \quad \mathbf{R}_{-x}(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{pmatrix}, \quad \mathbf{R}_z(\psi) = \begin{pmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{14.3}$$

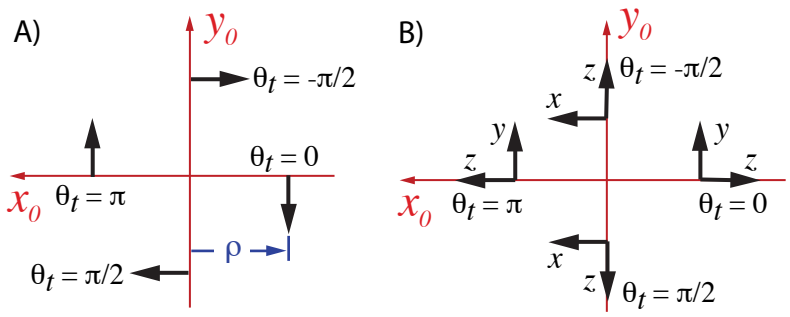


Figure 14.7: A) Rotation axes (bold arrows) for four different `ref_tilt` angles of $\theta_t = 0, \pm\pi/2$, and π. (x_0, y_0, z_0) are the local coordinates at the entrance end of the bend with the z_0 axis being directed into the page. Any rotation axis will be displaced by a distance of the bend radius `rho` from the origin. B) The (x, y, z) coordinates at the exit end of the bend for the same four `ref_tilt` angles. In this case the bend angle is taken to be $\pi/2$.

Notice that $\mathbf{R}_{-x}(\phi)$, for positive ϕ, represents a rotation around the negative x-axis.

An alternative representation of the $\mathbf{W}$ matrix (or any other rotation matrix) is to specify the axis $\mathbf{u}$ (normalized to 1) and angle of rotation β

$$\mathbf{W} = \begin{pmatrix} \cos\beta + u_x^2\,(1-\cos\beta) & u_x\,u_y\,(1-\cos\beta) - u_z\sin\beta & u_x\,u_z\,(1-\cos\beta) + u_y\sin\beta \\ u_y\,u_x\,(1-\cos\beta) + u_z\sin\beta & \cos\beta + u_y^2\,(1-\cos\beta) & u_y\,u_z\,(1-\cos\beta) - u_x\sin\beta \\ u_z\,u_x\,(1-\cos\beta) - u_y\sin\beta & u_z\,u_y\,(1-\cos\beta) + u_x\sin\beta & \cos\beta + u_z^2\,(1-\cos\beta) \end{pmatrix} \tag{14.4}$$

14.2.1 Lattice Element Positioning

Bmad, again following *MAD*, computes $\mathbf{V}$ and $\mathbf{W}$ by starting at the first element of the lattice and iteratively using the equations

$$\mathbf{V}_i = \mathbf{W}_{i-1}\,\mathbf{L}_i + \mathbf{V}_{i-1}, \tag{14.5}$$

$$\mathbf{W}_i = \mathbf{W}_{i-1}\,\mathbf{S}_i \tag{14.6}$$

$\mathbf{L}_i$ is the displacement vector for the i^{th} element and matrix $\mathbf{S}_i$ is the rotation of the local reference system of the exit end with respect to the entrance end. For clarity, the subscript i in the equations below will be dripped. For all elements whose reference orbit through them is a straight line, the corresponding $\mathbf{L}$ and $\mathbf{S}$ are

$$\mathbf{L} = \begin{pmatrix} 0 \\ 0 \\ L \end{pmatrix}, \quad \mathbf{S} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{14.7}$$

Where L is the length of the element.

For a `bend`, the axis of rotation is dependent upon the bend's `ref_tilt` angle (§4.6) as shown in Fig. 14.7A. The axis of rotation points in the negative y_0 direction for `ref_tilt` = 0 and is offset by the bend radius `rho`. Here (x_0, y_0, z_0) are the local coordinates at the entrance end of the bend with the z_0 axis being directed into the page in the figure. For a non-zero `ref_tilt`, the rotation axis is itself rotated about the z_0 axis by the value of `ref_tilt`. Fig. 14.7B shows the exit coordinates for four different values of `ref_tilt` and for a bend angle `angle` of $\pi/2$. Notice that for a bend in the horizontal $X - Z$ plane, a positive bend `angle` will result in a decreasing azimuth angle θ.

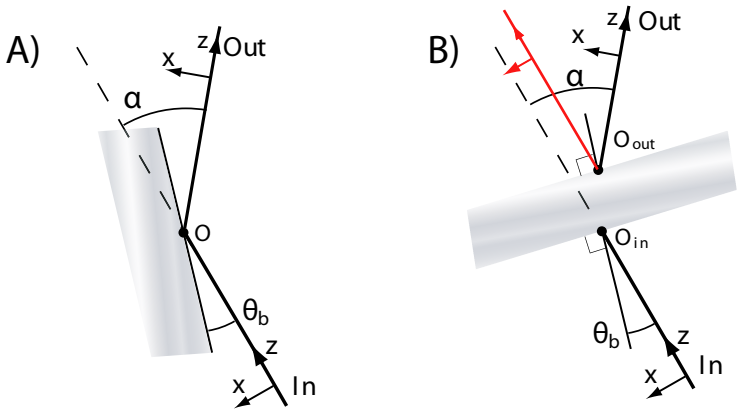


Figure 14.8: Mirror and crystal geometry. The geometry shown here is appropriate for a `ref_tilt` angle of $\theta_t = 0$. θ_g is the bend angle of the incoming (entrance) ray, and α_b is the total bend angle of the reference trajectory. A) Geometry for a mirror or a Bragg crystal. Point $\mathcal{O}$ is the origin of both the local coordinates just before and just after the reflection/diffraction. B) Geometry for a Laue crystal. Point $\mathcal{O}_{out}$ is the origin of the coordinates just after diffraction is displaced from the origin $\mathcal{O}_{in}$ just before diffraction due to the finite thickness of the crystal. here the bend angles are measured with respect to the line that is in the plane of the entrance and exit coordinates and perpendicular to the surface. For Laue diffraction, the user has the option of using the undiffracted beam (shown in red) as the reference trajectory.

For a bend, $\mathbf{S}$ is given using Eq. (14.4) with

$$\begin{aligned} \mathbf{u} &= (-\sin\theta_t, -\cos\theta_t, 0) \\ \beta &= \alpha_b \end{aligned} \tag{14.8}$$

where θ_t is the `ref_tilt` angle. The $\mathbf{L}$ vector for a `bend` is given by

$$\mathbf{L} = \mathbf{R}_z(\theta_t)\,\widetilde{\mathbf{L}}, \quad \widetilde{\mathbf{L}} = \begin{pmatrix} \rho(\cos\alpha_b - 1) \\ 0 \\ \rho\,\sin\alpha_b \end{pmatrix} \tag{14.9}$$

where α_b is the bend `angle` (§3.5) and ρ being the bend radius (`rho`). Notice that since $\mathbf{u}$ is perpendicular to z, the curvilinear reference coordinate system has no "torsion". That is, it is a Frenet-Serret coordinate system.

Note: An alternative equation for $\mathbf{S}$ for a bend is

$$\mathbf{S} = \mathbf{R}_z(\theta_t)\,\mathbf{R}_y(-\alpha_b)\,\mathbf{R}_z(-\theta_t) \tag{14.10}$$

The bend transformation above is so constructed that the transformation is equivalent to rotating the local coordinate system around an axis that is perpendicular to the plane of the bend. This rotation axis is invariant under the bend transformation. For example, for $\theta_t = 0$ (or π) the y-axis is the rotation axis and the y-axis of the local coordinates before the bend will be parallel to the y-axis of the local coordinates after the bend as shown in Fig. 14.7. That is, a lattice with only bends with $\theta_t = 0$ or π will lie in the horizontal plane (this assuming that the y-axis starts out pointing along the Y-axis as it does by default). For $\theta_t = \pm\pi/2$, the bend axis is the x-axis. A value of $\theta_t = +\pi/2$ represents a downward pointing bend.

14.2.2 Position Transformation When Transforming Coordinates

A point $\mathbf{Q}_g = (X, Y, Z)$ defined in the global coordinate system, when expressed in the coordinate system defined by $(\mathbf{V}, \mathbf{W})$ is

$$\mathbf{Q}_{VW} = \mathbf{W}^{-1} \left(\mathbf{Q}_g - \mathbf{V} \right) \tag{14.11}$$

This is essentially the inverse of Eq. (14.5). That is, vectors propagate inversely to the propagation of the coordinate system.

Using Eq. (14.11) with Eqs. (14.5), and (14.6), the transformation of a particle's position $\mathbf{q} = (x, y, z)$ and momentum $\mathbf{P} = (P_x, P_y, P_z)$ when the coordinate frame is transformed from frame $(\mathbf{V}_{i-1}, \mathbf{W}_{i-1})$ to frame $(\mathbf{V}_i, \mathbf{W}_i)$ is

$$\mathbf{q}_i = \mathbf{S}_i^{-1} \left(\mathbf{q}_{i-1} - \mathbf{L}_i \right), \tag{14.12}$$

$$\mathbf{P}_i = \mathbf{S}_i^{-1} \mathbf{P}_{i-1} \tag{14.13}$$

Notice that since $\mathbf{S}$ (and $\mathbf{W}$) is the product of orthogonal rotation matrices, $\mathbf{S}$ is itself orthogonal and its inverse is just the transpose

$$\mathbf{S}^{-1} = \mathbf{S}^T \tag{14.14}$$

14.2.3 Crystal and Mirror Element Coordinate Transformation

A `crystal` element (§3.31) diffracts photons and a `mirror` element (§3.31) reflects them. For a crystal setup for Bragg diffraction, and for a mirror, the reference orbit is modeled as a zero length bend with $\widetilde{\mathbf{L}} = (0, 0, 0)$, as shown in Fig. 14.8A. Shown in the figure is the geometry appropriate for a `ref_tilt` angle of $\theta_t = 0$ (the rotation axis is here the y-axis). Since the mirror or crystal element is modeled to be of zero length, the origin points (marked $\mathcal{O}$ in the figure) of the entrance and exit local coordinates are the same. For Laue diffraction, the only difference is that $\widetilde{\mathbf{L}}$ is non-zero due to the finite thickness of the crystal as shown in Fig. 14.8B. This results in a separation between the entrance coordinate origin $\mathcal{O}_{in}$ and the exit coordinate origin $\mathcal{O}_{out}$.

In all cases, the total bending angle is

$$\begin{aligned}
\alpha_b &= \text{bragg_angle_in} + \text{bragg_angle_out} && \text{! Crystal, graze_angle_in} = 0 \\
\alpha_b &= \text{graze_angle_in} + \text{graze_angle_out} && \text{! Crystal, graze_angle_in} \neq 0 \\
\alpha_b &= 2\,\text{graze_angle} && \text{! Mirror}
\end{aligned} \tag{14.15}$$

With a mirror or Bragg diffraction, the bend angles are measured with respect to the surface plane. With Laue diffraction the bend angles are measured with respect to the line in the bend plane perpendicular to the surface.

For Laue diffraction, the user has the option of using the undiffracted beam (shown in red) as the reference trajectory.

The orientation of the exit coordinates (the local coordinates after the reflection) are only affected by the element's `ref_tilt` and bend angle parameters and is independent of all other parameters such as the radius of curvature of the surface, etc. The local z-axis of the entrance coordinates along with the z-axis of the exit coordinates define a plane which is called the element's `bend plane`. For a mirror, the graze angle is a parameter supplied by the user. For a crystal, the Bragg angles are calculated so that the reference trajectory is in the middle of the Darwin curve. Calculation of the Bragg angles for a crystal is given in Section §23.4.1.

14.2.4 Patch and Floor_Shift Elements Entrance to Exit Transformation

For `patch` (§3.37) and `floor_shift` (§3.18) elements, the shift in the exit end reference coordinates is given by Eqs. (14.5) and (14.6) with

$$\mathbf{L} = \begin{pmatrix} \text{x\_offset} \\ \text{y\_offset} \\ \text{z\_offset} \end{pmatrix}$$
$$\mathbf{S} = \mathbf{R}_y(\text{x\_pitch})\, \mathbf{R}_{-x}(\text{y\_pitch})\, \mathbf{R}_z(\text{tilt}) \tag{14.16}$$

The difference here between `patch` and `floor_shift` elements is that, with a `patch` element, the shift is relative to the exit end of the previous element while, for a `floor_shift` element, the shift is relative to the reference point on the origin element specified by the `origin_ele` parameter of the `floor_shift`.

14.2.5 Fiducial and Girder Elments Origin Shift Transformation

For `fiducial` and `girder` elements, the alignment of the reference coordinates with respect to "`origin`" coordinates is analogous to Eqs. (14.16). Explicitly:

$$\mathbf{L} = \begin{pmatrix} \text{dx\_origin} \\ \text{dy\_origin} \\ \text{dz\_origin} \end{pmatrix}$$
$$\mathbf{S} = \mathbf{R}_y(\text{dtheta\_origin})\, \mathbf{R}_{-x}(\text{dphi\_origin})\, \mathbf{R}_z(\text{dpsi\_origin}) \tag{14.17}$$

14.2.6 Reflection Patch

A `Patch` (or a series of patches) that reflects the direction of the z-axis is called a `reflection patch`. By "reflected direction" it is meant that the dot product $\mathbf{z}_1 \cdot \mathbf{z}_2$ is negative where $\mathbf{z}_1$ is the z-axis vector at the `entrance` face and $\mathbf{z}_2$ is the z-axis vector at the `exit` face. This condition is equivalent to the condition that the associated $\mathbf{S}$ matrix (see Eq. (14.16)) satisfy:

$$S(3,3) < 0 \tag{14.18}$$

Using Eq. (14.16) gives, after some simple algebra, this condition is equivalent to

$$\cos(\text{x\_pitch})\, \cos(\text{y\_pitch}) < 0 \tag{14.19}$$

When there are a series of patches, The transformations of all the patches are concatenated together to form an effective $\mathbf{S}$ which can then be used with Eq. (14.18).

14.3 Transformation Between Laboratory and Element Body Coordinates

The `element body` coordinates are the coordinate system attached to an element. Without any misalignments, where "`misalignments`" are here defined to be any offset, pitch or tilt (§4.6), the `laboratory` coordinates (§14.1.1) and `element body` coordinates are the same. With misalignments, the transformation between `laboratory` and `element body` coordinates depends upon whether the local coordinate system is straight (§14.3.1) or bent (§14.3.2).

14.3.1 Straight Element Misalignment Transformation

For straight line elements, given a laboratory coordinate frame Λ_s with origin a distance s from the beginning of the element, misalignments will shift the coordinates to a new reference frame denoted E_s. Since misalignments are defined with respect to the middle of the element, the transformation between Λ_s and E_s is a three step process:

$$\Lambda_s \longrightarrow \Lambda_{\rm mid} \longrightarrow E_{\rm mid} \longrightarrow E_s \tag{14.20}$$

where $\Lambda_{\rm mid}$ and $E_{\rm mid}$ are the laboratory and element reference frames at the center of the element.

The first and last transformations from Λ_s to $\Lambda_{\rm mid}$ and from $E_{\rm mid}$ to E_s use Eqs. (14.5), (14.6), and (14.7) with the replacement $L \to L/2 - s$ for the first transformation and $L \to s - L/2$ for the third transformation. The middle transformation, by definition of the offset, pitch and tilt parameters is

$$\begin{aligned} \mathbf{L} &= \begin{pmatrix} \text{x\_offset} \\ \text{y\_offset} \\ \text{z\_offset} \end{pmatrix} \\ \mathbf{S} &= \mathbf{R}_y(\text{x\_pitch})\ \mathbf{R}_{-x}(\text{y\_pitch})\ \mathbf{R}_z(\text{tilt}) \end{aligned} \tag{14.21}$$

Notice that with this definition of how elements are misaligned, the position of a misaligned element depends only on the offsets, and is independent of the pitches and tilt.

14.3.2 Bend Element Misalignment Transformation

For `rbend` and `sbend` elements there is no `tilt` attribute. Rather, there is the `roll` attribute and a `ref_tilt` attribute. The latter affects both the reference orbit and the bend position (§4.6.2). Furthermore, `ref_tilt` is calculated with respect to the coordinates at the beginning of the bend while, like straight elements, `roll`, offsets, and pitches are calculated with respect to the center of the bend. The different reference frame used for `ref_tilt` versus everything else means that five transformations are needed to get from the laboratory frame to the element body frame (see Eq. (14.20)). Symbolically:

$$\Lambda_s \longrightarrow \Lambda_{\rm mid} \longrightarrow \Omega_{\rm mid} \longrightarrow \Omega_0 \longrightarrow E_0 \longrightarrow E_s \tag{14.22}$$

The first transformation, Λ_s to $\Lambda_{\rm mid}$, from laboratory coordinates at a distance s from the beginning of the element to laboratory coordinates at the center the bend is a rotation around the center of curvative of the bend and is given by Eqs. (14.5) and (14.6) with Eqs. (14.8) and (14.9) with the substitution $\alpha_b \to (L/2 - s)/\rho$.

The second transformation $\Lambda_{\rm mid}$ to $\Omega_{\rm mid}$ at the center of the element adds in the misalignments (Note that the coordinate frame $\Omega_{\rm mid}$ is neither a laboratory frame or an element frame so hence the use of a different symbol Ω). Explicitly, the $\Lambda_{\rm mid} \longrightarrow \Omega_{\rm mid}$ transformation is

$$\begin{aligned} \mathbf{L} &= \mathbf{L}_{\rm off} + [\mathbf{R}_z(\text{roll}) - \mathbf{1}]\ \mathbf{R}_z(\theta_t)\ \mathbf{R}_y(\alpha_b/2)\ \mathbf{L}_c \\ \mathbf{S} &= \mathbf{R}_y(\text{x\_pitch})\ \mathbf{R}_{-x}(\text{y\_pitch})\ \mathbf{R}_z(\text{roll}) \end{aligned} \tag{14.23}$$

where

$$\mathbf{L}_c = \begin{pmatrix} \rho(\cos(\alpha_b/2) - 1) \\ 0 \\ \rho\ \sin(\alpha_b/2) \end{pmatrix}, \qquad \mathbf{L}_{\rm off} = \begin{pmatrix} \text{x\_offset} \\ \text{y\_offset} \\ \text{z\_offset} \end{pmatrix} \tag{14.24}$$

The reason why $\mathbf{L}$ has a different form from straight line elements is due to the fact that the axis of rotation for a `roll` is displaced from the z-axis of the coordinate system at the center of the bend (see Fig. 4.3).

The third transformation from $\Omega_{\rm mid}$ to Ω_0 is like the first transformation and rotates from the center of the bend to the beginning. Again Eqs. (14.8) and (14.9) are used with the substitution $\alpha_b \to -L/2\rho$.

The fourth transformation Ω_0 to E_0 tilts the referece frame by an amount `ref_tilt`:

$$\mathbf{L} = 0, \quad \mathbf{S} = \mathbf{R}_z(\theta_t) \tag{14.25}$$

The fifth and final transformation, E_0 to E_s, like the first and third, rotates around the center of the bend but in this case, since we are dealing with element coordinates, the `ref_tilt` is ignored. That is, Eqs. (14.8) and (14.9) are used with the substitutions $\theta_t \to 0$ and $\alpha_b \to L/\rho$.

Notice that with this definition of how elements are misaligned, the position of a misaligned element depends only on the offsets, and is independent of the pitches and tilt. Also the orientation of an element depends only on the pitches and tilt (or roll and ref_tilt for a bend), and is independent of the offsets.

14.4 Phase Space Coordinates

14.4.1 Reference Particle, Reference Energy, and Reference Time

The `reference energy` and `reference time` are needed in evaluating the phase space coordinates of charged particles (§14.4.2).

All lattice elements, except for controller elements, have an associated `reference energy` energy. The reference energy at the start of a lattice's `root branch` (§1.2) is set in the lattice file by setting the reference momentum (`p0c`) or total energy (`E_tot`) using a `parameter` (§8.1) or `beginning` (§8.4) statement. For other branches, the energy at the start of the branch is set using the appropriate line parameter (§8.4) statement.

For most elements, the reference energy is the same as the reference energy of the preceeding element. The following elements are exceptions:

```
custom
em_field
hybrid
lcavity
patch
```

The reference energy of these elements is determined by tracking a particle (the "`reference particle`") through the element with the particle starting on the reference orbit and whose energy is equal to the reference energy. The energy of the particle at the downstream end is the reference energy of the element.

Besides the reference energy, lattice elements have an associated `reference time` which is computed, for most elements, by the time-of-flight of the `reference particle` assuming that the reference particle is following the reference orbit. Exceptions are `wiggler` elements which uses the time-of-flight of the actual undulating trajectory. [Actually what is used in the computation of the z phase space coordinate (Eq. (14.28)) is the sum of reference time deltas of the elements that a particle has passed through. It is not possible to assign a unique reference time to an element when particles are recirculating through elements as in a storage ring.]

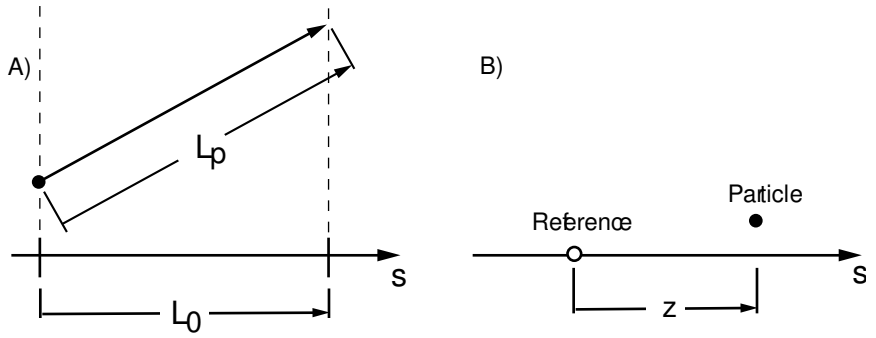


Figure 14.9: Interpreting phase space z at constant velocity: A) The change in z going through an element of length L_0 is $L_0 - L_p$. B) At constant time, z is the longitudinal distance between the reference particle and the particle.

14.4.2 Charged Particle Phase Space Coordinates

For charged particles (more correctly, for everything but photons (§14.4.4)), *Bmad* uses the canonical phase space coordinates

$$\mathbf{r}(s) = (x, p_x, y, p_y, z, p_z) \tag{14.26}$$

The longitudinal position s is the independent variable instead of the time. x and y, are the transverse coordinates of the particle as shown in Fig. 14.2. Note that x and y are independent of the position of the reference particle.

The phase space momenta p_x and p_y are normalized by the reference (sometimes called the design) momentum P_0

$$\begin{aligned} p_x &= \frac{P_x}{P_0} \\ p_y &= \frac{P_y}{P_0} \end{aligned} \tag{14.27}$$

where P_x and P_y are respectively the x and y momentums.

The phase space z coordinate is

$$\begin{aligned} z(s) &= -\beta(s)\, c\, (t(s) - t_0(s)) \\ &\equiv -\beta(s)\, c\, \Delta t(s) \end{aligned} \tag{14.28}$$

$t(s)$ is the time at which the particle is at position s, $t_0(s)$ is the time at which the reference particle is at position s, and β is v/c with v being the particle velocity (and not the reference velocity). The reference particle is, by definition, "synchronized" with elements whose fields are oscillating and therefore the actual fields a particle will see when traveling through such an element will depend upon the particle's phase space z. For example, the energy change of a particle traveling through an `lcavity` (§3.26) or `rfcavity` (§3.40) element is z dependent. Exception: With absolute time tracking (§22.1) fields are tied to the absolute time and not z.

If the particle's velocity is constant, and is the same as the velocity of the reference particle (for example, at high energy where $\beta = 1$ for all particles), then $\beta\, c\, t$ is just the path length. In this case, the change in z going through an element is

$$\Delta z = L_0 - L_p \tag{14.29}$$

where, as shown in Fig. 14.9A, L_0 is the path length of the reference particle (which is just the length of the element) and L_p is the path length of the particle in traversing the element. Another way of interpreting phase space z is that, at constant β, and constant time, z is the longitudinal distance between the particle and the reference particle as shown in Fig. 14.9B. with positive z indicating that the particle is ahead of the reference particle.

Do not confuse the phase space z with the z that is the particle's longitudinal coordinate in the local reference frame as shown in Fig. 14.2. By construction, this latter z is always zero.

Notice that if a particle gets an instantaneous longitudinal kick so that β is discontinuous then, from Eq. (14.28), phase space z is discontinuous even though the particle itself does not move in space. In general, from Eq. (14.28), The value of z for a particle at s_2 is related to the value of z for the particle at s_1 by

$$z_2 = \frac{\beta_2}{\beta_1} z_1 - \beta_2 c (\Delta t_2 - \Delta t_1) \tag{14.30}$$

$\Delta t_2 - \Delta t_1$ can be interpreted as the difference in transit time, between the particle and the reference particle, in going from s_1 to s_2.

The longitudinal phase space momentum p_z is given by

$$p_z = \frac{\Delta P}{P_0} \equiv \frac{P - P_0}{P_0} \tag{14.31}$$

where P is the momentum of the particle. For ultra–relativistic particles p_z can be approximated by

$$p_z = \frac{\Delta E}{E_0} \tag{14.32}$$

where E_0 is the reference energy (energy here always refers to the total energy) and $\Delta E = E - E_0$ is the deviation of the particle's energy from the reference energy. For an `Lcavity` element (§3.26) the reference momentum is *not* constant so the tracking for an `Lcavity` is not canonical.

MAD uses a different coordinate system where (z, p_z) is replaced by $(-c\Delta t, p_t)$ where $p_t \equiv \Delta E / P_0 c$. For highly relativistic particles the two coordinate systems are identical.

`Bmad_standard` (§5) tracking and transfer matrix calculations use the small angle (paraxial) approximation where it is assumed that $p_x, p_y \ll 1$. With this approximation, the relationship, between the phase space momenta and the slopes $x' \equiv dx/ds$ and $y' \equiv dy/ds$ is

$$x' \approx \frac{p_x}{1 + p_z}(1 + gx) \tag{14.33}$$

$$y' \approx \frac{p_y}{1 + p_z}(1 + gx) \tag{14.34}$$

$g = 1/\rho$ is the curvature function with ρ being the radius of curvature of the reference orbit and it has been assumed that the bending is in the x–z plane.

With the paraxial approximation, and in the relativistic limit, the change in z with position is

$$\frac{dz}{ds} = -g\,x - \frac{1}{2}(x'^2 + y'^2) \tag{14.35}$$

This shows that in a linac, without any bends, the z of a particle always decreases.

A particle can also have a spin. The spin is characterized by the spinor $\Psi = (\psi_1, \psi_2)^T$ where $\psi_{1,2}$ are complex numbers (§19.1).

For those programmers using the PTC software package directly (ignore this if you don't know what is being talked about here), Étienne Forest uses, by default, a different phase space coordinates which

uses $(t, \Delta E/P_0)$ instead of *Bmad*'s $(z, \Delta P/P_0)$. See Chapter §34 for more details. Which phase space coordinates are "better" is a matter of taste. In general, the equations of motion in a magnetic field are simpler with $(z, \Delta P/P_0)$ while the equations of motion in an electric field are simple with $(t, \Delta E/P_0)$.

14.4.3 Time-based Phase Space Coordinates

Some specialized routines (for example, time Runge Kutta tracking) use the time t as the independent variable for charged particle tracking. This is useful when particles can reverse direction since the normal z based tracking cannot handle this. Direction reversal can happen, for example, with low energy "dark current" electrons that are generated at the walls of the vacuum chamber.

When the tracking is time based the phase space coordinates are:

$$(x, c\,p_x, y, c\,p_y, z, c\,p_s) \tag{14.36}$$

The positions x, y, and z are the same as with phase spac coordinates (§14.4.2). The momenta are defined as

$$\begin{aligned} cp_x &\equiv mc^2\gamma\beta_x \\ cp_y &\equiv mc^2\gamma\beta_y \\ cp_s &\equiv mc^2\gamma\beta_s, \end{aligned} \tag{14.37}$$

and internally are stored in units of eV.

14.4.4 Photon Phase Space Coordinates

The phase space coordinates discussed above implicitly assume that particles are traveling longitudinally in only one direction. That is, the sign of the s component of the momentum cannot be determined from the phase space coordinates. This is generally fine for tracking high energy beams of charged particles but for photon tracking this would oftentimes be problematical. For photons, therefore, a different phase space is used:

$$(x, \beta_x, y, \beta_y, z, \beta_z) \tag{14.38}$$

Here $(\beta_x, \beta_y, \beta_z)$ is the normalized photon velocity with

$$\beta_x^2 + \beta_y^2 + \beta_z^2 = 1 \tag{14.39}$$

and (x, y, z) are the reference orbit coordinates with z being the distance from the start of the lattice element the photon is in.

In *Bmad*, the information associated with a photon include its phase space coordinates and time along with the photon energy and four parameters E_x, ϕ_x, and E_y, ϕ_y specifying the intensity and phase of the field along the x and y axes transverse to the direction of propagation. the field in the vicinity of the photon is

$$\begin{aligned} E_x(\mathbf{r}, t) &\sim E_x\, e^{i(k\,(z-z_0)-\omega\,(t-t_{ref})+\phi_x)} \\ E_y(\mathbf{r}, t) &\sim E_y\, e^{i(k\,(z-z_0)-\omega\,(t-t_{ref})+\phi_y)} \end{aligned} \tag{14.40}$$

where z_0 is the photon z position and and t_ref is the reference time.

The normalization between field and intensity is dependent upon the particular parameters of any given simulation and so must be determined by the program using *Bmad*.

Chapter 15

Electromagnetic Fields

15.1 Magnetostatic Multipole Fields

Start with the assumption that the local magnetic field has no longitudinal component (obviously this assumption does not work with, say, a solenoid). Following *MAD*, ignoring skew fields for the moment, the vertical magnetic field along the $y = 0$ axis is expanded in a Taylor series

$$B_y(x,0) = \sum_n B_n \frac{x^n}{n!} \tag{15.1}$$

Assuming that the reference orbit is locally straight (there are correction terms if the Reference Orbit is curved), the field is

$$B_x = \quad B_1 y + \quad B_2\, xy \quad + \frac{1}{6}B_3(3x^2y - y^3) \; + \; \ldots \tag{15.2}$$

$$B_y = B_0 + B_1 x + \frac{1}{2}B_2(x^2 - y^2) + \frac{1}{6}B_3(x^3 - 3xy^2) \; + \; \ldots \tag{15.3}$$

For some fields, the normalized integrated multipole $K_n L$ can be used when specifying magnetic multipole components

$$K_n L \equiv \frac{q\, L\, B_n}{P_0} \tag{15.4}$$

where q is the charge of the reference particle (in units of the elementary charge), L is the element length, and P_0 is the reference momentum (in units of eV/c). Note that P_0/q is sometimes written as $B\rho$. This is just an old notation where ρ is the bending radius of a particle with the reference energy in a field of strength B. Notice that P_0 is the local reference momentum at the element which may not be the same as the reference energy at the beginning of the lattice if there are `lcavity` elements (§3.26) present.

The kicks Δp_x and Δp_y that a particle experiences going through a multipole field is

$$\Delta p_x = \frac{-q\, L\, B_y}{P_0} \tag{15.5}$$

$$= -K_0 L - K_1 L\, x \; + \; \frac{1}{2}K_2 L(y^2 - x^2) + \frac{1}{6}K_3 L(3xy^2 - x^3) \; + \; \ldots$$

$$\Delta p_y = \frac{q\, L\, B_x}{P_0} \tag{15.6}$$

$$= \quad K_1 L\, y + \quad K_2 L\, xy \quad + \frac{1}{6}K_3 L(3x^2y - y^3) \; + \; \ldots$$

A positive $K_1 L$ quadrupole component gives horizontal focusing and vertical defocusing. The general form is

$$\Delta p_x = \sum_{n=0}^{\infty} \frac{K_n L}{n!} \sum_{m=0}^{\lfloor \frac{n}{2} \rfloor} \binom{n}{2m} (-1)^{m+1} x^{n-2m} y^{2m} \tag{15.7}$$

$$\Delta p_y = \sum_{n=0}^{\infty} \frac{K_n L}{n!} \sum_{m=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n}{2m+1} (-1)^{m} x^{n-2m-1} y^{2m+1} \tag{15.8}$$

where $\lfloor \alpha \rfloor$ means round down to the integer equal to or less than α and $\binom{a}{b}$ ("a choose b") denotes a binomial coefficient.

So far only the normal components of the field have been considered. If the fields associated with a particular B_n multipole component are rotated in the (x, y) plane by an angle θ_n, the magnetic field at a point (x, y) can be expressed in complex notation as

$$B_y(x, y) + iB_x(x, y) = \frac{1}{n!} B_n e^{-i(n+1)\theta_n} e^{in\theta} r^n \tag{15.9}$$

where (r, θ) are the polar coordinates of the point (x, y).

Note that, for compatibility with MAD, the $K0L$ component of a `Multipole` element rotates the reference orbit essentially acting as a zero length bend. This is not true for multipoles of any other type of element.

Instead of using magnitude K_n and rotation angle θ_n, Another representation is using normal $\widetilde{K}_n$ and skew $\widetilde{S}_n$. The conversion between the two are

$$\begin{aligned} \widetilde{K}_n &= K_n \cos((n+1)\theta_n) \\ \widetilde{S}_n &= K_n \sin((n+1)\theta_n) \end{aligned} \tag{15.10}$$

Another representation of the magnetic field used by *Bmad* divides the fields into normal b_n and skew a_n components. In terms of these components the magnetic field for the n^{th} order multipole is

$$\frac{q\,L}{P_0} (B_y + iB_x) = (b_n + ia_n) (x + iy)^n \tag{15.11}$$

The a_n, b_n representation of multipole fields can be used in elements such as quadrupoles, sextupoles, etc. to allow "error" fields to be represented. The conversion between (a_n, b_n) and $(K_n L, \theta_n)$ is

$$b_n + ia_n = \frac{1}{n!} K_n L\, e^{-i(n+1)\theta_n} \tag{15.12}$$

or

$$K_n L = n! \sqrt{a_n^2 + b_n^2} \tag{15.13}$$

$$\tan[(n+1)\theta_n] = \frac{-a_n}{b_n} \tag{15.14}$$

To convert a normal magnet (a magnet with no skew component) into a skew magnet (a magnet with no normal component) the magnet should be rotated about its longitudinal axis with a rotation angle of

$$(n+1)\theta_n = \frac{\pi}{2} \tag{15.15}$$

For example, a normal quadrupole rotated by 45° becomes a skew quadrupole.

The multipole fields can be "`reference energy`" scaled and/or "`element strength`" scaled. Scaling here means that the a_n and b_n values used in tracking are scaled from the input values given in the lattice file.

`Reference energy` scaling is applied if the `field_master` attribute (§4.2) is True for an element so that the multipole values specified in the lattice file are not reference energy normalized

$$\left[a_n, b_n\right] \longrightarrow \left[a_n, b_n\right] \cdot \frac{q}{P_0} \tag{15.16}$$

`Element strength` scaling is applied when the multipoles are associated with a non `AB_Multipole` element and if the `scale_multipoles` attribute (§4.14) is `True`. This scaling uses a measurement radius r_0 and a scale factor F:

$$\left[a_n, b_n\right] \longrightarrow \left[a_n, b_n\right] \cdot F \cdot \frac{r_0^{n_{\rm ref}}}{r_0^n} \tag{15.17}$$

r_0 is set by the `r0_mag` attribute of an element. F and $n_{\rm ref}$ are set automatically depending upon the type of element as shown in Table 15.1. The γ_p term is

Element	F	$n_{\rm ref}$
`Elseparator`	$\sqrt{\texttt{Hkick}^2 + \texttt{Vkick}^2}$	0
`Hkicker`	Kick	0
`Kicker`,`AC_Kicker`	$\sqrt{\texttt{Hkick}^2 + \texttt{Vkick}^2}$	0
`Rbend`	G * L	0
`Sbend`	G * L	0
`Vkicker`	Kick	0
`Wiggler`	$\frac{2\,c\,\texttt{L\_pole}\,B_{max}}{\pi\,\texttt{p0c}}$	0
`Quadrupole`	K1 * L	1
`Sol_Quad`	K1 * L	1
`Solenoid`	KS * L	1
`Sextupole`	K2 * L	2
`Octupole`	K3 * L	3

Table 15.1: F and $n_{\rm ref}$ for various elements.

15.2 Electrostatic Multipole Fields

Except for the `elseparator` element, *Bmad* specifies DC electric fields using normal b_{en} and skew a_{en} components (§4.14. The potential ϕ_n for the n^{th} order multipole is

$$\phi_n = -\,\mathrm{Re}\left[\frac{b_{en} - i a_{en}}{n+1}\,\frac{(x+iy)^{n+1}}{r_0^n}\right] \tag{15.18}$$

where r_0 is a "measurement radius" set by the `r0_elec` attribute of an element (§4.14).

The electric field for the n^{th} order multipole is

$$E_x - i E_y = (b_{en} - i a_{en})\,\frac{(x+iy)^n}{r_0^n} \tag{15.19}$$

Notice that the magnetic multipole components a_n and b_n are normalized by the element length, reference charge, and reference momentum (Eq. (15.11)) while their electric counterparts are not.

Using the paraxial approximation, The kick given a particle due to the electric field is

$$\frac{dp_x}{ds} = \frac{q\,E_x}{\beta\,P_0\,c}, \qquad \frac{dp_y}{ds} = \frac{q\,E_y}{\beta\,P_0\,c} \tag{15.20}$$

Where β is the normalized velocity.

15.3 Exact Multipole Fields in a Bend

For static magnetic and electric multipole fields in a bend, the spacial dependence of the field is different from multipole fields in an element with a straight geometry as given by Eqs. (15.11) and (15.19). The analysis of the multipole fields in a bend here follows McMillan[McMill75].

In the rest of this section, normalized coordinates $\widetilde{r} = r/\rho$, $\widetilde{x}/ = x/\rho$, and $\widetilde{y} = y/\rho$ will be used where ρ is the bending radius of the reference coordinate system, r is the distance, in the plane of the bend, from the bend center to the observation point, x is the distance in the plane of the from the reference coordinates to the observation point and y is the distance out-of-plane. With this convention $\widetilde{r} = 1 + \widetilde{x}$.

An electric or magnetic multipole can be characterized by a scalar potential ϕ with the field given by $-\nabla\phi$. The potential is a solution to Laplace's equation

$$\frac{1}{\widetilde{r}}\,\frac{\partial}{\partial\,\widetilde{r}}\left(\widetilde{r}\,\frac{\partial\,\phi}{\partial\,\widetilde{r}}\right) + \frac{\partial^2\phi}{\partial\,\widetilde{y}^2} = 0 \tag{15.21}$$

As McMillian shows, it is also possible to calculate the magnetic field by constructing the appropriate vector potential. However, from a practical point of view, it is simpler to use the scalar potential for both the magnetic and electric fields.

Solutions to Laplace's equation can be found in form

$$\phi_n^r = \frac{-1}{1+n}\sum_{p=0}^{\lfloor (n+1)/2\rfloor}\binom{n+1}{2\,p}\,(-1)^p\,F_{n+1-2p}(\widetilde{r})\,\widetilde{y}^{2p} \tag{15.22}$$

and in the form

$$\phi_n^i = \frac{-1}{1+n}\sum_{p=0}^{\lfloor n/2\rfloor}\binom{n+1}{2p+1}\,(-1)^p\,F_{n-2p}(\widetilde{r})\,\widetilde{y}^{2p+1} \tag{15.23}$$

where $\binom{a}{b}$ ("a choose b") denotes a binomial coefficient, n is the order number which can range from 0 to infinity, $\lfloor\alpha\rfloor$ means round down to the integer equal to or less than α. [Notice that here n is related to m in McMillian's paper by $m = n+1$. Also note that the ϕ^r and ϕ^i here have a normalization factor that is different from McMillian.]

In Eq. (15.23) the $F_p(\widetilde{r})$ are related by

$$F_{p+2} = (p+1)\,(p+2)\int_1^{\widetilde{r}}\frac{d\widetilde{r}}{\widetilde{r}}\left[\int_1^{\widetilde{r}} d\widetilde{r}\,\widetilde{r}\,F_p\right] \tag{15.24}$$

with the "boundary condition":

$$\begin{aligned} F_0(\widetilde{r}) &= 1 \\ F_1(\widetilde{r}) &= \ln\,\widetilde{r} \end{aligned} \tag{15.25}$$

This condition ensures that the number of terms in the sums in Eqs. (15.22) and (15.23) are finite. With this condition, all the F_p can be constructed:

$$\begin{aligned}
F_1 &= \ln \widetilde{r} = \widetilde{x} - \frac{1}{2}\widetilde{x}^2 + \frac{1}{3}\widetilde{x}^3 - \dots \\
F_2 &= \frac{1}{2}(\widetilde{r}^2 - 1) - \ln \widetilde{r} = \widetilde{x}^2 - \frac{1}{3}\widetilde{x}^3 + \frac{1}{4}\widetilde{x}^4 - \dots \\
F_3 &= \frac{3}{2}[-(\widetilde{r}^2 - 1) + (\widetilde{r}^2 + 1)\ln \widetilde{r}] = \widetilde{x}^3 - \frac{1}{2}\widetilde{x}^4 + \frac{7}{20}\widetilde{x}^5 - \dots \\
F_4 &= 3[\frac{1}{8}(\widetilde{r}^4 - 1) + \frac{1}{2}(\widetilde{r}^2 - 1) - (\widetilde{r}^2 + \frac{1}{2})\ln \widetilde{r}] = \widetilde{x}^4 - \frac{2}{5}\widetilde{x}^5 + \frac{3}{10}\widetilde{x}^6 - \dots \\
&\text{Etc...}
\end{aligned} \tag{15.26}$$

In terms of implementing these functions on a computer, the exact $\widetilde{r}$-dependent functions are problematical due to round off error near $\widetilde{x} = 0$. For example, Evaluating $F_4(\widetilde{r})$ at $\widetilde{x} = 10^{-4}$ results in a complete loss of accuracy (no significant digits!) when using double precision numbers. In practice, *Bmad* uses a Pade approximant for $\widetilde{x}$ small enough and then switches to the $\widetilde{r}$-dependent formulas for $\widetilde{x}$ away from zero.

For magnetic fields, the "real" ϕ_n^r solutions will correspond to skew fields and the "imaginary" ϕ_n^i solutions will correspond to normal fields

$$\mathbf{B} = -\frac{P_0}{q\,L}\sum_{n=0}^{\infty} \rho^n \left[a_n \widetilde{\nabla}\phi_n^r + b_n \widetilde{\nabla}\phi_n^i\right] \tag{15.27}$$

where the gradient derivatives of $\widetilde{\nabla}$ are with respect to the normalized coordinates. In the limit of infinite bending radius ρ, the above equations converge to the straight line solution given in Eq. (15.11).

For electric fields, the "real" solutions will correspond to normal fields and the "imaginary" solutions are used for skew fields

$$\mathbf{E} = -\sum_{n=0}^{\infty} \rho^n \left[a_{en} \widetilde{\nabla}\phi_n^i + b_{en} \widetilde{\nabla}\phi_n^r\right] \tag{15.28}$$

And this will converge to Eq. (15.19) in the straight line limit.

In the vertical plane, with $\widetilde{x} = 0$, the solutions ϕ_n^r and ϕ_n^i have the same variation in $\widetilde{y}$ as the multipole fields with a straight geometry. For example, the field strength of an $n = 1$ (quadrupole) multipole will be linear in $\widetilde{y}$ for $\widetilde{x} = 0$. However, in the horizontal direction, with $\widetilde{y} = 0$, the multipole field will vary like $dF_2/d\widetilde{x}$ which has terms of all orders in $\widetilde{x}$. In light of this, the solutions ϕ_n^r and ϕ_n^i are called "vertically pure" solutions.

It is possible to construct "horizontally pure" solutions as well. That is, it is possible to construct solutions that in the horizontal plane, with $\widetilde{y} = 0$, behave the same as the corresponding multipole fields with a straight geometry. A straight forward way to do this, for some given multipole of order n, is to construct the horizontally pure solutions, ψ_n^r and ψ_n^i, as linear superpositions of the vertically pure solutions

$$\psi_n^r = \sum_{k=n}^{\infty} C_{nk}\,\phi_k^r, \qquad \psi_n^i = \sum_{k=n}^{\infty} D_{nk}\,\phi_k^i \tag{15.29}$$

with the normalizations $C_{nn} = D_{nn} = 1$. The C_{nk} and D_{nk} are chosen, order by order, so that ψ_n^r and ψ_n^i are horizontally pure. For the real potentials, the C_{nk}, are obtained from a matrix $\mathbf{M}$ where M_{ij} is the coefficient of the $\widetilde{x}^j$ term of $(dF_i/d\widetilde{x})/i$ when F_i is expressed as an expansion in $\widetilde{x}$ (Eq. (15.26)). C_{nk}, $k = 0, \dots, \infty$ are the row vectors of the inverse matrix $\mathbf{M}^{-1}$. For the imaginary potentials, the D_{nk}

are constructed similarly but in this case the rows of $\mathbf{M}$ are the coefficients in $\widetilde{x}$ for the functions F_i. To convert between field strength coefficients, Eqs. (15.27) and (15.28) and Eqs. (15.29) are combined

$$a_n = \sum_{k=n}^{\infty} \frac{1}{\rho^{k-n}} C_{nk}\, \alpha_k, \quad a_{en} = \sum_{k=n}^{\infty} \frac{1}{\rho^{k-n}} D_{nk}\, \alpha_{ek},$$
$$b_n = \sum_{k=n}^{\infty} \frac{1}{\rho^{k-n}} D_{nk}\, \beta_k, \quad b_{en} = \sum_{k=n}^{\infty} \frac{1}{\rho^{k-n}} D_{nk}\, \beta_{ek} \tag{15.30}$$

where α_k, β_k, α_{ek}, and β_{ek} are the corresponding coefficients for the horizontally pure solutions.

When expressed as a function of $\widetilde{r}$ and $\widetilde{y}$, the vertically pure solutions ϕ_n have a finite number of terms (Eqs. (15.22) and (15.23)). On the other hand, the horizontally pure solutions ψ_n have an infinite number of terms.

The vertically pure solutions form a complete set. That is, any given field that satisfies Maxwell's equations and is independent of z can be expressed as a linear combination of ϕ_n^r and ϕ_n^i. Similarly, the horizontally pure solutions form a complete set. [It is, of course, possible to construct other complete sets in which the basis functions are neither horizontally pure nor vertically pure.]

This brings up an important point. To properly simulate a machine, one must first of all understand whether the multipole values that have been handed to you are for horizontally pure multipoles, vertically, pure multipoles, or perhaps the values do not correspond to either horizontally pure nor vertically pure solutions! Failure to understand this point can lead to differing results. For example, the chromaticity induced by a horizontally pure quadrupole field will be different from the chromaticity of a vertically pure quadrupole field of the same strength. With *Bmad*, the `exact_multipoles` (§3.5) attribute of a bend is used to set whether multipole values are for vertically or horizontally pure solutions. [Note to programmers: PTC always assumes coefficients correspond to horizontally pure solutions. The *Bmad* PTC interface will convert coefficients as needed.]

15.4 Map Decomposition of Magnetic and Electric Fields

Electric and magnetic fields can be parameterized as the sum over a number of functions with each function satisfying Maxwell's equations. These functions are also referred to as "maps", "modes", or "terms". *Bmad* has two parameterizations:

```
Cartesian_Map      ! §15.5.
Cylindrical_Map    ! §15.6
```

These parameterizations are two of the four `field map` parameterizations that *Bmad* defines §4.15.

The `cartesian_map` decomposition involves a set of terms, each term a solution the Laplace equation solved using separation of variables in Cartesian coordinates. This decomposition can be used for DC but not AC fields. See §15.5. for more details. The syntax for specifying the `cartesian_map` decomposition is discussed in §4.15.2.

The `cylindrical_map` decomposition can be used for both DC and AC fields. See §15.6 for more details. The syntax for specifying the `cylindrical_map` decomposition is discussed in §4.15.3.

15.5 Cartesian Map Field Decomposition

Electric and magnetic fields can be parameterized as the sum over a number of functions with each function satisfying Maxwell's equations. These functions are also referred to as "maps", "modes", or

"terms". *Bmad* has two types. The "`Cartesian`" decomposition is explained here. The other type is the `cylindrical` decomposition (§15.6).

The `Cartesian` decomposition implemented by *Bmad* involves a set of terms, each term a solution the Laplace equation solved using separation of variables in Cartesian coordinates. This decomposition is for DC electric or magnetic fields. No AC Cartesian Map decomposition is implemented by *Bmad*. In a lattice file, a `Cartesian` map is specified using the `cartesian_map` attribute as explained in Sec. §4.15.2.

The `Cartesian` decomposition is modeled using an extension of the method of Sagan, Crittenden, and Rubin[Sagan03]. In this decomposition, the magnetic(or electric field is written as a sum of terms B_i (For concreteness the symbol B_i is used but the equations below pertain equally well to both electric and magnetic fields) with:

$$\mathbf{B}(x,y,z) = \sum_i \mathbf{B}_i(x,y,z; A, k_x, k_y, k_z, x_0, y_0, \phi_z, family) \tag{15.31}$$

Each term B_i is specified using seven numbers $(A, k_x, k_y, k_z, x_0, y_0, \phi_z)$ and a switch called `family` which can be one of:

```
x,  qu
y,  sq
```

Roughly, taking the offsets x_0 and y_0 to be zero (see the equations below), the `x family` gives a field on-axis where the y component of the field is zero. that is, the `x` family is useful for simulating, say, magnetic vertical bend dipoles. The `y family` has a field that on-axis has no x component. The `qu family` has a magnetic quadrupole like (which for electric fields is skew quadrupole like) field on-axis and the `sq family` has a magnetic skew quadrupole like field on-axis. Additionally, assuming that the x_0 and y_0 offsets are zero, the `sq` family, unlike the other three families, has a nonzero on-axis z field component.

Each family has three possible forms These are designated as "`hyper-y`", "`hyper-xy`", and "`hyper-x`".

For the `x family` the `hyper-y` form is:

$$\begin{aligned}
B_x &= \quad A \frac{k_x}{k_y} \cos(k_x(x+x_0)) \cosh(k_y(y+y_0)) \cos(k_z z + \phi_z) \\
B_y &= \quad A \quad \sin(k_x(x+x_0)) \sinh(k_y(y+y_0)) \cos(k_z z + \phi_z) \\
B_s &= -A \frac{k_z}{k_y} \sin(k_x(x+x_0)) \cosh(k_y(y+y_0)) \sin(k_z z + \phi_z) \\
&\qquad \text{with } k_y^2 = k_x^2 + k_z^2
\end{aligned} \tag{15.32}$$

The `x family hyper-xy` form is:

$$\begin{aligned}
B_x &= \quad A \frac{k_x}{k_z} \cosh(k_x(x+x_0)) \cosh(k_y(y+y_0)) \cos(k_z z + \phi_z) \\
B_y &= \quad A \frac{k_y}{k_z} \sinh(k_x(x+x_0)) \sinh(k_y(y+y_0)) \cos(k_z z + \phi_z) \\
B_s &= -A \quad \sinh(k_x(x+x_0)) \cosh(k_y(y+y_0)) \sin(k_z z + \phi_z) \\
&\qquad \text{with } k_z^2 = k_x^2 + k_y^2
\end{aligned} \tag{15.33}$$

And the `x family hyper-x` form is:

$$\begin{aligned}
B_x &= \quad A \quad \cosh(k_x(x+x_0)) \cos(k_y(y+y_0)) \cos(k_z z + \phi_z) \\
B_y &= -A \frac{k_y}{k_x} \sinh(k_x(x+x_0)) \sin(k_y(y+y_0)) \cos(k_z z + \phi_z) \\
B_s &= -A \frac{k_z}{k_x} \sinh(k_x(x+x_0)) \cos(k_y(y+y_0)) \sin(k_z z + \phi_z) \\
&\qquad \text{with } k_x^2 = k_y^2 + k_z^2
\end{aligned} \tag{15.34}$$

The relationship between k_x, k_y, and k_z ensures that Maxwell's equations are satisfied. Notice that which form `hyper-y`, `hyper-xy`, and `hyper-x` a particular $\mathbf{B}_i$ belongs to can be computed by *Bmad* by looking at the values of k_x, k_y, and k_z.

Using a compact notation where Ch $\equiv$ cosh, subscript x is $k_x(x + x_0)$, subscript z is $k_z z + \phi_z$, etc., the `y family` of forms is:

Form	hyper-y	hyper-xy	hyper-x
B_x	$-A \frac{k_x}{k_y} \mathrm{S}_x \mathrm{Sh}_y \mathrm{C}_z$	$A \frac{k_x}{k_z} \mathrm{Sh}_x \mathrm{Sh}_y \mathrm{C}_z$	$A \quad \mathrm{Sh}_x \mathrm{S}_y \mathrm{C}_z$
B_y	$A \quad \mathrm{C}_x \mathrm{Ch}_y \mathrm{C}_z$	$A \frac{k_y}{k_z} \mathrm{Ch}_x \mathrm{Ch}_y \mathrm{C}_z$	$A \frac{k_y}{k_x} \mathrm{Ch}_x \mathrm{C}_y \mathrm{C}_z$
B_z	$-A \frac{k_z}{k_y} \mathrm{C}_x \mathrm{Sh}_y \mathrm{S}_z$	$-A \quad \mathrm{Ch}_x \mathrm{Sh}_y \mathrm{S}_z$	$-A \frac{k_z}{k_x} \mathrm{Ch}_x \mathrm{S}_y \mathrm{S}_z$
with	$k_y^2 = k_x^2 + k_z^2$	$k_z^2 = k_x^2 + k_y^2$	$k_x^2 = k_y^2 + k_z^2$

(15.35)

the `qu family` of forms is:

Form	hyper-y	hyper-xy	hyper-x
B_x	$A \frac{k_x}{k_y} \mathrm{C}_x \mathrm{Sh}_y \mathrm{C}_z$	$A \frac{k_x}{k_z} \mathrm{Ch}_x \mathrm{Sh}_y \mathrm{C}_z$	$A \quad \mathrm{Ch}_x \mathrm{S}_y \mathrm{C}_z$
B_y	$A \quad \mathrm{S}_x \mathrm{Ch}_y \mathrm{C}_z$	$A \frac{k_y}{k_z} \mathrm{Sh}_x \mathrm{Ch}_y \mathrm{C}_z$	$A \frac{k_y}{k_x} \mathrm{Sh}_x \mathrm{C}_y \mathrm{C}_z$
B_z	$-A \frac{k_z}{k_y} \mathrm{S}_x \mathrm{Sh}_y \mathrm{S}_z$	$-A \quad \mathrm{Sh}_x \mathrm{Sh}_y \mathrm{S}_z$	$-A \frac{k_z}{k_x} \mathrm{Sh}_x \mathrm{S}_y \mathrm{S}_z$
with	$k_y^2 = k_x^2 + k_z^2$	$k_z^2 = k_x^2 + k_y^2$	$k_x^2 = k_y^2 + k_z^2$

(15.36)

the `sq family` of forms is:

Form	hyper-y	hyper-xy	hyper-x
B_x	$-A \frac{k_x}{k_y} \mathrm{S}_x \mathrm{Ch}_y \mathrm{C}_z$	$A \frac{k_x}{k_z} \mathrm{Sh}_x \mathrm{Ch}_y \mathrm{C}_z$	$-A \quad \mathrm{Sh}_x \mathrm{C}_y \mathrm{C}_z$
B_y	$A \quad \mathrm{C}_x \mathrm{Sh}_y \mathrm{C}_z$	$A \frac{k_y}{k_z} \mathrm{Ch}_x \mathrm{Sh}_y \mathrm{C}_z$	$A \frac{k_y}{k_x} \mathrm{Ch}_x \mathrm{S}_y \mathrm{C}_z$
B_z	$-A \frac{k_z}{k_y} \mathrm{C}_x \mathrm{Ch}_y \mathrm{S}_z$	$-A \quad \mathrm{Ch}_x \mathrm{Ch}_y \mathrm{S}_z$	$A \frac{k_z}{k_x} \mathrm{Ch}_x \mathrm{C}_y \mathrm{S}_z$
with	$k_y^2 = k_x^2 + k_z^2$	$k_z^2 = k_x^2 + k_y^2$	$k_x^2 = k_y^2 + k_z^2$

(15.37)

The singular case where $k_x = k_y = k_z = 0$ is not allowed. If a uniform field is needed, a term with very small k_x, k_y, and k_z can be used. Notice that since k_y must be non-zero for the `hyper-y` forms (remember, $k_y^2 = k_x^2 + k_z^2$ for these forms and not all k's can be zero), and k_z must be non-zero for the `hyper-xy` forms, and k_x must be nonzero for the `hyper-x` forms. The magnetic field is always well defined even if one of the k's is zero.

15.6 Cylindrical Map Decomposition

Electric and magnetic fields can be parameterized as the sum over a number of functions with each function satisfying Maxwell's equations. These functions are also referred to as "maps", "modes", or

"terms". *Bmad* has two types. The "`cylindrical`" decomposition is explained here. The other type is the `Cartesian` decomposition (§15.6).

In a lattice file, a `cylindrical` map is specified using the `cylindrical_map` attribute as explained in Sec. §4.15.3.

The `cylindrical` decomposition takes one of two forms depending upon whether the fields are time varying or not. The DC decomposition is explained in Sec. §15.6.1 while the RF decomposition is explained in Sec. §15.6.2.

15.6.1 DC Cylindrical Map Decomposition

The DC `cylindrical` parametrization used by *Bmad* essentially follows Venturini et al.[Venturini98]. See Section §4.15 for details on the synax used to cylindrical maps in *Bmad*. The electric and magnetic fields are both described by a scalar potential

$$\mathbf{B} = -\nabla \psi_B, \qquad \mathbf{E} = \nabla - \psi_E \tag{15.38}$$

The scalar potentials both satisfy the Laplace equation $\nabla^2 \psi = 0$. The scalar potentials are decomposed as a sum of modes indexed by an integer m

$$\psi_B = \text{Re}\left[\sum_{m=0}^{\infty} \psi_{Bm}\right] \tag{15.39}$$

[Here and below, only equations for the magnetic field will be shown, the equations for the electric fields are similar.] The ψ_{Bm} are decomposed in z using a discrete Fourier sum.[1] Expressed in cylindrical coordinates the decomposition of ψ_{Bm} is

$$\psi_{Bm} = \sum_{n=-N/2}^{N/2-1} \psi_{Bmn} = \sum_{n=-N/2}^{N/2-1} \frac{-1}{k_n}\, e^{i\,k_n\,z}\, \cos(m\,\theta - \theta_{0m})\, b_m(n)\, I_m(k_n\,\rho) \tag{15.40}$$

where I_m is a modified Bessel function of the first kind, and the $b_m(n)$ are complex coefficients. [For electric fields, $e_m(n)$ is substituted for $b_m(n)$] In Eq. (15.40) k_n is given by

$$k_n = \frac{2\pi\, n}{N\, dz} \tag{15.41}$$

where N is the number of "sample points", and dz is the longitudinal "distance between points". That is, the above equations will only be accurate over a longitudinal length $(N-1)\,dz$. Note: Typically the sum in Eq. (15.40) and other equations below runs from 0 to $N-1$. Using a sum from $-N/2$ to $N/2-1$ gives exactly the same field at the sample points $(z = 0, dz, 2\,ds, \ldots)$ and has the virtue that the field is smoother in between.

The field associated with ψ_{Bm} is for $m = 0$:

$$\begin{aligned} B_\rho &= \text{Re}\left[\sum_{n=-N/2}^{N/2-1} e^{i\,k_n\,z}\, b_0(n)\, I_1(k_n\,\rho)\right] \\ B_\theta &= 0 \\ B_z &= \text{Re}\left[\sum_{n=-N/2}^{N/2-1} i\, e^{i\,k_n\,z}\, b_0(n)\, I_0(k_n\,\rho)\right] \end{aligned} \tag{15.42}$$

[1] Venturini uses a continuous Fourier transformation but *Bmad* uses a discrete transformation so that only a finite number of coefficients are needed.

And for $m \neq 0$:

$$\begin{aligned} B_\rho &= \text{Re}\left[\sum_{n=-N/2}^{N/2-1} \frac{1}{2}\, e^{i\,k_n\,z}\, \cos(m\,\theta - \theta_{0m})\, b_m(n) \left[I_{m-1}(k_n\,\rho) + I_{m+1}(k_n\,\rho)\right]\right] \\ B_\theta &= \text{Re}\left[\sum_{n=-N/2}^{N/2-1} \frac{-1}{2}\, e^{i\,k_n\,z}\, \sin(m\,\theta - \theta_{0m})\, b_m(n) \left[I_{m-1}(k_n\,\rho) - I_{m+1}(k_n\,\rho)\right]\right] \\ B_z &= \text{Re}\left[\sum_{n=-N/2}^{N/2-1} i\, e^{i\,k_n\,z}\, \cos(m\,\theta - \theta_{0m})\, b_m(n)\, I_m(k_n\,\rho)\right] \end{aligned} \tag{15.43}$$

While technically ψ_{Bm0} is not well defined due to the $1/k_n$ factor that is present, the field itself is well behaved. Mathematically, Eq. (15.40) can be corrected if, for $n = 0$, the term $I_m(k_n\,\rho)/k_n$ is replaced by

$$\frac{I_m(k_0\,\rho)}{k_0} \to \begin{cases} \rho & \text{if } m = 0 \\ \rho/2 & \text{if } m = 1 \\ 0 & \text{otherwise} \end{cases} \tag{15.44}$$

The magnetic vector potential for $m = 0$ is constructed such that only A_θ is non-zero

$$\begin{aligned} A_\rho &= 0 \\ A_\theta &= \text{Re}\left[\sum_{n=-N/2}^{N/2-1} \frac{i}{k_n}\, e^{i\,k_n\,z}\, b_0(n)\, I_1(k_n\,\rho)\right] \\ A_z &= 0 \end{aligned} \tag{15.45}$$

For $m \neq 0$, the vector potential is chosen so that A_θ is zero.

$$\begin{aligned} A_\rho &= \text{Re}\left[\sum_{n=-N/2}^{N/2-1} \frac{-i\,\rho}{2\,m}\, e^{i\,k_n\,z}\, \cos(m\,\theta - \theta_{0m})\, b_m(n) \left[I_{m-1}(k_n\,\rho) - I_{m+1}(k_n\,\rho)\right]\right] \\ A_\theta &= 0 \\ A_z &= \text{Re}\left[\sum_{n=-N/2}^{N/2-1} \frac{-i\,\rho}{m}\, e^{i\,k_n\,z}\, \cos(m\,\theta - \theta_{0m})\, b_m(n)\, I_m(k_n\,\rho)\right] \end{aligned} \tag{15.46}$$

Note: The description of the field using ‘‘`generalized gradients`’’[Newton99] is similar to the above equations. The difference is that, with the generalized gradient formalism, terms in θ and ρ are expanded in a Taylor series in x and y.

15.6.2 AC Cylindrical Map Decomposition

For RF fields, the `cylindrical` mode parameterization used by *Bmad* essentially follows Abell[Abell06]. The electric field is the real part of the complex field

$$\mathbf{E}(\mathbf{r}) = \sum_{j=1}^{M} \mathbf{E}_j(\mathbf{r})\, \exp[-2\,\pi\,i\,(f_j\,t + \phi_{0j})] \tag{15.47}$$

where M is the number of modes. Each mode satisfies the vector Helmholtz equation

$$\nabla^2 \mathbf{E}_j + k_{tj}^2 \, \mathbf{E}_j = 0 \tag{15.48}$$

where $k_{tj} = 2\,\pi\, f/c$ with f_j being the mode frequency.

The individual modes vary azimuthally as $\cos(m\,\theta - \theta_0)$ where m is a non-negative integer. [in this and in subsequent equations, the mode index j has been dropped.] For the $m = 0$ modes, there is an accelerating mode whose electric field is in the form

$$\begin{aligned} E_\rho(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} -e^{i\,k_n\,z}\, i\,k_n\, e_0(n)\, \widetilde{I}_1(\kappa_n, \rho) \\ E_\theta(\mathbf{r}) &= 0 \\ E_z(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} e^{i\,k_n\,z}\, e_0(n)\, \widetilde{I}_0(\kappa_n, \rho) \end{aligned} \tag{15.49}$$

where $\widetilde{I}_m$ is

$$\widetilde{I}_m(\kappa_n, \rho) \equiv \frac{I_m(\kappa_n\,\rho)}{\kappa_n^m} \tag{15.50}$$

with I_m being a modified Bessel function first kind, and κ_n is given by

$$\kappa_n = \sqrt{k_n^2 - k_t^2} = \begin{cases} \sqrt{k_n^2 - k_t^2} & |k_n| > k_t \\ -i\,\sqrt{k_t^2 - k_n^2} & k_t > |k_n| \end{cases} \tag{15.51}$$

with

$$k_n = \frac{2\pi\, n}{N\, dz} \tag{15.52}$$

N is the number of points where E_{zc} is evaluated, and dz is the distance between points. The length of the field region is $(N-1)\,dz$. When κ_n is imaginary, $I_m(\kappa_n\,\rho)$ can be evaluated through the relation

$$I_m(-i\,x) = i^{-m}\, J_m(x) \tag{15.53}$$

where J_m is a Bessel function of the first kind. The e_0 coefficients can be obtained given knowledge of the field at some radius R via

$$e_0(n) = \frac{1}{\widetilde{I}_0(\kappa_n, R)}\, \frac{1}{N} \sum_{p=0}^{N-1} e^{-2\,\pi\, i\, n\, p/N}\, E_z(R, p\,dz) \tag{15.54}$$

The non-accelerating $m = 0$ mode has an electric field in the form

$$\begin{aligned} E_\rho(\mathbf{r}) &= E_z(\mathbf{r}) = 0 \\ E_\theta(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} e^{i\,k_n\,z}\, b_0(n)\, \widetilde{I}_1(\kappa_n, \rho) \end{aligned} \tag{15.55}$$

where the b_0 coefficients can be obtained given knowledge of the field at some radius R via

$$b_0(n) = \frac{1}{\widetilde{I}_1(\kappa_n, R)}\, \frac{1}{N} \sum_{p=0}^{N-1} e^{-2\,\pi\, i\, n\, p/N}\, E_\theta(R, p\,dz) \tag{15.56}$$

For positive m, the electric field is in the form

$$\begin{aligned}
E_\rho(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} -i\, e^{i\, k_n\, z} \left[k_n\, e_m(n)\, \widetilde{I}_{m+1}(\kappa_n, \rho) + b_m(n)\, \frac{\widetilde{I}_m(\kappa_n, \rho)}{\rho} \right] \cos(m\,\theta - \theta_{0m}) \\
E_\theta(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} -i\, e^{i\, k_n\, z} \left[k_n\, e_m(n)\, \widetilde{I}_{m+1}(\kappa_n, \rho) + \right. \\
&\qquad \left. b_m(n) \left(\frac{\widetilde{I}_m(\kappa_n, \rho)}{\rho} - \frac{1}{m}\, \widetilde{I}_{m-1}(\kappa_n, \rho) \right) \right] \sin(m\,\theta - \theta_{0m}) \\
E_z(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} e^{i\, k_n\, z}\, e_m(n)\, \widetilde{I}_m(\kappa_n, \rho) \cos(m\,\theta - \theta_{0m})
\end{aligned} \tag{15.57}$$

The `e_m` and `b_m` coefficients can be obtained given knowledge of the field at some radius R via

$$\begin{aligned}
e_m(n) &= \frac{1}{\widetilde{I}_m(\kappa_n, R)}\, \frac{1}{N} \sum_{p=0}^{N-1} e^{-2\,\pi\, i\, n\, p/N}\, E_{zc}(R, p\, dz) \\
b_m(n) &= \frac{R}{\widetilde{I}_m(\kappa_n, R)} \left[\frac{1}{N} \sum_{p=0}^{N-1} i\, e^{-2\,\pi\, i\, n\, p/N}\, E_{\rho c}(R, p\, dz) - k_n\, e_m(n)\, \widetilde{I}_{m+1}(\kappa_n, R) \right]
\end{aligned} \tag{15.58}$$

where $E_{\rho c}$, $E_{\theta s}$, and E_{zc} are defined by

$$\begin{aligned}
E_\rho(R, \theta, z) &= E_{\rho c}(R, z)\, \cos(m\,\theta - \theta_{0m}) \\
E_\theta(R, \theta, z) &= E_{\theta s}(R, z)\, \sin(m\,\theta - \theta_{0m}) \\
E_z(R, \theta, z) &= E_{zc}(R, z)\, \cos(m\,\theta - \theta_{0m})
\end{aligned} \tag{15.59}$$

The above mode decomposition was done in the gauge where the scalar potential ψ is zero. The electric and magnetic fields are thus related to the vector potential $\mathbf{A}$ via

$$\mathbf{E} = -\partial_t\, \mathbf{A}, \qquad \mathbf{B} = \nabla \times \mathbf{A} \tag{15.60}$$

Using Eq. (15.47), the vector potential can be obtained from the electric field via

$$\mathbf{A}_j = \frac{-i\, \mathbf{E}_j}{2\,\pi\, f_j} \tag{15.61}$$

Symplectic tracking through the RF field is discussed in Section §22.4. For the fundamental accelerating mode, the vector potential can be analytically integrated using the identity

$$\int dx\, \frac{x\, I_1(a\, \sqrt{x^2 + y^2})}{\sqrt{x^2 + y^2}} = \frac{1}{a}\, I_0(a\, \sqrt{x^2 + y^2}) \tag{15.62}$$

15.7 Field Modeling Using Taylor Maps

Bmad has a number of `field map` models that can be used to model electric or magnetic fields (§4.15). One model involves a set of Taylor maps. This model is restricted to modeling DC fields. In a lattice file, the `Taylor` field model is specified using the `taylor_field` attribute as exaplined in Sec. §4.15.5.

The Taylor field model specifies the field using a set of Taylor maps. Each map defines the field in the transverse (x, y) plane at constant longitudinal z: That is, each Taylor map is comprised of three Taylor series, one for each field component, and each Taylor series is a polynomial in x and y:

$$\mathbf{B} = (B_x(x, y; i), B_y(x, y; i), B_z(x, y; i)) \tag{15.63}$$

where the B_x, B_y, B_z on the right hand side represent the tree Taylor series, i denotes the Taylor map at $z_i = i \cdot dz$ where dz is the spacing between maps. The maps are restricted to be equally spaced to enable higher order integration schemes in PTC (See `default_integ_order` in §9.3). [Note: Each Taylor field model specifes either an electric or magnetic field. In this section, the symbol B can refer to either magnetic or electric fields.]

Interpolation of the field in *Bmad* is done using a cubic spline fit. Given a position (x, y, z), the field $\mathbf{B}$ and transverse field derivatives $\partial\mathbf{B}/\partial\mathbf{r}$, are evaluated at two positions $\mathbf{r}_0 = (x, y, z_{i0})$ and $\mathbf{r}_1 = (x, y, z_{i1})$ with z_{i0} being the positions of the maps to either side of z. The longitudinal field derivatives are derived from the transverse ones using Maxwell's equations:

$$\frac{\partial B_x}{\partial z} = \frac{\partial B_z}{\partial x}, \qquad \frac{\partial B_y}{\partial z} = \frac{\partial B_z}{\partial y}, \qquad \frac{\partial B_z}{\partial z} = -\left(\frac{\partial B_x}{\partial x} + \frac{\partial B_y}{\partial y}\right) \tag{15.64}$$

Once the field and longitudinal derivatives are known at $\mathbf{r}_0$ and $\mathbf{r}_1$, a standard cubic spline interpolation is done.

Note: If the Taylor field model uses curved coordinates, There will be inaccuracies in the computed field to the extent that Maxwell's equations have to be modified to take into account the coordinate curvature.

15.8 RF fields for Field_Calc = Bmad_Standard

The following describes the how RF fields are calculated when the the `field_calc` attribute of an RF element is set to `bmad_standard`.

The traveling wave model for RF cavities the field, in cylindrical coordinates is

$$\begin{aligned} E_s(r, \phi, s, t) &= G \cos\bigl(k\, s - \omega\, t + 2\,\pi\,\phi\bigr) \\ E_r(r, \phi, s, t) &= \frac{1}{2}\, G\, k\, r \sin\bigl(k\, s - \omega\, t + 2\,\pi\,\phi\bigr) \\ B_\phi(r, \phi, s, t) &= \frac{1}{2\,c}\, G\, k\, r \sin\bigl(k\, s - \omega\, t + 2\,\pi\,\phi\bigr) \end{aligned} \tag{15.65}$$

where G is the accelerating gradient, $k = \omega/c$ is the wave number with ω beging the RF frequency.

For standing wave cavities, the RF fields are modeled as N half-wave cells, each having a length of $\lambda/2$ where $\lambda = 2\,\pi/k$ is the wavelength. If the length of the RF element is not equal to the length of N cells, the "active region" is centered in the element and the regions to either side are treated as field free.

The field in the standing wave cell is modeled either with a $p = 0$ or $p = 1$ longitudinal mode. The $p = 1$ longitudinal mode (which is the default if the longitudinal mode is not set in the element), models the fields as a pillbox with the transverse wall at infinity as detailed in Chapter 3, Section VI of reference [Lee99]

$$\begin{aligned} E_s(r, \phi, s, t) &= 2\, G\, \cos(k\, s)\, \cos(\omega\, t + 2\,\pi\,\phi) \\ E_r(r, \phi, s, t) &= G\, k\, r\, \sin(k\, s)\, \cos(\omega\, t + 2\,\pi\,\phi) \\ B_\phi(r, \phi, s, t) &= \frac{-1}{c}\, G\, k\, r\, \cos(k\, s)\, \sin(\omega\, t + 2\,\pi\,\phi) \end{aligned} \tag{15.66}$$

The overall factor of 2 in the equation is present to ensure that an ultra-relativistic particle entering with $\phi = 0$ will experience an average gradient equal to G.

For the $p = 0$ longitudinal mode, a "pseudo TM_{010}" mode is used that has the correct symmetry:

$$\begin{aligned} E_s(r,\phi,s,t) &= 2\,G\,\sin(k\,s)\,\sin(\omega\,t + 2\,\pi\,\phi) \\ E_r(r,\phi,s,t) &= -G\,k\,r\,\cos(k\,s)\,\sin(\omega\,t + 2\,\pi\,\phi) \\ B_\phi(r,\phi,s,t) &= \frac{1}{c}\,G\;k\,r\,\sin(k\,s)\,\cos(\omega\,t + 2\,\pi\,\phi) \end{aligned} \tag{15.67}$$

15.9 Wake fields

15.9.1 Short–Range Wakes

Wake field effects are divided into short–range (within a bunch) and long–range (between bunches).

Only the transverse dipole and longitudinal monopole components of the short–range wake field are modeled. The longitudinal monopole energy kick dE for the i^{th} (trailing) macroparticle due to the wake from the j^{th} (leading) macroparticle is computed from the equation

$$\Delta p_z(i) = \frac{-e\,L}{v\,P_0}\left(\frac{1}{2}W_\parallel^{SR}(0)\,|q_i| + \sum_{j\neq i} W_\parallel^{SR}(dz_{ij})\,|q_j|\right) \tag{15.68}$$

where v is the particle velocity, e is the charge on an electron, q is the macroparticle charge, L is the cavity length, dz_{ij} is the longitudinal distance between the i^{th} and j^{th} macroparticles, $W_\parallel^{SR}$ is the short–range longitudinal wake field function.

For the transverse kick, there are two types of wakes: Those that are dependent upon transverse offset of the leading particle (but independent of the position of the trailing particle) and those that are dependent upon the transverse offset of the trailing particle (but independent of the position of the leading particle. If the beam chamber has azimuthal symmetry, the only wakes present are those that are dependent upon the offset of the leading particle.

Bmad assumes that the beam chamber wall is mirror symmetric with respect to both the x-axis and y-axis. In this case, a horizontal displacement (of either particle) results in a horizontal kick and similarly for vertical displacements. That is, the horizontal and vertical wakes are decoupled.

With the above symmetry assumpltion, the transverse kick $\Delta p_x(i)$ for the i^{th} macroparticle due to the dipole short–range transverse wake field is modeled with the equation

$$\Delta p_x(i) = \frac{-e\,L\,\sum_j |q_j|\,x\,W_\perp^{SR}(dz_{ij})}{v\,P_0} \tag{15.69}$$

Where x is the horizontal displacement of the leading or trailing particle as appropriate. There is a similar equation for $\Delta p_y(i)$. $W_\perp^{SR}$ is the transverse short–range wake function.

The wake field functions $W_\parallel^{SR}$ and $W_\perp^{SR}$ can be specified in a *Bmad* lattice file using "pseudo" modes where

$$W(z) = \sum_i A_i\,e^{d_i z}\,\sin(k_i\,z + \phi_i) \tag{15.70}$$

The parameters (A_i, d_i, k_i, ϕ_i) are chosen to fit the calculated wake potential (§4.19.1). The reason why the mode approach is used in *Bmad* is due to the fact that, using pseudo modes, the calculation time

scales as the number of particles N while a calculation based upon a table of wake vs z would scale as N^2. [The disadvantage is that initially the user must proform a fit to the wake potential to generate the mode parameter values.]

15.9.2 Long–Range Wakes

Following Chao[Chao93] Eq. 2.88, the long–range wake fields are characterized by a set of cavity modes. The wake function W_m for a mode of order m is given by

$$W_m(t) = -c \left(\frac{R}{Q}\right)_m e^{-\omega\, t/2Q} \sin(\omega\, t) \tag{15.71}$$

The mode strength $(R/Q)_m$ has units of Ohms/meter^{2m}. The lattice syntax for defining long-range wakes is discussed in Sec. §4.19.2.

Assuming that the macroparticle generating the wake is offset a distance r_w along the x–axis, a trailing macroparticle will see a kick

$$\begin{aligned} \Delta\mathbf{p}_\perp &= -C\, I_m\, W_m(t)\, m\, r^{m-1} \left(\hat{\mathbf{r}} \cos m\theta - \hat{\theta} \sin m\theta\right) && (15.72)\\ &= -C\, I_m\, W_m(t)\, m\, r^{m-1} \left(\hat{\mathbf{x}} \cos[(m-1)\theta] - \hat{\mathbf{y}} \sin[(m-1)\theta]\right) \\ \Delta p_z &= -C\, I_m\, W'_m(t)\, r^m \cos m\theta && (15.73) \end{aligned}$$

where m is the order of the mode, C is given by

$$C = \frac{e}{c\, P_0} \tag{15.74}$$

and

$$I_m = q_w\, r_w^m \tag{15.75}$$

with q_w being the magnitude of the charge on the particle. Generalizing the above, a macroparticle at (r_w, θ_w) will generate a wake

$$\begin{aligned} -\Delta p_x + i\Delta p_y &= C\, I_m\, W_m(t)\, m\, r^{m-1}\, e^{-im\theta_w}\, e^{i(m-1)\theta} && (15.76)\\ \Delta p_z &= C\, I_m\, W'_m(t)\, r^m \cos[m(\theta - \theta_w)] && (15.77) \end{aligned}$$

Comparing Eq. (15.76) to (15.9), and using the relationship between kick and field as given by (15.5) and (15.6), shows that the form of the wake field transverse kick is the same as for a multipole of order $n = m - 1$.

The wake field felt by a particle is due to the wake fields generated by all the particles ahead of it. If the wake field kicks are computed by summing over all particle pairs, the computation will scale as N^2 where N is the number of particles. This quickly becomes computationally exorbitant. A better solution is to keep track of the wakes in a cavity. When a particle comes through, the wake it generates is simply added to the existing wake. This computation scales as N and makes simulations with large number of particles practical.

To add wakes together, a wake must be decomposed into its components. Spatially, there are normal and skew components and temporally there are sin and cosine components. This gives 4 components which will be labeled $a_{\cos}$, $a_{\sin}$, $b_{\cos}$, and $b_{\sin}$. For a mode of order m, a particle passing through at a

time t_w with respect to the reference particle will produce wake components

$$\begin{aligned}
\delta a_{\sin,m} &= \quad c \left(\frac{R}{Q}\right)_m e^{\omega\, t_w/2Q} \cos(\omega\, t_w)\, I_m \sin(m\theta_w) \\
\delta a_{\cos,m} &= -\, c \left(\frac{R}{Q}\right)_m e^{\omega\, t_w/2Q} \sin(\omega\, t_w)\, I_m \sin(m\theta_w) \\
\delta b_{\sin,m} &= \quad c \left(\frac{R}{Q}\right)_m e^{\omega\, t_w/2Q} \cos(\omega\, t_w)\, I_m \cos(m\theta_w) \\
\delta b_{\cos,m} &= -\, c \left(\frac{R}{Q}\right)_m e^{\omega\, t_w/2Q} \sin(\omega\, t_w)\, I_m \cos(m\theta_w)
\end{aligned} \tag{15.78}$$

These are added to the existing wake components. The total is

$$a_{\sin,m} = \sum_{\text{particles}} \delta a_{\sin,m} \tag{15.79}$$

with similar equations for $a_{\cos,m}$ etc. Here the sum is over all particles that cross the cavity before the kicked particle. To calculate the kick due to wake, the normal and skew components are added together

$$\begin{aligned}
a_m &= e^{-\omega\, t/2Q} \left(a_{\cos,m} \cos(\omega\, t) - a_{\sin,m} \sin(\omega\, t)\right) \\
b_m &= e^{-\omega\, t/2Q} \left(b_{\cos,m} \cos(\omega\, t) - b_{\sin,m} \sin(\omega\, t)\right)
\end{aligned} \tag{15.80}$$

Here t is the passage time of the particle with respect to the reference particle. In analogy to Eq. (15.76) and (15.77), the kick is

$$-\Delta p_x + i\Delta p_y = C\, m\, (b_m + i a_m)\, r^{m-1}\, e^{i(m-1)\theta} \tag{15.81}$$

$$\Delta p_z = -C\, r^m \left((b'_m + i a'_m) e^{im\theta} + (b'_m - i a'_m) e^{-im\theta}\right) \tag{15.82}$$

where $a' \equiv da/dt$ and $b' \equiv db/dt$.

When simulating trains of bunches, the exponential factor $\omega\, t_w/2Q$ in Eq. (15.78) can become very large. To prevent numerical overflow, *Bmad* uses a reference time $z_{\rm ref}$ so that all times t in the above equations are replaced by

$$t \longrightarrow t - t_{\rm ref} \tag{15.83}$$

The above equations were developed assuming cylindrical symmetry. With cylindrical symmetry, the cavity modes are actually a pair of degenerate modes. When the symmetry is broken, the modes no longer have the same frequency. In this case, one has to consider a mode's polarization angle ϕ. Equations (15.80) and (15.81) are unchanged. In place of Eq. (15.78), the contribution of a particle to a mode is

$$\begin{aligned}
\delta a_{\sin,m} &= \quad c \left(\frac{R}{Q}\right)_m e^{\omega\, t_w/2Q} \cos(\omega\, t_w)\, I_m \left[\sin(m\theta_w) \sin^2(m\phi) + \cos(m\theta_w) \sin(m\phi) \cos(m\phi)\right] \\
\delta a_{\cos,m} &= -\, c \left(\frac{R}{Q}\right)_m e^{\omega\, t_w/2Q} \sin(\omega\, t_w)\, I_m \left[\sin(m\theta_w) \sin^2(m\phi) + \cos(m\theta_w) \sin(m\phi) \cos(m\phi)\right] \\
\delta b_{\sin,m} &= \quad c \left(\frac{R}{Q}\right)_m e^{\omega\, t_w/2Q} \cos(\omega\, t_w)\, I_m \left[\cos(m\theta_w) \cos^2(m\phi) + \sin(m\theta_w) \sin(m\phi) \cos(m\phi)\right] \\
\delta b_{\cos,m} &= -\, c \left(\frac{R}{Q}\right)_m e^{\omega\, t_w/2Q} \sin(\omega\, t_w)\, I_m \left[\cos(m\theta_w) \cos^2(m\phi) + \sin(m\theta_w) \sin(m\phi) \cos(m\phi)\right]
\end{aligned} \tag{15.84}$$

Each mode is characterized by an R/Q, Q, ω, and m. Notice that R/Q is defined so that it includes the cavity length. Thus the long–range wake equations, as opposed to the short–range ones, do not have any explicit dependence on L.

To make life more interesting, different people define R/Q differently. A common practice is to define an R/Q "at the beam pipe radius". In this case the above equations must be modified to include factors of the beam pipe radius. Another convention uses a "linac definition" which makes R/Q twice as large and adds a factor of 2 in Eq. (15.71) to compensate.

Chapter 16

Fringe Fields

The fringe field kick is divided into two pieces. The first piece is called the `hard edge` fringe kick and is the kick in the limit that the longitudinal extent of the fringe is zero. The second piece is the `soft edge` fringe kick which is the fringe kick with the fringe having a finite longitudinal extent minus the hard edge fringe kick. That is

```
fringe kick = hard fringe kick + soft fringe kick
```

The advantage of separating the fringe kick in this way is that the hard fringe can be used without having to know anything about the longitudinal extent of the fringe field. In many cases, this is a good enough approximation.

16.1 Bend Soft Edge Fringe Map

Bmad defines the bend soft edge map in terms of the field integral F_{H1} for the entrance end and F_{H2} for the exit end given by (see Eq. (3.6))

$$F_{H1} \equiv F_{int}\, H_{gap} = \int_{pole} ds\, \frac{B_y(s)\,(B_{y0} - B_y(s))}{2\,B_{y0}^2} \tag{16.1}$$

With a similar equation for F_{H2}. The soft edge map is then

$$\begin{aligned} x_2 &= x_1 + c_1\, p_z \\ p_{y2} &= p_{y1} + c_2\, y_1 - c_3\, y_1^3 \\ z_2 &= z_1 + \frac{1}{1+p_{z1}} \left(c_1\, p_{x1} + \frac{1}{2}\, c_2\, y_1^2 - \frac{1}{4}\, c_3\, y_1^4 \right) \end{aligned} \tag{16.2}$$

For the entrance face:

$$c_1 = \frac{g_{\rm tot}\, F_{H1}^2}{2\,(1+p_z)}, \qquad c_2 = \frac{2\, g_{\rm tot}^2\, F_{H1}}{1+p_z}, \qquad c_3 = 0 \tag{16.3}$$

with `g_tot` is the total bending strength

$$g_{\rm tot} = {\rm g} + {\rm g\_err} \tag{16.4}$$

`g` being the reference bend strength and `g_err` being bend the difference between the actual and reference bend strengths (§3.5).

For the exit face, the subsitution is made

$$\begin{aligned} F_{H1} &\to F_{H2} \\ g_{\rm tot} &\to -g_{\rm tot} \end{aligned} \tag{16.5}$$

When the SAD bend soft edge map is used (§4.20), the map is the same except that the value of c_3 is

$$c_3 = \frac{8\, g_{\rm tot}^2}{F_{H1}\,(1+p_z)} \tag{16.6}$$

It might seem strange that c_3 diverges to infinity as F_H goes to zero since naively one would expect the soft edge kick to vanish in the hard edge limit where the fringe has no longitudinal extent. However, in the hard edge limit, the field does not obey Maxwell's equations. The limiting map, as F_H goes to zero, has fields that diverge to infinity and this exaplains why the full (hard + soft) limiting map is not the same as the hard edge map at the limit of zero longitudinal extent.

For a `sad_mult` element, the field integrals are characterized by parameters `fb1` (entrance end) and `fb2` (exit end) which correspond to the SAD `fb1` and `fb2` parameters. These are related to the bend parameters by

$$\text{fb1} = 12\, F_{H1}, \qquad \text{fb2} = 12\, F_{H2} \tag{16.7}$$

16.2 Bend Hard Edge Fringe Map

The bend fringe kick is a combination of the equations developed by Hwang and Lee[Hwang15] modified to include quadrupole terms as given in Section 5.3.1 of Iselin[Iselin94]. The Lie map generator Ω_M given by Hwang and Lee Eqs. (35) and (36) is used under the conditions that

$$K_0 = K_1 = K_3 = K_4 = K_5 = K_6 = 0 \tag{16.8}$$

The generator used by *Bmad* for the entrance fringe is:

$$\begin{aligned} \Omega_{M1} = {} & \frac{(x^2-y^2)\, g_{\rm tot}\, \tan(e_1)}{2} + \frac{y^2\, g_{\rm tot}^2\, \sec^3(e_1)\,[1+\sin^2(e_1)]\, f_{int}\, h_{gap}}{2\,(1+p_z)} \\ & + \frac{x^3\,[4\,K_1\,\tan(e_1) - g_{\rm tot}^2\,\tan^3(e_1)]}{12\,(1+p_z)} + \frac{x\,y^2\,[-4\,K_1\,\tan(e_1) + g_{\rm tot}^2\,\tan(e_1)\,\sec^2(e_1)]}{4\,(1+p_z)} \\ & + \frac{(x^2\,p_x - 2\,x\,y\,p_y)\, g_{\rm tot}\,\tan^2(e_1)}{2\,(1+p_z)} - \frac{y^2\,p_x\, g_{\rm tot}\,[1+\tan^2(e_1)]}{2\,(1+p_z)} \end{aligned} \tag{16.9}$$

where $g_{\rm tot}$ is the total bending strength (design + error). The generator for the exit fringe is

$$\begin{aligned} \Omega_{M2} = {} & \frac{(x^2-y^2)\, g_{\rm tot}\, \tan(e_2)}{2} + \frac{y^2\, g_{\rm tot}^2\, \sec^3(e_2)\,[1+\sin^2(e_2)]\, f_{int}\, h_{gap}}{2\,(1+p_z)} \\ & + \frac{x^3\,[4\,K_1\,\tan(e_2) - g_{\rm tot}^2\,\tan^3(e_2)]}{12\,(1+p_z)} + \frac{x\,y^2\,[-4\,K_1\,\tan(e_2) + g_{\rm tot}^2\,\tan(e_2)\,\sec^2(e_2)]}{4\,(1+p_z)} \\ & + \frac{(-x^2\,p_x + 2\,x\,y\,p_y)\, g_{\rm tot}\,\tan^2(e_2)}{2\,(1+p_z)} + \frac{y^2\,p_x\, g_{\rm tot}\,[1+\tan^2(e_2)]}{2\,(1+p_z)} \end{aligned} \tag{16.10}$$

The map $\mathcal{M}$ is obtained from the equation $\mathcal{M} = \exp[:\Omega_M:]$. To second order in the transverse coordinates the map can be obtained by expanding the exponantial to second order

$$\mathcal{M} \simeq 1 + :\Omega_M: + \frac{1}{2} :\Omega_M: :\Omega_M: \tag{16.11}$$

The transport for the entrance fringe is then

$$\begin{aligned}
\Delta x &= \frac{g_{\rm tot}}{2\,(1+p_z)}\,\left[-x^2\,\tan^2(e_1) + y^2\sec^2(e_1)\right] \\
\Delta p_x &= x\,g_{\rm tot}\,\tan(e_1) + \frac{y^2\,g_{\rm tot}^2\,[\tan(e_1) + 2\,\tan^3(e_1)]}{2\,(1+p_z)} \\
&\qquad + \frac{(x^2-y^2)\,K_1\tan(e_1)}{1+p_z} + \frac{(x\,p_x - y\,p_y)\,g_{\rm tot}\,\tan^2(e_1)}{1+p_z} \\
\Delta y &= \frac{x\,y\,g_{\rm tot}\,\tan^2(e_1)}{1+p_z} \\
\Delta p_y &= y\left[-g_{\rm tot}\,\tan(e_1) + \frac{g_{\rm tot}^2\,[1+\sin^2(e_1)]\,\sec^3(e_1)}{1+p_z}\,f_{int}\,h_{gap}\right] \\
&\qquad - \frac{(x\,p_y\,g_{\rm tot}\,\tan^2(e_1)}{1+p_z} - \frac{y\,p_x\,g_{\rm tot}\,[1+\tan^2(e_1)]}{1+p_z} - \frac{2\,x\,y\,K_1\tan(e_1)}{(1+p_z)} \\
\Delta z &= \frac{\widehat{\Omega}_{M1}}{1+p_z}
\end{aligned} \tag{16.12}$$

where $\widehat{\Omega}_{M1} = \Omega_{M1} - (x^2-y^2)\,g_{\rm tot}\,\tan(e_1)/2$. The transport for the exit fringe is

$$\begin{aligned}
\Delta x &= \frac{g_{\rm tot}}{2\,(1+p_z)}\,\left[x^2\,\tan^2(e_2) - y^2\sec^2(e_2)\right] \\
\Delta p_x &= x\,g_{\rm tot}\,\tan(e_2) - \frac{(x^2+y^2)\,g_{\rm tot}^2\,\tan^3(e_2)}{2\,(1+p_z)} \\
&\qquad + \frac{(x^2-y^2)\,K_1\tan(e_2)}{1+p_z} + \frac{(-x\,p_x + y\,p_y)\,g_{\rm tot}\,\tan^2(e_2)}{1+p_z} \\
\Delta y &= -\frac{x\,y\,g_{\rm tot}\,\tan^2(e_2)}{1+p_z} \\
\Delta p_y &= y\left[-g_{\rm tot}\,\tan(e_2) + \frac{g_{\rm tot}^2\,[1+\sin^2(e_2)]\,\sec^3(e_2)}{1+p_z}\,f_{int}\,h_{gap}\right] + \frac{x\,y\,g_{\rm tot}^2\,\sec^2(e_2)\,\tan(e_2)}{1+p_z} \\
&\qquad + \frac{(x\,p_y\,g_{\rm tot}\,\tan^2(e_1)}{1+p_z} + \frac{y\,p_x\,g_{\rm tot}\,[1+\tan^2(e_1)]}{1+p_z} - \frac{2\,x\,y\,K_1\tan(e_2)}{(1+p_z)} \\
\Delta z &= \frac{\widehat{\Omega}_{M2}}{1+p_z}
\end{aligned} \tag{16.13}$$

where $\widehat{\Omega}_{M2} = \Omega_{M2} - \frac{(x^2-y^2)\,g_{\rm tot}\,\tan(e_2)}{2}$

16.3 Quadrupole Soft Edge Fringe Map

Only the quadrupole soft edge fringe is modeled in *Bmad*. The model is adapted from SAD[SAD]. The fringe map is:

$$\begin{aligned}
x_2 &= x_1\,e^{g_1} + g_2\,p_{x1} \\
p_{x2} &= p_{x1}e^{-g_1} \\
y_2 &= y_1\,e^{-g_1} - g_2\,p_{y1} \\
p_{y2} &= p_{y1}e^{g_1} \\
z_2 &= z_1 - \left[g_1\,x_1\,p_{x1} + g_2\,\left(1+\frac{g_1}{2}\right)\,e^{-g_1}\,p_{x1}^2\right] + \left[g_1\,y_1\,p_{y1} + g_2\,\left(1-\frac{g_1}{2}\right)\,e^{g_1}\,p_{y1}^2\right]
\end{aligned} \tag{16.14}$$

where

$$g_1 = \frac{K_1\,\mathrm{fq1}}{1+p_z}, \qquad g_2 = \frac{K_1\,\mathrm{fq2}}{1+p_z} \tag{16.15}$$

K_1 is the quadrupole strength, and `fq1` and `fq2` are the fringe quadrupole parameters. These parameters are related to the field integral I_n via

$$\mathrm{fq1} = I_1 - \frac{1}{2}\,I_0^2, \qquad \mathrm{fq2} = I_2 - \frac{1}{3}\,I_0^3 \tag{16.16}$$

where I_n is defined by

$$I_n = \frac{1}{K_1}\int_{-\infty}^{\infty} \left(K_1(s) - H(s-s_0)\,K_1\right)(s-s_0)^n\,ds \tag{16.17}$$

and $H(s)$ is the step function

$$H(s) = \begin{cases} 1 & s > 0 \\ 0 & s < 0 \end{cases} \tag{16.18}$$

and it is assumed that the quadrupole edge is at s_0 and the interior is in the region $s > s_0$.

See Sec. §4.20 for the relation between `fq1` / `fq2` and the corresponding `f1` and `f2` parameters of SAD.

16.4 Magnetic Multipole Hard Edge Fringe

The magnetic multipole hard edge fringe field is modeled using the method shown in Forest[Forest98]. For the m^{th} order multipole the Lee transform is (Forest Eq. (13.29)):

$$\begin{aligned} f_\pm &= \mp\Re\left[\frac{(b_m + i\,a_m)\,(x+i\,y)^{m+1}}{4\,(m+2)\,(1+\delta)}\,\left\{x\,p_x + y\,p_y + i\frac{m+3}{m+1}(x\,p_x - y\,p_y)\right\}\right] \\ &\equiv \frac{p_x\,f^x + p_y\,f^y}{1+\delta} \end{aligned} \tag{16.19}$$

The multipole strengths a_m and b_m are given by (15.9) and the second equation defines f^x and f^y. On the right had side of the first equation, the minus sign is appropriate for particles entering the magnet and the plus sign is for particle leaving the magnet. Notice that here the multipole order m is equivalent to $n-1$ in Forest's notation.

With this, the implicit multipole map is (Forest Eq. (13.31))

$$\begin{aligned} x^f &= x - \frac{f^x}{1+\delta} \\ p_x &= p_x^f - \frac{p_x^f\,\partial_x f^x + p_y^f\,\partial_x f^y}{1+\delta} \\ y^f &= y - \frac{f_y}{1+\delta} \\ p_y &= p_y^f - \frac{p_x^f\,\partial_y f^x + p_y^f\,\partial_y f^y}{1+\delta} \\ \delta^f &= \delta \\ z^f &= \frac{p_x^f\,f^x + p_y^f\,f^y}{(1+\delta)^2} \end{aligned} \tag{16.20}$$

16.5 Electrostatic Multipole Hard Edge Fringe

The electric multipole hard edge fringe field, to lowest order, consists of just a longitudinal field. The integrated longitudinal field at constant (x, y) for the n^{th} order multipole is simply obtained by requireing that the curl of the field is zero. This gives:

$$\int E_s(x, y)\, ds = \phi_n(x, y) \tag{16.21}$$

where ϕ_n is given in Eq. (15.18). [For a magnetic multipole there is an analogous equation.]

The effect on the spin when tracking through the fringe field of a multipole field tends to be weak. As such, this hard edge model is sufficient. and the spin is tracked using the T-BMT equation (Eq. (19.1)).

Chapter 17

Multiparticle Simulation

Bmad has routines for tracking two types of objects called "`particles`" and "`macroparticles`". `Particles` are characterized by a six-vector representing the particle's phase space coordinates and a pair of complex numbers characterizing the particle's spin. A macroparticle is like a particle with the addition of a 6×6 "sigma" matrix characterizing the size of the macroparticle.

Macroparticle tracking was implemented in *Bmad* in order to simulate particle bunches. The idea was that far fewer macroparticles than particles would be needed to characterize a bunch. In practice, it was found that the complexity of handling the macroparticle sigma matrix more than offset the reduction in the number of particles needed. Hence, while the basic macroparticle tracking routines still exist, macroparticle tracking is not currently maintained and the use of this code is discouraged. However macroparticle tracking could be revived in the future if there is a demonstrated need for it.

Particle tracking can be divided into "single particle" tracking and "beam" tracking. Single particle tracking is simply tracking a single particle. Beam tracking is tracking an ensemble of particles divided up into a number of bunches that make up a "beam".

17.1 Bunch Initialization

[Developed by Michael Saelim]

To better visualize the evolution of a particle beam, it is sometimes convenient to initialize the beam with the particles regularly spaced. The following two algorithms are implemented in *Bmad* for such a purpose.

See Chapter `c:beam.init` for details on the standard input format used by *Bmad* based programs for reading in bunch initialization parameters.

17.1.1 Elliptical Phase Space Distribution

To observe nonlinear effects on the beam, it is sometimes convenient to initialize a bunch of particles in a way that puts more particles in the tails of the bunch than one would normally have with the standard method of seeding particles using a Gaussian distribution. In order to preserve the emittance, a distribution with more particles in the tail needs to decrease the charge per tail particle relative to the core. This feature, along with a regular distribution, are contained in the following ``ellipse''

distribution algorithm.

Consider the two dimensional phase space (x, p_x). The transformation to action-angle coordinates, (J, ϕ), is

$$J = \frac{1}{2}[\gamma x^2 + 2\alpha x x' + \beta x'^2] \tag{17.1}$$

$$\tan\phi = \frac{-\beta\,(x' + \alpha\,x)}{x} \tag{17.2}$$

The inverse is

$$\begin{pmatrix} x \\ x' \end{pmatrix} = \sqrt{2J}\begin{pmatrix} \sqrt{\beta} & 0 \\ -\frac{\alpha}{\sqrt{\beta}} & -\frac{1}{\sqrt{\beta}} \end{pmatrix}\begin{pmatrix} \cos\phi \\ \sin\phi \end{pmatrix}. \tag{17.3}$$

In action-angle coordinates, the normalized Gaussian phase space distribution, $\rho(J, \phi)$, is

$$\rho(J, \phi) = \frac{1}{2\pi\varepsilon} e^{-\frac{J}{\varepsilon}}. \tag{17.4}$$

where the emittance ε is just the average of J over the distribution

$$\varepsilon = \langle J \rangle \equiv \int dJ\, d\phi\, J\rho(J, \phi). \tag{17.5}$$

The beam sizes σ and σ' are

$$\sigma = \sqrt{\langle x^2 \rangle} = \sqrt{\varepsilon\beta} \tag{17.6}$$

$$\sigma' = \sqrt{\langle x'^2 \rangle} = \sqrt{\varepsilon\gamma}, \tag{17.7}$$

and the covariance is

$$\langle x x' \rangle = -\varepsilon\alpha. \tag{17.8}$$

The `ellipse` algorithm starts by partitioning phase space into regions bounded by ellipses of constant $J = B_n$, $n = 0, \ldots N_J$. The boundary values B_n are chosen so that, except for the last boundary, the $\sqrt{B_n}$ are equally spaced

$$B_n = \begin{cases} \frac{\varepsilon}{2}\left(\frac{n_\sigma\, n}{N}\right)^2 & \text{for } 0 \le n < N_J \\ \infty & \text{for } n = N_J \end{cases} \tag{17.9}$$

where n_σ is called the **‘‘`boundary sigma cutoff`’’**. Within each region, an elliptical shell of constant J_n is constructed with N_ϕ particles equally spaced in ϕ. The charge q_n of each particle of the n^{th} ellipse is chosen so that the total charge of all the particles of the ellipse is equal to the total charge within the region

$$N_\phi\, q_n = \int_{B_{n-1}}^{B_n} dJ \int_0^{2\pi} d\phi\, \rho(J, \phi) = \exp\left(-\frac{B_{n-1}}{\varepsilon}\right) - \exp\left(-\frac{B_n}{\varepsilon}\right) \tag{17.10}$$

The value of J_n is chosen to coincide with the average J within the region

$$N_\phi\, q_n\, J_n = \int_{B_{n-1}}^{B_n} dJ \int_0^{2\pi} d\phi\, J\, \rho(J, \phi) = \varepsilon(\xi + 1)e^{-\xi}\Big|_{\frac{B_n}{\varepsilon}}^{\frac{B_{n-1}}{\varepsilon}} \tag{17.11}$$

The `ellipse` phase space distribution is thus

$$\rho_{model}(J, \phi) = q_{tot} \sum_{n=1}^{N_J} q_n\, \delta(J - J_n) \sum_{m=1}^{N_\phi} \delta(\phi - 2\pi\frac{m}{N_\phi}) \tag{17.12}$$

where q_{tot} is the total charge. At a given point in the lattice, where the Twiss parameters are known, the input parameters needed to construct the `ellipse` phase space distribution is n_σ, N_J, N_ϕ, and q_{tot}.

The `ellipse` distribution is two dimensional in nature but can easily be extended to six dimensions.

17.1.2 Kapchinsky-Vladimirsky Phase Space Distribution

The Kapchinsky-Vladimirsky (`KV`) distribution can be thought of as a four dimensional analog of the `ellipse` distribution with only one elliptical shell. Consider a 4D phase space (x, x', y, y'). Using this framework, a 4D Gaussian distribution is

$$\rho(J_x, \phi_x, J_y, \phi_y) = \frac{1}{(2\pi)^2 \varepsilon_x \varepsilon_y}\, exp(-\frac{J_x}{\varepsilon_x})\, exp(-\frac{J_y}{\varepsilon_y}) \tag{17.13}$$

$$= \frac{1}{(2\pi)^2 \varepsilon_x \varepsilon_y}\, exp(-\frac{I_1}{\varepsilon}), \tag{17.14}$$

where the orthogonal action coordinates are:

$$I_1 = \left(\frac{J_x}{\varepsilon_x} + \frac{J_y}{\varepsilon_y}\right)\varepsilon \tag{17.15}$$

$$I_2 = \left(-\frac{J_x}{\varepsilon_y} + \frac{J_y}{\varepsilon_x}\right)\varepsilon \tag{17.16}$$

with $\varepsilon = (\frac{1}{\varepsilon_x^2} + \frac{1}{\varepsilon_y^2})^{-1/2}$. The reverse transformation is:

$$J_x = \left(\frac{I_1}{\varepsilon_x} - \frac{I_2}{\varepsilon_y}\right)\varepsilon \tag{17.17}$$

$$J_y = \left(\frac{I_1}{\varepsilon_y} + \frac{I_2}{\varepsilon_x}\right)\varepsilon. \tag{17.18}$$

The `KV` distribution is

$$\rho(I_1, I_2, \phi_x, \phi_y) = \frac{1}{A}\delta(I_1 - \xi), \tag{17.19}$$

where $A = \frac{\varepsilon_x \varepsilon_y}{\varepsilon^2}\xi(2\pi)^2$ is a constant which normalizes the distribution to 1. By choosing a particular ξ, and iterating over the domain of the three remaining coordinates, one can populate a 3D subspace of constant density.

The range in I_2 to be iterated over is constrained by $J_x,\ J_y \geq 0$. Thus $I_2 is in the range [-\frac{\varepsilon_x}{\varepsilon_y} I_1, \frac{\varepsilon_y}{\varepsilon_x} I_1]$. This range is divided into N regions of equal size, with a ring of particles placed in the middle of each region. The angle variables are also constrained to $\phi_x, \phi_y \in [0, 2\pi]$, with each range divided into M_x and M_y regions, respectively. Each of these regions will have a particle placed in its center.

The weight of a particle is determined by the total weight of the region of phase space it represents. Because the density ρ is only dependent on I_1,

$$q = \int_0^\infty dI_1 \int_{I_2}^{I_2+\Delta I_2} dI_2 \int_{\phi_x}^{\phi_x+\Delta\phi_x} d\phi_x \int_{\phi_y}^{\phi_y+\Delta\phi_y} d\phi_y\ \frac{1}{A}\delta(I_1 - \xi) \tag{17.20}$$

$$= \frac{1}{A}\Delta I_2 \Delta\phi_x \Delta\phi_y. \tag{17.21}$$

To represent the distribution with particles of equal weight, we must partition (I_2, ϕ_x, ϕ_y)-space into regions of equal volume.

The weight of each particle is

$$q = \frac{1}{N M_x M_y} = \frac{1}{N_{tot}} \tag{17.22}$$

where N_{tot} is the total number of particles

17.2 Touschek Scattering

[Developed by Michael Ehrlichman]

Touschek scattering occurs when a single scattering event between two particles in the same beam transfers transverse momentum to longitudinal momentum, and the resulting change in longitudinal momentum results in the loss of one or both particles. In the case of storage rings, these losses impose a beam lifetime. In low-emittance storage rings, Touschek scattering can be the dominant mechanism for particle loss. In the case of linear accelerators, these losses generate radiation in the accelerator tunnel. When the scattered particles collide with the beam chamber, x-rays are produced which can damage equipment and impose a biohazard. Studies of Touschek scattering typically look at beam lifetime and locations where scattering occurs and where particles are lost.

A commonly utilized theory for studying Touschek scattering is from Piwinski [Piwin98]. A basic outline of the derivation is,

1. Scatter two particles from a bunch in their COM frame using the relativistic Moller cross-section.
2. Boost from COM frame to lab frame. Changes to longitudinal momentum end up amplified by a factor of γ.
3. Integrate over 3D Gaussian distribution of particle positions and angles.

During the derivation many approximations are made which lead to a relatively simple formula. The integration is set up such that only those collisions which will result in particle loss are counted. The formula takes the momentum aperture as a parameter. The resulting formula is reproduced here to give the reader an idea of what influences the scattering rate, and how one might go about evaluating the formula,

$$R = \frac{r_e^2 c \beta_x \beta_y \sigma_h N_p^2}{8\sqrt{\pi}\beta^2\gamma^4\sigma_{x\beta}^2\sigma_{y\beta}^2\sigma_s\sigma_p}\int_{\tau_m}^{\infty}\left(\left(2+\frac{1}{\tau}\right)^2\left(\frac{\tau/\tau_m}{1+\tau}-1\right)+1-\frac{\sqrt{1+\tau}}{\sqrt{\tau/\tau_m}}\right.$$
$$\left.-\frac{1}{2\tau}\left(4+\frac{1}{\tau}\right)\ln\frac{\tau/\tau_m}{1+\tau}\right)\frac{\sqrt{\tau}}{\sqrt{1+\tau}}e^{-B_1\tau}I_0\left[B_2\tau\right]d\tau, \quad (17.23)$$

where $\tau_m = \beta^2\delta_m^2$ and δ_m is the momentum aperture. This formula gives the rate at which particles are scattered out of the bunch. It is assumed that two particles are lost per scattering event, one with too much energy and one with too little energy. If a machine with an unsymmetric momentum aperture is being studied, then the formula should be evaluated twice, once for each aperture, and the results averaged. Refer to [Piwin98] for definitions of the parameters involved. This formula is implemented in BMAD as part of the `touschek_mod` module.

Different formulas for calculating the Touschek scattering rate exist elsewhere in the literature. For example, Wiedemann [Wiede99], presents a formula with a simpler integrand. This formula, originally from a paper by LeDuff [Duff87], is derived in a fashion similar to Piwinski except that the formula does not take dispersion into account and uses a non-relativistic scattering cross-section. Since Piwinski's formula is the most robust, it is the one used in *Bmad*.

Particles are lost from Touschek scattering due to two effects. In storage rings, there is a momentum aperture defined by the RF system that is often referred to as the RF bucket. If the δp imparted by a Touschek scattering event exceeds this RF bucket, then the particle will no longer undergo synchrotron oscillations with the rest of the bunch and will coast through the accelerator. Second, if the Touschek scattering event occurs in a dispersive region, the scattered particles will take on a finite J and undergo

betatron oscillations. These oscillations can be large in amplitude and may cause the particles to collide with the beam pipe. To first order, the amplitude of J due to a scattering event that imparts a momentum deviation of Δp is,

$$J \approx \gamma_0 \mathcal{H}_0 \frac{\Delta p^2}{2}, \tag{17.24}$$

where γ_0 is relativistic γ and $\mathcal{H}_0$ is the dispersion invariant.

17.3 Macroparticles

Note: The macroparticle tracking code is not currently maintained in favor of tracking an ensemble of particles where each particle is specified by a position without a sigma matrix. The following is present for historical reference only.

A macroparticle[Brown77] is represented by a centroid position $\bar{\mathbf{r}}$ and a 6×6 $\boldsymbol{\sigma}$ matrix which defines the shape of the macroparticle in phase space. $\sigma_i = \sqrt{\boldsymbol{\sigma}(i,i)}$ is the RMS sigma for the i^{th} phase space coordinate. For example $\sigma_z = \sqrt{\boldsymbol{\sigma}(5,5)}$.

$\boldsymbol{\sigma}$ is a real, non-negative symmetric matrix. The equation that defines the ellipsoid at a distance of n–sigma from the centroid is

$$(\mathbf{r} - \bar{\mathbf{r}})^t \boldsymbol{\sigma}^{-1} (\mathbf{r} - \bar{\mathbf{r}}) = n \tag{17.25}$$

where the t superscript denotes the transpose. Given the sigma matrix at some point $s = s_1$, the sigma matrix at a different point s_2 is

$$\boldsymbol{\sigma}_2 = \mathbf{M}_{12}\, \boldsymbol{\sigma}_1\, \mathbf{M}_{12}^t \tag{17.26}$$

where $\mathbf{M}_{12}$ is the Jacobian of the transport map from point s_1 to s_2.

The Twiss parameters can be calculated from the sigma matrix. The dispersion is given by

$$\begin{aligned} \sigma(1,6) &= \eta_x\, \sigma(6,6) \\ \sigma(2,6) &= \eta_x'\, \sigma(6,6) \\ \sigma(3,6) &= \eta_y\, \sigma(6,6) \\ \sigma(4,6) &= \eta_y'\, \sigma(6,6) \end{aligned} \tag{17.27}$$

Ignoring coupling for now, the betatron part of the sigma matrix can be obtained from the linear equations of motion. For example, using

$$x = \sqrt{2\,\beta_x\,\epsilon_x}\cos\phi_x + \eta_x\, p_z \tag{17.28}$$

Solving for the first term on the RHS, squaring and averaging over all particles gives

$$\beta_x\,\epsilon_x = \sigma(1,1) - \frac{\sigma^2(1,6)}{\sigma(6,6)} \tag{17.29}$$

It is thus convenient to define the betatron part of the sigma matrix

$$\sigma_\beta(i,j) \equiv \sigma(i,j) - \frac{\sigma(i,6)\,\sigma(j,6)}{\sigma(6,6)} \tag{17.30}$$

and in terms of the betatron part the emittance is

$$\epsilon_x^2 = \sigma_\beta(1,1)\,\sigma_\beta(2,2) - \sigma_\beta^2(1,2) \tag{17.31}$$

and the Twiss parameters are

$$\epsilon_x \begin{pmatrix} \beta_x & -\alpha_x \\ -\alpha_x & \gamma_x \end{pmatrix} = \begin{pmatrix} \sigma_\beta(1,1) & \sigma_\beta(1,2) \\ \sigma_\beta(1,2) & \sigma_\beta(2,2) \end{pmatrix} \tag{17.32}$$

If there is coupling, the transformation between the 4×4 transverse normal mode sigma matrix $\boldsymbol{\sigma}_a$ and the 4×4 laboratory matrix $\boldsymbol{\sigma}_x$ is

$$\boldsymbol{\sigma}_x = \mathbf{V}\, \boldsymbol{\sigma}_a \mathbf{V}^t \tag{17.33}$$

where $\mathbf{V}$ is given by Eq. (20.5).

The sigma matrix is the same for all macroparticles and is determined by the local Twiss parameters:

$$\begin{aligned} \sigma(1,1) &= \epsilon_x\, \beta_x \\ \sigma(1,2) &= -\epsilon_x \alpha_x \\ \sigma(2,2) &= \epsilon_x\, \gamma_x = \epsilon_x\, (1+\alpha_x^2)/\beta_x \\ \sigma(3,3) &= \epsilon_y\, \beta_y \\ \sigma(3,4) &= -\epsilon_y \alpha_b \\ \sigma(3,4) &= \epsilon_y\, \gamma_y = \epsilon_y\, (1+\alpha_b^2)/\beta_y \\ \sigma(i,j) &= 0 \quad \text{otherwise} \end{aligned} \tag{17.34}$$

The centroid energy of the k^{th} macroparticle is

$$E_k = E_b + \frac{(n_{mp} - 2\,k + 1)\, \sigma_E\, N_{\sigma E}}{n_{mp}} \tag{17.35}$$

where E_b is the central energy of the bunch, n_{mp} is the number of macroparticles, σ_E is the energy sigma, and $N_{\sigma E}$ is the number of sigmas in energy that the range of macroparticle energies cover. The charge of each macroparticle is, within a constant factor, the charge contained within the energy region $E_k - dE_{mp}/2$ to $E_k + dE_{mp}/2$ assuming a Gaussian distribution where the energy width dE_{mp} is

$$dE_{mp} = \frac{2\, \sigma_E\, N_{\sigma E}}{n_{mp}} \tag{17.36}$$

17.4 Space Charge and Coherent Synchrotron Radiation

The electric field $\mathbf{E}$ felt by particle A due to particle B can be described using the Liénard-Wiechert formula [Sagan09]. The field is singular as the distance between particles goes to zero so one approach to handling this is to decompose the field into two parts: One part, called the "space charge" (SC) or "Coulomb" term, $\mathbf{E}_{SC}$ is the field that would result if the particles where moving without acceleration along a straight line. The "Coherent Synchrotron Radiation" (CSR) term $\mathbf{E}_{CSR}$ is everything else $\mathbf{E}_{CSR} \equiv \mathbf{E} - \mathbf{E}_{SC}$. Generally, the longitudinal component of the SC kick is negligible compared to the CSR kick at large enough particle energies.

The SC term is singular at small distances while the CSR term is not. This being the case, it is possible to model the CSR term using a 1-dimensional formalism where the beam is approximated as a line charge[Sagan09, Sagan17]. In this formalism, the CSR kick is strictly longitudinal.

Transport through a lattice element with SC and CSR involves a beam of particles. The lattice element is divided up into a number of slices. Transport through a slice is a two step process. The first step is to give all the particles a kick due to SC and CSR. The second step is transport of all particles from one slice to the next without any interaction between particles. User settable parameters pertinent to the CSR calculation are listed in §9.4.

17.4.1 1_Dim CSR Calculation

When an element's `csr_method` is set to `1_dim` (§5.5), The particle-particle CSR kick is calculated by dividing the bunch longitudinally into a number of bins. To smooth the computed bin densities, each particle of the bunch is considered to have a triangular density distribution as shown in Fig. 17.1. The particle density of a bin is calculated by summing the contribution from all the particles. The contribution of a given particle to a given bin is calculated from the overlap of the particle's triangular density distribution with the bin. For the CSR kick, the density is actually calculated for a second set of staggered bins that have been offset by 1/2 the bin width with respect to the first set. This gives the density at the edges of the original set of bins. The density is considered to vary linearly between the computed density points. For a description of the parameters that affect the CSR calculation see Section §9.4.

17.4.2 Slice Space Charge Calculation

When an element's `space_charge_method` is set to `slice` (§5.5), the calculation of the SC kick uses, the same particle binning as is used with the `1_dim` CSR calculation (§17.4.1). The kick is divided into longitudinal and a transverse parts. The transverse part uses the same Bassetti–Erskine complex error function formula[Talman87] as with the beam-beam interaction (§22.5) except here, since all the particles are moving in the same direction, the kicks due to the electric and magnetic fields generated

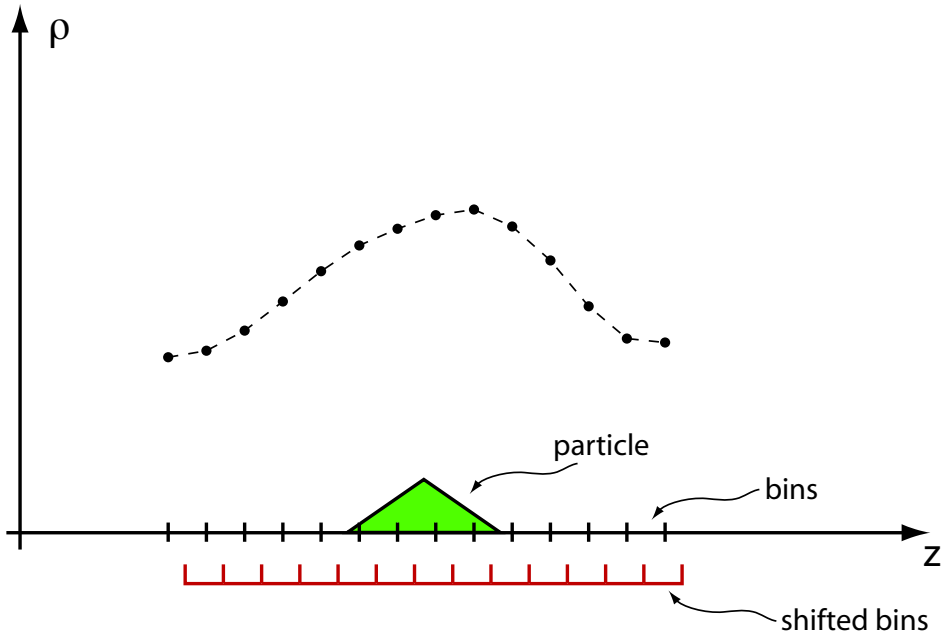


Figure 17.1: The Coherent Synchrotron Radiation kick is calculated by dividing longitudinally a bunch into a number of bins. To smooth the computed densities, each particle of the bunch is considered to have a triangular density distribution.

by a given particle tend to cancel

$$K_y(\text{CS}) + i\,K_x(\text{CS}) = \frac{r_e\,\rho(z)}{\gamma^3\,e}\cdot\sqrt{\frac{2\,\pi\,(\sigma_x+\sigma_y)}{\sigma_x-\sigma_y}} \left\{ w\left[\frac{x+i\,y}{\sqrt{2(\sigma_x^2-\sigma_y^2)}}\right] - \exp\left[-\frac{x^2}{2\,\sigma_x^2}-\frac{y^2}{2\,\sigma_y^2}\right]\cdot w\left[\frac{x\,\frac{\sigma_y}{\sigma_x}+i\,y\,\frac{\sigma_x}{\sigma_y}}{\sqrt{2(\sigma_x^2-\sigma_y^2)}}\right]\right\} \tag{17.37}$$

where $K(\text{CS})$ is the CS kick per unit length of travel of the beam, $\rho(z)$ is the density of particles per unit length evaluated at the z position of the kicked particle, e is the charge on the electron, and w is the complex error function.

The longitudinal SC kick is given by Eq. (31) of Sagan[Sagan09]

$$dK_{\text{SC}} = \frac{r_c mc^2\,\text{sign}(\zeta)\rho(z')dz'}{\sigma_x\,\sigma_y\,\exp\left[\frac{x^2}{2\,\sigma_x^2}+\frac{y^2}{2\,\sigma_y^2}\right]+\frac{\sigma_x^2+\sigma_y^2}{\sigma_x+\sigma_y}\,\gamma|\zeta|+\gamma^2\zeta^2}\ , \tag{17.38}$$

where ζ is the longitudinal distance between the kick point and the slice doing the kicking. There are two simulation modes for the longitudinal SC kick. In both these modes, the kick is evaluated at the center plane of each slice. The kick is a sum kicks from all the slices. Since the thickness of the slices is, in general, not negligible, the the integral over a slice is used to calculate the kick. The total kick $K_{\text{SC}}(j)$ at slice j is

$$K_{\text{SC}}(j) = \sum_i \int_{\zeta_{ij}-dz_{slice}/2}^{\zeta_{ij}+dz_{slice}/2} d\zeta\,dK_{\text{SC}} \tag{17.39}$$

where the sum is over all slices i, ζ_{ij} is the distance between slices i and j, and dz_{slice} is the slice thickness. An analytic expression of the above integral is easily calculated assuming that the charge density $\rho(z)$ is linearly varying within a given slice. For brevity's sake, the calculation is not explicitly presented here. Once the kick at the slice center planes is calculated, the kick given to a particle is calculated using linear interpolation.

One mode for calculating the transverse SC kick which is computationally fast, ignores the transverse dependence of the kick and just evaluates the kick on the beam centerline. The other simulation mode represents the kick due to a given slice using a Padé approximant of form

$$\int_{\zeta_{ij}-dz_{slice}/2}^{\zeta_{ij}+dz_{slice}/2} d\zeta\,dK_{\text{SC}} \simeq \frac{1}{a_{00}+a_{20}x^2+a_{40}x^4+a_{02}y^2+a_{04}y^4+a_{22}x^2\,y^2} \tag{17.40}$$

the `a_mn` are calculated from an analytic formula derived from integrating Eq. (17.38). The reason for using this form is that it is a reasonable approximation even for very large x or y in that the actual and approximate kick both go to zero in this limit. That this Padé approximant is reasonable is dependent upon the fact that all the a_{mn} for a slice are either all positive or all negative. Kicks from different slices can be combined using standard Differential Algebra techniques to give a summed kick in the same form as above. To avoid divergences, for a given j where the kick is evaluated, all the kicks from slices with negative coefficients are combined together and all the kicks from slices with positive coefficients are combined together and the total kick is then the sum of the "positive kick" part and the "negative kick" part. The kick applied to a particle is calculated by first evaluating the kick, at the particle's x and y, at the neighboring slices and then using linear interpolation.

Note: Match elements (§3.30) can have orbit shifts which are not well handled by the CSR algorithm. For this reason, match elements are ignored in the CSR calculation.

17.4.3 FFT_3D Space Charge Calculation

When an element's `space_charge_method` is set to `fft_3d` (§5.5), the space charge calculation uses code from the OpenSC package developed by Rob Ryne and Chris Mayes [Ryne18]. The method works by calculating the field due to the particle on a 3D grid and then interpolating the field to the points where the particles are. This method will be able to handle lower energy bunches than the `slice` method (§17.4.2) the disadvantage is that the `fft_3d` method will be slower. Another point to keep in mind is that the `fft_3d` method currently ignores image charge forces due to an emitting cathode and so will not be accruate for tracking at the very lowest energies near the emitter.

Note: The mesh size is set by the `bmad_com` parameter `space_charge_mesh_size` (§9.2).

17.5 High Energy Space Charge

Bmad has a code module for simulating the effect of space charge (SC) at high energies. This is separate from the regular space charge calculation of §17.4. Thus it should be noted that turning on of both the regular space charge and the high energy space charge in the same element will result in double counting of the space charge effect.

The advantage of the high energy space charge algorithm is that the kick on a given particle is computed assuming a Gaussian beam with the beam size calculated using emittances supplied by the user. Thus the high energy space charge calculation can be done in single particle tracking (§17) as opposed to the beam tracking that must be used for the regular space charge calculation. The other advantage is that the high energy space charge calculation is quick since it is assumed that the kick is small enough so that the kick is only applied once per lattice element. The disadvantage of the high energy space charge calculation is that there is the assumption that the beam distribution is Gaussian which is generally acceptable for storage rings at relatively high energy but will not accurate in other situations.

If a *Bmad* based program has been constructed to use the high energy space charge module (the documentation for the program should indicate if this is true), the high energy space charge force can be turned on or off by setting the `parameter[high_energy_space_charge_on]` parameter (§8.1, §8.4).

The high energy space charge kick is computed assuming a gaussian bunch shape

$$K_y + i\,K_x = \frac{r_e\,N}{\gamma^3\,\sigma_z}\,\exp\left[\frac{z^2}{2\,\sigma_z}\right]\cdot\sqrt{\frac{\sigma_x+\sigma_y}{\sigma_x-\sigma_y}} \left\{ w\left[\frac{x+i\,y}{\sqrt{2(\sigma_x^2-\sigma_y^2)}}\right] - \exp\left[-\frac{x^2}{2\,\sigma_x^2}-\frac{y^2}{2\,\sigma_y^2}\right]\cdot w\left[\frac{x\,\frac{\sigma_y}{\sigma_x}+i\,y\,\frac{\sigma_x}{\sigma_y}}{\sqrt{2(\sigma_x^2-\sigma_y^2)}}\right]\right\} \tag{17.41}$$

where N is the number of particles in the bunch. This equation is similar to Eq. (17.37) except that $\rho(z)$ has been replaced assuming that the longitudinal distribution is Gaussian. For particles close to the bunch core the kick is linear with displacement giving rise to a tune shift [Decking00].

The high energy space charge calculation ignores any CSR effects and ignores any longitudinal kicks and is thus not a good approximation at lower energies. See the discussion in [Sagan09] for more details.

Chapter 18

Synchrotron Radiation

18.1 Synchrotron Radiation Damping and Excitation

Emission of synchrotron radiation by a particle can be decomposed into two parts. The deterministic average energy emitted produces damping while the stochastic fluctuating part of the energy spectrum produces excitation[Jowett87].

The treatment of radiation damping by *Bmad* essentially follows *MAD*. The average change in energy ΔE of a particle going through a section of magnet due to synchrotron radiation is

$$\frac{\Delta E}{E_0} = -k_d \, (1 + p_z) \tag{18.1}$$

where

$$k_d \equiv \frac{2\, r_e}{3} \, \gamma_0^3 \, \left\langle g_0^2 \right\rangle \, L_p \, (1 + p_z) \tag{18.2}$$

r_e is the classical electron radius, L_p is the actual path length, γ_0 is the energy factor of an on-energy particle, $1/g_0$ is the bending radius of an on–energy particle, and $\left\langle g_0^2 \right\rangle$ is an average of g_0^2 over the actual path.

The energy lost is given by

$$\frac{\Delta E}{E_0} = -k_f \, (1 + p_z) \tag{18.3}$$

where

$$k_f \equiv \left(\frac{55\, r_e \, \hbar \, c}{24 \sqrt{3}\, m_e} \, L_p \, \gamma_0^5 \left\langle g_0^3 \right\rangle \right)^{1/2} (1 + p_z) \, \xi \tag{18.4}$$

ξ is a Gaussian distributed random number with unit sigma and zero mean.

Using Eqs. (18.2) and (18.4) the total change in p_z can be written as

$$\Delta p_z = \frac{\Delta E}{E_0} = -k_E \, (1 + p_z) \tag{18.5}$$

where

$$k_E = k_d + k_f \tag{18.6}$$

Since the radiation is emitted in the forward direction the angles x' and y' are invariant which leads to the following equations for the changes in p_x and p_y

$$\begin{aligned}\Delta p_x &= -k_E\, p_x \\ \Delta p_y &= -k_E\, p_y\end{aligned} \tag{18.7}$$

The above formalism does not take into account the fact that radiation is emitted with a $1/\gamma$ angular distribution. This means that the calculated vertical emittance for a lattice with bends only in the horizontal plane and without any coupling elements such as skew quadrupoles will be zero. Typically, in practice, the vertical emittance will be dominated by coupling so this approximation is generally a good one.

18.2 Synchrotron Radiation Integrals

The synchrotron radiation integrals are used to compute emittances, the energy spread, etc. The standard formulas assume no coupling between the horizontal and vertical planes[Helm73, Jowett87]. With coupling, the equations need to be generalized and this is detailed below.

In the general case, the curvature vector $\mathbf{g} = (g_x, g_y)$, which points away from the center of curvature of the particle's orbit and has a magnitude of $|\mathbf{g}| = 1/\rho$, where ρ is the radius of curvature (see Fig. 14.2), does not lie in the horizontal plane. Similarly, the dispersion $\boldsymbol{\eta} = (\eta_x, \eta_y)$ will not lie in the horizontal plane. With this notation, the synchrotron integrals for coupled motion are:

$$I_0 = \oint ds\, \gamma_0\, g \tag{18.8}$$

$$I_1 = \oint ds\, \mathbf{g}\cdot\boldsymbol{\eta} \equiv \oint ds\, (g_x\, \eta_x + g_y\, \eta_y) \tag{18.9}$$

$$I_2 = \oint ds\, g^2 \tag{18.10}$$

$$I_3 = \oint ds\, g^3 \tag{18.11}$$

$$I_{4a} = \oint ds\, \left[g^2\, \mathbf{g}\cdot\boldsymbol{\eta}_a + \nabla g^2\cdot\boldsymbol{\eta}_a\right] \tag{18.12}$$

$$I_{4b} = \oint ds\, \left[g^2\, \mathbf{g}\cdot\boldsymbol{\eta}_b + \nabla g^2\cdot\boldsymbol{\eta}_b\right] \tag{18.13}$$

$$I_{4z} = \oint ds\, \left[g^2\, \mathbf{g}\cdot\boldsymbol{\eta} + \nabla g^2\cdot\boldsymbol{\eta}\right] \tag{18.14}$$

$$I_{5a} = \oint ds\, g^3\, \mathcal{H}_a \tag{18.15}$$

$$I_{5b} = \oint ds\, g^3\, \mathcal{H}_b \tag{18.16}$$

$$I_{6b} = \oint ds\, g^3\, \beta_b \tag{18.17}$$

where γ_0 is that usual relativistic factor and $\mathcal{H}_a$ is

$$\mathcal{H}_a = \gamma_a\, \eta_a^2 + 2\,\alpha_a\, \eta_a\, \eta_a' + \beta_a \eta_a'^2 \tag{18.18}$$

with a similar equation for $\mathcal{H}_b$. Here $\boldsymbol{\eta}_a = (\eta_{ax}, \eta_{ay})$, and $\boldsymbol{\eta}_b = (\eta_{bx}, \eta_{by})$ are the dispersion vectors for the a and b modes respectively in x–y space (these 2–vectors are not to be confused with the dispersion

4–vectors used in the previous section). The position dependence of the curvature function is:

$$\begin{aligned} g_x(x,y) &= g_x + x\,k_1 + y\,s_1 \\ g_y(x,y) &= g_y + x\,s_1 - y\,k_1 \end{aligned} \tag{18.19}$$

where k_1 is the quadrupole moment and s_1 is the skew–quadrupole moment. Using this gives on–axis ($x = y = 0$)

$$\nabla g^2 = 2\,(g_x k_1 + g_y s_1,\; g_x s_1 - g_y k_1) \tag{18.20}$$

I_0 is not a standard radiation integral. It is useful, though, in calculating the average number of photons emitted. For electrons:

$$\mathcal{N} = \frac{5\,r_f}{2\sqrt{3}\,\hbar\,c}\,I_0 \tag{18.21}$$

where $\mathcal{N}$ is the average number of photons emitted by a particle over one turn, and the "classical radius factor" r_f is

$$r_f = \frac{e^2}{4\,\pi\,\epsilon_0} \tag{18.22}$$

r_f has a value of $1.4399644 \cdot 10^{-9}$ meters-eV for all particles of charge ± 1.

In a dipole a non–zero e_1 or e_2 gives a contribution to I_4 via the $\nabla g^2 \cdot \boldsymbol{\eta}$ term. The edge field is modeled as a thin quadrupole of length δ and strength $k = -\tan(e)/\delta$. It is assumed that $\mathbf{g}$ rises linearly within the edge field from zero on the outside edge of the edge field to its full value on the inside edge of the edge field. Using this in Eq. (18.20) and integrating over the edge field gives the contribution to I_4 from a non–zero e_1 as

$$I_{4z} = -\tan(e_1)\,g^2\,(\cos(\theta)\,\eta_x + \sin(\theta)\,\eta_y) \tag{18.23}$$

With an analogous equation for a finite e_2. The extension to I_{4a} and I_{4b} involves using $\boldsymbol{\eta}_a$ and $\boldsymbol{\eta}_b$ in place of $\boldsymbol{\eta}$. In Eq. (18.23) θ is the `tilt` angle which is non–zero if the bend is not in the horizontal plane.

The above integrals are invariant under rotation of the (x, y) coordinate system and reduce to the standard equations when $g_y = 0$ as they should.

There are various parameters that can be expressed in terms of these integrals. The I_1 integral can be related to the momentum compaction α_p via

$$I_1 = \alpha_p\,L \tag{18.24}$$

where L is the storage ring circumference. The energy loss per turn is

$$U_0 = \frac{2\,r_e E_0^4}{3\,(mc^2)^3} I_2 \tag{18.25}$$

where E_0 is the nominal energy and r_e is the classical electron radius (electrons are assumed here but the formulas are easily generalized).

The damping partition numbers are

$$J_a = 1 - \frac{I_{4a}}{I_2}\,, \quad J_b = 1 - \frac{I_{4b}}{I_2}\,, \text{ and } \quad J_z = 2 + \frac{I_{4z}}{I_2}\,. \tag{18.26}$$

Since

$$\boldsymbol{\eta}_a + \boldsymbol{\eta}_b = \boldsymbol{\eta}\,, \tag{18.27}$$

Robinson's theorem, $J_a + J_b + J_z = 4$, is satisfied. Alternatively, the exponential damping coefficients per turn are

$$\alpha_a = \frac{U_0\,J_a}{2E_0}\,, \quad \alpha_b = \frac{U_0\,J_b}{2E_0}\,, \text{ and } \quad \alpha_z = \frac{U_0\,J_z}{2E_0}\,. \tag{18.28}$$

The energy spread is given by

$$\sigma_{pz}^2 = \left(\frac{\sigma_E}{E_0}\right)^2 = C_q \gamma_0^2 \frac{I_3}{2I_2 + I_{4z}} \tag{18.29}$$

where γ_0 is the usual energy factor and

$$C_q = \frac{55}{32\sqrt{3}} \frac{\hbar}{mc} = 3.84 \times 10^{-13} \text{ meter for electrons} \tag{18.30}$$

If the synchrotron frequency is not too large, the bunch length is given by

$$\sigma_z^2 = \frac{I_1}{M(6,5)} \sigma_{pz}^2 \tag{18.31}$$

where $M(6,5)$ is the $(6,5)$ element for the 1–turn transfer matrix of the storage ring. Finally, the emittances are given by

$$\begin{aligned} \epsilon_a &= \frac{C_q}{I_2 - I_{4a}} \gamma_0^2 I_{5a} \\ \epsilon_b &= \frac{C_q}{I_2 - I_{4b}} \left(\gamma_0^2 I_{5b} + \frac{13}{55} I_{6b}\right) \end{aligned} \tag{18.32}$$

The I_{6b} term come from the finite vertical opening angle of the radiation[Rauben91]. Normally this term is very small compared to the emittance due to coupling or vertical kicks due to magnet misalignment.

For a non-circular machine, radiation integrals are still of interest if there are bends or steering elements. However, in this case, the appropriate energy factors must be included to take account any changes in energy due to any `lcavity` elements. For a non-circular machine, the I_1 integral is not altered and the I_4 integrals are not relevant. The other integrals become

$$L_2 = \int ds\, g^2\, \gamma_0^4 \tag{18.33}$$

$$L_3 = \int ds\, g^3\, \gamma_0^7 \tag{18.34}$$

$$L_{5a} = \int ds\, g^3\, \mathcal{H}_a\, \gamma_0^6 \tag{18.35}$$

$$L_{5b} = \int ds\, g^3\, \mathcal{H}_b\, \gamma_0^6 \tag{18.36}$$

In terms of these integrals, the energy loss through the lattice is

$$U_0 = \frac{2\, r_e\, mc^2}{3} L_2 \tag{18.37}$$

The energy spread assuming σ_E is zero at the start and neglecting any damping is

$$\sigma_E^2 = \frac{4}{3} C_q\, r_e\, \left(mc^2\right)^2\, L_3 \tag{18.38}$$

The above equation is appropriate for a linac. In a storage ring, where there are energy oscillations, the growth of σ_E^2 due to quantum excitation is half that. One way to explain this is that in a storage ring, the longitudinal motion is "shared" between the z and pz coordinates and, to preserve phase space volume, this reduces σ_E^2 by a factor of 2.

Again neglecting any initial beam width, the transverse beam size at the end of the lattice is

$$\epsilon_a = \frac{2}{3} C_q \, r_e \, \frac{L_{5a}}{\gamma_f}$$
$$\epsilon_b = \frac{2}{3} C_q \, r_e \, \frac{L_{5b}}{\gamma_f} \tag{18.39}$$

Where γ_f is the final gamma.

Chapter 19

Spin Dynamics

19.1 Equations of Motion

The propagation of the classical spin vector $\mathbf{S}$ is described in the local reference frame (§14.1.1) by a modified Thomas-Bargmann-Michel-Telegdi (T-BMT) equation[Hoff06]

$$\frac{\mathrm{d}}{\mathrm{d}s}\mathbf{S} = \left\{\frac{(1+\mathbf{r}_t\cdot\mathbf{g})}{c\,\beta_z}\,(\boldsymbol{\Omega}_{BMT}+\boldsymbol{\Omega}_{EDM}) - \mathbf{g}\times\widehat{\mathbf{z}}\right\}\times\mathbf{S} \tag{19.1}$$

where $\mathbf{g}$ is the bend curvature function which points away from the center of curvature of the particle's reference orbit (see Fig. 14.2), $\mathbf{r}_t = (x, y)$ are the transverse coordinates, $c\,\beta_z$ is the longitudinal component of the velocity, and $\widehat{\mathbf{z}}$ is the unit vector in the z-direction. $\boldsymbol{\Omega}_{BMT}$ is the usual T-BMT precession vector due to the particle's magnetic moment and $\boldsymbol{\Omega}_{EDM}$ is the precession vector due to a finite Electric Dipole Moment (EDM) [Silenko08]. Note: The value for the EDM is set by `bmad_com%electric_dipole_moment` (§9.2).

$$\begin{aligned}\boldsymbol{\Omega}_{BMT}(\mathbf{r},\mathbf{P},t) &= -\frac{q}{m\,c}\left[\left(\frac{1}{\gamma}+a\right)c\,\mathbf{B} - \frac{a\,\gamma\,c}{1+\gamma}\,(\boldsymbol{\beta}\cdot\mathbf{B})\,\boldsymbol{\beta} - \left(a+\frac{1}{1+\gamma}\right)\boldsymbol{\beta}\times\mathbf{E}\right] \\ &= -\frac{q}{m\,c}\left[\left(\frac{1}{\gamma}+a\right)c\,\mathbf{B}_\perp + \frac{(1+a)\,c}{\gamma}\,\mathbf{B}_\parallel - \left(a+\frac{1}{1+\gamma}\right)\boldsymbol{\beta}\times\mathbf{E}\right]\end{aligned} \tag{19.2}$$

and

$$\boldsymbol{\Omega}_{EDM}(\mathbf{r},\mathbf{P},t) = -\frac{q\,\eta}{2\,m\,c}\left[\mathbf{E} - \frac{\gamma}{1+\gamma}\,(\boldsymbol{\beta}\cdot\mathbf{E})\,\boldsymbol{\beta} + c\,\boldsymbol{\beta}\times\mathbf{B}\right] \tag{19.3}$$

Here $\mathbf{E}(\mathbf{r},t)$ and $\mathbf{B}(\mathbf{r},t)$ are the electric and magnetic fields, $\mathbf{B}_\perp$ and $\mathbf{B}_\parallel$ are the components perpendicular and parallel to the particle's momentum, γ is the particle's relativistic gamma factor, q, and m are the particle's charge and mass, $\boldsymbol{\beta}$ is the normalized velocity, $a = (g-2)/2$ is the particle's anomalous gyromagnetic moment (values given in Table 2.2), and η is the normalized electric dipole moment which is related to the dipole moment $\mathbf{d}$ via

$$\mathbf{d} = \frac{\eta}{2}\,\frac{q}{m\,c}\,\mathbf{S} \tag{19.4}$$

Note: Some authors define η without the factor of c is the denominator.

It is important to keep in mind that the a and g-factors used here are defined using Eq. (2.1) which, in the case of nuclei and other composite baryonic particles, differs from the conventional definition Eq. (2.3). See the discussion after Eq. (2.1).

19.2 Quaternion Representation of Spin Rotations

Bmad uses a quaternion representation for spin rotations. The following is a brief introduction to quaternions. For more information, the reader is referred to the Web. In particular, the `Quaternions and spatial rotation` article at `wikipedia.org`.

A quaternion $\mathbf{q}$ is a 4-vector $\mathbf{q} = (q_1, q_x, q_y, q_z)$ where the components are reals if they represent a rotation and are Taylor series if they represent a rotation map (§21.2). A quaternion can also be represented in the form:

$$\mathbf{q} = q_1 + q_x\,\mathbf{i} + q_y\,\mathbf{j} + q_z\mathbf{k} \tag{19.5}$$

where $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ are the *fundamental quaternion units* with the properties

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{i}\,\mathbf{j}\,\mathbf{k} = -1, \quad \mathbf{i}\,\mathbf{j} = \mathbf{k}, \quad \mathbf{j}\,\mathbf{i} = -\mathbf{k}, \quad \text{etc.} \tag{19.6}$$

$\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ do not commute among themselves but do commute with real numbers.

The q_1 component of a quaternion is called the `real` (or sometimes `scalar`) part and the other three components are called the `imaginary` (or sometimes `vector`) part.

When a quaternion represents a rotation, $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ can be thought of as representing unit vectors along the three Cartesian axes $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ respectively. A rotation through an angle θ around the unit axis $\mathbf{u} = (u_x, u_y, u_z)$ is represented by the quaternion

$$\mathbf{q} = \cos\frac{\theta}{2} + (u_x\,\mathbf{i} + u_y\,\mathbf{j} + u_z\,\mathbf{k})\sin\frac{\theta}{2} \tag{19.7}$$

A rotation quaternion $\mathbf{q}$ has normalization 1:

$$1 = ||\mathbf{q}|| = \sqrt{q_1^2 + q_x^2 + q_y^2 + q_z^2} \tag{19.8}$$

Such a quaternion is called a `unit` quaternion. The quaternion conjugate $\mathbf{q}^*$ is defined by

$$\mathbf{q}^* = q_1 - q_x\,\mathbf{i} - q_y\,\mathbf{j} - q_z\mathbf{k} \tag{19.9}$$

and the inverse $\mathbf{q}^{-1}$ is given by

$$\mathbf{q}^{-1} = \frac{\mathbf{q}^*}{||\mathbf{q}||^2} \tag{19.10}$$

Given an ordinary spatial vector $(\mathbf{r}_x, \mathbf{r}_y, \mathbf{r}_z)$, this vector is represented by a quaternion $\mathbf{r} = (0, \mathbf{r}_x, \mathbf{r}_y, \mathbf{r}_z)$. The rotation of $\mathbf{r}$ through a rotation represented by quaternion $\mathbf{q}$ to position $\mathbf{r}'$ is given by

$$\mathbf{r}' = \mathbf{q}\,\mathbf{r}\,\mathbf{q}^* \tag{19.11}$$

The transformation matrix corresponding to the transformation of Eq. (19.11) is

$$\mathbf{R} = \begin{pmatrix} q_1^2 + q_x^2 - q_y^2 - q_z^2 & 2\,q_x\,q_y - 2\,q_1\,q_z & 2\,q_x\,q_z + 2\,q_1\,q_y \\ 2\,q_x\,q_y + 2\,q_1\,q_z & q_1^2 - q_x^2 + q_y^2 - q_z^2 & 2\,q_y\,q_z - 2\,q_1\,q_x \\ 2\,q_x\,q_z - 2\,q_1\,q_y & 2\,q_y\,q_z + 2\,q_1\,q_x & q_1^2 - q_x^2 - q_y^2 + q_z^2 \end{pmatrix} \tag{19.12}$$

In quaternion notation, two rotations $\mathbf{q}_1$ and $\mathbf{q}_2$ are combined into one rotation using the formula

$$\mathbf{q}_{12} = \mathbf{q}_2\,\mathbf{q}_1 \tag{19.13}$$

where $\mathbf{q}_{12}$ is the result of rotation $\mathbf{q}_1$ followed by $\mathbf{q}_2$. Note that this equation is analogous to how rotation matrices are combined (for more information see [Quat]).

19.3 Invariant Spin Field

In a storage ring, the `invariant spin field` $\mathbf{n}(\mathbf{r}, s)$ [Hoff06, Duan15], at some phase space position $\mathbf{r} = (x, p_x, y, p_y, z, p_z)$ and at some point s in the ring, is the continuous function with unit amplitude that satisifies

$$\mathbf{n}(\mathcal{M}_r \mathbf{r}, s) = \mathcal{M}_s(\mathbf{r})\, \mathbf{n}(\mathbf{r}, s) \tag{19.14}$$

where $\mathcal{M}_r$ is the orbital part of the 1-turn transfer map and $\mathcal{M}_s(\mathbf{r})$ is the spin part of the map which is a function of $\mathbf{r}$. In general, it is not straightforward to calculate $\mathbf{n}$. The exceptional case (besides the cases where there is a resonance) is if the particle is on the closed orbit $\mathbf{r}_0$. In this case, since $\mathcal{M}_r \mathbf{r}_0 = \mathbf{r}_0$, and since M_s($\mathbf{r}$) is a rotation matrix, Eq. (19.14) can be solved to give the invariant spin field on the closed orbit which is denoted $\mathbf{n}_0$.

Once the invariant spin field has been calculated, various quantities of interest can be computed. For example, given some initial distribution of spins in a beam, the maximum possible time averaged polarization $\langle \mathbf{S} \rangle_{\max}$ is

$$\langle \mathbf{S} \rangle_{\max} = \int d\mathbf{r}\, \rho(\mathbf{r})\, \mathbf{n}(\mathbf{r}) \tag{19.15}$$

where the integral is over the beam phase space space density ρ and the longitudinal s-dependence is implicit. The above equation neglects any single spin polarization or depolarization processes. Notice that what is calculated is a time averaged quantity. Instantaneously, the beam can be fully polarized but the average over many turns, at some given position s, cannot exceed $\langle \mathbf{S} \rangle_{\max}$.

Another quantity that can be computed from knowledge of $\mathbf{n}$ is the equilibrium polarization of a beam $\mathbf{P}_{dk}$ using the Derbenev-Kondratenko-Mane formula [Barber99]:

$$\mathbf{P}_{dk}(s) = -\mathbf{n}(s)\, P_{dk} \tag{19.16}$$

$$P_{dk} = -\frac{8}{5\sqrt{3}} \frac{\oint ds \left\langle g^3\, \widehat{\mathbf{b}} \cdot \left(\mathbf{n} - \frac{\partial \mathbf{n}}{\partial \delta} \right) \right\rangle}{\oint ds \left\langle g^3 \left(1 - \frac{2}{9} (\mathbf{n} \cdot \widehat{\mathbf{s}})^2 + \frac{11}{18} \left| \frac{\partial \mathbf{n}}{\partial \delta} \right|^2 \right) \right\rangle}$$

where <> denotes an average over phase space, $g = 1/\rho$ is the bending strength (ρ is the bending radius), δ is the fractional energy deviation which, for ultra-relativistic particles, is the same as phase space p_z, $\widehat{\mathbf{s}}$ is the unit vector in the direction of motion, and $\widehat{\mathbf{b}}$ is defined to be

$$\widehat{\mathbf{b}} \equiv \frac{\widehat{\mathbf{s}} \times d\widehat{\mathbf{s}}/ds}{d\widehat{\mathbf{s}}/ds} \tag{19.17}$$

Notice that $\widehat{\mathbf{b}}$ is the direction of the magnetic field when $\widehat{\mathbf{s}}$ is perpendicular to the magnetic field and when there is no electric field.

The time dependence of the polarization is [Barber99])

$$\mathbf{P}(t) = \mathbf{P}_{dk}\, (1 - \exp -t/\tau_{dk}) + \mathbf{P}_0\, \exp -t/\tau_{dk} \tag{19.18}$$

where $\mathbf{P}_0$ is the initial polarization and the polarization rate τ_{dk}^{-1} is

$$\tau_{dk}^{-1} = \frac{5\sqrt{3}}{8} \frac{r_e\, \gamma^5\, \hbar}{m} \frac{1}{C} \oint ds \left\langle g^3 \left(1 - \frac{2}{9} (\mathbf{n} \cdot \widehat{\mathbf{s}})^2 + \frac{11}{18} \left| \frac{\partial \mathbf{n}}{\partial \delta} \right|^2 \right) \right\rangle \tag{19.19}$$

τ_{dk}^{-1} can be decomposed into two parts:

$$\tau_{dk}^{-1} = \tau_{st}^{-1} + \tau_{dep}^{-1} \tag{19.20}$$

where τ_{st}^{-1} is the Sokolov-Ternov poarization rate and the depolarization rate τ_{dep}^{-1} is given by the formula

$$\tau_{dk}^{-1} = \frac{5\sqrt{3}}{8}\frac{r_e\,\gamma^5\,\hbar}{m}\frac{1}{C}\oint ds\,\left\langle g^3\,\frac{11}{18}\left|\frac{\partial \mathbf{n}}{\partial \delta}\right|^2\right\rangle \tag{19.21}$$

19.4 SLIM Formalism

The `SLIM` formalism[Duan15, Barber99], indroduced by Alex Chao, is a way to represent the first order orbital + spin transport as an 8×8 matrix which then can be analyzed. The idea is to expand the transport map around the closed orbit $(\mathbf{r}_0, \mathbf{n}_0)$ where $\mathbf{r}_0$ is the orbital closed orbit and $\mathbf{n}_0$ is the invariant spin field on the closed orbit. The spin coordinates are expressed using coordinates $(\mathbf{l}_0(s), \mathbf{n}_0(s), \mathbf{m}_0(s))$[1], where $\mathbf{l}_0$ and $\mathbf{m}_0$ are axes constructed to be periodic in s and transverse to $\mathbf{n}_0$, with $(\mathbf{l}_0(s), \mathbf{n}_0(s), \mathbf{m}_0(s))$ forming a right hand coordinate system.

The variation of the spin component along the $\mathbf{n}_0$ axis will be second order and therefore, to first order, can be ignored. Thus the first order map in the $(\mathbf{r}_0, (\mathbf{l}_0, \mathbf{m}_0))$ coordinate system between two any points s_1 and s_2 is an 8×8 matrix $\widetilde{\mathbf{M}}$ which is written in the form

$$\widetilde{\mathbf{M}}(s1, s2) = \begin{pmatrix} \mathbf{M}_{6\times 6} & \mathbf{0}_{6\times 2} \\ \mathbf{G}_{2\times 6} & \mathbf{D}_{2\times 2} \end{pmatrix} \tag{19.22}$$

Where $\mathbf{M}$ is the 6×6 orbital transport matrix, $\mathbf{G}$ represents the coupling between orbital and spin coordinates, and $\mathbf{D}$ is the 2×2 rotation matrix for the spin transport of a particle on the closed orbit. The upper right block $\mathbf{0}_{6\times 2}$ in the $\widetilde{\mathbf{M}}$ matrix is zero since Stern-Gerlach effects are ignored. If $\widetilde{\mathbf{M}}$ is the 1-turn matrix ($s_2 = s_1$), the phase advance of $\mathbf{D}$ is the spin tune.

To compute $\widetilde{\mathbf{M}}$, the first step is to calculate the $(\mathbf{l}_0, \mathbf{n}_0, \mathbf{m}_0)$ coordinate system. If the entire ring is being analyzed, $\mathbf{n}_0$ at some starting point can be calculated (§19.3). If only part of the ring is being analyzed, the orientation of $\mathbf{n}_0$ at the start of the section will be an input parameter (that is, it is given by the User and not calculated). After $\mathbf{n}_0$ is known at some s-position, $\mathbf{l}_0$ and $\mathbf{m}_0$ at that s-position can be choisen somewhat arbitrarily to form a right handed coordinate system.

After the $(\mathbf{l}_0, \mathbf{n}_0, \mathbf{m}_0)$ coordinates have been calculated at some initial point, the axes can be transported using the $\mathbf{q}_0$ quaternion. When analyzing only a section of a ring, there is no identifiable spin tune so nothing further needs to be done. In this case, the $\mathbf{D}$ matrix is just a unit matrix. When analyzing an entire ring, it is sometimes desireable to rotate the $\mathbf{l}_0(s)$ and $\mathbf{m}_0(s)$ axes to give a uniform phase advance as a function of s. When analyzing one-turn maps, it is generally not necessary to rotate the $\mathbf{l}_0(s)$ and $\mathbf{m}_0(s)$ axes. In this case the spin phase advance as a function of s will be zero except just before the starting position where there will be a discontinuous jump in phase.

Once the $(\mathbf{l}_0, \mathbf{n}_0, \mathbf{m}_0)$ coordinates have been calculated, the matrices $\mathbf{G}$ and $\mathbf{D}$ can be calculated from the spin transport map (which *Bmad* calculates via PTC (§34)). To first order, the spin transport map $\mathbf{q}_s$, represented as a quaternion, can be written in the form

$$\mathbf{q}_s = \mathbf{q}_0 + \sum_{i=1}^{6} r_i\,\mathbf{q}_i \tag{19.23}$$

The quaternion $\mathbf{q}_0$ is the zeroth order part of the map and the six quaternions $\mathbf{q}_i$, $i = 1, \ldots, 6$ are the first order part with $\mathbf{r}$ being the orbital phase space point the map is being evaluated at. Let $\mathbf{q}_{lnm}(s)$

[1] Different authors will use different conventions for the ordering of $\mathbf{l}_0$, $\mathbf{n}_0$, and $\mathbf{m}_0(s)$. The ordering used here reflects the fact that in many rings the $\mathbf{n}_0$ axis will point in the vertical y-direction.

be the quaternion that transforms from $(\mathbf{l}_0, \mathbf{n}_0, \mathbf{m}_0)$ coordinates to (x, y, z) coordinates at a given point s. With this, the spin transport $\widehat{\mathbf{q}}$ from s_1 to s_2 in the $(\mathbf{l}_0, \mathbf{n}_0, \mathbf{m}_0)$ coordinate system is

$$\widehat{\mathbf{q}}_s(s_1, s_2) = \mathbf{q}_{lnm}(s_2)\, \mathbf{q}_s(s_1, s_2)\, \mathbf{q}_{lnm}^{-1}(s_1) \tag{19.24}$$

The zeroth order part of this map

$$\widehat{\mathbf{q}}_0(s_1, s_2) = \mathbf{q}_{lnm}(s_2)\, \mathbf{q}_0(s_1, s_2)\, \mathbf{q}_{lnm}^{-1}(s_1) \tag{19.25}$$

represents a rotation around the $\mathbf{n}_0$ axis. Converting this to a rotation matrix $\mathbf{R}_0$ via Eq. (19.12)) the $\mathbf{D}$ matrix is then

$$\mathbf{D}(s_1, s_2) = \begin{pmatrix} R_0(1,1) & R_0(1,3) \\ R_0(3,1) & R_0(3,3) \end{pmatrix} \tag{19.26}$$

The $\mathbf{G}$ matrix is calulated from the first order part of $\widehat{\mathbf{q}}_s$

$$\widehat{\mathbf{q}}_i = \mathbf{q}_{lnm}(s_2)\, \mathbf{q}_i(s_1, s_2)\, \mathbf{q}_{lnm}^{-1}(s_1) \tag{19.27}$$

Using Eq. (19.23) in Eq. (19.12) and keeping only first order terms gives

$$\begin{aligned} \mathbf{G}(1,j) &= 2(\widehat{q}_{0,x}\, \widehat{q}_{j,y} + \widehat{q}_{0,y}\, \widehat{q}_{j,x} + \widehat{q}_{0,1}\, \widehat{q}_{j,z} + \widehat{q}_{0,z}\, \widehat{q}_{j,1}) \\ \mathbf{G}(2,j) &= 2(\widehat{q}_{0,1}\, \widehat{q}_{j,x} + \widehat{q}_{0,x}\, \widehat{q}_{j,1} + \widehat{q}_{0,y}\, \widehat{q}_{j,z} + \widehat{q}_{0,z}\, \widehat{q}_{j,y}), \quad j = 1, \ldots, 6 \end{aligned} \tag{19.28}$$

[Do not be confused by the x, y and z subscripts which refer to the components of $\widehat{q}$ as definded in Eq. (19.5). $\widehat{q}$ rotates spins in the $(\mathbf{l}_0, \mathbf{n}_0, \mathbf{m}_0)$ coordinate system. Not the (x, y, z) coordinate system.]

Lattice design consists of minimizing elements of the $\mathbf{G}$ matrix since that will minimize $\partial\mathbf{n}/\partial\delta$.

Calculation of $\partial\mathbf{n}/\partial\delta$ from the 8×8 matrix $\widetilde{\mathbf{M}}$ (Eq. (19.22)) is given by [Barber99]. The eigenvectors $\mathbf{q}_j$ and eigenvalues λ_j of $\widetilde{\mathbf{M}}$, $j = 1, \ldots 8$, are first computed. These eigenvectors are of the form

$$\begin{aligned} \mathbf{q}_j &= \begin{pmatrix} \mathbf{v}_j \\ \mathbf{w}_j \end{pmatrix}, \quad j = 1, \ldots, 6 \\ \mathbf{q}_j &= \begin{pmatrix} \mathbf{0}_6 \\ \mathbf{w}_j \end{pmatrix}, \quad j = 7, 8 \end{aligned} \tag{19.29}$$

where $\mathbf{v}_j$ are eigenvectors of the orbital submatrix $\mathbf{M}$, and the $\mathbf{w}_j$ are computed via

$$\mathbf{w}_j = -\left[\mathbf{D} - \lambda_j\right]^{-1} \mathbf{G}\, \mathbf{v}_j \tag{19.30}$$

With this, $\partial\mathbf{n}/\partial\delta$ is computed via

$$\frac{\partial \mathbf{n}}{\partial \delta} = \sum_{j=1}^{6} A_j\, \mathbf{w}_j \tag{19.31}$$

where the A_j are computed by inverting the equation

$$[0, 0, 0, 0, 0, 1]^t = \sum_{j=1}^{6} A_j\, \mathbf{v}_j \tag{19.32}$$

where the superscript t means transpose.

An alternative to using Eqs. (19.31) and (19.32) that is given in the literature is to first normalize the eigenvectors via [Chao79]

$$\mathbf{v}_j^{*t}\, \mathbf{S} \mathbf{v}_j = i, \quad j = 1, 3, 5 \tag{19.33}$$

where the eigenvectors have been arranged in complex conjugate pairs with $\mathbf{v}_{j+1} = \mathbf{v}_j^*$, $j = 1, 3, 5$ with the superscript $*$ denoting the complex conjugate. In Eq. (19.33) $\mathbf{S}$ is the matrix

$$\mathbf{S} = \begin{pmatrix} 0 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{19.34}$$

With this, $\partial \mathbf{n}/\partial \delta$ is computed via

$$\frac{\partial \mathbf{n}}{\partial \delta} = i \sum_{j=1}^{6} \mathbf{v}_{j5}^* \, \mathbf{w}_j \tag{19.35}$$

From a computational standpoint, the advantage of Eq. (19.32) over Eq. (19.35) is that Eq. (19.32) can be used even if there are no synchrotron oscillations (no RF cavities) where the energy is a constant of the motion and the $\mathbf{M}$ orbital matrix has degenerate eigenvectors which prevent the normalization condition Eq. (19.33) from being used.

19.5 Spinor Notation

The following descibes the old spinor representation used by *Bmad* to represent spins. This documentation is kept as an aid for comparison with the spin tracking literature.

In the SU(2) representation, a spin $\mathbf{s}$ is written as a spinor $\Psi = (\psi_1, \psi_2)^T$ where $\psi_{1,2}$ are complex numbers. The conversion between SU(2) and SO(3) is

$$\mathbf{S} = \Psi^\dagger \boldsymbol{\sigma} \, \Psi \qquad \longleftrightarrow \qquad \Psi = \frac{e^{i\xi}}{\sqrt{2\,(P + s_3)}} \begin{pmatrix} P + s_3 \\ s_1 + i s_2 \end{pmatrix} \tag{19.36}$$

Where ξ is an unmeasureable phase factor, and P is the polarization. $P = 1$ for a single particle. Also $\boldsymbol{\sigma} = (\sigma_x, \sigma_y, \sigma_z)$ are the three Pauli matrices

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \qquad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{19.37}$$

In polar coordinates

$$\Psi = \begin{pmatrix} \psi_1 \\ \psi_2 \end{pmatrix} = \sqrt{P}\, e^{i\xi} \begin{pmatrix} \cos\frac{\theta}{2} \\ e^{i\phi} \sin\frac{\theta}{2} \end{pmatrix} \qquad \longleftrightarrow \qquad \mathbf{S} = P \begin{pmatrix} \sin\theta\cos\phi \\ \sin\theta\sin\phi \\ \cos\theta \end{pmatrix} \tag{19.38}$$

Due to the unitarity of the spin vector, $|\psi_1|^2 + |\psi_2|^2 = P$. The spinor eigenvectors along the x, y and z axes are

$$\begin{aligned} \Psi_{x+} &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}, & \Psi_{x-} &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \\ \Psi_{y+} &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ i \end{pmatrix}, & \Psi_{y-} &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -i \end{pmatrix}, \\ \Psi_{z+} &= \begin{pmatrix} 1 \\ 0 \end{pmatrix}, & \Psi_{z-} &= \begin{pmatrix} 0 \\ -1 \end{pmatrix}. \end{aligned} \tag{19.39}$$

In spinor notation, the T-BMT equation can be written as

$$\frac{\mathrm{d}}{\mathrm{d}t}\Psi = -\frac{i}{2}\left(\boldsymbol{\sigma}\cdot\boldsymbol{\Omega}\right)\Psi = -\frac{i}{2}\begin{pmatrix}\Omega_z & \Omega_x - i\,\Omega_y\\ \Omega_x + i\,\Omega_y & -\Omega_z\end{pmatrix}\Psi \tag{19.40}$$

The solution leads to a rotation of the spin vector by an angle α around a unit vector $\widehat{\mathbf{n}}$ represented as

$$\begin{aligned}\Psi_f &= \exp\left[-i\frac{\alpha}{2}\widehat{\mathbf{n}}\cdot\boldsymbol{\sigma}\right]\Psi_i\\ &= \left[\cos\left(\frac{\alpha}{2}\right)\mathbf{1}_2 - i\left(\widehat{\mathbf{n}}\cdot\boldsymbol{\sigma}\right)\sin\left(\frac{\alpha}{2}\right)\right]\Psi_i\\ &= \mathbf{A}\Psi_i.\end{aligned} \tag{19.41}$$

where Ψ_i is the initial spin state, Ψ_f is the final spin state, and $\mathbf{A}$, which describes the spin transport, is the SU(2) matrix representation of the quaternian $(a_0, \mathbf{a}) = (\cos(\alpha/2), -\sin(\alpha/2)\,\widehat{\mathbf{n}})$. $\mathbf{A}$ has the normalization condition $a_0^2 + \boldsymbol{a}^2 = 1$. Thus the three components $\boldsymbol{a} = (a_1, a_2, a_3)$ completely describe $\mathbf{A}$.

With spinors, the matrix representation of the observable $S_{\mathbf{u}}$ corresponding to the measurement of the spin along the unit vector $\mathbf{u}$ is

$$S_{\mathbf{u}} \equiv \frac{\hbar}{2}\,\boldsymbol{\sigma}\cdot\mathbf{u} \tag{19.42}$$

$$= \frac{\hbar}{2}\begin{pmatrix}u_z & u_x - i\,u_y\\ u_x + i\,u_y & u_z\end{pmatrix} \tag{19.43}$$

The expectation value of this operator, $\Psi^\dagger\,\mathbf{S}_u\,\Psi$, representing the spin of a particle, satisfies the equation of motion of a classical spin vector in the particle's instantaneous rest frame.

For a distribution of spins, the polarization P_s along the unit vector $\mathbf{u}$ is defined as the absolute value of the average expectation value of the spin over all N particles times $\frac{2}{\hbar}$,

$$P_s = \frac{2}{\hbar}\frac{1}{N}\sum_{j=1}^{N}\Psi_j^\dagger S_{\mathbf{u}}\Psi_j \tag{19.44}$$

See § §16.5 for formulas for tracking a spin through a multipole fring field.

Chapter 20

Linear Optics

20.1 Coupling and Normal Modes

The coupling formalism used by *Bmad* is taken from the paper of Sagan and Rubin[Sagan99]. The main equations are reproduced here with the notation change that $\mathbf{A}$ and $\mathbf{B}$ is replaced by $\mathbf{Q}$ and $\mathbf{W}$[1]

The analysis starts with the map $\mathbf{T}(s)$ for the transverse two–dimensional phase space coordinates $\mathbf{x} = (x, x', y, y')$. In ring, with a closed geometry, this map will be a one-turn map starting and ending at some point s. For a machine with open geometry, $\mathbf{T}(0)$ can be computed from the initial Twiss and coupling parameters and $\mathbf{T}(s)$ can then be computed by propagating with the transfer map $\mathbf{M}_{0s}$ from 0 to s:

$$\mathbf{T}(s) = \mathbf{M}_{0s}\,\mathbf{T}(0)\,\mathbf{M}_{0s}^{-1} \tag{20.1}$$

$\mathbf{T}$ can be decomposed using a similarity transformation can be written as

$$\mathbf{T} = \mathbf{V}\,\mathbf{U}\,\mathbf{V}^{-1}, \tag{20.2}$$

where $\mathbf{V}$ is symplectic, and $\mathbf{U}$ is of the form

$$\mathbf{U} = \begin{pmatrix} \mathbf{Q} & \mathbf{0} \\ \mathbf{0} & \mathbf{W} \end{pmatrix}. \tag{20.3}$$

Since $\mathbf{U}$ is uncoupled the standard Twiss analysis can be performed on the matrices $\mathbf{Q}$ and $\mathbf{W}$. The normal modes are labeled q and w and if the one–turn matrix $\mathbf{T}$ is uncoupled then w corresponds to the horizontal mode and w corresponds to the vertical mode.

The $\mathbf{Q}$ and $\mathbf{W}$ matrices can be parameterized using the standard 1-dimensional Twiss parameterization:

$$\mathbf{Q} = \begin{pmatrix} \cos\theta_q + \alpha_q \sin\theta_q & \beta_q \sin\theta_q \\ -\gamma_q \sin\theta_q & \cos\theta_q - \alpha_q \sin\theta_q \end{pmatrix} \tag{20.4}$$

and $\mathbf{V}$ is written in the form

$$\mathbf{V} = \begin{pmatrix} \gamma\mathbf{I} & \mathbf{C} \\ -\mathbf{C}^{+} & \gamma\mathbf{I} \end{pmatrix}, \tag{20.5}$$

[1] The reason why q and w are used here to denote the modes is to avoid confusion with the *Bmad* convention of using the labels a and b to represent the same modes throughout the lattice. At the start of the lattice, by convention, the a mode is the same as the q mode and the b mode is the same as the w mode. If there is a "mode flip" (see Sagan and Rubin for an explanation of this term) at some spot in the lattice, the a mode before the mode flip will be the same physical mode after the mode flip (and similarly for the b mode. That is, after the mode flip the a mode will be the same as the w mode and the b mode will be the same as the q mode.

where $\mathbf{C}$ is a 2x2 matrix and + superscript denotes the symplectic conjugate:

$$\mathbf{C}^{+} = \begin{pmatrix} C_{22} & -C_{12} \\ -C_{21} & C_{11} \end{pmatrix}. \tag{20.6}$$

Since we demand that $\mathbf{V}$ be symplectic we have the condition

$$\gamma^2 + ||\mathbf{C}|| = 1, \tag{20.7}$$

and $\mathbf{V}^{-1}$ is given by

$$\mathbf{V}^{-1} = \begin{pmatrix} \gamma\mathbf{I} & -\mathbf{C} \\ \mathbf{C}^{+} & \gamma\mathbf{I} \end{pmatrix}. \tag{20.8}$$

$\mathbf{C}$ is a measure of the coupling. $\mathbf{T}$ is uncoupled if and only if $\mathbf{C} = \mathbf{0}$.

It is useful to normalize out the $\beta(s)$ variation in the above analysis. Normalized quantities being denoted by a bar above them. The normalized normal mode matrix $\overline{\mathbf{U}}$ is defined by

$$\overline{\mathbf{U}} = \mathbf{G}\,\mathbf{U}\,\mathbf{G}^{-1}, \tag{20.9}$$

Where $\mathbf{G}$ is given by

$$\mathbf{G} \equiv \begin{pmatrix} \mathbf{G}_q & \mathbf{0} \\ \mathbf{0} & \mathbf{G}_w \end{pmatrix}, \tag{20.10}$$

with

$$\mathbf{G}_q = \begin{pmatrix} \frac{1}{\sqrt{\beta_q}} & 0 \\ \frac{\alpha_q}{\sqrt{\beta_q}} & \sqrt{\beta_q} \end{pmatrix}, \tag{20.11}$$

with a similar equation for $\mathbf{G}_w$. With this definition, the corresponding $\overline{\mathbf{Q}}$ and $\overline{\mathbf{W}}$ (cf. Eq. (20.3)) are just rotation matrices. The relationship between $\mathbf{T}$ and $\overline{\mathbf{U}}$ is

$$\mathbf{T} = \mathbf{G}^{-1}\,\overline{\mathbf{V}}\,\overline{\mathbf{U}}\,\overline{\mathbf{V}}^{-1}\,\mathbf{G}, \tag{20.12}$$

where

$$\overline{\mathbf{V}} = \mathbf{G}\,\mathbf{V}\,\mathbf{G}^{-1}. \tag{20.13}$$

Using Eq. (20.10), $\overline{\mathbf{V}}$ can be written in the form

$$\overline{\mathbf{V}} = \begin{pmatrix} \gamma\mathbf{I} & \overline{\mathbf{C}} \\ -\overline{\mathbf{C}}^{+} & \gamma\mathbf{I} \end{pmatrix}, \tag{20.14}$$

with the normalized matrix $\overline{\mathbf{C}}$ given by

$$\overline{\mathbf{C}} = \mathbf{G}_q\,\mathbf{C}\,\mathbf{G}_w^{-1}. \tag{20.15}$$

The normal mode coordinates $\mathbf{q} = (q, q', w, w')$ are related to the laboratory frame via

$$\mathbf{q} = \mathbf{V}^{-1}\,\mathbf{x}. \tag{20.16}$$

In particular the normal mode dispersion $\boldsymbol{\eta}_q = (\eta_q, \eta'_q, \eta_w, \eta'_w)$ is related to the laboratory frame dispersion $\boldsymbol{\eta}_x = (\eta_x, \eta'_x, \eta_y, \eta'_y)$ via

$$\boldsymbol{\eta}_q = \mathbf{V}^{-1}\,\boldsymbol{\eta}_x. \tag{20.17}$$

When there is no coupling ($\mathbf{C} = 0$), $\boldsymbol{\eta}_q$ and $\boldsymbol{\eta}_x$ are equal to each other.

20.2 Dispersion Calculation

The dispersion (η) and the dispersion derivative (η') are defined by the equations

$$\begin{aligned} \eta_x(s) &\equiv \left.\frac{dx}{dp_z}\right|_s, \qquad \eta'_x(s) \equiv \left.\frac{d\eta_x}{ds}\right|_s = \left.\frac{dx'}{dp_z}\right|_s \\ \eta_y(s) &\equiv \left.\frac{dy}{dp_z}\right|_s, \qquad \eta'_y(s) \equiv \left.\frac{d\eta_y}{ds}\right|_s = \left.\frac{dy'}{dp_z}\right|_s \\ \eta_z(s) &\equiv \left.\frac{dz}{dp_z}\right|_s \end{aligned} \tag{20.18}$$

Given the dispersion at a given point, the dispersion at some other point is calculated as follows: Let $\mathbf{r} = (x, p_x, y, p_y, z, p_z)$ be the reference orbit, around which the dispersion is to be calculated. Let $\mathbf{V}$ and $\mathbf{M}$ be the zeroth and first order components of the transfer map between two points labeled 1 and 2:

$$\mathbf{r}_2 = \mathbf{M}\,\mathbf{r}_1 + \mathbf{V} \tag{20.19}$$

Define the dispersion vector $\boldsymbol{\eta}$ by

$$\boldsymbol{\eta} = \left(\eta_x, \eta'_x\,(1+p_z), \eta_y, \eta'_y\,(1+p_z), \eta_z, 1\right) \tag{20.20}$$

Differentiating Eq. (20.19) with respect to energy, the dispersion at point 2 in terms of the dispersion at point 1 is

$$\boldsymbol{\eta}_2 = \left[\frac{dp_{z2}}{dp_{z1}}\right]^{-1} [\mathbf{M}\,\boldsymbol{\eta}_1] + \mathbf{V}_\eta \tag{20.21}$$

where

$$\mathbf{V}_\eta = \left[\frac{dp_{z2}}{dp_{z1}}\right]^{-1} \frac{1}{1+p_{z1}} \begin{pmatrix} M_{12}\,p_{x1} + M_{14}\,p_{y1} \\ M_{22}\,p_{x1} + M_{24}\,p_{y1} \\ M_{32}\,p_{x1} + M_{34}\,p_{y1} \\ M_{42}\,p_{x1} + M_{44}\,p_{y1} \\ M_{52}\,p_{x1} + M_{54}\,p_{y1} \\ M_{62}\,p_{x1} + M_{64}\,p_{y1} \end{pmatrix} - \begin{pmatrix} 0 \\ \frac{p_{x2}}{1+p_{z2}} \\ 0 \\ \frac{p_{y2}}{1+p_{z2}} \\ 0 \\ 0 \end{pmatrix} \tag{20.22}$$

The sixth row of the matrix equation gives dp_{z1}/dp_{z2}. Explicitly

$$\frac{dp_{z2}}{dp_{z1}} = \sum_{i=1}^{6} M_{6i}\,\eta_{1i} + \frac{M_{62}\,p_{x1} + M_{64}\,p_{y1}}{1+p_{z1}} \tag{20.23}$$

For everything except `RFcavity` and `Lcavity` elements, dp_{z2}/dp_{z1} is 1.

For a non-circular machine, there are two ways one can imagine defining the dispersion: Either with respect to changes in energy at the beginning of the machine or with respect to the local change in energy at the point of measurement. The former definition will be called "non-local dispersion" and the latter definition will be called "local dispersion". For a circular machine, local dispersion is always used. The dispersion defined in the above equations, which is what *Bmad* uses in calculations, is the local dispersion. The non-local dispersion $\widetilde{\boldsymbol{\eta}}(s_1)$ at some point s_1 is related to the local dispersion $\boldsymbol{\eta}(s_1)$ via

$$\widetilde{\boldsymbol{\eta}}(s_1) = \frac{dp_{z1}}{dp_{z0}}\,\boldsymbol{\eta}(s_1) \tag{20.24}$$

where s_0 is the beginning of the machine.

For a non-circular machine, there are advantages and disadvantages to using either local or non-local dispersion. Local dispersion has the problem that dp_{z2}/dp_{z1} in Eq. (20.21) may go through zero at a point producing infinite dispersions at that point. The non-local dispersion has the merit of reflecting what one would measure if the starting energy of the beam is veried. The local dispersion, on the other hand, reflects the correlations between the particle energy and particle position within a beam.

Chapter 21

Taylor Maps

21.1 Taylor Maps

A transport map $\mathcal{M} : \mathcal{R}^6 \to \mathcal{R}^6$ through an element or a section of a lattice is a function that maps the starting phase space coordinates $\mathbf{r}(\text{in})$ to the ending coordinates $\mathbf{r}(\text{out})$

$$\mathbf{r}(\text{out}) = \mathcal{M}\left(\delta\mathbf{r}\right) \tag{21.1}$$

where

$$\delta\mathbf{r} = \mathbf{r}(\text{in}) - \mathbf{r}_{\text{ref}} \tag{21.2}$$

$\mathbf{r}_{\text{ref}}$ is the reference orbit at the start of the map around which the map is made. In many cases the reference orbit is the zero orbit. For a storage ring, the closed orbit is commanly used for the reference orbit.

$\mathcal{M}$ in the above equation is made up of six functions $\mathcal{M}_i : \mathcal{R}^6 \to \mathcal{R}$. Each of these functions maps to one of the $r(\text{out})$ coordinates. Each of these functions can be expanded in a Taylor series and truncated at some order. Each Taylor series is in the form

$$r_i(\text{out}) = \sum_{j=1}^{N} C_{ij} \prod_{k=1}^{6} \left(\delta r_k\right)^{e_{ijk}} \tag{21.3}$$

Where the C_{ij} are coefficients and the e_{ijk} are integer exponents. The order of a given term associated with index i, j is the sum over the exponents

$$\text{order}_{ij} = \sum_{k=1}^{6} e_{ijk} \tag{21.4}$$

The order of the entire map is the order at which the map is truncated.

The standard *Bmad* routine for printing a Taylor map might produce something like this:

```
Taylor Terms:
Out      Coef              Exponents           Order       Reference
-------------------------------------------------
 1:     -0.600000000000   0  0  0  0  0  0       0       0.200000000
 1:      1.000000000000   1  0  0  0  0  0       1
 1:      0.145000000000   2  0  0  0  0  0       2
```

```
-------------------------------------------------
 2:     -0.185000000000   0  0  0  0  0  0       0       0.000000000
 2:      1.300000000000   0  1  0  0  0  0       1
 2:      3.800000000000   2  0  0  0  0  1       3
-------------------------------------------------
 3:      1.000000000000   0  0  1  0  0  0       1       0.100000000
 3:      1.600000000000   0  0  0  1  0  0       1
 3:    -11.138187077310   1  0  1  0  0  0       2
-------------------------------------------------
 4:      1.000000000000   0  0  0  1  0  0       1       0.000000000
-------------------------------------------------
 5:      0.000000000000   0  0  0  0  0  0       0       0.000000000
 5:      0.000001480008   0  1  0  0  0  0       1
 5:      1.000000000000   0  0  0  0  1  0       1
 5:      0.000000000003   0  0  0  0  0  1       1
 5:      0.000000000003   2  0  0  0  0  0       2
-------------------------------------------------
 6:      1.000000000000   0  0  0  0  0  1       1       0.000000000
```

Each line in the example represents a single `Taylor term`. The Taylor terms are grouped into 6 `Taylor series`. There is one series for each of the output phase space coordinate. The first column in the example, labeled "out", (corresponding to the i index in Eq. (21.3)) indicates the Taylor series: $1 = x(out)$, $2 = p_x(out)$, etc. The 6 exponent columns give the e_{ijk} of Eq. (21.3). In this example, the second Taylor series (`out` = 2), when expressed as a formula, would read:

$$p_x(out) = -0.185 + 1.3\,\delta p_x + 3.8\,\delta x^2\,\delta p_z \tag{21.5}$$

The reference column in the above example shows the input coordinates around which the Taylor map is calculated. In this case, the reference coordinates where

$$(x, p_x, y, p_y, z, p_z)_{ref} = (0.2, 0, 0.1, 0, 0, 0, 0) \tag{21.6}$$

The choice of the reference point will affect the values of the coefficients of the Taylor map. As an example, consider the 1-dimension map

$$x(out) = A\,\sin(k\,\delta x) \tag{21.7}$$

Then a Taylor map to 1^{st} order is

$$x(out) = c_0 + c_1\,\delta x \tag{21.8}$$

where

$$c_1 = A\,k\,\cos(k\,x_{\rm ref}) \tag{21.9}$$
$$c_0 = A\,\sin(k\,x_{\rm ref}) \tag{21.10}$$

21.2 Spin Taylor Map

A Taylor map that fully describes spin (§19.1) and orbital motion, would consist of nine Taylor series (six for the orbital phase space variables and three for the spin components) and each Taylor series would be a polynomial in nine variables.

To simplify things, *Bmad* assumes that the effect on the orbital phase space due to the spin orientation is negligible. That is, Stern-Gerlach effects are ignored. With this assumption, the orbital part of

the map is only dependent on the six orbital variables. This means that $\mathbf{\Omega}_{BMT}$ and $\mathbf{\Omega}_{EDM}$ in the Thomas-Bargmann-Michel-Telegdi equation (§19.1), are assumed independent of the spin. Thus the spin transport is just a rotation. *Bmad* represents this rotation using a quaternion (§19.2). Each of the four components of the quaternion is a Taylor series and the full phase space plus spin transport uses 10 (= 6 orbital + 4 spin) Taylor series with each Taylor series only being dependent on the six orbital phase space coordinates.

Spin transport involves:

1. Using the six orbital coordinates, evaluate the four spin Taylor series to produce a quaternion $\mathbf{q}$.
2. Normalize the quaternion to one: $\mathbf{q} \longrightarrow \mathbf{q}/|\mathbf{q}|$.
3. Rotate the spin vector: $\mathbf{S} \longrightarrow \mathbf{q}\,\mathbf{S}\,\mathbf{q}^{-1}$.

The normalization of the quaternion is needed since the truncation of the map to a finite order will produce errors in the magnitude of the quaternion.

The standard *Bmad* routine for printing a spin Taylor map will produce a result that is very similar as that produced for the orbital phase space. The difference is that there will only be four Taylor series labeled (`S1`, `Sx`, `Sy`, `Sz`) for the four components of the quaternion. Also the reference orbit will not be shown (it is exactly the same as the orbital phase space reference orbit).

Note: When tracking a particle's spin through a map, the quaternion used to rotate the spin is always normalized to one so that the magnitude of the spin will be invariant.

21.3 Symplectification

If the evolution of a system can be described using a Hamiltonian then it can be shown that the linear part of any transport map (the Jacobian) must obey the symplectic condition. If a matrix $\mathbf{M}$ is not symplectic, Healy[Healy86] has provided an elegant method for finding a symplectic matrix that is "close" to $\mathbf{M}$. The procedure is as follows: From $\mathbf{M}$ a matrix $\mathbf{V}$ is formed via

$$\mathbf{V} = \mathbf{S}(\mathbf{I} - \mathbf{M})(\mathbf{I} + \mathbf{M})^{-1} \tag{21.11}$$

where $\mathbf{S}$ is the matrix

$$\mathbf{S} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix} \tag{21.12}$$

$\mathbf{V}$ is symmetric if and only if $\mathbf{M}$ is symplectic. In any case, a symmetric matrix $\mathbf{W}$ near $\mathbf{V}$ can be formed via

$$\mathbf{W} = \frac{\mathbf{V} + \mathbf{V}^t}{2} \tag{21.13}$$

A symplectic matrix $\mathbf{F}$ is now obtained by inverting (21.11)

$$\mathbf{F} = (\mathbf{I} + \mathbf{S}\mathbf{W})(\mathbf{I} - \mathbf{S}\mathbf{W})^{-1} \tag{21.14}$$

21.4 Map Concatenation and Feed-Down

Of importance in working with Taylor maps is the concept of `feed-down`. This is best explained with an example. To keep the example simple, the discussion is limited to one phase space dimension so that

the Taylor maps are a single Taylor series. Take the map M_1 from point 0 to point 1 to be

$$M_1 : x_1 = x_0 + 2 \tag{21.15}$$

and the map M_2 from point 1 to point 2 to be

$$M_2 : x_2 = x_1^2 + 3\,x_1 \tag{21.16}$$

Then concatenating the maps to form the map M_3 from point 0 to point 2 gives

$$M_3 : x_2 = (x_0 + 2)^2 + 3(x_0 + 2) = x_0^2 + 7\,x_0 + 10 \tag{21.17}$$

However if we are evaluating our maps to only 1^{st} order the map M_2 becomes

$$M_2 : x_2 = 3\,x_1 \tag{21.18}$$

and concatenating the maps now gives

$$M_3 : x_2 = 3(x_0 + 2) = 3\,x_0 + 6 \tag{21.19}$$

Comparing this to Eq. (21.17) shows that by neglecting the 2^{nd} order term in Eq. (21.16) leads to 0^{th} and 1^{st} order errors in Eq. (21.19). These errors can be traced to the finite 0^{th} order term in Eq. (21.15). This is the principal of feed–down: Given M_3 which is a map produced from the concatenation of two other maps, M_1, and M_2

$$M_3 = M_2(M_1) \tag{21.20}$$

Then if M_1 and M_2 are correct to nth order, M_3 will also be correct to nth order as long as M_1 has no constant (0^{th} order) term. [Notice that a constant term in M_2 does not affect the argument.] What happens if we know there are constant terms in our maps? One possibility is to go to a coordinate system where the constant terms vanish. In the above example that would mean using the coordinate $\widetilde{x}_0$ at point 0 given by

$$\widetilde{x}_0 = x_0 + 2 \tag{21.21}$$

21.5 Symplectic Integration

Symplectic integration, as opposed to concatenation, never has problems with feed–down. The subject of symplectic integration is too large to be covered in this guide. The reader is referred to the book "Beam Dynamics: A New Attitude and Framework" by Étienne Forest[Forest98]. A brief synopsis: Symplectic integration uses as input 1) The Hamiltonian that defines the equations of motion, and 2) a Taylor map M_1 from point 0 to point 1. Symplectic integration from point 1 to point 2 produces a Taylor map M_3 from point 0 to point 2. Symplectic integration can produce maps to arbitrary order. In any practical application the order n of the final map is specified and in the integration procedure all terms of order higher than n are ignored. If one is just interested in knowing the final coordinates of a particle at point 2 given the initial coordinates at point 1 then M_1 is just the constant map

$$M_1 : x_1 = c_i \tag{21.22}$$

where c_i is the initial starting point. The order of the integration is set to 0 so that all non–constant terms are ignored. The final map is also just a constant map

$$M_3 : x_2 = c_f \tag{21.23}$$

If the map from point 1 to point 2 is desired then the map M_1 is just set to the identity map

$$M_1 : x_1 = x_0 \tag{21.24}$$

In general it is impossible to exactly integrate any non–linear system. In practice, the symplectic integration is achieved by slicing the interval between point 1 and point 2 into a number of (generally equally spaced) slices. The integration is performed, slice step by slice step. This is analogous to integrating a function by evaluating the function at a number of points. Using more slices gives better results but slows down the calculation. The speed and accuracy of the calculation is determined by the number of slices and the `order` of the integrator. The concept of integrator order can best be understood by analogy by considering the trapezoidal rule for integrating a function of one variable:

$$\int_{y_a}^{y_b} f(y)\,dy = h\left[\frac{1}{2}f(y_a) + \frac{1}{2}f(y_b)\right] + o(h^3\,f^{(2)}) \tag{21.25}$$

In the formula $h = y_b - y_a$ is the slice width. $0(h^3\,f^{(2)})$ means that the error of the trapezoidal rule scales as the second derivative of f. Since the error scales as $f^{(2)}$ this is an example of a second order integrator. To integrate a function between points y_1 and y_N we slice the interval at points $y_2 \ldots y_{N-1}$ and apply the trapezoidal rule to each interval. Examples of higher order integrators can be found, for example, in Numerical Recipes[Press92]. The concept of integrator order in symplectic integration is analogous.

The optimum number of slices is determined by the smallest number that gives an acceptable error. The slice size is given by the `ds_step` attribute of an element (§5.4). Integrators of higher order will generally need a smaller number of slices to achieve a given accuracy. However, since integrators of higher order take more time per slice step, and since it is computation time and not number of slices which is important, only a measurement of error and calculation time as a function of slice number and integrator order will unambiguously give the optimum integrator order and slice width. In doing a timing test, it must be remembered that since the magnitude of any non-linearities will depend upon the starting position, the integration error will be dependent upon the starting map M_1. *Bmad* has integrators of order 2, 4, and 6 (§5.4). Timing tests performed for some wiggler elements (which have strong nonlinearities) showed that, in this case, the 2^{nd} order integrator gave the fastest computation time for a given accuracy. However, the higher order integrators may give better results for elements with weaker nonlinearities.

Chapter 22

Tracking of Charged Particles

Bmad can track both charged particles and X-rays. This chapter deals with charged particles and X-rays are handled in chapter §23.

For tracking and transfer map calculations (here generically called "tracking"), *Bmad* has various methods that can be applied to a given element (Cf. Chapter §5). This chapter discusses the `bmad_standard` calculation that is the default for almost all element types and the `symp_lie_bmad` calculation that does symplectic integration.

Generally, it will be assumed that tracking is in the forward direction.

22.1 Relative Versus Absolute Time Tracking

Unlike other elements, the kick given a particle going through an `lcavity`, `rfcavity`, or possibly an `em_field` element depends upon the time that the particle enters the element relative to some "RF clock". *Bmad* has two modes for calculating this time called "`relative time tracking`" and "`absolute time tracking`". The switch to set the type of tracking for a lattice is `parameter[absolute_time_tracking]` (§8.1).

The phase of the RF, $\phi_{\rm rf}$, is determined by

$$\phi_{\rm rf} = \phi_{\rm t} + \phi_{\rm ref} \tag{22.1}$$

where $\phi_{\rm t}$ is the part of the phase that depends upon the time t and $\phi_{\rm ref}$ is a fixed phase offset (generally set in the lattice file) and independent of the particle coordinates. See Eqs. (3.26) and (3.40)

The phase $\phi_{\rm t}$ is

$$\phi_{\rm t} = f_{\rm rf}\, t_{\rm eff} \tag{22.2}$$

where $f_{\rm rf}$ is the RF frequency, and $t_{\rm eff}$ is the effective time. With `relative time tracking`, which *Bmad* uses by default, $t_{\rm eff}$ is a function of the phase space coordinate z (§14.4.2) via

$$t_{\rm eff}(s) = t_0(s) - t_0(s_{\rm ent}) - \frac{z(s)}{(\beta\, c)} \tag{22.3}$$

where t_0 is the reference time (see Eq. (14.28)) and $s_{\rm ent}$ is the s-position at the upstream end of the element. $t_{\rm eff}$ is defined such that a particle entering an element with $z = 0$ has $t_{\rm eff} = 0$.

With `absolute time tracking`, $t_{\rm eff}$ is defined by

$$t_{\rm eff}(s) = t(s) - t_0(s_{\rm ent}) \tag{22.4}$$

$t_0(s_{\rm ent})$, by definition, equal to the time of the reference particle at the entrance end of the element. With multipass §7.2, $t_0(s_{\rm ent})$ is set by the time of the reference particle at the entrance end of the element on the first pass. For absolute time tracking, it is important to keep in mind that $t_0(s_{\rm ent})$ is a property of the element independent of how tracking is done. Thus, if a particle goes through a particular element multiple times, the value of $t_{\rm ent}$ will be the same for each transit.

To understand the difference between relative and absolute time tracking, consider a particle traveling on the reference orbit along side the reference particle in a circular ring with one RF cavity. This particle always has $z = 0$ and thus, with `relative time tracking`, $t_{\rm eff}$ will always be zero at the entrance to the cavity. With `absolute time tracking`, the particle, on the first turn, will have $t_{\rm eff}$ equal to zero. However, on subsequent turns (or subsequent passes if using multipass), the time will increase by the revolution time $t_{\rm C}$ on each turn. If the RF frequency $f_{\rm rf}$ is some multiple of the revolution harmonic, the RF phase with absolute vs relative time tracking will be some multiple of $2\,\pi$ and thus RF kick given the particle will be the same in both cases. However, if the RF frequency is not some multiple of the revolution harmonic, there will be a difference in the RF kicks (except for the kick on the first turn).

There are advantages and disadvantages to using either relative or absolute time tracking. Absolute time tracking is more correct since RF cavities may have frequencies that are not comensurate with the revolution time. The problem with absolute time tracking is that the transfer map through the cavity is now a function of time and therefore is a function of z *and* the turn number. This complicates lattice analysis. For example, standard element transfer maps use phase space coordinates so with absolute time tracking, one has a different map for each turn.

With relative time tracking the transfer map problem is swept under the rug. The penalty for using relative time tracking is that results can be unphysical. For example, with relative time tracking, the closed orbit is essentially independent of the RF frequency. From a different angle this can be viewed as a desirable feature since if one is only interested in, say, calculating the Twiss parameters, it can be an annoyance to have to worry that the ring one has constructed have a length that is exactly conmensurate with the RF frequency. And it is potentially confusing to see non-zero closed orbits when one is not expecting it due to a mismatch between the ring circuference and the RF frequency or due to RF cavities not being spaced a multiple of the RF wavelength apart.

The above discussion is limited to the cavity fundamental mode. Long-range wake fields, on the other hand, cannot be synchronized to the z coordinate since, in general, their frequencies are not commensurate with the fundamental mode frequency. For simulating the long-range wakes, the kick is thus, by necessity, tied to the absolute time. The exception is that a wake associated with the fundamental mode (that is, has the same frequency as the fundamental mode) will always use relative time if the fundamental is using relative time and vice versa.

Do not confuse absolute time tracking with the `time_runge_kutta` tracking method (§5.1). The `time_runge_kutta` method uses time as the independent variable instead of z. Absolute time tracking just means that the RF phase is dependent upon the time instead of z. It is perfectly possible to use absolute time tracking with code that uses z as the independent variable.

22.2 Element Coordinate System

The general procedure for tracking through an element makes use of `element reference` coordinates (also called just `element` coordinates). Without any offsets, pitches or tilt (§4.6), henceforth called

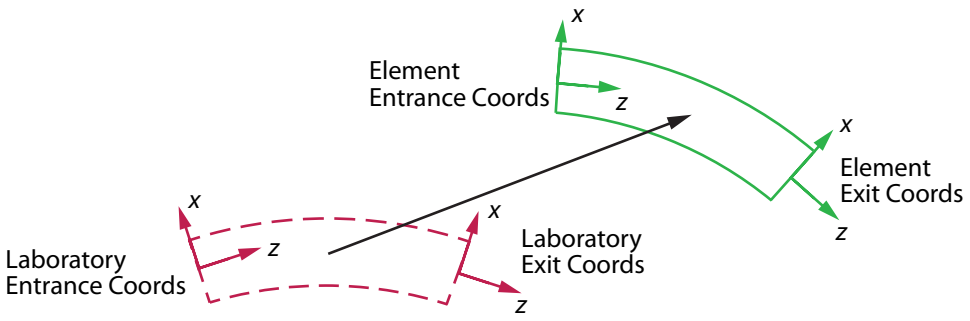


Figure 22.1: `Element` coordinates are coordinates attached to the physical element (solid green outline). The `laboratory` coordinates are fixed at the nominal position of the element (red dashed outline).

.

"misalignments", the `element` coordinates are the same as the `laboratory reference` coordinates (or simply `laboratory` coordinates) (§14.1.1). The `element` coordinates stay fixed relative to the element. Therefore, if the element is misaligned, the `element coordinates` will follow as the element shifts in the laboratory frame as shown in Fig. 22.1.

Tracking a particle through an element is a three step process:

1. At the entrance end of the element, transform from the `laboratory` coordinates to the entrance `element` coordinates.

2. Track through the element ignoring any misalignments.

3. At the exit end of the element, transform from the exit `element` reference frame to the `laboratory` reference frame.

The transformation between `laboratory` and `element` reference frames is given in §14.3.1 and §14.3.2.

22.3 Hamiltonian

The time dependent Hamiltonian H_t in the curvilinear coordinate system shown in Fig. 14.2 is ([Ruth87])

$$H_t = \widetilde{\psi} + \left[\left(\frac{p_s - a_s}{1 + g\,x} \right)^2 + \widetilde{m}^2 + (p_x - a_x)^2 + (p_y - a_y)^2 \right]^{1/2} \tag{22.5}$$

where $(p_x, p_y, p_s/(1+gx))$ are the momentum normalized by P_0, ρ being the local radius of curvature of the reference particle, and $\widetilde{m}$, $\mathbf{a}$ and $\widetilde{\psi}$ are the normalized mass, vector, and scalar potentials:

$$\widetilde{m} = \frac{m\,c^2}{c\,P_0} \qquad \left(a_x, a_y, \frac{a_s}{1 + g\,x} \right) = \frac{q\,\mathbf{A}}{P_0\,c} \qquad \widetilde{\psi}(x, y, z) = \frac{q\,\psi}{P_0\,c} \tag{22.6}$$

In terms of the normalized velocities β_x, β_y, the canonical momentum are

$$p_x = \frac{m\,c^2}{P_0\,c}\,\beta_x + a_x, \qquad p_y = \frac{m\,c^2}{P_0\,c}\,\beta_y + a_y \tag{22.7}$$

The s-dependent Hamiltonian is obtained from H_t by solving for $-p_s$ and using a contact transformation to convert to *Bmad* coordinates (§14.4.2). For particles propagating in the positive s direction, the s-dependent Hamiltonian is, assuming $\widetilde{\psi}$ is zero

$$H \equiv H_s = -(1+g\,x)\sqrt{(1+p_z)^2-(p_x-a_x)^2-(p_y-a_y)^2} - a_s + \frac{1}{\beta_0}\sqrt{(1+p_z)^2+\widetilde{m}^2} \tag{22.8}$$

where β_0 is the reference velocity and the equality $(1+p_z)^2 = (E/c\,P_0)^2 - \widetilde{m}^2$ has been used. The last term on the RHS of Eq. (22.8) accounts for the fact that the *Bmad* canonical z (Eq. (14.28)) has an "extra" term $\beta\,c\,t_0$ so that *Bmad* canonical z is with respect to the reference particle's z.

The equations of motion are

$$\frac{dq_i}{ds} = \frac{\partial H}{\partial p_i} \qquad \frac{dp_i}{ds} = -\frac{\partial H}{\partial q_i} \tag{22.9}$$

Without an electric field, ψ is zero. Assuming a non-curved coordinate system ($g = 0$), and using the paraxial approximation (which expands the square root in the Hamiltonian assuming the transverse momenta are small) (§14.4.2), Eq. (22.8) becomes

$$H = \frac{(p_x-a_x)^2}{2(1+p_z)} + \frac{(p_y-a_y)^2}{2(1+p_z)} - (1+g\,x)\,(1+p_z) - a_s + \frac{1}{\beta_0}\sqrt{(1+p_z)^2+\widetilde{m}^2} \tag{22.10}$$

Once the transverse trajectory has been calculated, the longitudinal position z_2 at the exit end of an element is obtained from symplectic integration of Eq. (22.10)

$$z_2 = z_1 - \frac{1}{2(1+p_{z1})^2}\int ds\,\left[(p_x-a_x)^2+(p_y-a_y)^2\right] - \int ds\,g\,x \tag{22.11}$$

where z_1 is the longitudinal position at the entrance end of the element. Using the equations of motion Eqs. (22.9) this can also be rewritten as

$$z_2 = z_1 - \frac{1}{2}\int ds\,\left[\left(\frac{dx}{ds}\right)^2+\left(\frac{dy}{ds}\right)^2\right] - \int ds\,g\,x \tag{22.12}$$

For some elements, `bmad_standard` uses a truncated Taylor map for tracking. For elements without electric fields where the particle energy is a constant, the transfer map for a given coordinate r_i may be expanded in a Taylor series

$$r_{i,2} \to m_i + \sum_{j=1}^{4} m_{ij}\,r_{j,1} + \sum_{j=1}^{4}\sum_{k=j}^{4} m_{ijk}\,r_{j,1}\,r_{k,1} + \ldots \tag{22.13}$$

where the map coefficients $m_{ij\ldots}$ are functions of p_z. For linear elements, the transfer map is linear for the transverse coordinates and quadratic for $r_i = z$.

Assuming mid–plane symmetry of the magnetic field, so that a_x and a_y can be set to zero[Iselin94], The vector potential up to second order is (cf. Eq. (15.1))

$$a_s = -k_0\left(x - \frac{g\,x^2}{2(1+g\,x)}\right) - \frac{1}{2}k_1\left(x^2-y^2\right) \tag{22.14}$$

For backwards propagation, where particle are traveling in the $-\mathbf{s}$ direction and where p_s is negative, solving for p_s involves using a different part of the square root branch. There is also an overall negative sign coming from switching from using s as the independent variable to $\widetilde{s} \equiv -s$ as the independent variable. the Hamiltonian $H_{\widetilde{s}}$ is then

$$H_{\widetilde{s}} = -(1+g\,x)\sqrt{(1+p_z)^2-(p_x-a_x)^2-(p_y-a_y)^2} + a_s + \frac{1}{\beta_0}\sqrt{(1+p_z)^2+\widetilde{m}^2} \tag{22.15}$$

22.4 Symplectic Integration

Using Eq. (22.10) the Hamiltonian is written in the form

$$H = H_x + H_y + H_z \tag{22.16}$$

where

$$H_x = \frac{(p_x - a_x)^2}{2(1+\delta)}, \qquad H_y = \frac{(p_y - a_y)^2}{2(1+\delta)}, \qquad H_s = -a_s \tag{22.17}$$

For tracking, the element is broken up into a number of slices set by the element's `ds_step` attribute. For each slice, the tracking uses a quadratic symplectic integrator I:

$$I = T_{s/2}\, I_{x/2}\, I_{y/2}\, I_s\, I_{y/2}\, I_{x/2}\, T_{s/2} \tag{22.18}$$

$T_{s/2}$ is just a translation of the s variable:

$$s \rightarrow s + \frac{ds}{2} \tag{22.19}$$

And the other integrator components are

$$\begin{aligned} I_{x/2} &= \exp\left(: -\frac{ds}{2} H_x :\right) \\ I_{y/2} &= \exp\left(: -\frac{ds}{2} H_y :\right) \\ I_s &= \exp\left(: -ds\, H_s :\right) \end{aligned} \tag{22.20}$$

The evaluation of $I_{x/2}$ and $I_{y/2}$ is tricky since it involves both transverse position and momentum variables. The trick is to split the integration into three parts. For $I_{x/2}$ this is

$$\begin{aligned} I_{x/2} &= \exp\left(: -\frac{ds}{2} \frac{(p_x - A_x)^2}{2(1+\delta)} :\right) \\ &= \exp\left(: -\int A_x\, dx :\right) \exp\left(: -\frac{ds}{2} \frac{p_x^2}{2(1+\delta)} :\right) \exp\left(: \int A_x\, dx :\right) \end{aligned} \tag{22.21}$$

With an analogous expression for $I_{y/2}$.

For magnetic elements that do not have longitudinal fields (quadrupoles, sextupoles, etc.), a_x and a_y can be taken to be zero (cf. Eq. (22.14)).

For `lcavity` and `rfcavity` elements, the vector potential is computed from Eq. (15.61).

22.5 BeamBeam Tracking

A beam-beam element (§3.3) simulates the effect on a tracked particle of an opposing beam of particles moving in the opposite direction. The opposing beam, called the "strong" beam, is assumed to be Gaussian in shape.

The strong beam is divided up into `n_slice` equal charge (not equal thickness) slices. Propagation through the strong beam involves a kick at the charge center of each slice with drifts in between the kicks. The kicks are calculated using the standard Bassetti–Erskine complex error function formula[Talman87].

Even though the strong beam can have a finite `sig_z`, the length of the element is always considered to be zero. This is achieved by adding drifts at either end of any tracking so that the longitudinal starting point and ending point are identical. The longitudinal s–position of the `BeamBeam` element is at the center of the strong bunch. For example, with `n_slice` = 2 the calculation would proceed as follows:

1. Start with the reference particle at the center of the strong bunch.
2. Propagate (drift) backwards to the center of the first slice.
3. Apply the beam–beam kick due to the first slice.
4. Propagate (drift) forwards to the center of the second slice.
5. Apply the beam–beam kick due to the second slice.
6. Propagate (drift) backwards to end up with the reference particle at the center of the strong bunch.

22.6 Bend Element: Body Tracking

For a bend without a `k1` component the tracking uses the exact formulas from Forest[Forest98] Eq. 12.18:

$$
\begin{aligned}
x_2 &= \frac{1}{g_{\rm tot}} \sqrt{(1+p_{z1})^2 - p_{x2}^2 - p_{y1}^2} - \frac{1}{g\, g_{\rm tot}} \frac{dp_{x2}}{dL} - \frac{1}{g} \\
p_{x2} &= p_{x1}\, \cos(g\, L) + \left[\sqrt{(1+p_{z1})^2 - p_{x1}^2 - p_{y1}^2} - \frac{g_{\rm tot}}{g}\, (1 + x_1\, g) \right] \sin(g\, L) \\
y_2 &= y_1 + \frac{g\, p_{y1}\, L}{g_{\rm tot}} + \frac{p_{y1}}{g_{\rm tot}} \left[\sin^{-1}\left(\frac{p_{x1}}{\sqrt{(1+p_{z1})^2 - p_{y1}}} \right) - \sin^{-1}\left(\frac{p_{x2}}{\sqrt{(1+p_{z1})^2 - p_{y1}}} \right) \right] \\
p_{y2} &= p_{y1} \\
z_2 &= z_1 + \frac{g\, (1+p_{z1})\, L}{g_{\rm tot}} - L + \frac{1+p_{z1}}{g_{\rm tot}} \left[\sin^{-1}\left(\frac{p_{x1}}{\sqrt{(1+p_{z1})^2 - p_{y1}}} \right) - \sin^{-1}\left(\frac{p_{x2}}{\sqrt{(1+p_{z1})^2 - p_{y1}}} \right) \right] \\
p_{z2} &= p_{z1}
\end{aligned}
\tag{22.22}
$$

where $g_{\rm tot}$ is the total bending strength (design + error).

For a bend with a finite `k1`, the Hamiltonian for the body of an `sbend` is

$$
H = (k_0 - g)\, ix - g\, x\, p_z + \frac{1}{2} \left((k_1 + g\, k_0) x^2 - k_1\, y^2 \right) + \frac{p_x^2 + p_y^2}{2(1+p_z)} \tag{22.23}
$$

This is simply solved

$$
\begin{aligned}
x_2 &= c_x\, (x - x_c) + s_x\, \frac{p_{x1}}{1+p_{z1}} + x_c \\
p_{x2} &= \tau_x\, \omega_x^2\, (1+p_{z1})\, s_x\, (x - x_c) + c_x\, p_{x1} \\
y_2 &= c_y\, y_1 + s_y\, \frac{p_{y1}}{1+p_{z1}} \\
p_{y2} &= \tau_y\, \omega_y^2\, (1+p_{z1})\, s_y\, y_1 + c_y\, p_{y1} \\
z_2 &= z_1 + m_5 + m_{51}(x - x_c) + m_{52} p_{x1} + m_{511}\, (x - x_c)^2 + \\
&\qquad m_{512}\, (x - x_c)\, p_{x1} + m_{522}\, p_{x1}^2 + m_{533}\, y^2 + m_{534}\, y_1\, p_{y1} + m_{544}\, p_{y1}^2 \\
p_{z2} &= p_{z1}
\end{aligned}
\tag{22.24}
$$

where

$$\begin{aligned} k_x &= k_1 + g\,k_0 & \omega_x &\equiv \sqrt{\frac{|k_x|}{1+p_{z1}}} \\ x_c &= \frac{g\,(1+p_{z1}) - k_0}{k_x} & \omega_y &\equiv \sqrt{\frac{|k_1|}{1+p_{z1}}} \end{aligned} \tag{22.25}$$

and

$$\begin{array}{lllll} & k_x > 0 & k_x < 0 & k_1 > 0 & k_1 < 0 \\ c_x = & \cos(\omega_x L) & \cosh(\omega_x L) & c_y = \cosh(\omega_y L) & \cos(\omega_y L) \\ s_x = & \dfrac{\sin(\omega_x L)}{\omega_x} & \dfrac{\sinh(\omega_x L)}{\omega_x} & s_y = \dfrac{\sinh(\omega_y L)}{\omega_y} & \dfrac{\sin(\omega_y L)}{\omega_y} \\ \tau_x = & -1 & +1 & \tau_y = +1 & -1 \end{array} \tag{22.26}$$

and

$$\begin{aligned} m_5 &= -g\,x_c\,L \\ m_{51} &= -g\,s_x & m_{52} &= \frac{\tau_x\,g}{1+p_{z1}}\,\frac{1-c_x}{\omega_x^2} \\ m_{511} &= \frac{\tau_x\,\omega_x^2}{4}\,(L - c_x\,s_x) & m_{533} &= \frac{\tau_y\,\omega_y^2}{4}\,(L - c_y\,s_y) \\ m_{512} &= \frac{-\tau_x\,\omega_x^2}{2\,(1+p_{z1})}\,s_x^2 & m_{534} &= \frac{-\tau_y\,\omega_y^2}{2\,(1+p_{z1})}\,s_y^2 \\ m_{522} &= \frac{-1}{4\,(1+p_{z1})^2}\,(L + c_x\,s_x) & m_{544} &= \frac{-1}{4\,(1+p_{z1})^2}\,(L + c_y\,s_y) \end{aligned}$$

22.7 Drift Tracking

Bmad uses the exact map for a drift This gives the map

$$\begin{aligned} x_2 &= x_1 + \frac{L\,p_{x1}}{(1+p_{z1})\,p_l} \\ p_{x2} &= p_{x1} \\ y_2 &= y_1 + \frac{L\,p_{y1}}{(1+p_{z1})\,p_l} \\ p_{y2} &= p_{y1} \\ z_2 &= z_1 + \left(\frac{\beta}{\beta_{ref}} - \frac{1}{p_l}\right) L \\ p_{z2} &= p_{z1} \end{aligned} \tag{22.27}$$

where β is the normalized particle velocity, β_{ref} is the reference particle's normalized velocity, and p_l is the longitudinal momentum

$$p_l = \sqrt{1 - \frac{p_x^2 + p_y^2}{(1+p_z)^2}} \tag{22.28}$$

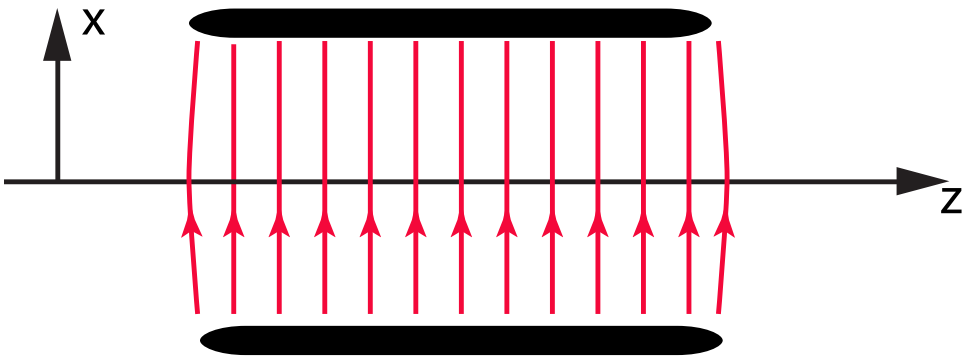


Figure 22.2: Elseparator Electric field. The fringe field lines break the translational invariance in x.

.

22.8 ElSeparator Tracking

[Thanks to Étienne Forest for the derivation of the elseparator equation of motion.]

The Hamiltonian for an electric separator is

$$H = -p_s = -\left\{\left(\frac{1}{\beta_0} + \delta + k_E\, x\right)^2 - \widetilde{m}^2 - p_x^2 - p_y^2\right\}^{1/2} \tag{22.29}$$

Here the canonical coordinates $(-c\,t, \delta$ are being used, $\widetilde{m}$ is defined in Eq. (22.6), and $p_s = -H$ is just the longitudinal momentum. In the above equation, k_E is the normalized field

$$k_E = \frac{q\,E}{P_0\,c} \tag{22.30}$$

The field is taken to be pointing along the x-axis with positive k_E accelerating a particle in the positive x direction. To solve the equations of motion, a "hard edge" model is used where k_E is constant inside the separator and the field ends abruptly at the separator edges.

Since, as shown in Fig. 22.2, the fringe fields break the translational invariance in x, it is important here that the $x = 0$ plane be centered within the separator plates. With this, the canonical momentum δ just outside the separator assumes its free space form of $\delta = (E - E_0)/E_0)$. This is analogous to the case of a `solenoid` where, to ensure that the canonical transverse momenta assume their free space form just outside the solenoid, the `z`-axis must be along the centerline of the solenoid.

The solution of the equations of motion is:

$$\begin{aligned}
x &= (x_0 - x_c)\,\cosh\left(\frac{k_E\,L}{p_s}\right) + \frac{p_{x0}}{k_E}\,\sinh\left(\frac{k_E\,L}{p_s}\right) + x_c \\
p_x &= k_E\,(x_0 - x_c)\,\sinh\left(\frac{k_E\,L}{p_s}\right) + p_{x0}\,\cosh\left(\frac{k_E\,L}{p_s}\right) \\
y &= y_0 + L\,\frac{p_{y0}}{p_s} \\
p_y &= p_{y0} \\
c\,\delta t &= \int_0^L -\frac{\partial H}{\partial \delta} = (x_0 - x_c)\,\sinh\left(\frac{k_E\,L}{p_s}\right) + \frac{p_{x0}}{k_E}\left[\cosh\left(\frac{k_E\,L}{p_s}\right) - 1\right]
\end{aligned} \tag{22.31}$$

where the critical position x_c is

$$x_c = -\frac{\widetilde{E}}{k_E} \tag{22.32}$$

and

$$\widetilde{E} \equiv \frac{1}{\beta_0} + \delta = \frac{E}{P_0\, c} \tag{22.33}$$

Eqs. (22.31) predict that for $x < x_c$ and $p_{x0} = 0$ a particle will, unphysically, accelerate in the negative x direction. In actuality, a particle in this instance will be reflected backwards by the longitudinal component of the edge field. Specifically, the argument of the square root in Eq. (22.29) must be non-negative and a particle will only make it through the speparator if

$$x_0 > \frac{1}{k_E}\left(\sqrt{\widetilde{m}^2 + p_{x0}^2 + p_{y0}^2} - \widetilde{E}\right) \tag{22.34}$$

22.9 Kicker, Hkicker, and Vkicker, Tracking

The Hamiltonian for a horizontally deflecting kicker or separator is

$$H = \frac{p_x^2 + p_y^2}{2(1+p_z)} - k_0\, x \tag{22.35}$$

This gives the map

$$\begin{aligned}
x_2 &= x_1 + \frac{1}{1+p_{z1}}\left(L\, p_{x1} + \frac{1}{2} k_0\, L^2\right) \\
p_{x2} &= p_{x1} + k_0\, L \\
y_2 &= y_1 + \frac{L\, p_{y1}}{1+p_{z1}} \\
p_{y2} &= p_{y1} \\
z_2 &= z_1 - \frac{L}{2(1+p_{z1})^2}\left(p_{x1}^2 + p_{y1}^2 + p_{x1}\, k_0\, L + \frac{1}{3} k_0^2\, L^2\right) \\
p_{z2} &= p_{z1}
\end{aligned} \tag{22.36}$$

The generalization when the kick is not in the horizontal plane is easily derived.

22.10 Lcavity Tracking

The transverse trajectory through an `Lcavity` is modeled using equations developed by Rosenzweig and Serafini[Rosen94] (R&S) with

$$\begin{aligned}
b_0 &= 1 \\
b_{-1} &= 1
\end{aligned} \tag{22.37}$$

and all other b_n set to zero.

The transport equations in R&S were developed in the ulta-relativistic limit with $\beta = 1$. To extend these equations, the transport through the cavity body (R&S Eq. (9)) has been modified to give the

correct phase-space area at non ultra-relativistic energies:

$$\begin{pmatrix} x \\ x' \end{pmatrix}_2 = \begin{pmatrix} \cos(\alpha) & \sqrt{\frac{8}{\eta(\Delta\phi)}}\,\frac{\beta_1\,\gamma_1}{\gamma'}\,\cos(\Delta\phi)\,\sin(\alpha) \\ -\sqrt{\frac{\eta(\Delta\phi)}{8}}\,\frac{\gamma'}{\beta_2\,\gamma_2\,\cos(\Delta\phi)}\,\sin(\alpha) & \frac{\beta_1\,\gamma_1}{\beta_2\,\gamma_2}\,\cos(\alpha) \end{pmatrix} \begin{pmatrix} x \\ x' \end{pmatrix}_1 \tag{22.38}$$

The added factors of β give the matrix the correct determinate of $\beta_1\,\gamma_1/\beta_2\,\gamma_2$. *While the added factors of β do correct the phase space area, the above equation can only be considered as a rough approximation for simulating particles when β is significantly different from 1. Indeed, the only accurate way to simulate such particles is by integrating through the actual field [Cf. Runge Kutta tracking (§5.1)]*

The change in z going through a cavity is calculated by first calculating the particle transit time Δt

$$\begin{aligned} c\,\Delta t &= \int_{s_1}^{s_2} ds\,\frac{1}{\beta(s)} \\ &= \int_{s_1}^{s_2} ds\,\frac{E}{\sqrt{E^2-(mc^2)^2}} \\ &= \frac{c\,P_{z2}-c\,P_{z1}}{G} \end{aligned} \tag{22.39}$$

where it has been assumed that the accelerating gradient G is constant through the cavity. In this equation $\beta = v/c$, E is the energy, and P_{z1} and P_{z2} are the entrance and exit momenta. Using Eq. (14.28), the change in z is thus

$$z_2 = \frac{\beta_2}{\beta_1}\,z_1 - \beta_2\left(\frac{c\,P_{z2}-c\,P_{z1}}{G} - \frac{c\,\overline{P}_{z2}-c\,\overline{P}_{z1}}{\overline{G}}\right) \tag{22.40}$$

where $\overline{P}$ and $\overline{G}$ are the momentum and gradient of the reference particle.

Note that the above transport equations are only symplectic on-axis There are second order terms in the transverse coordinates that are missing. To obtain a proper symplectic matrix, the `symplectify` attribute of an `lcavity` element (§5.6) can be set to True.

22.11 Octupole Tracking

The Hamiltonian for an upright octupole is

$$H = \frac{p_x^2+p_y^2}{2(1+p_z)} + \frac{k_3}{24}(x^4 - 6\,x^2\,y^2 + y^4) \tag{22.41}$$

An octupole is modeled using a kick-drift-kick model.

22.12 Patch Tracking

The transformation of the reference coordinates through a "standard" patch (a patch where custom fields are not used) is given by Eqs. (14.5) and (14.6). At the entrance end of the patch, a particle's position and momentum in the entrance coordinate system will be

$$\begin{aligned} \mathbf{r} &= (x, y, 0) \\ \mathbf{P} &= (P_x, P_y, P_z) = \left(p_x, p_y, \pm\sqrt{(1+p_z)^2 - p_x^2 - p_y^2}\right)\,P_{0\text{ent}} \end{aligned} \tag{22.42}$$

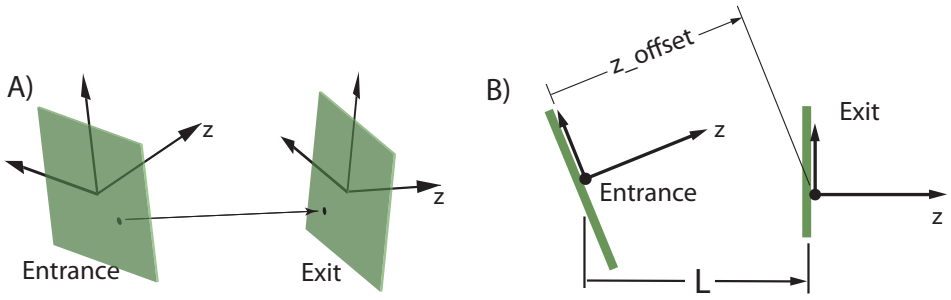


Figure 22.3: Standard tracking through a patch element. A particle's starting coordinate at the entrance end of the patch has, by construction, coordinate $z = 0$. The particle is drifted, as in a field free region, between the entrance $z = 0$ plane and the exit $z = 0$ plane.

where p_x, p_y and p_z are the phase space momenta, and z, which is coordinate z and not phase space z, is always zero by construction as shown in Fig. 22.3 [Also see Fig. 14.2 and the discussion in §14.4.2.] The sign of the longitudinal momentum P_z is determined by whether the particle is traveling in the positive s or negative s direction (which will occur when an element is flipped longitudinally).

The transformation between entrance and exit coordinate systems is given by Eqs. (14.12) and (14.13)

$$\begin{aligned} \mathbf{r} &\to \mathbf{S}^{-1}\,(\mathbf{r} - \mathbf{L}_{\rm off}) \\ \mathbf{P} &\to \mathbf{S}^{-1}\,\mathbf{P} \end{aligned} \tag{22.43}$$

where $\mathbf{L}_{\rm off}$ is given by Eq. (14.16)

After this transformation, the particle must be propagated by a longitudinal length $-r_z$ to intersect the $r_z = 0$ plane of the exit face.

$$\begin{aligned} \mathbf{r} &\to (r_x - r_z\,\frac{P_x}{P_z}, r_y - r_z\,\frac{P_y}{P_z}, 0) \\ \mathbf{P} &\to \mathbf{P} \end{aligned} \tag{22.44}$$

The final $\mathbf{r}$ and $\mathbf{P}$ can now be used compute the particles phase space coordinates, along with the time t and the reference time $t_{\rm ref}$ at the exit end.

$$\begin{aligned} x &\to r_x & p_x &\to \frac{P_x}{P_{0\rm exi}} \\ y &\to r_y & p_y &\to \frac{P_y}{P_{0\rm exi}} \\ z &\to z + r_z\,\frac{|\mathbf{P}|}{P_z} + L_0\,\frac{\beta}{\beta_0} + \beta\,{\rm t\_offset} & p_z &\to \frac{(1+p_z)\,P_{0\rm ent} - P_{0\rm exi}}{P_{0\rm exi}} \\ t &\to t - r_z\,\frac{|\mathbf{P}|}{P_z\,\beta} & t_{\rm ref} &\to t_{\rm ref} + {\rm t\_offset} + L_0\,\frac{1}{\beta_0} \end{aligned} \tag{22.45}$$

where the exit reference momentum $P_{0\rm exi}$ is related to the entrance reference momentum $P_{0\rm ent}$ through `e_tot_offset`. In the above equation, β is the particle velocity, β_0 is the velocity of the reference particle, and L_0 is the drift length of the reference particle

$$L_0 = \frac{1}{S_{33}^{-1}}\,\left(S_{31}^{-1}\,{\rm x\_offset} + S_{32}^{-1}\,{\rm y\_offset} + S_{33}^{-1}\,{\rm z\_offset}\right) \tag{22.46}$$

22.13 Quadrupole Tracking

The `bmad_standard` calculates the transfer map through an upright quadrupole and then transforms that map to the laboratory frame.

The Hamiltonian for an upright quadrupole is

$$H = \frac{p_x^2 + p_y^2}{2(1+p_z)} + \frac{k_1}{2}(x^2 - y^2) \tag{22.47}$$

This is simply solved

$$\begin{aligned}
x_2 &= c_x\, x_1 + s_x\, \frac{p_{x1}}{1+p_{z1}} \\
p_{x2} &= \tau_x\, \omega^2\ (1+p_{z1})\, s_x\, x_1 + c_x\, p_{x1} \\
y_2 &= c_y\, y_1 + s_y\, \frac{p_{y1}}{1+p_{z1}} \\
p_{y2} &= \tau_y\, \omega^2\ (1+p_{z1})\, s_y\, y_1 + c_y\, p_{y1} \\
z_2 &= z_1 + m_{511}\, x_1^2 + m_{512}\, x_1\, p_{x1} + m_{522}\, p_{x1}^2 + m_{533}\, y_1^2 + m_{534}\, y_1\, p_{y1} + m_{544}\, p_{y1}^2 \\
p_{z2} &= p_{z1}
\end{aligned} \tag{22.48}$$

where

$$\omega \equiv \sqrt{\frac{|k_1|}{1+p_{z1}}} \tag{22.49}$$

and

$$\begin{array}{lll|lll}
 & k_1 > 0 & k_1 < 0 & & k_1 > 0 & k_1 < 0 \\
c_x = & \cos(\omega\, L) & \cosh(\omega\, L) & c_y = & \cosh(\omega\, L) & \cos(\omega\, L) \\
s_x = & \dfrac{\sin(\omega\, L)}{\omega} & \dfrac{\sinh(\omega\, L)}{\omega} & s_y = & \dfrac{\sinh(\omega\, L)}{\omega} & \dfrac{\sin(\omega\, L)}{\omega} \\
\tau_x = & -1 & +1 & \tau_y = & +1 & -1
\end{array} \tag{22.50}$$

with this

$$\begin{aligned}
m_{511} &= \frac{\tau_x\ \omega^2}{4}\,(L - c_x\, s_x) & m_{533} &= \frac{\tau_y\ \omega^2}{4}\,(L - c_y\, s_y) \\
m_{512} &= \frac{-\tau_x\ \omega^2}{2\,(1+p_{z1})}\, s_x^2 & m_{534} &= \frac{-\tau_y\ \omega^2}{2\,(1+p_{z1})}\, s_y^2 \\
m_{522} &= \frac{-1}{4\,(1+p_{z1})^2}\,(L + c_x\, s_x) & m_{544} &= \frac{-1}{4\,(1+p_{z1})^2}\,(L + c_y\, s_y)
\end{aligned} \tag{22.51}$$

22.14 RFcavity Tracking

Tracking through an rfcavity uses a kick-drift-kick model. The kick is a pure energy kick (see equations in §3.40) and the phase of the RF is calculated under the assumption that the waveform moves at a phase velocity equal to the velocity of the reference particle.

The transverse forces due to the RF are ignored. This is a resonable approximation when the acceleration is small. `Lcavity` elements should be used in place of `rfcavity` elements when this is not so.

22.15 Sad_Mult Tracking

The "hard edge" fringe field kick is taken from Forest[Forest98] Eqs. (13.29) and onward. In the notation of *Bmad*, and taking into account both normal and skew terms, Eq. (13.29) is for the $m$þorder multipole (what Forest labels $n+1$)

$$f_{\pm} = \mp\Re\frac{(b_m + i\,a_m)\,(x+i\,y)^{(m+1)}}{4\,(m+2)\,(1+p_z)}\left[x\,p_x + y\,p_y + i\frac{m+3}{m+1}(x\,p_y - y\,p_x)\right] \tag{22.52}$$

The "soft edge" dipole fringe for `sad_mult` elements is a generalization of the soft edge dipole fringe for a SAD bend element. For the entrance kick the equations are:

$$\begin{aligned}
x_2 &= x_1 + \frac{\delta_1}{1+\delta_1}\,\Delta x_{fx}, \qquad p_{x2} = p_{x1} + \frac{1}{1+\delta_1}\,\left[\Delta x_{fy}\,v - \Delta x_{fay}\,v^3\right] \\
y_2 &= y_1 - \frac{\delta_1}{1+\delta_1}\,\Delta y_{fy}, \qquad p_{y2} = p_{y1} + \frac{1}{1+\delta_1}\,\left[\Delta y_{fx}\,w - \Delta y_{fax}\,w^3\right] \\
z_2 &= z_1 + \frac{1}{(1+\delta_1)^2}\,\left[\Delta x_{fx}\,p_{x1} - \Delta y_{fy}\,p_{y1} + \frac{1}{2}\,(\Delta y_{fx} + \Delta x_{fy})\,w^2 - \frac{1}{4}(\Delta y_{fax} + \Delta x_{fay})\,w^4\right]
\end{aligned} \tag{22.53}$$

where

$$\begin{aligned}
\Delta x_{fx} &= \frac{K_0\,F_B^2}{24\,L}, & \Delta y_{fx} &= \frac{K_0^2\,F_B}{6\,L^2}, & \Delta y_{fax} &= \frac{2\,K_0^2}{3\,F_B\,L^2}, \\
\Delta y_{fy} &= \frac{SK_0\,F_B^2}{24\,L}, & \Delta x_{fy} &= \frac{SK_0^2\,F_B}{6\,L^2}, & \Delta x_{fay} &= \frac{2\,SK_0^2}{3\,F_B\,L^2}, \\
v &= \cos\theta\,x_1 + \sin\theta\,y_1, & w &= -\sin\theta\,x_1 + \cos\theta\,y_1, & \tan\theta &= \frac{-SK_0}{K_0}
\end{aligned} \tag{22.54}$$

22.16 Sextupole Tracking

The Hamiltonian for an upright sextupole is

$$H = \frac{p_x^2 + p_y^2}{2(1+p_z)} + \frac{k_2}{6}(x^3 - 3\,x\,y^2) \tag{22.55}$$

Tracking through a sextupole uses a kick-drift-kick model.

22.17 Sol_Quad Tracking

The Hamiltonian is

$$H = \frac{(p_x + \frac{k_s}{2}\,y)^2}{2(1+p_z)} + \frac{(p_y - \frac{k_s}{2}\,x)^2}{2(1+p_z)} + \frac{k_1}{2}(x^2 - y^2) \tag{22.56}$$

Solving the equations of motion gives

$$\begin{aligned} x_2 &= m_{11}\, x_1 + m_{12}\, p_{x1} + m_{13}\, y_1 + m_{14}\, p_{y1} \\ p_{x2} &= m_{21}\, x_1 + m_{22}\, p_{x1} + m_{23}\, y_1 + m_{24}\, p_{y1} \\ y_2 &= m_{31}\, x_1 + m_{32}\, p_{x1} + m_{33}\, y_1 + m_{34}\, p_{y1} \\ p_{y2} &= m_{41}\, x_1 + m_{42}\, p_{x1} + m_{43}\, y_1 + m_{44}\, p_{y1} \\ z_2 &= z_1 + \sum_{j=1}^{4} \sum_{k=j}^{4} m_{5jk}\, r_j\, r_k \\ p_{z2} &= p_{z1} \end{aligned} \tag{22.57}$$

where

$$\begin{aligned} m_{11} &= \frac{1}{2\,f}\,(f_{0+}\,c + f_{0-}\,c_h) & m_{31} &= -m_{24} \\ m_{12} &= \frac{1}{2\,f\,(1+p_{z1})}\left(\frac{f_{++}}{\omega_+}\,s + \frac{f_{--}}{\omega_-}\,s_h\right) & m_{32} &= -m_{14} \\ m_{13} &= \frac{\widetilde{k}_s}{4\,f}\left(\frac{f_{+-}}{\omega_+}\,s + \frac{f_{-+}}{\omega_-}\,s_h\right) & m_{33} &= \frac{1}{2\,f}\,(f_{0-}\,c + f_{0+}\,c_h) \\ m_{14} &= \frac{\widetilde{k}_s}{f\,(1+p_{z1})}\,(-c + c_h) & m_{34} &= \frac{1}{2\,f\,(1+p_{z1})}\left(\frac{f_{+-}}{\omega_+}\,s + \frac{f_{-+}}{\omega_-}\,s_h\right) \\ m_{21} &= \frac{-(1+p_{z1})}{8\,f}\left(\frac{\xi_{1+}}{\omega_+}\,s + \frac{\xi_{2+}}{\omega_-}s_h\right) & m_{41} &= -m_{23} \\ m_{22} &= m_{11} & m_{42} &= -m_{13} \\ m_{23} &= \frac{\widetilde{k}_s^3\,(1+p_{z1})}{4\,f}\,(c - c_h) & m_{43} &= \frac{-(1+p_{z1})}{8\,f}\left(\frac{\xi_{1-}}{\omega_+}\,s + \frac{\xi_{2-}}{\omega_-}\,s_h\right) \\ m_{24} &= \frac{\widetilde{k}_s}{4\,f}\left(\frac{f_{++}}{\omega_+}\,s + \frac{f_{--}}{\omega_-}\,s_h\right) & m_{44} &= m_{33} \end{aligned} \tag{22.58}$$

and

$$\begin{aligned} \widetilde{k}_1 &= \frac{k_1}{1+p_{z1}} & \widetilde{k}_s &= \frac{k_s}{1+p_{z1}} \\ f &= \sqrt{\widetilde{k}_s^4 + 4\,\widetilde{k}_1^2} & f_{\pm 0} &= f \pm \widetilde{k}_s^2 \\ f_{0\pm} &= f \pm 2\,\widetilde{k}_1 & f_{\pm\pm} &= f \pm \widetilde{k}_s^2 \pm 2\,\widetilde{k}_1 \\ \omega_+ &= \sqrt{\frac{f_{+0}}{2}} & \omega_- &= \sqrt{\frac{f_{-0}}{2}} \\ s &= \sin(\omega_+\,L) & s_h &= \sinh(\omega_-\,L) \\ c &= \cos(\omega_+\,L) & c_h &= \cosh(\omega_-\,L) \\ \xi_{1\pm} &= \widetilde{k}_s^2\,f_{+\mp} \pm 4\,\widetilde{k}_1\,f_{+\pm} & \xi_{2\pm} &= \widetilde{k}_s^2\,f_{-\pm} \pm 4\,\widetilde{k}_1\,f_{-\mp} \end{aligned} \tag{22.59}$$

The m_{5jk} terms are obtained via Eq. (22.11)

$$\begin{aligned} m_{5jk} = -\frac{\tau_{jk}}{2(1+p_{z1})^2}\int ds\,\Bigg[&\left(m_{2j} + \frac{k_s}{2}\,m_{3j}\right)\left(m_{2k} + \frac{k_s}{2}\,m_{3k}\right) + \\ &\left(m_{4j} - \frac{k_s}{2}\,m_{1j}\right)\left(m_{4k} - \frac{k_s}{2}\,m_{1k}\right)\Bigg] \end{aligned} \tag{22.60}$$

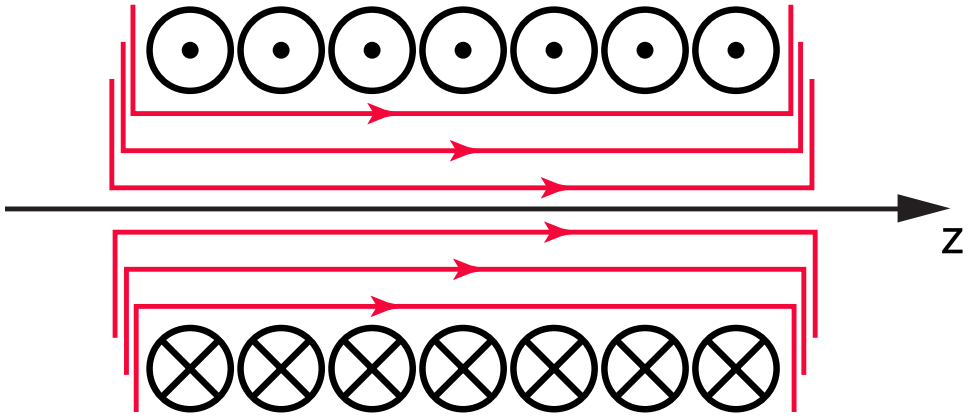


Figure 22.4: Solenoid with a hard edge. The field is assumed to end abruptly at the edges of the solenoid. Here, for purposes of illustration, the field lines at the ends are displaced from one another.

.

where

$$\tau_{jk} = \begin{cases} 1 & j = k \\ 2 & j \neq k \end{cases} \tag{22.61}$$

The needed integrals involve the product of two trigonometric or hyperbolic functions. These integrals are trivial to do but the explicit equations for m_{5jk} are quite long and in the interests of brevity are not reproduced here.

22.18 Solenoid Tracking

The Hamiltonian for a solenoid is

$$H = \frac{\left(p_x + \frac{k_s}{2}\, y\right)^2}{2\,(1+p_z)} + \frac{\left(p_y - \frac{k_s}{2}\, x\right)^2}{2\,(1+p_z)} \tag{22.62}$$

where the normalized field k_s is

$$k_s = \frac{B}{P_0} \tag{22.63}$$

The solution to the equations of motion for a constant k_s are

$$\begin{aligned}
x_2 &= \frac{1+c}{2}\, x_1 + \frac{s}{k_s}\, p_{x1} + \frac{s}{2}\, y_1 + \frac{1-c}{k_s}\, p_{y1} \\
p_{x2} &= \frac{-k_s\, s}{4}\, x_1 + \frac{1+c}{2}\, p_{x1} - \frac{k_s\,(1-c)}{4}\, y_1 + \frac{s}{2}\, p_{y1} \\
y_2 &= \frac{-s}{2}\, x_1 - \frac{1-c}{k_s}\, p_{x1} + \frac{1+c}{2}\, y_1 + \frac{s}{k_s}\, p_{y1} \\
p_{y2} &= \frac{k_s\,(1-c)}{4}\, x_1 + \frac{-s}{2}\, p_{x1} - \frac{k_s\, s}{4}\, y_1 + \frac{1+c}{2}\, p_{y1} \\
z_2 &= z_1 + \frac{L}{2\,(1+p_{z1})^2} \left[\left(p_{x1} + \frac{k_s}{2}\, y_1 \right)^2 + \left(p_{y1} - \frac{k_s}{2}\, x_1 \right)^2 \right] \\
p_{z2} &= p_{z1}
\end{aligned} \tag{22.64}$$

where

$$c = \cos\left(\frac{k_s}{2} L\right)$$
$$s = \sin\left(\frac{k_s}{2} L\right) \tag{22.65}$$

To be useful, the canonical momenta p_x and p_y in the above equations must be connected to the cononical momenta used for other elements (drifts, quadrupoles, etc.) that may be placed to either side of the solenoid. These side elements use zero a_x and a_y (cf. Eq. (22.7)). The vector potential used in the solenoid canonical momenta may be made zero at the edges of the solenoid if the solenoid fringe field is assumed to end abruptly at the edges of the solenoid (as shown in Fig. 22.4), and the reference axis **z**-axis (at $x = y = 0$) is placed along the centerline of the solenoid so that there is cylendrical symmetry around the **z**-axis.

22.19 Symplectic Tracking with Cartesian Modes

The method for symplectic integration for elements that define the magnetic field using a Cartesian mode decomposition (§4.15.2) is outlined in §22.4. The vector potential is constructed to avoid singularities when one of the wave vectors k_x, k_y, or k_z is zero.

For the `x family` the vector potential is:

Form	hyper-y	hyper-xy	hyper-x
A_x	$A \dfrac{k_z}{k_y^2} \mathrm{S}_x \mathrm{Sh}_y \mathrm{S}_z$	$A \dfrac{1}{k_y} \mathrm{Sh}_x \mathrm{Sh}_y \mathrm{S}_z$	$A \dfrac{k_z}{k_x k_y} \mathrm{Sh}_x \mathrm{S}_y \mathrm{S}_z$
A_y	0	0	0
A_z	$A \dfrac{k_x}{k_y^2} \mathrm{C}_x \mathrm{Sh}_y \mathrm{C}_z$	$A \dfrac{k_x}{k_y k_z} \mathrm{Ch}_x \mathrm{Sh}_y \mathrm{C}_z$	$A \dfrac{1}{k_y} \mathrm{Ch}_x \mathrm{S}_y \mathrm{C}_z$

For the `y family` the vector potential is:

Form	hyper-y	hyper-xy	hyper-x
A_x	0	0	0
A_y	$-A \dfrac{k_z}{k_x k_y} \mathrm{S}_x \mathrm{Sh}_y \mathrm{S}_z$	$-A \dfrac{1}{k_x} \mathrm{Sh}_x \mathrm{Sh}_y \mathrm{S}_z$	$-A \dfrac{k_z}{k_x^2} \mathrm{Sh}_x \mathrm{S}_y \mathrm{S}_z$
A_z	$-A \dfrac{1}{k_x} \mathrm{S}_x \mathrm{Ch}_y \mathrm{C}_z$	$-A \dfrac{k_y}{k_x k_z} \mathrm{Sh}_x \mathrm{Ch}_y \mathrm{C}_z$	$-A \dfrac{k_y}{k_x^2} \mathrm{Sh}_x \mathrm{C}_y \mathrm{C}_z$

For the `qu family` the vector potential is:

Form	hyper-y	hyper-xy	hyper-x
A_x	$A \dfrac{1}{k_z} \mathrm{S}_x \mathrm{Ch}_y \mathrm{S}_z$	$A \dfrac{k_y}{k_z^2} \mathrm{Sh}_x \mathrm{Ch}_y \mathrm{S}_z$	$A \dfrac{k_y}{k_x k_z} \mathrm{Sh}_x \mathrm{C}_y \mathrm{S}_z$
A_y	$-A \dfrac{k_x}{k_y k_z} \mathrm{C}_x \mathrm{Sh}_y \mathrm{S}_z$	$-A \dfrac{k_x}{k_z^2} \mathrm{Ch}_x \mathrm{Sh}_y \mathrm{S}_z$	$-A \dfrac{1}{k_z} \mathrm{Ch}_x \mathrm{S}_y \mathrm{S}_z$
A_z	0	0	0

For the `sq family` the vector potential is:

Form	hyper-y	hyper-xy	hyper-x
A_x	$A\,\dfrac{1}{k_z}\;\mathrm{C}_x\,\mathrm{Sh}_y\,\mathrm{S}_z$	$A\,\dfrac{k_y}{k_z^2}\,\mathrm{Ch}_x\,\mathrm{Sh}_y\,\mathrm{S}_z$	$A\,\dfrac{k_y}{k_x\,k_z}\,\mathrm{Ch}_x\,\mathrm{S}_y\,\mathrm{S}_z$
A_y	$A\,\dfrac{k_x}{k_y\,k_z}\,\mathrm{S}_x\,\mathrm{Ch}_y\,\mathrm{S}_z$	$-A\,\dfrac{k_x}{k_z^2}\,\mathrm{Sh}_x\,\mathrm{Ch}_y\,\mathrm{S}_z$	$A\,\dfrac{1}{k_z}\;\mathrm{Sh}_x\,\mathrm{C}_y\,\mathrm{S}_z$
A_z	0	0	0

Chapter 23

Tracking of X-Rays

Bmad can track both charged particles and X-rays. This chapter deals with X-rays. Charged particles are handled in chapter §22.

23.1 Coherent and Incoherent Photon Simulations

Bmad can track photons either `coherenly` or `incoherently`. In both cases, the photon has a transverse electric field

$$(E_x, E_y) \tag{23.1}$$

E_x and E_y are complex and therefore have both amplitude and phase information. When photons are tracked incoherently, the phase information is not used for calculating X-ray intensities.

In addition to coherent and incoherent tracking, partially coherent simulations can be done by using sets of photons with the photons in any one set treated as coherent and the photons between sets being treated as incoherent.

23.1.1 Incoherent Photon Tracking

In a simulation with incoherent photons, some number of photons, N_0, will be generated and the ith photon ($i = 1, \ldots, N_0$) will have a initial "electric field" components $E_{x0}(i), E_{y0}(i)$ assigned to it. The field amplitude E_0 will be $\sqrt{E_{x0}^2 + E_{y0}^2}$.

At some an observation point, the power S per unit area falling on some small area dA due to either x or y component of the electric field is

$$S_{x,y} = \frac{\alpha_p}{N_0\, dA} \sum_{j \in \text{hits}} E_{x,y}^2(j) \tag{23.2}$$

where α_p is a constant that can be chosen to fit the simulation against experimental results, and the sum is over photons who intersect the area. The factors of N_0 and dA in the above equation make, within statistical flucuations, S independent of N_0 and, for dA small enough, S will be independent of dA as it should be. The total power is just $S_x + S_y$.

When traveling through vacuum, the electric field of a photon is a constant. As an example, consider a point source raidiating uniformly in 4π solid angle with each photon having the same initial field E_0. An

observation area dA situated a distance R from the source will intercept $N_0\,dA/4\,\pi\,R^2$ photons which gives a power of

$$S_w = \frac{\alpha_p\,E_0^2}{4\,\pi\,R^2} \tag{23.3}$$

which falls off as $1/R^2$ as expected.

At some places the light may be split into various "channels". An example is Laue diffraction where X-rays can excite the α and β branches of the dispersion surface. Or a partially silvered mirror where some of the light is reflected and some is transmitted. In such a case, the probability P_i of a photon traveling down the i^{th} channel is

$$P_i\,\widehat{E}_i^2 = \frac{S_i}{S_0} \tag{23.4}$$

where S_i is the power flowing into channel i, S_0 is the power flowing into the junction, and $\widehat{E}_i = E_i/E_0$ is the ratio of the electric field amplitudes of any photon just before and just after being shunted into the i^{th} channel. The probabilities must be properly normalized

$$\sum P_i = 1 \tag{23.5}$$

If the ratio of the electric field of any photon just before and just after being shunted into the i^{th} channel is not a constant, than $\widehat{E}_i$ must be adjusted so that $\widehat{E}_i^2$ is equal to the average of $\widehat{E}_i^2(j)$ for all photons j channeled into channel i.

As long as Eqs. (23.4) and (23.5) are satisfied, the choice of the P_i, and $\widehat{E}_i$ are arbitrary. This freedom allows simulation to be optimized for efficiency. For example, In an actual experiment much of the light can be lost never to reach a detector and be counted. To decrease the simulation time, simulated photons may be limited to be generated with a direction to be within some solid angle Ω_1 if photons with a direction outside this solid angle will not contribute to the simulation results. In this case, there are two channels. Channel 1 consists of all photons whose direction is within Ω_1 and channel 2 is all the other photons. To limit the photons to channel 1, P_1 is taken to be 1 and P_2 is taken to be 0. Additionaly, if the light, say, is being generated isotropically from a surface into a $\Omega_0 = 2\,\pi$ solid angle then

$$\widehat{E}_1 = \sqrt{\frac{\Omega_1}{\Omega_0}} \tag{23.6}$$

$\widehat{E}_2$ is infinite here but since no photons are generated in channel 2 this is not a problem.

23.1.2 Coherent Photon Tracking

In a simulation with coherent photons, some number of photons, N_0, will be generated and the ith photon $(i = 1, \ldots, N_0)$ will have an initial electric field $E_{x0}(i), E_{y0}(i)$ assigned to it. These quantities will be complex.

At some an observation point, the field E at some small area dA due to either x or y component of the electric field is

$$E = \frac{\alpha_p}{N_0\,dA} \sum_{j \in \text{hits}} E(j) \tag{23.7}$$

where α_p is a constant that can be chosen to fit the simulation against experimental results, and the sum is over photons who intersect the area. In the above equation $E(j)$ is either the x or y component of the electric field as is appropriate. The factors of N_0 and dA in the above equation make, within statistical flucuations, E independent of N_0 and, for dA small enough, E will be independent of dA as it should be.

When traveling through a a vacuum, the photons travel ballistically in straight lines. This is justified by using the stationary phase approximation with Kirchhoff's integral. the electric field of a photon varies with the propagation length. There is nothing physical in this and is just a way to make the bookkeeping come out correctly. As an example, consider a point source raidiating uniformly in 4π solid angle with each photon having the same initial field component (either x or y) E_1. An observation area dA situated a distance R from the source will intercept $N_0\,dA/4\,\pi\,R^2$ photons and each photon will have a field of $E_1\,R\,\exp(i\,k\,R)$ where k is the photon wave number (all photons must have the same k to be coherent). This gives an electric field at the observation point of

$$E = \frac{\alpha_p\,E_1\,\exp(i\,k\,R)}{4\,\pi\,R} \tag{23.8}$$

which falls off as $1/R$ as expected.

At a `diffraction_plate` element where diffraction effects are to be simulated, the following procedure is used:

1. The electric field components are multiplied by the propagation length L:

$$E \to E\,L \tag{23.9}$$

   The propagation length is reset to zero so that the at the next point where the propagation length is factored into the electric field the propagation length will be the length starting at the aperture.

2. Depending upon the program, the photon is is either given a random direction over $2\,\pi$ solid angle or the photon's direction is restricted to be within some solid angle choosen to increase the probability that the photon will make it through some downstream aperture.

   If the photon is restricted to some aperture dependent solid angle of area Ω, the photon's electric field is scalled by

$$E \to E\,\frac{\Omega}{4\,\pi} \tag{23.10}$$

3. The electric field components are scaled by

$$E \to E\,\frac{k}{4\,\pi\,i}\,(\cos\theta_1 + \cos\theta_2) \tag{23.11}$$

   where θ_1 and θ_2 are the dirction cosines of the incoming and outgoing directions of the photon with respect to the longitudinal reference axis.

This algorithm is designed so that the resulting fields at points downstream from the aperture as computed from a simulation will, to within statistical errors, be the same as one would get using Kirchoff's integral. That is, the simulation is constructed to be a Monte Carlo integration of Kirchhoff's integral.

What is, and what is not considered a place where there are diffraction effects is dependent upon the problem. For example, there are diffraction effects associated with light reflecting from a mirror (or any other object) of finite size. If these effects are important to the experiment, then a procedure similar to the one above must be followed.

At places where there are no diffraction effects a simulation can treat the photons ballistically or can use the aperture procedure outlined above. While in theory it is possible to choose what to do, in practice the aperture procedure increases the number of photons that must be tracked for a given resolution. Thus, from a practical standpoint the ballistic alternative should always be used.

As explaned in §23.1.1, at some places the light may be split into various "channels". With coherent photons, the analog to Eq. (23.4) is

$$P_i \widehat{E}_i = \frac{E_i}{E_0} \tag{23.12}$$

where here $\widehat{E}_i$ can be complex to take into account phase shifts. The same considerations about choosing the P_i and $\widehat{E}_i$ apply to coherent photons as incoherent photons. In particular, $\widehat{E}_1$ for the case of isotropic emission from a surface as in the equample in §23.1.1 (cf. Eq. (23.6)) is

$$\widehat{E}_1 = \frac{\Omega_1}{\Omega_0} \tag{23.13}$$

23.1.3 Parially Coherent Photon Simulations

When there is partial coherence the photons must be divided into sets. All of the photons of a given set are considered coherent while the photons of different sets are treated incoherently.

The procedure is to track all the photons of one set coherently and calculate the field using equation Eq. (23.7). The fields of different sets are then combined to calculate a power using Eq. (23.2).

23.2 Element Coordinate System

The general procedure for tracking through an element makes use of `element reference` coordinates (also called just `element` coordinates). Without any offsets, pitches or tilt (§4.6), henceforth called "misalignments", the `element` coordinates are the same as the `laboratory reference` coordinates (or simply `laboratory` coordinates) (§14.1.1). The `element` coordinates stay fixed relative to the element. Therefore, if the element is misaligned, the `element coordinates` will follow as the element shifts in the laboratory frame as shown in Fig. 22.1.

For `crystal` (§3.8), `mirror` (§3.31), and `multilayer_mirror` (§3.33) elements, the "kinked" reference trajectory through the element complicates the calculation. For these elements, there are three coordinate systems attached to the element as shown in Fig. 23.1. Besides the `element entrance` and `element exit` coordinates, there are `element surface` coordinates with z perpendicular to the surface pointing inward.

Tracking a particle through an element is therefore a three step transformation:

1. At the entrance end of the element, transform from the laboratory reference coordinates to the element's `entrance` or `surface` coordinates.
2. Track through the element ignoring any misalignments.
3. At the exit end of the element, transform from the element coordinates to the `laboratory exit` coordinates.

23.2.1 Transform from Laboratory Entrance to Element Coordinates

For elements that have a reference orbit kink (§23.2), the element coordinates here are the `surface` coordinates. Otherwise the element coordinates are the entrance coordinates.

1. Apply offsets, pitches and tilt using the formulas in §14.2.2 along with Eqs. (14.6), and (14.16).

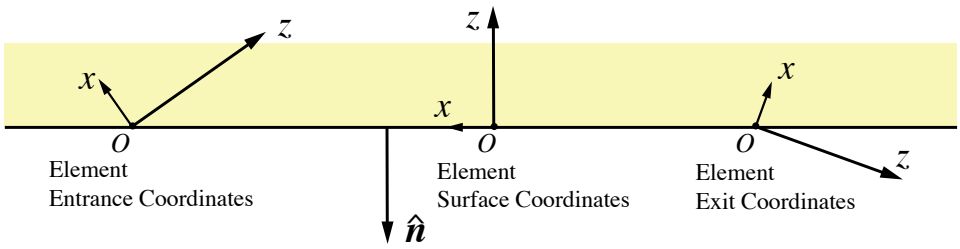


Figure 23.1: The three element coordinate systems for `crystal` (Bragg configuration), `mirror`, and `multilayer_mirror` elements. The origin **O** of all three are the same but are shown spread out for clarity. $\hat{\mathbf{n}}$ is the normal to the element surface.

.

2. Apply the `tilt` to the electric field (Eq. (14.40)).

3. For `crystal`, `mirror`, and `multilayer_mirror` elements rotate to element surface coordinates.

4. Transform the photon's position as if in a drift by a distance $-z$ where z is the photon's longitudinal coordinate. That is, z will be zero at the end of the transform to element coordinates (remember that z is the distance from the start of the element (§14.4.4)).

23.2.2 Transform from Element Exit to Laboratory Coordinate

The back transformation from element to laboratory coordinates is accomplished by the transformation

1. For `crystal`, `mirror`, and `multilayer_mirror` elements rotate to element from element surface coordinates to element exit coordinates

2. Apply the reverse `tilt` to the electric field (Eq. (14.40)).

3. Apply reverse offsets, pitches and tilt using the formulas in §14.2.2 along with Eqs. (14.6), and (14.16).

23.3 Transformation for Mirror and Crystal Elements Between Laboratory and Element Coordinates

23.3.1 Transformation from Laboratory to Element Coordinates

With photons, the intensities must also be transformed. The transformation from the entrance laboratory coordinates to the entrance element coordinates is:

1. Track as in a drift a distance `z_offset_tot`.

2. Apply offsets and pitches: The effective "length" of the element is zero (§14.2.3) so the origin of

the element coordinates is the same point around which the element is pitched so

$$\begin{aligned} x_1 &= x_0 - x_{\rm off} \\ p_{x1} &= p_{x0} - (1 + p_{z0})\, x'_{pitch} \\ y_1 &= y_0 - y_{\rm off} \\ p_{y1} &= p_{x0} - (1 + p_{z0})\, y'_{pitch} \\ z_1 &= z_0 + x'_{pitch}\, x_1 + y'_{pitch}\, y_1 \end{aligned} \tag{23.14}$$

where $x_{\rm off} \equiv$ `x_offset`, $x'_{pitch} \equiv$ `x_pitch`, etc.

3. Apply `ref_tilt` and `tilt`:

$$\begin{aligned} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} &= \mathbf{R}(\theta_{tot}) \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \\ \begin{pmatrix} p_{x2} \\ p_{y2} \end{pmatrix} &= \mathbf{R}(\theta_{tot}) \begin{pmatrix} p_{x1} \\ p_{y1} \end{pmatrix} \\ \begin{pmatrix} \mathbf{E}_{x2} \\ \mathbf{E}_{y2} \end{pmatrix} &= \mathbf{R}(\theta_{tot}) \begin{pmatrix} \mathbf{E}_{x1} \\ \mathbf{E}_{y1} \end{pmatrix} \end{aligned} \tag{23.15}$$

where $\mathbf{E}$ is shorthand notation for

$$\mathbf{E} \equiv E\, e^{i\,\phi} \tag{23.16}$$

with E being the field intensity and ϕ being the field phase angle. In the above equations $\mathbf{R}$ is the rotation matrix

$$\mathbf{R}(\theta) = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \tag{23.17}$$

with θ_{tot} being

$$\theta_{tot} = \begin{cases} \texttt{ref\_tilt} + \texttt{tilt} + \texttt{tilt\_corr} & \texttt{for crystal elements} \\ \texttt{ref\_tilt} + \texttt{tilt} & \texttt{for mirror elements} \end{cases} \tag{23.18}$$

The `tilt_corr` correction is explained in §23.4.2.

23.3.2 Transformation from Element to Laboratory Coordinates

The back transformation from exit element coordinates to exit laboratory coordinates is accomplished by the transformation

1. Apply `ref_tilt` and `tilt`: `ref_tilt` rotates the exit laboratory coordinates with respect to the exit element coordinates in the same way `ref_tilt` rotates the entrance laboratory coordinates with respect to the entrance element coordinates. The forward and back transformations are thus just inverses of each other. With `tilt`, this is not true. `tilt`, unlike `ref_tilt`, does not rotate the output laboratory coordinates. There is the further complication in that `tilt` is a rotation about the *entrance* laboratory coordinates. The first step is to express `tilt` with respect to the exit coordinates. This is done with the help of the $\mathbf{S}$ matrix of Eq. (14.8) with α_t given by Eq. (14.15). The effect of the `tilt` can be modeled as a rotation vector $\mathbf{e}_{in}$ in the entrance laboratory coordinates pointing along the z-axis

$$\mathbf{e}_{in} = (0, 0, \text{tilt}) \tag{23.19}$$

In the exit laboratory coordinates, the vector $\mathbf{e}_{out}$ is

$$\mathbf{e}_{out} = \mathbf{S}\,\mathbf{e}_{in} \tag{23.20}$$

The z component of $\mathbf{e}_{out}$ combines with `ref_tilt` to give the transformation

$$\begin{aligned}
\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} &= \mathbf{R}(-\theta_t)\ \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \\
\begin{pmatrix} p_{x2} \\ p_{y2} \end{pmatrix} &= \mathbf{R}(-\theta_t)\ \begin{pmatrix} p_{x1} \\ p_{y1} \end{pmatrix} \\
\begin{pmatrix} \mathbf{E}_{x2} \\ \mathbf{E}_{y2} \end{pmatrix} &= \mathbf{R}(-\theta_t)\ \begin{pmatrix} \mathbf{E}_{x1} \\ \mathbf{E}_{y1} \end{pmatrix}
\end{aligned} \tag{23.21}$$

where θ_t is ref_tilt $+ \mathbf{e}_{out,z}$. The x and y components of $\mathbf{e}_{out}$ give rotations around the x and y axes

$$p_{x3} = p_{x2} - \mathbf{e}_{out,y}$$

$$p_{y3} = p_{y2} + \mathbf{e}_{out,x} \tag{23.22}$$

$$z_3 = z_2 + x_2\,\mathbf{e}_{out,y} - y_2\,\mathbf{e}_{out,x} \tag{23.23}$$

2. Apply pitches: Since pitches are defined with respect to the entrance laboratory coordinates, they have to be translated to the exit laboratory coordinates

$$\mathbf{P}_{out} = \mathbf{S}\,\mathbf{P}_{in} \tag{23.24}$$

where $\mathbf{P}_{in} = (x'_{pitch}, y'_{pitch}, 0)$ is the pitch vector in the entrance laboratory frame and $\mathbf{P}_{out}$ is the vector in the exit laboratory frame. The transformation is then

$$p_{x4} = p_{x3} - \mathbf{P}_{out,y}$$

$$p_{y4} = p_{y3} + \mathbf{P}_{out,x} \tag{23.25}$$

$$z_4 = z_3 + x_3\,\mathbf{P}_{out,y} - y_3\,\mathbf{P}_{out,x} \tag{23.26}$$

3. Apply offsets: Again, offsets are defined with respect to the entrance laboratory coordinates. Like pitches, the translation is

$$\mathbf{O}_{out} = \mathbf{S}\,\mathbf{O}_{in} \tag{23.27}$$

where $\mathbf{O}_{in} = (x_{\rm off}, y_{\rm off}, s_{\rm off})$ is the offset in the entrance laboratory frame. The transformation is

$$x_5 = x_4 + \mathbf{O}_{out,x} - p_{x4}\,\mathbf{O}_{out,z}$$

$$y_5 = y_4 + \mathbf{O}_{out,y} - p_{y4}\,\mathbf{O}_{out,z} \tag{23.28}$$

$$z_5 = z_4 + \mathbf{O}_{out,z} \tag{23.29}$$

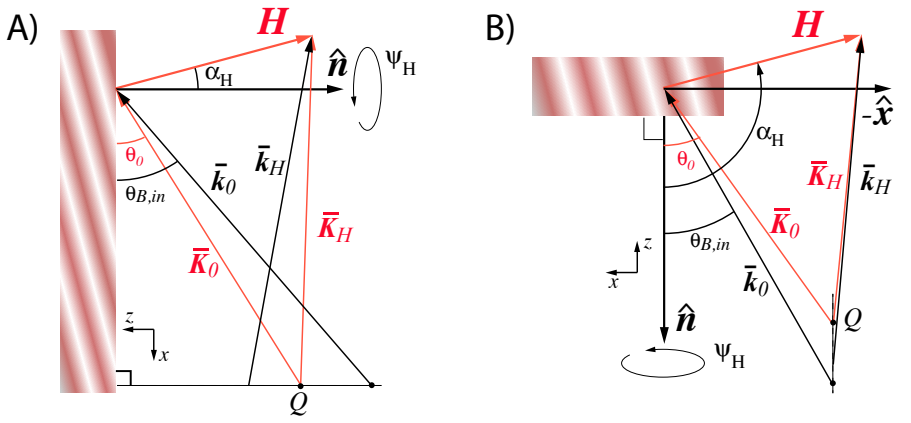


Figure 23.2: Reference trajectory reciprocal space diagram for for A) Bragg diffraction and B) Laue diffraction. The bar over the vectors indicates that they refer to the reference trajectory. The x-z coordinates shown are the element surface coordinates. All points in the diagram are in the plane of the paper except for the tip of $\mathbf{H}$. $\overline{\mathbf{K}}_0$, and $\overline{\mathbf{K}}_H$ are the wave vectors inside the crystal and $\overline{\mathbf{k}}_0$ and $\overline{\mathbf{k}}_H$ are the wave vectors outside the crystal. The reference photon traveling along the reference trajectory has $\overline{\mathbf{K}}_0$ and $\overline{\mathbf{K}}_H$ originating at the Q point. For Laue diffraction, the crystal faces are assumed parallel. For Bragg diffraction the crystal normal is in the $-\hat{\mathbf{x}}$ direction while for Laue diffraction the crystal normal is in the $-\hat{\mathbf{z}}$ direction

23.4 Crystal Element Tracking

[Crystal tracking developed by Jing Yee Chee, Ken Finkelstein, and David Sagan]

Crystal diffraction is modeled using dynamical diffraction theory. The notation here follows Batterman and Cole[Bater64]. The problem can be divided up into two parts. First the reference trajectory must be calculated. This means calculating the incoming grazing angle $\theta_{B,in}$ and outgoing grazing angle $\theta_{B,out}$ as well as calculating the transformations between the various coordinate systems. This is done in §23.4.1, §23.4.2, and §23.4.3. The second part is the actual tracking of the photon and this is covered in §23.4.5 §23.4.6 and §23.4.7.

23.4.1 Calculation of Entrance and Exit Bragg Angles

Fig. 23.2 shows the geometry of the problem. The bar over the vectors indicates that they refer to the reference trajectory. The reference trajectory is calculated such that the reference photon will be in the center of the Darwin curve. That is, the internal wave vectors $\overline{\mathbf{K}}_0$ and $\overline{\mathbf{K}}_H$ originate from the Q point (See [Bater64] Figs. 8 and 29).

The external wave vectors $\mathbf{k_0}$, and $\mathbf{k_H}$ and the internal wave vectors have magnitude

$$|\mathbf{k_0}| = |\mathbf{k_H}| = \frac{\mathbf{1}}{\lambda} \tag{23.30}$$

$$|\overline{\mathbf{K}}_0| = |\overline{\mathbf{K}}_H| = \frac{1-\delta}{\lambda} \tag{23.31}$$

where λ is the wavelength, and δ is

$$\delta = \frac{\lambda^2 r_e}{2\,\pi\,V}\,F_0' = \frac{\Gamma}{2}\,F_0' = \frac{1}{2}\,\Gamma\,F_0' \tag{23.32}$$

with r_e being the classical electron radius, V the unit cell volume, and F_0' is the real part of the F_0 structure factor.

In element surface coordinates (which will be the coordinate system used henceforth), $\overline{\mathbf{k}}_0$ lies in the x-z plane. $\overline{\mathbf{K}}_0$ is related to $\overline{\mathbf{k}}_0$ via Batterman Eq. (25)

$$\mathbf{K}_0 = \mathbf{k}_0 + q_0\,\widehat{\mathbf{n}} \tag{23.33}$$

where the value of q_0 is to be determined. Here, and in equations below, if the equation is true in general, and not just for the reference trajectory, the bar superscript is dropped.

Since $\widehat{\mathbf{n}}$ is in the $-\widehat{\mathbf{x}}$ direction, $\overline{\mathbf{K}}_0$ is also in the x-z plane. Thus $\overline{\mathbf{k}}_0$ and $\overline{\mathbf{K}}_0$ can be written in the form

$$\begin{aligned}
\overline{\mathbf{k}}_0 &= \frac{1}{\lambda}\begin{pmatrix} -\cos\theta_{B,in} \\ 0 \\ \sin\theta_{B,in}\end{pmatrix}, \qquad \overline{\mathbf{K}}_0 = \frac{1-\delta}{\lambda}\begin{pmatrix} -\cos\theta_0 \\ 0 \\ \sin\theta_0\end{pmatrix} \qquad \text{[Bragg]} \\
\overline{\mathbf{k}}_0 &= \frac{1}{\lambda}\begin{pmatrix} \sin\theta_{B,in} \\ 0 \\ \cos\theta_{B,in}\end{pmatrix}, \qquad \overline{\mathbf{K}}_0 = \frac{1-\delta}{\lambda}\begin{pmatrix} \sin\theta_0 \\ 0 \\ \cos\theta_0\end{pmatrix} \qquad \text{[Laue]}
\end{aligned} \tag{23.34}$$

Where, as shown in Fig. 23.2, $\theta_{B,in}$, and θ_0 are the angles of $\overline{\mathbf{k}}_0$ and $\overline{\mathbf{K}}_0$ with respect to the x-axis for Bragg reflections and with respect to the z-axis for Laue reflection.

α_H (`alpha_angle`) is the angle that $\mathbf{H}$ makes with respect to the $-\widehat{\mathbf{z}}$ axis and ψ_H (`psi_angle`) is the rotation of $\mathbf{H}$ around the $-\widehat{\mathbf{z}}$ axis such that for $\psi_H = 0$, $\mathbf{H}$ is in the x-z plane and oriented as shown in Fig. 23.2. Thus

$$\mathbf{H} \equiv \frac{1}{d}\widehat{\mathbf{H}} = \frac{1}{d}\begin{pmatrix} -\sin\alpha_H\,\cos\psi_H \\ \sin\alpha_H\,\sin\psi_H \\ -\cos\alpha_H\end{pmatrix} \tag{23.35}$$

where $\widehat{\mathbf{H}}$ is $\mathbf{H}$ normalized to 1. α_H is determined via the setting of `b_param` and via Eq. (3.10).

The vectors $\mathbf{K}_0$ and $\mathbf{H}$ must add up to the reciprocal lattice vector $\mathbf{K}_H$

$$\mathbf{K}_H = \mathbf{K}_0 + \mathbf{H} \tag{23.36}$$

Taking the length of both sides of this equation and using Eqs. (23.31), (23.34), and (23.35) gives for θ_0

$$\sin\theta_0 = \begin{cases} \dfrac{-\beta\,\widehat{H}_z - \widehat{H}_x\sqrt{\widehat{H}_x^2 + \widehat{H}_z^2 - \beta^2}}{\widehat{H}_x^2 + \widehat{H}_z^2} & \texttt{Bragg} \\[2ex] \dfrac{-\beta\,\widehat{H}_x + \widehat{H}_z\sqrt{\widehat{H}_x^2 + \widehat{H}_z^2 - \beta^2}}{\widehat{H}_x^2 + \widehat{H}_z^2} & \texttt{Laue} \end{cases} \tag{23.37}$$

where

$$\beta \equiv \frac{\lambda}{2\,d\,(1-\delta)} \tag{23.38}$$

Once θ_0 has been calculated, $\theta_{B,in}$ can be calculated from Eq. (23.33)

$$\cos\theta_{B,in} = (1-\delta)\,\cos\theta_0 \quad \text{[Bragg]} \tag{23.39}$$

$$\sin\theta_{B,in} = (1-\delta)\,\sin\theta_0 \quad \text{[Laue]} \tag{23.40}$$

The outgoing reference wave vector k_H is computed using the equation

$$\mathbf{K}_H = \mathbf{k}_H + q_H\,\widehat{\mathbf{n}} \tag{23.41}$$

Using this with Eqs. (23.35) and (23.36) gives

$$\begin{aligned} \overline{k}_{H,x} &= \overline{K}_{H,z} = \frac{1}{d}\,\widehat{H}_x + \overline{k}_{0,x} \\ \overline{k}_{H,y} &= \overline{K}_{H,y} = \frac{1}{d}\,\widehat{H}_y \\ \overline{k}_{H,z} &= \sqrt{\frac{1}{\lambda^2} - \overline{k}_{H,x}^2 - \overline{k}_{H,y}^2} \end{aligned} \tag{23.42}$$

The total bending angle of the reference trajectory is then

$$\theta_{bend} = \tan^{-1}\left(\frac{|\overline{\mathbf{k}}_0 \times \overline{\mathbf{k}}_H|}{\overline{\mathbf{k}}_0 \cdot \overline{\mathbf{k}}_H}\right) \tag{23.43}$$

The outgoing Bragg angle $\theta_{B,out}$ is then *defined* to be the difference between the total bend angle and the entrance Bragg angle.

$$\theta_{B,out} \equiv \theta_{bend} - \theta_{B,in} \tag{23.44}$$

23.4.2 Crystal Coordinate Transformations

There are four transformations needed between coordinates denoted by $\mathbf{\Sigma}_1$, $\mathbf{\Sigma}_2$, $\mathbf{\Sigma}_3$, and $\mathbf{\Sigma}_4$

$\mathbf{\Sigma}_1$ `Transform from laboratory entrance to element entrance coordinates.`
$\mathbf{\Sigma}_2$ `Transform from element entrance to surface coordinates.`
$\mathbf{\Sigma}_3$ `Transform from surface to element exit coordinates.`
$\mathbf{\Sigma}_4$ `Transform from element exit to laboratory exit coordinates.`

The total transformation is just the map represented by $\mathbf{S}$ and $\mathbf{V}$ of Eqs. (14.5) and (14.6)

$$[\mathbf{S}, \mathbf{V}] = \mathbf{\Sigma}_4\,\mathbf{\Sigma}_3\,\mathbf{\Sigma}_2\,\mathbf{\Sigma}_1 \tag{23.45}$$

The transformation $\mathbf{\Sigma}_1$ is given in §23.3.1 and the transformation $\mathbf{\Sigma}_4$ is given in §23.3.2. In general, the transformation $\mathbf{\Sigma}_1$ needs a "tilt correction" (Eq. (23.18)), as explained below, when ψ_H is nonzero. [The exception is when the `undiffracted` or `forward_diffracted` beam is tracked with Laue geometry. In these cases, no tilt correction is needed.] Since this tilt correction is independent of any misalignments, the tilt correction calculation proceeds assuming here that there are no misalignments. The finite $\mathbf{V}$ due to the finite crystal thickness in Laue diffraction will also be ignored for the moment.

Without misalignments, and with ψ_H zero, the transformation $\mathbf{\Sigma}_1$ is, as it is for every other type of element, just the unit matrix.

$$\mathbf{\Sigma}_1 = \mathbf{I} \tag{23.46}$$

That is, the two coordinate systems are identical. Furthermore, the transformation $\mathbf{\Sigma}_2$ from element entrance coordinates to surface coordinates is a rotation around the y axis

$$\begin{aligned} \mathbf{\Sigma}_2 = \mathbf{R}_y(\theta_{B,in}) &\equiv \begin{pmatrix} \cos\theta_{B,in} & 0 & \sin\theta_{B,in} \\ 0 & 1 & 0 \\ -\sin\theta_{B,in} & 0 & \cos\theta_{B,in} \end{pmatrix} && \text{[Laue]} \\ &= \mathbf{R}_y(\theta_{B,in} - \frac{\pi}{2}) && \text{[Bragg]} \end{aligned} \tag{23.47}$$

The transformation from element surface coordinates to element exit coordinates, $\mathbf{\Sigma}_3$, is another rotation around the y axis

$$\begin{aligned} \mathbf{\Sigma}_3 &= \mathbf{R}_y(\theta_{B,out}) && \text{[Laue]} \\ &= \mathbf{R}_y(\theta_{B,out} + \frac{\pi}{2}) && \text{[Bragg]} \end{aligned} \tag{23.48}$$

and the transformation from element exit coordinates to laboratory exit coordinates, $\mathbf{\Sigma}_{out}$ is the unity matrix

$$\mathbf{\Sigma}_4 = \mathbf{I} \tag{23.49}$$

Thus, the combined transformation $\mathbf{S}$ from laboratory entrance to laboratory exit coordinates is a rotation around the y axis of $\theta_{B,in} + \theta_{B,out}$ as explained in section §14.2

$$\mathbf{S} = \mathbf{\Sigma}_4\,\mathbf{\Sigma}_3\,\mathbf{\Sigma}_2\,\mathbf{\Sigma}_1 = \mathbf{R}_y(\theta_{B,in} + \theta_{B,out}) \tag{23.50}$$

When ψ_H is non-zero, the situation is complicated since, if $\mathbf{S}$ as calculated above is used, the vector $\overline{\mathbf{k}}_H$ would be bent out of the x-z plane even though it has been assumed that the `ref_tilt` θ_t is zero. But $\overline{\mathbf{k}}_H$ points in the same direction as the z axis of the outgoing reference trajectory. Furthermore, by *definition*, the reference trajectory has the form given by Eq. (14.8) with the $\mathbf{R}_z(\theta_t)$ matrix depending only upon the `ref_tilt` parameter (which is here taken to be zero). To satisfy Eq. (14.8), the crystal must be reorientated to keep the $\mathbf{k}_H$ vector in the x-z plane of the laboratory entrance coordinates. The reorientation is done by rotating the crystal about the laboratory entrance $\mathbf{z}$ axis by an amount θ_{corr} (`tilt_corr`).

With this tilt correction the transformation $\mathbf{\Sigma}_1$ is a rotation about the z axis

$$\mathbf{\Sigma}_1 = \begin{pmatrix} \cos\theta_{corr} & -\sin\theta_{corr} & 0 \\ \sin\theta_{corr} & \cos\theta_{corr} & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{23.51}$$

To calculate a value for θ_{corr}, note that the transformation $\mathbf{\Sigma}_2$ from element entrance coordinates to element surface coordinates is not affected by a finite ψ_H and so Eq. (23.47) is unmodified. The $\mathbf{k}_H$ vector, expressed in laboratory entrance coordinates, is $\mathbf{\Sigma}_1^{-1}\,\mathbf{\Sigma}_2^{-1}\,\mathbf{k}_H$ where the components of $\mathbf{k}_H$ are given by Eq. (23.42). To satisfy Eq. (14.8), this vector must have zero y component

$$\left(\mathbf{\Sigma}_1^{-1}\,\mathbf{\Sigma}_2^{-1}\,\mathbf{k}_H\right) \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = 0 \tag{23.52}$$

Solving gives

$$\theta_{corr} = \tan^{-1}\frac{k_{H,y}}{k_{H,z}\,\sin\theta_{B,in} - k_{H,x}\,\cos\theta_{B,in}} \tag{23.53}$$

The transformation $\mathbf{\Sigma}_3$ from element surface coordinates to element exit coordinates is now obtained by requiring that the total transformation from laboratory entrance to laboratory exit coordinates be the $\mathbf{R}_y(-\alpha_b)$ matrix given in Eq. (14.8)

$$\mathbf{\Sigma}_3\,\mathbf{\Sigma}_2\,\mathbf{\Sigma}_1 = \begin{pmatrix} \cos\theta_{bend} & 0 & -\sin\theta_{bend} \\ 0 & 1 & 0 \\ \sin\theta_{bend} & 0 & \cos\theta_{bend} \end{pmatrix} \tag{23.54}$$

In the above equation, the transformation $\mathbf{\Sigma}_4$ has been dropped since it is the unit matrix independent of ψ_H.

For Laue diffraction when the non-diffracted beam is tracked, the exit coordinate system corresponds to the entrance coordinate system. That is, $\mathbf{V}$ is the unit matrix. In this case, there is no tilt correction and $\mathbf{\Sigma}_3 = \mathbf{R}_y(-\theta_{B,in})$ is just the inverse of $\mathbf{\Sigma}_2$.

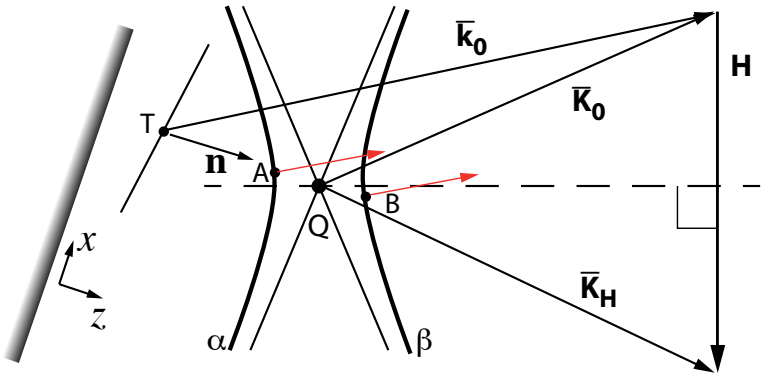


Figure 23.3: Energy flow used to determine the reference orbit for Laue diffraction.

23.4.3 Laue Reference Orbit

For Laue diffraction, with the reference orbit following the `undiffracted` beam, the reference orbit at the exit surface is just the extension of the reference orbit at the entrance surface. Since the reference orbit's direction is $\bar{\mathbf{k}}_0$. the reference orbit displacement vector $\mathbf{L}$ (cf. Eq. (14.5)) is given by

$$\mathbf{L} = \frac{t^2}{d\bar{\mathbf{k}}_0 \cdot \mathbf{t}} d\bar{\mathbf{k}}_0 \qquad \text{[undiffracted]} \tag{23.55}$$

where

$$\mathbf{t} = \begin{pmatrix} 0 \\ 0 \\ t \end{pmatrix} \tag{23.56}$$

with t being the crystal thickness and the z-axis pointing into the crystal as illustrated in Fig. 23.3. The $\mathbf{S}$ roatation matrix (Eq. (14.6)) for the undiffracted beam referece will be the unit matrix.

With the reference orbit following the `forward_diffracted` or `Bragg_diffracted` beam, the displacement vector $\mathbf{L}$ follows the energy flow associated with the tie points labeled A or B in Fig. 23.3. These tie points are defined by the intersection of the dispersion surfaces and the vector $\mathbf{n}$ originating from the point T as shown in the figure. The energy flow is perpendicular to the dispersion surface and it can be shown that since, by construction, $\mathbf{n}$ goes through the Q point, and since the dispersion surfaces are hyperbolies, the energy flows from A and B tie points are colinear. The direction of the energy flow is given by:

$$\bar{\mathbf{K}}_f = \xi_H \bar{\mathbf{K}}_H + \xi_0 \bar{\mathbf{K}}_0 \tag{23.57}$$

where ξ_H and ξ_0 are given by [Bater64] Eq. (18) (See section §23.4.5 below). $\mathbf{L}$ is thus

$$\mathbf{L} = \frac{t^2}{\bar{\mathbf{K}}_f \cdot \mathbf{t}} \bar{\mathbf{K}}_f \tag{23.58}$$

where $\mathbf{t}$ is a vector whose length is the thickness of the crystal oriented perpendicular to the crystal surfaces (that is, in the z direction of the local coordinate system). At the exit surface, if the reference orbit is following the `forward_diffracted` beam, the orientation of the `element exit` coordinates will be the same as the orientation of the `element entrance` coordinates. That is, $\mathbf{S}$ (Eq. (14.6)) is the unit matrix. If the reference orbit is following the `Bragg diffracted` beam, $\mathbf{S}$ is the same as for Bragg diffraction

$$\mathbf{S} = \begin{pmatrix} \cos\theta_{bend} & 0 & -\sin\theta_{bend} \\ 0 & 1 & 0 \\ \sin\theta_{bend} & 0 & \cos\theta_{bend} \end{pmatrix} \tag{23.59}$$

23.4.4 Crystal Surface Reflections and Refractions

There are corrections to the field amplitude and phase when a photon reflects or refracts from the surface of a crystal. A plane wave is incident on a crystal surface with

$$E = \widehat{E}_0 \exp(i\,\mathbf{k}_0\,\mathbf{r}) \tag{23.60}$$

An outgoing plane wave has a field

$$E = \widehat{E}_1 \exp(i\,\mathbf{k}_1\,\mathbf{r}) \tag{23.61}$$

A simulation of this condition will start with a number of photons with wave vector $\mathbf{k}_0$ and electric field E_0. After reflecting from the surface, the photons will have wave vector $\mathbf{k}_1$. Now imagine a set of N photons that flow through an planar area dA_0, perpendicular to the incoming beam, before being reflected from the surface.

Since the electric field is $\widehat{E}_0$, when tracking incoherent photons

$$\widehat{E}_0^2 = \frac{\alpha_p\,E_0^2\,N}{dA_0} \tag{23.62}$$

where α_p is the simulation constant (cf. Eq. (23.2). After the photons are reflected they will have some field E_1 and thus

$$\widehat{E}_1^2 = \frac{\alpha_p\,E_1^2\,N}{dA_1} \tag{23.63}$$

Where dA_1 is the area that the phtons flow through which is related to dA_0 via

$$\frac{dA_1}{dA_0} = \frac{\mathbf{k}_1\cdot\mathbf{z}}{\mathbf{k}_0\cdot\mathbf{z}} \equiv |b| \tag{23.64}$$

Combining the above three equations, the change in field for a photon as it reflects from the surface is

$$\frac{E_1}{E_0} = \frac{\widehat{E}_1}{\widehat{E}_0}\sqrt{|b|} \qquad \text{Incoherent} \tag{23.65}$$

For coherent photon tracking the electric field at dA_0 is

$$\widehat{E}_0 = \frac{\alpha_p\,E_0\,N}{dA_0} \tag{23.66}$$

After the photons are reflected they will have some field E_1 and thus

$$\widehat{E}_1 = \frac{\alpha_p\,E_1\,N}{dA_1} \tag{23.67}$$

Combining these equations the change in field for a photon as it reflects from the surface is

$$\frac{E_1}{E_0} = \frac{\widehat{E}_1}{\widehat{E}_0}\,|b| \qquad \text{Coherent} \tag{23.68}$$

Additionally, for coherent tracking, all photons in a plane wave must have the same phase when passing through an area transverse to the wave. Thus the two photons labeled a and b in Fig. **??** must have the same phase advance in going from dA_0 to dA_1. The difference in the phase advance for photon b relative to a from dA_0 to the surface is $\mathbf{k}_0\cdot\mathbf{r}$ where $\mathbf{r}$ is the vector between where photon b hits the surface relative to phton a. Similarly, the difference in the phase advance for photon b relative to a from

the surface to dA_0 is $-\mathbf{k}_1 \cdot \mathbf{r}$. Since the total phase advance for both photons is the same from dA_0 to dA_1 the phase shift $d\phi_b$ of photon b as it is reflected from the surface relative to the phase shift $d\phi_a$ is

$$d\phi_b = d\phi_a - (\mathbf{k}_1 - \mathbf{k}_0) \cdot \mathbf{r} \tag{23.69}$$

This shift in the reflection phase can be related to the lattice diffration planes. The wave vector difference can be written

$$\mathbf{k}_1 - \mathbf{k}_0 = \mathbf{H} + q\,\widehat{\mathbf{n}} \tag{23.70}$$

where $\widehat{\mathbf{n}}$ is perpendicular to the surface. Combining Eqs. (23.69) and (23.70) and since $\mathbf{r}$ is in the plane of the surface

$$d\phi_b = d\phi_a - \mathbf{H} \cdot \mathbf{r} \tag{23.71}$$

This shows that the reflection shift has the same periodicity as the pattern of the lattice planes at the surface of the crystal. Notice that for a mirror, where one point on the surface is the same as any other, $d\phi_b$ must be equal to $d\phi_a$. Using this in Eq. (23.69) gives

$$\mathbf{k}_1 \cdot \mathbf{r} = \mathbf{k}_0 \cdot \mathbf{r} \tag{23.72}$$

and since $|\mathbf{k}_1| = |\mathbf{k}_0|$ this proves that the angle of incidence is equal to the angle of reflection for a mirror.

In practice, the registration of the surface planes with respect to the surface is not specified in a simulation. Thus the reflection phase shift can only be calculated up to a constant offset.

23.4.5 Bragg Crystal Tracking

The starting photon coordinates are specified in the laboratory entrance coordinates. The transformation from laboratory entrance coordinates to element entrance coordinates $\widetilde{\mathbf{k}}_0$ is given in §23.3. The transformation to element surface coordinates $\mathbf{k}_0$ is

$$\mathbf{k}_0 = \boldsymbol{\Sigma}_2\,\widetilde{\mathbf{k}}_0 \tag{23.73}$$

with $\boldsymbol{\Sigma}_2$ given by Eq. (23.47). The outgoing wave vector $\mathbf{k}_H$ is related to $\mathbf{k}_0$ via

$$\mathbf{k}_H = \mathbf{k}_0 + \mathbf{H} + q_t\,\widehat{\mathbf{n}} \tag{23.74}$$

where q_t is determined by using Eqs. (23.34) and (23.35) in Eq. (23.30)

$$\begin{aligned} k_{H,x} &= k_{0,x} + H_x \\ k_{H,y} &= k_{0,y} + H_y \\ k_{H,z} &= \sqrt{\lambda^2 - k_{H,x}^2 - k_{H,y}^2} \end{aligned} \tag{23.75}$$

To compute the field amplitude of the outgoing photon, the equation to be solved is ([Bater64] Eq. (21))

$$\xi_0\,\xi_H = \frac{1}{4}\,k^2\,P^2\,\Gamma^2\,F_H\,F_{\bar{H}} \tag{23.76}$$

where ξ_0 and ξ_H are given by [Bater64] Eq. (18) and P is the polarization factor

$$P = \begin{cases} 1 & \sigma \text{ polarization state} \\ \cos 2\theta_g & \pi \text{ polarization state} \end{cases} \tag{23.77}$$

$2\theta_g$ is the angle between $\mathbf{K}_0$ and $\mathbf{K}_H$ which is well approximated by $\theta_{B,in} + \theta_{B,out}$.

The solution to Eq. (23.76) is ([Bater64] Eq. (31))

$$\begin{aligned}\xi_0 &= \frac{1}{2}\,k\,|P|\,\Gamma\,[F_H\,F_{\bar{H}}]^{1/2}\,|b|^{1/2}\,[\eta \pm (\eta^2 + \mathrm{sgn}(b))^{1/2}] \\ \xi_H &= \frac{1}{2}\,k\,|P|\,\Gamma\,[F_H\,F_{\bar{H}}]^{1/2}\,\frac{1}{|b|^{1/2}\,[\eta \pm (\eta^2 + \mathrm{sgn}(b))^{1/2}]}\end{aligned} \tag{23.78}$$

where the + part of ± is for the α branch and the − part of ± is for the β branch and sgn is the sign function

$$\mathrm{sgn}(b) \equiv \begin{cases} 1 & b > 0 \\ -1 & b < 0 \end{cases} \tag{23.79}$$

and η is given by [Blas94] Eq. (5)

$$\eta = \frac{-b\,a + \Gamma\,F_0\,(1-b)}{2\,\Gamma\,|P|\,\sqrt{|b|\,F_H\,F_{\bar{H}}}} \tag{23.80}$$

with the asymmetry factor b for the photon being tracked being given by [Blas94] Eq. (3)

$$b \equiv \frac{\hat{\mathbf{n}} \cdot \hat{\mathbf{k}}_0}{\hat{\mathbf{n}} \cdot \widehat{(\mathbf{k}_0 + \mathbf{H})}} \tag{23.81}$$

and the angular deviation variable a is given by [Blas94] Eq. (4)

$$a \equiv \frac{H^2 + 2\,\mathbf{k}_0 \cdot \mathbf{H}}{k_0^2} = -2\,\Delta\theta\,\sin(2\theta_B) \tag{23.82}$$

Once ξ_0 and ξ_H are determined, the ratio of the incoming and outgoing fields for the α or β branches can be computed via ([Bater64] Eq. (24))

$$r_E \equiv \frac{\mathbf{E}_H}{\mathbf{E}_0} = \frac{-\,2\,\xi_0}{k\,P\,\Gamma\,F_{\bar{H}}} = \frac{-\,k\,P\,\Gamma\,F_H}{2\,\xi_H} \tag{23.83}$$

where the α or β subscript has been supressed. The total field which is the sum of the fields on the branches is computed using the boundary conditions

$$\mathbf{E}_0 = \mathbf{E}_{0\alpha} + \mathbf{E}_{0\beta}, \qquad 0 = \mathbf{E}_{H\alpha} + \mathbf{E}_{H\beta} \tag{23.84}$$

Using the above two equations gives

$$\begin{aligned}\mathbf{E}_{0\alpha} &= \mathbf{E}_0\,\frac{r_{E\beta}}{r_{E\beta} - r_{E\alpha}} & \mathbf{E}_{H\alpha} &= \mathbf{E}_0\,\frac{r_{E\alpha}\,r_{E\beta}}{r_{E\beta} - r_{E\alpha}} \\ \mathbf{E}_{0\beta} &= -\mathbf{E}_0\,\frac{r_{E\alpha}}{r_{E\beta} - r_{E\alpha}} & \mathbf{E}_{H\beta} &= -\mathbf{E}_0\,\frac{r_{E\alpha}\,r_{E\beta}}{r_{E\beta} - r_{E\alpha}}\end{aligned} \tag{23.85}$$

As can be seen from Battermann and Cole Figs. (8) and (29), the α tie point is excited and the β tie point is not if $\xi_{0\alpha} < \xi_{0\beta}$ and vice versa. Since only one tie point is excited, The external field ratio is equal to the internal field ratio

$$\frac{E_H^e}{E_0^i} = \frac{E_{Hj}}{E_{0j}} \tag{23.86}$$

where j is α or β as appropriate.

23.4.6 Coherent Laue Crystal Tracking

Laue diffraction has two interior wave fields (branches), labeled α and β, corresponding to the two tie points that are excited on the two dispersion surfaces. For coherent tracking, a photon has some probability to be channeled to follow the α or β branch. The electric field ratios $\widehat{E}_\alpha$ and $\widehat{E}_\beta$ (cf. Eq. (23.12)) are taken to be equal to each other. With this choice, the probabilities P_α and P_β for being channeled to the α or β branches are such that a branch with a greater intensity will have a greater number of photons chanelled down it.

When a crystal's `ref_orbit_follows` parameter is set to `bragg_diffracted`, The branching probabilities are

$$P_\alpha = \frac{|E_{H\alpha}|}{|E_{H\alpha}| + |E_{H\beta}|}, \qquad P_\beta = \frac{|E_{H\beta}|}{|E_{H\alpha}| + |E_{H\beta}|}, \qquad \widehat{E}_{H\alpha} = \widehat{E}_{H\beta} = \frac{|E_{H\alpha}| + |E_{H\beta}|}{|E_0^i|} \tag{23.87}$$

where (see Batermann and Cole[Bater64] Eqs (42)),

$$\begin{aligned} E_{H\alpha} &= -E_0^i \frac{|b|^{1/2}}{2\cosh v} \frac{|P|}{P} \frac{[F_H F_{\overline{H}}]^{1/2}}{F_H} \exp(-2\pi i \mathbf{K}'_{H\alpha} \cdot \mathbf{r}_\alpha) \exp(-2\pi \mathbf{K}''_{H\alpha} \cdot \mathbf{r}_\alpha) \\ E_{H\beta} &= E_0^i \frac{|b|^{1/2}}{2\cosh v} \frac{|P|}{P} \frac{[F_H F_{\overline{H}}]^{1/2}}{F_H} \exp(-2\pi i \mathbf{K}'_{H\beta} \cdot \mathbf{r}_\beta) \exp(-2\pi \mathbf{K}''_{H\beta} \cdot \mathbf{r}_\beta) \end{aligned} \tag{23.88}$$

where $\mathbf{r}_\alpha$ and $\mathbf{r}_\beta$ are the vectors from the entrance surface to the exit surface for the α and β wave fields

$$\mathbf{r}_\alpha = \frac{t^2}{\mathbf{S}_\alpha \cdot \mathbf{t}} \mathbf{S}_\alpha, \qquad \mathbf{r}_\beta = \frac{t^2}{\mathbf{S}_\beta \cdot \mathbf{t}} \mathbf{S}_\beta \tag{23.89}$$

with

$$\begin{aligned} \mathbf{S}_\alpha &= e^{-2v} \mathbf{s}_0 + \left| b \frac{F_H F_{\overline{H}}}{F_H^2} \right| \mathbf{s}_H \\ \mathbf{S}_\beta &= e^{2v} \mathbf{s}_0 + \left| b \frac{F_H F_{\overline{H}}}{F_H^2} \right| \mathbf{s}_H \end{aligned} \tag{23.90}$$

The phase shift of the electric field is obtained from the phase of $E_{H\alpha}$ if the photon is channeled into the α branch and $E_{H\beta}$ if the photon is channeled into the β branch.

When a crystal's `ref_orbit_follows` parameter is set to `forward_diffracted` or `undefracted`, the algorithm is similar to the `bragg_diffracted` case except $E_{0\alpha}$ and $E_{0\beta}$ are used in place of $E_{H\alpha}$ and $E_{H\beta}$ with

$$\begin{aligned} E_{0\alpha} &= E_0^i \frac{e^{-v}}{2\cosh v} \exp(-2\pi i \mathbf{K}'_{0\alpha} \cdot \mathbf{r}_\alpha) \exp(-2\pi \mathbf{K}''_{0\alpha} \cdot \mathbf{r}_\alpha) \\ E_{0\beta} &= E_0^i \frac{e^{-v}}{2\cosh v} \exp(-2\pi i \mathbf{K}'_{0\beta} \cdot \mathbf{r}_\alpha) \exp(-2\pi \mathbf{K}''_{0\beta} \cdot \mathbf{r}_\beta) \end{aligned} \tag{23.91}$$

Since a simulation photon has two polarization components, the above equations are used for one polarization component and for the second polarization component the same branch is used as for the first with an appropriately scaled $\widehat{E}$.

23.4.7 Incoherent Laue Crystal Tracking

Laue diffraction has two interior wave fields (branches), labeled α and β, corresponding to the two tie points that are excited on the two dispersion surfaces. For incoherent tracking it is assumed that

these wave fields overlap at the exit surface ([Bater64] Eq. (87)) and so add coherently. This is a good approximation if the crystal is very thin where the wave fields do not travel an appreciable spatial distance and is also a good approximation when the crystal is thick since the β branch will be heavily attenuated. At intermediate thicknesses this approximation is good when a photon is near the Bragg angle since, in this case, the fields will be traveling in similar directions.

Another approximation is that the path

$$E_{H\alpha} = -E_0^i \frac{|b|^{1/2}}{2\cosh v} \frac{|P|}{P} \frac{[F_H F_{\overline{H}}]^{1/2}}{F_H} \exp(-2\pi i \mathbf{K}'_{H\alpha} \cdot \mathbf{r}_\alpha) \exp(-2\pi \mathbf{K}''_{H\alpha} \cdot \mathbf{r}_\alpha)$$
$$E_{H\beta} = E_0^i \frac{|b|^{1/2}}{2\cosh v} \frac{|P|}{P} \frac{[F_H F_{\overline{H}}]^{1/2}}{F_H} \exp(-2\pi i \mathbf{K}'_{H\beta} \cdot \mathbf{r}_\beta) \exp(-2\pi \mathbf{K}''_{H\beta} \cdot \mathbf{r}_\beta) \quad (23.92)$$

23.5 X-ray Targeting

X-rays can have a wide spread of trajectories resulting in many "doomed" photons that hit apertures or miss the detector with only a small fraction of "successful" photons actually contributing to the simulation results. The tracking of doomed photons can therefore result in an appreciable lengthening of the simulation time. To get around this, *Bmad* can be setup to use what is called "targeting" to greatly reduce the number of doomed photons generated.

Photons can be generated either at a source like a `wiggler` element or at a place where diffraction is simulated like at a `diffraction_plate` element. To be able to do targeting, an element with apertures defined must be present downstream from the generating element. The idea is to only generate photons that are going in the general direction of the "target" which is the space within the aperture.

A necessary restriction for targeting to work is that the photon must travel in a straight line through all elements between the generating element and the element with the apertures. So, for example, a `crystal` element would not be allowed between the two two elements. A `crystal` element could be the aperture element as long as the aperture was defined before photons were diffracted. That is, if the aperture was at the upstream end of the crystal or was defined with respect to the `crystal` surface.

The target is defined by the four corners of the aperture. In `element` coordinates, the (x, y, z) values of the corners are:

```
(-x1_limit, -y1_limit, z_lim)
(-x1_limit,  y2_limit, z_lim)
( x2_limit, -y1_limit, z_lim)
( x2_limit,  y2_limit, z_lim)
```

where `x1_limit`, etc. are the aperture limits (§4.8) and `z_lim` will be zero except if the element's `aperture_at` parameter is set to `entrance_end` in which case `z_lim` will be set to `-L` where `L` is the length of the element.

If the aperture is associated with a curved surface (for example with a `crystal` element), four extra corner points are also used to take into account that the aperture is not planar. These extra points have (x, y, z) values in `element` coordinates of

```
(-x1_limit, -y1_limit, z_surface(-x1_limit, -y1_limit))
(-x1_limit,  y2_limit, z_surface(-x1_limit,  y2_limit))
( x2_limit, -y1_limit, z_surface( x2_limit, -y1_limit))
( x2_limit,  y2_limit, z_surface( x2_limit,  y2_limit))
```

where `z_surface(x,y)` is the z value of the surface at the particular (x, y) point being used. Notice that in this case `z_lim` is zero.

The coordinates of the four or eight corner points are converted from `element` coordinates of the aperture element to `element` coordinates of the photon generating element. Additionally, the approximate center of the aperture, which in `element` coordinates of the aperture element is $(0, 0, z_lim)$, is converted to `element` coordinates of the photon generating element.

The above calculation only has to be done once at the beginning of a simulation.

When a photon is to be emitted from a given point $(x_{emit}, y_{emit}, z_{emit})$, the problem is how to restrict the velocity vector $(\beta_x, \beta_y, \beta_z)$ (which is normalized to 1) to minimize the number of doomed photons generated. The problem is solved by constructing a vector $\mathbf{r}$ for each corner point:

$$\mathbf{r} = (x_{lim}, y_{lim}, z_{lim}) - (x_{emit}, y_{emit}, z_{emit}) \tag{23.93}$$

The direction of each $\mathbf{r}$ is characterized in polar coordinates (ϕ, y) defined by

$$\begin{aligned} y &= \frac{r_y}{|r|} \\ \tan\phi &= \frac{r_x}{r_z} \end{aligned} \tag{23.94}$$

For now make the assumption that r_z is positive and larger than r_x and r_y for all $\mathbf{r}$. Let ϕ_{max} and ϕ_{min} be the maximum and minimum ϕ values over all the $\mathbf{r}$. Similarly, let y_{min} and y_{max} be the minimum and maximum y values over all the $\mathbf{r}$. The rectangle in (ϕ, y) space defined by these four min and max values almost covers the projection of the aperture onto the unit sphere. There is a correction that must be made due to the fact that a straight line of constant y in (x, y, z) space projects to a curved line when projected onto (ϕ, y) space. Therefore a correction must be made to y_{min} when $y_{min} < 0$:

$$y_{min} \to \frac{y_min}{\sqrt{(1 - y_{min}^2)\,\cos^2(phi_{max} - \phi_{min})/2 + y_{min}^2}} \tag{23.95}$$

with a similar correction for y_{max} that must be made when $y_{max} > 0$.

The above prescription works as long as the projection of the aperture onto (ϕ, y) space does not touch the branch cut at $\phi = \pi$ or cover the singular points $y = \pm 1$. Generally these restrictions are fullfilled since $\mathbf{z}$ is the direction of the reference orbit. If this is not the case, a transformation can be made where rotation matrices are constructed to transform between the `element` coordinates of the emitting element and what are called `target` coordinates defined so that $\mathbf{r}$ for the `center` point has the form $(0, 0, |r|)$. The procedure for calculating the photon velocity vector is now

1. Rotate all the corner $\mathbf{r}$ from `element` to `target` coordinates.
2. Calculate min and max values for ϕ and y.
3. Calculate the velocity vector such that the (ϕ, y) of this vector falls within the min and max values in the last step.
4. Rotate the velocity vector back to `element` coordinates.

Chapter 24

Simulation Modules

In the *Bmad* "ecosystem", various modules have been developed to simulate machine hardware. This chapter provides documentation.

24.1 Tune Tracker Simulator

[Tune tracker simulation developed by Michael Ehrlichman]

The digital tune tracker (dTT) is device for determining the fractional tune of a storage ring and exciting resonant beam oscillations. In short, the dTT starts with a beam position monitor (BPM) to detect the beam oscillations. The signal from the BPM is feed into a phase-locked loop (PLL) circuit which oscillates at the oscillation frequency the beam. A signal from the PLL is used to modulate a kicker to keep the beam oscillating at the beam's oscillation frequency. The tune tracker is capable of exciting both betatron and synchrotron oscillations.

In general, a PLL is a control system for matching the frequency of a VCO to some incoming periodic reference signal. It does this by adjusting the frequency of the VCO according to the phase difference between the VCO output and the reference signal. A general diagram of a PLL is shown in Fig. 24.1.

In Fig. 24.1, the phase detector compares the two incoming signals, and outputs a signal that is proportional to the difference in phase. There are various ways to implement a phase detector. The tune tracker phase detector is a mixer followed by low-pass filter, which will be discussed in more detail below. The loop controller, also known as a loop filter, is usually some combination of (P)roportional, (I)ntegration, and (D)ifferentiation paths. The loop controller sums these paths, and the resulting signal is sent to the

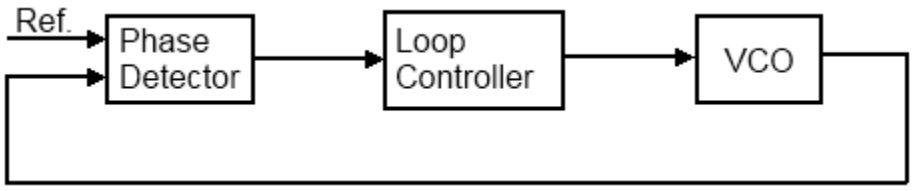


Figure 24.1: General diagram of a phase lock loop. The loop adjusts the speed of a VCO according to the phase difference between the VCO and an incoming signal.

VCO. The output of the loop controller rises when the VCO is slower than the reference, and drops when the VCO is faster. The gains of the three PID paths are adjusted to produce a control loop that is stable and can efficiently lock onto the reference signal from some incorrect initial frequency. A real-world PLL also needs to track perturbations to the reference frequency.

A diagram of the digital tune tracker is shown in Fig. 24.2. This function takes in one new BPM measurement per call, and returns a sinusoidal signal that has the same frequency as the BPM signal. The phase of the returned signal differs from the phase of the BPM signal by ϕ_0, the phase advance from position at the BPM to position at the kicker on the next turn.

Comparing this diagram to the PLL in Fig. 24.1, the boxes from *Subtract closed orbit offset* to the end of the *Low Pass Filter* comprise the phase detector. After the *Low Pass Filter* to just before the *Gain Kvco* box comprises the loop controller. The *Gain Kvco* box comprises the VCO. The feedback loop consists of the update to ω that follows *Gain Kvco*.

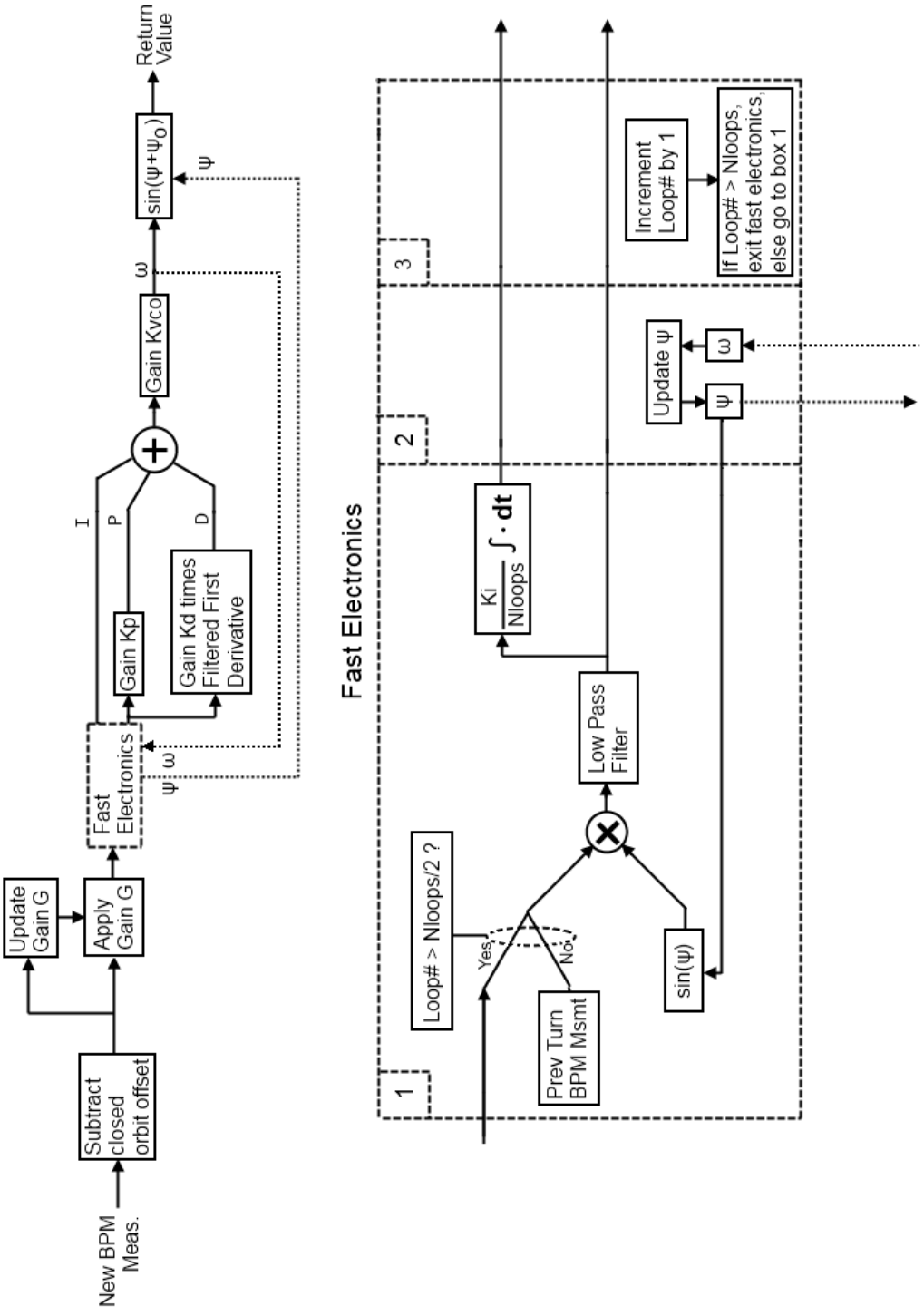


Figure 24.2: Flow chart of tune tracker module functions.

24.1.1 Tune Tracker Components.

New BPM Meas. This consists of one BPM measurement. It is either horizontal or vertical position at the BPM location. It is assumed that each measurement is separated by the same Δt, and that Δt is an integer multiple of the ring period.

Subtract closed orbit offset Due to misalignments or the presence of wigglers, the closed orbit at the BPM may be non-zero. The closed orbit is subtracted from the incoming data. The closed orbit offset is a constant parameter set by the user during initialization.

Apply Gain G This block adjusts the incoming measurement such that,

$$Meas.\ Out = \frac{Meas.\ In}{G}$$

The purpose of this block is to normalize the BPM measurements. When the tune tracker first starts, the amplitude of the beam oscillations is small. When the tune tracker is locked on, the amplitude of the beam oscillations becomes large. The set of loop controller gains that allows the tune tracker to quickly lock onto the initially small signal are not optimal when the beam oscillations are large. This normalization overcomes that problem, allowing for a set of loop controller gains that are optimal both at start up and when oscillations are large.

Update Gain G This is a digital filter of the form,

$$G = g_a \mathrm{abs}\left(x\right) + \left(1 - g_a\right) G,$$

which is a exponentially weighted moving average, or equivalently, a low-pass filter. x is the incoming data point. g_a is a hard coded time constant. It is set so that G equilibrates to the rms beam size after about 10 periods.

Fast Electronics Fast electronics contains the components of the tune tracer that cycle every few RF periods. Whereas the rest of the tune tracker cycles once per storage ring period, the fast electronics cycle every few RF buckets.

Prev Turn BPM Msmt Stores the previous turn's BPM measurement.

Loop# $>$ $N_{loops}/2$? N_{loops} is the number of times the fast electronics cycle every storage ring period. For the first half of these loops, the BPM measurement from the previous turn is used. For the last half, the current BPM measurement is used.

sin (ψ) Sinusoid of the current modulator angle ψ.

$\boldsymbol{\psi}$ Current modulator angle. Updated at the end of every fast electronics loop.

$\times$ **(times)** The BPM measurement and $\sin\left(\psi\right)$ are multiplied. Let the BPM signal be parameterized as $\cos\left(\omega_{beam} t\right)$, and write ψ as $\omega_{vco} t$ so we have,

$$\cos\left(\omega_{beam} t\right) \sin\left(\omega_{vco} t\right).$$

A trig identity allows this to be written as,

$$\frac{1}{2}\left(\sin\left(\left(\omega_{beam} + \omega_{vco}\right) t\right) - \sin\left(\left(\omega_{beam} - \omega_{vco}\right) t\right)\right).$$

If ω_{vco} is roughly ω_{beam}, then the first sin term has roughly twice the beam frequency and the second term has a very small frequency. Therefore, passing the multiplied signal through a low-pass filter removes the first term, leaving only the second term.

In stable operation, ω_{vco} makes small amplitude oscillations about ω_{beam}. In this case, the time integral of $(\omega_{beam} - \omega_{vco})\,t$ is small for all t, and a small angle approximation for sine can be applied, and the output of the low pass filter can be written as,

$$(\omega_{beam} - \omega_{vco})\,t, \tag{24.1}$$

which is proportional to the phase difference between the beam and the VCO. Thus completes the phase detector.

$\frac{K_I}{N_{loops}} \int \cdot dt$ This is the (I)ntegrated channel of the PID controller. It integrates the output of the phase detector, providing negative feedback. If the integrated phase is positive, the I channel slows the VCO down. If negative, it speeds the VCO up. This channel is expected to equilibrate to some non-zero value proportional to the difference between the beam frequency and the VCO base frequency.

Update ψ This increments the modulator phase according to,

$$\psi_{n+1} = \psi_n + \frac{T_{ring}}{N_{loops}}\omega_{vco}. \tag{24.2}$$

In order to more accurately model things, ω_{vco} is not updated within the fast electronics loop.

Increment Loop# by 1 Loop# counts the number of times the fast electronics has been looped through.

If Loop# $>$ N_{loops}, exit, else, loop N_{loops} is the number of times the fast electronics cycle per ring period.

Gain K_P This is the (P)roportional channel of the PID controller. Is provides negative feedback proportional to the output of the phase detector. This channel responds faster than the integrated channel and so helps the tune tracker track sudden changes in the frequency of the beam. In practice, it also damps tune tracker oscillations. If K_P is too small, the VCO may oscillate indefinitely around the fractional tune of the beam. See section §24.1.2 below for more details.

Gain K_D $\times$ filtered derivative This is the (D)ifferential channel of the PID controller. Provides a filtered first derivative of the output of the phase detector. The derivative is obtained a polynomial fitted to a history of the phase detector output. This channel can be used to smooth out the VCO response and reduce overshoot. See section §24.1.2 below for more details.

+ (sum) This block sums the three output of the PID controller.

Gain K_{vco} This block represents the response of a voltage controlled oscillator. The frequency of the output of a VCO is proportional to the voltage of its input. This block simply multiplies the output of the PID controller by a constant. This block updates the VCO frequency.

sin $(\psi + \psi_0)$ Calculates kicker amplitude to be returned to the calling program. This is the kick amplitude that should be used during the next turn. ψ_0 is the phase advance from position at the BPM to position at next turn's kicker.

24.1.2 Tuning

Tuning the tune tracker consists of adjusting the four gain parameters K_P, K_I, K_D, and K_{vco}. The goal is to set the parameters such that the tune tracker quickly locks onto the fractional tune of the beam, and settles quickly. If multiple tunes are being explored, then the size of the convergence region is also

Parameter	Rise Time	Overshoot	Settling Rate	Precision	Stability Region
K_P	slight decrease	decrease	faster	degrade	shrink
K_I	decrease	increase	slower	no change	shrink
K_D [see note]	slight increase	slight decrease	slightly faster	degrade	grow

Table 24.1: Effect on VCO response of increasing K_P, K_I, or K_D. *Note:* The effects of the derivative channel are theoritical and have not been verified extensively.

important. i.e. The set of parameters should work for a wide range of fractional tunes. If jitter is being explored, then the ability of the tune tracker to track a varying fractional tune is also important.

A plot of the VCO speed during typical tune tracker operation is shown in Fig. 24.3. A tune tracker with a base period of 0.571 oscillations per ring period tracks a beam with a fractional tune of 0.5691. Marked on this plot are 4 important features. They are the rise time, overshoot, settling rate, and precision. The rise time is how long it takes the tune tracker to approach the fractional tune of the beam. This is the location of the first peak after operation begins. The overshoot is the difference between the amplitude of the first peak and the fractional tune. The settling rate measures the decay rate of the oscillations of the VCO about the fractional tune. The precision is the noise in the VCO frequency. The discrete nature of the BPM measurements introduces noise at the phase detector stage. The proportional channel adds this noise to the VCO frequency Shown in Tab. [24.1] is the effect increasing the PID controller gains.

Stability region refers to the range of fractional tunes that the tune tracker will converge to when starting from an initial tune in that range. In practice, if K_P and K_I are increased to optimize response to a particular fractional tune, the stability region will shrink. e.g. A large K_P and K_I could be found that very quickly lock on to a fractional tune of 0.569, but are unstable for 0.622. But note that a smaller K_P and K_I can be found that give acceptible performance at 0.569, and also acceptible performance at 0.622 (and even 0.074).

24.1.3 Programmer Instructions

This seciton contains instructions on how to implement the tune tracker module and how to use the tune tracker driver program. The tune tracker is written as an object or class, and so supports multiple instances.

After setting up the tune tracker in a program, it is best to first verify that the tune tracker is efficiently locking onto the beam and that the beam oscillations are indeed growing with time. The tune tracker should lock on after about 10,000 turns, and the beam oscillations should reach equilibrium amplitude after about 10 damping periods.

See the tune tracker section in the Physics chapter of this manual for an example of how the VCO frequency should respond with time in a successful implementation. Guidelines for setting the gain parameters are also located in the Physics chapter.

24.1.4 Tune Tracker Module

The tune tracker module has four public functions. They are,

```
Function id = init_dTT(tt_param_struct)     !Constructor, creates new tune tracker
                                            !instance, handles initialization.
```

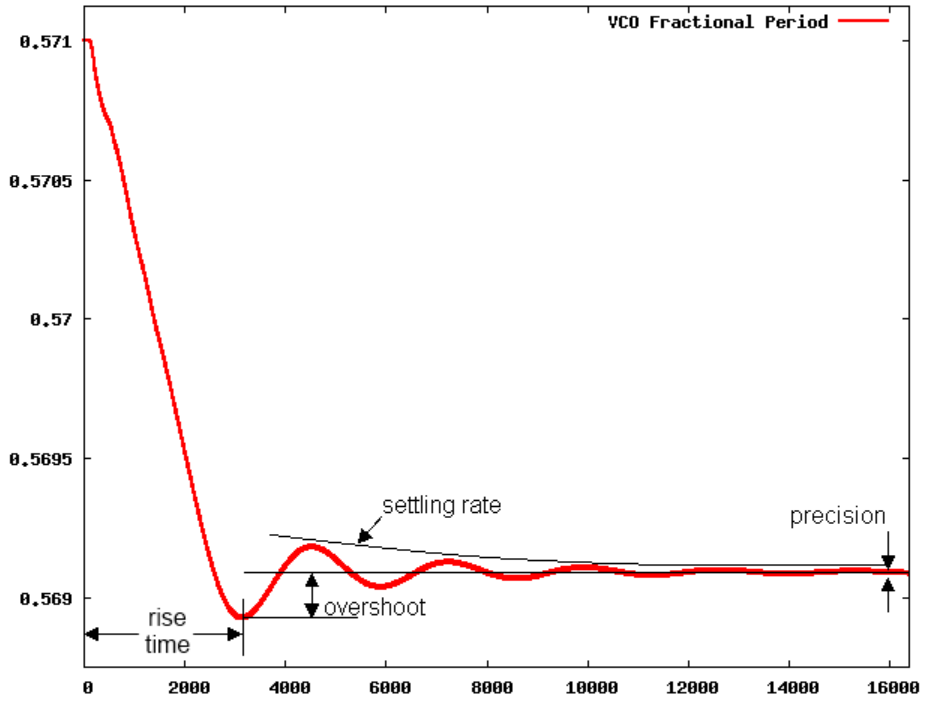


Figure 24.3: Plot of VCO response of typical tune tracker setup. VCO base frequency is 0.571. Beam fractional tune is 0.5691.

```
                                        !Passed tt_param_struct, Returns id of
                                        !created instance.
Function z = TT_update(bpm_msmt,id)     !Main function, passed one new BPM
                                        !measurement and returns VCO modulator
                                        !amplitude.
Subroutine dest_dTT(id)                 !Destructor, closes log files and
                                        !deallocates memory.
                                        !Takes as argument id of instance
                                        !to be destroyed.
Function z = get_dTT(name,id)           !Get function, name is either 'wf',
                                        !in which case the function returns
                                        !the VCO frequency ω0 + Δω,
                                        !or 'dw', in which case just Δω
                                        !is returned.
```

Pseudocode for a tune tracker implementation is below.

```
Populate tt_param_struct with tune tracker parameters.
id = init_dTT(tt_param_struct)
Loop over turns:
  Track once through the ring, obtain orbit.
  bpm_msmt = x or y at BPM location.
  z = TT_update(bpm_msmt,id)
  Adjust kicker amplitude (or cavity phase) according to z
```

```
dest_dTT(id)
```

See `tune_tracker.f90` for the contents of `tt_param_struct`.

If a horizontal or vertical tune tracker is to be implemented, the BPM measrements should be the x or y coordinate at the BPM location, and the kicks should be changes in x' or y' at the kicker location. If a longitudinal tune tracker is to be implemented the BPM measurements should be the x coordinate in a dispersive region, and the kicks should be changes in phase at an RF cavity. Similarly, if a longitudinal tune tracker is being used in conjunction with a horizontal tune tracker, the BPM of the horizontal tune tracker should be placed in a region of low dispersion. This will prevent the horizontal tune tracker from inadvertantly locking onto the synchrotron tune.

24.1.5 Tune Tracker Example Program

An example program is located in `src/examples/tune_tracker`. This program contains code for implementing simultaneous horizontal, vertical, and longitudinal tune trackers.

For most implementations of the tune tracker module, a good approach would be to either use a modified version of the tune tracker example program, or copy and paste code from the example program. Keep in mind that the example program implements multiple simultaneous tune trackers. An implementation of just one tune tracker would be much simpler.

The example program performs the following steps.

1. Read in file which contains tune tracker parameters such as gains.
2. Parse lattice with `twiss3` subroutines. The `twiss3` subroutines are needed to obtain the longitudinal phase advance when implementing longitudinal tune trackers.
3. Check that the kick attribute of the kicker element is actually free.
4. Calculate phase advance from position at BPM to position at kicker. Note that this is the phase of the kicker *on the next turn*.
5. Creates log files which record position at BPM and kicker, the VCO frequency, and slope at the kicker.
6. Loops over turns, passing BPM measurements to `TT_update` and adjusting kicker amplitudes based on the returned value. Note that it will take the beam oscillations several damping times to equilibriate. This often means tracking for several 10,000 turns. Use of the save state parameter may shorten simulation times in certain applications.
7. Calculates and FFT of the BPM data and records in log file.

Also located in `src/examples/tune_tracker` is an example .in file that implements horizontal, vertical, and longitudinal tune trackers. The gains used in the .in file were selected for their large convergence region. More agressive tuning may lead to faster lock on. See the Physics section for details.

24.1.6 Save States

If `tt_param_struct%useSaveState` is set to true, then the destructor `dest_dTT` will save the tune tracker state variables to a file called `tt_state.#`. Similarly, the constructor `init_dTT` will set the state

variables according to the save state file. The constructor will also return the beam coordinates to be used at element zero of the ring.

Save states allow you to pick up where you left off, and not have to wait for the beam oscillations to equilibriate. For example, you could have one program that tracks for 100,000 turns, allowing the tune tracker to lock on and the beam excitation to equilibriate with radiation damping. The simulation writes a save state file which can be used as a starting point for other simulations that perform studies on the excited beam.

24.2 Instrumental Measurements

Bmad has the ability to simulate instrumental measurement errors for orbit, dispersion, betatron phase, and coupling measurements. The appropriate attributes are listed in §4.21 and the conversion formulas are outlined below.

24.2.1 Orbit Measurement

For orbits, the relationship between measured position $(x,y)_{\rm meas}$ and true position $(x,y)_{true}$ is

$$\begin{pmatrix} x \\ y \end{pmatrix}_{\rm meas} = n_f \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} + \mathbf{M}_m \left[\begin{pmatrix} x \\ y \end{pmatrix}_{true} - \begin{pmatrix} x \\ y \end{pmatrix}_0 \right] \tag{24.3}$$

with

$$\begin{pmatrix} x \\ y \end{pmatrix}_0 = \begin{pmatrix} x_{\rm err} - x_{\rm cal} \\ y_{\rm err} - y_{\rm cal} \end{pmatrix} \tag{24.4}$$

and

$$\mathbf{M}_m = \begin{pmatrix} g_x \cos(d\theta + d\psi) & g_x \sin(d\theta + d\psi) \\ -g_y \sin(d\theta - d\psi) & g_y \cos(d\theta - d\psi) \end{pmatrix} \tag{24.5}$$

where

$$\begin{aligned} d\psi &= \psi_{\rm err} - \psi_{\rm cal} \\ d\theta &= \theta_{\rm err} - \theta_{\rm cal} \\ g_x &= 1 + dg_{x,\rm err} - dg_{x,\rm cal} \\ g_y &= 1 + dg_{y,\rm err} - dg_{y,\rm cal} \end{aligned} \tag{24.6}$$

r_1 and r_2 are Gaussian random numbers whose distribution is centered at zero and has unit width. n_f is the noise factor inherent in the measurement, $(x,y)_{\rm err}$ are monitor offset errors and $(x,y)_{\rm cal}$ are the offset calibration factors. $\theta_{\rm err}$ and $\phi_{\rm err}$ are error tilt and "crunch" angles, and $\theta_{\rm cal}$ and $\phi_{\rm cal}$ are the corresponding calibration angles. Finally, $dg_{x,\rm err}$ and $dg_{y,\rm err}$ are the horizontal and vertical gain errors, and $dg_{x,\rm cal}$ and $dg_{y,\rm cal}$ are the corresponding calibration gains.

The calibration variables are useful for simulating the process where a measurement or series of measurements is analyzed to find the values of the error parameters. In this case, the measured position $(x,y)_m$ represents the beam position corrected for "known" offsets, tilts, and gain errors.

24.2.2 Dispersion Measurement

A dispersion measurement is considered to be the result of measuring the orbit at two different energies. The measured values are then

$$\begin{pmatrix}\eta_x\\ \eta_y\end{pmatrix}_{\rm meas} = \frac{\sqrt{2}\,n_f}{dE/E}\begin{pmatrix}r_1\\ r_2\end{pmatrix} + \mathbf{M}_m\begin{pmatrix}\eta_x\\ \eta_y\end{pmatrix}_{true} \tag{24.7}$$

The factor of $\sqrt{2}$ comes from the fact that there are two measurements.

24.2.3 Coupling Measurement

The coupling measurement is considered to be the result of measuring the beam at a detector over N_s turns while the beam oscillates at a normal mode frequency with some amplitude $A_{\rm osc}$. The measured coupling is computed as follows. First, consider excitation of the a-mode which can be written in the form:

$$\begin{pmatrix}x_i\\ y_i\end{pmatrix}_{\rm true} = A_{\rm osc}\begin{pmatrix}\cos\phi_i\\ K_{22a}\cos\phi_i + K_{12a}\sin\phi_i\end{pmatrix}_{\rm true} \tag{24.8}$$

i is the turn number and ϕ_i is the oscillation phase on the i^{th} turn. The coefficients K_{22a} and K_{12a} are related to the coupling $\overline{\mathbf{C}}$ via David and Rubin[Sagan99] Eq. 54:

$$\begin{aligned} K_{22a} &= \frac{-\sqrt{\beta_b}}{\gamma\sqrt{\beta_a}}\,\overline{\mathbf{C}}_{22}\\ K_{12a} &= \frac{-\sqrt{\beta_b}}{\gamma\sqrt{\beta_a}}\,\overline{\mathbf{C}}_{12}\end{aligned} \tag{24.9}$$

To apply the measurement errors, consider the general case where the beam's oscillations are split into two components: One component being in-phase with some reference oscillator (which is oscillating with the same frequency as the beam) and a component oscillating out-of-phase:

$$\begin{pmatrix}x_i\\ y_i\end{pmatrix}_{\rm true} = \begin{pmatrix}q_{a1x}\\ q_{a1y}\end{pmatrix}_{\rm true} A_{\rm osc}\,\cos(\phi_i + d\phi) + \begin{pmatrix}q_{a2x}\\ q_{a2y}\end{pmatrix}_{\rm true} A_{\rm osc}\,\sin(\phi_i + d\phi) \tag{24.10}$$

where $d\phi$ is the phase of the reference oscillator with respect to the beam. Comparing Eq. (24.8) with Eq. (24.10) gives the relation

$$\begin{aligned} K_{22a} &= \frac{q_{a1x}\,q_{a1y} + q_{a2x}\,q_{a2y}}{q_{a1x}^2 + q_{a2x}^2}\\ K_{12a} &= \frac{q_{a1x}\,q_{a2y} - q_{a2x}\,q_{a1y}}{q_{a1x}^2 + q_{a2x}^2}\end{aligned} \tag{24.11}$$

This equation is general and can be applied in either the true or measurement frame of reference. Eq. (24.3) can be used to transform $(x_i, y_i)_{\rm true}$ in Eq. (24.8) to the measurement frame of reference. Only the oscillating part is of interest. Averaging over many turns gives

$$\begin{pmatrix}q_{a1x}\\ q_{a1y}\end{pmatrix}_{\rm meas} = \mathbf{M}_m\begin{pmatrix}q_{a1x}\\ q_{a1y}\end{pmatrix}_{\rm true}, \qquad \begin{pmatrix}q_{a2x}\\ q_{a2y}\end{pmatrix}_{\rm meas} = \mathbf{M}_m\begin{pmatrix}q_{a2x}\\ q_{a2y}\end{pmatrix}_{\rm true} \tag{24.12}$$

This neglects the measurement noise. A calculation shows that the noise gives a contribution to the measured K_{22a} and K_{12a} of

$$K_{22a} \to K_{22a} + r_1\frac{n_f}{N_s\,A_{\rm osc}}, \qquad K_{12a} \to K_{12a} + r_2\frac{n_f}{N_s\,A_{\rm osc}} \tag{24.13}$$

Using the above equations, the transformation from the true coupling to measured coupling is as follows: From a knowledge of the true $\overline{\mathbf{C}}$ and Twiss values, the true K_{22a} and K_{12a} can be calculated via Eq. (24.9). Since the value of $d\phi$ does not affect the final answer, $d\phi$ in Eq. (24.10) is chosen to be zero. Comparing this to Eq. (24.8) gives

$$\begin{pmatrix} q_{a1x} \\ q_{a1y} \end{pmatrix}_{\text{true}} = \begin{pmatrix} 1 \\ K_{22a} \end{pmatrix}_{\text{true}}, \qquad \begin{pmatrix} q_{a2x} \\ q_{a2y} \end{pmatrix}_{\text{true}} = \begin{pmatrix} 0 \\ K_{12a} \end{pmatrix}_{\text{true}} \tag{24.14}$$

Now Eq. (24.12) is used to convert to the measured q's and Eq. (24.11) then gives the measured K_{22a} and K_{12a}. Finally, Applying Eq. (24.13) and then Eq. (24.9) gives the measured $\overline{\mathbf{C}}_{22}$ and $\overline{\mathbf{C}}_{12}$.

A similar procedure can be applied to b-mode oscillations to calculate values for the measured $\overline{\mathbf{C}}_{11}$ and $\overline{\mathbf{C}}_{12}$. K_{11b} and K_{12b} are defined by

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix}_{\text{true}} = A_{\text{osc}} \begin{pmatrix} K_{11b}\,\cos\phi_i + K_{12b}\sin\phi_i \\ \cos\phi_i \end{pmatrix}_{\text{true}} \tag{24.15}$$

Comparing this to David and Rubin[Sagan99] Eq. 55 gives

$$\begin{aligned} K_{11b} &= \frac{\sqrt{\beta_a}}{\gamma\sqrt{\beta_b}}\,\overline{\mathbf{C}}_{11} \\ K_{12b} &= \frac{-\sqrt{\beta_a}}{\gamma\sqrt{\beta_b}}\,\overline{\mathbf{C}}_{12} \end{aligned} \tag{24.16}$$

The q_{x1b}, q_{y1b}, q_{x2b} and q_{y2b} are defined by using Eq. (24.10) with the "a" subscript replaced by "b". The relationship between K and q is then

$$\begin{aligned} K_{11b} &= \frac{q_{b1y}\,q_{b1x} + q_{b2y}\,q_{b2x}}{q_{b1y}^2 + q_{b2y}^2} \\ K_{12b} &= \frac{q_{b1y}\,q_{b2x} - q_{b2y}\,q_{b1x}}{q_{b1y}^2 + q_{b2y}^2} \end{aligned} \tag{24.17}$$

24.2.4 Phase Measurement

Like the coupling measurement, the betatron phase measurement is considered to be the result of measuring the beam at a detector over N_s turns while the beam oscillates at a normal mode frequency with some amplitude A_{osc}. Following the analysis of the previous subsection, the phase ϕ is

$$\begin{pmatrix} \phi_a \\ \phi_b \end{pmatrix}_{\text{meas}} = \begin{pmatrix} \phi_a \\ \phi_b \end{pmatrix}_{true} + \frac{n_f}{N_s\,A_{\text{osc}}} \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} - \begin{pmatrix} \tan^{-1}\left(\frac{q_{a2x}}{q_{a1x}}\right) \\ \tan^{-1}\left(\frac{q_{b2y}}{q_{b1y}}\right) \end{pmatrix}_{\text{meas}} \tag{24.18}$$

Part III

Programmer's Guide

Chapter 25

Bmad Programming Overview

25.1 Manual Notation

Bmad defines a number of structures and these structures may contain components which are structures, etc. In order to keep the text in this manual succinct when referring to components, the enclosing structure name may be dropped. For example, the `lat_struct` structure looks like

```
type lat_struct
  character(40) name
  type (mode_info_struct) a, b, z
  type (lat_param_struct) param
  type (ele_struct), pointer ::  ele(:)
  type (branch_struct), allocatable :: branch(:)
  ... etc. ...
end type
```

In this example, "`%a`" could be used to refer to, the `a` component of the `lat_struct`. To make it explicit that this is a component of a `lat_struct`, "`lat_struct%a`" is an alternate possibility. Since the vast majority of structures have the "_struct" suffix, this may be shortened to "`lat%a`". A similar notation works for subcomponents. For example, a `branch_struct` looks like

```
type branch_struct
  character(40) name
  integer ix_from_ele                  ! Index of branching element
  integer, pointer :: n_ele_track      ! Number of tracking elements
  integer, pointer :: n_ele_max
  type (ele_struct), pointer :: ele(:) ! Element array
  ... etc. ...
end type
```

The `ele` component of the `branch` component of the `lat_struct` can be referred to using "`lat%branch%ele`", "`%branch%ele`", or "`%ele`". Potentially, the last of these could be confused with the "`lat%ele`" component so "`%ele`" would only be used if the meaning is unambiguous in the context.

25.2 The Bmad Libraries

The code that goes into a program based upon *Bmad* is divided up into a number of libraries. The *Bmad* web site has general information on the organization of these libraries including information on

obtaining and compiling programs. The *Bmad* web site is at:
https://www.classe.cornell.edu/bmad

The *Bmad* libraries are divided into two groups. One group of libraries contains "in-house" developed code. The other ''package'' libraries consist of "external" code that *Bmad* relies upon.

The in-house developed code libraries are:

bmad
The bmad library contains the routines for relativistic charged particle simulation including particle tracking, Twiss calculations, symplectic integration, etc., etc.

cpp_bmad_interface The cpp_bmad_interface library is for interfacing *Bmad* with C++. This library defines a set of C++ classes corresponding to the major *Bmad* structures. Along with this, the library contains conversion routines to move information between the C++ classes and the corresponding *Bmad* structures.

sim_utils
The sim_utils library contains a set of miscellaneous helper routines. Included are routines for string manipulation, file manipulation, matrix manipulation, spline fitting, Gaussian random number generation, etc.

The package libraries are:

forest
This is the PTC/FPP (Polymorphic Tracking Code / Fully Polymorphic Package) library of Étienne Forest that handles Taylor maps to any arbitrary order (this is also known as Truncated Power Series Algebra (TPSA)). See Chapter 34 for more details. FPP/PTC is a very general package and *Bmad* only makes use of a small part of its features. For more inform ation see the FPP/PTC manual[Forest02]. The core Differential Algebra (DA) package used by PTC was developed by Martin Berz[Berz89].

fftw
FFTW is a C subroutine library for computing the discrete Fourier transform in one or more dimensions. FFTW has a Fortran 2003 API.

gsl / fgsl
The Gnu Scientific Library (GSL), written in C, provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total. The FGSL library provides a Fortran interface to the GSL library.

hdf5
hdf5 is a library for for storing and managing data. In particular, *Bmad* uses this library for storing particle position data.

lapack / lapack95
lapack is a widely used package of linear algebra routines written in Fortran77. The lapack95 library provides a Fortran95 interface to lapack.

open_spacecharge
The open_spacecharge library provides low energy tracking with space charge effects.

PGPLOT
The pgplot Graphics Subroutine Library is a Fortran or C-callable, device-independent graphics package for making simple scientific graphs. Documentation including a user's manual may be obtained from the pgplot web site at

www.astro.caltech.edu/~tjp/pgplot.

One disadvantage of pgplot for the programmer is that it is not the most user friendly. To remedy this, there is a set of Fortran90 wrapper subroutines called quick_plot. The quick_plot suite is part of the sim_utils library and is documented in Chapter 37.

plplot
The plplot library is an updated version of pgplot. The plplot library can be used as a replacement for pgplot. The quick_plot suite, which is part of the sim_utils library and is documented in Chapter 37, provides wrapper routines for plplot to make things more programmer friendly.

recipes
Numerical Recipes is a set of subroutines for doing scientific computing including Runge–Kutta integration, FFTs, interpolation and extrapolation, etc., etc. The documentation for this library is the books "Numerical Recipes, in Fortran, The Art of Scientific Computing" and "Numerical Recipes in Fortran90, the Art of Parallel Scientific Computing"[Press92]. The first book explains how the subroutines work and the second book explains what the argument lists for the Fortran90 version of the subroutines are. You do need both books if you want to use Numerical Recipes. For *Bmad*, this library has been modified to handle double precision reals which is the standard for the other libraries (See §25.4). Additionally, some routines have been modified so that Numerical Recipes can be built as a shared object library. If you get a compiler complaint about a call to a Numerical Recipes routine missing a required argument, look in the code for that routine for documentation.

xraylib
The XRAYLIB library provides routines for obtaining parameters pertinent to the X-ray interaction with matter.

xsif
xsif is a library from SLAC to read in xsif format files. See §2.1 for more details. The only *Bmad* routine to use this library is xsif_parser.

25.3 Using getf and listf for Viewing Routine and Structure Documentation

As can be seen from the program example in Chapter 26 there is a lot going on behind the scenes even for this simple program. This shows that programming with *Bmad* can be both easy and hard. Easy in the sense that a lot can be done with just a few lines. The hard part comes about since there are many details that have to be kept in mind in order to make sure that the subroutines are calculating what you really want them to calculate.

To help with the details, all *Bmad* subroutines have in their source (.f90) files a comment block that explains the arguments needed by the subroutines and explains what the subroutine does. To help quickly access these comments, there are two Python scripts that are supplied with the *Bmad* distribution that are invoked with the commands listf and getf.

The getf command is used to locate routines and structures, and to type out information on them. The form of the command is

```
getf <name>
```

This searches for any routine or structure with the name <name>. <name> may contain the wild–cards "*" and "." where "*" matches to any number of characters and "." matches to any single character. For example:

```
getf bmad_parser
getf lat_struct
getf twiss_at_.
```

The third line in this example will match to the routine twiss_at_s but not the routine twiss_at_start. You may or may not have to put quotation marks if you use wild card characters. As an example, the command getf twiss_struct produces:

```
/home/cesrulib/cesr_libs/devel/cvssrc/bmad/modules/twiss_mod.f90
  type twiss_struct
    real(rp) beta, alpha, gamma, phi, eta, etap
    real(rp) sigma, emit
  end type
```

The first line shows the file where the structure is located (This is system and user dependent so don't be surprised if you get a different directory when you use getf). The rest of the output shows the definition of the twiss_struct structure. The result of issuing the command getf relative_tracking_charge is:

```
File: ../../bmad/modules/bmad_utils_mod.f90
!+
! Function relative_tracking_charge (orbit, param) result (rel_charge)
!
! Routine to determine the relative charge/mass of the particle being
! tracked relative to the charge of the reference particle.
!
! Input:
!   orbit -- coord_struct: Particle position structure.
!   param -- lat_param_struct: Structure holding the reference particle id.
!
! Output:
!   rel_charge -- real(rp): Relative charge/mass
!-
function relative_tracking_charge (orbit, param) result (rel_charge)
```

The first line again shows in what file the subroutine is located. The rest of the output explains what the routine does and how it can be called.

The getf command can also be used to search for global integer and real parameter constants. For example

```
getf c_light
```

will give the result:

```
File: ../sim_utils/interfaces/physical_constants.f90
    real(rp), parameter :: c_light = 2.99792458d8              ! speed of light
```

[Global constants are contants defined in a module that have global scope (defined before the contains statement).] For parameters whose name ends with a dollar sign "$" character (§25.5, the dollar sign suffix may be omitted in the search string. For example the search

```
getf quadrupole
```

will give the result

```
File: ../../bmad/modules/bmad_struct.f90
    integer, parameter :: drift$ = 1, sbend$ = 2, quadrupole$ = 3, group$ = 4, ...
```

Since the dollar sign is a special character for the Python regexp module used by `getf`, to include a dollar sign in the search string the dollar sign must be prefixed by three back slashes. Thus the search

```
getf quadrupole\\\$
```

will also locate the value of `quadrupole$`.

The `listf` command is like the `getf` command except that only the file name where a routine or structure is found is printed. The `listf` command is useful if you want to just find out where a routine or structure definition lives. For example, the `listf relative*` command would produce

```
File: ../../bmad/code/relative_mode_flip.f90
    function relative_mode_flip (ele1, ele2) result (rel_mode)

File: ../../bmad/modules/bmad_utils_mod.f90
    function relative_tracking_charge (orbit, param) result (rel_charge)
```

The way `getf` and `listf` work is that they search a list of directories to find the `bmad`, `sim_utils`, and `tao` libraries. Currently the libraries in the *Bmad* distribution that were not developed at Cornell are not searched. This is primarily due to the fact that, to save time, `getf` and `listf` make assumptions about how documentation is arranged in a file and the non–Cornell libraries do not follow this format.

25.4 Precision of Real Variables

Historically, *Bmad* come in two flavors: One version where the real numbers are single precision and a second version with double precision reals. Which version you are working with is controlled by the kind parameter `rp` (Real Precision) which is defined in the `precision_def` module. On most platforms, single precision translates to `rp` = 4 and double precision to `rp` = 8. The double precision version is used by default since round-off errors can be significant in some calculations. Long–term tracking is an example where the single precision version is not adequate. Changing the precision means recompiling all the libraries except `PTC` and `pgplot`. You cannot mix and match. Either you are using the single precision version or you are using the double precision version. Currently, *Bmad* is always compiled double precision and it is a near certainty that there would have to be some fixes if there was ever a need for compiling single precision.

To define floating point variables in Fortran with the correct precision, use the syntax ‘‘`real(rp)`’’. For example:

```
real(rp) var1, var2, var3
```

When you want to define a literal constant, for example to pass an argument to a subroutine, add the suffix `_rp` to the end of the constant. For example

```
var1 =  2.0_rp * var2
call my_sub (var1, 1.0e6_rp)
```

Note that `2_rp` is different from `2.0_rp`. `2_rp` is an integer of kind `rp`, not a real.

Independent of the setting of `rp`, the parameters `sp` and `dp` are defined to give single and double precision numbers respectively.

25.5 Programming Conventions

Bmad subroutines follow the following conventions:

A “$” suffix denotes a parameter: A dollar sign “$” at the end of a name denotes an parameter. For example, in the above program, to check whether an element is a quadrupole one would write:

```
if (lat%ele(i)%key == quadrupole$) ...
```

Checking the source code one would find in the module `bmad_struct`

```
integer, parameter :: drift$ = 1, sbend$ = 2, quadrupole$ = 3, group$ = 4
```

One should always use the parameter name instead of the integer it represents. That is, one should never write

```
if (lat%ele(i)%key == 3) ...  ! DO NOT DO THIS!
```

For one, using the name makes the code clearer. However, more importantly, the integer value of the parameters may at times be shuffled for practical internal reasons. The use of the integer value could thus lead to disastrous results.

By convention all names ending in "$" are parameters. And most "dollar sign" parameters are integers but there are exceptions. For example, the parameter `real_garbage$` is a real number. To find the value of a dollar sign parameter, the `getf` or `listf` (§25.3) commands can be used.

Structure names have a "_struct" suffix: For example: `lat_struct`, `ele_struct`, etc. Structures without a `_struct` are usually part of Étienne's PTC/FPP package.

Chapter 26

An Example Bmad Based Program

To get the general feel for how *Bmad* works before getting into the nitty–gritty details in subsequent chapters, this chapter analyzes an example test program.

26.1 Programming Setup

Information on how to setup your work envirnoment for the compiling and linking of programs can be obtained from the *Bmad* web site at:

https://www.classe.cornell.edu/bmad

In particular, instructions for compling, linking, and running of the example program can be obtained from the web page:

https://www.classe.cornell.edu/bmad

See the section titled `Example Compiling/Linking and Running of a Bmad Based Program`.

26.2 A First Program

Consider the example program shown in Fig. 26.1. The source code for this program is provided with *Bmad* in the directory:

```
$ACC_ROOT_DIR/examples/simple_bmad_program
```

[Note: `$ACC_ROOT_DIR` is the root directory of the *Bmad* `Distribution` when you are running "off-site" and is root directory of the `Release` when you are running "on-site".]

```
1  program test
2
3  use bmad                ! Define the structures we need to know about.
4  implicit none
5  type (lat_struct), target :: lat   ! This structure holds the lattice info
6  type (ele_struct), pointer :: ele, cleo
7  type (ele_pointer_struct), allocatable :: eles(:)
8  type (all_pointer_struct) a_ptr
9  integer i, ix, n_loc
10 logical err
11
12 ! Programs should always implement "intelligent bookkeeping".
13 bmad_com%auto_bookkeeper = .false.
14
15 ! Read in a lattice, and modify the ks solenoid strength of "cleo_sol".
16
17 call bmad_parser ("lat.bmad", lat)  ! Read in a lattice.
18
19 call lat_ele_locator ('CLEO_SOL', lat, eles, n_loc, err)  ! Find element
20 cleo => eles(1)%ele                        ! Point to cleo_sol element.
21 call pointer_to_attribute (cleo, 'KS', .true., a_ptr, err) ! Point to KS attribute.
22 a_ptr%r = a_ptr%r + 0.001         ! Modify KS component.
23 call set_flags_for_changed_attribute (cleo, a_ptr%r)
24 call lattice_bookkeeper (lat)
25 call lat_make_mat6 (lat, cleo%ix_ele)      ! Remake transfer matrix
26
27 ! Calculate starting Twiss params if the lattice is closed,
28 ! and then propagate the Twiss parameters through the lattice.
29
30 if (lat%param%geometry == closed$) call twiss_at_start (lat)
31 call twiss_propagate_all (lat)      ! Propagate Twiss parameters
32
33 ! Print info on the first 11 elements
34
35 print *, ' Ix  Name              Ele_type                    S      Beta_a'
36 do i = 0, 10
37   ele => lat%ele(i)
38   print '(i4,2x,a16,2x,a,2f12.4)', i, ele%name, key_name(ele%key), ele%s, ele%a%beta
39 enddo
40
41 ! print information on the CLEO_SOL element.
42
43 print *
44 print *, '!-----------------------------------------------------------'
45 print *, '! Information on element: CLEO_SOL'
46 print *
47 call type_ele (cleo, .false., 0, .false., 0, .true., lat)
48
49 deallocate (eles)
50
51 end program
```

Figure 26.1: Example Bmad program

26.3 Explanation of the Simple_Bmad_Program

A line by line explanation of the example program follows. The `use bmad` statement at line 3 defines the *Bmad* structures and defines the interfaces (argument lists) for the *Bmad* subroutines. In particular, the `lat` variable (line 5), which is of type `lat_struct` (§28.3), holds all of the lattice information: The list of elements, their attributes, etc. The setting of `bmad_com%auto_bookkeeper` to `False` in line 13 enables the "intelligent" bookkeeping of lattice attributes as discussed in §28.6). The call to bmad_parser (line 17) causes the lattice file `lat.bmad` to be parsed and the lattice information is stored the `lat` variable. Note: To get a listing of the `lat_struct` components or to find out more about `bmad_parser` use the `getf` command as discussed in §25.3.

The routine lat_ele_locator (§29.3) is used in line 19 to find the element in the lattice with the name `CLEO_SOL`. Line 20 defines a pointer variable named `cleo` which is used here as shortcut notation rather than having to write `eles(1)%ele` when refering to this element. The call to `pointer_to_attribute` in line 21 sets up a pointer structure `a_ptr%r` to point to the `KS` solenoid strength component of `cleo`. [`a_ptr` also has components `a_ptr%i` and `a_ptr%l` to point to integrer or logical components if needed. See §29.4.]

Line 22 changes the `ks` solenoid strength of `cleo`. Since an element attribute has been changed, the call to set_flags_for_changed_attribute in line 23 is needed for *Bmad* to inform *Bmad* that this attribute has changed and the call to lattice_bookkeeper does the necessary lattice bookkeeping (§28.6).

The call to lat_make_mat6 in line 25 recalculates the linear transfer matrix for the `CLEO_SOL` element.

In line 30, the program checks if the lattice is circular (§8.1) and, if so, uses the routine twiss_at_start to multiply the transfer matrices of the individual elements together to form the 1–turn matrix from the start of the lat back to the start. From this matrix `twiss_at_start` calculates the Twiss parameters at the start of the lattice and puts the information into `lat%ele(0)` (§31.2). The next call, to twiss_propagate_all, takes the starting Twiss parameters and, using the transfer matrices of the individual elements, calculates the Twiss parameters at all the elements. Notice that if the lattice is not circular, The starting Twiss parameters will need to have been defined in the lattice file.

The program is now ready output some information. Lines 24 through 28 of the program print information on the first 11 elements in the lattice. The do-loop is over the array `lat%ele(:)`. Each element of the array holds the information about an individual lattice element as explained in Chapter 28. The `lat%ele(0)` element is basically a marker element to denote the beginning of the array (§6). Using the pointer `ele` to point to the individual elements (line 37) makes for a cleaner syntax and reduces typing. The table that is produced is shown in lines 1 through 12 of Fig. 26.2. The first column is the element index i. The second column, `ele%name`, is the name of the element. The third column, key_name(elethe name of the element class. `ele%key` is an integer denoting what type of element (quadrupole, wiggler, etc.) it is. `key_name` is an array that translates the integer key of an element to a printable string. The fourth column, `ele%s`, is the longitudinal position at the exit end of the element. Finally, the last column, `ele%x%beta`, is the a–mode (nearly horizontal mode) beta function.

The type_ele routine on line 47 of the program is used to type out the `CLEO_SOL`'s attributes and other information as shown on lines 14 through 41 of the output (more on this later).

This brings us to the lattice file used for the input to the program. The call to `bmad_parser` shows that this file is called `simple_bmad_program/lat.bmad`. In this file there is a call to another file

```
call, file = "layout.bmad"
```

It is in this second file that the layout of the lattice is defined. In particular, the `line` used to define the element order looks like

```
cesr: line = (IP_L0, d001, DET_00W, d002, Q00W, d003, ...)
```

```
use, cesr
```

If you compare this to the listing of the elements in Fig. 26.2 you will find differences. For example, element #2 in the program listing is named `CLEO_SOL\3`. From the definition of the `cesr` line this should be `d001` which, if you look up its definition in `layout.bmad` is a drift. The difference between lattice file and output is due to the presence the `CLEO_SOL` element which appears in `lat.bmad`:

```
ks_solenoid    := -1.0e-9 * clight * solenoid_tesla / beam[energy]
cleo_sol: solenoid, l = 3.51, ks = ks_solenoid, superimpose
```

The solenoid is 3.51 meters long and it is superimposed upon the lattice with its center at $s = 0$ (this is the default if the position is not specified). When `bmad_parser` constructs the lattice list of elements the superposition of `IP_L0`, which is a zero–length marker, with the solenoid does not modify `IP_L0`. The superposition of the `d001` drift with the solenoid gives a solenoid with the same length as the drift. Since this is a "new" element, `bmad_parser` makes up a name that reflects that it is basically a section of the solenoid it came from. Next, since the `CLEO_SOL` element happens to only cover part of the `Q00W` quadrupole, `bmad_parser` breaks the quadrupole into two pieces. The piece that is inside the solenoid is a `sol_quad` and the piece outside the solenoid is a regular quadrupole. See §7.1 for more details. Since the center of the `CLEO_SOL` is at $s = 0$, half of it extends to negative s. In this situation, `bmad_parser` will wrap this half back and superimpose it on the elements at the end of the lattice list near $s = s_{lat}$ where s_{lat} is the length of the lattice. As explained in Chapter 28, the lattice list that is used for tracking extends from `lat%ele(0)` through `lat%ele(n)` where `n = lat%n_ele_track`. The `CLEO_SOL` element is put in the section of `lat%ele(n)` with `n > lat%n_ele_track` since it is not an element to be tracked through. The `Q00W` quadrupole also gets put in this part of the list. The bookkeeping information that the `cleo_sol\3` element is derived from the `cleo_sol` is put in the `cleo_sol` element as shown in lines 33 through 41 of the output. It is now possible in the program to vary, say, the strength of the `ks` attribute of the `CLEO_SOL` and have the `ks` attributes of the dependent ("`super_slave`") elements updated with one subroutine call. For example, the following code increases the solenoid strength by 1% lattice_bookkeeper lat_ele_locator

```
call lat_ele_locator ('CLEO_SOL', lat, eles, n_loc, err)
eles(1)%ele(ix)%value(ks$) = eles(1)%ele%value(ks$) * 1.01
call lattice_bookkeeper (lat)
```

Bmad takes care of the bookkeeping. In fact `control_bookkeeper` is automatically called when transfer matrices are remade so the direct call to `control_bookkeeper` may not be necessary.

Running the program (§26.1) gives the output as shown in Fig. 26.2.

```
    Ix  Name              Ele_type                  S      Beta_a
     0  BEGINNING         BEGINNING_ELE        0.0000      0.9381
     1  IP_L0             MARKER               0.0000      0.9381
     2  CLEO_SOL#3        SOLENOID             0.6223      1.3500
     3  DET_00W           MARKER               0.6223      1.3500
     4  CLEO_SOL#4        SOLENOID             0.6380      1.3710
     5  Q00W\CLEO_SOL     SOL_QUAD             1.7550      7.8619
     6  Q00W#1            QUADRUPOLE           2.1628     16.2350
     7  D003              DRIFT                2.4934     27.4986
     8  DET_01W           MARKER               2.4934     27.4986
     9  D004              DRIFT                2.9240     46.6018
    10  Q01W              QUADRUPOLE           3.8740     68.1771

  !----------------------------------------------------------
  ! Information on element: CLEO_SOL

  Element #              871
  Element Name: CLEO_SOL
  Key: Solenoid
  S_start, S:    766.671421,      1.755000
  Ref_time:  5.854050E-09

  Attribute values [Only non-zero/non-default values shown]:
   1   L                          =  3.5100000E+00 m
   5   KS                         = -8.5023386E-02 1/m
  10   FRINGE_TYPE                =  None (1)
  11   FRINGE_AT                  =  Both_Ends (3)
  13   SPIN_FRINGE_ON             =  T (1)
  31   L_HARD_EDGE                =  3.5100000E+00 m
  47   PTC_CANONICAL_COORDS       =  1.0000000E+00
  49   BS_FIELD                   =  1.5000000E+00 T
  50   DELTA_REF_TIME             =  1.1708100E-08 sec
  53   P0C                        =  5.2890000E+09 eV      BETA            =  0.999999995
  54   E_TOT                      =  5.2890000E+09 eV      GAMMA           =  1.0350315E+04
  66   NUM_STEPS                  = 18
  67   DS_STEP                    =  2.0000000E-01 m

       TRACKING_METHOD            =  Bmad_Standard       APERTURE_AT              =  Exit_End
       MAT6_CALC_METHOD           =  Bmad_Standard       APERTURE_TYPE            =  Rectangular
       SPIN_TRACKING_METHOD       =  Tracking            OFFSET_MOVES_APERTURE    =  F
       PTC_INTEGRATION_TYPE       =  Matrix_Kick         SYMPLECTIFY              =  F
       CSR_METHOD                 =  Off                 FIELD_MASTER             =  F
       SPACE_CHARGE_METHOD        =  Off
       FIELD_CALC                 =  Bmad_Standard

Slave_status: Free

Lord_status:  Super_Lord
Slaves:
   Index   Name           Type
     865   Q00E\CLEO_SOL  Sol_Quad
     866   CLEO_SOL#1     Solenoid
     868   CLEO_SOL#2     Solenoid
       2   CLEO_SOL#3     Solenoid
       4   CLEO_SOL#4     Solenoid
       5   Q00W\CLEO_SOL  Sol_Quad

Twiss at end of element:
                         A              B           Cbar                        C_mat
Beta (m)         7.73293815    88.01448113  | -0.16691726   0.00910908     -0.05360747   0.23764236
Alpha           -6.94661336    -1.53007417  |  2.24946759  -0.02027746      0.03925732   0.14506787
Gamma (1/m)      6.36956306     0.03796110  |  Gamma_c =   1.00851669       Mode_Flip = F
Phi (rad)        1.00744612     1.55387058            X              Y              Z
```

Chapter 27

The ele_struct

This chapter describes the `ele_struct` which is the structure that holds all the information about an individual lattice element: quadrupoles, separators, wigglers, etc. The `ele_struct` structure is shown in Figs. 27.1 and 27.2. This structure is somewhat complicated, however, in practice, a lot of the complexity is generally hidden by the *Bmad* bookkeeping routines.

As a general rule, for variables like the Twiss parameters that are not constant along the length of an element, the value stored in the corresponding component in the `ele_struct` is the value at the downstream end of the element.

For printing information about an element, the type_ele or type_ele routines can be used (§26.2). The difference between the two is that `type_ele` will print to the terminal window while `type_ele` will return an array of strings containing the element information.

```
type ele_struct
  character(40) name                      ! name of element \sref{c:ele.string}.
  character(40) type                      ! type name \sref{c:ele.string}.
  character(40) alias                     ! Another name \sref{c:ele.string}.
  character(40) component_name            ! Used by overlays, multipass patch, etc.
  character(200), pointer :: descrip      ! Description string.
  type (twiss_struct)  a, b, z            ! Twiss parameters at end of element \sref{c:normal.modes}.
  type (xy_disp_struct) x, y              ! Projected dispersions \sref{c:normal.modes}.
  type (ac_kicker_struct), pointer :: ac_kick  ! ac_kicker element parameters.
  type (bookkeeping_state_struct) bookkeeping_state ! Element attribute bookkeeping
  type (branch_struct), pointer :: branch      ! Pointer to branch containing element.
  type (controller_struct), pointer :: control ! For group and overlay elements.
  type (cartesian_map_struct), pointer :: cartesian_map(:)     ! Used to define DC fields
  type (cylindrical_map_struct), pointer :: cylindrical_map(:) ! Used to define DC fields
  type (ele_struct), pointer :: lord           ! Pointer to a slice lord.
  type (taylor_field_struct), pointer :: taylor_field(:)       ! Used to define DC and AC fields.
  type (grid_field_struct), pointer :: grid_field(:)           ! Used to define DC and AC fields.
  type (fibre), pointer :: ptc_fiber           ! PTC tracking.
  type (floor_position_struct) floor           ! Global floor position.
  type (ptc_genfield_struct), pointer :: ptc_genfield  ! For symp_map
  type (mode3_struct), pointer :: mode3        ! Full 6-dimensional normal mode decomposition.
  type (photon_element_struct), pointer :: photon
  type (rad_int_ele_cache_struct), pointer :: rad_int_cache
                                               ! Radiation integral calc cached values
  type (space_charge_struct), pointer :: space_charge
  type (taylor_struct) :: taylor(6)            ! Orbital Taylor map.
  type (taylor_struct) :: spin_taylor(0:3)     ! Spin Taylor map.
  type (wake_struct), pointer :: wake    ! Wakes
    ele_struct definition continued on next figure...
```

Figure 27.1: The `ele_struct`. structure definition. The complete structure is shown in this and the following figure.

```
    ... ele_struct definition continued from previous figure.
  type (wall3d_struct) :: wall3d               ! Chamber or capillary wall
  type(coord_struct) map_ref_orb_in            ! Transfer map ref orbit at upstream end of element.
  type(coord_struct) map_ref_orb_out           ! Transfer map ref orbit at downstream end of element.
  type(coord_struct) time_ref_orb_in           ! Reference orbit at upstream end for ref_time calc.
  type(coord_struct) time_ref_orb_out          ! Reference orbit at downstream end for ref_time calc.
  real(rp) value(num_ele_attrib$)              ! attribute values.
  real(rp) old_value(num_ele_attrib$)          ! Used to see if %value(:) array has changed.
  real(rp) vec0(6)                             ! 0th order transport vector.
  real(rp) mat6(6,6)                           ! 1st order transport matrix.
  real(rp) c_mat(2,2)                          ! 2x2 C coupling matrix
  real(rp) gamma_c                             ! gamma associated with C matrix
  real(rp) s_start                             ! longitudinal ref position at entrance_end
  real(rp) s                                   ! longitudinal position at the downstream end.
  real(rp) ref_time                            ! Time ref particle passes downstream end.
  real(rp), pointer :: r(:,:,:)                ! For general use. Not used by Bmad.
  real(rp), pointer :: a_pole(:)               ! multipole
  real(rp), pointer :: b_pole(:)               ! multipoles
  real(rp), pointer :: a_pole_elec(:)          ! Electrostatic multipoles.
  real(rp), pointer :: b_pole_elec(:)          ! Electrostatic multipoles.
  real(rp), pointer :: custom(:)               ! Custom attributes
  integer key                    ! key value
  integer sub_key                ! Records bend input type (rbend$, sbend$).
  integer ix_ele                 ! Index in lat%branch(n)%ele(:) array [n = 0 <==> lat%ele(:)].
  integer ix_branch              ! Index in lat%branch(:) array [0 => In lat%ele(:)].
  integer lord_status            ! overlay_lord$, etc.
  integer n_slave                ! Number of slaves
  integer n_slave_field          ! Number of field slaves
  integer ix1_slave              ! Pointer to lat%control array
  integer slave_status           ! super_slave$, etc.
  integer n_lord                 ! Number of lords
  integer n_lord_field           ! Number of field lords
  integer ic1_lord               ! Pointer to lat%ic array.
  integer ix_pointer             ! For general use. Not used by Bmad.
  integer ixx, iyy               ! Index for Bmad internal use
  integer mat6_calc_method       ! bmad_standard$, taylor$, etc.
  integer tracking_method        ! bmad_standard$, taylor$, etc.
  integer spin_tracking_method   ! bmad_standard$, symp_lie_ptc$, etc.
  integer ptc_integration_type   ! drift_kick$, matrix_kick$, etc.
  integer field_calc             ! Used with Runge-Kutta integrators.
  integer aperture_at            ! Aperture location: exit_end$, ...
  integer aperture_type          ! Type of aperture: rectanular$, elliptical$, or custom$.
  integer orientation            ! -1 -> Element is longitudinally reversed. +1 -> Normal.
  logical symplectify            ! Symplectify mat6 matrices.
  logical mode_flip              ! Have the normal modes traded places?
  logical multipoles_on          ! For turning multipoles on/off
  logical scale_multipoles       ! multipole components scaled by the strength of element?
  logical taylor_map_includes_offsets       ! Taylor map calculated with element offsets?
  logical field_master           ! Calculate strength from the field value?
  logical is_on                  ! For turning element on/off.
  logical logic                  ! For general use. Not used by Bmad.
  logical bmad_logic             ! For Bmad internal use only.
  logical select                 ! For element selection. Used by make_hybrid_ring, etc.
  logical csr_method             ! Coherent synchrotron radiation calculation
  logical space_charge_method    ! Space charge method.
  logical offset_moves_aperture  ! element offsets affects aperture?
end type
```

Figure 27.2: The `ele_struct`. The complete structure is shown in this and the preceding figure.

27.1 Initialization and Pointers

The ele_struct has a number of components and subcomponents that are pointers and this raises a deallocation issue. Generally, most ele_struct elements are part of a lat_struct variable (§28.2) and such elements in a lat_struct are handled by the lat_struct allocation/deallocation routines. In the case where a local ele_struct variable is used within a subroutine or function, the ele_struct variable must either be defined with the save attribute

```
type (ele_struct), save :: ele          ! Use the save attribute
logical, save :: init_needed = .false.
...
if (init_needed) then
  call init_ele (ele, quadrupole$)     ! Initialize element once
  init_needed = .false.
endif
```

or the pointers within the variable must be deallocated with a call to deallocate_ele_pointers:

```
type (ele_struct) ele
...
call init_ele (ele, sbend$)            ! Initialize element each time
...
call deallocate_ele_pointers (ele) ! And deallocate.
```

In the "normal" course of events, the pointers of an ele_struct variable should not be pointing to the same memory locations as the pointers of any other ele_struct variable. To make sure of this, the equal sign in the assignment ele1 = ele2 is overloaded by the routine ele_equal_ele. The exception here are the "Electro-magnetic field component" pointers ele%wig_term, ele%em_field%mode(:)%map, and ele%em_field%mode(:)%grid. Since these components potentially contain large arrays, and since the individual sub-components of these components are not likely to be individually modified, The field component pointers of ele1 and ele2 after the set ele1 = ele2 will point at the same memory locations.

Note: The assignment ele1 = ele2 will not modify ele1%ix_ele or ele1%ix_branch. If ele1 is associated with a lattice then ele1%lat will also be unaffected.

27.2 Element Attribute Bookkeeping

When a value of an attribute in an element changes, the values of other attributes may need to be changed (§4.1). Furthermore, in a lattice, changes to one element may necessitate changes to attribute values in other elements. For example, changing the accelerating gradient in an lcavity will change the reference energy throughout the lattice.

The attribute bookkeeping for a lattice can be complicated and, if not done intelligently, can cause programs to be slow if attributes are continually being changed. In order to keep track what bookkeeping has been done, the ele%status component is used by the appropriate bookkeeping routines for making sure the bookkeeping overhead is keep to a minimum. However, "intelligent" bookkeeping is only done if explicitly enabled in a program. See §28.6 for more details.

27.3 String Components

The %name, %type, %alias, and %descrip components of the ele_struct all have a direct correspondence with the name, type, alias, and descrip element attributes in an input lattice file (§4.3). On

input (§30.1), from a lattice file, `name`, `type`, and `alias` attributes will be converted to to uppercase before being loaded into an `ele_struct`. To save memory, since `%descrip` is not frequently used, `%descrip` is a pointer that is only allocated if `descrip` is set for a given element.

27.4 Element Key

The `%key` integer component gives the class of element (`quadrupole`, `rfcavity`, etc.). In general, to get the corresponding integer parameter for an element class, just add a "$" character to the class name. For example `quadrupole$` is the integer parameter for `quadrupole` elements. The `key_name` array converts from integer to the appropriate string. For example:

```
type (ele_struct) ele
if (ele%key == wiggler$) then       ! Test if element is a wiggler.
print *, 'This element: ', key_name(ele%key) ! Prints, for example, 'WIGGLER'
```

Note: The call to `init_ele` is needed for any `ele_struct` defined outside of a `lat_struct` structure.

The `%sub_key` component is only used for bend element. When a lattice file is parsed, (§30.1), all `rbend` elements are converted into `sbend` elements (§3.5). To keep track of what the original definition of the element was, the `%sub_key` component will be set to `sbend$` or `rbend$` whatever is appropriate. The `%sub_key` component does not affect any calculations and is only used in the routines that recreate lattice files from a `lat_struct` (§30.3).

27.5 The %value(:) array

Most of the real valued attributes of an element are held in the `%value(:)` array. For example, the value of the `k1` attribute for a quadrupole element is stored in `%value(k1$)` where `k1$` is an integer parameter that *Bmad* defines. In general, to get the correct index in `%value(:)` for a given attribute, add a "$" as a suffix. To convert from an attribute name to its index in the `%value` array use the attribute_index routine. To go back from an index in the `%value` array to a name use the attribute_name routine. Example:

```
type (ele_struct) ele
call init_ele (ele, quadrupole$)    ! Initialize element
ele%value(k1$) = 0.3                                    ! Set K1 value
print *, 'Index for Quad K1:  ', attribute_index(ele, 'K1') ! prints: '4' (= k1$)
print *, 'Name for Quad k1$: ', attribute_name (ele, k1$)   ! prints: 'K1'
```

The list of attributes for a given element type is given in the writeup for the different element in Chapter 3.

To obtain a list of attribute names and associated `%value(:)` indexes, the program `element_attributes` can be used. This program is included in the standard *Bmad* distribution.

Besides real valued attributes, the `value(:)` array also holds logical, integer, and, as explained below, "switch" attributes. To find out the type of a given attribute, use the function attribute_type. See the routine `type_ele` for an example of how `attribute_type` is used.

An example of a logical attribute is the `flexible` logical of match elements which is stored in `%value(flexible$)`. To evaluate logical attributes, the functions is_true(param) or is_false(param) should be used.

Integer attributes stored in the `value(:)` array include `n_slice` (stored in `%value(n_slice$)`). With integer attributes, the `nint(param)` Fortran instrinsic should be used for evaluation.

A switch attribte is an attribute whose value is one of a certain set of integers where each integer corresponds to some "state". For example, the `fringe_at` switch which, as explained in §4.20, may have

one of four values. Generally, the integer parameters that correspond to the states of a switch can be constructed by putting a "$" after the associated name. Thus, with the `fringe_at` switch, the four integer parameters are `no_end$`, `both_ends$`, `entrance_end$`, and `exit_end$`. For example:

```
if (nint(ele%value(fringe_type$)) == soft_edge_only$) then
  ...
```

For printing purposes, to convert a switch value to the appropriate string, use the routineswitch_attrib_value_name can be used.

The `%field_master` logical within an element sets whether it is the normalized strength or field strength that is the independent variable. See §4.1 for more details.

The `%old_value(:)` component of the `ele_struct` is used by the attribute_bookkeeper routine to check for changes for changes in the `%value(:)` array since the last time the `attribute_bookkeeper` routine had been called. If nothing has been changed, the `attribute_bookkeeper` routine knows not to waste time recalculating dependent values. Essentially what this means is that the `%old_value(:)` array should not be modified outside of `attribute_bookkeeper`.

27.6 Connection with the Lat_Struct

If an element is part of a `lat_struct` (§28), the `%ix_ele` and `%ix_branch` components of the `ele_struct` identify where the element is. Additionally, the `%lat` component will point to the encomposing lattice. That is

```
type (lat_struct), pointer :: lat
type (ele_struct), pointer :: ele2
if (ele%ix_ele > -1) then
  ie = ele%ix_ele
  ib = ele%ix_branch
  lat => ele%lat
  ele2 => lat%branch(ib)%ele(ie)
  print *, associated(ele2, ele)  ! Will print True.
endif
```

In this example the `ele2` pointer is constructed to point to the `ele` element. The test (`ele%ix_ele` > -1) is needed since `ele_struct` elements may exist outside of any `lat_struct` instance. Such "external" elements always have `%ix_ele` < 0. A value for `%ix_ele` of -2 is special in that it prevents the deallocate_ele_pointers routine from deallocating the pointers of an element which has its `%ix_ele` set to -2.

An element "slice" is an example of an element that exists external to any `lat_struct` instance. A slice is an `ele_struct` instance that represents some sub-section of a given element. Element slices are useful when tracking particles only part way through an element (§32.7).

27.7 Limits

The aperture limits (§4.8) in the `ele_struct` are:

```
%value(x1_limit$)
%value(x2_limit$)
%value(y1_limit$)
%value(y2_limit$)
```

The values of these limits along with the `%aperture_at`, `%aperture_type`, and `%offset_moves_aperture` components are used in tracking to determine if a particle has hit the vacuum chamber wall. See Section §32.8 for more details.

27.8 Twiss Parameters, etc.

The components %a, %b, %z, %x, %y, %c_mat, %gamma_c, %mode_flip, and mode3 hold information on the Twiss parameters, dispersion, and coupling at the downstream end of the element. See Chapter 31 for more details.

27.9 Element Lords and Element Slaves

In *Bmad*, elements in a lattice can control other elements. The components that determine this control are:

```
%slave_status
%n_slave
%n_slave_field
%ix1_slave
%lord_status
%n_lord
%n_lord_field
%ic1_lord
%component_name
```

This is explained fully in the chapter on the lat_struct (§28).

27.10 Group and Overlay Controller Elements

Group and overlay elements use the %control_var(:) array for storing information about the control variables. Each element in the array represents a single variable. %control_var(:) is an array of controller_var_struct structures and these structures look like:

```
type controller_struct
  character(40) :: name = ''
  real(rp) :: value = 0
  real(rp) :: old_value = 0
end type
```

The %old_value component is only used for group elements.

See Section §29.2 for an example of setting up a controller element within a program.

27.11 Coordinates, Offsets, etc.

The "upstream" and "downstream" ends of an element are, by definition, where the physical ends of the element would be if there were no offsets. In particular, if an element has a finite z_offset, the physical ends will be displaced from upstream and downstream ends. See §32.2 for more details.

The %floor component gives the "laboratory" global "floor" coordinates (§14.2) at the downstream end of the element. These coordinates are computed without misalignments. That is, the coordinates are not "body" coordinates. The components of the %floor structure are

```
type floor_position_struct
  real(rp) r(3)               ! Offset from origin
```

```
    real(rp) w(3,3)             ! Orientation matrix (Eq. (14.2))
    real(rp) theta, phi, psi    ! Angular orientation
  end type
```

The routine ele_geometry will calculate an element's floor coordinates given the floor coordinates at the beginning of the element. In a lattice, the lat_geometry routine will calculate the floor coordinates for the entire lattice using repeated calls to `ele_geometry`.

The positional offsets (§4.6) for an element from the reference orbit are stored in

```
%value(x_offset$)
%value(y_offset$)
%value(z_offset$)
%value(x_pitch$)
%value(y_pitch$)
%value(tilt$)
```

If the element is supported by a `girder` element (§3.20) then the `girder` offsets are added to the element offsets and the total offset with respect to the reference coordinate system is stored in:

```
%value(x_offset_tot$)
%value(y_offset_tot$)
%value(z_offset_tot$)
%value(x_pitch_tot$)
%value(y_pitch_tot$)
%value(tilt_tot$)
```

If there is no `girder`, the values for `%value(x_offset_tot$)`, etc. are set to the corresponding values in `%value(x_offset$)`, etc. Thus, to vary the position of an individual element the values of `%value(x_offset$)`, etc. are changed and to read the position of an element a program should look at `%value(x_offset_tot$)`, etc.

The longitudinal position at the downstream end of an element is stored in `%s` and the reference time is stored in `%ref_time`. This reference time is calculated assuming that the reference time is zero at the start of the lattice. Also stored in the `ele_struct` is the reference time at the start of the element and the differnece in the reference time between the end and the beginning. These are given in `%value(ref_time_start$)` and `%value(delta_ref_time$)` respectively.

Notice that the reference time used to calculate the z phase space coordinate (Eq. (14.28)) may be different from `%ref_time`. For example, with multiple bunches the z phase space coordinate is generally taken to be with respect to a reference particle at the center of the bunch the particle is in. And, at a given element, the reference time of the different bunch reference particles will be different. Another example happens when a particle is tracked through multiple turns. In this case the reference time at a given element will depend upon the turn number.

27.12 Transfer Maps: Linear and Non-linear (Taylor)

The routine make_mat6 computes the linear transfer matrix (Jacobian) along with the zeroth order transfer vector. This matrix is stored in `%mat6(6,6)` and the zeroth order vector is stored in `%vec0(6)`. The reference orbit at the upstream end of the element about which the transfer matrix is computed is stored in `%map_ref_orb_in` and the reference orbit at the downstream end is stored in `%map_ref_orb_out`. In the calculation of the transfer map, the vector `%vec0` is set so that

```
map_ref_orb_out = %mat6 * map_ref_orbit_in + %vec0
```

The reason redundant information is stored in the element is to save computation time.

To compute the transfer maps for an entire lattice use the routine lat_make_mat6.

The Taylor map (§5) for an element is stored in `%taylor(1:6)`. Each `%taylor(i)` is a `taylor_struct` structure that defines a Taylor series:

```
type taylor_struct
  real (rp) ref
  type (taylor_term_struct), pointer :: term(:) => null()
end type
```

Each Taylor series has an array of `taylor_term_struct` terms defined as

```
type taylor_term_struct
  real(rp) :: coef
  integer :: exp(6)
end type
```

The coefficient for a Taylor term is stored in `%coef` and the six exponents are stored in `%exp(6)`.

To see if there is a Taylor map associated with an element the association status of `%taylor(1)%term` needs to be checked. As an example the following finds the order of a Taylor map.

```
type (ele_struct) ele
...
if (associated(ele%taylor(1)%term) then  ! Taylor map exists
  taylor_order = 0
  do i = 1, 6
    do j = 1, size(ele%taylor(i)%term)
      taylor_order = max(taylor_order, sum(ele%taylor(i)%term(j)%exp)
    enddo
  enddo
else  ! Taylor map does not exist
  taylor_order = -1  ! flag non-existence
endif
```

The Taylor map is made up around some reference phase space point corresponding to the coordinates at the upstream of the element. This reference point is saved in `%taylor(1:6)%ref`. Once a Taylor map is made, the reference point is not needed in subsequent calculations. However, the Taylor map itself will depend upon what reference point is chosen (§21.1).

When using the `symp_map$` tracking method (§5.1), the pointer to the partially inverted Taylor map is stored in the `%gen_field` component of the `ele_struct`. The actual storage of the map is handled by the PTC library (§25.2). The PTC partially inverted map does not have any zeroth order terms so the zeroth order terms are stored in the `%gen0(6)` vector.

27.13 Reference Energy and Time

The reference energy and reference time are computed around a reference orbit which is different from the reference orbit used for computing transfer maps (§27.12). The energy and time reference orbit for an element is stored in

```
ele%time_ref_orb_in        ! Reference orbit at upstream end
ele%time_ref_orb_out       ! Reference orbit at downstream end
```

Generally `ele%time_ref_orb_in` is the zero orbit. The exception comes when an element is a `super_slave`. In this case, the reference orbit through the super_slaves of a given `super_lord` is constructed to be continuous. This is done for consistancey sake. For example, to ensure that when a marker is superimposed on top of a wiggler the reference orbit, and hence the reference time, is not altered.

27.14 EM Fields

%em_field component holds information on the electric and magnetic fields of an element (§3.16) Since ele%em_field is a pointer its association status must be tested before any of its sub–components are accessed.

```
type (ele_struct) ele
...
if (associated(ele%em_field)) then
  ...
```

The ele%em_field component is of type em_fields_struct which holds an array of modes

```
type em_fields_struct
  type (em_field_mode_struct), allocatable :: mode(:)
end type
```

Each mode has components

```
type em_field_mode_struct
  integer m                     ! Mode varies as cos(m*phi - phi_0)
  real(rp) freq                 ! Oscillation frequency (Hz)
  real(rp) :: f_damp = 0        ! 1/Q damping factor
  real(rp) :: phi0_autoscale = 0     ! Mode oscillates as: twopi * (f * t + phi0_autoscale)
  real(rp) :: phi0_azimuth = 0  ! Azimuthal orientation of mode.
  real(rp) :: field_scale = 1   ! Factor to scale the fields by
  type (em_field_mode_map_struct), pointer :: map => null()
  type (em_field_grid_struct), pointer :: grid => null()
end type
```

27.15 Wakes

The ele%wake component holds information on the wakes associated with an element. Since ele%wake is a pointer, its association status must be tested before any of its sub–components are accessed.

```
type (ele_struct) ele
...
if (associated(ele%wake)) then
  ...
```

Bmad observes the following rule: If %wake is associated, it is assumed that all the sub–components (%wake%sr_table, etc.) are associated. This simplifies programming in that you do not have to test directly the association status of the sub–components.

See §15.9 for the equations used in wake field calculations. Wake fields are stored in the %wake struct:

```
type wake_struct
  character(200) :: sr_file = ' '
  character(200) :: lr_file = ' '
  type (wake_sr_mode_struct) :: sr_long
  type (wake_sr_mode_struct) :: sr_trans
  type (wake_lr_struct), allocatable :: lr(:)
  real(rp) :: z_sr_max = 0
end type
```

The short–range wake parameterization uses pseudo–modes (Eq. (15.70)). This parameterization utilizes the %wake%sr_mode_long, and %wake%sr_mode_trans arrays for the longitudinal and transverse modes respectively. The structure used for the elements of these arrays are:

```
type wake_sr_mode_struct  ! Pseudo-mode short-range wake struct
  real(rp) amp        ! Amplitude
  real(rp) damp       ! Damping factor.
  real(rp) freq       ! Frequency in Hz
  real(rp) phi        ! Phase in radians/2pi
  real(rp) norm_sin   ! non-skew sin-like component of the wake
  real(rp) norm_cos   ! non-skew cos-like component of the wake
  real(rp) skew_sin   ! skew sin-like component of the wake
  real(rp) skew_cos   ! skew cos-like component of the wake
end type
```

The wake field kick is calculated from Eq. (15.70). `%amp`, `%damp`, `%freq`, and `%phi` are the input parameters from the lattice file. the last four components (`%norm_sin`, etc.) store the accumulated wake: Before the bunch passes through these are set to zero and as each particle passes through the cavity the contribution to the wake due to the particle is calculated and added the components.

`%wake%z_sr_mode_max` is the maximum z value beyond which the pseudo mode representation is not valid. This is set in the input lattice file.

The `%wake%lr` array stores the long–range wake modes. The structure definition is:

```
type wake_lr_struct   ! Long-Range Wake struct
  real(rp) freq       ! Actual Frequency in Hz
  real(rp) freq_in    ! Input frequency in Hz
  real(rp) R_over_Q   ! Strength in V/C/m^2
  real(rp) Q          ! Quality factor
  real(rp) angle      ! polarization angle (radians/2pi).
  integer m           ! Order (1 = dipole, 2 = quad, etc.)
  real(rp) norm_sin   ! non-skew sin-like component of the wake
  real(rp) norm_cos   ! non-skew cos-like component of the wake
  real(rp) skew_sin   ! skew sin-like component of the wake
  real(rp) skew_cos   ! skew cos-like component of the wake
  logical polarized   ! Polarized mode?
end type
```

This is similar to the `sr_mode_wake_struct`. `%freq_in` is the actual frequency in the input file. bmad_parser will set `%freq` to `%freq_in` except when the `lr_freq_spread` attribute is non-zero in which case `bmad_parser` will vary `%freq` as explained in §3.26. `%polarized` is a logical that indicates whether the mode has a polarization angle. If so, then `%angle` is the polarization angle.

27.16 Wiggler Types

The `%sub_key` component of the `ele_struct` is used to distinguish between `map type` and `periodic type` wigglers (§27.4):

```
if (ele%key == wiggler$ .and. ele%sub_key == map_type$) ...
if (ele%key == wiggler$ .and. ele%sub_key == periodic_type$) ...
```

For a `map` type wiggler, the wiggler field terms (§3.47.2) are stored in the `%wig_term(:)` array of the `element_struct`. This is an array of `wig_term_struct` structure. A `wig_term_struct` looks like:

```
type wig_term_struct
  real(rp) coef
  real(rp) kx, ky, kz
  real(rp) phi_z
```

```
  integer type      ! hyper_y$, hyper_xy$, or hyper_x$
end type
```

A `periodic` wiggler will have a single `%wig_term(:)` term that can be used for tracking purposes, etc. The setting for this `wig_term` element is

```
ele%wig_term(1)%ky     = pi / ele%value(l_pole$)
ele%wig_term(1)%coef   = ele%value(b_max$)
ele%wig_term(1)%kx     = 0
ele%wig_term(1)%kz     = ele%wig_term(1)%ky
ele%wig_term(1)%phi_z  = (ele%value(l_pole$) - ele%value(l$)) / 2
ele%wig_term(1)%type   = hyper_y$
```

27.17 Multipoles

The multipole components of an element (See §15.1) are stored in the pointers `%a_pole(:)` and `%b_pole(:)`. If `%a_pole` and `%b_pole` are allocated they always have a range `%a_pole(0:n_pole_maxx)` and `%b_pole(0:n_pole_maxx)`. Currently `n_pole_maxx` = 20. For a `Multipole` element, the `%a_pole(n)` array stores the integrated multipole strength `KnL`, and the `%b_pole(n)` array stores the tilt `Tn`.

A list of *Bmad* routines for manipulating multipoles can be found in §39.26.

27.18 Tracking Methods

A number of `ele_struct` components control tracking and transfer map calculations. These are:

```
%mat6_calc_method
%tracking_method
%taylor_order
%symplectify
%multipoles_on
%taylor_map_includes_offsets
%is_on
%csr_method
%space_charge_method
%offset_moves_apaerture
```

See Chapter §32 for more details.

27.19 Custom and General Use Attributes

There are four components of an `ele_struct` that are guaranteed to never be used by any *Bmad* routine and so are available for use by someone writing a program. These components are:

```
%r(:,:,:)                   ! real(rp), pointer.
%custom(:)                  ! real(rp), pointer.
%value(scratch1$:scratch5$) ! Five elements of the %value(:) array.
%ix_pointer                 ! integer.
%logic                      ! logical.
```

Values for `ele%r` and `ele%custom` can be set in the lattice file (§2.9). If values are set for `ele%r` or `ele%custom`, these arrays will be expanded in size if needed.

Accessing the `ele%custom` array should be done using the standard accessor routines (§29.4). For example, if the lattice file being used defines a custom attribute called `rise_time` (§2.9):

```
parameter[custom_attribute1] = "rise_time"
```

then a program can access the `rise_time` attribute via:

```
type (ele_struct), poiner :: ele
type (all_pointer_struct) a_ptr
...
ele => ...            ! Point ele to some element in the lattice
call pointer_to_attribute (ele, 'RISE_TIME', .true., a_ptr, err_flag)
print *, 'RISE_TIME Attribute has value:', a_ptr%r
```

Note: Even if there are custom attributes associated with a given type of element (say, all quadrupoles), a given element of that type may not have its `%custom(:)` array allocated. [In this case, none of the custom values have been set so are zero by definition.] In the above example, the `%custom` array will be allocated if needed in the call to `pointer_to_attribute`.

If not defined through a lattice file, custom attributes can also be defined directly from within a program using the set_custom_attribute_name routine. For example:

```
logical err_flag
...
call set_custom_attribute_name ('QUADRUPOLE::ERROR_CURRENT', err_flag)
```

Note: When there is a superposition (§7.1), the `super_slave` elements that are formed do *not* have any custom attributes assigned to them even when their `super_lord` elements have custom attributes. This is done since the *Bmad* bookkeeping routines are not able to handle the situation where a `super_slave` element has multiple `super_lord` elements and thus the custom attributes from the different `super_lord` elements have to be combined. Proper handling of this situation is left to any custom code that a program implements to handle custom attributes.

27.20 Bmad Reserved Variables

indexele_struct!%ixx indexele_struct!%bmad_logic indexele_struct!%const A number of `ele_struct` components are reserved for use by *Bmad* routines only. These are:

```
%ixx
%bmad_logic
```

The `%ixx` and `%bmad_logic` components are used for internal *Bmad* bookkeeping purposes.

Chapter 28

The lat_struct

The lat_struct is the structure that holds of all the information about a lattice (§1.3). The components of a lat_struct are listed in Fig. 28.1.

```
type lat_struct
  character(40) name                        ! Name in USE statement
  character(40) lattice                     ! Lattice name
  character(80) input_file_name             ! Lattice input file name
  character(80) title                       ! From TITLE statement
  type (expression_atom_struct), allocatable :: constant(:)  ! Constants defined in the lattice
  type (mode_info_struct) a, b, z             ! Tunes, etc.
  type (lat_param_struct) param               ! Parameters
  type (bookkeeping_state_struct) lord_state  ! lord bookkeeping status.
  type (ele_struct) ele_init                  ! For use by any program
  type (ele_struct), pointer ::  ele(:) => null()  ! Array of elements [=> branch(0)].
  type (branch_struct), allocatable :: branch(:)   ! Branch(0:) array
  type (control_struct), allocatable :: control(:) ! Control list
  type (photon_reflect_surface_struct), pointer :: surface(:) => null()
  type (coord_struct) particle_start      ! Starting coords
  type (pre_tracker_struct) pre_tracker   ! For OPAL/IMPACT-T
  real(rp), allocatable :: custom(:)      ! Custom attributes.
  integer version                         ! Version number
  integer n_ele_track                     ! Number of lat elements to track through.
  integer n_ele_max                       ! Index of last valid element in %ele(:) array
  integer n_control_max                   ! Last index used in control_array
  integer n_ic_max                        ! Last index used in ic_array
  integer input_taylor_order              ! As set in the input file
  integer, allocatable :: ic(:)           ! Index to %control(:)
  integer :: photon_type = incoherent$    ! Or coherent$. For X-ray simulations.
  logical absolute_time_tracking          ! Use absolute time in lcavity and rfcavity tracking?
  logical ptc_uses_hard_edge_drifts       ! Associated ptc layout have hard edge model drifts?
end type
```

Figure 28.1: Definition of the lat_struct.

The %ele_init component within the lat_struct is not used by *Bmad* and is available for general program use.

28.1 Initializing

Normally initialization of a lat_struct lattice is done by bmad_parser when a lattice file is parsed and does not have to be done by the programmer. When a programmer needs to initialize a lattice, however, init_lat is used to initalize the lattice with a single branch. After this initial setup, the routines allocate_branch_array and allocate_lat_ele_array can be used to set up additional branches. Example:

```
type (lat_struct) lat
...
call init_lat (lat, 1000)           ! Branch(0) has 1001 elements.
call allocate_branch_array (lat, 2) ! Allocate Branch(1) and Branch(2).
call allocate_lat_ele_array (lat, 20, 1)  ! Branch(1) has 21 elements
call allocate_lat_ele_array (lat, 30, 2)  ! Branch(2) has 31 elements.
```

28.2 Pointers

Since the lat_struct has pointers within it, there is an extra burden on the programmer to make sure that allocation and deallocation is done properly. To this end, the equal sign has been overloaded by the routine lat_equal_lat so that when one writes

```
type (lat_struct) lattice1, lattice2
! ... some calculations ...
lattice1 = lattice2
```

the pointers in the lat_struct structures will be handled properly. The result will be that lattice1 will hold the same information as lattice2 with all the lattice elements in the same place but the pointers in lattice1 will point to different locations in physical memory so that changes to one lattice will not affect the other.

deallocate_lat_pointers Initial allocation of the pointers in a lat_struct variable is generally handled by the bmad_parser and lat_equal_lat routines. Once allocated, local lat_struct variables must have the save attribute or the pointers within must be appropriately deallocated before leaving the routine.

```
type (lat_struct), save :: lattice     ! Either do this at the start or ...
...
call deallocate_lat_pointers (lattice) ! ... Do this at the end.
```

Using the save attribute will generally be faster but will use more memory. Typically using the save attribute will be the best choice.

28.3 Branches in the lat_struct

The lattice is divided up into the "root branch" (§6.6) and, if there are fork or photon_fork elements, a number "forked" branches.

The branches of a lattice is contained in the lat%branch(0:) array. The %branch(0:) array is always indexed from 0 with the 0 branch being a root branch. The definition of the branch_struct structure is

	Element index	
section	*min*	*max*
tracking	0	%n_ele_track
control	%n_ele_track+1	%n_ele_max

Table 28.1: Bounds of the tracking and control parts of the root branch (lat%branch(0)%ele(:)) array.

```
type branch_struct
  character(40) name
  integer ix_branch                    ! Index in lat%branch(:) array.
  integer ix_from_branch               ! -1 => No forking element to this branch.
  integer ix_from_ele                  ! Index of forking element
  integer, pointer :: n_ele_track      ! Number of tracking elements
  integer, pointer :: n_ele_max
  type (mode_info_struct), pointer :: a, b, z
  type (ele_struct), pointer :: ele(:)
  type (lat_param_struct), pointer :: param
  type (wall3d_struct), pointer :: wall3d(:)
  type (ptc_branch1_info_struct) ptc
  type (normal_form_struct) normal_form_with_rf, normal_form_no_rf
end type
```

The value of the %branch(i)%ix_branch conponent is the branch index and will thus have the value i. This can be useful when passing a branch to a subroutine. The %branch(i)%ix_from_branch component gives the branch index of the branch that the i^{th} branch branched off from. %branch(i)%ix_from_ele gives the index in the %branch(j)%ele(:) array of the fork or photon_fork element that marks the beginning of the i^{th} branch. Example:

```
type (lat_struct), target :: lat
type (ele_struct), pointer :: ele
...
ib = lat%branch(3)%ix_from_branch
ie = lat%branch(3)%ix_from_ele
! ele is the fork or photon_fork element for lat%branch(3)
ele => lat%branch(ib)%ele(ie)
! This is the same as the above.
ele => pointer_to_ele(lat%branch(3)%ix_from_branch, lat%branch(3)%ix_from_ele)
```

The %branch%ele(:) array holds the array of elements in the branch. Historically, the lat_struct was developed at the start of the *Bmad* project and branches were implemented well after that. To maintain compatibility with older code, the following components point to the same memory blocks

```
lat%ele(:)              <---> lat%branch(0)%ele(:)
lat%n_ele_track         <---> lat%branch(0)%n_ele_track
lat%n_ele_max           <---> lat%branch(0)%n_ele_max
lat%param               <---> lat%branch(0)%param
```

All %branch%ele(:) arrays are allocated with zero as the lower bound. The %ele(0) element of all branches is an beginning_ele element with its %name component set to "BEGINNING". %ele(0)%mat6 is always the unit matrix. For the root branch, the %branch(0)%ele(0:) array is divided up into two parts: The "tracking" part and a "control" part (also called the "lord" part). The tracking part of this array holds the elements that are tracked through. The control part holds elements that control attributes of other elements (§28.5). The bounds of these two parts is given in Table 28.1. Only the root

branch has a lord section so %branch%n_ele_track and %branch%n_ele_max are the same for all other branches. Since the root branch can also be accessed via the lat%ele(:) array, code that deals with the lord section of the lattice may use lat%ele(:) in place of lat%branch(0)%ele(:).

for a given fork or photon_fork element, the index of the branch that is being forked to and the index of the element that is being forked to is stored in:

```
ix_branch  = nint(branch_ele%value(ix_branch_to$))  ! branch index
ix_element = nint(branch_ele%value(ix_element_to$)) ! element index
direction  = nint(branch_ele%value(direction$))
```

The direction will be +1 for forward forking and -1 for backward forking.

28.4 Param_struct Component

The %param component within each lat%branch(:) is a lat_param_struct structure whose definition is shown in Fig. 28.2 This structure would be more aptly named branch_param_struct but is named otherwise for historical reasons.

%param%total_length is the length of the branch that a beam tracks through defined by

```
%param%total_length = %ele(n_ele_track)%s - %ele(0)%s
```

Normally %ele(0)%s = 0 so %param%total_length = %ele(n_ele_track)%s but this is not always the case.

%param%n_part is the number of particles in a bunch and is used by beambeam element to determine the strength of the beambeam interaction. %param%n_part is also used by lcavity elements for wake field calculations.

For closed branches, %param%t1_with_RF and %param%t1_no_RF are the 1–turn transfer matrices from the start of the branch to the end. %param%t1_with_RF is the full transfer matrix with RF on. %param%t1_no_RF is the transverse transfer matrix with RF off. %param%t1_no_RF is used to compute the Twiss parameters. When computing the Twiss parameters %param%stable is set according to whether the matrix is stable or not. If the matrix is not stable the Twiss parameters cannot be computed. If unstable, %param%unstable_factor will be set to the growth rate per turn of the unstable mode.

```
type lat_param_struct
  real(rp) n_part             ! Particles/bunch (for beambeam elements).
  real(rp) total_length       ! total_length of lattice
  real(rp) unstable_factor    ! closed branch: growth rate/turn.
                              !   all branches: |orbit/limit|
  real(rp) t1_with_RF(6,6)    ! Full 1-turn 6x6 matrix
  real(rp) t1_no_RF(6,6)      ! Transverse 1-turn 4x4 matrix (RF off).
  integer particle            ! +1 = positrons, -1 = electrons, etc.
  integer geometry            ! open$, etc...
  integer ixx                 ! Integer for general use
  logical stable              ! For closed branch. Is lat stable?
  type (bookkeeper_status_struct) bookkeeping_state
                                      ! Overall status for the branch.
end type
```

Figure 28.2: Definition of the param_struct.

Besides being set when the 1–turn transfer matrix is calculated, `%param%unstable_factor` will be set if a particle is lost in tracking to:

```
orbit_amplitude / limit - 1
```

The particle type for a branch is stored in the integer variable `%param%particle`. The value of this variable will encode for a fundamental particle, atom, or molecule. See the file `particle_species_mod.f90` for more details. If the particle corresponds to a fundamental particle, `%param%particle` will correspond to one of the following constants:

```
electron$,      positron$,
muon$,          antimuon$,
proton$,        antiproton$,
photon$,        pion_0$,
pion_minus$,    pion_plus$
deuteron$       deuteron_0$
```

To print the name of the particle use the function `particle_name`. A particles mass and charge can be obtained from the functions `mass_of` and `charge_of` respectively. `charge_of` returns the particle's charge in units of e. Example:

```
type (lat_struct) lat
...
print *, 'Beam Particles are: ', particle_name(lat%param%particle)
if (lat%param%particle == proton$) print *, 'I do not like protons!'
print *, 'Particle mass (eV):    ', mass_of(lat%param%particle)
print *, 'Particle charge:       ', charge_of(lat%param%particle)
```

28.5 Elements Controlling Other Elements

In the `lat_struct` structure, certain elements in the `%ele(:)` array (equivalent to the `%branch(0)%ele(:)` array), called `lord` elements, can control the attributes (component values) of other `%branch(:)%ele(:)` elements. Elements so controlled are called `slave` elements. The situation is complicated by the fact that a given element may simultaneously be a `lord` and a `slave`. For example, an `overlay` element (§3.36) is a lord since it controls attributes of other elements but an `overlay` can itself be controlled by other `overlay` and `group` elements. In all cases, circular lord/slave chains are not permitted.

The lord and slave elements can be divided up into classes. What type of lord an element is, is set by the value of the element's `ele%lord_status` component. Similarly, what type of slave an element is is set by the value of the element's `ele%slave_status` component. Nomenclature note: An element may be referred to by it's `%lord_status` or `%slave_status` value. For example, an element with `ele%lord_status` set to `super_lord$` can be referred to as a "`super_lord`" element.

The value of the `ele%lord_status` component can be one of:

super_lord$
: A `super_lord` element is created when elements are superimposed on top of other elements (§7.1).

girder_lord$
: A `girder_lord` element is a `girder` element (§3.20). That is, the element will have `ele%key` = `girder$`.

multipass_lord$
: `multipass_lord` elements are created when multipass lines are present (§7.2).

overlay_lord$
: An `overlay_lord` is an `overlay` element (§3.36). That is, such an element will have `ele%key` = `overlay$`.

group_lord$
: A `group_lord` is a `group` element (§3.21). That is, such an element will have `ele%key` = `group$`.

not_a_lord$
: This element does not control anything.

Any element whose `%lord_status` is something other than `not_a_lord$` is called a `lord` element. In the `tracking part` of the branch (§28.3), `%lord_status` will always be `not_a_lord$`. In the `lord section` of the branch, under normal circumstances, there will never be any `not_a_lord` elements. However, it is permissible, and sometimes convenient, for programs to set the `%lord_status` of a lord element to `not_a_lord$`.

The possible values for the `ele%slave_status` component are:

multipass_slave$
: A `multipass_slave` element is the slave of a `multipass_lord` (§7.2).

slice_slave$
: A `slice_slave` element represents a longitudinal slice of another element. Slice elements are not part of the lattice but rather are created on-the-fly when, for example, a program needs to track part way through an element.

super_slave$
: A `super_slave` element is an element in the tracking part of the branch that has one or more `super_lord` lords (§7.1).

minor_slave$
: A `minor_slave` element is an element that is not a `slice_slave` and does not have a `major` lord. Major lords are `super_lord`s and `multipass_lord`s. A `minor_slave` element will some have attributes that are controlled by `overlay_lords`, `group_lords`, or `girder_lords`.

free$
: A `free` element is one that has no lords. [But there still might be field overlap from other elements.]

`super_slave` elements always appear in the tracking part of the branch. The other types can be in either the tracking or control parts of the branch.

Only some combinations of `%lord_status` values and `%slave_status` values are permissible for a given element. Table 28.2a lists the valid combinations. Thus, for example, it is *not* possible for an element to be simultaneously a `super_lord` and a `super_slave`.

For lord/slave pairs, Table 28.2b lists the valid combinations of `%lord_status` values in the lord element and `%slave_status` values in the slave element. Thus, for example, a `super_slave` may only be controlled by a `super_lord`. In the example in Section §7.2, element `A` would be a `multipass_lord` and `A\1` and `A\2` would be `multipass_slaves`. When superposition is combined with multipass, the elements in the tracking part of the branch will be `super_slaves`. These elements will be controlled by `super_lords` which will also be `multipass_slaves` and these `super_lord`/`multipass_slave` elements will be controlled by `multipass_lords`. This is illustrated in Fig. 28.3.

The number of slave elements that a lord controls is given by the value of the lord's `%n_slave` component. Additionally, the number of lord elements that the slave has is given by the value of the slave's. `%n_lord`

ele%slave_status	ele%lord_status					
	not_a_lord$	group_lord$	girder_lord$	overlay_lord$	multipass_lord$	super_lord$
free$	X	X	X	X	X	X
minor_slave$	X	X	X	X	X	X
multipass_slave$	X					X
slice_slave$	X					
super_slave$	X					

(a) Possible ele%lord_status and ele%slave_status combinations within an individual element.

slave%slave_status	lord%lord_status					
	not_a_lord$	group_lord$	girder_lord$	overlay_lord$	multipass_lord$	super_lord$
free$						
minor_slave$		X	X	X		
multipass_slave$		X		X	1	
super_slave$		X		X		X

(b) Possible %lord_status and %slave_status combinations for any lord/slave pair.

Table 28.2: Possible %lord_status/%slave_status combinations. "X" marks a possible combination. "1" indicates that the slave will have exactly one lord of the type given in the column.

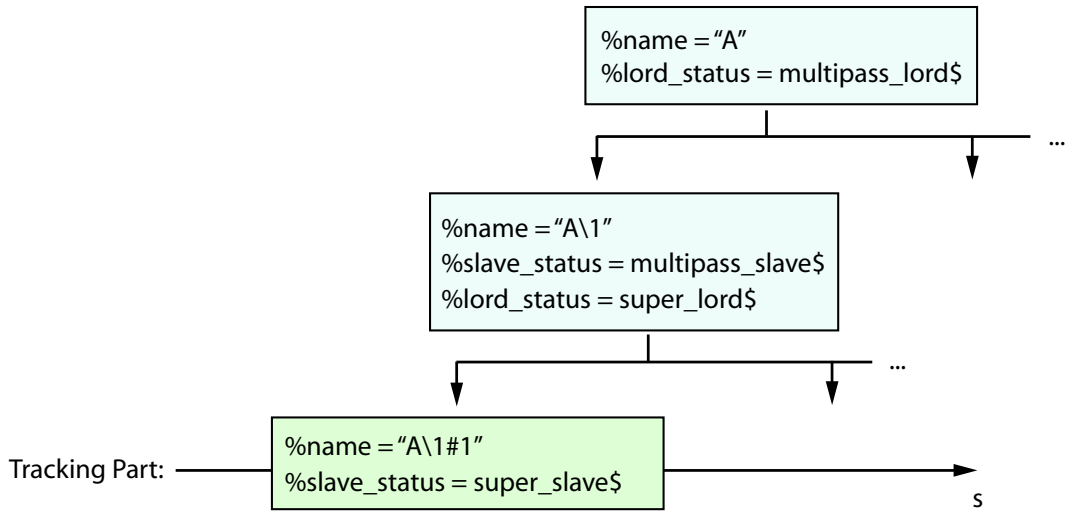


Figure 28.3: Example of multipass combined with superposition. A `multipass_lord` element named `A` controls a set of `multipass_slaves` (only one shown). The `multipass_slave` elements are also `super_lord` elements and they will control `super_slave` elements in the tracking part of the branch.

component. To find the slaves and lords of a given element, use the routines pointer_to_slave and pointer_to_lord. Example:

```
type (lat_struct), target :: lat
type (ele_struct), pointer :: this_ele, lord_ele, slave_ele
...
this_ele => lat%ele(321)    ! this_ele points to a given element in the lattice

do i = 1, this_ele%n_lord   ! Loop over all lords of this_ele
  ! lord_ele points to the i^th lord element of this_ele
  lord_ele => pointer_to_lord (this_ele, i)
  ...
enddo
```

```
do i = 1, this_ele%n_slave  ! Loop over all slaves of this_ele
  ! slave_ele points to the i^th slave element of this_ele
  slave_ele => pointer_to_slave (this_ele, i)
  ...
enddo
```

The lord/slave bookkeeping is bidirectional. That is, for any given element, call it `this_ele`, consider the i^{th} lord:

```
lord_ele_i => pointer_to_lord (this_ele, i)
```

then there will always be some index j such that the element pointed to by

```
pointer_to_slave(lord_ele_i, j)
```

is the original element `this_ele`. The same is true for the slaves of any given element. That is, for the i^{th} slave

```
slave_ele_i => pointer_to_slave (this_ele, i)
```

there will always be some index j such that the element pointed to by

```
pointer_to_lord(slave_ele_i, j)
```

The following ordering of slaves and lords is observed:

Slaves of a super_lord:
The associated `super_slave` elements of a given `super_lord` element are ordered from the entrance end of the `super_lord` to the exit end. That is, in the code snippet above, `pointer_to_slave (this_ele, 1)` will point to the slave at the start of the `super_lord` and `pointer_to_slave (this_ele, this_ele%n_lord)` will point to the slave at the exit end of the `super_lord`.

Slaves of a multipass_lord:
The associated `multipass_slave` elements of a `multipass_lord` element are ordered by pass number. That is, in the code snippet above, `pointer_to_slave (this_ele, i)` will point to the slave of the i^{th} pass.

Lord of a multipass_slave:
A `multipass_slave` will have exactly one associated `multipass_lord` and this lord will be the first one. That is, `pointer_to_lord (this_ele, 1)`.

The element control information is stored in the `lat%control(:)` array. Each element of this array is a `control_struct` structure

```
type control_struct
  type (expression_atom_struct), allocatable :: stack(:) ! Evaluation stack
  type (lat_ele_loc_struct) slave ! Slave location
  type (lat_ele_loc_struct) lord  ! Lord location
  integer ix_attrib               ! index of controlled attribute
end type
```

Each element in the `lat%control(:)` array holds the information on one lord/slave pair. The `%lord` component gives the location of the lord element which is always in the root branch — branch 0. The `%slave` component give the element location of the slave element. The `%stack` and `%ix_attrib` components are used to store the arithmetic expression and attribute index for `overlay` and `group` control. The appropriate control_struct for a given lord/slave pair can be obtained from the optional fourth argument of the pointer_to_lord and pointer_to_slave functions. Example: The following prints a list of the slaves, along with the attributes controlled and coefficients, on all group elements in a lattice.

```
type (lat_struct), target :: lat
type (ele_struct), pointer :: lord, slave
type (control_struct), pointer :: con
...
do i = lat%n_ele_track+1, lat%n_ele_max  ! loop over all lords
  lord => lat%ele(i)
  if (lord%lord_status = group_lord$) then
    print *, 'Slaves for group lord: ', lord%name
    do j = 1, lord%n_slave
      slave => pointer_to_slave (lord, j, con)
      attrib_name = attribute_name (slave, con%ix_attrib)
      print *, i, slave%name, attrib_name, con%coef
    enddo
  endif
enddo
```

The elements in the `lat%control(:)` array associated with the slaves of a given lord are in the same order as the slaves and the index of the associated `lat%control(:)` element of the first slave is given by the `%ix1_slave` component of the lord Example:

```
type (lat_struct), target :: lat
type (ele_struct), pointer :: lord, slave
type (control_struct), pointer :: con1, con2
...
lord => lat%ele(i)                    ! Point to some lord
do j = 1, lord%n_slave
  slave => pointer_to_slave (lord, j, con1)
  con2 => lat%control(lord%ix1_slave+j-1) ! con1 and con2 are the same.
enddo
```

Except for a `slice_slave`, the `%ic1_lord`, `%n_lord`, and `%n_lord_field` components of a given slave element, along with the `lat%ic(:)` array, can be used to find the lords of the slave. Simplified, the code for the pointer_to_lord function is:

```
function pointer_to_lord (slave, ix_lord, con, ...) result (lord_ptr)
  implicit none
  type (lat_struct), target :: lat
  type (ele_struct) slave
  type (ele_struct), pointer :: lord_ptr
  type (control_struct), pointer, optional :: control
  integer ix_lord, icon
  !
  icon = lat%ic(slave%ic1_lord + ix_lord - 1)
  lord_ptr => lat%ele(lat%control(icon)%lord%ix_ele)
  if (present(con)) con => lat%control(icon)
end function
```

This method for finding the lords of an element is considered "private". That is, no code outside of the official *Bmad* library should rely on this.

`slice_slave` element bookkeeping has is different depending upon whether the element being sliced is a `super_slave` or not. If the element being sliced is a `super_slave`, a `slice_slave` element that is created is, for bookkeeping purposes, considered to be a slave of the `super_slave`'s lords. In this case, the bookkeeping is exactly the same as that of any `super_slave`, and `pointer_to_lord` will return a pointer to one of the `super_slave`'s lords.

On the other hand, if a non `super_slave` element is being sliced, the `%lord` pointer component of the `slice_slave` will be set to point to the element being sliced.

28.6 Lattice Bookkeeping

The term "lattice bookkeeping" refers to the updating of the appropriate parameter values when a given parameter in the lattice is changed. For example, if the accelerating gradient of an `lcavity` element is modified, the reference energy parameter of all elements downstream of the `lcavity` will need to be changed and this can also alter the transfer maps of the `lcavity` and downstream elements. *Bmad* divides the lattice bookkeeping into "core" part and everything else. The core part itself is divided into five parts:

Attribute bookkeeping
This refers to intra-element dependent attribute bookkeeping (§4.1).

Control bookkeeping
This refers to Lord/Slave bookkeeping for `overlay` (§3.36) and `group` (§3.21)elements, and for `superposition` (§7.1) and `multipass` (§7.2) lords.

Floor Position bookkeeping This refers to bookkeeping to keep track of an elements global "floor" position stored in the `ele%floor` structure.

Length bookkeeping This refers to bookkeeping to keep track of the longitudinal s-position of an element stored in the `ele%s` component.

Reference Energy bookkeeping This refers to the reference energy assigned to each element (§33.6). `ele%value(E_tot$)` and `ele%value(p0c$)`

Historically, as the concept of lattice bookkeeping was being developed, to be back compatible with existing programs, calls to bookkeeping routines were added to calculational routines such as the tracking routine track1 and the routine for calculating the linear transfer map make_mat6. This "automatic" bookkeeping system is inefficient since there is no good way to keep track of what element attributes have been modified which leads to redundant bookkeeping calculations. Eventually, as *Bmad* developed and became more complicated, it was found that the unnecessary bookkeeping load was generally causing a significant slowdown in program execution time — even in programs where no element attributes were changed. To avoid this, an "intelligent" bookkeeping system was developed. In order to be back compatible with existing programs, the automatic bookkeeping system is the default. However, given the fact that the automatic bookkeeping system has known deficiencies, and given the overhead with maintaining two bookkeeping systems, the current plan is to start phasing out the automatic bookkeeping system sometime in the not-so-far future. Thus old programs should be converted to the new system and all new programs should use the new bookkeeping system.

To use intelligent bookkeeping, a program must set the global `bmad_com%auto_bookkeepper` to false. This is done once at the start of the program. When a set of attributes needs to be modified, the set_flags_for_changed_attribute routine must be called for each element attribute that is set. After all the attributes have been set, lattice_bookkeeper is called to do the core bookkeeping. Example

```
type (lat_struct) lat
...
bmad_com%auto_bookkeeper = .false.    ! Only needs to be done once.
...
lat%ele(i)%value(gradient$) = 1.05e6  ! Change, say, the gradient of an RFCavity
```

```
call set_flags_for_changed_attribute (lat%ele(i), lat%ele(i)%value(gradient$))

... Set attributes of other elements ...

call lattice_bookkeeper (lat)         ! Do once after all attribute sets done.
```

The argument list for `set_flags_for_changed_attribute` is

```
set_flags_for_changed_attribute (ele, attribute)
```

The `attribute` argument may be either real, integer, or logical.

The `set_flags_for_changed_attribute` routine sets flags in the `ele%status` structure. This structure is of type `bookkeeper_status_struct` and looks like

```
type bookkeeper_status_struct
  integer attributes      ! Intra element dependent attribute status
  integer control         ! Lord/slave bookkeeping status
  integer floor_position  ! Global (floor) geometry status
  integer length          ! Longitudinal position status
  integer ref_energy      ! Reference energy status
  integer mat6            ! Linear transfer map status
  integer rad_int         ! Radiation integrals cache status
end type
```

All components of this structure give the status of some lattice bookkeeping aspect. The first five components of this structure correspond to the five core bookkeeping parts discussed above. The other two components are discussed below.

Possible values for the status components are

```
super_ok$
ok$
stale$
```

The `set_flags_for_changed_attribute` routine sets the appropriate status components of an element to `stale$` which marks that element for the appropriate bookkeeping. When the bookkeeping is done by `lattice_bookkeeper`, the `stale$` status components are set to `ok$`. The `super_ok$` value is reserved for use by any program that needs to do its own custom bookkeeping. How this works is as follows: The *Bmad* bookkeeping routines will never convert a status component with value `super_ok$` to `ok$` without first doing some needed bookkeeping. Thus if a program sets a status component to `super_ok$` and then later on finds that the status component is set to `ok$`, the program knows that bookkeeping has been done. An example will make this clear. Suppose a program needs to keep track of a collection of high order transfer maps between various points in a lattice. Suppose that the constant calculation of these maps would slow the program done so it is desired to recalculate a given map only when necessary. To implement this, the program could set the `ele%status%mat6` attribute of all the element to `super_ok$` when the maps are calculated. If the program subsequently finds a `ele%status%mat6` attribute of an element set to `ok$` it knows that it should recalculate any transfer maps that span that element.

It is guaranteed that when `lattice_bookkeeper` is run, all five core status components will not be `stale$`. The routines used by `lattice_bookkeeper` are:

```
attribute_bookkeeper      ! Intra-element attributes
control_bookkeeper        ! Lord/slave control
s_calc                    ! Longitudinal element s-position
lat_geometry              ! Global (floor) positions.
lat_compute_ref_energy_and_time  ! Reference energy
```

In general, these routines should not be called directly since the correct way to do things is not always straight forward. See the code for `lattice_bookkeeper` for more details.

After the core bookkeeping is done, a program can call lat_make_mat6 to remake the transfer matrices. `lat_make_mat6` will remake the transfer matrices if either the `ele%status%mat6` flag is `stale$` or the reference orbit around which the existing transfer matrix was computed has shifted. `lat_make_mat6` will set the `ele%status%mat6` flag to `ok$` for all elements whose transfer matrices are recomputed.

28.7 particle_start Component

The `lat%particle_start` component is a `coord_struct` structure for holding the information obtained from `particle_start` statements (§8.2) in a *Bmad* lattice file.

This component is not used in any standard *Bmad* calculation. It is up to an individual program to use as desired.

28.8 Custom Parameters

Custom parameters defined for the lattice as a whole (§2.9 are stored in `lat%custom`. The following shows how to print a table of the custom parameters

```
type (lat_struct) lat
character(80) aname
...
if (allocated(lat%custom)) then
  do i = 1, size(lat%custom)
    aname = attribute_name(def_parameter$, i+custom_attribute0$)
    if (aname(1:1) == '!') cycle    ! Ignore non-existant parameters
    print '(a, es12.4)', '  parameter[' // trim(aname) // '] = ', lat%custom(i)
  enddo
endif
```

Chapter 29

Lattice Element Manipulation

29.1 Creating Element Slices

It is sometimes convenient to split an element longitudinally into "slices" that represent a part of the element. This is complicated by the fact that elements are not necessarily uniform. For example, map type wigglers are nonuniform and bend elements have end effects. Furthermore, attributes like `hkick` need to be scaled with the element length.

To create an element slice, the routine create_element_slice can be used. Example:

```
type (ele_struct) ele, sliced_ele
...
sliced_ele = ele
sliced_ele%value(l$) = l_slice ! Set the sliced element's length
call create_element_slice (sliced_ele, ele, l_start, param, ...)
```

See the documentation on `create_element_slice` for more details (§25.3).

29.2 Adding and Deleting Elements From a Lattice

Modifying the number of elements in a lattice involves a bit of bookkeeping. To help with this there are a number of routines.

The routine remove_eles_from_lat is used to delete elements from a lattice.

For adding elements there are three basic routines: To add a lord element, the new_control routine is used. To add a new element to the tracking part of the lattice, use the insert_element routine. Finally, to split an element into two pieces, the routine split_lat is used. These basic routines are then used in such routines as create_overlay that creates overlay elements, create_group which creates group elements, add_superimpose which superimposes elements, etc. Example:

```
type (lat_struct), target :: lat
type (ele_struct), pointer :: g_lord, slave

type (control_struct) con(1)
integer ix, n
logical err_flag
```

```
...
call new_control (lat, ix)
g_lord => lat%ele(ix)
allocate (ele%control_var(1))
ele%control_var(1)%name = 'A'
call reallocate_expression_stack(con(1)%stack, 10))
call expression_string_to_stack ('3.2*A^2', con(1)%stack, n, err_flag)
con(1)%ix_attrib = k1$
call lat_ele_locator ('Q1W', lat, eles)
con(1)%slave = ele_to_lat_loc(eles(1)%ele)
call create_group (g_lord, con, err_flag)
```

This example constructs a group element with one variable with name `A` controlling the `K1` attribute of element `Q1W` using the expression "$3.2 \cdot A^2$" where `A` is the name of the control variable.

For constructing `group` elements (but not `overlay` elements), the controlled attribute (set by `con(1)%ix_attrib` in the above example) can be set to, besides the set of element attributes, any one in the following list:

```
accordion_edge$  ! Element grows or shrinks symmetrically
start_edge$      ! Varies element's upstream edge s-position
end_edge$        ! Varies element's downstream edge s-position
s_position$      ! Varies element's overall s-position. Constant length.
```

See Section §3.21 for the meaning of these attributes

29.3 Finding Elements

The routine lat_ele_locator can be used to search for an element in a lattice by name or key type or a combination of both. Example:

```
type (lat_struct) lat
type (ele_pointer_struct), allocatable :: eles(:)
integer n_loc; logical err
...
call lat_ele_locator ("quad::skew*", lat, eles, n_loc, err)
print *, 'Quadrupole elements whose name begins with the string "SKEW":'
print *, 'Name                 Branch_index        Element_index'
do i = 1, n_loc  ! Loop over all elements found to match the search string.
  print *, eles(i)%ele%name, eles(i)%ele%ix_branch, eles(i)%ele%ix_ele
enddo
```

This example finds all elements where `ele%key` is `quadrupole$` and `ele%name` starts with "`skew`". See the documentation on `lat_ele_locator` for more details on the syntax of the search string.

The `ele_pointer_struct` array returned by `lat_ele_locator` is an array of pointers to `ele_struct` elements

```
type ele_pointer_struct
  type (ele_struct), pointer :: ele
end type
```

The `n_loc` argument is the number of elements found and the `err` argument is set True on a decode error of the search string.

Once an element (or elements) is identified in the lattice, it's attributes can be altered. However, care must be taken that an element's attribute can be modified (§4.1). The function `attribute_free` will check if an attribute is free to vary.

```
type (lat_struct) lat
integer ix_ele
...
call lat_ele_locator ('Q10W', lat, eles, n_loc, err)   ! look for an element 'Q10W'
free = attribute_free (eles(i)%ele, 'K1', lat, .false.)
if (.not. free) print *, 'Cannot vary k1 attribute of element Q10W'
```

29.4 Accessing Named Element Attributes

A "named" parameter of the `ele_struct` structure is a parameter that has an associated name that can be used in a lattice file. For example, the quadrupole strength is named `K1` (§3.39). This parameter is stored in the `ele%value(:)` array. Specifically at `ele%value(k1$)`.

Historically, named parameters where always accessed directly but this has proved to be somewhat problematical for a number of reasons. For one, something like `ele%value(k1$)` will always have a value even if the associated lattice element does not have an associated `K1` parameter (For example, a sextupole does not have a `K1` parameter). Another issue involves allocation since components like `ele%a_pole(:)` are pointers that are not necessarily allocated.

To get around some of these issues, accessor functions have been developed for all non-character based named attributes. These accessor functions are:

```
pointer_to_attribute    ! pointer_to_attribute
pointers_to_attribute   ! pointers_to_attribute
set_ele_attribute       ! set_ele_attribute
value_of_attribute      ! value_of_attribute
```

The workhorse is `pointer_to_attribute` that returns a pointer to the appropriate attribute. The returned pointer argument is actually an instance of an `all_pointer_struct` which looks like:

```
type all_pointer_struct
  real(rp), pointer :: r => null()
  integer, pointer :: i => null()
  logical, pointer :: l => null()
end type
```

When the `all_pointer_struct` argument is returned, one (or zero if the attribute name is not recognized) of the pointer components will be associated. For example:

```
type (ele_struct) ele
type (all_pointer_struct) attrib_ptr
...
call pointer_to_attribute (ele, 'A3_ELEC', .true., attrib_ptr, err)
attrib_ptr%r = 0.34
call attribute_set_bookkeeping (ele, 'A3_ELEC', err_flag, attrib_ptr)
call lattice_bookkeeper (lat)  ! Bookkeeping needed due to parameter change
```

Also see the example program in §26.2.

The `set_ele_attribute` routine is useful when there is user input since this routine can evaluate expressions. For example:

```
type (lat_struct) lat
type (ele_pointer_struct), allocatable :: eles(:)
integer n_loc, n
logical err_flag, make_xfer_mat
...
```

```
call lat_ele_locator ('Q01W', lat, eles, n_loc, err_flag)
do n = 1, n_loc
  call set_ele_attribute (eles(n)%ele, 'K1 = 0.1*c_light', lat, err_flag)
enddo
```

This example sets the `K1` attribute of all elements named `Q01W`. `set_ele_attribute` checks whether an element is actually free to be varied and sets the `err_flag` logical accordingly. An element's attribute may not be freely varied if, for example, the attribute is controlled via an `Overlay`.

Chapter 30

Reading and Writing Lattices

30.1 Reading in Lattices

xsif_parser bmad_parserbmad_parser2

Bmad has routines for reading XSIF (§2.1) and *Bmad* formatted lattice files. The subroutine to read in an XSIF lattice file is xsif_parser. There are two subroutines in *Bmad* to read in a *Bmad* standard lattice file: bmad_parser and bmad_parser2. `bmad_parser` is used to initialize a `lat_struct` (§28) structure from scratch using the information from a lattice file. Unless told otherwise, after reading in the lattice, `bmad_parser` will compute the 6x6 transfer matrices for each element and this information will be stored in the `digested file` (§2.4) that is created. Notice that `bmad_parser` does *not* compute any Twiss parameters.

bmad_parser2 is typically used after `bmad_parser` if there is additional information that needs to be added to the lattice. For example, consider the case where the aperture limits for the elements is stored in a file that is separate from the main lattice definition file and it is undesireable to put a `call` statement in one file to reference the other. To read in the lattice information along with the aperture limits, there are two possibilities: One possibility is to create a third file that calls the first two:

```
! This is a file to be called by bmad_parser
call, file = 'lattice_file'
call, file = 'aperture_file'
```

and then just use `bmad_parser` to parse this third file. The alternative is to use `bmad_parser2` so that the program code looks like:

```
! program code to read in everything
type (lat_struct) lat
call bmad_parser ('lattice_file', lat)       ! read in a lattice.
call bmad_parser2 ('aperture_file', lat)     ! read in the aperture limits.
```

An alternative to using `bmad_parser` and `xsif_parser` is to use the combined *Bmad* and XSIF parser bmad_and_xsif_parser. This parser will assume that the input file is using *Bmad* syntax unless the file name is prefixed by the string "`xsif::`".

30.2 Digested Files

Since parsing can be slow, once the `bmad_parser` routine has transfered the information from a lattice file into the `lat_struct` it will make what is called a digested file. A digested file is an image of the `lat_struct` in binary form. When `bmad_parser` is called, it first looks in the same directory as the lattice file for a digested file whose name is of the form:

```
'digested_' // LAT_FILE
```

where `LAT_FILE` is the lattice file name. If `bmad_parser` finds the digested file, it checks that the file is not out–of–date (that is, whether the lattice file(s) have been modified after the digested file is made). `bmad_parser` can do this since the digested file contains the names and the dates of all the lattice files that were involved. Also stored in the digested file is the "*Bmad* `version number`". The *Bmad* version number is a global parameter that is increased (not too frequently) each time a code change involves modifying the structure of the `lat_struct` or `ele_struct`. If the *Bmad* version number in the digested file does not agree with the number current when `bmad_parser` was compiled, or if the digested file is out–of–date, a warning will be printed, and `bmad_parser` will reparse the lattice and create a new digested file.

Since computing Taylor Maps can be very time intensive, `bmad_parser` tries to reuse Taylor Maps it finds in the digested file even if the digested file is out–of–date. To make sure that everything is OK, `bmad_parser` will check that the attribute values of an element needing a Taylor map are the same as the attribute values of a corresponding element in the digested file before it reuses the map. Element names are not a factor in this decision.

This leads to the following trick: If you want to read in a lattice where there is no corresponding digested file, and if there is another digested file that has elements with the correct Taylor Maps, then, to save on the map computation time, simply make a copy of the digested file with the digested file name corresponding to the first lattice.

read_digested_bmad_file write_digested_bmad_file The digested file is in binary format and is not human readable but it can provide a convenient mechanism for transporting lattices between programs. For example, say you have read in a lattice, changed some parameters in the `lat_struct`, and now you want to do some analysis on this modified `lat_struct` using a different program. One possibility is to have the first program create a digested file

```
call write_digested_bmad_file ('digested_file_of_mine', lat)
```

and then read the digested file in with the second program

```
call read_digested_bmad_file ('digested_file_of_mine', lat)
```

An alternative to writing a digested file is to write a lattice file using `write_bmad_lattice_file`

30.3 Writing Lattice files

write_bmad_lattice_file To create a *Bmad* lattice file from a `lat_struct` instance, use the routine write_bmad_lattice_file. *MAD*–8, *MAD*–X, or `XSIF` compatible lattice files can be created from a `lat_struct` variable using the routine write_lattice_in_foreign_format:

```
type (lat_struct) lat              ! lattice
...
call bmad_parser (bmad_lat_file, lat)              ! Read in a lattice
call write_lattice_in_foreign_format ('lat.mad', 'MAD-8', lat)  ! create MAD file
```

Information can be lost when creating a *MAD* or `XSIF` file. For example, neither *MAD* nor `XSIF` has the concept of things such as `overlays` and `groups`.

Chapter 31

Normal Modes: Twiss Parameters, Coupling, Emittances, Etc.

31.1 Components in the Ele_struct

The `ele_struct` (§27) has a number of components that hold information on the Twiss parameters, dispersion, and coupling at the exit end of the element. The Twiss parameters of the three normal modes (§20.1) are contained in the `ele%a`, `ele%b`, and `ele%z` components which are of type `twiss_struct`:

```
type twiss_struct
  real(rp) beta         ! Twiss Beta function
  real(rp) alpha        ! Twiss Alpha function
  real(rp) gamma        ! Twiss gamma function
  real(rp) phi          ! Normal mode Phase advance
  real(rp) eta          ! Normal mode dispersion
  real(rp) etap         ! Normal mode dispersion derivative
  real(rp) sigma        ! Normal mode beam size
  real(rp) sigma_p      ! Normal mode beam size derivative
  real(rp) emit         ! Geometric emittance
  real(rp) norm_emit    ! Energy normalized emittance (= γ  ϵ)
end type
```

The projected horizontal and vertical dispersions in an `ele_struct` are contained in the `ele%x` and `ele%y` components. These components are of type `xy_disp_struct`:

```
type xy_disp_struct
  real(rp) eta     ! Projected dispersion
  real(rp) etap    ! Projected dispersion derivative.
end type
```

The components `ele%emit`, `ele%norm_emit`, `ele%sigma`, `ele%sigma_p` are not set by the standard *Bmad* routines and are present for use by any program.

The relationship between the projected and normal mode dispersions are given by Eq. (20.16). The 2x2 coupling matrix $\mathbf{C}$ (Eq. (20.5)) is stored in the `ele%c_mat(2,2)` component of the `ele_struct` and the γ factor of Eq. (20.5) is stored in the `ele%gamma_c` component. There are several routines to manipulate the coupling factors. For example:

```
c_to_cbar(ele, cbar_mat)             ! Form Cbar(2,2) matrix
make_v_mats(ele, v_mat, v_inv_mat)   ! Form V coupling matrices.
```

See §39.22 for a complete listing of such routines.

Since the normal mode and projected dispersions are related, when one is changed within a program the appropriate change must be made to the other. To make sure everything is consistent, the set_flags_-for_changed_attribute routine can be used. Example:

```
type (lat_struct), target :: lat
real(rp), pointer :: attrib_ptr
...
attrib_ptr => lat%ele(ix_ele)%value(k1$) ! Point to some attribute.
attrib_ptr = value                       ! Change the value.
call set_flags_for_changed_attribute (lat%ele(ix_ele), attrib_ptr)
```

The %mode_flip logical component of an ele_struct indicates whether the a and b normal modes have been flipped relative to the beginning of the lattice. See Sagan and Rubin[Sagan99] for a discussion of this. The convention adopted by *Bmad* is that the %a component of all the elements in a lattice will all correspond to the same physical normal mode. Similarly, the %b component of all the elements will all correspond to some (other) physical normal mode. That is, at an element where there is a mode flip (with %mode_flip set to True), the %a component actually corresponds to the **B** matrix element in Eq. (20.3) and vice versa. The advantage of this convention is that calculations of mode properties (for example the emittance), can ignore whether the modes are flipped or not.

The normal mode analysis of Sagan and Rubin, while it has the benefit of simplicity, is strictly only applicable to lattices where the RF cavities are turned off. The full 6-dimensional analysis is summarized by Wolski[Wolski06]. The normal_mode3_calc routine perform the full analysis. The results are put in the %mode3 component of the ele_struct which is of type mode3_struct:

```
type mode3_struct
  real(rp) v(6,6)
  type (twiss_struct) a, b, c
  type (twiss_struct) x, y
end type
```

The 6-dimensional mode3%v(6,6) component is the analog of the 4-dimensional **V** matrix appearing in Eq. (20.2).

31.2 Tune and Twiss Parameter Calculations

A calculation of the Twiss parameters starts with the Twiss parameters at the beginning of the lattice. For linear machines, these Twiss parameters are generally set in the input lattice file (§8.4). For circular machines, the routine twiss_at_start may be used (§8.4)

```
type (lat_struct) lat
...
if (lat%param%geometry == closed$) call twiss_at_start(lat)
```

In either case, the initial Twiss parameters are placed in lat%ele(0). The tune is placed in the variables lat%a%tune and lat%b%tune.

To propagate the Twiss, coupling and dispersion parameters from the start of the lattice to the end, the routine, twiss_propagate_all can be used. This routine works by repeated calls to twiss_propagate1 which does a single propagation from one element to another. The Twiss propagation depends upon the transfer matrices having already computed (§32). twiss_propagate_all also computes the Twiss parameters for all the lattice branches.

Before any Twiss parameters can be calculated, the transfer matrices stored in the lattice elements must be computed. bmad_parser does this automatically about the zero orbit. If, to see nonlinear effects,

a different orbit needs to be used for the reference, The routine twiss_and_track can be used. For example:

```
type (lat_struct) lat
type (coord_struct), allocatable :: orbit(:)
call bmad_parser ('my_lattice', lat)
call twiss_and_track (lat, orb, ok)
```

Once the starting Twiss parameters are set, twiss_propagate_all can be used to propagate the Twiss parameters to the rest of the elements

The routine twiss_and_track_at_s can be used to calculate the Twiss parameters at any given longitudinal location. Alternatively, to propagate the Twiss parameters partially through a given element use the routine twiss_and_track_intra_ele.

31.3 Tune Setting

The routine set_tune can be used to set the transverse tunes:

```
set_tune (phi_a_set, phi_b_set, dk1, lat, orb_, ok)
```

`set_tune` varies quadrupole strengths until the desired tunes are achieved. As input,`set_tune` takes an argument `dk1(:)` which is an array that specifies the relative change to be make to the quadrupoles in the lattice.

To set the longitudinal (synchrotron) tune, the routine set_z_tune can be used. set_z_tune works by varying rf cavity voltages until the desired tune is achieved.

31.4 Emittances & Radiation Integrals

See Section §18.2 for details on the radiation integral formulas.

The routine radiation_integrals is used to calculate the normal mode emittances along with the radiation integrals:

```
type (lat_struct) lat
type (normal_modes_struct) modes
type (rad_int_all_ele_struct) ele_rad_int
...
call radiation_integrals (lat, orbit, modes, rad_int_by_ele = ele_rad_int)
```

The `modes` argument, which is of type `normal_modes_struct`, holds the radiation integrals integrated over the entire lattice.

```
type normal_modes_struct
  real(rp) synch_int(0:3) ! Synchrotron integrals I0, I1, I2, and I3
  real(rp) sigE_E         ! SigmaE/E
  real(rp) sig_z          ! Sigma_Z
  real(rp) e_loss         ! Energy loss / turn (eV)
  real(rp) rf_voltage     ! Total rfcavity voltage (eV)
  real(rp) pz_aperture    ! pz aperture limit
  type (anormal_mode_struct)  a, b, z
  type (linac_normal_mode_struct) lin
end type
```

In particular, the `%a`, `%b`, and `%z` components, which are of type `anormal_mode_struct` hold the emittance values:

```
type anormal_mode_struct
  real(rp) emittance        ! Beam emittance
  real(rp) synch_int(4:6)   ! Synchrotron integrals
  real(rp) j_damp           ! damping partition number
  real(rp) alpha_damp       ! damping per turn
  real(rp) chrom            ! Chromaticity
  real(rp) tune             ! "Fractional" tune in radians
end type
```

The `ele_rad_int` argument, which is is of type `rad_int_all_ele_struct`, holds the radiation integrals on an element-by-element basis.

```
type rad_int_all_ele_struct
  type (rad_int1_struct), allocatable :: ele(:) ! Array is indexed from 0
end type
```

31.5 Chromaticity Calculation

For a circular lattice, chrom_calc calculates the chromaticity by calculating the tune change with change in beam energy.

chrom_tune sets the chromaticity by varying the sextupoles. This is a very simple routine that simply divides the sextupoles into two families based upon the local beta functions at the sextupoles.

Chapter 32

Tracking and Transfer Maps

32.1 The coord_struct

The `coord_struct` holds the coordinates of a particle at a given longitudinal position in the lattice. The definition of the `coord_struct` is

```
  type coord_struct
    real(rp) vec(6)     ! (x, px, y, py, z, pz)
    real(rp) s          ! Longitudinal position.
    real(rp) t          ! Absolute time (not relative to reference).
    real(rp) spin(3)    ! (x, y, z) Spin vector
    real(rp) field(2)   ! Photon (x, y) field intensity.
    real(rp) phase(2)   ! Photon (x, y) phase.
    real(rp) charge     ! charge in a particle (Coul).
    real(rp) path_len   ! path length (used by coherent photons).
    real(rp) r          ! For general use. Not used by Bmad.
    real(rp) p0c        ! For non-photons: Reference momentum. Negative -> going backwards.
                        !     For photons: Photon momentum (not reference).
    real(rp) beta       ! Velocity / c_light.
    integer ix_ele      ! Index of element particle was tracked through.
                        !   May be -1 or -2 if element is not associated with a lattice.
    integer ix_user     ! Not used by Bmad
    integer state       ! alive$, lost$, lost_neg_x_aperture$, etc.
    integer direction   ! Longitudinal direction of motion = +/- 1
    integer species     ! Positron$, proton$, etc.
    integer location    ! upstream_end$, inside$, or downstream_end$
end type
```

The components of the `coord_struct`:

%beta
: The normalized velocity v/c is stored in `%beta`.

%direction
: Longitudinal direction of travel. A setting of +1 is in the forward (downstream) direction and a setting of -1 is in the reverse (upstream) direction (§14.1.2). Notice that the setting of `direction` is independent of the orientation of the lattice element the particle is traveling through. That is,

for an element with reversed `orientation` (`ele%orientation` = -1), a particle with `direction` = 1 will be traveling towards the `entrance` end of the element and with `direction` = -1 the particle will be traveling towards the `exit` end (§14.1.2).

%field_x, %field_y
The `%field_x` and `%field_y` components are for photon tracking and are in units of field/sqrt(cross-section-area). That is, the square of these units is an intensity. It is up to individual programs to define an overall scaling factor for the intensity if desired.

%ix_ele
The `%ix_ele` component gives the index of the element in the `lat%branch(ib)%ele(:)` array that was tracked through. If the element tracked through is not associated with a lattice, `%ix_ele` is set to -1. When initializing a `coord_struct` (see below), `%ix_ele` will be initialized to `not_set$`.

%ix_user
The `%ix_user` component is for use by code outside of the *Bmad* library. This component will not be modified by *Bmad*.

%location
The `%location` component indicates where a particle is longitudinally with respect to the element being tracked. `%location` will be on of:

```
entrance_end$
inside$
exit_end$
```

`entrance_end$` indicates that the particle is at the element's entrance ($-s$) end and `exit_end$` indicates that the particle is at the element's exit ($+s$) end. `inside$` indicates that the particle is in between. If the element has edge fields (for example, the `e1` and `e2` edge fields of a bend), a particle at the `entrance_end$` or `exit_end$` is considered to be just outside the element.

%p0c
For charged-particles, the reference momentum in eV is stored in the `%p0c` component. For photons, `%p0c` is the actual (not reference) momentum. For charged-particles, `%p0c` may be negative if the particle is traveling backwards longitudinally. For photons, `%vec(6)` (β_z) will be negative if the photon is going backward.

%r
The `%r` component is for use by code outside of the *Bmad* library. *Bmad* will not modify this component.

%s
The `%s` component gives the absolute s-position of the particle. When tracking through an element (say with Runge-Kutta tracking), and when the particle coordinates is expressed in element body coordinates (§14.3), the s-position at any point within the element, by definition, is independent of any misalignments the element has as long as the element is not reversed. If the element is reversed, the s-position is reversed as well.

%spin(3)
The `%spin(3)` component gives a particle's (x, y, z) spin vector (§19.1).

%state
The `%state` component will be one of:

```
not_set$
alive$
lost$
```

```
lost_neg_x_aperture$
lost_pos_x_aperture$
lost_neg_y_aperture$
lost_pos_y_aperture$
lost_z_aperture$
```

The `not_set$` setting indicates that the `coord_struct` has not yet been used in tracking. The `alive$` setting indicates that the particle is alive. If a particle is "dead", the `%state` component will be set to one of the other settings. The `lost_neg_x_aperture$` setting indicates that the particle was lost at an aperture on the $-x$ side of the element. The `lost_z_aperture$` setting is used to indicate that the particle tried to "turn around". This can happen, for example, with strong magnetic fields or when a particle has been decelerated too much. The reason why the particle is marked lost in this case is due to the fact that s-based tracking algorithms cannot handle particles that reverse direction. The exception is that the `time_runge_kutta` (§5.1) tracking method can handle particle reversal so in this case, particles will not be declared lost if they reverse direction.

The `lost$` setting is used when neither of the other `lost_*_aperture$` settings are not appropriate. For example, `lost$` is used in Runge-Kutta tracking when the adaptive step size becomes too small (this may happen if the fields do not obey Maxwell's equations).

To convert the integer value of `%state` to a string that can be printed, use the function coord_state_name

```
type (coord_struct) orbit
print *, 'State of the orbit: ', coord_state_name(orbit%state)
```

%t

`%t` gives the absolute time.

%vec(:)

The `%vec(:)` array defines the phase space coordinants (§14.4.2). Note that for photons, the definition of the phase space coordinates (§14.4.4) is different from that used for charged particles.

To initialize a `coord_struct` so it can be used as the start of tracking, the init_coord routine can be used:

```
type (coord_struct) start_orb
real(rp) phase_space_start(6)
...
phase_space_start = [...]
call init_coord (start_orb, phase_space_start, lat%ele(i), lat%param%particle)
```

Here `init_coord` will initialize `start_orb` appropriately for tracking through element `lat%ele(i)` with the particle species set to the species of the reference particle given by `lat%param%particle`.

32.2 Tracking Through a Single Element

track1 is the routine used for tracking through a single element

```
type (coord_struct), start_orb, end_orb
type (ele_struct) ele
real(rp) start_phase_space(6)
logical err
...
start_phase_space = [...]
call init_coord (start_orb, start_phase_space, ele, photon$)
```

```
call track1 (start_orb, ele, end_orb, err_flag = err)
if (.not. particle_is_moving_forward(end_orb)) then
  print *, 'Particle is lost and gone forever...'
endif
```

To check if a particle is still traveling in the forward direction, the particle_is_moving_forward routine can be used as shown in the above example.

The "virtual" entrance and exit ends of a lattice element are, by definition, where the physical ends of the element would be if there were no offsets. In particular, if an element has a finite `z_offset` (§27.11), the physical ends will be displaced from the virtual ends. The position `ds` of a particle with respect to the physical entrance end of the element is

```
ds = coord%s - (ele%s + ele%value(z_offset_tot$) - ele%value(l$))
```

When tracking through an element, the starting and ending positions always correspond to the virtual ends. If there is a finite `z_offset`, the tracking of the element will involve tracking through drifts just before and just after the tracking of the body of the element so that the particle ends at the proper virtual exit end.

Note: The z phase space component of the orbit (`%vec(5)`) is independent of the value of `ele%ref_time` even though the reference time is used to define z (See Eq. (14.28)). This is true since the starting reference time that is used for a particle is arbitrary. For example, when tracking multiple bunches, the reference time is typically set so that a particle at the center of a bunch has $z = 0$. Also, in a ring, `ele%ref_time` is only the reference time for the first turn through an element. Since *Bmad* does not keep track of turn number, there is no way for *Bmad* to know what the true reference time is other than to calculate it from the value of z!

32.3 Tracking Through a Lattice Branch

When tracking through a lattice branch, one often defines an array of `coord_structs` – one for each element of the lattice branch. In this case, the i^{th} `coord_struct` corresponds to the particle coordinates at the end of the i^{th} element. Since the number of elements in the lattice is not known in advance, the array must be declared to be allocatable. The lower bound of the array must be set to zero to match a `lat%branch(i)%ele(:)` array. The upper bound should be the upper bound of the `%branch(i)%ele(:)` array. The routine reallocate_coord will allocate an array of `coord_structs`:

```
type (coord_struct), allocatable :: orbit(:)
type (lat_struct) lat
...
call reallocate_coord (orbit, lat, ix_branch)
```

Alternatively, the `save` attribute can be used so that the array stays around until the next time the routine is called

```
type (coord_struct), allocatable, save :: orb(:)
```

Saving the `coord_stuct` is faster but leaves memory tied up. Note that in the main program, the `save` attribute is not permitted If a `coord_struct` array is passed to a routine, the routine must explicitly set the lower bound to zero if the array is not declared as allocatable:

```
subroutine my_routine (orbit1, orbit2)
  use bmad
  implicit none
  type (coord_struct), allocatable :: orbit1(:)  ! OK
  type (coord_struct) orbit2(0:)                 ! Also OK
  ...
```

Declaring the array allocatable is mandatory if the array is to be resized or the array is passed to a routine that declares it allocatable.

For an entire lattice, the `coord_array_struct` can be used to define an array of `coord_array` arrays:

```
type coord_array_struct
  type (coord_struct), allocatable :: orb(:)
end type
```

The routine reallocate_coord_array will allocate an `coord_array_struct` instance

```
type (coord_array_struct), allocatable :: all_orbit(:)
type (lat_struct) lat
...
call reallocate_coord_array (all_orbit, lat)
...
```

Once an array of `coord_struct` elements is defined, the track_all routine can be used to track through a given lattice branch

```
type (coord_struct), allocatable :: orbit(:)
integer ib, track_state
...
ib = 1                        ! Branch to track through
call init_coord(orbit(0), init_phase_space, lat%branch(ib)%ele(0), proton$)
call track_all (lat, orbit, ib, track_state, err_flag)
if (track_state /= moving_forward$) then
  print *, 'Particle lost at element:', track_state
  print *, 'Aperture lost at: ', coord_state_name(orbit(track_state)%state)
```

After tracking, `orbit(i)` will correspond to the particles orbit at the end of `lat%branch(ib)%ele(i)`.

For routines like `track_all` where an array of `coord_structs` is used, an integer `track_state` argument is provided that is set to `moving_forward$` if the particle survives to the end, or is set to the index of the element at which the particle either hit an aperture or the particle's longitudinal velocity is reversed.

The reason why the reversal of the particle's longitudinal velocity stops tracking is due to the fact that the standard tracking routines, which are s-based (that is, use longitudinal position s as the independent coordinate), are not designed to handle particles that reverse direction. To properly handle this situation, time-based tracking needs to be used (§32.11). Notice that this is different from tracking a particle in the reversed ($-s$) direction.

Alternatively to `track_all`, the routine track_many can be used to track through a selected number of elements or to track backwards (See §32.14).

The `track_all` routine serves as a good example of how tracking works. A condensed version of the code is shown in Fig. 32.1. The call to `track1` (line 18) tracks through one element from the exit end of the $n-1^{st}$ element to the exit end of the n^{th} element.

```
1     subroutine track_all (lat, orbit, ix_branch, track_state, err_flag)
2       use bmad
3       implicit none
4       type (lat_struct), target :: lat
5       type (branch_struct), pointer :: branch
6       type (coord_struct), allocatable :: orbit(:)
7       integer, optional :: ix_branch, track_state
8       logical, optional :: err_flag
9       logical err
10
11      !
12
13      branch => lat%param(integer_option(0, ix_branch))
14      branch%param%ix_track = moving_forward
15      if (present(track_state)) track_state = moving_forward\$
16
17      do n = 1, branch%n_ele_track
18        call track1 (orbit(n-1), branch%ele(n), branch%param, orbit(n), err_flag = err)
19        if (.not. particle_is_moving_forward(orbit(n))) then
20          if (present(track_state)) track_state = n
21          orbit(n+1:)%status = not_set$
22          return
23        endif
24      enddo
25    end subroutine
```

Figure 32.1: Condensed track_all code.

32.4 Forking from Branch to Branch

Tracking from a `fork` or `photon_fork` (§3.19) element to the target `branch` is not "automatic". That is, since the requirements of how to handle forking can vary greatly from one situation to the next, *Bmad* does not try to track from one `branch` to the next in any one of its tracking routines.

The discussion here is restricted to the case where the particle being tracked is simply transferred from the forking element to the target branch. [Thus the subject of photon generation is not covered here.]

There are two cases discussed here. The first case is when a given branch (called `to_branch`) has an associated forking element in the `from_branch` that forks to the beginning of the `to_branch`. Appropriate code is:

```
type (lat_struct), target :: lat   ! Lattice
type (branch_struct) :: to_branch  ! Given target branch
type (branch_struct), pointer  :: from_branch ! Base branch
type (ele_struct), pointer :: fork_ele
type (coord_struct), allocatable :: from_orbit(:), to_orbit(:)
integer ib_from, ie_from

ib_from  = to_branch%ix_from_branch

if (ib_from < 0) then
  ! Not forked to ...

else
```

```
    from_branch => lat%branch(ib_from)
    ie_from = to_branch%ix_from_ele
    fork_ele => from_branch%ele(ie_from)
    to_orbit(0) = from_orbit(ie_from)
    call transfer_twiss (fork_ele, to_branch%ele(0))
  endif
```

`from_orbit(0:)` and `to_orbit(0:)` are arrays holding the orbits at the exit end of the elements for the `from_branch` and `to_branch` respectively. The call to transfer_twiss transfers the Twiss values to the `to_branch` which can then be propagated through the `to_branch` using `twiss_propagate_all`.

The second case starts with the `fork_ele` forking element. This is similar to the first case but is a bit more general since here the element, called `to_ele` in the `to_branch` that is connected to `fork_ele` need not be the starting element of `to_branch`.

```
type (lat_struct), target :: lat   ! Lattice
type (branch_struct), pointer :: to_branch  ! Target branch
type (ele_struct), pointer :: to_ele
type (coord_struct), allocatable :: from_orbit(:), to_orbit(:)
integer ib_to, ie_to

ib_to  = nint(fork_ele%value(ix_to_branch$))
ie_to  = nint(fork_ele%value(ix_to_element$))

to_branch => lat%branch(ib_to)
to_ele => to_branch%ele(ie_to)
to_orbit(to_ele%ix_ele) = from_orbit(fork_ele%ix_ele)
```

Notice that, by convention, the transferred orbit is located at the exit end of the `to_ele`.

32.5 Multi-turn Tracking

Multi-turn tracking over a branch is simply a matter of setting the coordinates at the beginning zeroth element equal to the last tracked element within a loop:

```
type (lat_struct) lat             ! lattice to track through
type (coord_struct), allocatable :: orbit(:)
...
call reallocate_coord (orbit, lat, ix_branch = 1)
orbit(0)%vec = [0.01, 0.2, 0.3, 0.4, 0.0, 0.0] ! init
do i = 1, n_turns
  call track_all (lat, orbit, 1)
  orbit(0) = orbit(lat%branch(1)%n_ele_track)
end do
```

Often times it is only the root branch, `branch(0)`, that is to be tracked. In this case, the above reduces to

```
type (lat_struct) lat             ! lattice to track through
type (coord_struct), allocatable :: orbit(:)
...
call reallocate_coord (orbit, lat%n_ele_max)
orbit(0)%vec = [0.01, 0.2, 0.3, 0.4, 0.0, 0.0] ! init
do i = 1, n_turns
  call track_all (lat, orbit)
```

```
    orbit(0) = orbit(lat%n_ele_track)
  end do
```

32.6 Closed Orbit Calculation

For a circular lattice the closed orbit may be calculated using `closed_orbit_calc`. By default this routine will track in the forward direction which is acceptable unless the particle you are trying to simulate is traveling in the reverse direction and there is radiation damping on. In this case you must tell `closed_orbit_calc` to do backward tracking. This routine works by iteratively converging on the closed orbit using the 1–turn matrix to calculate the next guess. On rare occasions if the nonlinearities are strong enough, this can fail to converge. An alternative routine is `closed_orbit_from_tracking` which tries to do things in a more robust way but with a large speed penalty.

32.7 Partial Tracking through elements

There are several routines for tracking partially through an element:

```
twiss_and_track_at_s
twiss_and_track_intra_ele
track_from_s_to_s
twiss_and_track_from_s_to_s
mat6_from_s_to_s
```

These routines make use of element "slices" (§29.1) which are elements that represent some sub-section of an element. There are two routines for creating slices:

```
create_element_slice
create_uniform_element_slice
```

It is important to note that to slice up a given element, the `s_to_s` tracking routines will not always work. For example, consider the case where a given element is followed by a zero length multipole. If `track_from_s_to_s` is called with a value for `s2` (the value at the end of the track) which corresponds to the exit end of this element, the result will also include tracking through the zero length multipole. Thus, in the case where a given element is to be sliced, one or the other of the two slice routines given above must be first used to create an element slice then this slice can be used for tracking.

32.8 Apertures

The routine check_aperture_limit checks the aperture at a given element. The `ele%aperture_type` component determines the type of aperture. Possible values for `ele%aperture_type` are

```
rectangular$
elliptical$
custom$
```

With `custom$`, a program needs to be linked with a custom version of check_aperture_limit_custom.

The logical `bmad_com%param%aperture_limit_on` determines if element apertures (See §4.8) are used to determine if a particle has been lost in tracking. The default `bmad_com%aperture_limit_on` is True. Even if this is False there is a "hard" aperture limit set by `bmad_com%max_aperture_limit`. This hard limit is used to prevent floating point overflows. The default hard aperture limit is 1000 meters.

Additionally, even if a particle is within the hard limit, some routines will mark a particle as lost if the tracking calculation will result in an overflow.

`lat%param%lost` is the logical to check to see if a particle has been lost. `lat%param%ix_lost` is set by `track_all` and gives the index of the element at which a particle is lost. `%param%end_lost_at` gives which end the particle was lost at. The possible values for `lat%param%end_lost_at` are:

```
entrance_end$
exit_end$
```

When tracking forward, if a particle is lost at the exit end of an element then the place where the orbit was outside the aperture is at `orbit(ix)` where `ix` is the index of the element where the particle is lost (given by `lat%param%ix_lost`). If the particle is lost at the entrance end then the appropriate index is one less (remember that `orbit(i)` is the orbit at the exit end of an element).

To tell how a particle is lost, check the `lat%param%plane_lost_at` parameter. Possible values for this are:

```
x_plane$
y_plane$
z_plane$
```

`x_plane$` and `y_plane$` indicate that the particle was lost either horizontally, or vertically. `z_plane$` indicates that the particle was turned around in an `lcavity` element. That is, the cavity was decelerating the particle and the particle did not not have enough energy going into the cavity to make it to the exit.

32.9 Custom Tracking

Custom code can be used for tracking. This is discussed in detail in sections §33.2 and §33.3.

32.10 Tracking Methods

For each element the method of tracking may be set either via the input lattice file (see §5.1) or directly in the program by setting the `%tracking_method` attribute of an element

```
type (ele_struct) ele
...
ele%tracking_method = symp_lie_ptc$  ! for symp_lie_ptc, tracking
print *, 'Tracking_method: ', calc_method_name(ele%tracking_method)
```

To form the corresponding parameter to a given tracking method just put "$" after the name. For example, the `bmad_standard` tracking method is specified by the `bmad_standard$` parameter. To convert the integer `%tracking_method` value to a string suitable for printing, use the `tracking_method_name` array.

It should be noted that except for `linear` tracking, none of the *Bmad* tracking routines make use of the `ele%mat6` transfer matrix. The reverse, however, is not true. The transfer matrix routines (`lat_make_mat6`, etc.) will do tracking.

For determining what tracking methods are valid for a given element, use valid_tracking_method and valid_mat6_calc_method functions

```
print *, 'Method is valid: ', valid_tracking_method(ele, symp_lie_ptc$)
```

Bmad simulates radiation damping and excitation by applying a kick just before and after each element.

32.11 Using Time as the Independent Variable

Time tracking uses time as the independent variable as opposed to the standard s based tracking. Time tracking is useful when a particle's trajectory can reverse itself longitudinally. For example, low energy particles generated when a relativistic particle hits the vacuum chamber wall are good candidates for time tracking.

Currently, the only `ele%tracking_method` available for time tracking is `time_runge_kutta$`. Time tracking needs extra bookkeeping due to the fact that the particle may reverse directions. See the `dark_current_tracker` program as an example.

Note: Using time as the independent variable can be used with both absolute and relative time tracking (§22.1).

32.12 Absolute/Relative Time Tracking

Absolute or relative time tracking (§22.1) can be set after the lattice file is parsed, by setting the `%absolute_time_tracking` component of the `lat_struct`. when `%absolute_time_tracking` is toggled, the autoscale_phase_and_amp must be called to reset the appropriate phase offsets and scale amplitudes.

32.13 Taylor Maps

A list of routines for manipulating Taylor maps is given in §39.36. The order of the Taylor maps is set in the lattice file using the `parameter` statement (§8.1). In a program this can be overridden using the routine set_ptc. The routine taylor_coef can be used to get the coefficient of any given term in a Taylor map.

```
type (taylor_struct) t_map(6)
...
print *, 'out(4)=coef * in(1)^2:', taylor_coef(t_map(4), 1, 1)
print *, 'out(4)=coef * in(1)^2:', taylor_coef(t_map(4), [2,0,0,0,0,0])
```

Transfer Taylor maps for an element are generated as needed when the `ele%tracking_method` or `ele%mat6_calc_method` is set to `Symp_Lie_Bmad`, `Symp_Lie_PTC`, `Symp_Map`, or `Taylor`. Since generating a map can take an appreciable time, *Bmad* follows the rule that once generated, these maps are never regenerated unless an element attribute is changed. To generate a Taylor map within an element irregardless of the `ele%tracking_method` or `ele%mat6_calc_method` settings use the routine ele_to_taylor. This routine will kill any old Taylor map before making any new one. To kill a Taylor map (which frees up the memory it takes up) use the routine kill_taylor.

To test whether a `taylor_struct` variable has an associated Taylor map. That is, to test whether memory has been allocated for the map, use the Fortran associated function:

```
type (bmad_taylor) taylor(6)
...
if (associated(taylor(1)%term)) then  ! If has a map ...
  ...
```

To concatenate the Taylor maps in a set of elements the routine concat_taylor can be used

```
type (lat_struct) lat          ! lattice
type (taylor_struct) taylor(6)  ! taylor map
```

```
...
call taylor_make_unit (taylor)  ! Make a unit map
do i = i1+1, i2
  call concat_taylor (taylor, lat%ele(i)%taylor, taylor)
enddo
```

The above example forms the transfer Taylor map starting at the end of element `i1` to the end of element `i2`. Note: This example assumes that all the elements have a Taylor map. The problem with concatenating maps is that if there is a constant term in the map "feed down" can make the result inaccurate (§21.1. To get around this one can "track" a taylor map through an element using symplectic integration.

```
type (lat_struct) lat          ! lattice
type (taylor_struct) taylor(6)  ! taylor map
...
call taylor_make_unit (taylor)  ! Make a unit map
do i = i1+1, i2
  call call taylor_propagate1 (taylor, lat%ele(i), lat%param)
enddo
```

Symplectic integration is typically much slower than concatenation. The width of an integration step is given by `%ele%value(ds_step$`. The attribute `%ele%value(num_steps$)`, which gives the number of integration steps, is a dependent variable (§4.1) and should not be set directly. The order of the integrator (§21.1) is given by `%ele%integrator_order`. PTC (§34) currently implements integrators of order 2, 4, or 6.

32.14 Tracking Backwards

Tracking backwards happens when a particle goes in the direction of decreasing `s`. This is indicated in the `coord_struct` by `coord%direction = -1`.

The `time_runge_kutta` `tracking_method` is able to handle the situation where a particle would reverse direction due to string electric or magnetic fields. All other tracking methods are not able to handle this since they are position (s) based, instead of time based. With non `time_runge_kutta` tracking methods, the equations of motion become singular when a particle "tries" to reverse direction. In such a situation, the particle will be marked as lost and the `coord_struct` will have `%status` $/=$ `alive$`.

The "problem" with tracking backwards is that the reference time $t_0(s)$ that is used to compute the z phase space coordinate (Eq. (14.28)) is independent of the motion of any particle. That is, a particle traveling backwards will have a large negative z. As an alternative to tracking backwards, reversing the lattice and tracking forwards is possible (§32.15).

One restriction with backwards tracking is that, for simplicities sake, *Bmad* does not compute transfer matrices for propagation in the backwards direction. Tracking with reversed elements does not have this restriction.

32.15 Reversed Elements and Tracking

With a lattice element that is reversed (`s:ele.reverse`), the transfer map and transfer matrix that is stored in the element is, just like for a non-reversed element, appropriate for a particle traveling in the $+s$ direction.

32.16 Beam (Particle Distribution) Tracking

Tracking with multiple particles is done with a `beam_struct` instance:

```
type beam_struct
  type (bunch_struct), allocatable :: bunch(:)
end type
```

A `beam_struct` is composed of an array of bunches of type `bunch_struct`:

```
type bunch_struct
  type (coord_struct), allocatable :: particle(:)
  integer, allocatable :: ix_z(:)  ! bunch%ix_z(1) is index of head particle, etc.
  real(rp) charge_tot  ! Total charge in bunch (Coul).
  real(rp) charge_live ! Total charge of live particles in bunch (Coul).
  real(rp) z_center    ! Longitudinal center of bunch (m). Note: Generally, z_center of
                       !   bunch #1 is 0 and z_center of the other bunches is negative.
  real(rp) t_center    ! Center of bunch creation time relative to head bunch.
  integer species      ! electron$, proton$, etc.
  integer ix_ele       ! Element this bunch is at.
  integer ix_bunch     ! Bunch index. Head bunch = 1, etc.
end type
```

The `bunch_struct` has an array of particles of type `coord_struct` (§32.1).

Initializing a `beam_struct` to conform to some initial set of Twiss parameters and emittances is done using the routine init_beam_distribution:

```
type (lat_struct) lat
type (beam_init_struct) beam_init
type (beam_struct) beam
...
call init_beam_distribution (lat%ele(0), lat%param, beam_init, beam)
```

The `lat%ele(0)` argument, which is of type `ele_struct`, gives the twiss parameters to initialize the beam to. In this case, we are starting tracking from the beginning of the lattice. The `beam_init` argument which is of type `beam_init` gives additional information, like emittances, which is needed to initialize the beam. See chapter §10 for more details.

Tracking a beam is done using the track_beam routine

```
type (lat_struct) lat
type (beam_struct) beam
...
call track_beam (lat, beam)
```

or, for tracking element by element, track1_bunch can be used.

For analyzing a bunch of particles, that is, for computing such things as the sigma matrix from the particle distribution, the calc_bunch_params routine can be used.

Notice that when a particle bunch is tracked to a given longitudinal position in the lattice, all the particles of the bunch are at that longitudinal position (this is no different if particles are tracked individually independent of the bunch). Given that the bunch has a non-zero bunch length, the current time $t(s)$ associated with the particles will be different for different particles (See Eq. (14.28)). If it is desired to reconstruct the shape of the bunch at *constant time*, each particle must be tracked either forward or backwards by an appropriate amount. Since this tracking generally involves only very short distances, it is usually acceptable to ignore any fields and to propagate the particles as if they were in a field free region.

32.17 Spin Tracking

See Section §5.3 for a list of spin tracking methods available. To turn spin tracking on, use the `bmad_com%spin_tracking_on` flag. `ele%spin_tracking_method` sets the method used for spin tracking. After properly initializing the spin in the `coord_struct`, calls to `track1` will track both the particle orbit and the spin.

The Sokolov-Ternov effect[Barber99] is the self-polarization of charged particle beams due to asymmetric flipping of a particle's spin when the particle is bent in a magnetic field. Whether this effect is included in a simulation is determined by the setting of `bmad_com%spin_sokolov_ternov_flipping_on`. Also, spin flipping will *not* be done if spin tracking is off or both radiation damping and excitation are off.

32.18 X-ray Targeting

X-rays can have a wide spread of trajectories resulting in many "doomed" photons that hit apertures or miss the detector with only a small fraction of "successful" photons actually contributing to the simulation results. The tracking of doomed photons can therefore result in an appreciable lengthening of the simulation time. To get around this, *Bmad* can be setup to use what is called "targeting" to minimize the number of doomed photons generated.

This is explained in detail in §23.5. The coordinates of the four or eight corner points and the center target point are stored in:

```
gen_ele%photon%target%corner(:)%r(1:3)
gen_ele%photon%target%center%r(1:3)
```

where `gen_ele` is the generating element (not the element with the aperture).

Chapter 33

Miscellaneous Programming

33.1 Custom and Hook Routines

Bmad calculations, like particle tracking through a lattice element, can be customized using what are called "`custom`" and "`hook`" routines. The general idea is that a programmer can implement custom code which is linked into a program and this custom code will be called at the appropriate time by *Bmad*. Thus, for example, custom code can be created for Runge-Kutta tracking that calculates the electromagnetic field of some complicated electromagnet.

To enable *Bmad* to be able to call customized code, there are a number of "`entry` points" defined in the *Bmad* code. At each entry point, a "`dummy`" version of a custom and hook routine is called. For `hook` routines, this dummy version does nothing except to keep the linker happy when customized hook routine is not implemented by a program. For `custom` routines, the dummy version will issue an error message since it should not have been called. That is, the difference between `custom` and `hook` routines is that the `hook` routine is always called at the appropriate time without regard to the type of lattice element under consideration or what tracking method is being used. See below for more details.

Note: Custom and Hook entry points are added to *Bmad* on an as-needed basis. If you have a need that is not met by the existing set of entry points, please contact a *Bmad* maintainer.

Customization is done by copying the appropriate dummy routine to the same area where code files for the program to be customized are, modifying the dummy routine to do what you want it to do, and then relinking the program.

Important:

- Do not put the non-dummy version of a custom routine into a library that is linked to the program. This will generally result in the non-dummy version *not* being linked to.
- Do not modify directly any dummy routine in the `bmad` directory. This is unnecessary and it complicates updating your local copy of the *Bmad* code as the *Bmad* code is developed over time.

While coding a custom routine, it is important to remember that it is *not* permissible to modify any routine argument that does not appear in the list of output arguments shown in the comment section at the top of the file.

Note: Remember to modify the appropriate compile script file (typically named something like "cmake.XXX") so that any routines that you have added to your program area are compiled and linked.

33.2 Custom Calculations

There are essentially two ways to do `custom` (as opposed to `hook`) calculations. One way involves using a `custom` element (§3.10). The other way involves setting the appropriate `method` component of an element to `custom`. An appropriate method component is one of

`tracking_method`	§5.1
`mat6_calc_method`	§5.2
`field_calc`	§5.4
`aperture_type`	§4.8

There are eight routines that implement custom calculations:

```
check_aperture_limit_custom
em_field_custom
init_custom
make_mat6_custom
radiation_integrals_custom
track1_custom
track1_spin_custom
wall_hit_handler_custom
```

[Use `getf` for more details about the argument lists for these routines.]

The dummy versions of these custom routines, if called, will print an error message and stop the program. The exception is the dummy `init_custom` routine which will simply do nothing when called.

The init_custom routine is called by bmad_parser at the end of parsing for any lattice element that is a `custom` element or has set any one of the element components as listed above to `custom`. The `init_custom` routine can be used to initialize the internals of the element. For example, consider a `custom` element defined in a lattice file by

```
my_element: custom, val1 = 1.37, descrip = "field.dat", mat6_calc_method = tracking
```

In this example, the `descrip` (§4.3) component is used to specify the name of a file that contains parameters for this element. When `init_custom` is called for this element (see below), the file can be read and the parameters stored in the element structure. Besides the `ele%value` array, parameters may be stored in the general use components given in §27.19.

The make_mat6_custom routine is called by the track1 routine when calculating the transfer matrix through an element.

The track1_custom routine is called by the track1 routine when the `tracking_method` for the element is set to `custom`. Further customization can be set by the routines track1_preprocess and track1_-postprocess. See Section §33.3 for more details.

A potential problem with `track1_custom` is that the calling routine, that is `track1`, does some work like checking aperture, etc. (see the `track1` code for more details). If this is not desired, the `track1_preprocess` routine (§33.3) can be used to do custom tracking and to make sure that `track1` does not do any extra calculations. This is accomplished by putting the custom tracking code in `track1_preprocess` and by setting the `finished` argument of `track1_preprocess` to True.

The check_aperture_limit_custom routine is used to check if a particle has hit an aperture in tracking. It is called by the standard *Bmad* routine check_aperture_limit when `ele%aperture_type` is set to `custom$`. A `custom` element has the standard limit attributes (§4.8) so a `custom` element does not have to implement custom aperture checking code.

The em_field_custom routine is called by the electro-magnetic field calculating routine em_field_calc when `ele%field_calc` is set to `custom$`. As an alternative to `em_field_custom`, a `custom` element can use a field map (§4.15) to characterize the element's electromagnetic fields.

Note: When tracking through a `patch` element, the first step is to transform the particle's coordinates from the entrance frame to the exit frame. This is done since it simplifies the tracking. [The criterion for stopping the propagation of a particle through a `patch` is that the particle has reached the exit face and the calculation to determine if a particle has reached the exit face is simplified if the particle's coordinates are expressed in the coordinate frame of the exit face.] Thus for `patch` elements, unlike all other elements, the particle coordinates passed to em_field_custom are the coordinates with respect to the exit coordinate frame and not the entrance coordinate frame. If field must be calculated in the entrance coordinate frame, a transformation between entrance and exit frames must be done:

```
subroutine em_field_custom (ele, param, s_rel, time, orb, &
                                  local_ref_frame, field, calc_dfield, err_flag)
use lat_geometry_mod
...
real(rp) w_mat(3,3), w_mat_inv(3,3), r_vec(3), r0_vec(3)
real(rp), pointer :: v(:)
...
! Convert particle coordinates from exit to entrance frame.
v => ele%value   ! v helps makes code compact
call floor_angles_to_w_mat (v(x_pitch$), v(y_pitch$), v(tilt$), w_mat, w_mat_inv)
r0_vec = [v(x_offset$), v(y_offset$), v(z_offset$)]
r_vec = [orb%vec(1), orb%vec(3), s_rel]  ! coords in exit frame
r_vec = matmul(w_mat, r_vec) + r0_vec      ! coords in entrance frame

! Calculate field and possibly field derivative
...

! Convert field from entrance to exit frame
field%E = matmul(w_mat_inv, field%E)
field%B = matmul(w_mat_inv, field%B)
if (logic_option(.false., calc_dfield)) then
  field%dE = matmul(w_mat_inv, matmul(field%dE, w_mat))
  field%dB = matmul(w_mat_inv, matmul(field%dB, w_mat))
endif
```

The wall_hit_handler_custom routine is called when the Runge-Kutta tracking code odeint_bmad detects that a particle has hit a wall (§4.11). [This is separate from hitting an aperture that is only defined at the beginning or end of an lattice element.] The dummy `wall_hit_handler_custom` routine does nothing. To keep tracking, the particle must be marked as alive

```
subroutine wall_hit_handler_custom (orb, ele, s, t)
  ...
  orb%state = alive$   ! To keep on truckin'
  ...
```

Note: `odeint_bmad` normally does not check for wall collisions. To change the default behavior, the `runge_kutta_com` common block must modified. This structure is defined in `runge_kutta_mod.f90`:

```
type runge_kutta_common_struct
  logical :: check_wall_aperture = .false.
  integer :: hit_when = outside_wall$   ! or wall_transition$
end type

type (runge_kutta_common_struct), save :: runge_kutta_com
```

To check for wall collisions, the `%check_wall_aperture` component must be set to true. The `%hit_when` components determines what constitutes a collision. If this is set to `outside_wall$` (the default),

then any particle that is outside the wall is considered to have hit the wall. If `%hit_when` is set to `wall_transition$`, a collision occurs when the particle crosses the wall boundary. The distinction between `outside_wall$` and `wall_transition$` is important if particles are to be allowed to travel outside the wall.

33.3 Hook Routines

A `hook` routine is like a `custom` routine in that a `hook` routine can be used for customizing a *Bmad* calculation by replacing the `dummy` version of a `hook` routine with customized code. The difference is that the `hook` routine is always called at the appropriate time without regard to the type of lattice element under consideration or what tracking method is being used. There are three `hook` routines that are available:

```
apply_element_edge_kick_hook
ele_geometry_hook
ele_to_fibre_hook
time_runge_kutta_periodic_kick_hook
track1_bunch_hook
track1_preprocess
track1_postprocess
track1_wake_hook
```

The `apply_element_edge_kick_hook` routine can be used for custom tracking through a fringe field. See the documentation in the file `apply_element_edge_kick_hook.f90` for more details.

The `ele_geometry_hook` routine can be used for custom calculations of the global geometry of an element. This is useful, for example, for a support table on a kinematic mount since *Bmad* does not have the knowledge to calculate the table orientation from the position of the mount points. See the documentation in the file `ele_geometry_hook.f90` for more details.

The ele_to_fibre_hook routine can be used to customize how the PTC fibre corresponding to a *Bmad* lattice element is constructed.

The `time_runge_kutta_periodic_kick_hook` routine can be used to introduce a time dependent kick when doing tracking with `time_runge_kutta`. This routine could be used, for example, to add the kick due to a passing beam ! on a residual gas ion that is being tracked. See the documentation in the file `time_runge_kutta_periodic_kick_hook.f90` for more details.

The `track1_bunch_hook` routine can be used for custom bunch tracking through an element.

The `track1_preprocess` and `track1_postprocess` routines are called by the track1 routine. [Additionally, if the element being tracked through has its tracking method set to `custom`, the `track1_custom` routine is called.] The `track1_preprocess` and `track1_postprocess` routines are useful for a number of things. For example, if the effect of an electron cloud is to be modeled, these two routines can be used to put in half the electron cloud kick at the beginning of an element and half the kick at the end.

The routine `track1_preprocess` has an additional feature in that it has an argument `radiation_included` that can be set to `True` if the routine `track1_custom` will be called and `track1_custom` will be handling radiation damping and excitation effects.

The `track1_wake_hook` can be used to apply custom wakes.

33.4 Physical and Mathematical Constants

Common physical and mathematical constants that can be used in any expression are defined in the file:

```
sim_utils/interfaces/physical_constants.f90
```

The following constants are defined

```
pi = 3.14159265358979d0
twopi = 2 * pi
fourpi = 4 * pi
sqrt_2 = 1.41421356237310d0
sqrt_3 = 1.73205080757d0
complex: i_imaginary = (0.0d0, 1.0d0)

e_mass = 0.51099906d-3   ! DO NOT USE! In GeV
p_mass   = 0.938271998d0   ! DO NOT USE! In GeV

m_electron = 0.51099906d6  ! Mass in eV
m_proton   = 0.938271998d9 ! Mass in eV

c_light = 2.99792458d8             ! speed of light
r_e = 2.8179380d-15                ! classical electron radius
r_p = r_e * m_electron / m_proton  ! proton radius
e_charge = 1.6021892d-19           ! electron charge

h_planck = 4.13566733d-15          ! eV*sec Planck's constant
h_bar_planck = 6.58211899d-16      ! eV*sec h_planck/twopi

mu_0_vac = fourpi * 1e-7                   ! Permeability of free space
eps_0_vac = 1 / (c_light**2 * mu_0_vac)    ! Permittivity of free space

classical_radius_factor = r_e * m_electron ! Radiation constant

g_factor_electron = 0.001159652193    ! Anomalous gyro-magnetic moment
g_factor_proton   = 1.79285           ! Anomalous gyro-magnetic moment
```

33.5 Global Coordinates and S-positions

The routine lat_geometry will compute the global floor coordinates at the end of every element in a lattice. `lat_geometry` works by repeated calls to ele_geometry which takes the floor coordinates at the end of one element and calculates the coordinates at the end of the next. For conversion between orientation matrix $\mathbf{W}$ (§14.2) and the orientation angles θ, ϕ, ψ, the routines floor_angles_to_w_mat and floor_w_mat_to_angles can be used.

The routine s_calc calculates the longitudinal s positions for the elements in a lattice.

33.6 Reference Energy and Time

The reference energy and time for the elements in a lattice is calculated by lat_compute_ref_energy_and_time. The reference energy associated with a lattice element is stored in

```
ele%value(E_tot_start$)   ! Total energy at upstream end of element (eV)
ele%value(p0c_start$)     ! Momentum * c_light at upstream end of element (eV)
ele%value(E_tot$)         ! Total energy at downstream end (eV)
ele%value(p0c$)           ! Momentum * c_light at downstream end(eV)
```

Generally, the reference energy is constant throughout an element so that `%value(E_tot_start$` = `%value(E_tot$` and `%value(p0c_start$` = `%value(p0c$`. Exceptions are elements of type:

```
custom,
em_field,
hybrid, or
lcavity
```

In any case, the starting `%value(E_tot_start$` and `%value(p0c_start$` values of a given element will be the same as the ending `%value(E_tot$` and `%value(p0c$` energies of the previous element in the lattice.

The reference time and reference transit time is stored in

```
ele%ref_time                ! Ref time at downstream end
ele%value(delta_ref_time$)
```

The reference orbit for computing the reference energy and time is stored in

```
ele%time_ref_orb_in       ! Reference orbit at upstream end
ele%time_ref_orb_out      ! Reference orbit at downstream end
```

Generally `ele%time_ref_orb_in` is the zero orbit. The exception comes when an element is a `super_slave`. In this case, the reference orbit through the super_slaves of a given `super_lord` is constructed to be continuous. This is done for consistency sake. For example, to ensure that when a marker is superimposed on top of a wiggler the reference orbit, and hence the reference time, is not altered.

`group` (§3.21), `overlay` (§3.36), and `super_lord` elements inherit the reference from the last slave in their slave list (§28.5). For `super_lord` elements this corresponds to inheriting the reference energy of the slave at the downstream end of the `super_lord`. For `group` and `overlay` elements a reference energy only makes sense if all the elements under control have the same reference energy.

Additionally, photonic elements like `crystal`, `capillary`, `mirror` and `multilayer_mirror` elements have an associated photon reference wavelength

```
ele%value(ref_wavelength$)      ! Meters.
```

33.7 Global Common Structures

There are two common variables used by Bmad for communication between routines. These are `bmad_com`, which is a `bmad_common_struct` structure, and `global_com` which is a `global_common_struct` structure. The `bmad_com` structure is documented in Section §9.3.

The `global_common_struct` is meant to hold common parameters that should not be modified by the user.

```
type global_common_struct
  logical be_thread_safe = F    ! Avoid thread unsafe practices?
  logical exit_on_error  = T    ! Exit program on error?
end type
```

A global variable `global_com` is defined in the `sim_utils` library:

```
type (global_common_struct), save :: global_com
```

And various routines use the settings in `global_com`.

`%be_thread_safe`
: Toggle to prevent non thread safe calculational optimizations from being done. See Sec. §33.8 for more details.

`%exit_on_error`
: The `%exit_on_error` component tell a routine if it is OK to stop a program on a severe error. Stopping is generally the right thing when a program is simply doing a calculation and getting a wrong answer is not productive. In control system programs and in interactive programs like `Tao`, it is generally better not to stop on an error.

33.8 Parallel Processing

Bmad was initially developed without regard to parallel processing. When a demand for multithreading capability arose, *Bmad* was modified to meet the need. In order to retain some of the speedups that can be achieved with saved variables, a switch was introduced in the `global_common_struct` (§33.7) called `%be_thread_safe`. The default is False. This should be set True in a a multithreaded program:

```
global_com%be_thread_safe = .true.
```

One rule to follow in multithreaded programs: `lat_struct` instances must be local.

Another rule is that PTC code (§34) is not thread safe.

Chapter 34

PTC/FPP

The PTC/FPP library of Étienne Forest handles Taylor maps to any arbitrary order. this is also known as Truncated Power Series Algebra (TPSA). The core Differential Algebra (DA) package used by FPP was developed by Martin Berz[Berz89].

FPP
: The "Fully Polymorphic Package" (`FPP`) library implements Differential Algebra (DA) for the manipulation of Taylor maps. `FPP` is purely mathematical in nature. It has no knowledge of accelerators, magnetic fields, particle tracking etc.

PTC
: The Polymorphic Tracking Code `PTC` library is for accelerator simulation. It uses `FPP` as a back end for calculating such things as one turn maps.

PTC is used by *Bmad* when constructing Taylor maps and when the `tracking_method` §5.1) is set to `symp_lie_ptc`.

For more information see the FPP/PTC manual[Forest02].

34.1 Phase Space

PTC uses different longitudinal phase space coordinates compared to *Bmad*. *Bmad*'s phase space coordinates are (§14.4.2)

$$(x, p_x, y, p_y, z, p_z) \tag{34.1}$$

In PTC one can choose between several different coordinate systems. The one that Bmad uses is

$$(x, p_x, y, p_y, p_t, c\Delta t) \tag{34.2}$$

where

$$p_t = \frac{\Delta E}{P_0} \tag{34.3}$$

This choice of phase space is set in set_ptc. Specifically, the PTC global variable `DEFAULT`, which is of type `internal_states`, has the `%time` switch set to `True`.

vec_bmad_to_ptc and vec_ptc_to_bmad are conversion routines that translate between the two. Actually there are a number of conversion routines that translate between *Bmad* and PTC structures. See §39.31 for more details.

34.2 PTC Initialization

One important parameter in PTC is the order of the Taylor maps. By default *Bmad* will set this to 3. The order can be set within a lattice file using the parameter[taylor_order] attribute. In a program the order can be set using set_ptc. In fact set_ptc must be called by a program before PTC can be used. bmad_parser will do this when reading in a lattice file. That is, if a program does not use bmad_parser then to use PTC it must call set_ptc. Note that resetting PTC to a different order reinitializes PTC's internal memory so one must be careful if one wants to change the order in mid program.

34.3 PTC Structures Compared to Bmad's

Bmad uses a lat_struct structure to hold the information on a machine and a lat_struct has an array of branch_structs (the %branch(:) component) with each branch_struct holding an array of ele_structs (the %ele(:) component). The ele_struct holds the information on the individual elements. An ele_struct holds information about both the physical element and the reference orbit through it.

PTC has a somewhat different philosophy as illustrated in Fig. 34.1. A PTC mad_universe structure is very roughly equivalent to a *Bmad* lat_struct. That is, both structures can contain the description for an entire accelerator complex. Note that it is standard in PTC to use two mad_universe structures called m_u and m_t. These two are defined globally. The difference between m_u and m_t is that m_u is used as a bookkeeping device for convenient accessing of all lattice elements. On the other hand, m_t contains the layouts that can be used for tracking.

equivalent to a *Bmad* branch_struct. A layout has a pointer to a linked list of fibre structures. Each fibre has a pointer to a magnet structure which holds the information about the physical element and each fibre holds information about the reference orbit through the element.

With PTC, The top level structure mad_universe has two components called %first and %last which are pointers to the ends of an array of layout_array structures. Each layout_array holds a layout structure. A layout structure has pointers to the previous and next layouts making a linked list of layouts indicated by the horizontal arrows. Each layout has pointers to a linked list of fibre structures. The fibre structures represent the reference trajectory through an element. Each fibre structure has a pointer to a element and an elementp structures which represent the physical element. With *Bmad*, the lat_struct roughly corresponds to the PTC layout_array(:), the branch_struct roughly corresponds to the PTC layout and the element_struct roughly corresponds to the PTC fibre, element and elementp structures.

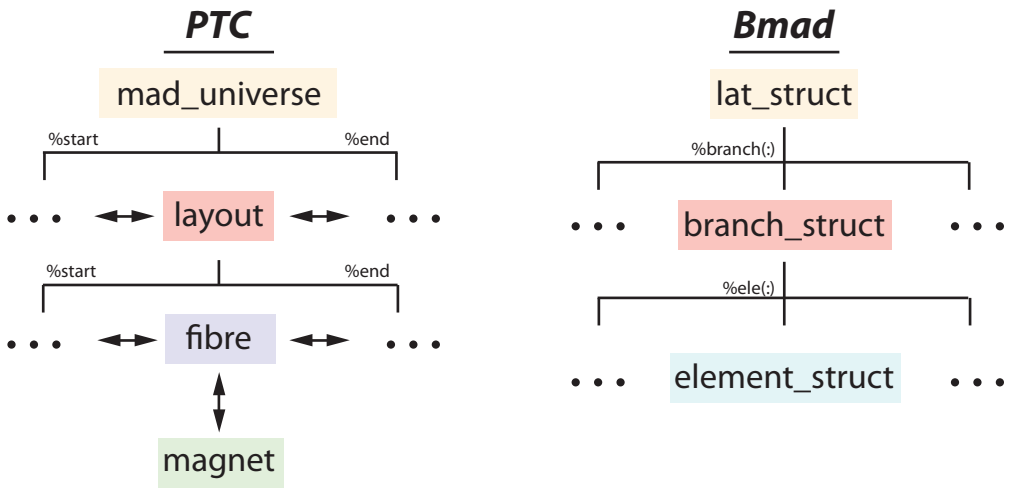


Figure 34.1: Simplified diagram showing the organization of the major PTC structures involved in defining a lattice contrasted with *Bmad*.

34.4 Variable Initialization and Finalization

PTC variables must be initialized and finalized. This is done with the`alloc()` and `kill()` routines. In addition, the `real_8_init` routine can initialize a `real_8` array:

```
type (real_8) y8(6)
...
call real_8_init (y8)
call kill (y8)
```

34.5 Correspondence Between Bmad Elements and PTC Fibres

.

When a PTC `layout` is created from a *Bmad* `lat_struct` instance using the routine lat_to_ptc_layout, the correspondence between the *Bmad* elements and the PTC fibres is maintained through the `ele%ptc_fibre` pointer. The following rules apply:

1. There will be marker `fibres` at the beginning and end of the `layout`. The beginning `fibre` will correspond to `branch%ele(0)`. The end `fibre` will not have a corresponding *Bmad* element.

2. Generally there will be a one-to-one correspondence between `fibres` and `branch%ele` elements. The exception is where a "hard edge" model is used for tracking. In this case, there will be three `fibres` for the *Bmad* element: Two drift `fibres` with a `fibre` of the appropriate type in between. In this case, `ele%ptc_fibre` will point to the last (drift) `fibre`.

Remember: The attributes like reference energy, etc. for a *Bmad* `ele_struct` instance are referenced to the exit end of the element. For PTC the reference edge for a `fibre` is the entrance end.

34.6 Taylor Maps

FPP stores its `real_8` Taylor maps in such a way that it is not easy to access them directly to look at the particular terms. To simplify life, Étienne has implemented the `universal_taylor`structure:

```
type universal_taylor
  integer, pointer  :: n       ! Number of coefficients
  integer, pointer  :: nv      ! Number of variables
  real(dp), pointer :: c(:)    ! Coefficients C(N)
  integer, pointer  :: j(:,:)  ! Exponents of each coefficients J(N,NV)
end type
```

Bmad always sets `nv` = 6. *Bmad* overloads the equal sign to call routines to convert between Étienne's `real_8` Taylor maps and `universal_taylor`:

```
type (real_8) tlr(6)           ! Taylor map
type (universal_taylor) ut(6)  ! Taylor map
...
tlr = ut                       ! Convert universal_taylor -> real_8
ut = tlr                       ! Convert real_8 -> universal_taylor
```

34.7 Patches

There is a significant difference between how patches are treated in PTC and *Bmad*. In PTC, a patch is just though of as a coordinate transformation for propagating a particle from one `fibre` to the next. As such, the `patch` is part of a `fibre`. That is, any `fibre` representing tracking through quadrupoles, bends, etc. will have patches for the entrance and exit ends of the `fibre`.

With *Bmad*, on the other hand, a `patch` is a "first class" element on par with all other elements be they quadrupoles, bends, etc. When translating a `patch` from *Bmad* to PTC, the `patch` is represented in PTC as a `marker` element with a patch at the exit end.

34.8 Number of Integration Steps & Integration Order

"Drift like" elements in PTC will use, by default, only one integration step. *Bmad* uses the default when translating from *Bmad* lattice elements to PTC fibres. The *Bmad* lattice elements that are drift like are:

```
drift
ecollimator
instrument
monitor
pipe
rcollimator
```

When tracking, there is a trade-off between step size and integrator order. Higher order means fewer steps are needed to get the same accuracy. But one higher order step is computationally more intensive then one lower order step so what is the optimum order and number of steps is dependent upon various factors like magnet strength and how fast the field is varying. Generally, when the field is varying, such as in a wiggler, lower order and more steps are favored. Also spin tracking is always 2nd order in PTC. So going to higher order for the orbital tracking with less steps will cause the spin tracking to be less accurate.

The way PTC "resplitting" routines work is that, for a given element, they start by assuming that the tracking will be done using a 2^{nd} order integrator, They then compute the number of steps needed based upon the electric and magnetic field strengths. This number is compared to a crossover limit point here named C_1. If the number of steps is less than or equal to C_1 then the resplitting routine stops and tracking will thereafter be done with a 2^{nd} order integrator with the calculated number of steps. On the other hand, if the number of steps is greater than C_1, the resplitting routine will redo the calculation assuming 4^{th} order integration. With 4^{th} order integration, the number of calculated steps will compared to a different crossover limit point here called C_2. Again, if the number of steps is less than or equal to C_2, the routine will assign 4^{th} order tracking to the element. Otherwise, the routine will assign 6^{th} order tracking to the element with an appropriate number of steps.

The default crossover limit points are

$[C_1, C_2] = [30, 60]$ For wiggler type elements.
$[C_1, C_2] = [4, 18]$ For all other elements.

The greater number for wigglers is a reflection of the fact that the wiggler field is not constant.

34.9 Internal_State

The `internal_state` structure looks like:

```
type internal_state
   integer totalpath      ! total time or path length is used
   logical(lp) time       ! Time is used instead of path length
   logical(lp) radiation  ! Radiation damping (but not excitation) is turned on
   logical(lp) nocavity   ! Cavity is turned into a drift
   logical(lp) fringe     ! fringe fields are turned on (mainly for quadrupoles)
   logical(lp) stochastic ! Random Stochastic kicks to x(5)
   logical(lp) envelope   ! Stochastic envelope terms tracked in probe_8
   logical(lp) para_in    ! If true, parameters in the map are included
   logical(lp) only_4d    ! REAL_8 Taylor in (x,p_x,y,p_y)
   logical(lp) delta      ! REAL_8 Taylor in (x,p_x,y,p_y,delta)
   logical(lp) spin       ! Spin is tracked
   logical(lp) modulation ! One modulated family tracked by probe
   logical(lp) only_2d    ! REAL_8 Taylor in (x,p_x)
   logical(lp) full_way   !
end type internal_state
```

Chapter 35

OPAL

OPAL (Object Oriented Parallel Accelerator Library) is a tool for charged-particle optic calculations in large accelerator structures and beam lines including 3D space charge. OPAL is built from first principles as a parallel application, OPAL admits simulations of any scale: on the laptop and up to the largest High Performance Computing (HPC) clusters available today. Simulations, in particular HPC simulations, form the third pillar of science, complementing theory and experiment.

OPAL includes various beam line element descriptions and methods for single particle optics, namely maps up to arbitrary order, symplectic integration schemes and lastly time integration. OPAL is based on IPPL (Independent Parallel Particle Layer) which adds parallel capabilities. Main functions inherited from IPPL are: structured rectangular grids, fields and parallel FFT and particles with the respective interpolation operators. Other features are, expression templates and massive parallelism (up to 8000 processors) which makes is possible to tackle the largest problems in the field.

The manual can be obtained at

`amas.web.psi.ch/docs/opal/`

35.1 Phase Space

OPAL uses different longitudinal phase space coordinates compared to *Bmad*. *Bmad*'s phase space coordinates are

$$(x, p_x/p_0, y, p_y/p0, -\beta c(t - t_0), (p - p_0)/p_0) \tag{35.1}$$

OPAL uses

$$(x, \gamma\beta_x, y, \gamma\beta_y, z, \gamma\beta_z) \tag{35.2}$$

`convert_particle_coordinates_s_to_t` and `convert_particle_coordinates_s_to_t` are conversion routines ...

Chapter 36

C++ Interface

To ease the task of using *C++* routines with *Bmad*, there is a library called `cpp_bmad_interface` which implements a set of *C++* classes in one–to–one correspondence with the major *Bmad* structures. In addition to the *C++* classes, the *Bmad* library defines a set of conversion routines to transfer data values between the *Bmad* Fortran structures and the corresponding *C++* classes.

The list of all classes is given in the file

```
cpp_bmad_interface/include/cpp_bmad_classes.h
```

The general rule is that the equivalent class to a *Bmad* structure named `xxx_struct` will be named `CPP_xxx`. Additionally, for each *Bmad* structure, there is a opaque class named `Bmad_xxx_class` for use in the translation code discussed below. The names of these opaque classes have the form `Bmad_xxx_class` and are used to define pointer instances in routine argument lists.

36.1 C++ Classes and Enums

Generally, The *C++* classes have been set up to simply mirror the corresponding *Bmad* structures. For example, the `CPP_lat` class has a string component named `.version` that mirrors the `%version` component of the `lat_struct` structure. There are some exceptions. For example, structure components that are part of `PTC` (§1.5) are not present in the classes.

While generally the same component name is used for both the *Bmad* structures and the *C++* classes, in the case where there is a *C++* reserved word conflict, the *C++* component name will be different.

A header file `bmad_enums.h` defines corresponding *Bmad* parameters for all *C++* routine. The *Bmad* parameters are in a namespace called `Bmad`. The convention is that the name of a corresponding *C ++* parameter is obtained by dropping the ending `$` (if there is one) and converting to uppercase. For example, `electron$` on the Fortran side converts to `Bmad::ELECTRON` in *C++*.

All of the *C++* class components that are arrays or matrices are zero based so that, for example, the index of the `.vec[i]` array in a `CPP_coord` runs from 0 through 5 and not 1 through 6 as on the Fortran side. Notice that for a `lat_struct` the `%ele(0:)` component has a starting index of zero so there is no off–by–one problem here. The exception to this rule is the `%value(:)` array of the `ele_struct` which has a span from 1 to `num_ele_attrib$`. In this case, To keep the conversion of the of constructs like `ele%value(k1$)` consistant, the corresponding `ele.value[]` array has goes from 0 to `Bmad::NUM_ELE_ATTRIB` with the 0th element being unused.

```
1     subroutine f_test
2       use bmad_cpp_convert_mod
3       implicit none
4
5       interface
6         subroutine cpp_routine (f_lat, c_coord) bind(c)
7           import f_lat, c_ptr
8           type (lat_struct) :: f_lat
9           type (c_ptr), value :: c_coord
10        end subroutine
11      end interface
12
13      type (lat_struct), target :: lattice   // lattice on Fortran side
14      type (coord_struct), target :: orbit
15      type (c_ptr), value :: c_lat
16      ! ...
17      call lat_to_c (c_loc(lattice), c_lat)    ! Fortran side convert
18      call cpp_routine (c_lat, c_loc(orbit))   ! Call C++ routine
19      call lat_to_f (c_lat, c_loc(lattice))    ! And convert back
20    end subroutine
```

Figure 36.1: Example Fortran routine calling a *C++* routine.

```
1     #include "cpp_bmad_classes.h"
2
3     using namespace Bmad;
4
5     extern "C" cpp_routine (CPP_lat& c_lat, Bmad_coord_class* f_coord,  f_lat) {
6       CPP_coord c_coord;
7       coord_to_c (f_coord, c_coord);        // C++ side convert
8       // ... do calculations ...
9       cout << c_lat.name << "  " << c_lat.ele[1].value[K1] << endl;
10      coord_to_f (c_coord, f_coord);        // And convert back
11    }
```

Figure 36.2: Example *C++* routine callable from a Fortran routine.

36.2 Conversion Between Fortran and C++

A simple example of a Fortran routine calling a *C++* routine is shown in Figs. 36.1 and 36.2. Conversion between structure and classes can happen on either the Fortran side or the *C++* side. In this example, the `lat_struct` / `CPP_lat` conversion is on the Fortran side and the `coord_struct` / `CPP_coord` is on the *C++* side.

On the Fortran side, the interface block defines the argument list of the *C++* routine being called.

On the *C++* side, `f_coord` is an instance of the `Bmad_coord_class` opaque class.

A *C++* routine calling a Fortran routine has a similar structure to the above example. The interface block in Fig. 36.1 can be used as a prototype. For additional examples of conversion between Fortran and *C++*, look at the test code in the directory

```
cpp_bmad_interface/interface_test
```

Chapter 37

Quick_Plot Plotting

The plotting package included in the *Bmad* distribution is PGPLOT (see §25.2). One drawback of PGPLOT is that the arguments to PGPLOT's subroutines are not always conveniently structured. to remedy this a suite of wrapper routines have been developed which can be used to drive PGPLOT. This suite is called *Quick Plot* and lives in the `sim_utils` library which comes with the Bmad distribution. A quick reference guide can be seen online by using the command `getf quick_plot`. For quick identification in a program, all *Quick Plot* subroutines start with a `qp_` prefix. Also, by convention, all PGPLOT subroutines start with a `pg` prefix.

While *Quick Plot* covers most of the features of PGPLOT, *Quick Plot* is still a work in progress. For example, contour plots have not yet been implemented in *Quick Plot*. If you see a feature that is lacking in *Quick Plot* please do not hesitate to make a request to `dcs16@cornell.edu`.

Note: PGPLOT uses single precision real(4) numbers while *Quick Plot* uses real(rp) numbers. If you use any PGPLOT subroutines directly be careful of this.

```
1   program example_plot
2     use quick_plot
3     integer id
4     character(1) ans
5
6     ! Generate PS and X-windows plots.
7     call qp_open_page ("PS-L")  ! Tell \quickplot to generate a PS file.
8     call plot_it              ! Generate the plot
9     call qp_close_page        ! quick_plot.ps is the file name
10    call qp_open_page ("X", id, 600.0_rp, 470.0_rp, "POINTS")
11    call plot_it
12    write (*, "(a)", advance = "NO") " Hit any class to end program: "
13    accept "(a)", ans
14
15  !-----------------------------------------------------------------------
16  contains
17  subroutine plot_it                            ! This generates the plot
18    real(rp), allocatable :: x(:), y(:), z(:), t(:)
19    real(rp) x_axis_min, x_axis_max, y_axis_min, y_axis_max
20    integer x_places, x_divisions, y_places, y_divisions
21    character(80) title
22    logical err_flag
23    namelist / parameters / title
24
25    ! Read in the data
26    open (1, file = "plot.dat", status = "old")
27    read (1, nml = parameters)                  ! read in the parameters.
28    call qp_read_data (1, err_flag, x, 1, y, 3, z, 4, t, 5) ! read in the data.
29    close (1)
30
31    ! Setup the margins and page border and draw the title
32    call qp_set_page_border (0.01_rp, 0.02_rp, 0.2_rp, 0.2_rp, "%PAGE")
33    call qp_set_margin (0.07_rp, 0.05_rp, 0.05_rp, 0.05_rp, "%PAGE")
34    call qp_draw_text (title, 0.5_rp, 0.85_rp, "%PAGE", "CT")
35
36    ! draw the left graph
37    call qp_set_box (1, 1, 2, 1)
38    call qp_calc_and_set_axis ("X", minval(x), maxval(x), 4, 8, "ZERO_AT_END")
39    call qp_calc_and_set_axis ("Y", minval(z), maxval(z), 4, 8, "GENERAL")
40    call qp_draw_axes ("X\dlab\u", "\gb(\A)")
41    call qp_draw_data (x, y, symbol_every = 0)
42
43    call qp_save_state (.true.)
44    call qp_set_symbol_attrib (times$, color = blue$, height = 20.0_rp)
45    call qp_set_line_attrib ("PLOT", color = blue$, style = dashed$)
46    call qp_draw_data (x, z, symbol_every = 5)
47    call qp_restore_state
48
49    ! draw the right graph. star5_filled$ is a five pointed star.
50    call qp_save_state (.true.)
51    call qp_set_box (2, 1, 2, 1)
52    call qp_set_graph_attrib (draw_grid = .false.)
53    call qp_set_symbol_attrib (star5_filled$, height = 10.0_rp)
54    call qp_set_axis ("Y", -0.1_rp, 0.1_rp, 4, 2)
55    call qp_set_axis ('Y2', 1.0_rp, 100.0_rp, label = 'Y2 axis', &
56                               draw_numbers = .true., ax_type = 'LOG')
57    call qp_draw_axes ("\m1 \m2 \m3 \m4 \m5 \m6 \m7", "\fsLY\fn", title = "That Darn Graph")
58    call qp_draw_data (x, t, draw_line = .false., symbol_every = 4)
59    call qp_restore_state
60  end subroutine
61  end program
```

Figure 37.1: *Quick Plot* example program.

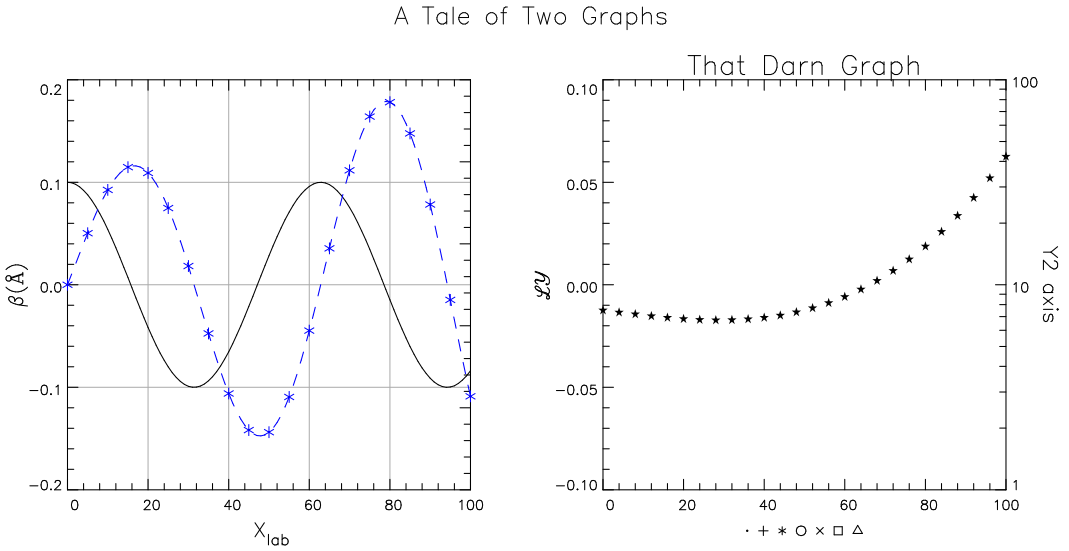


Figure 37.2: Output of plot_example.f90.

37.1 An Example

An example of how *Quick Plot* can be used in a program is shown in Fig. 37.1. In the *Bmad* distribution a copy of this program is in the file

```
sim_utils/plot_example/plot_example.f90
```

The `plot_example.f90` program generates the figure shown in Fig. 37.2 from the input file named `plot.dat`. The first few lines of the data file are

```
&parameters
  title = "A Tale of Two Graphs"
/

Any junk here...

Col1      Col2      Col3      Col4      Col5
   0    0.0000    0.1000    0.0000   -0.0125
   1    0.0001    0.0995    0.0101   -0.0127
   2    0.0004    0.0980    0.0203   -0.0130
   3    0.0009    0.0955    0.0304   -0.0132
   ...
```

The program first creates a PostScript file for printing on lines 7 through 9 and then makes an X–windows plot on lines 10 and 11. The write/accept lines 12 and 13 are to pause the program to prevent the X-window from immediately closing upon termination of the program.

The heart of the plotting is in the subroutine `plot_it` beginning on line 17. The namelist read on line 27 shows how both parameters and data can be stored in the same file so that a plotting program can be automatically told what the appropriate plot labels are. The qp_draw_text call on line 34 draws the title above the two graphs.

The qp_read_data call on line 28 will skip any "header" lines (lines that do not begin with something that looks like a number) in the data file. In this instance qp_read_data will read the first, third forth and fifth data columns and put them into the `x`, `y`, `z`, and `t` arrays.

qp_set_page_border, qp_set_box, and qp_set_margin sets where the graph is going to be placed. qp_set_box(1, 1, 2, 1) on line 37 tells *Quick Plot* to put the first graph in the left box of a 2 box grid. The qp_set_margin on line 33 sets the margins between the box and the graph axes.

qp_calc_and_set_axis on lines 38 and 39 are used to scale the axes. "ZERO_AT_END" ensures that the x–axis starts (or stops) at zero. qp_calc_and_set_axis is told to restrict the number of major divisions to be between 4 and 8. For the horizontal axis, as can be seen in Fig. 37.2, it chooses 5 divisions.

After drawing the first data curve (the solid curve) in the left graph, the routines qp_set_symbol_attrib and qp_set_line_attrib are called on lines 44 and 45 to plot the next data curve in blue with a dashed line style. By default, this curve goes where the last one did: in the left graph. To keep the setting of the line and symbol attributes from affecting other plots the routines qp_save_state and qp_restore_state on lines 43 and 47 are used. qp_save_state saves the current attributes in an attribute stack. qp_restore_state restores the saved attributes from the attribute stack. qp_draw_axes is called on line 40 to draw the x and y-axes along, and qp_draw_data is called on lines 41 and 46 to draw the two data curves.

Lines 50 through 60 draw the third curve in the right hand graph. The qp_set_axis call on lines 55/56 sets a log scale for the y2 (right hand) axis. The syntax of the string arguments of qp_draw_axes in lines 40 and 57/58 comes from PGPLOT and allows special symbols along with subscripts and superscripts.

37.2 Plotting Coordinates

Quick Plot uses the following concepts as shown in Fig. 37.3

```
PAGE  -- The entire drawing surface.
BOX   -- The area of the page that a graph is placed into.
GRAPH -- The actual plotting area within the bounds of the axes.
```

In case you need to refer to the PGPLOT routines the correspondence between this and PGPLOT is:

```
QUICK_PLOT    PGPLOT
----------    ------
PAGE          VIEW SURFACE
BOX           No corresponding entity.
```

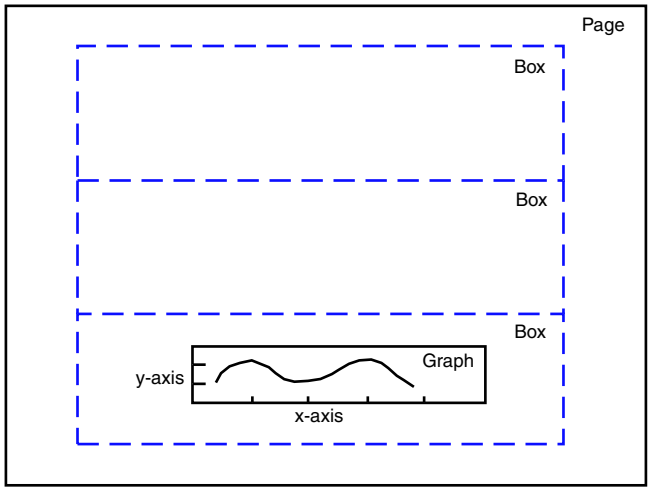


Figure 37.3: A Graph within a Box within a Page.

```
GRAPH          VIEWPORT and WINDOW
```

Essentially the VIEWPORT is the region outside of which lines and symbols will be clipped (if clipping is turned on) and the WINDOW defines the plot area. I'm not sure why PGPLOT makes a distinction, but VIEWPORT and WINDOW are always the same region.

qp_open_page determines the size of the `page` if it is settable (like for X–windows). The page is divided up into a grid of boxes. For example, in Fig. 37.3, the grid is 1 box wide by 3 boxes tall. The border between the grid of boxes and the edges of the page are set by qp_set_page_border. The box that the graph falls into is set by qp_set_box. The default is to have no margins with 1 box covering the entire page. The qp_set_margin routine sets the distance between the box edges and the axes (See the PGPLOT manual for more details).

37.3 Length and Position Units

Typically there is an optional `units` argument for *Quick Plot* routines that have length and/or position arguments. For example, using `getf` one can see that the arguments for qp_draw_rectangle are

```
Subroutine qp_draw_rectangle (x1, x2, y1, y2, units, color, width, style, clip)
```

The `units` argument is a character string which is divided into three parts. The syntax of the `units` argument is

```
unit_type/ref_object/corner
```

The first part `unit_type` gives the type of units

```
"%"       -- Percent.
"DATA"    -- Data units. (Draw default)
"MM"      -- millimeters.
"INCH"    -- Inches. (Set default)
"POINTS"  -- Printers points. NOT PIXELS. (72 points = 1 inch).
```

Note: For displays with a resolution of 72 pixes / inch, `POINTS` corresponds to pixels but many displays have a higher resolution. The second and third parts give the reference point for a position. The second part specifies the reference object

```
"PAGE"  -- Relative to the page (Set default).
"BOX"   -- Relative to the box.
"GRAPH" -- Relative to the graph (Draw default).
```

The third part gives corner of the reference object that is the reference point

```
"LB"    -- Left Bottom (Set and Draw default).
"LT"    -- Left Top.
"RB"    -- Right Bottom.
"RT"    -- Right Top.
```

Notes:

- The `DATA` unit type, by definition, always uses the lower left corner of the `GRAPH` as a reference point.
- For the `%` `unit_type` the `/` between `unit_type` and `ref_object` can be omitted.
- If the `corner` is specified then the `ref_object` must appear also.
- Everything must be in upper case.
- For some routines (qp_set_margin, etc.) only a relative distance is needed. In this case the `ref_object/corner` part, if present, is ignored.

- The `units` argument is typically an optional argument. If not present the default units will be used. There are actually two defaults: The draw default is used for drawing text, symbols, or whatever. The set default is used for setting margins, and other lengths. Initially the draw default is `DATA/GRAPH/LB` and the set default is `INCH/PAGE/LB`. Use qp_set_parameters to change this.

Examples:

```
"DATA"          -- This is the draw default.
"DATA/GRAPH/LB" -- Same as above.
"DATA/BOX/RT"   -- ILLEGAL: DATA must always go with GRAPH/LB.
"%PAGE/LT"      -- Percentage of page so (0.0, 1.0) = RT of page.
"%BOX"          -- Percentage of box so (1.0, 1.0) = RT of box.
"INCH/PAGE"     -- Inches from LB of page.
```

37.4 Y2 and X2 axes

The top and right axes of a graph are known as `X2` and `Y2` respectively as shown in Fig. 37.3. Normally the `X2` axis mirrors the `X` axis and the `Y2` axis mirrors the `Y` axis in that the tick marks and axis numbering for the `X2` and `Y2` axes are the same as the `X` and `Y` axes respectively. qp_set_axis can be used to disable mirroring. For example:

```
call qp_set_axis ("Y2", mirror = .false.)  ! y2-axis now independent of y.
```

qp_set_axis can also be used to set `Y2` axis parameters (axis minimum, maximum, etc.) and setting the `Y2` or `X2` axis minimum or maximum will, by default, turn off mirroring.

Note that the default is for the `X2` and `Y2` axis numbering not to be shown. To enable or disable axis numbering again use qp_set_axis. For example:

```
call qp_set_axis ("Y2", draw_numbers = .true.)  ! draw y2 axis numbers
```

To plot data using the `X2` or `Y2` scale use the qp_use_axis routine. For example:

```
call qp_save_state (.true.)
call qp_use_axis (y = 'Y2')
! ... Do some data plotting here ...
call qp_restore_state
```

37.5 Text

PGPLOT defines certain escape sequences that can be used in text strings to draw Greek letters, etc. These escape sequences are given in Table 37.2.

PGPLOT defines a text background index:

```
        -1 - Transparent background.
         0 - Erase underlying graphics before drawing text.
  1 to 255 - Opaque with the number specifying the color index.
```

37.6 Styles

Symbolic constants have been defined for *Quick Plot* subroutine arguments that are used to choose various styles. As an example of this is in lines 44 and 45 of Fig. 37.1. The numbers in the following are the PGPLOT equivalents.

The *Quick Plot* line styles are:

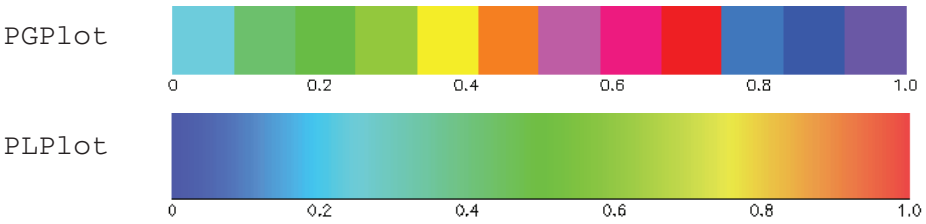


Figure 37.4: Continuous colors using the function `pg_continuous_color` in PGPlot and PLPlot. Typical usage: `call qp_routine(..., color = pg_continuous_color(0.25_rp), ...)`

```
1 -- solid$                  Solid
2 -- dashed$                 Dashed
3 -- dash_dot$               Dash--dot
4 -- dotted$                 Dotted
5 -- dash_dot3$              Dash--dot--dot--dot
```

The color styles in *Quick Plot* are:

```
0 -- White$   (actually the background color)
1 -- Black$   (actually the foreground color)
2 -- Red$
3 -- Green$
4 -- Blue$
5 -- Cyan$
6 -- Magenta$
7 -- Yellow$
8 -- Orange$
9 -- Yellow_Green$
10 -- Light_Green$
11 -- Navy_Blue$
12 -- Purple$
13 -- Reddish_Purple$
14 -- Dark_Grey$
15 -- Light_Grey$
```

Integers from [17, (largest integer)] represent continuous colors. The function `pq_continuous_color` maps [0.0, 1.0] to these integers. See Fig. 37.4.

The fill styles are:

```
1 -- solid_fill$
2 -- no_fill$
3 -- hatched$
4 -- cross_hatched$
```

The symbol types are:

```
0 -- square_sym$
1 -- dot_sym$
2 -- plus_sym$
3 -- times_sym$
4 -- circle_sym$
5 -- x_sym$
7 -- triangle_sym$
8 -- circle_plus_sym$
```

```
  9 -- circle_dot_sym$
 10 -- square_concave_sym$
 11 -- diamond_sym$
 12 -- star5_sym$
 13 -- triangle_filled_sym$
 14 -- red_cross_sym$
 15 -- star_of_david_sym$
 16 -- square_filled_sym$
 17 -- circle_filled_sym$
 18 -- star5_filled_sym$
```

Beside this list, PGPLOT maps other numbers onto symbol types. The PGPLOT list of symbols is:

```
-3 ... -31 - a regular polygon with abs(type) edges.
        -2 - Same as -1.
        -1 - Dot with diameter = current line width.
 0 ...  31 - Standard marker symbols.
32 ... 127 - ASCII characters (in the current font).
                E.G. to use letter F as a marker, set type = ICHAR("F").
     > 127 - A Hershey symbol number.
```

Table 37.1 shows some of the symbols and there associated numbers. Note: At constant height PGPLOT gives symbols of different size. To partially overcome this, *Quick Plot* scales some of the symbols to give a more uniform appearance. Table 37.1 was generated using a height of 40 via the call

```
call qp_draw_symbol (0.5_rp, 0.5_rp, "%BOX", k, height = 40.0_rp)
```

Table 37.3 shows how the character string `"\g<r>"`, where `"<r>"` is a Roman letter, map onto the Greek character set.

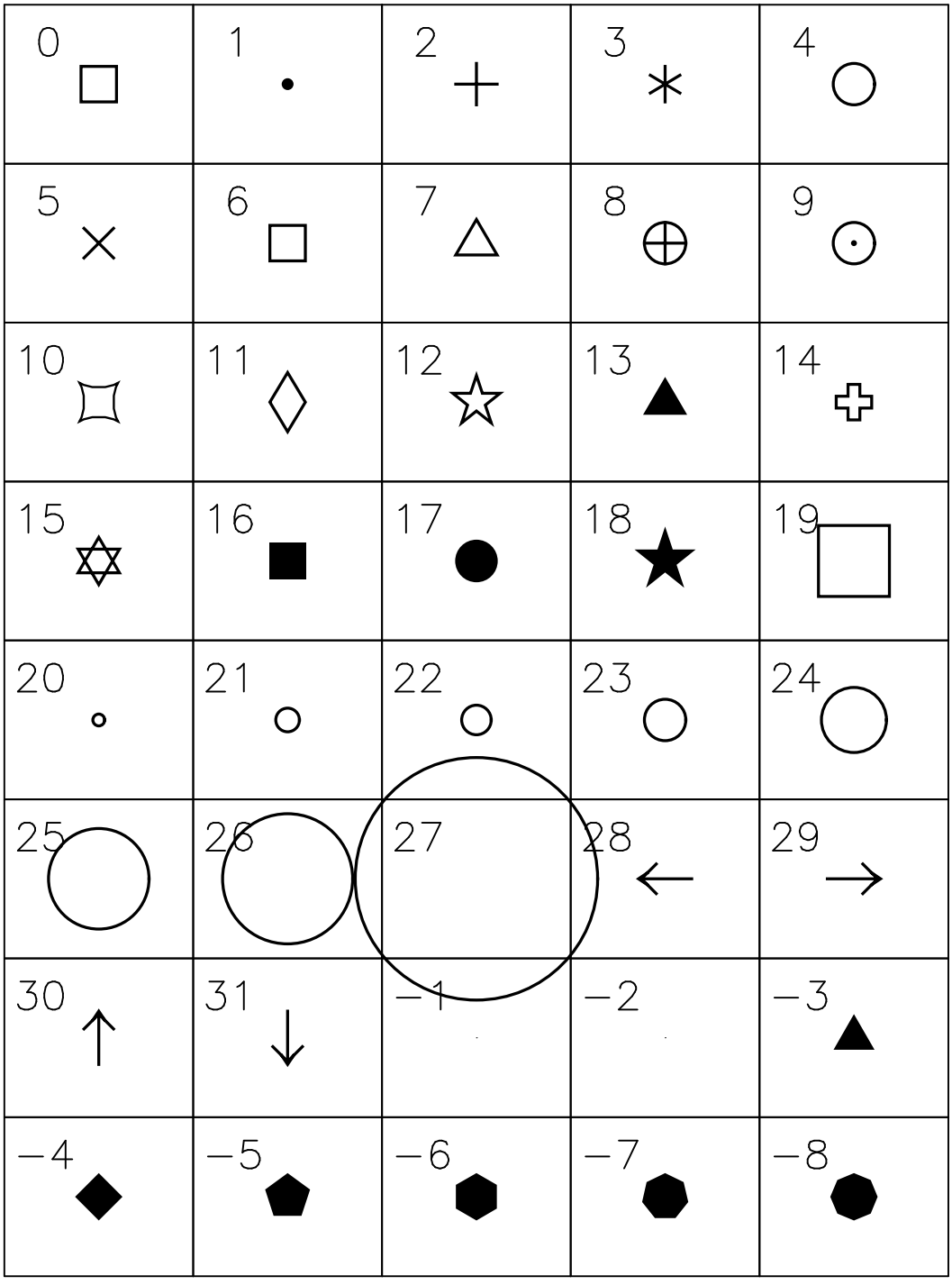


Table 37.1: Plotting Symbols at Height = 40.0

\u	Start a superscript or end a subscript
\d	Start a subscript or end a superscript. \u and \d must always be used in pairs
\b	Backspace (i.e., do not advance text pointer after plotting the previous character)
\fn	Switch to Normal font (1)
\fr	Switch to Roman font (2)
\fi	Switch to Italic font (3)
\fs	Switch to Script font (4)
\\	Backslash character (\)
\x	Multiplication sign (×)
\.	Centered dot (·)
\A	Angstrom symbol (Å)
\gx	Greek letter corresponding to roman letter x
\mn \mnn	Graph marker number n or nn (1-31)
\(nnnn)	Character number nnnn (1 to 4 decimal digits) from the Hershey character set; the closing parenthesis may be omitted if the next character is neither a digit nor ")". This makes a number of special characters (e.g., mathematical, musical, astronomical, and cartographical symbols) available.

Table 37.2: PGPLOT Escape Sequences.

Roman	a	b	g	d	e	z	y	h	i	k	l	m
Greek	α	β	γ	δ	ϵ	ζ	η	θ	ι	κ	λ	μ
Roman	n	c	o	p	r	s	t	u	f	x	q	w
Greek	ν	ξ	o	π	ρ	σ	τ	υ	ϕ	χ	ψ	ω
Roman	A	B	G	D	E	Z	Y	H	I	K	L	M
Greek	A	B	Γ	Δ	E	Z	H	Θ	I	K	Λ	M
Roman	N	C	O	P	R	S	T	U	F	X	Q	W
Greek	N	Ξ	O	Π	P	Σ	T	Υ	Φ	X	Ψ	Ω

Table 37.3: Conversion for the string "\g<r>" where "<r>" is a Roman character to the corresponding Greek character.

37.7 Structures

Quick Plot uses several structures to hold data. The structure that defines a line is a `qp_line_struct`

```
type qp_line_struct
  integer width   ! Line width. Default = 1
  integer color   ! Line color. Default = black$
  integer style   ! Line style. Default = solid$
end type
```

The `qp_symbol_struct` defines how symbols are drawn

```
type qp_symbol_struct
  integer  type        ! Default = circle_dot$
  real(rp) height      ! Default = 6.0 (points)
  integer  color       ! Default = black$
  integer  fill        ! Default = solid_fill$
  integer  line_width  ! Default = 1
end type
```

The `qp_axis_struct` defines how axes are drawn

```
type qp_axis_struct
  character(80) label       ! Axis label.
  real(rp) min              ! Axis range left/bottom number.
  real(rp) max              ! Axis range right/top number.
  real(rp) number_offset    ! Offset in inches of numbering from the axis line.
                            !  Default = 0.05
  real(rp) label_offset     ! Offset in inches of the label from the numbering.
                            !  Default = 0.05
  integer label_color       ! black$ (default), red$, etc.
  real(rp) major_tick_len   ! Length of the major ticks in inches. Def = 0.10
  real(rp) minor_tick_len   ! Length of the minor ticks in inches. Def = 0.06
  integer major_div         ! Number of major divisions. Default = 5
  integer major_div_nominal ! Nominal value. Def = 5.
  integer minor_div         ! Number of minor divisions. 0 = auto-choose. Default = 0
  integer minor_div_max     ! Maximum number for auto choose. Default = 5
  integer places            ! Places after the decimal point. Default = 0
  character(16) type        ! 'LINEAR' (default), 'LOG', or 'CUSTOM'.
  character(16) bounds      ! 'GENERAL' (default), 'ZERO_AT_END', ZERO_SYMMETRIC.
  integer tick_side         ! +1 = draw to the inside (def), 0 = both, -1 = outside.
  integer number_side       ! +1 = draw to the inside, -1 = outside (default).
  logical draw_label        ! Draw the label? Default = True.
  logical draw_numbers      ! Draw the numbering? Default = True.
end type
```

Finally, the `qp_plot_struct` is a container for the axis that make up a plot

```
type qp_plot_struct
  character(80) :: title = ' '
  type (qp_axis_struct) x, y, x2, y2
  type (qp_axis_struct), pointer :: xx, yy  ! Pointer to axes used for plotting.
  logical :: draw_box    = .true.
  logical :: draw_title  = .true.
  logical :: draw_grid   = .true.
  logical :: x2_mirrors_x = .true.
  logical :: y2_mirrors_y = .true.
```

```
    logical :: xx_points_to_x
    logical :: yy_points_to_y
  end type
```

Chapter 38

Helper Routines

This chapter gives an overview of various computational helper routines.

38.1 Nonlinear Optimization

Nonlinear optimization is the process of finding a minimum (or maximum) of a nonlinear function (the "merit" function). Nonlinear optimization is frequently used for lattice design or matching of data to a model. For more information on this see the *Tao* manual.

In terms of routines for implementing nonlinear optimization the Numerical Recipes library (§25.2 that is distributed along with *Bmad* contains several. In particular, the routine super_mrqmin which implements the Levenberg–Marquardt is an excellent routine for finding local minimum when the merit function can be expressed as the sum of quadratic terms. Another routine, `frprmn`, which is an implementation of the Fletcher–Reeves algorithm, is also good at finding local minimum and has the advantage that as input it does not need a derivative matrix as does Levenberg–Marquardt. The disadvantage of Fletcher–Reeves is that it is slower than Levenberg–Marquardt.

A second implementation of Levenberg–Marquardt available with *Bmad* is opti_lmdif which is Fortran90 version of the popular `lmdif` routine. Also available is opti_de which implements the Differential Evolution algorithm of Storn and Price[Storn96]. This routine is good for finding global minima but can be slow.

Another routine that should be mentioned is the `amoeba` routine from Numerical Recipes that implements the downhill simplex method of Neider and Mead. This routine is robust but slow but is easily parallelized so it is a good routine for parallel processing.

38.2 Matrix Manipulation

There are a number of *Bmad* routines for matrix manipulation as listed in §39.22. In fact, Fortran90 has a number of intrinsic matrix routines as well but this is outside the scope of this manual. The following example shows some of the *Bmad* matrix routines mat_inverse and mat_make_unit.

```
real(rp) mat(6,6), mat_inv(6,6)
call mat_make_unit (mat)    ! make a unit matrix
call mat_inverse (mat, mat_inv) ! Compute the inverse matrix.
```

Chapter 39

Bmad Library Routine List

Below are a list of *Bmad* and sim_utils routines sorted by their functionality. Use the `getf` and `listf` (§25.3) scripts for more information on individual routines. This list includes low level routines that are not generally used in writing code for a program but may be useful in certain unique situations. Excluded from the list are very low level routines that are solely meant for *Bmad* internal use.

39.1 Beam: Low Level Routines

The following helper routines are generally not useful for general use.

bend_edge_kick (ele, param, particle_at, orb, mat6, make_matrix, track_spin)
Subroutine to track through the edge field of an sbend. Reverse tracking starts with the particle just outside the bend and

find_bunch_sigma_matrix (particle, charge, bunch_params, sigma_s)
Routine to find the sigma matrix elements of a particle distribution.

init_spin_distribution (beam_init, bunch)
Initializes a spin distribution according to init_beam%spin

order_particles_in_z (bunch)
Routine to order the particles longitudinally in terms of decreasing %vec(5). That is from large z (head of bunch) to small z.

track1_bunch (bunch_start, lat, ele, bunch_end, err, centroid, direction)
Routine to track a bunch of particles through an element.

track1_bunch_hom (bunch_start, ele, param, bunch_end, direction)
Routine to track a bunch of particles through an element.

39.2 Beam: Tracking and Manipulation

See §32.16 for a discussion of using a collection of particles to simulate a bunch.

bbi_kick (x_norm, y_norm, r, kx, ky)
Routine to compute the normalized kick due to the beam-beam interaction using the normalized position for input.

calc_bunch_params (bunch, bunch_params, err, print_err)
Finds all bunch parameters defined in bunch_params_struct, both normal-mode and projected

calc_bunch_params (bunch, bunch_params, plane, slice_center, slice_spread, err, print_err)
Finds all bunch parameters for a slice through the beam distribution.

init_beam_distribution (ele, param, beam_init, beam, err_flag)
Routine to initialize a distribution of particles matched to the Twiss parameters, centroid position, and Energy - z correlation

init_bunch_distribution (ele, param, beam_init, ix_bunch, bunch, err_flag)
Routine to initialize either a random or tail-weighted distribution of particles.

reallocate_beam (beam, n_bunch, n_particle)
Routine to reallocate memory within a beam_struct.

reallocate_bunch (bunch, n_particle)
Subroutine to reallocate particles within a bunch_struct.

track_beam (lat, beam, ele1, ele2, err, centroid, direction)
Routine to track a beam of particles from the end of lat%ele(ix1) Through to the end of lat%ele(ix2).

39.3 Branch Handling Routines

allocate_branch_array (lat, upper_bound)
Routine to allocate or re-allocate an branch array. The old information is saved.

transfer_branch (branch1, branch2)
Routine to set branch2 = branch1. This is a plain transfer of information not using the overloaded equal.

transfer_branches (branch1, branch2)
Routine to set branch2 = branch1. This is a plain transfer of information not using the overloaded equal.

39.4 Coherent Synchrotron Radiation (CSR)

csr_bin_particles (particle, csr, err_flag)
Routine to bin the particles longitudinally in s.

csr_bin_kicks (ds_kick_pt, csr, err_flag)
Routine to cache intermediate values needed for the csr calculations.

csr_kick_calc (csr, particle)
Routine to calculate the longitudinal coherent synchrotron radiation kick.

i_csr (kick1, i_bin, csr) result (i_this)
Routine to calculate the CSR kick integral.

z_calc_csr (d, k_factor, bin, small_angle_approx, dz_dd) result (z_this)
Routine to calculate the distance between the source particle and the kicked particle.

d_calc_csr (dz_particles, k_factor, bin, small_angle_approx) result (d_this)
Routine to calculate the distance between source and kick points.

39.5 Collective Effects

setup_ultra_rel_space_charge_calc (calc_on, lattice, n_part, mode, closed_orb)
Routine to initialize constants needed by the transverse space charge tracking routine track1_-space_charge.

touschek_lifetime (mode, Tl, lat)
Routine to calculate the Touschek lifetime for a lat.

39.6 Custom Routines

apply_element_edge_kick_hook (orb, fringe_info, track_ele, param, finished, mat6, make_matrix, rf_time)

Routine that can be customized to track through the edge field of an element. This routine is always called by apply_element_edge_kick.

check_aperture_limit_custom (orb, ele, particle_at, param, err_flag)
Routine to check if an orbit is outside an element's aperture. Used when `ele%aperture_type` is set to `custom$`

ele_geometry_hook (floor0, ele, floor, finished, len_scale)
Routine that can be customized to calculate the floor position of an element.

ele_to_fibre_hook (ele, ptc_fibre, param)
Routine that can be customized for creating a PTC fibre from a Bmad element. This routine is always called by ele_to_fibre.

em_field_custom(ele, param, s_rel, orbit, local_ref_frame, field,
calc_dfield, err_flag, calc_potential, use_overlap,
grid_allow_s_out_of_bounds, rf_time, used_eles)
Custom routine for calculating fields.

init_custom (ele, err_flag)
Routine for initializing custom elements or elements that do custom calculations.

make_mat6_custom (ele, param, start_orb, end_orb, err_flag)
Routine for custom calculations of the 6x6 transfer matrices.

radiation_integrals_custom (lat, ir, orb, err_flag)
User supplied routine to calculate the synchrotron radiation integrals for a custom element.

time_runge_kutta_periodic_kick_hook (orbit, z_phase, ele, param, stop_time, init_needed)

Custom routine to add a kick to a particle at periodic times.

track1_bunch_hook (bunch_start, lat, ele, bunch_end, err, centroid, direction, finished)

Routine that can be customized for tracking a bunch through a single element.

track1_custom (start_orb, ele, param, end_orb, err_flag, finished, track)
Dummy routine for custom tracking.

track1_postprocess (start_orb, ele, param, end_orb)
Dummy routine for post processing after the track1 routine is done.

track1_preprocess (start_orb, ele, param, err_flag, finished, radiation_included, track)
Dummy routine for pre processing at the start of the track1 routine.

track1_spin_custom (start, ele, param, end, err_flag, make_quaternion)
Dummy routine for custom spin tracking. This routine needs to be replaced for a custom calculation.

track1_wake_hook (bunch, ele, finished)
Routine that can be customized for tracking through a wake.

wall_hit_handler_custom (orb, ele, s)
This routine is called by the Runge-Kutta integrator odeint_bmad when a particle hits a wall.

39.7 Electro-Magnetic Fields

em_field_calc (ele, param, s_pos, orbit, local_ref_frame, field, calc_dfield, err_flag, calc_potential, use_overlap, grid_allow_s_out_of_bounds, rf_time, used_eles, err_print_out_of_bounds)
Routine to calculate the E and B fields for an element.

em_field_custom(ele, param, s_rel, time, orb, local_ref_frame, field, calc_dfield, err_flag)
Custom routine for calculating fields.

39.8 Helper Routines: File, System, and IO

append_subdirectory (dir, sub_dir, dir_out, err)
Routine to combine a directory specification with a subdirectory specification to form a complete directory

cesr_iargc ()
Platform independent function to return the number of command line arguments. Use this with cesr_getarg.

cesr_getarg (i_arg, arg)
Platform independent function to return the i'th command line argument. Use this with cesr_iargc.

dir_close ()
Routine to close a directory that was opened with dir_open. Also see dir_read.

dir_open (dir_name) result (opened)
Routine to open a directory to obtain a list of its files. Use this routine with dir_read and dir_close.

dir_read (file_name) result (valid)
Routine to get the names of the files in a directory. Use this routine with dir_open and dir_close.

file_suffixer (in_file_name, out_file_name, suffix, add_switch)
Routine to add/replace a suffix to a file name.

get_tty_char (this_char, wait, flush)
Routine for getting a single character from the terminal. Also see: get_a_char

get_a_char (this_char, wait, ignore_this)
Routine for getting a single character from the terminal. Also see: get_tty_char

get_file_time_stamp (file, time_stamp)
Routine to get the "last modified" time stamp for a file.

lunget()
Function to return a free file unit number to be used with an open statement.

milli_sleep (milli_sec)
Routine to pause the program for a given number of milli-seconds.

out_io (...)
Routine to print to the terminal for command line type programs. The idea is that for programs with a gui this routine can be easily replaced with another routine.

out_io_called (level, routine_name)
Dummy routine for linker. See out_io for more details.

out_io_end ()
Dummy routine for linker. See out_io for more details.

out_io_line (line)
Dummy routine for linker. See out_io for more details.

output_direct (file_unit, print_and_capture, min_level, max_level)
Routine to set where the output goes when out_io is called. Output may be sent to the terminal screen, written to a file, or both. Also can be used to restrict output verbosity.

read_a_line (prompt, line_out, trim_prompt, prompt_color, prompt_bold, history_file)

Routine to read a line of input from the terminal. The line is also add to the history buffer so that the up-arrow

skip_header (ix_unit, error_flag)
Routine to find the first line of data in a file.

splitfilename(filename, path, basename, is_relative) result (ix_char)
Routine to take filename and splits it into its constituent parts, the directory path and the base file name.

system_command (line)
Routine to execute an operating system command from within the program.

type_this_file (filename)
Routine to type out a file to the screen.

39.9 Helper Routines: Math (Except Matrix)

complex_error_function (wr, wi, zr, zi)
This routine evaluates the function w(z) in the first quadrant of the complex plane.

cross_product (a, b)
Returns the cross product of a x b

linear_fit (x, y, n_data, a, b, sig_a, sig_b)
Routine to fit to y = A + B x

modulo2 (x, amp)
Function to return y = x + 2 * n * amp, n is an integer, such that y is in the interval [-amp, amp].

ran_engine (set, get, ran_state)
Routine to set what random number generator algorithm is used. If this routine is never called then pseudo_random$ is used.

ran_gauss (harvest)
Routine to return a Gaussian distributed random number with unit sigma.

ran_gauss_converter (set, set_sigma_cut, get, get_sigma_cut, ran_state)
Routine to set what conversion routine is used for converting uniformly distributed random numbers to Gaussian distributed random numbers.

ran_seed_put (seed, ran_state)
Routine to seed the random number generator.

ran_seed_get (seed, ran_state)
Routine to return the seed used for the random number generator.

ran_uniform (harvest)
Routine to return a random number uniformly distributed in the interval [0, 1]. This routine uses the same algorithm as ran from

spline_akima (spline, ok)
Given a set of (x,y) points we want to interpolate between the points. This routine computes the semi-hermite cubic spline developed by akima

spline_evaluate (spline, x, ok, y, dy)
Routine to evaluate a spline at a set of points.

super_ludcmp (a,indx,d, err)
This routine is essentially ludcmp from Numerical Recipes with the added feature that an error flag is set instead of bombing the program when there is a problem.

39.10 Helper Routines: Matrix

mat_eigen (mat, eigen_val, eigen_vec, error, print_err)
Routine for determining the eigen vectors and eigen values of a matrix.

mat_inverse (mat, mat_inv, ok, print_err)
Routine to take the inverse of a square matrix.

mat_make_unit (mat)
routine to create a unit matrix.

make_pseudoinverse(mat,mat_pinv,svd_condition,print_err,ok) Routine to make the Moore-Penrose pseudoinverse of a rectangular matrix via LAPACK DGESDD routine for singular value decomposition. If svd_condition is specified, then only those singular values where s(i)/s(1) > svd_condition are used to construct the pseudoinverse.

mat_rotation (mat, angle, bet_1, bet_2, alph_1, alph_2)
Routine to construct a 2x2 rotation matrix for translation from point 1 to point 2.

mat_symplectify (mat_in, mat_symp, p0_ratio, r_root)
Routine to form a symplectic matrix that is approximately equal to the input matrix.

mat_symp_error (mat, p0_ratio, err_mat) result (error)
Routine to check the symplecticity of a square matrix

mat_symp_conj (mat) result (mat_conj)
Routine to take the symplectic conjugate of a square matrix.

mat_symp_decouple (t0, stat, u, v,
ubar, vbar, g, twiss1, twiss2, gamma, type_out)
Routine to find the symplectic eigen–modes of the one turn 4x4 coupled transfer matrix T0.

mat_type (mat, nunit, header, num_form)
Routine to output matrices to the terminal or to a file

39.11 Helper Routines: Miscellaneous

date_and_time_stamp (string, numeric_month)
Routine to return the current date and time in a character string.

err_exit()
Routine to first show the stack call list before exiting. This routine is typically used when a program detects an error condition.

integer_option (integer_default, opt_integer)
Function to return True or False depending upon the state of an optional integer.

is_false (param) result (this_false)
Routine to translate from a real number to a boolian True or False. Translation: 0 = False, nonzero = True.

is_true (param) result (this_true)
Routine to translate from a real number to a boolian True or False. Translation: 0 = False, nonzero = True.

logic_option (logic_default, opt_logic)
Function to return True or False depending upon the state of an optional logical.

re_allocate (ptr_to_array, n, exact)
Function to reallocate a pointer to an array of strings, integers, reals, or logicals.

re_associate (array, n)
Function to reassociate an allocatable array of strings, integers, reals, or logicals.

real_option (real_default, opt_real)
Function to return True or False depending upon the state of an optional real.

string_option (string_default, opt_string)
Routine to return True or False depending upon the state of an optional string.

39.12 Helper Routines: String Manipulation

downcase_string (string)
Routine to convert a string to lowercase:

index_nocase (string, match_str) result (indx)
Function to look for a sub-string of string that matches match_str. This routine is similar to the fortran INDEX function

is_integer (string)
Function to tell if the first word in a string is a valid integer.

is_logical (string, ignore) result (good)
Function to test if a string represents a logical. Accepted possibilities are (individual characters can be either case):

is_real (string, ignore) result (good)
Function to test if a string represents a real number.

match_reg (str, pat)
Function for matching with regular expressions. Note: strings are trimmed before comparison.

match_wild (string, template) result (this_match)
Function to do wild card matches. Note: trailing blanks will be discarded before any matching is done.

match_word (string, names, ix, exact_case, can_abbreviate, matched_name)
Routine to match the first word in a string against a list of names. Abbreviations are accepted.

on_off_logic (logic, true_str, false_str) result (name)
Function to return the string "ON" or "OFF".

str_match_wild(str, pat) result (a_match)
Function to match a character string against a regular expression pattern.

string_to_int (line, default, err_flag, err_print_flag)
Routine to convert a string to an integer.

string_to_real (line, default, err_flag, err_print_flag)
Routine to convert a string to an real.

string_trim(in_string, out_string, word_len)
Routine to trim a string of leading blanks and/or tabs and also to return the length of the first word.

string_trim2 (in_str, delimitors, out_str, ix_word, delim, ix_next)
Routine to trim a string of leading delimiters and also to return the length of the first word.

str_downcase (dst, src)
Routine to convert a string to down case.

str_substitute (string, str_match, str_replace, do_trim)
Routine to substitute all instances of one sub-string for another in a string

upcase (str_in) result (str_out)
Routine to convert a string to upper case.

upcase_string (string)
Routine to convert a string to uppercase:

39.13 Helper Routines: Switch to Name

coord_state_name (coord_state) result (state_str)
Routine to return the string representation of a coord%state state.

39.14 Inter-Beam Scattering (IBS)

ibs_lifetime(lat,ibs_sim_params,maxratio,lifetime,granularity)
This module computes the beam lifetime due to the diffusion process according to equation 12

39.15 Lattice: Element Manipulation

These routine are for adding elements, moving elements, etc.

add_lattice_control_structs (ele, n_add_slave, n_add_lord, n_add_slave_field, n_add_lord_field, add_at_end)
Routine to adjust the control structure of a lat so that extra control elements can be added.

add_superimpose (lat, super_ele_in, ix_branch, err_flag, super_ele_out, save_null_drift, create_jumbo_slave, ix_insert)
Routine to make a superimposed element.

attribute_bookkeeper (ele, param, force_bookkeeping)
Routine to make sure the attributes of an element are self-consistent.

autoscale_phase_and_amp(ele, param, err_flag, scale_phase, scale_amp, call_bookkeeper)
Routine to set the phase offset and amplitude scale of the accelerating field. This routine works on lcavity, rfcavity and e_gun elements.

create_element_slice (sliced_ele, ele_in, l_slice, offset, param, include_upstream_end, include_downstream_end, err_flag, old_slice)
Routine to transfer the %value, %wig_term, and %wake%lr information from a superposition lord to a slave when the slave has only one lord.

create_field_overlap (lat, lord_name, slave_name, err_flag)
Subroutine to add the bookkeeping information to a lattice for an element's field overlapping another element.

create_group (lord, contrl, err, err_print_flag)
Routine to create a group control element.

create_girder (lat, ix_girder, contrl, girder_info, err_flag)
Routine to add the controller information to slave elements of an girder_lord.

create_overlay (lord, contrl, err, err_print_flag)
Routine to add the controller information to slave elements of an overlay_lord.

create_wiggler_model (wiggler_in, lat)
Routine to create series of bend and drift elements to serve as a model for a wiggler. This routine uses the mrqmin nonlinear optimizer to vary the parameters in the wiggler

insert_element (lat, insert_ele, insert_index, ix_branch, orbit)
Routine to Insert a new element into the tracking part of the lat structure.

make_hybrid_lat (lat_in, lat_out, use_taylor, orb0_arr)
Routine to concatenate together elements to make a hybrid lat

new_control (lat, ix_ele)
Routine to create a new control element.

pointer_to_attribute (ele, attrib_name, do_allocation, a_ptr, err_flag, err_print_flag, ix_attrib)
Returns a pointer to an attribute of an element with name attrib_name.

pointers_to_attribute (lat, ele_name, attrib_name, do_allocation, ptr_array, err_flag, err_print_flag, eles, ix_attrib)
Returns an array of pointers to an attribute with name attrib_name within elements with name ele_name.

pointer_to_branch
Routine to return a pointer to a lattice branch.

pointer_to_next_ele (this_ele, offset, skip_beginning, follow_fork) result (next_ele)
Function to return a pointer to the Nth element relative to this_ele in the array of elements in a lattice branch.

pointer_to_ele (lat, ix_ele, ix_branch) result (ele_ptr)
pointer_to_ele (lat, ele_loc_id) result (ele_ptr)
Routine to point to a given element.

pointer_to_element_at_s (branch, s, choose_max, err_flag, s_eff, position) result (ele)

Function to return a pointer to the element at position s.

remove_eles_from_lat (lat, check_sanity)
Routine to remove an elements from the lattice.

set_ele_attribute (ele, set_string, err_flag, err_print_flag)
Routine to set an element's attribute.

set_ele_status_stale (ele, status_group, set_slaves)
Routine to set a status flags to stale in an element and the corresponding ones for any slaves the element has.

set_status_flags (bookkeeping_state, stat)
Routine to set the bookkeeping status block.

split_lat (lat, s_split, ix_branch, ix_split, split_done, add_suffix, check_sanity, save_null_drift, err_flag, choose_max)
Routine to split a lat at a point.

value_of_attribute (ele, attrib_name, err_flag, err_print_flag, err_value) result (value)

Returns the value of an element attribute.

39.16 Lattice: Geometry

ele_geometry (floor_start, ele, floor, len_scale, set_ok, ignore_patch_err)
Routine to calculate the physical (floor) placement of an element given the placement of the preceding element. This is the same as the MAD convention.

floor_angles_to_w_mat (theta, phi, psi, w_mat, w_mat_inv)
Routine to construct the W matrix that specifies the orientation of an element in the global "floor" coordinates. See the Bmad manual for more details.

coords_floor_to_relative (floor0, global_position, calculate_angles, is_delta_position) result (local_position)
Returns local floor position relative to floor0 given a global floor position. This is an essentially an inverse of routine coords_relative_to_floor.

floor_w_mat_to_angles (w_mat, theta, phi, psi, floor0)
Routine to construct the angles that define the orientation of an element in the global "floor" coordinates from the W matrix. See the Bmad manual for more details.

lat_geometry (lat)
Routine to calculate the physical placement of all the elements in a lattice. That is, the physical machine layout on the floor.

coords_relative_to_floor (floor0, dr, theta, phi, psi) result (floor1)
Starting from a given reference frame and given a shift in position, return the resulting reference frame.

patch_flips_propagation_direction (x_pitch, y_pitch) result (is_flip)
Routine to determine if the propagation direction is flipped in a patch. This is true if the tranformation matrix element S(3,3) = cos(x_pitch) * cos(y_pitch)

coords_local_curvilinear_to_floor (local_position, ele, in_ele_frame, w_mat, calculate_angles, use_patch_entrance) result (global_position)
Given a position local to ele, return global floor coordinates.

coords_floor_to_local_curvilinear (global_position, ele, status, w_mat, use_patch_entrance) result(local_position)
Given a position in global coordinates, return local curvilinear coordinates in ele relative to floor0

s_calc (lat)
Routine to calculate the longitudinal distance S for the elements in a lat.

w_mat_for_x_pitch (x_pitch, return_inverse)
Routine to return the transformation matrix for an x_pitch.

w_mat_for_y_pitch (y_pitch, return_inverse)
Routine to return the transformation matrix for an y_pitch.

w_mat_for_tilt (tilt, return_inverse)
Routine to return the transformation matrix for an tilt.

39.17 Lattice: Informational

attribute_free (ix_ele, attrib_name, lat, err_print_flag, except_overlay) result (free)
attribute_free (ele, attrib_name, lat, err_print_flag, except_overlay) result (free)
attribute_free (ix_ele, ix_branch, attrib_name, lat, err_print_flag, except_overlay) result (free)
Overloaded function to check if an attribute is free to vary.

attribute_index (ele, name, full_name)
Function to return the index of an attribute for a given element type and the name of the attribute

attribute_name (ele, ix_att)
Function to return the name of an attribute for a particular type of element.

attribute_type (attrib_name, ele) result (attrib_type)
Routine to return the type (logical, integer, real, or named) of an attribute.

branch_name(branch) result (name)
Routine to return a string with the lattice branch name encoded. This routine is useful for error messages.

check_if_s_in_bounds (branch, s, err_flag, translated_s)
Routine to check if a given longitudinal position s is within the bounds of a given branch of a lattice.

element_at_s (lat, s, choose_max, ix_branch, err_flag, s_eff, position) result (ix_ele)
Routine to return the index of the element at position s.

ele_has_offset (ele) result (has_offset)
Function to tell if an element has a non-zero offset, pitch or tilt.

ele_loc_to_string (ele, show_branch0) result (str)
Routine to encode an element's location into a string.

ele_to_lat_loc (ele) result (ele_loc)
Function to return an lat_ele_loc_struct identifying where an element is in the lattice.

equivalent_taylor_attributes (ele_taylor, ele2) result (equiv)
Routine to see if two elements are equivalent in terms of their attributes so that their Taylor Maps, if they existed, would be the same.

find_element_ends (ele, ele1, ele2, ix_multipass)
Routine to find the end points of an element.

get_slave_list (lord, slaves, n_slave)
Routine to get the list of slaves for an element.

key_name (key_index)
Translate an element key index (EG: quadrupole$, etc.) to a character string.

key_name_to_key_index (key_str, abbrev_allowed) result (key_index)
Function to convert a character string (eg: "drift") to an index (eg: drift$).

lat_ele_locator (loc_str, lat, eles, n_loc, err, above_ubound_is_err, ix_dflt_branch)
Routine to locate all the elements in a lattice that corresponds to loc_str.

lat_sanity_check (lat, err_flag)
Routine to check the control links in a lat structure, etc.

n_attrib_string_max_len () result (max_len)
Routine to return the maximum number of characters in any attribute name known to bmad.

name_to_list (lat, ele_names)
Routine to make a list of the elements in a lat whose name matches the names in the ele_names list.

num_lords (slave, lord_type) result (num)
Routine to return the number of lords of a given type for a given lattice element.

num_lords (slave, lord_type) result (num)
Routine to return the number of lords of a lattice element of a certain type.

pointer_to_indexed_attribute (ele, ix_attrib, do_allocation, a_ptr, err_flag, err_print_flag)
Returns a pointer to an attribute of an element ele with attribute index ix_attrib.

pointer_to_lord (slave, ix_lord, control, ix_slave, field_overlap_ptr) result (lord_ptr)
Function to point to a lord of a slave.

pointer_to_multipass_lord (ele, ix_pass, super_lord) result (multi_lord)
Routine to find the multipass lord of a lattice element. A multi_lord will be found for:

pointer_to_slave (lord, ix_slave, control, field_overlap_ptr) result (slave_ptr)
Function to point to a slave of a lord.

rf_is_on (branch) result (is_on)
Routine to check if any rfcavity is powered in a branch.

switch_attrib_value_name (attrib_name, attrib_value, ele, is_default, name_list) result (val_name)
Routine to return the name corresponding to the value of a given switch attribute.

type_ele (ele, type_zero_attrib, type_mat6, type_taylor, twiss_out, type_control, type_wake, type_floor_coords, type_field, type_wall, lines, n_lines)
Subroutine to print or put in a string array information on a lattice element.

type_twiss (ele, frequency_units, compact_format, lines, n_lines)
Subroutine to print or put in a string array Twiss information from an element.

valid_tracking_method (ele, species, tracking_method) result (is_valid)
Routine to return whether a given tracking method is valid for a given element.

valid_mat6_calc_method (ele, species, mat6_calc_method) result (is_valid)
Routine to return whether a given mat6_calc method is valid for a given element.

39.18 Lattice: Low Level Stuff

bracket_index (s_arr, i_min, i_max, s, ix)
Routine to find the index ix so that s(ix) $\leq$ s $<$ s(ix+1). If s $<$ s(1) then ix = 0

check_controller_controls (contrl, name, err)
Routine to check for problems when setting up group or overlay controllers.

deallocate_ele_pointers (ele, nullify_only, nullify_branch, dealloc_poles)
Routine to deallocate the pointers in an element.

re_allocate_eles (eles, n, save_old, exact)
Routine to allocate an array of ele_pointer_structs.

twiss1_propagate (twiss1, mat2, ele_key, length, twiss2, err)
Routine to propagate the twiss parameters of a single mode.

39.19 Lattice: Manipulation

allocate_element_array (ele, upper_bound, init_ele0)
Routine to allocate or re-allocate an element array.

allocate_lat_ele_array (lat, upper_bound, ix_branch)
Routine to allocate or re-allocate an element array.

control_bookkeeper (lat, ele, err_flag)
Routine to calculate the combined strength of the attributes for controlled elements.

deallocate_ele_array_pointers (eles)
Routine to deallocate the pointers of all the elements in an element array and the array itself.

deallocate_lat_pointers (lat)
Routine to deallocate the pointers in a lat.

init_ele (ele, key, sub_key, ix_ele, branch)
Routine to initialize an element.

init_lat (lat, n)
Routine to initialize a Bmad lat.

lattice_bookkeeper (lat, err_flag)
Routine to do bookkeeping for the entire lattice.

reallocate_coord (coord, n_coord)
Routine to reallocate an allocatable coord_struct array to at least: coord(0:n_coord).

reallocate_coord_array (coord_array, lat)
Routine to allocate an allocatable coord_array_struct array to the proper size for a lattice.

set_custom_attribute_name (custom_name, err_flag, custom_index)
Routine to add custom element attributes to the element attribute name table.

set_ele_defaults (ele, do_allocate)
Subroutine to set the defaults for an element of a given type.

set_on_off (key, lat, switch, orb, use_ref_orb, ix_branch, saved_values, attribute)
Routine to turn on or off a set of elements (quadrupoles, RF cavities, etc.) in a lat.

transfer_ele (ele1, ele2, nullify_pointers)
Routine to set ele2 = ele1. This is a plain transfer of information not using the overloaded equal.

transfer_eles (ele1, ele2)
Routine to set ele2(:) = ele1(:). This is a plain transfer of information not using the overloaded equal.

transfer_ele_taylor (ele_in, ele_out, taylor_order)
Routine to transfer a Taylor map from one element to another.

transfer_lat (lat1, lat2)
Routine to set lat2 = lat1. This is a plain transfer of information not using the overloaded equal.

transfer_lat_parameters (lat_in, lat_out)
Routine to transfer the lat parameters (such as lat%name, lat%param, etc.) from one lat to another.

zero_ele_kicks (ele)
Subroutine to zero any kick attributes like hkick, $bl_v kick$, etc. See also: ele_has_kick, ele_has_-offset, zero_ele_offsets.

zero_ele_offsets (ele)
Routine to zero the offsets, pitches and tilt of an element.

39.20 Lattice: Miscellaneous

c_multi (n, m, no_n_fact, c_full)
Routine to compute multipole factors: c_multi(n, m) = +/- ("n choose m")/n!

ele_compute_ref_energy_and_time (ele0, ele, param, err_flag)
Routine to compute the reference energy and reference time at the end of an element given the reference enegy and reference time at the start of the element.

lat_compute_ref_energy_and_time (lat, err_flag)
Routine to compute the reference energy for each element in a lattice.

field_interpolate_3d (position, field_mesh, deltas, position0)
Function to interpolate a 3d field.

order_super_lord_slaves (lat, ix_lord)
Routine to make the slave elements of a super_lord in order.

release_rad_int_cache (ix_cache)
Routine to release the memory associated with caching wiggler values.

set_flags_for_changed_attribute (ele, attrib)
Routine to mark an element as modified for use with "intelligent" bookkeeping.

39.21 Lattice: Reading and Writing Files

aml_parser (lat_file, lat, make_mats6, digested_read_ok, use_line, err_flag)
Routine to parse an AML input file and put the information in a lat_struct.

bmad_and_xsif_parser (lat_file, lat, make_mats6, digested_read_ok, use_line, err_flag)
Subroutine to parse either a Bmad or XSIF (extended standard input format) lattice files.

bmad_parser (lat_file, lat, make_mats6, digested_read_ok, use_line, err_flag)
Routine to parse (read in) a Bmad input file.

bmad_parser2 (lat_file, lat, orbit, make_mats6, err_flag)
Routine to parse (read in) a Bmad input file to modify an existing lattice.

write_lattice_in_foreign_format (out_type, out_file_name, lat, ref_orbit, use_matrix_model, include_apertures, dr12_drift_max, ix_start, ix_end, ix_branch, converted_lat, err)
Routine to write a mad or xsif lattice file using the information in a lat_struct.

combine_consecutive_elements (lat)
Routine to combine consecutive elements in the lattice that have the same name. This allows simplification, for example, of lattices where elements have been split to compute the beta function at the center.

create_sol_quad_model (sol_quad, lat)
Routine to create series of solenoid and quadrupole elements to serve as a replacement model for a sol_quad element.

create_unique_ele_names (lat, key, suffix)
Routine to give elements in a lattice unique names.

read_digested_bmad_file (digested_file, lat, inc_version, err_flag, parser_calling)
Routine to read in a digested file.

write_bmad_lattice_file (bmad_file, lat, err, output_form)
Routine to write a Bmad lattice file using the information in a lat_struct.

write_digested_bmad_file (digested_name, lat, n_files, file_names, extra, err_flag)
Routine to write a digested file.

xsif_parser (xsif_file, lat, make_mats6, digested_read_ok, use_line, err_flag)
Routine to parse an XSIF (extended standard input format) lattice file.

39.22 Matrices

c_to_cbar (ele, cbar_mat)
Routine to compute Cbar from the C matrix and the Twiss parameters.

cbar_to_c (cbar_mat, a, b, c_mat)
Routine to compute C coupling matrix from the Cbar matrix and the Twiss parameters.

clear_lat_1turn_mats (lat)
Clear the 1-turn matrices in the lat structure.

concat_transfer_mat (mat_1, vec_1, mat_0, vec_0, mat_out, vec_out)
Routine to concatinate two linear maps

determinant (mat) result (det)
Routine to take the determinant of a square matrix This routine is adapted from Numerical Recipes.

do_mode_flip (ele, err_flag)
Routine to mode flip the Twiss parameters of an element

make_g2_mats (twiss, g2_mat, g2_inv_mat)
Routine to make the matrices needed to go from normal mode coords to coordinates with the beta function removed.

make_g_mats (ele, g_mat, g_inv_mat)
Routine to make the matrices needed to go from normal mode coords to coordinates with the beta function removed.

make_mat6 (ele, param, start_orb, end_orb, err_flag)
Routine to make the 6x6 transfer matrix for an element.

make_v_mats (ele, v_mat, v_inv_mat)
Routine to make the matrices needed to go from normal mode coords to X-Y coords and vice versa.

mat6_from_s_to_s (lat, mat6, vec0, s1, s2, orbit, ix_branch, one_turn, unit_start, err_flag, ele_save)
Subroutine to calculate the transfer map between longitudinal positions s1 to s2.

mat6_to_taylor (vec0, mat6, bmad_taylor)
Routine to form a first order Taylor map from the 6x6 transfer matrix and the 0th order transfer vector.

match_ele_to_mat6 (ele, start_orb, mat6, vec0, err_flag, twiss_ele, include_delta_time)

Routine to make the 6 x 6 transfer matrix from the twiss parameters.

multi_turn_tracking_to_mat (track, i_dim, map1, map0, track0, chi)
Routine to analyze 1-turn tracking data to find the 1-turn transfer matrix and the closed orbit offset.

transfer_matrix_calc (lat, xfer_mat, xfer_vec, ix1, ix2, ix_branch, one_turn)
Routine to calculate the transfer matrix between two elements. If ix1 and ix2 are not present the full 1–turn matrix is calculated.

one_turn_mat_at_ele (ele, phi_a, phi_b, mat4)
Routine to form the 4x4 1-turn coupled matrix with the reference point at the end of an element.

lat_make_mat6 (lat, ix_ele, ref_orb, ix_branch, err_flag)
Routine to make the 6x6 linear transfer matrix for an element

taylor_to_mat6 (a_taylor, r_in, vec0, mat6, r_out)
Routine to calculate the linear (Jacobian) matrix about some trajectory from a Taylor map.

transfer_mat2_from_twiss (twiss1, twiss2, mat)
Routine to make a 2 x 2 transfer matrix from the Twiss parameters at the end points.

transfer_mat_from_twiss (ele1, ele2, orb1, orb2, m)
Routine to make a 6 x 6 transfer matrix from the twiss parameters at the beginning and end of the element.

twiss_from_mat2 (mat_in, twiss, stat, type_out)
Routine to extract the Twiss parameters from the one-turn 2x2 matrix

twiss_from_mat6 (mat6, orb0, ele, stable, growth_rate, status, type_out)
Routine to extract the Twiss parameters from the one-turn 6x6 matrix

twiss_to_1_turn_mat (twiss, phi, mat2)
Routine to form the 2x2 1-turn transfer matrix from the Twiss parameters.

39.23 Matrix: Low Level Routines

Listed below are helper routines that are not meant for general use.

sol_quad_mat6_calc (ks_in, k1_in, length, ele, orbit, mat6, make_matrix)
Routine to calculate the transfer matrix for a combination solenoid/quadrupole element.

tilt_mat6 (mat6, tilt)
Routine to transform a 6x6 transfer matrix to a new reference frame that is tilted in (x, Px, y, Py) with respect to the old reference frame.

39.24 Measurement Simulation Routines

Routines to simulate errors in orbit, dispersion, betatron phase, and coupling measurements

check_if_ele_is_monitor (ele, err)
Routine to check that the element is either an instrument, monitor, or marker. This routine is private and not meant for general use.

to_eta_reading (eta_actual, ele, axis, reading, err)
Compute the measured dispersion reading given the true dispersion and the monitor offsets, noise, etc.

to_orbit_reading (orb, ele, axis, reading, err)
Calculate the measured reading on a bpm given the actual orbit and the BPM's offsets, noise, etc.

to_phase_and_coupling_reading (ele, reading, err)
Find the measured coupling values given the actual ones

39.25 Multipass

multipass_all_info (lat, info)
Routine to put multipass to a multipass_all_info_struct structure.

multipass_chain (ele, ix_pass, n_links, chain_ele)
Routine to return the chain of elements that represent the same physical element when there is multipass.

pointer_to_multipass_lord (ele, lat, ix_pass, super_lord) result (multi_lord)
Routine to find the multipass lord of a lattice element. A multi_lord will be found for:

39.26 Multipoles

ab_multipole_kick (a, b, n, ref_species, ele_orientation, coord, kx, ky, dk, pole_type, scale)

Routine to put in the kick due to an ab_multipole.

multipole_kicks (knl, tilt, ref_species, ele, orbit, pole_type, ref_orb_offset)
Routine to put in the kick due to a multipole.

mexp (x, m) result (this_exp)
Returns x**m with 0**0 = 0.

multipole_ab_to_kt (an, bn, knl, tn)
Routine to convert ab type multipoles to kt (MAD standard) multipoles.

multipole_ele_to_ab (ele, use_ele_tilt, ix_pole_max, a, b, pole_type, include_kicks)
Routine to put the scaled element multipole components (normal and skew) into 2 vectors.

multipole_ele_to_kt (ele, use_ele_tilt, ix_pole_max, knl, tilt, pole_type)
Routine to put the scaled element multipole components (strength and tilt) into 2 vectors.

multipole_init(ele, who, zero)
Routine to initialize the multipole arrays within an element.

multipole_kick (knl, tilt, n, ref_species, ele_orientation, coord, pole_type, ref_orb_offset)

Routine to put in the kick due to a multipole.

multipole_kt_to_ab (knl, tn, an, bn)
Routine to convert kt (MAD standard) multipoles to ab type multipoles.

39.27 Nonlinear Optimizers

opti_lmdif (vec, n, merit, eps) result(this_opti)
Function which tries to get the merit function(s) as close to zero as possible by changing the values in vec. Multiple merit functions can be used.

initial_lmdif()
Routine that clears out previous saved values of the optimizer.

suggest_lmdif (XV, FV, EPS, ITERMX, at_end, reset_flag)
Reverse communication routine.

super_mrqmin (y, weight, a,
chisq, funcs, storage, alamda, status, maska)
Routine to do non-linear optimizations. This routine is essentially mrqmin from Numerical Recipes with some added features.

opti_de (v_best, generations, population, merit_func, v_del, status)
Differential Evolution for Optimal Control Problems. This optimizer is based upon the work of Storn and Price.

39.28 Overloading the equal sign

These routines are overloaded by the equal sign so should not be called explicitly.

branch_equal_branch (branch1, branch2)
Routine that is used to set one branch equal to another.

bunch_equal_bunch (bunch1, bunch2)
Routine that is used to set one macroparticle bunch to another. This routine takes care of the pointers in bunch1.

coord_equal_coord (coord1, coord2)
Routine that is used to set one coord_struct equal to another.

ele_equal_ele (ele_out, ele_in)
Routine that is used to set one element equal to another. This routine takes care of the pointers in ele1.

lat_equal_lat (lat_out, lat_in)
Routine that is used to set one lat equal to another. This routine takes care of the pointers in lat1.

lat_vec_equal_lat_vec (lat1, lat2)
Routine that is used to set one lat array equal to another. This routine takes care of the pointers in lat1(:).

universal_equal_universal (ut1, ut2)
Routine that is used to set one PTC universal_taylor structure equal to another.

39.29 Particle Coordinate Stuff

angle_to_canonical_coords (orbit)
Routine to convert from angle (x, x', y, y', z, z') coordinates to canonical (x, px, y, py, z, pz) coordinates.

convert_coords (in_type_str, coord_in, ele, out_type_str, coord_out, err_flag)
Routine to convert between lab frame, normal mode, normalized normal mode, and action-angle coordinates.

convert_pc_to (pc, particle, E_tot, gamma, kinetic, beta, brho, beta1, err_flag)
Routine to calculate the energy, etc. from a particle's momentum.

convert_total_energy_to (E_tot, particle, gamma, kinetic, beta, pc, brho, beta1, err_flag)
Routine to calculate the momentum, etc. from a particle's total energy.

init_coord (orb, vec, ele, element_end, particle, direction, E_photon, t_offset, shift_vec6, spin)
Routine to initialize a coord_struct.

type_coord (coord)
Routine to type out a coordinate.

39.30 Photon Routines

bend_photon_init (g_bend_x, g_bend_y, gamma, orbit, E_min, E_max, E_integ_prob, vert_angle_min, vert_angle_max, vert_angle_symmetric, emit_probability)
Routine to initialize a photon generated by a charged particle in a bend.

bend_photon_vert_angle_init (E_rel, gamma, r_in, invert) result (r_in)
Routine to convert a "random" number in the interval [0,1] to a photon vertical emission angle for a simple bend.

bend_photon_energy_init (r_in) result (E_rel)
Routine to convert a random number in the interval [0,1] to a photon energy.

39.31 Interface to PTC

concat_real_8 (y1, y2, y3, r2_ref, keep_y1_const_terms)
Routine to concatenate two real_8 taylor series.

ele_to_fibre (ele, ptc_fibre, param, use_offsets, integ_order, steps, for_layout, track_particle, use_hard_edge_drifts, kill_layout)
Routine to convert a Bmad element to a PTC fibre element.

map_coef (y, i, j, k, l)
Function to return the coefficient of the map y(:) up to 3rd order.

kill_ptc_genfield (ptc_genfield)
Subroutine to kill a ptc_genfield.

kill_ptc_layouts (lat)
Routine to kill the layouts associated with a Bmad lattice.

kind_name (this_kind)
Function to return the name of a PTC kind.

normal_form_rd_terms(normal_form, rf_on, order)
Calculates driving terms à la [Bengt97] from the one-turn map.

real_8_to_taylor (y8, beta0, beta1, bmad_taylor)
Routine to convert from a real_8 taylor map in Étienne's PTC to a taylor map in Bmad.

real_8_init (y, set_taylor)
Routine to allocate a PTC real_8 variable.

remove_constant_taylor (taylor_in, taylor_out, c0, remove_higher_order_terms)
Routine to remove the constant part of a taylor series.

lat_to_ptc_layout (lat, use_hard_edge_drifts)
Routine to create a PTC layout from a Bmad lat.

set_ptc (e_tot, particle, taylor_order, integ_order, n_step, no_cavity, exact_modeling, exact_misalign, init_complex, force_init)
Routine to initialize PTC.

sort_universal_terms (ut_in, ut_sorted)
Routine to sort the taylor terms from "lowest" to "highest".

taylor_to_real_8 (bmad_taylor, beta0, beta1, ptc_re8, ref_orb_ptc, exi_orb_ptc)
Routine to convert from a taylor map in Bmad to a real_8 taylor map in Étienne's PTC.

type_layout (lay)
Routine to print the global information in a PTC layout.

type_map1 (y, type0, n_dim)
Routine to type the transfer map up to first order.

type_fibre (ptc_fibre, print_coords, lines, n_lines)
Routine to print the global information in a fibre.

type_map (y)
Routine to type the transfer maps of a real_8 array.

type_real_8_taylors (y)
Routine to type out the taylor series from a real_8 array.

taylor_to_genfield (bmad_taylor, ptc_genfield, c0)
Routine to construct a genfield (partially inverted map) from a taylor map.

universal_to_bmad_taylor (u_taylor, bmad_taylor)
Routine to convert from a universal_taylor map in Étienne's PTC to a taylor map in Bmad.

vec_bmad_to_ptc (vec_bmad, beta0, vec_ptc, conversion_mat)
Routine to convert from Bmad to PTC coordinates.

vec_ptc_to_bmad (vec_ptc, beta0, vec_bmad, conversion_mat, state)
Routine to convert from PTC to Bmad coordinates.

39.32 Quick Plot Routines

39.32.1 Quick Plot Page Routines

qp_open_page (page_type, i_chan, x_len, y_len, units, plot_file, scale)
Routine to Initialize a page (window) for plotting.

qp_select_page (iw)
Routine to switch to a particular page for drawing graphics.

qp_close_page()
Routine to finish plotting on a page.

39.32.2 Quick Plot Calculational Routines

qp_axis_niceness (imin, imax, divisions) result (score)
Routine to calculate how "nicely" an axis will look. The higher the score the nicer.

qp_calc_and_set_axis (axis_str, data_min, data_max, div_min, div_max, bounds, axis_type, slop_factor)
Routine to calculate a "nice" plot scale given the minimum and maximum of the data.

qp_calc_axis_params (data_min, data_max, div_min, div_max, axis, slop_factor)
Routine to calculate a "nice" plot scale given the minimum and maximum of the data. This is similar to calc_axis_scale.

qp_calc_axis_divisions (axis_min, axis_max, div_min, div_max, divisions)
Routine to calculate the best (gives the nicest looking drawing) number of major divisions for fixed axis minimum and maximum.

qp_calc_axis_places (axis)
Routine to calculate the number of decimal places needed to display the axis numbers.

qp_calc_axis_scale (data_min, data_max, axis, niceness_score, slop_factor)
Routine to calculate a "nice" plot scale given the minimum and maximum of the data.

qp_calc_minor_div (delta, div_max, divisions)
Routine to calculate the number of minor divisions an axis should have.

qp_convert_rectangle_rel (rect1, rect2)
Routine to convert a "rectangle" (structure of 4 points) from one set of relative units to another

39.32.3 Quick Plot Drawing Routines

qp_clear_box()
Routine to clear the current box on the page.

qp_clear_page()
Routine to clear all drawing from the page.

qp_draw_circle (x0, y0, r, angle0, del_angle, units, width, color, line_pattern, clip)
Routine to plot a section of an ellipse.

qp_draw_ellipse (x0, y0, r_x, r_y, theta_xy, angle0, del_angle, units, width, color, line_pattern, clip)
Routine to plot a section of an ellipse.

qp_draw_axes(x_lab, y_lab, title, draw_grid)
Routine to plot the axes, title, etc. of a plot.

qp_draw_data (x_dat, y_dat, draw_line, symbol_every, clip)
Routine to plot data, axes with labels, a grid, and a title.

qp_draw_graph (x_dat, y_dat, x_lab, y_lab, title, draw_line, symbol_every, clip)
Routine to plot data, axes with labels, a grid, and a title.

qp_draw_graph_title (title)
Routine to draw the title for a graph.

qp_draw_grid()
Routine to draw a grid on the current graph.

qp_draw_histogram (x_dat, y_dat, fill_color, fill_pattern, line_color, clip)
Routine to plot data, axes with labels, a grid, and a title.

qp_draw_curve_legend (x_origin, y_origin, units, line, line_length, symbol, text, text_offset, draw_line, draw_symbol, draw_text)
Routine to draw a legend with each line in the legend having a line, a symbol, some text.

qp_draw_text_legend (text, x_origin, y_origin, units)
Routine to draw a legend of lines of text.

qp_draw_main_title (lines, justify)
Routine to plot the main title at the top of the page.

qp_draw_polyline (x, y, units, width, color, line_pattern, clip, style)
Routine to draw a polyline.

qp_draw_polyline_no_set (x, y, units)
Routine to draw a polyline. This is similar to qp_draw_polyline except qp_set_line_attrib is not called.

qp_draw_polyline_basic (x, y)
Routine to draw a polyline. See also qp_draw_polyline

qp_draw_line (x1, x2, y1, y2, units, width, color, line_pattern, clip, style)
Routine to draw a line.

qp_draw_rectangle (x1, x2, y1, y2, units, color, width, line_pattern, clip, style)
Routine to draw a rectangular box.

qp_draw_symbol (x, y, units, type, height, color, fill_pattern, line_width, clip)
Draws a symbol at (x, y)

qp_draw_symbols (x, y, units, type, height, color,
fill_pattern, line_width, clip, symbol_every)
Draws a symbol at the (x, y) points.

qp_draw_text (text, x, y, units, justify, height, color, angle,
background, uniform_spacing, spacing_factor)
Routine to draw text.

qp_draw_text_no_set (text, x, y, units, justify, angle)
Routine to display on a plot a character string. See also: qp_draw_text.

qp_draw_text_basic (text, len_text, x0, y0, angle, justify)
Routine to display on a plot a character string. See also: qp_draw_text.

qp_draw_x_axis (who, y_pos)
Routine to draw a horizontal axis.

qp_draw_y_axis (who, x_pos)
Routine to draw a horizontal axis.

qp_paint_rectangle (x1, x2, y1, y2, units, color, fill_pattern)
Routine to paint a rectangular region a specified color. The default color is the background color (white$).

qp_to_axis_number_text (axis, ix_n, text)
Routine to form the text string for an axis number.

39.32.4 Quick Plot Set Routines

qp_calc_and_set_axis (axis, data_min, data_max,
div_min, div_max, bounds, axis_type, slop_factor)
Routine to calculate a "nice" plot scale given the minimum and maximum of the data.

qp_eliminate_xy_distortion(axis_to_scale)
This routine will increase the x or y margins so that the conversion between data units and page units is the same for the x and y axes.

qp_set_axis (axis_str, a_min, a_max, div, places, label, draw_label,
draw_numbers, minor_div, minor_div_max, mirror, number_offset,
label_offset, major_tick_len, minor_tick_len, ax_type)
Routine to set (but not plot) the min, max and divisions for the axes of the graph.

qp_set_box (ix, iy, ix_tot, iy_tot)
Routine to set the box on the physical page. This routine divides the page into a grid of boxes.

qp_set_graph (title)
Routine to set certain graph attributes.

qp_set_graph_limits()
Routine to calculate the offsets for the graph. This routine also sets the PGPLOT window size equal to the graph size.

qp_set_graph_placement (x1_marg, x_graph_len, y1_marg, y_graph_len, units)
Routine to set the placement of the current graph inside the box. This routine can be used in place of qp_set_margin.

qp_set_layout (x_axis, y_axis, x2_axis, y2_axis,
x2_mirrors_x, y2_mirrors_y, box, margin, page_border)
Routine to set various attributes. This routine can be used in place of other qp_set_* routines.

qp_set_line (who, line)
Routine to set the default line attributes.

qp_set_margin (x1_marg, x2_marg, y1_marg, y2_marg, units)
Routine to set up the margins from the sides of the box (see QP_SET_BOX) to the edges of the actual graph.

qp_set_page_border (x1_b, x2_b, y1_b, y2_b, units)
Routine to set the border around the physical page.

qp_set_page_border_to_box ()
Routine to set the page border to correspond to the region of the current box. This allows qp_set_box to subdivide the current box.

qp_set_clip (clip)
Routine to set the default clipping state.

qp_set_parameters (text_scale, default_draw_units, default_set_units,
default_axis_slop_factor)
Routine to set various quick plot parameters.

qp_subset_box (ix, iy, ix_tot, iy_tot, x_marg, y_marg)
Routine to set the box for a graph. This is the same as qp_set_box but the boundaries of the page are taken to be the box boundaries.

qp_set_symbol (symbol)
Routine to set the type and size of the symbols used in plotting data. See the pgplot documentation for more details.

qp_set_symbol_attrib (type, height, color, fill_pattern, line_width, clip)
Routine to set the type and size of the symbols used in plotting data.

qp_set_line_attrib (style, width, color, pattern, clip)
Routine to set the default line attributes.

qp_set_graph_attrib (draw_grid, draw_title)
Routine to set attributes of the current graph.

qp_set_text_attrib (who, height, color,
background, uniform_spacing, spacing_factor)
Routine to set the default text attributes.

qp_use_axis (x, y)
Routine to set what axis to use: X or X2, Y or Y2.

39.32.5 Informational Routines

qp_get_axis_attrib (axis_str, a_min, a_max, div, places, label, draw_label, draw_numbers, minor_div, mirror, number_offset, label_offset, major_tick_len, minor_tick_len, ax_type)
Routine to get the min, max, divisions etc. for the X and Y axes.

qp_get_layout_attrib (who, x1, x2, y1, y2, units)
Routine to get the attributes of the layout.

qp_get_line (style, line)
Routine to get the default line attributes.

qp_get_parameters (text_scale, default_draw_units, default_set_units, default_axis_slop_factor)
Routine to get various quick_plot parameters.

qp_get_symbol (symbol)
Routine to get the symbol parameters used in plotting data. Use qp_set_symbol or qp_set_symbol_attrib to set symbol attributes.

qp_text_len (text)
Function to find the length of a text string.

39.32.6 Conversion Routines

qp_from_inch_rel (x_inch, y_inch, x, y, units)
Routine to convert from a relative position (an offset) in inches to other units.

qp_from_inch_abs (x_inch, y_inch, x, y, units)
Routine to convert to absolute position (x, y) from inches referenced to the Left Bottom corner of the page

qp_text_height_to_inches(height_pt) result (height_inch)
Function to convert from a text height in points to a text height in inches taking into account the text_scale.

qp_to_inch_rel (x, y, x_inch, y_inch, units)
Routine to convert a relative (x, y) into inches.

qp_to_inch_abs (x, y, x_inch, y_inch, units)
Routine to convert an absolute position (x, y) into inches referenced to the Left Bottom corner of the page.

qp_to_inches_rel (x, y, x_inch, y_inch, units)
Routine to convert a relative (x, y) into inches.

qp_to_inches_abs (x, y, x_inch, y_inch, units)
Routine to convert an absolute position (x, y) into inches referenced to the left bottom corner of the page.

39.32.7 Miscellaneous Routines

qp_read_data (iu, err_flag, x, ix_col, y, iy_col, z, iz_col, t, it_col)
Routine to read columns of data.

39.32.8 Low Level Routines

qp_clear_box_basic (x1, x2, y1, y2)
Routine to clear all drawing from a box. That is, white out the box region.

qp_clear_page_basic()
Routine to clear all drawing from the page.

qp_close_page_basic()
Routine to finish plotting on a page. For X this closes the window.

qp_convert_point_rel (x_in, y_in, units_in, x_out, y_out, units_out)
Routine to convert a (x, y) point from from one set of relative units to another.

qp_convert_point_abs (x_in, y_in, units_in, x_out, y_out, units_out)
Routine to convert a (x, y) point from from one set of absolute units to another.

qp_draw_symbol_basic (x, y, symbol)
Routine to draw a symbol.

qp_init_com_struct ()
Routine to initialize the common block qp_state_struct. This routine is not for general use.

qp_join_units_string (u_type, region, corner, units)
Routine to form a units from its components.

qp_justify (justify)
Function to convert a justify character string to a real value representing the horizontal justification.

qp_open_page_basic (page_type, x_len, y_len, plot_file,
x_page, y_page, i_chan, page_scale)
Routine to Initialize a page (window) for plotting.

qp_paint_rectangle_basic (x1, x2, y1, y2, color, fill_pattern)
Routine to fill a rectangle with a given color. A color of white essentially erases the rectangle.

qp_pointer_to_axis (axis_str, axis_ptr)
Routine to return a pointer to an common block axis.

qp_restore_state()
Routine to restore saved attributes. Use qp_save_state to restore the saved state.

qp_restore_state_basic (buffer_basic)
Routine to restore the print state.

qp_save_state (buffer_basic)
Routine to save the current attributes. Use qp_restore_state to restore the saved state.

qp_save_state_basic ()
Routine to save the print state.

qp_select_page_basic (iw)
Routine to switch to a particular page for drawing graphics.

qp_set_char_size_basic (height)
Routine to set the character size.

qp_set_clip_basic (clip)
Routine to set the clipping state. Note: This affects both lines and symbols.

qp_set_color_basic (ix_color)
Routine to set the color taking into account that GIF inverts the black for white.

qp_set_graph_position_basic (x1, x2, y1, y2)
Routine to set the position of a graph. Units are inches from lower left of page.

qp_set_line_width_basic (line_width)
Routine to set the line width.

qp_set_symbol_fill_basic (fill)
Routine to set the symbol fill style.

qp_set_symbol_size_basic (height, symbol_type, uniform_size)
Routine to set the symbol_size

qp_set_text_background_color_basic (color)
Routine to set the character text background color.

qp_split_units_string (u_type, region, corner, units)
Routine to split a units string into its components.

qp_text_len_basic (text)
Function to find the length of a text string.

qp_translate_to_color_index (name)
Routine to translate from a string to a color index.

39.33 Spin Tracking

spinor_to_polar (spinor) result (polar)
Routine to convert a spinor into polar coordinates.

polar_to_vec (polar) result (vec)
Routine to convert a spin vector from polar coordinates to Cartesian coordinates.

polar_to_spinor (polar) result (coord)
Routine to convert a spin vector in polar coordinates to a spinor.

vec_to_polar (vec, phase) result (polar)
Routine to convert a spin vector from Cartesian coordinates to polar coordinates preserving the complex phase.

spinor_to_vec (spinor) result (vec)
Routine to convert a spinor to a spin vector in Cartesian coordinates.

vec_to_spinor (vec, phase) result (coord)
Routine to convert a spin vector in Cartesian coordinates to a spinor using the specified complex phase.

angle_between_polars (polar1, polar2)
Function to return the angle between two spin vectors in polar coordinates.

spin_omega (field, orbit, sign_z_vel, phase_space_coords), result (omega)
Return the modified T-BMT spin omega vector.

track1_spin (start_orb, ele, param, end_orb, make_quaternion)
Routine to track the particle spin through one element.

39.34 Transfer Maps: Routines Called by make_mat6

`Make_mat6` is the routine for calculating the transfer matrix (Jacobin) through an element. The routines listed below are used by `make_mat6`. In general a program should call `make_mat6` rather than using these routines directly.

make_mat6_bmad (ele, param, start_orb, end_orb, err)
Routine to make the 6x6 transfer matrix for an element using closed formulas.

make_mat6_custom (ele, param, c0, c1, err_flag)
Routine for custom calculations of the 6x6 transfer matrices.

make_mat6_symp_lie_ptc (ele, param, start_orb, end_orb)
Routine to make the 6x6 transfer matrix for an element using the PTC symplectic integrator.

make_mat6_taylor (ele, param, start_orb, end_orb, err_flag)
Routine to make the 6x6 transfer matrix for an element from a Taylor map.

make_mat6_tracking (ele, param, start_orb, end_orb, err_flag)
Routine to make the 6x6 transfer matrix for an element by tracking 7 particle with different starting conditions.

39.35 Transfer Maps: Complex Taylor Maps

add_complex_taylor_term (bmad_complex_taylor, coef, exp)
Subroutine add_complex_taylor_term (bmad_complex_taylor, coef, i1, i2, i3, i4, i5, i6, i7, i8, i9) Routine to add a complex_taylor term to a complex_taylor series.

complex_taylor_coef (bmad_taylor, expn)
Function complex_taylor_coef (bmad_complex_taylor, i1, i2, i3, i4, i5, i6, i7, i8, i9) Function to return the coefficient for a particular complex_taylor term from a complex_taylor Series.

complex_taylor_equal_complex_taylor (complex_taylor1, complex_taylor2)
Subroutine that is used to set one complex_taylor equal to another. This routine takes care of the pointers in complex_taylor1.

complex_taylor_make_unit (complex_taylor)
Subroutine to make the unit complex_taylor map: r(out) = Map * r(in) = r(in)

complex_taylor_exponent_index(expn) result(index)
Function to associate a unique number with a complex_taylor exponent.

complex_taylor_to_mat6 (a_complex_taylor, r_in, vec0, mat6, r_out)
Subroutine to calculate, from a complex_taylor map and about some trajectory: The 1st order (Jacobian) transfer matrix.

complex_taylors_equal_complex_taylors (complex_taylor1, complex_taylor2)
Subroutine to transfer the values from one complex_taylor map to another: complex_taylor1 <= complex_taylor2

init_complex_taylor_series (complex_taylor, n_term, save)
Subroutine to initialize a Bmad complex_taylor series (6 of these series make a complex_taylor map). Note: This routine does not zero the structure. The calling

kill_complex_taylor (complex_taylor)
Subroutine to deallocate a Bmad complex_taylor map.

mat6_to_complex_taylor (vec0, mat6, complex_taylor)
Subroutine to form a first order complex_taylor map from the 6x6 transfer matrix and the 0th order transfer vector.

sort_complex_taylor_terms (complex_taylor_in, complex_taylor_sorted)
Subroutine to sort the complex_taylor terms from "lowest" to "highest" of a complex_taylor series.

track_complex_taylor (start_orb, complex_taylor, end_orb)
Subroutine to track using a complex_taylor map.

truncate_complex_taylor_to_order (complex_taylor_in, order, complex_taylor_out)
Subroutine to throw out all terms in a complex_taylor map that are above a certain order.

type_complex_taylors (complex_taylor, max_order, lines, n_lines, file_id, out_type, clean)

Subroutine to output a Bmad complex_taylor map.

39.36 Transfer Maps: Taylor Maps

add_taylor_term (bmad_taylor, coef, expn, replace)
add_taylor_term (bmad_taylor, coef, i1, i2, i3, i4, i5, i6, i7, i8, i9, replace)
Overloaded routine to add a Taylor term to a Taylor series.

concat_ele_taylor (taylor1, ele, taylor3)
Routine to concatenate two taylor maps.

concat_taylor (taylor1, taylor2, taylor3)
Routine to concatenate two taylor series: taylor3(x) = taylor2(taylor1(x))

ele_to_taylor (ele, param, orb0, taylor_map_includes_offsets, orbital_taylor, spin_taylor)

Routine to make a Taylor map for an element. The order of the map is set by set_ptc.

equivalent_taylor_attributes (ele1, ele2) result (equiv)
Routine to see if to elements are equivalent in terms of attributes so that their Taylor Maps would be the same.

init_taylor_series (bmad_taylor, n_term, save_old)
Routine to initialize a Bmad Taylor series.

kill_taylor (bmad_taylor)
Routine to deallocate a Bmad Taylor map.

mat6_to_taylor (mat6, vec0, bmad_taylor)
Routine to form a first order Taylor map from the 6x6 transfer matrix and the 0th order transfer vector.

sort_taylor_terms (taylor_in, taylor_sorted, min_val)
Routine to sort the taylor terms from "lowest" to "highest" of a Taylor series.

taylor_coef (bmad_taylor, expn)
Function to return the coefficient for a particular taylor term from a Taylor Series.

taylor_equal_taylor (taylor1, taylor2)
Routine to transfer the values from one taylor map to another: Taylor1 $\leq$ Taylor2

transfer_map_calc (lat, t_map, err_flag, ix1, ix2, ref_orb, ix_branch, one_turn, unit_start, concat_if_possible)
Routine to calculate the transfer map between two elements.

transfer_map_from_s_to_s (lat, t_map, s1, s2, ref_orb, ix_branch, one_turn, unit_start, err_flag, concat_if_possible)
Subroutine to calculate the transfer map between longitudinal positions s1 to s2.

taylor_minus_taylor (taylor1, taylor2) result (taylor3)
Routine to add two taylor maps.

taylor_plus_taylor (taylor1, taylor2) result (taylor3)
Routine to add two taylor maps.

taylors_equal_taylors (taylor1, taylor2)
Routine to transfer the values from one taylor map to another.

taylor_make_unit (bmad_taylor, ref_orbit)
Routine to make the unit Taylor map

taylor_to_mat6 (a_taylor, c0, mat6, c1)
Routine to calculate the linear (Jacobian) matrix about some trajectory from a Taylor map.

taylor_inverse (taylor_in, taylor_inv, err)
Routine to invert a taylor map.

taylor_propagate1 (bmad_taylor, ele, param, track_particle)
Routine to track a real_8 taylor map through an element. The alternative routine, if ele has a taylor series, is concat_taylor.

track_taylor (start_orb, bmad_taylor, ref_orb)
Routine to track using a Taylor map.

transfer_ele_taylor (ele_in, ele_out, taylor_order)
Routine to transfer a Taylor map from one element to another.

truncate_taylor_to_order (taylor_in, order, taylor_out)
Routine to throw out all terms in a taylor map that are above a certain order.

type_taylors (bmad_taylor, max_order, lines, n_lines, file_id, out_type, clean)
Routine to output a Bmad taylor map.

39.37 Transfer Maps: Driving Terms

srdt_calc (lat, srdt_sums, order, n_slices_gen_opt, n_slices_sxt_opt, per_ele_out)
Calculates driving terms using summations over sextupole moments like those in [Bengt97] and [Wang12]. Often called resonance driving terms (RDTs), though strictly speaking not all terms drive resonances. Terms that are first and second order in sextupole moment are included. See srdt_mod for a list of available driving terms.

make_srdt_cache(lat,n_slices_gen,n_slices_sxt,cache)
Used to speed up calculation of the 2nd order driving terms. Makes an $N_{k2} \times N_{k2} \times 11$ array that contains the precomputed cross-products of the linear optics at each sextupole moment.

srdt_calc_with_cache(lat, srdt_sums, order, n_slices_gen_opt, n_slices_sxt_opt, cache, per_ele_out)
Same as 39.37, but makes use of a cache precomputed by 39.37 to speed up the calculations.

srdt_lsq_solution(lat, var_indexes, ls_soln, n_slices_gen_opt, n_slices_sxt_opt, chrom_set_x_opt, chrom_set_y_opt)
Given a lattice and the indices of the sextupole moments to use as variables, finds the least squares solution to the sextupole moments that minimize the 1st order driving terms. If there are more variables than driving terms, then the solution sets the driving terms to zero and minimizes the sum of the squares of the variables. If there are fewer variables than driving terms, then the solution is that which minimizes the sum of the squares of the driving terms.

39.38 Tracking and Closed Orbit

The following routines perform tracking and closed orbit calculations.

check_aperture_limit (orb, ele, particle_at, param, old_orb, check_momentum)
Routine to check if an orbit is outside an element's aperture.

check_aperture_limit_custom (orb, ele, particle_at, param, err_flag)
Routine to check if an orbit is outside an element's aperture. Used when `ele%aperture_type` is set to `custom$`

closed_orbit_calc (lat, closed_orb, i_dim, direction, ix_branch, err_flag, print_err)
Routine to calculate the closed orbit at the beginning of the lat.

closed_orbit_from_tracking (lat, closed_orb, i_dim, eps_rel, eps_abs, init_guess, err_flag)
Routine to find the closed orbit via tracking.

compute_even_steps (ds_in, length, ds_default, ds_out, n_step)
Routine to compute a step size ds_out, close to ds_in, so that an integer number of steps spans the length.

dynamic_aperture1 (lat, orb0, theta_xy, Sxy, aperture_param, aperture, check_xy_init)

Routine to determine the dynamic aperture of a lattice via tracking.

multi_turn_tracking_analysis (track, i_dim, track0, ele, stable, growth_rate, chi, err_flag)
Routine to analyze multi-turn tracking data to get the Twiss parameters etc.

multi_turn_tracking_to_mat (track, i_dim, mat1, track0, chi)
Routine to analyze 1-turn tracking data to find the 1-turn transfer matrix and the closed orbit offset.

offset_particle (ele, param, set, orbit, set_tilt, set_hvkicks, drift_to_edge, s_pos, s_out, set_spin, mat6, make_matrix)
Routine to effectively offset an element by instead offsetting the particle position to correspond to the local element coordinates.

offset_photon (ele, orbit, set, offset_position_only, rot_mat)
Routine to effectively offset an element by instead offsetting the photon position to correspond to the local crystal or mirror coordinates.

orbit_amplitude_calc (ele, orb, amp_a, amp_b, amp_na, amp_nb)
Routine to calculate the "invariant" amplitude of a particle at a particular point in its orbit.

particle_is_moving_backwards (orbit) result (is_moving_backward)
Routine to determine if a particle is moving in the backward -s direction. If not moving backward it is dead or is moving backward.

particle_is_moving_forward (orbit) result (is_moving_forward)
Routine to determine if a particle is moving in the forward +s direction. If not moving forward it is dead or is moving backward.

tilt_coords (tilt_val, coord, mat6, make_matrix)
Routine to effectively tilt (rotate in the x-y plane) an element by instead rotating the particle position with negative the angle.

track1 (start_orb, ele, param, end_orb, track, err_flag, ignore_radiation, mat6, make_matrix)

Routine to track through a single element.

track1_bunch_csr (bunch_start, ele, centroid, bunch_end, err, s_start, s_end)
Routine to track a bunch of particles through the element lat%ele(ix_ele) with csr radiation effects.

track1_spin_custom (start, ele, param, end, err_flag, track, make_quaternion)
Dummy routine for custom spin tracking. This routine needs to be replaced for a custom calculation.

track_all (lat, orbit, ix_branch, track_state, err_flag, orbit0)
Routine to track through the lat.

track_from_s_to_s (lat, s_start, s_end, orbit_start, orbit_end, all_orb, ix_branch, track_state)
Routine to track a particle between two s-positions.

track_many (lat, orbit, ix_start, ix_end, direction, ix_branch, track_state)
Routine to track from one element in the lat to another.

twiss_and_track (lat, orb, ok)
twiss_and_track (lat, orb_array, ok)
Routine to calculate the twiss parameters, transport matrices and orbit.

twiss_and_track_at_s (lat, s, ele_at_s, orb, orb_at_s, ix_branch, err, use_last, compute_floor_coords)
Routine to calculate the Twiss parameters and orbit at a particular longitudinal position.

twiss_and_track_from_s_to_s (branch, orbit_start, s_end, orbit_end, ele_start, ele_end, err, compute_floor_coords)
Routine to track a particle from one location to another.

twiss_and_track_intra_ele (ele, param, l_start, l_end, track_upstream_end, track_downstream_end, orbit_start, orbit_end, ele_start, ele_end, err, compute_floor_coords)
Routine to track a particle within an element.

twiss_from_tracking (lat, ref_orb0, symp_err, err_flag, d_orb)
Routine to compute from tracking the Twiss parameters and the transfer matrices for every element in the lat.

wall_hit_handler_custom (orb, ele, s)
This routine is called by the Runge-Kutta integrator odeint_bmad when a particle hits a wall.

39.39 Tracking: Low Level Routines

absolute_time_tracking (ele) result (is_abs_time)
Routine to return a logical indicating whether the tracking through an element should use absolute time or time relative to the reference particle.

odeint_bmad (orbit, ele, param, s1_body, s2_body, err_flag, track)
Routine to do Runge Kutta tracking.

create_uniform_element_slice (ele, param, i_slice, n_slice_tot, sliced_ele, s_start, s_end)
Routine to create an element that represents a slice of another element. This routine can be used for detailed tracking through an element.

track_a_drift (orb, length, mat6, make_matrix, include_ref_motion)
Routine to track through a drift.

track_a_bend (orbit, ele, param, mat6, make_matrix)
Particle tracking through a bend element.

39.40 Tracking: Mad Routines

make_mat6_mad (ele, param, c0, c1)
Routine to make the 6x6 transfer matrix for an element from the 2nd order MAD transport map. The map is stored in ele%taylor.

make_mad_map (ele, param, energy, map)
Routine to make a 2nd order transport map a la MAD.

mad_add_offsets_and_multipoles (ele, map)
Routine to add in the effect of element offsets and/or multipoles on the 2nd order transport map for the element.

mad_drift (ele, energy, map)
Routine to make a transport map for a drift space. The equivalent MAD-8 routine is: TMDRF

mad_elsep (ele, energy, map)
Routine to make a transport map for an electric separator. The equivalent MAD-8 routine is: TMSEP

mad_sextupole (ele, energy, map)
Routine to make a transport map for an sextupole. The equivalent MAD-8 routine is: TMSEXT

mad_sbend (ele, energy, map)
Routine to make a transport map for a sector bend element. The equivalent MAD-8 routine is: TMBEND

mad_sbend_fringe (ele, energy, into, map)
Routine to make a transport map for the fringe field of a dipole. The equivalent MAD-8 routine is: TMFRNG

mad_sbend_body (ele, energy, map)
Routine to make a transport map for the body of a sector dipole. The equivalent MAD-8 routine is: TMSECT

mad_tmfoc (el, sk1, c, s, d, f)
Routine to compute the linear focusing functions. The equivalent MAD-8 routine is: TMFOC

mad_quadrupole (ele, energy, map)
Routine to make a transport map for an quadrupole element. The equivalent MAD-8 routine is: TMSEXT

mad_rfcavity (ele, energy, map)
Routine to make a transport map for an rfcavity element. The equivalent MAD-8 routine is: TMRF

mad_solenoid (ele, energy, map)
Routine to make a transport map for an solenoid. The equivalent MAD-8 routine is: TMSEXT

mad_tmsymm (te)
routine to symmetrize the 2nd order map t. The equivalent MAD-8 routine is: tmsymm

mad_tmtilt (map, tilt)
Routine to apply a tilt to a transport map. The equivalent MAD-8 routine is: TMTILT

mad_concat_map2 (map1, map2, map3)
Routine to concatenate two 2nd order transport maps.

mad_track1 (c0, map, c1)
Routine to track through a 2nd order transfer map. The equivalent MAD-8 routine is: TMTRAK

track1_mad (start_orb, ele, param, end_orb)
Routine to track through an element using a 2nd order transfer map. Note: If map does not exist then one will be created.

mad_map_to_taylor (map, energy, taylor)
Routine to convert a mad order 2 map to a taylor map.

taylor_to_mad_map (taylor, energy, map)
Routine to convert a Taylor map to a mad order 2 map. If any of the Taylor terms have order greater than 2 they are ignored.

make_unit_mad_map (map)
Routine to initialize a 2nd order transport map to unity.

39.41 Tracking: Routines called by track1

Note: Unless you know what you are doing do not call these routines directly. Rather use track1.

symp_lie_bmad (ele, param, start_orb, end_orb, make_matrix, track, offset_ele)
Symplectic integration through an element to 0th or 1st order.

track1_bmad (start_orb, ele, param, end_orb, err_flag, mat6, make_matrix)
Particle tracking through a single element BMAD_standard style.

track1_custom (start_orb, ele, param, end_orb, err_flag, finished, track)
Dummy routine for custom tracking.

track1_linear (start_orb, ele, param, end_orb)
Particle tracking through a single element using the transfer matrix..

track1_postprocess (start_orb, ele, param, end_orb)
Dummy routine for post processing after the track1 routine is done.

track1_preprocess (start_orb, ele, param, err_flag, finished, radiation_included, track)
Dummy routine for pre processing at the start of the track1 routine.

track1_radiation (orbit, ele, param, edge)
Routine to put in radiation damping and/or fluctuations.

track1_runge_kutta (start_orb, ele, param, end_orb, err_flag, track)
Routine to do tracking using Runge-Kutta integration.

track1_symp_lie_ptc (start_orb, ele, param, end_orb, track)
Particle tracking through a single element using a Hamiltonian and a symplectic integrator.

track1_symp_map (start_orb, ele, param, end_orb)
Particle tracking through a single element using a partially inverted taylor map (In PTC/FPP this is called a genfield).

track1_taylor (start_orb, ele, param, end_orb, taylor, mat6, make_matrix)
Routine to track through an element using the elements taylor series.

track1_time_runge_kutta(start_orb, ele, param, end_orb, err_flag, track)
Routine to track a particle through an element using Runge-Kutta time-based tracking.

39.42 Twiss and Other Calculations

calc_z_tune (lat, ix_branch)
Routine to calculate the synchrotron tune from the full 6X6 1 turn matrix.

chrom_calc (lat, delta_e, chrom_x, chrom_y, err_flag, pz,
low_E_lat, high_E_lat, low_E_orb, high_E_orb, ix_branch)
Routine to calculate the chromaticities by computing the tune change when then energy is changed.

chrom_tune (lat, delta_e, target_x, target_y, err_tol, err_flag)
Routine to set the sextupole strengths so that the lat has the desired chromaticities.

quad_beta_ave (ele, beta_a_ave, beta_b_ave)
Routine to compute the average betas in a quad.

radiation_integrals (lat, orbit, mode, ix_cache, ix_branch, rad_int_by_ele)
Routine to calculate the synchrotron radiation integrals, the emittance, and energy spread.

radiation_integrals_custom (lat, ir, orb, err_flag)
User supplied routine to calculate the synchrotron radiation integrals for a custom element.

relative_mode_flip (ele1, ele2)
Function to see if the modes of ELE1 are flipped relative to ELE2.

set_tune (phi_a_set, phi_b_set, dk1, lat, orb, ok)
Routine to Q_tune a lat. This routine will set the tunes to within 0.001 radian (0.06 deg).

set_z_tune (lat, z_tune, ok)
Routine to set the longitudinal tune by setting the RF voltages in the RF cavities.

transfer_twiss (ele_in, ele_out, reverse)
Routine to transfer the twiss parameters from one element to another.

twiss_and_track (lat, orb)
Routine to calculate the Twiss and orbit parameters. This is not necessarily the fastest routine.

twiss_at_element (ele, start, end, average)
Routine to return the Twiss parameters at the beginning, end, or the average of an element.

twiss_and_track_at_s (lat, s, ele, orb_, here)
Routine to calculate the Twiss parameters and orbit at a particular longitudinal position.

twiss_at_start (lat, status, ix_branch)
Routine to calculate the Twiss parameters at the start of the lat.

twiss_from_tracking (lat, closed_orb_, d_orb, error)
Routine to compute from tracking, for every element in the lat, the Twiss parameters and the transfer matrices.

twiss_propagate1 (ele1, ele2, err_flag)
Routine to propagate the Twiss parameters from the end of ELE1 to the end of ELE2.

twiss_propagate_all (lat, ix_branch, err_flag, ie_start, ie_end, zero_uncalculated)
Routine to propagate the Twiss parameters from the start to the end.

twiss_to_1_turn_mat (twiss, phi, mat2)
Routine to form the 2x2 1-turn transfer matrix from the Twiss parameters.

39.43 Twiss: 6 Dimensional

normal_mode3_calc (t6, tune, B, HV, above_transition)
Decompose a 2n x 2n symplectic matrix into normal modes. For more details see:

twiss3_propagate_all (lat, ix_branch)
Routine to propagate the twiss parameters using all three normal modes.

twiss3_propagate1 (ele1, ele2, err_flag)
Routine to propagate the twiss parameters using all three normal modes.

twiss3_at_start (lat, err_flag, ix_branch, tune3)
Routine to propagate the twiss parameters using all three normal modes.

39.44 Wake Fields

init_wake (wake, n_sr_long, n_sr_trans, n_lr_mode, n_lr_spline, always_allocate)
Routine to initialize a wake struct.

randomize_lr_wake_frequencies (ele, set_done)
Routine to randomize the frequencies of the lr wake HOMs.

sr_long_wake_particle (ele, orbit)
Subroutine to apply the short-range wake kick to a particle and then add to the existing short-range wake the contribution from the particle.

sr_trans_wake_particle (ele, orbit)
Subroutine to apply the short-range wake kick to a particle and then add to the existing short-range wake the contribution from the particle.

track1_sr_wake (bunch, ele)
Routine to apply the short range wake fields to a bunch.

track1_lr_wake (bunch, ele)
Routine to put in the long-range wakes for particle tracking.

zero_lr_wakes_in_lat (lat)
Routine to zero the long range wake amplitudes for the elements that have long range wakes in a lattice.

39.45 C/C++ Interface

fscalar2scalar (f_scalar, n) result (c_scalar)
Function to translate a scalar from Fortran form to C form.

fvec2vec (f_vec, n) result (c_vec)
Function to translate a vector from Fortran form to C form.

mat2vec (mat, n) result (vec)
Function to take a matrix and turn it into an array in C standard row-major order.

tensor2vec (tensor, n) result (vec)
Function to take a tensor and turn it into an array in C standard row-major order::

vec2mat (vec, mat)
Routine to take a an array in C standard row-major order and turn it into a matrix.

vec2tensor (vec, tensor)
Routine to take a an array in C standard row-major order and turn it into a tensor.

remove_null_in_string (str_in, str_out
Routine to convert a null character in a string to a blank.

f_logic (logic) result (f_log)
Function to convert from a C logical to a Fortran logical.

f_logic_int (logic) result (f_log)
Function to convert from a C logical to a Fortran logical. This function is overloaded by f_logic.

f_logic_bool (logic) result (f_log)
Function to convert from a C logical to a Fortran logical. This function is overloaded by f_logic.

remove_null_in_string_arr (str_in, str_out)
This routine overloaded by: remove_null_in_string

remove_null_in_string_char (str_in, str_out)
This routine overloaded by: remove_null_in_string

to_c_str (f_string, c_string)
Subroutine to append a null (0) character at the end of a string (trimmed of trailing blanks) so it will look like a C character array.

to_f_str (c_string, f_string)
Subroutine to append a null (0) character at the end of a string (trimmed of trailing blanks) so it will look like a C character array.

Part IV

Bibliography and Index

Bibliography

[Abell06] Dan Abell, "Numerical computation of high-order transfer maps for rf cavities", Phys. Rev. ST Accel. Beams, **9**, pp. 052001, (2006).

[Akima70] H. Akima, "A New Method of Interpolation and Smooth Curve Fitting Based on Local Procedures," J. Assoc. Comp. Mach., **17** pp. 589-602 (1970).

[AML] The Accelerator Markup Language / Universal Accelerator Project web page:
https://www.classe.cornell.edu/~dcs/aml/

[Barber99] D. P. Barber and G. Ripken, "Computer Algorithms and Spin Matching," *Handbook of Accelerator Physics and Engineering,* 3rd Edition, World Scientific Pub. Co. Inc. (1999).

[Bater64] B. Batterman, and H. Cole, "Dynamical Diffraction of X Rays by Perfect Crystals", Rev. Mod. Phys.,**36**, 3, pp. 681–717, (1964).

[Berz89] M. Berz, "Differential Algebraic Description of Beam Dynamics to Very High Orders," Particle Accelerators, Vol. 24, pp. 109-124, (1989).

[Bengt97] J. Bengtsson, "The Sextupole Scheme for the Swiss Light Source (SLS): An Analytic Approach," SLS Note 9/97, Paul Scherrer Institut, (1997).

[Wang12] C. Wang, "Explicit formulas for 2nd-order driving terms due to sextupoles and chromatic effects of quadrupoles," ANL/APS/LS-330, Argonne National Laboratory, (2012).

[Blas94] R. C. Blasdell and A. T. Macrander, "Modifications to the 1989 SHADOW ray-tracing code for general asymmetric perfect-crystal optics," Nuc. Instr. & Meth. A **347**, 320 (1994).

[Bmad] The Bmad web site:
https://www.classe.cornell.edu/bmad

[Brown77] K. L. Brown, F. Rothacker, D. C. Carey, and Ch. Iselin, "TRANSPORT Appendix," Fermilab, unpublished, (December 1977).

[Chao79] Alexander W. Chao, "Evaluation of beam distribution parameters in an electron storage ring," Journal of Applied Physics **50**, 595 (1979).

[Chao93] Alexander Chao, *Physics of Collective Beam Instabilities in High Energy Accelerators*, Wiley, New York (1993).

[Corbett99] J. Corbett and Y. Nosochkov, "Effect of Insertion Devices in SPEAR–3," Proc. 1999 Part. Acc. Conf., p. 238, (1999).

[Decking00] W. Decking, R. Brinkmann, "Space Charge Problems in the TESLA Damping Ring", Europ. Accel. Phys. Conf., Vienna, (2000).

[Duff87] J. Le Duff, *Single and Multiple Touschek Effects*. Proc. CAS Berlin 1987, CERN 89-01, (1987).

[Duan15] Zhe Duan, Mei Bai, Desmond P. Barber, Qing Qin, "A Monte-Carlo simulation of the equilibrium beam polarization in ultra-high energy electron (positron) storage rings," Nuc. Instrum. and methods in Phys. Research A **793**, pp 81-91, (2015).

[Forest02] É. Forest, F. Schmidt, E. McIntosh, *Introduction to the Polymorphic Tracking Code*, CERN-SL-2002-044 (AP), and KEK-Report 2002-3 (2002). Can be obtained at:
https://frs.web.cern.ch/frs/report/sl-2002-044.pdf

[Forest06] Étienne Forest, 'Geometric integration for particle accelerators," J. Phys. A: Math. Gen. **39** (2006) 5321âĂŞ5377.

[Forest88] É. Forest, J. Milutinovic, "Leading Order Hard Edge Fringe Fields Effects Exact in $(1+\delta)$ and Consistent with Maxwell's Equations for Rectilinear Magnets," Nuc. Instrum. and Methods in Phys. Research A **269**, pp 474-482, (1988).

[Forest98] É. Forest, *Beam Dynamics: A New Attitude and Framework*, Harwood Academic Publishers, Amsterdam (1998).

[Grote96] H. Grote, F. C. Iselin, *The MAD Program User's Reference Manual*, Version 8.19, CERN/SL/90-13 (AP) (REV. 5) (1996). Can be obtained at:
https://mad.web.cern.ch/mad

[Healy86] L. M. Healy, *Lie Algebraic Methods for Treating Lattice Parameter Errors in Particle Accelerators*. Doctoral thesis, University of Maryland, unpublished, (1986).

[Helm73] R. H. Helm, M. J. Lee, P. L. Morton, and M. Sands, "Evaluation of Synchrotron Radiation Integrals," IEEE Trans. Nucl. Sci. NS-20, 900 (1973).

[Hoff06] G. Hoffstaetter, *Hight-Energy Polarized Proton Beams, A Modern View*, Springer. Springer Tracks in Modern Physics Vol 218, (2006).

[Hwang15] Hwang and S. Y. Lee, "Dipole Fringe Field Thin Map for Compact Synchrotrons", Phys. Rev. ST Accel. Beams, **12**, 122401, (2015).

[Iselin94] F. C. Iselin, *The MAD program Physical Methods Manual*, unpublished, (1994). Can be obtained at:
https://mad.home.cern.ch/mad

[Jowett87] J. M. Jowett, "Introductory Statistical Mechanics for Electron Storage Rings," AIP Conf. Proc. 153, Physics of Part. Acc., M. Month and M. Dienes Eds., pp. 864, (1987).

[Kohn95] V. G. Kohn, "On the Thcory of Reflectivitlby an X-Ray Multilaler Mirror" physica status solidi (b), **187**, 61, (1995).

[Lee99] S. Y. Lee, *Accelerator Physics, World Scientific, Singapore, (1999).*

[McMill75] *E. M. McMillan, "Multipoles in Cylindrical Coordinates," Nucl. Instrum. Meth. 127, 471 (1975).*

[Newton99] *D. Newton and A. Wolski, "Fast, Accurate Calculation of Dynamical Maps from Magnetic Field Data Using Generalised Gradients," Proc. PAC09 (2009).*

[Press92] *W. Press, B. Flannery, S. Teukolsky, and W. Wetterling,* Numerical Recipes in Fortran, the Art of Scientific Computing*, Second Edition, Cambridge University Press, New York, (1992).*
W. Press, B. Flannery, S. Teukolsky, and W. Wetterling, Numerical Recipes in Fortran90, the Art of Parallel Scientific Computing*, Cambridge University Press, New York, (1996).*

[Piwin98] *Anton Piwinski,* The Touschek Effect in Strong Focusing Storage Rings*. DESY Report 98-179, 1998.*

[Quat] *Wikipedia article, "Quaternions and spatial rotation",*
https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation.

[Rauben91] *T. Raubenheimer, "Tolerances to Limit the Vertical Emittance in Future Storage Rings", Particle Accelerators, 1991,* **36***, pp.75-119. SLAC-PUB-4937 Rev., (1991).*

[Rio98] *Manuel Sanchez del Rio, "Ray tracing simulations for crystal optics," Proc. SPIE 3448, Crystal and Multilayer Optics,* **230** *(1998).*

[Rosen94] *J. Rosenzweig and L. Serafini, "Transverse Particle Motion in Radio–Frequency Linear Accelerators," Phys Rev E, Vol. 49, p. 1599, (1994).*

[Ruth87] *R. D. Ruth, "Single-Particle Dynamics in Circular Accelerators," in AIP Conference Proceedings* **153**, Physics of Particle Accelerators*, pp. 152–235, M. Month and M. Dienes editors, American Institute of Physics, New York (1987).*

[Ryne18] *C. E. Mayes, R. D. Ryne, D. C. Sagan, "3D SPACE CHARGE IN BMAD", 9th Int. Part. Accel. Conf., Vancouver, pp. 3428–3430, (2018). Software available at:* https://github.com/RobertRyne/OpenSpaceCharge.

[SAD] *D. Zhou and K. Oide, "Maps Used in SAD" (unpublished). Also see:*
https://acc-physics.kek.jp/SAD/

[Sagan03] *D. Sagan, J. Crittenden, and D. Rubin. "A Symplectic Model for Wigglers," Part. Acc. Conf. (2003).*

[Sagan99] *D. Sagan and D. Rubin "Linear Analysis of Coupled Lattices," Phys. Rev. ST Accel. Beams* **2***, 074001 (1999).*
https://link.aps.org/doi/10.1103/PhysRevSTAB.2.074001

[Sagan09] *D. Sagan, G. Hoffstaetter, C. mayes, and U. Sae-Ueng, "Extended one-dimensional method for coherent synchrotron radiation including shielding," Phys. Rev. Accel. & Beams* **12***, 040703 (2009).*

[Sagan17] *D. Sagan and C. Mayes, "Coherent Synchrotron Radiation Simulations for Off-Axis Beams Using the Bmad Toolkit", Proc. Europ. Part. Accel. Conf. p. 2829 — 31 (2006).*

[Silenko08] *Alexander Silenko, "Equation of Spin Motion in Storage Rings in a Cylindrical Coordinate System," Phys. Rev. ST Accel. Beams,* **9** *pp. 034003, (2006).*

[Storn96] *R. Storn, and K. V. Price, "Minimizing the real function of the ICEC'96 contest by differential evolution" IEEE conf. on Evolutionary Computation, 842-844 (1996).*

[Talman87] *R. Talman, "Multiparticle Phenomena and Landau Damping," in AIP Conf. Proc.* **153**, Physics of Particle Accelerators*, pp. 789–834, M. Month and M. Dienes editors, American Institute of Physics, New York (1987).*

[Tao] *D. Sagan, J. Smith, The Tao Manual. Can be obtained at:* https://www.classe.cornell.edu/bmad/tao_entry_point.html

[Venturini98] *M Venturini, D Abell, A Dragt, "Map Computation from Magnetic Field Data and Application to the LHC High-Gradient Quadrupoles," Proc. 1998 International Comp. Accel. Phys. Conf. (ICAP '98),* **980914***, 184 (1998).*

[Wiede99] *H. Wiedemann,* Particle Accelerator Physics*, Springer, New York, 3rd Edition (2007).*

[Wolski06] *A. Wolski, "Alternative approach to general coupled linear optics," Phys. Rev. ST Accel. Beams 9, 024001 (2006).*

[Wyckoff65] *R. W. G. Wyckoff,* Crystal Structures*, Interscience Publ. (1965).*

[Schoon11] *T. Schoonjans et al. "The xraylib library for X-ray-matter interactions. Recent developments," Spectrochimica Acta Part B: Atomic Spectroscopy* **66***, pp. 776-784 (2011).*

[Tenen01] *P. Tenenbaum, "LIBXSIF, A Stand alone Library for Parsing the Standard Input Format," Proc. 2001 Part. Acc. Conf. p. 3093 — 95 (2001). Documentation at* https://www-project.slac.stanford.edu/lc/ilc/TechNotes/LCCNotes/PDF/LCC-0060%20rev.1.pdf

Routine Index

Index