



Date: 2024-10-3  
Version: 0.5.5

The  
**Accelerator**  
**Lattice.jl**  
Reference Manual

David Sagan



# Contents

<b>I</b>	<b>Lattice Construction and Manipulation</b>	<b>9</b>
<b>1</b>	<b>Introduction and Concepts</b>	<b>11</b>
1.1	Introduction	11
1.2	Documentation	11
1.3	Brief History	11
1.4	Acknowledgements	12
1.5	Using AcceleratorLattice.jl	12
1.6	Manual Conventions	12
1.7	Lattice Elements	12
1.8	Lattice Branches	13
1.9	Lattices	13
1.10	AcceleratorLattice Conventions	14
1.11	Minimal Working Lattice Example	14
1.12	Differences From Bmad	14
<b>2</b>	<b>Lattice Elements</b>	<b>17</b>
2.1	Element Types	17
2.2	Instantiating a Lattice Element	17
2.3	Element Parameter Groups	18
2.4	Element Parameters	18
2.5	How Element Parameters are Stored in an Element	19
2.6	Bookkeeping and Dependent Element Parameters	20
2.7	Defining a New Element Type	20
2.8	Defining New Element Parameters	21
<b>3</b>	<b>Lattice Element Types</b>	<b>23</b>
3.1	ACKicker	24
3.2	BeamBeam	25
3.3	BeginningEle	26
3.4	Bend	27
3.5	Collimator	28
3.6	Converter	28
3.7	CrabCavity	28
3.8	Drift	29
3.9	EGun	29
3.10	Fiducial	29
3.11	FloorShift	30
3.12	Foil	30
3.13	Fork	30

3.14	Girder . . . . .	33
3.15	Instrument . . . . .	35
3.16	Kicker . . . . .	35
3.17	LCavity . . . . .	35
3.18	Marker . . . . .	35
3.19	Mask . . . . .	35
3.20	Match . . . . .	35
3.21	Multipole . . . . .	36
3.22	NullEle . . . . .	36
3.23	Octupole . . . . .	36
3.24	Patch . . . . .	37
3.25	Quadrupole . . . . .	37
3.26	RFCavity . . . . .	37
3.27	Sextupole . . . . .	38
3.28	Solenoid . . . . .	38
3.29	Taylor . . . . .	38
3.30	ThickMultipole . . . . .	38
3.31	Undulator . . . . .	38
3.32	UnionEle . . . . .	38
3.33	Wiggler . . . . .	38
<b>4</b>	<b>Enums and Holy Traits</b>	<b>39</b>
4.1	Enums . . . . .	39
4.1.1	BendType Enum Group . . . . .	40
4.1.2	BodyLoc Enum Group . . . . .	40
4.1.3	BranchGeometry Enum Group . . . . .	40
4.1.4	Cavity Enum Group . . . . .	41
4.1.5	ParticleState Enum Group . . . . .	41
4.1.6	Loc Enum Group . . . . .	41
4.1.7	Select Enum Group . . . . .	42
4.1.8	ExactMultipoles Enum Group . . . . .	42
4.1.9	FiducialPt Enum Group . . . . .	42
4.2	Holy Traits . . . . .	42
4.2.1	ApertureShape Holy Trait Group . . . . .	42
4.2.2	EleGeometry Holy Trait Group . . . . .	43
<b>5</b>	<b>Element Parameter Groups</b>	<b>45</b>
5.1	ACKickerGroup . . . . .	46
5.2	AlignmentGroup . . . . .	47
5.3	ApertureGroup . . . . .	48
5.4	BMultipoleGroup . . . . .	50
5.5	BeamBeamGroup . . . . .	51
5.6	BendGroup . . . . .	51
5.7	DescriptionGroup . . . . .	55
5.8	DownstreamReferenceGroup . . . . .	55
5.9	EMultipoleGroup . . . . .	56
5.10	FloorPositionGroup . . . . .	57
5.11	ForkGroup . . . . .	57
5.12	GirderGroup . . . . .	57
5.13	InitParticleGroup . . . . .	58
5.14	LengthGroup . . . . .	58

5.15	LordSlaveStatusGroup . . . . .	58
5.16	MasterGroup . . . . .	58
5.17	OriginEleGroup . . . . .	59
5.18	PatchGroup . . . . .	59
5.19	RFGroup . . . . .	62
5.20	RFAutoGroup . . . . .	62
5.21	ReferenceGroup . . . . .	62
5.22	SolenoidGroup . . . . .	63
5.23	TrackingGroup . . . . .	63
5.24	TwissGroup . . . . .	64
<b>6</b>	<b>Constructing Lattices</b>	<b>65</b>
<b>7</b>	<b>Multipass</b>	<b>67</b>
7.1	Multipass Fundamentals . . . . .	67
7.2	The Reference Energy in a Multipass Line . . . . .	69
<b>8</b>	<b>Superposition</b>	<b>71</b>
8.1	Superposition on a Drift . . . . .	71
8.2	Superposition and Sub-Lines . . . . .	76
8.3	Jumbo Super_Slaves . . . . .	77
8.4	Changing Element Lengths when there is Superposition . . . . .	78
<b>9</b>	<b>Customizing Lattices</b>	<b>79</b>
<b>10</b>	<b>Utilities</b>	<b>81</b>
10.1	Iteration Over Elements in a Lattice . . . . .	81
10.2	Searching for Lattice Elements . . . . .	81
10.3	Particle Properties Conversion Functions . . . . .	81
10.4	Math Utilities . . . . .	81
10.5	Lattice Manipulation . . . . .	81
10.6	Miscellaneous Utilities . . . . .	81
<b>II</b>	<b>Conventions and Physics</b>	<b>83</b>
<b>11</b>	<b>Coordinates</b>	<b>85</b>
11.1	Coordinate Systems . . . . .	85
11.2	Element Entrance and Exit Coordinates . . . . .	86
11.3	Branch Coordinates Construction . . . . .	86
11.4	Floor Coordinates . . . . .	88
11.4.1	Lattice Element Positioning . . . . .	90
11.4.2	Position Transformation When Transforming Coordinates . . . . .	91
11.4.3	Crystal and Mirror Entrance to Exit Coordinate Transformation . . . . .	92
11.4.4	Patch and FloorShift Entrance to Exit Transformation . . . . .	93
11.4.5	Reflection Patch . . . . .	93
11.4.6	Fiducial and Girder Coordinate Transformation . . . . .	93
11.5	Transformation Between Branch and Element Body Coordinates . . . . .	93
11.5.1	Straight Element Branch to Body Coordinate Transformation . . . . .	94
11.5.2	Bend Element Branch to Body Coordinate Transformation . . . . .	94

<b>12 Electromagnetic Fields</b>	<b>97</b>
12.1 Magnetostatic Multipole Fields . . . . .	97
12.2 Electrostatic Multipole Fields . . . . .	99
12.3 Exact Multipole Fields in a Bend . . . . .	99
12.4 Map Decomposition of Magnetic and Electric Fields . . . . .	101
12.5 Cartesian Map Field Decomposition . . . . .	102
12.6 Cylindrical Map Decomposition . . . . .	104
12.6.1 DC Cylindrical Map Decomposition . . . . .	104
12.6.2 AC Cylindrical Map Decomposition . . . . .	106
12.7 Generalized Gradient Map Field Modeling . . . . .	108
12.8 RF fields . . . . .	110
 <b>III Developer's Guide</b>	 <b>113</b>
<b>13 Defining New Lattice Elements</b>	<b>115</b>
13.1 Defining new Element Parameters . . . . .	115
13.2 Defining a New Element . . . . .	115
<b>14 Lattice Bookkeeping</b>	<b>117</b>
14.1 Lord/Slave Bookkeeping . . . . .	117
14.2 Girders . . . . .	117
14.3 Superposition . . . . .	118
14.4 Lord/Slave Element Pointers . . . . .	118
14.5 Element Parameter Access . . . . .	118
14.6 Changed Parameters and Auto-Bookkeeping . . . . .	118
<b>15 Design Decisions</b>	<b>119</b>
 <b>IV Bibliography</b>	 <b>123</b>

# List of Figures

3.1	Bend geometry . . . . .	27
3.2	Cornell/Brookhaven CBETA ERL/FFAG machine with fork elements. . . . .	30
3.3	Girder example. . . . .	34
3.4	The branch reference coordinates in a Patchelement. . . . .	36
5.1	Element alignment. . . . .	47
5.2	Alignment geometry. . . . .	48
5.3	Apertures. . . . .	48
5.4	Bend geometry . . . . .	52
5.5	Patch Element. . . . .	59
8.1	Superposition Offset. . . . .	71
11.1	The three coordinate system used by <i>AcceleratorLattice.jl</i> . . . . .	85
11.2	Lattice elements as LEGO blocks. . . . .	86
11.3	Branch coordinates construction. . . . .	87
11.4	The Floor Coordinate System . . . . .	89
11.5	Orientation of a Bend. . . . .	90
11.6	Mirror and crystal geometry . . . . .	92





# List of Tables

3.1	Table of element types. . . . .	23
4.1	Table of enum groups. . . . .	40
5.1	Table of element parameter groups. . . . .	45



## Part I

# Lattice Construction and Manipulation



# Chapter 1

## Introduction and Concepts

### 1.1 Introduction

This chapter is an introduction to, the *AcceleratorLattice.jl* package which is part of the greater *SciBmad* ecosystem of toolkits and programs for accelerator simulations. With *AcceleratorLattice.jl*, lattices can be constructed and manipulated. Essentially, a `lattice` instance contains a set of “`branches`” and a branch contains an array of lattice `elements` with each element representing an object like a magnet or a RF cavity. A branch can be used to describe such things as LINACs, storage rings, injection lines, X-ray beam lines, etc. Different branches in a lattice can be connected together. For example, an injection line branch can be connected to a storage ring branch or the lines of two rings can be connected together to form a colliding beam machine. This ability to describe the interconnections between branches means that a lattice instance can hold all the information about an entire machine complex from beam creation to dump lines enabling a single lattice to be used as the basis of start-to-end simulations.

The sole purpose of the *AcceleratorLattice.jl* package is to implement methods for lattice construction. Other stuff, like tracking and lattice analysis (for example, calculating closed orbits and Twiss functions), is left to other packages in the *SciBmad* ecosystem.

### 1.2 Documentation

There are three main sources of documentation of the *AcceleratorLattice.jl* package. One source is this PDF manual which gives in-depth documentation. A second source is the web based introduction and overview guide. Finally, functions, structs and other objects are documented in the code files themselves. Taking advantage of Julia’s built-in documentation system, this code-file documentation can be accessed via using Julia’s REPL.

### 1.3 Brief History

*AcceleratorLattice.jl* has its origins in the *Bmad* [Sagan:Bmad2006] ecosystem of toolkits and programs developed over several decades at Cornell University. While the development of *AcceleratorLattice.jl* is heavily influenced by the experience — both good and bad — of the development and use of *Bmad* as well as experience with other accelerator simulation programs, the code of the two are com-

pletely separate with *Bmad* being written in Fortran and *AcceleratorLattice.jl* being written in Julia.

The *Julia* language itself is used as the basis for constructing lattices with *AcceleratorLattice.jl*. Other simulation programs have similarly utilized the underlying programming language for constructing lattices[**Appleby:Merlin2020**, **Iadarola:Xsuite2023**]. This is in marked contrast to many accelerator simulation programs such programs as MAD[**Grote:MAD1989**], Elegant[**Borland:Elegant2000**], and *Bmad*. By using Julia for the lattice language, the user will automatically have access to such features as plotting, optimization packages, linear algebra packages, etc. This gives a massive boost to the versatility and usability of any *SciBmad* simulation program. Moreover, maintainability is greatly enhanced due to the reduction in the amount of code that needs to be developed.

## 1.4 Acknowledgements

Thanks must go to the people who have contributed to this effort and without whom *SciBmad* would only be a shadow of what it is today:

Étienne Forest (aka Patrice Nishikawa), Dan Abell, Scott Berg, Oleksii Beznosov, Alexander Coxe, Laurent Deniau, Auralee Edelen, Ryan Foussel, Juan Pablo Gonzalez-Aguilera, Georg Hoffstaetter, Chris Mayes, Matthew Signorelli, Hugo Slepicka

## 1.5 Using AcceleratorLattice.jl

*AcceleratorLattice.jl* is hosted on GitHub. The official repository is at

`github.com/bmad-sim/AcceleratorLattice.jl`

The `README.md` file there has instructions on how to install *AcceleratorLattice.jl*.

A `using` statement must be given before using *AcceleratorLattice.jl* in Julia

```
using AcceleratorLattice
```

## 1.6 Manual Conventions

This manual has the following conventions:

**Type fields:** Fields of a type are also referred to as **components** or **parameters**. A component `c` of a type `S` can be referred to as `S.c`. In the case of lattice elements, `Ele` (the abstract type that all elements inherit from) is used represent any of the subtypes such as `Quadrupole`, etc. If the component is an array, the notation `S.c[]` can be used to emphasize this.

## 1.7 Lattice Elements

The basic building block used to describe an accelerator is the lattice **element**. An element is generally something physical like a bending magnet or a quadrupole, or a diffracting crystal.

Lattice elements can be divided into two classes. One class are the elements that particles are tracked through. These “tracking” elements are contained in the “tracking branches” (§1.8) of the lattice. Other elements, called “lord” elements, are used to represent relationships between elements. “`Super_lord`”

elements (§8) are used when elements overlap spatially. “`Multipass_lord`” elements (7) are used when a beam goes through the same elements multiple times like in a recirculating Linac or when different beam go through the same elements like in the interaction region of a colliding beam machine.

## 1.8 Lattice Branches

The next level up from lattice `elements` are the `branches`. Each branch holds an array of lattice elements. A branch is of type `Branch`.

There are two types of `branches`: branches whose `Branch.type` parameter is set to a suitable subtype of `LordBranch` holds Lord elements and branches whose `Branch.type` parameter is set to `TrackingBranch` holds an ordered list of elements that can be tracked through. *AcceleratorLattice.jl* defines three lord branches named:

```
"super"      -- Contains super lord elements.
"multipass"   -- Contains multipass lord elements.
"girder"      -- Contains Girder elements.
```

Additional lord branches may be added by the user if desired.

A tracking branch can represent a LINAC, X-Ray beam line, storage ring, etc. For all tracking branches, the first element in the element array must be of type `BeginningEle` (§3.3). Additionally, for all tracking branches, the end element must be of type `Marker` (§??).

All tracking branches have a name `Branch.name` inherited from the `BeamLine` that defines the branch in the lattice file and branches contains an array of elements `Branch.ele[]`. If the `BeamLine` used to instantiate a tracking branch does not have a name, The default name is used. The default name is “bN” where N is the index of the branch in the `Lattice.branch` array.

## 1.9 Lattices

A `lattice` (§1.9) is the root structure holding the information about a “machine”. A machine may be as simple as a line of elements (like the elements of a Linac) or as complicated as an entire accelerator complex with multiple storage rings, Linacs, transfer lines, etc. All lattices are of type `Lattice`.

Essentially, a `lattice`, has an array `Lattice.branch[]` of `branches` with each branch describing part of the machine. Branches can be interconnected to form a unified whole. Tracking branches can be interconnected using `Fork` elements (§3.13). 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 but is not forked to by any other `branch` 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.

Branches can be accessed by name using the overloaded index operator for `Lattice`. For example, if `lat` is an instance of a lattice, the super lord branch (§1.8), which has the name “`super`”, can be accessed via:

```
lat["super"]      # Access by branch name
lat.branch[2]     # Access by branch index
```

Where it is assumed for this example that the super lord branch has index 2.

Similarly, lattice elements can be accessed by name or by index. For example, if `lat` is a lattice instance,

and "q1" is the name of an element or elements that are in a branch named "b2", the following are equivalent:

```
elist = lat["q1"]
elist = find(lat, "q1")
b2 = lat.branch["b2"]; elist = b2["q1"]
```

`elist` will be a vector of elements since a name may match to multiple elements.

## 1.10 AcceleratorLattice Conventions

*AcceleratorLattice.jl* has the following conventions:

**Evaluation is at upstream end:** For lattice element parameters that are *s*-dependent, the evaluation location is the **upstream** edge of the element (§11.3). These parameters include the element's floor position, the reference energy/momentum, and the *s*-position.

## 1.11 Minimal Working Lattice Example

The following is a minimal example of constructing a lattice with a quadrupole, drift, and then a bend:

```
using AcceleratorLattice
@ele begin_ele = BeginningEle(pc_ref = 1e7, species_ref = species("electron"))
@ele q = Quadrupole(L = 0.6, K2 = 0.3)
@ele d = Drift(L = 0.4)
@ele b = Bend(L = 1.2, angle = 0.001)

a_line = beamline("simple_line", [begin_ele, q, d, b])
lat = Lattice("simple_lat", [a_line])
```

## 1.12 Differences From Bmad

There are many differences between *AcceleratorLattice.jl* and *Bmad*. Many of these will be fairly obvious. Some differences to be aware of:

*Bmad* is generally case insensitive (except for things like file names). *AcceleratorLattice.jl*, like the Julia language, is case sensitive. With *Bmad*, the branch array within a lattice and the element array within a branch is indexed from zero. With *SciBmad*, indexing of `Lattice.branch[]` and `branch.ele[]` is from one conforming to the Julia standard.

The *Bmad* names for the coordinate systems (§11.1) was somewhat different and not always consistent. The `floor` and `element` body names are the same but `machine` coordinates are called the `laboratory` in *Bmad*.

Evaluation was at the downstream end (§1.10) in *Bmad* not the upstream end.

With *Bmad* a value for any aperture limits of zero means the limit does not exist. with *AcceleratorLattice.jl* a value of `NaN` means the aperture does not exist. Additionally, with *Bmad* a positive value for `x1_limit` or `y1_limit` meant that the aperture was on the negative side of the *x*-axis or *y*-axis respectively. With *AcceleratorLattice.jl*, a positive value for `x_limit[1]` or `y_limit[1]`



means the aperture is on the positive side of the **x-axis** or **y-axis** respectively. This makes the notation consistent across the different ways to specify apertures (compare with **Mask** element syntax.).

*AcceleratorLattice.jl* defines the reference point for misalignment of a **Bend** element as the center of the chord between the entrance and exit end points. With *Bmad*, the reference point is at the center of the reference trajectory arc between the entrance and the exit. An additional difference is that the *Bmad* **roll** misalignment is called **tilt** under *AcceleratorLattice.jl*.

*Bmad* does not allow redefinition of named variables nor elements. *AcceleratorLattice.jl* allows this.

With *Bmad*, the beginning and end elements are implicitly inserted into a branch line. With *AcceleratorLattice.jl*, only an end element will be implicitly inserted if the end of the beamline is not a marker. Also with *Bmad* the beginning element is always named **Beginning** while with *AcceleratorLattice.jl* there is no restriction on the name.

Restrictions on the order of statements used to create a lattice are different. For example, in *Bmad*, a statement defining a lattice element can be placed anywhere except if there is an **expand\_lattice** statement and the element is not being used with superposition in which case the element definition must be before the **expand\_lattice** statement. With *AcceleratorLattice.jl*, element definitions must come before the element is used in a line.

With *Bmad* superposition of two non-drift elements, if there existed the appropriate combined type, will result in a **super\_slave** of the appropriate combined type. For example, a **solenoid** superimposed over a **quadrupole** would give a **sol\_quad super\_slave** with **solenoid** and **quadrupole super\_lords**. The problem here is that calculation of the **super\_slave** parameters may not be possible. For example if the **super\_lord** elements are misaligned, in general it is not possible to compute a corresponding **super\_slave** misalignment. To avoid this, *AcceleratorLattice.jl* creates a **UnionEle super\_slave** element (which in *Bmad* is known as a “jumbo” **super\_slave**). It is up to the tracking routines to figure out how to track through a **UnionEle**

In *Bmad* there are two types of bends called **sbend** and **rbend**. This organization was inherited from *MAD*. While both **sbends** and **rbends** represent the same physical type of bend, the two have different ways to specify the bend parameters. This can be confusing since **rbends** and **sbends** use the same names for different parameters. For example, the length **l** for an **sbend** is the arc length but for an **rbend** it is the chord length. To avoid confusion, *AcceleratorLattice.jl* combines the two into a single **Bend** type with distinct parameter names. For example, **L** is the arc length and **L\_chord** is the chord length.



## Chapter 2

# Lattice Elements

This chapter discusses lattice elements including how to create them and how to manipulate them.

## 2.1 Element Types

Lattice element types (`Quadrupole`, `RFCavity`, etc.) are structs that inherit from the abstract type `Ele`. Lattice elements documentation is in chapter §3. In the REPL, to see a list of all element types, use the command `subtypes(Ele)`:

```
julia> subtypes(Ele)
41-element Vector{Any}:
  ACKicker
  BeamBeam
  BeginningEle
  Bend
  ...
```

## 2.2 Instantiating a Lattice Element

Elements are defined using the `@ele` or `@eles` macros. The general syntax of the `@ele` macro is:

```
@ele eleName = eleType(param1 = val1, param2 = val2, ...)
```

where `eleName` is the name of the element, `eleType` is the type of element, `param1`, `param2`, etc. are parameter names and `val1`, `val2`, etc. are the parameter values. Example:

```
@ele qf = Quadrupole(L = 0.6, Kn1 = 0.370)
```

The `@ele` macro will construct a *Julia* variable with the name `eleName`. Additionally, the element that this variable references will also hold `eleName` as the name of the element. So with this example, `qf.name` will be the string "qf". If multiple elements are being defined in a group, a single `@eles` macro can be used instead of multiple `@ele` macros using the syntax:

```
@eles begin
  eleName1 = eleType1(p11 = v11, p12 = v12, ...)
  eleName2 = eleType2(p21 = v21, p22 = v22, ...)
  ... etc...
end
```

Example:

```
@eles begin
    s1 = Sextupole(L = ...)
    b2 = Bend(...)
    ...
end
```

## 2.3 Element Parameter Groups

Generally, element parameters are grouped into “**element parameter group**” structs which inherit from the abstract type `EleParameterGroup`. Element parameter documentation is in Chapter §5. In the REPL, To see a list of parameter groups, use the `subtypes` function:

```
julia> subtypes(EleParameterGroup)
28-element Vector{Any}:
  AlignmentGroup
  ApertureGroup
  BMultipoleGroup
  ...
```

Chapter §3 documents the parameters groups that are associated with any particular element type. In the REPL, the associated parameter groups can be viewed using Julia’s help function. Example:

```
help?> Quadrupole
mutable struct Quadrupole <: Ele
  Type of lattice element.

  Associated parameter groups
  =====
  • AlignmentGroup -> Element position/orientation shift.
  • ApertureGroup -> Vacuum chamber aperture.
  • BMultipoleGroup -> Magnetic multipoles.
  • EMultipoleGroup -> Electric multipoles.
  • FloorPositionGroup -> Floor position and orientation.
  • LengthGroup -> Length and s-position parameters.
  • LordSlaveGroup -> Element lord and slave status.
  • MasterGroup -> Contains field_master parameter.
  • ReferenceGroup -> Reference energy and species.
  • DescriptionGroup -> String labels for element.
  • TrackingGroup -> Default tracking settings.
```

Alternatively,

## 2.4 Element Parameters

For example, the `LengthGroup` holds the length and s-positions of the element and is defined by:

```
@kwdef struct LengthGroup <: EleParameterGroup
    L::Number = 0.0          # Length of element
    s::Number = 0.0          # Starting s-position
    s_downstream::Number = 0.0 # Ending s-position
    orientation::Int = 1      # Longitudinal orientation
end
```

The `@kwdef` macro automatically defines a keyword-based constructor for `LengthGroup`. See the Julia manual for more information on `@kwdef`. To see a list of all element parameter groups use the `subtypes(ElParamterGroup)` command. To see the components of a given group use the `fieldnames` function. For information on a given element parameter use the `info(::Symbol)` function where the argument is the symbol corresponding to the component. For example, the information on the `s_downstream` parameter which is a field of the `LengthGroup` is:

```
julia> info(:s_downstream)
User name:      s_downstream
Stored in:      LengthGroup.s_downstream
Parameter type: Number
Units:          m
Description:     Longitudinal s-position at the downstream end.
```

Notice that the argument to the `info` function is the symbol associated with the parameter. the “user name” is the name used when setting the parameter. For instance, if `q` is a lattice element, `q.s_downstream` would be used to access the `s_downstream` component of `q`. This works, even though `s_downstream` is not a direct component of an element, since the dot selection operator for lattice elements has been overloaded as explained in §2.5. For most parameters, the user name and the name of the corresponding component in the element parameter group are the same. However, there are exceptions. For example:

```
julia> info(:theta)
User name:      theta_floor
Stored in:      FloorPositionGroup.theta
Parameter type: Number
Units:          rad
Description:     Element floor theta angle orientation
```

In this example, the user name is `theta_floor` so that this parameter can be set via

```
@ele bg = BeginningEle(theta_floor = 0.3)    # Set at element definition time.
bg.theta_floor = 2.7                          # Or set after definition.
```

But the component in the `FloorPositionGroup` is `theta` so

```
bg.FloorPositionGroup.theta = 2.7    # Equivalent to the set above.
```

## 2.5 How Element Parameters are Stored in an Element

All lattice element types have a single field of type `Dict{Symbol,Any}` named `pdict`. The values of `pdict` will, with a few exceptions, be an element parameter group. The corresponding key for a parameter group in `pdict` is the symbol associated with the type. For example, a `LengthGroup` struct would be stored in `pdict[:LengthGroup]`.

To (partially) hide the complexity of parameter groups, the dot selection operator is overloaded for elements. This is achieved by overloading the `Base.setproperty` and `Base.getproperty` functions, which get called when the dot selection operator is used. For example, if `q` is an element instance, `q.s` will get mapped to `q.pdict[:LengthGroup].s`. Additionally, `q.LengthGroup` is mapped to `q.pdict[:LengthGroup]`.

Besides simplifying the syntax, overloading the dot selection operator has a second purpose which is to allow the *AcceleratorLattice.jl* bookkeeping routines to properly do dependent parameter bookkeeping (§??). To illustrate this, consider the following two statements which both set the `s_downstream` parameter of an element named `q1`:

```
q1.pdict[:Length_group].s_downstream = q1.pdict[:Length_group].s +
                                         q1.pdict[:Length_group].L

q1.s_downstream = q1.s + q1.L
```

These two statements are not equivalent. The difference is that in the first statement when `setproperty` is called to handle `q1.pdict`, the code will simply return `q1.pdict` (the code knows that `pdict` is special) and do nothing else. However, with the second statement, `setproperty` not only sets `q1.s_downstream` but additionally records the set by adding an entry to `q1.pdict[:changed]` which is a dict within `pdict`. The key of the entry will, in this case, be the symbol `:s_downstream` and the value will be the old value of the parameter. When the `bookkeeper(::Lattice)` function is called (§??), the bookkeeping code will use the entries in `ele.pdict[:changed]` to limit the bookkeeping to what is necessary and thus minimize computation time. Knowing what has been changed is also important in resolving what dependent parameters need to be changed. For example, if the bend `angle` is changed, the bookkeeping code will set the bending strength `g` using the equation  $g = \text{angle} / L$ . If, instead, `g` is changed, the bookkeeping code will set `angle` appropriately.

While the above may seem complicated, in practice the explicit use of `q1.pdict` should be avoided since it prevents the bookkeeping from dealing with dependent parameters. The place where `q1.pdict` is needed is in the bookkeeping code itself to avoid infinite loops.

## 2.6 Bookkeeping and Dependent Element Parameters

After lattice parameters are changed, the function `bookkeeper(::Lattice)` needs to be called so that dependent parameters can be updated. Since bookkeeping can take a significant amount of time if bookkeeping is done every time a change to the lattice is made, and since there is no good way to tell when bookkeeping should be done, After lattice expansion, `bookkeeper(::Lattice)` is never called directly by *AcceleratorLattice.jl* functions and needs to be called by the User when appropriate (generally before tracking or other computations are done).

Broadly, there are two types of dependent parameters: intra-element dependent parameters where the changed parameters and the dependent parameters are all within the same element and cascading dependent parameters where changes to one element cause changes to parameters of elements downstream.

The cascading dependencies are:

**s-position dependency:** Changes to an elements length `L` or changes to the beginning element's `s` parameter will result in the s-positions of all downstream elements changing.

**Reference energy dependency:** Changes to the beginning element's reference energy (or equivalently the reference momentum), or changes to the **voltage** of an `LCavity` element will result in the reference energy of all downstream elements changing.

**Floor position dependency:** The position of a lattice element in the floor coordinate system (§11.4) is affected by a) the lengths of all upstream elements, b) the bend angles of all upstream elements, and c) the position in floor coordinates of the beginning element.

## 2.7 Defining a New Element Type

To construct a new type, use the `@construct_ele_type` macro. Example:

```
@construct_ele_type MyEle
```

And this defines a new type called `MyEle` which inherits from the abstract type `Ele` and defines `MyEle` to have a single field called `pdict` which is of type `Dict{Symbol,Any}`. This macro also pushes the name

## **2.8 Defining New Element Parameters**





## Chapter 3

# Lattice Element Types

This chapter discusses the various types of elements available in *AcceleratorLattice.jl*. These elements are:

<i>Element</i>	<i>Section</i>	<i>Element</i>	<i>Section</i>
ACKicker	3.1	Marker	3.18
BeamBeam	3.2	Mask	3.19
BeginningEle	3.3	Match	3.20
Bend	3.4	Multipole	3.21
Converter	3.6	NullEle	3.22
Collimator	3.5	Octupole	3.23
CrabCavity	3.7	Patch	3.24
Drift	3.8	Quadrupole	3.25
EGun	3.9	RFCavity	3.26
Fiducial	3.10	Sextupole	3.27
FloorShift	3.11	Solenoid	3.28
Foil	3.12	Taylor	3.29
Fork	3.13	ThickMultipole	3.30
Girder	3.14	Undulator	3.31
Instrument	3.15	UnionEle	3.32
Kicker	3.16	Wiggler	3.33
LCavity	3.17		

Table 3.1: Table of element types.

### 3.1 ACKicker

An `ac_kicker` element simulates a “slow” time dependent kicker element.

NOTE: This Element is in development and is incomplete. Missing: Need to document `amp_function` function to return the kick amplitude.

Element parameter groups associated with this element type are:

---

<code>AlignmentGroup</code>	-> Element position/orientation shift. §??
<code>ApertureGroup</code>	-> Vacuum chamber aperture. §5.3
<code>BMultipoleGroup</code>	-> Magnetic multipoles. §5.4
<code>FloorPositionGroup</code>	-> Floor floor position and orientation. §5.10
<code>LengthGroup</code>	-> Length and s-position parameters. §5.14
<code>LordSlaveGroup</code>	-> Element lord and slave status. §5.15
<code>MasterGroup</code>	-> Contains <code>field_master</code> parameter. §5.16
<code>DescriptionGroup</code>	-> Element descriptive strings. §5.7
<code>TrackingGroup</code>	-> Default tracking settings. §5.23
<code>ReferenceGroup</code>	-> Reference energy and species. §5.21
<code>DownstreamReferenceGroup</code>	-> Reference energy and species at downstream end. §5.8

---

The calculated field will only obey Maxwell’s equations in the limit that the time variation of the field is “slow”:

$$\omega \ll \frac{c}{r} \quad (3.1)$$

where  $\omega$  is the characteristic frequency of the field variation,  $c$  is the speed of light, and  $r$  is the characteristic size of the `ACKicker` 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.

## 3.2 BeamBeam

A `beambeam` element simulates an interaction with an opposing (“strong”) beam traveling in the opposite direction.

NOTE: This Element is in development and is incomplete

Element parameter groups associated with this element type are:

---

FloorPositionGroup	-> Floor floor position and orientation.	§5.10
LengthGroup	-> Length and s-position parameters.	§5.14
LordSlaveGroup	-> Element lord and slave status.	§5.15
DescriptionGroup	-> Element descriptive strings.	§5.7
TrackingGroup	-> Default tracking settings.	§5.23
ReferenceGroup	-> Reference energy and species.	§5.21
DownstreamReferenceGroup	-> Reference energy and species at downstream end.	§5.8

---

### 3.3 BeginningEle

A `BeginningEle` element must be present as the first element of every tracking branch. (§1.8).

Element parameter groups associated with this element type are:

---

<code>FloorPositionGroup</code>	-> Floor floor position and orientation. §5.10
<code>InitParticleGroup</code>	-> Initial particle position and spin. §5.13
<code>LengthGroup</code>	-> Length and s-position parameters. §5.14
<code>LordSlaveGroup</code>	-> Element lord and slave status. §5.15
<code>DescriptionGroup</code>	-> Element descriptive strings. §5.7
<code>TrackingGroup</code>	-> Default tracking settings. §5.23
<code>TwissGroup</code>	-> Initial Twiss and coupling parameters. §5.24
<code>ReferenceGroup</code>	-> Reference energy and species. §5.21
<code>DownstreamReferenceGroup</code>	-> Reference energy and species at downstream end. §5.8

---

Example:

```
@ele bg = BeginningEle(species_ref = Species("proton"), pc_ref = 1e11)
```

## 3.4 Bend

A **Bend** element represents a dipole bend. Bends have a design bend angle and bend radius which determines the location of downstream elements as documented in §??. The actual bending strength that a particle feels can differ from the design value as detailed below.

Element parameter groups associated with this element type are:

---

AlignmentGroup	-> Element position/orientation shift. §??
ApertureGroup	-> Vacuum chamber aperture. §5.3
BMultipoleGroup	-> Magnetic multipoles. §5.4
BendGroup	-> Bend element parameters. §5.6
EMultipoleGroup	-> Electric multipoles. §5.9
FloorPositionGroup	-> Floor floor position and orientation. §5.10
LengthGroup	-> Length and s-position parameters. §5.14
LordSlaveGroup	-> Element lord and slave status. §5.15
MasterGroup	-> Contains field_master parameter. §5.16
DescriptionGroup	-> Element descriptive strings. §5.7
TrackingGroup	-> Default tracking settings. §5.23
ReferenceGroup	-> Reference energy and species. §5.21
DownstreamReferenceGroup	-> Reference energy and species at downstream end. §5.8

---

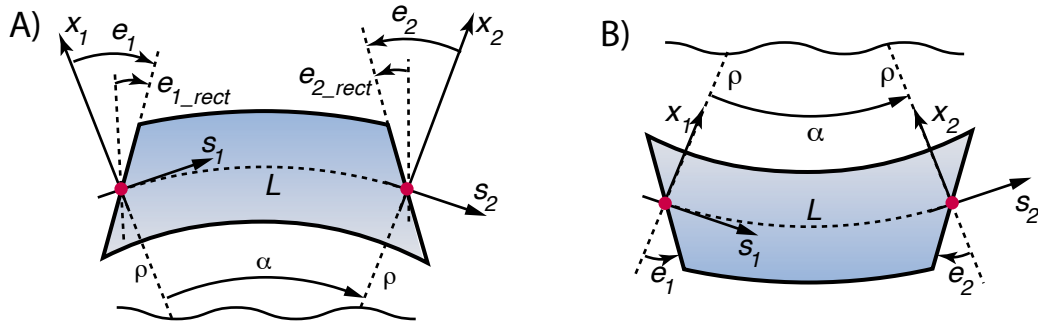


Figure 3.1: Bend geometry. Red dots are the entry and exit points that define the origin for the coordinate systems at the entry end ( $s_1, x_1$ ) and exit ends ( $s_2, x_2$ ) respectively. In the figure, the angle  $\alpha$  is denoted  $\alpha$  and the radius  $\rho$  is denoted  $\rho$ . A) Bend geometry with positive bend angle. For the geometry shown,  $g$ ,  $\text{angle}$ ,  $\rho$ ,  $e_1$ ,  $e_2$ ,  $e_{1\_rect}$ , and  $e_{2\_rect}$  are all positive. B) Bend geometry with negative bend angle. For the geometry shown,  $g$ ,  $\text{angle}$ ,  $\rho$ ,  $e_1$ ,  $e_2$ ,  $e_{1\_rect}$ , and  $e_{2\_rect}$  are all negative. Note: The figures are drawn for zero  $\text{ref\_tilt}$  where the rotation axis is parallel to the  $y$ -axis.

The **BendGroup** group (§5.6) contains the parameters that define the shape of the bend.

Example:

```
@ele b03w = Bend(l = 0.6, g = 0.017, kn1 = 0.003)
```

### 3.5 Collimator

Collimators are field free elements that can collimate beam particles.

Element parameter groups associated with this element type are:

---

FloorPositionGroup	-> Floor floor position and orientation.	§5.10
LengthGroup	-> Length and s-position parameters.	§5.14
LordSlaveGroup	-> Element lord and slave status.	§5.15
DescriptionGroup	-> Element descriptive strings.	§5.7
TrackingGroup	-> Default tracking settings.	§5.23
ReferenceGroup	-> Reference energy and species.	§5.21
DownstreamReferenceGroup	-> Reference energy and species at downstream end.	§5.8

---

### 3.6 Converter

Converter elements convert from one particle species to another. For example, converting electrons hitting on a metal target into positrons.

NOTE: This Element is in development and is incomplete.

Element parameter groups associated with this element type are:

---

FloorPositionGroup	-> Floor floor position and orientation.	§5.10
LengthGroup	-> Length and s-position parameters.	§5.14
LordSlaveGroup	-> Element lord and slave status.	§5.15
DescriptionGroup	-> Element descriptive strings.	§5.7
TrackingGroup	-> Default tracking settings.	§5.23
ReferenceGroup	-> Reference energy and species.	§5.21
DownstreamReferenceGroup	-> Reference energy and species at downstream end.	§5.8

---

### 3.7 CrabCavity

A CrabCavity is an RF cavity that gives a  $z$ -dependent transverse kick. This is useful in colliding beam machines, where there is a finite crossing angle at the interaction point, to rotate the beams near the IP.

NOTE: This Element is in development and is incomplete.

Element parameter groups associated with this element type are:

---

FloorPositionGroup	-> Floor floor position and orientation.	§5.10
LengthGroup	-> Length and s-position parameters.	§5.14
LordSlaveGroup	-> Element lord and slave status.	§5.15
DescriptionGroup	-> Element descriptive strings.	§5.7
TrackingGroup	-> Default tracking settings.	§5.23
ReferenceGroup	-> Reference energy and species.	§5.21
DownstreamReferenceGroup	-> Reference energy and species at downstream end.	§5.8

---

## 3.8 Drift

A `Drift` is a field free element.

Element parameter groups associated with this element type are:

---

<code>FloorPositionGroup</code>	-> Floor floor position and orientation. §5.10
<code>LengthGroup</code>	-> Length and s-position parameters. §5.14
<code>LordSlaveGroup</code>	-> Element lord and slave status. §5.15
<code>DescriptionGroup</code>	-> Element descriptive strings. §5.7
<code>TrackingGroup</code>	-> Default tracking settings. §5.23
<code>ReferenceGroup</code>	-> Reference energy and species. §5.21
<code>DownstreamReferenceGroup</code>	-> Reference energy and species at downstream end. §5.8

---

## 3.9 EGun

An `EGun` element represents an electron gun and encompasses a region starting from the cathode where the electrons are generated.

NOTE: This Element is in development and is incomplete.

Element parameter groups associated with this element type are:

---

<code>FloorPositionGroup</code>	-> Floor floor position and orientation. §5.10
<code>LengthGroup</code>	-> Length and s-position parameters. §5.14
<code>LordSlaveGroup</code>	-> Element lord and slave status. §5.15
<code>DescriptionGroup</code>	-> Element descriptive strings. §5.7
<code>TrackingGroup</code>	-> Default tracking settings. §5.23
<code>ReferenceGroup</code>	-> Reference energy and species. §5.21
<code>DownstreamReferenceGroup</code>	-> Reference energy and species at downstream end. §5.8

---

## 3.10 Fiducial

A `Fiducial` element is used to fix the position and orientation of the reference orbit within the floor coordinate system at the location of the `Fiducial` element. A `Fiducial` element will affect the floor coordinates (§11.4) of elements both upstream and downstream of the fiducial element.

NOTE: This Element is in development and is incomplete.

Element parameter groups associated with this element type are:

---

<code>FloorPositionGroup</code>	-> Floor floor position and orientation. §5.10
<code>LengthGroup</code>	-> Length and s-position parameters. §5.14
<code>LordSlaveGroup</code>	-> Element lord and slave status. §5.15
<code>DescriptionGroup</code>	-> Element descriptive strings. §5.7
<code>TrackingGroup</code>	-> Default tracking settings. §5.23
<code>ReferenceGroup</code>	-> Reference energy and species. §5.21
<code>DownstreamReferenceGroup</code>	-> Reference energy and species at downstream end. §5.8

---

### 3.11 FloorShift

A **FloorShift** element shifts the reference orbit in the floor coordinate system without affecting particle tracking. That is, in terms of tracking, a **FloorShift** element is equivalent to a **Marker** (§3.18) element.

NOTE: This Element is in development and is incomplete.

### 3.12 Foil

A **Foil** element represents a planar sheet of material which can strip electrons from a particle. In conjunction, there will be scattering of the particle trajectory as well as an associated energy loss.

NOTE: This Element is in development and is incomplete.

### 3.13 Fork

A **Fork** element marks a branching point in a lattice branch. Examples include **Fork** from a ring to an extraction line or an X-ray beam line, or **Fork** from the end of an injection line to someplace in a ring. An example is shown in Fig. 3.2.

Element parameter groups associated with this element type are:

---

FloorPositionGroup	-> Floor floor position and orientation. §5.10
LengthGroup	-> Length and s-position parameters. §5.14
DescriptionGroup	-> Element descriptive strings. §5.7
TrackingGroup	-> Default tracking settings. §5.23
ForkGroup	-> Fork parameters. §5.11
ReferenceGroup	-> Reference energy and species. §5.21
DownstreamReferenceGroup	-> Reference energy and species at downstream end. §5.8

---

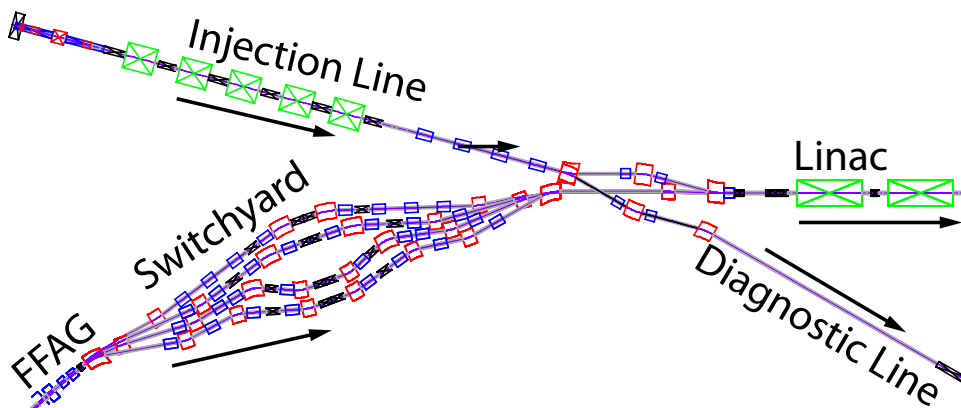


Figure 3.2: Section of the 8-pass (4 passes with increasing energy and 4 passes with decreasing energy) Cornell/Brookhaven CBETA ERL/FFAG machine. Fork elements are used to connect the injection line to the ring and to connect the ring to a diagnostic line. The geometry of the switchyard, used to correct the timings of the differing energy beams, is done using Patch elements.



The **branch** containing a **Fork** element is called the “**base branch**”. The **branch** that the **Fork** element points to is called the “**forked-to branch**”. The `to_ele...`

Fork elements may be put in a lattice by including them in beamlines before a lattice is instantiated. After a lattice has been instantiated, Fork elements can be inserted by using the **superimpose!** or **insert!** functions.

The components of this group are:

```
to_line::UnionBeamLine,Branch  - Beam line to fork to
to_ele::UnionString,Element    - Element forked to.
direction::Int                 - Longitudinal Direction of injected beam.
```

If a fork creates a new branch, and if the reference energy or species type for the forked-to branch is the particle associated with a branch can be set by setting the **particle** attribute of the **Fork** element.

Forked-to branches can themselves have **Fork** elements. A branch always starts out tangential to the line it is branching from. A **patch** element (§3.24) can be used to reorient the reference orbit as needed. Example:

```
@ele x_patch = Patch, (offset[1] = 0.01)
@ele pb = Fork(to_line = x_line)
@ele bgn = BeginningEle(p0c = 1e3, species_ref = Species("photon")
                                ! Photon reference momentum
from_line = BeamLine([... a, pb, b, ...]) ! Defines base branch
x_line = BeamLine([bgn, x_patch, x1, x2, ...]) ! Defines forked-to branch
```

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 **Fork** element does not change.

Forking elements do not have orientational attributes like **x\_pitch** and **tilt** (??). If the orientation of the forked-to 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 **Fork** 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_ele = f1, direction = -1
x_f2: fork, to_line = ring2, to_ele = 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 (§??) 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_ele` attribute for a `Fork` element is used to designate the element of the forked-to branch that the `Fork` element connects to. To keep things conceptually simple, the `to_ele` must be a “marker-like” element which has zero length and unit transfer matrix. Possible `to_ele` types are:

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

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

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. If the reference particle of a branch line is different from the reference particle in the base branch, the reference energy (or reference momentum) of a forked-to branch line needs to be set using line parameter statements (§??). If the reference particle of a branch line is the same as the reference particle in the base branch, the reference energy will default to the reference energy of the base branch if the reference energy is not set for the branch.

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
ring[E_tot] = 1.7e9                       ! Ring ref energy.
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 (§??), the branches defined by the `use` statement (§??) are searched for fork elements that branch to new forked-to 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 forked-to 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.

If the forked-to branch and base branch both have the same reference particle, and if the element forked into is the beginning element, the reference energy and momentum of the forked-to branch will be set to the reference energy and momentum at the fork element. In this case, neither the reference energy nor reference momentum of the forked-to branch should be set. If it is desired to have the reference energy/momentum of the forked-to branch different from what is inherited from the fork element, a patch element (§3.24) can be used at the beginning of the forked-to branch. In all other cases, where either the two branches have different reference particles or the fork connects to something other than the beginning element, there is no energy/momentum inheritance and either the reference energy or reference momentum of the forked-to branch must be set.

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.

## 3.14 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.

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.

Element parameter groups associated with this element type are:

---

FloorPositionGroup	-> Floor floor position and orientation. §5.10
LengthGroup	-> Length and s-position parameters. §5.14
DescriptionGroup	-> Element descriptive strings. §5.7
AlignmentGroup	-> Alignment with respect to the reference. §5.2

---

A simple example of a girder is shown in Fig. 3.3. 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:

```
lat = Lattice(...)          # Create lattice
create_external_ele(lat)    # Optional: Create external elements
@ele g1 = Girder(supported = [A, B, C], ...)
create_girder!(g1)
```

The `create_girder` command must appear after the lattice has been constructed. The list of `supported` elements must contain only elements that are in a single lattice. Be careful here since lattice creation involves creating copies of the elements in the `BeamLines` that define the lattice. Use of the function `create_external_ele` may be useful here. The `find` function may also be used to search for the appropriate elements in the lattice.

The list of supported elements does not have to be in any order and may contain elements from multiple branches. A **Girder** may not support slave elements. If a super slave or multipass slave element is in

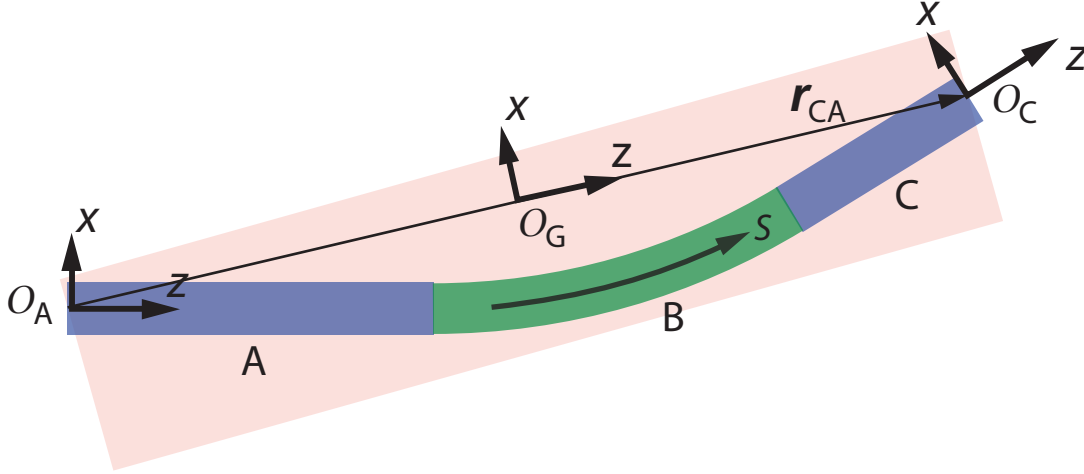


Figure 3.3: Girder supporting three elements labeled A, B, and C.  $\mathcal{O}_A$  is the reference frame at the upstream end of element A (§11.3),  $\mathcal{O}_C$  is the reference frame at the downstream end of element C, and  $\mathcal{O}_G$  is the default `origin` reference frame of the girder if the `origin_ele` parameter is not set.  $\mathbf{r}_{CA}$  is the vector from  $\mathcal{O}_A$  to  $\mathcal{O}_C$ . The length  $l$  of the girder is set to be the difference in  $s$  between points  $\mathcal{O}_C$  and  $\mathcal{O}_A$ .

the list, the slave will be removed and the corresponding lords of the slave will be substituted into the list.

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.

If all the supported elements of a `Girder` are contained within a single lattice branch (lord elements are considered to be in the branch(s) that their slaves are in), The length  $L$  of the `Girder` is calculated by the difference in  $s$ -position from the upstream end of the supported element with minimal  $s$ -position to the downstream end of the supported element with the maximal  $s$ -position. If there are supported elements in multiple branches, the length is set to `NaN`. The girder length is not used in any calculations.

The reference frame from which a `Girder`'s orientation is measured is set by the `origin_ele` and `origin_ele_ref_point` parameters (§5.17). Orientation shifts are controlled by the `AlignmentGroup` (§??).

When a girder is shifted in space, the elements it supports are also shifted. In this case, the orientation attributes 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`, etc, 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  $s = 0$  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 `x_offset`, denoted `x_offset_tot` of `q2` is  $0.06 = 0.01 + 0.03 + 0.02$ . The total `x_pitch`, denoted `x_pitch_tot`, 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.

## 3.15 Instrument

An `Instrument` is like a `Drift` except it represents some measurement device.

## 3.16 Kicker

A `Kicker` element gives particles a kick.

NOTE: This Element is in development and is incomplete.

## 3.17 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 (§??) through an `LCavity` is not constant.

NOTE: This Element is in development and is incomplete.

## 3.18 Marker

A `Marker` is a zero length element meant to mark a position in the machine.

## 3.19 Mask

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

NOTE: This Element is in development and is incomplete.

## 3.20 Match

A `Match` element is used to adjust Twiss and orbit parameters.

NOTE: This Element is in development and is incomplete.

### 3.21 Multipole

A `Multipole` is a thin magnetic multipole lens.

### 3.22 NullEle

A `NullEle` is used for bookkeeping purposes. For example, a `NullEle` can be used as the default value for a function argument or as a temporary place marker in a lattice.

### 3.23 Octupole

An `Octupole` is a magnetic element with a cubic field dependence with transverse position (§12.1).

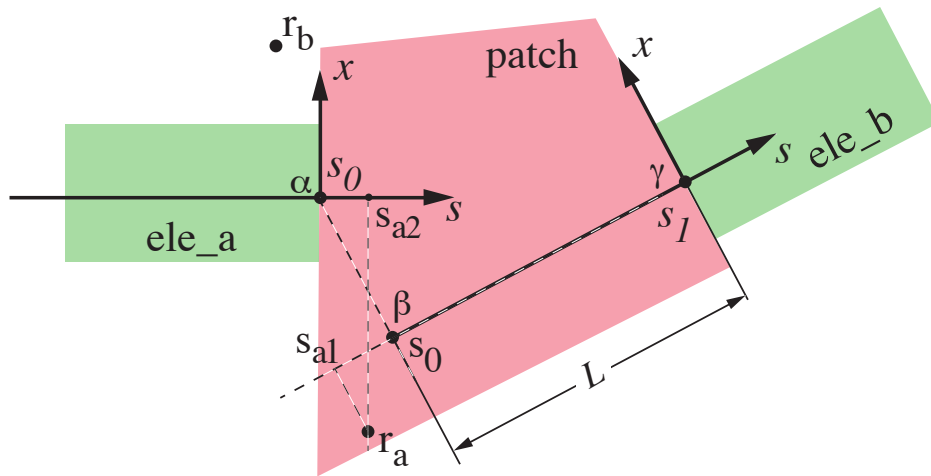


Figure 3.4: The branch reference coordinates in a `Patch` element. The `Patch` element, shown schematically as an irregular quadrilateral, is sandwiched between elements `ele_a` and `ele_b`.  $L$  is the length of the `Patch`. In this example, the `Patch` has a finite `y_rot`.

## 3.24 Patch

A **Patch** element shifts the reference orbit and time. Also see **FloorShift** (§3.11) and **Fiducial** (§3.10) elements. A common application of a patch is to orient two branch lines with respect to each other.

A **Patch** (§3.24) element is different in that there is no “natural” coordinate system to use within the Patch. This is generally not an issue when the region inside the Patch is field and aperture free since particle tracking can be done in one step from edge to edge. However, when there are fields or apertures an internal coordinate system is needed so that the fields or apertures can be unambiguously positioned.

There

Generally, if a particle is reasonably near the branch reference curve, there is a one-to-one mapping between the particle’s position and branch  $(x, y, s)$  coordinates.

with a non-zero **x\_rot** or non-zero **y\_rot** breaks the one-to-one mapping. This is illustrated in Fig. 3.4. The **Patch** element, shown schematically as an, irregular quadrilateral, is sandwiched between elements **ele\_a** and **ele\_b**. The branch coordinate system with origin at  $\alpha$  are the coordinates at the end of **ele\_a**. The coordinates at the end of the **Patch** has its origin labeled  $\gamma$ . By convention, the length of the patch  $L$  is taken to be the longitudinal distance from  $\alpha$  to  $\gamma$  with the **Patch**’s exit coordinates defining the longitudinal direction. The “beginning” point of the **Patch** on the reference curve a distance  $L$  from point  $\gamma$  is labeled  $\beta$  in the figure.

In the branch  $(x, y, s)$  coordinate system a particle at  $\alpha$  will have some value  $s = s_0$ . A particle at point  $\beta$  will have the same value  $s = s_0$  and a particle at  $\gamma$  will have  $s = s_1 = s_0 + L$ . A particle at point  $r_a$  in Fig. 3.4 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 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, *AcceleratorLattice.jl* 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. 3.4 can simultaneously be considered to be in either **ele\_a** or the **Patch**. While this creates an ambiguity it does not complicate tracking.

## 3.25 Quadrupole

A **Quadrupole** is a magnetic element with a linear field dependence with transverse offset (§12.1).

## 3.26 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 (§??) through an **RFcavity** is constant.

NOTE: This Element is in development and is incomplete.

### 3.27 Sextupole

A **Sextupole** is a magnetic element with a quadratic field dependence with transverse offset (§12.1).

### 3.28 Solenoid

A **solenoid** is an element with a longitudinal magnetic field.

### 3.29 Taylor

A **Taylor** element is a Taylor map (§??) that maps the input orbital phase space and possibly spin coordinates of a particle to the output orbital and spin coordinates at the exit end of the element.

NOTE: This Element is in development and is incomplete.

### 3.30 ThickMultipole

A **ThickMultipole** is a general non-zero length multipole element.

### 3.31 Undulator

An **Undulator** is an element with a periodic array of alternating bends. Also see **Wiggler** elements.

NOTE: This Element is in development and is incomplete.

### 3.32 UnionEle

A **UnionEle** is an element that contains a collection of other elements. A **UnionEle** is used when elements overlap spatially which happens with superposition (§8).

### 3.33 Wiggler

A **Wiggler** is an element with a periodic array of alternating bends. Also see **Undulator** elements.

NOTE: This Element is in development and is incomplete.



## Chapter 4

# Enums and Holy Traits

Enums (§4.1) and Holy traits (§4.2) are used to define “switches” which are variables whose value can be one of a set of named constants. A web search will provide documentation.

The advantage of Holy traits is that they can be used with function dispatch. The disadvantage is that the same Holy trait value name cannot be used with multiple groups. Generally, if function dispatch is not needed (which is true for the majority of cases), switch groups are defined using enums.

### 4.1 Enums

*AcceleratorLattice.jl* uses the package `EnumX.jl` to define enums (enumerated numbers). Essentially what happens is that for each enum group there is a group name, For example `BendType`, along with a set of values which, for `BendType`, is `SECTOR` and `RECTANGULAR`. Values are always referred to by their "full" name which in this example is `BendType.SECTOR` and `BendType.RECTANGULAR`. Exception: `BranchGeometry.CLOSED` and `BranchGeometry.OPEN` are used often enough so that the constants `OPEN` and `CLOSED` are defined.

The group name followed by a `.T` suffix denotes the enum type. For example:

```
struct ApertureGroup <: EleParameterGroup
    aperture_type::ApertureShape.T = ApertureShape.ELLIPTICAL
    aperture_at::BodyLoc.T = BodyLoc.ENTRANCE_END
    ...
end
```

The `enum` function is used to convert a list into an enum group and export the names. The `enum` function also overloads `Base.string` so that something like `string(Lord.NOT)` will return `"Lord.NOT"` instead of just `"NOT"` (an issue with the `EnumX.jl` package). See the documentation for `enum` for more details.

The `enum_add` function is used to add values to an existing enum group. See the documentation for `enum_add` for more details. This function is used with code extensions to customize *AcceleratorLattice.jl*.

The enum groups are:

<i>Group</i>	<i>Section</i>	<i>Group</i>	<i>Section</i>
BendType	§4.1.1	Slave	§5.15
BodyLoc	§4.1.2	Loc	§4.1.6
BranchGeometry	§4.1.3	Select	§4.1.7
Cavity	§4.1.4	ExactMultipoles	§4.1.8
Lord	§5.15	FiducialPt	§4.1.9
ParticleState	§4.1.5		

Table 4.1: Table of enum groups.

### 4.1.1 BendType Enum Group

Type of Bend element. Possible values:

BendType  
  .SECTOR           – Sector shape  
  .RECTANGULAR   – Rectangular shape

BendType is used with the `bend_type` parameter of the `BendGroup` parameter group (§5.6). The `bend_type` parameter gives the “logical” shape of the bend. The setting of `bend_type` is only relevant when the bend curvature is varied. See §5.6 for more details.

### 4.1.2 BodyLoc Enum Group

Longitudinal location with respect to an element’s body coordinates. Possible values:

BodyLoc  
  .ENTRANCE\_END   – Body entrance end  
  .CENTER         – Element center  
  .EXIT\_END       – Body exit end  
  .BOTH\_ENDS      – Both ends  
  .NOWHERE       – No location  
  .EVERYWHERE     – Everywhere

BodyLoc enums are useful to locate things that are “attached” to an element. For example, specifying where apertures are placed.

### 4.1.3 BranchGeometry Enum Group

Geometry of a lattice branch. Used for setting a branch’s `geometry` parameter. Possible values:

BranchGeometry  
  .OPEN           – Open geometry like a Linac. Default  
  .CLOSED         – Closed geometry like a storage ring.

A branch with a `CLOSED` geometry is something like a storage ring where the particle beam recirculates through the machine. A branch with an `OPEN` geometry is something like a linac. In this case, the initial Twiss parameters need to be specified at the beginning of the branch. If the `geometry` is not specified, `OPEN` is the default.

Since the geometry is widely used, `OPEN` and `CLOSED` have been defined and can be used in place of `BranchGeometry.OPEN` and `BranchGeometry.CLOSED`.

Notice that by specifying a `CLOSED` geometry, it does *not* mean that the downstream end of the last element of the branch has the same floor coordinates (§11.4) as the floor coordinates at the beginning. Setting the geometry to `CLOSED` simply signals to a program to compute the periodic orbit and periodic Twiss parameters as opposed to calculating orbits and Twiss parameters based upon initial orbit and Twiss parameters given at the beginning of the branch. Indeed, it is sometimes convenient to treat branches as closed even though there is no closure in the floor coordinate sense. For example, when a machine has a number of repeating “periods”, it may be convenient to only use one period in a simulation. Since *AcceleratorLattice.jl* ignores closure in the floor coordinate sense, it is up to the lattice designer to ensure that a branch is truly geometrically closed if that is desired.

#### 4.1.4 Cavity Enum Group

Type of RF cavity. Possible values:

```
Cavity
  .STANDING_WAVE    – Standing wave cavity
  .TRAVELING_WAVE   – Traveling wave cavity
```

#### 4.1.5 ParticleState Enum Group

State of a particle. Possible values:

```
ParticleState
  .PREBORN           – State before emission from cathode.
  .ALIVE             – Alive and kicking.
  .LOST              – Particle has been lost.
  .LOST_NEG_X        – Hit aperture in the -x direction.
  .LOST_POS_X        – Hit aperture in the +x direction.
  .LOST_NEG_Y        – Hit aperture in the -y direction.
  .LOST_POS_Y        – Hit aperture in the +y direction.
  .LOST_PZ           – Lost all forward momentum.
  .LOST_Z            – Out of RF bucket.
```

The `LOST` value is used when it is not possible to assign the particle state to one of the other lost values.

The `.LOST_PZ` value is used by *s* (longitudinal position) based trackers which are not able to handle particles changing their longitudinal motion direction. For tracking something like dark current electrons which can go back and forth longitudinally, a time based tracker is needed.

#### 4.1.6 Loc Enum Group

Longitudinal location with respect to element’s branch coordinates. Possible values:

```
Loc
  .UPSTREAM_END      – Upstream element end
  .CENTER            – center of element
  .INSIDE            – Somewhere inside
  .DOWNSTREAM_END    – Downstream element end
```

### 4.1.7 Select Enum Group

Specifies where to select if there is a choice of elements or positions. Possible values:

```
Select
    .UPSTREAM      – Select upstream
    .DOWNSTREAM    – Select downstream
```

### 4.1.8 ExactMultipoles Enum Group

How multipoles are handled in a Bend element. Possible values:

```
ExactMultipoles
    .OFF              – Bend curvature not taken into account.
    .HORIZONTALLY_PURE – Coefficients correspond to horizontally pure multipoles.
    .VERTICALLY_PURE  – Coefficients correspond to vertically pure multipoles.
```

### 4.1.9 FiducialPt Enum Group

Fiducial point location with respect to element’s branch coordinates. Possible values:

```
FiducialPt
    .ENTRANCE_END – Entrance end of element
    .CENTER       – Center of element
    .EXIT_END     – Exit end of element
    .NONE         – No point chosen
```

## 4.2 Holy Traits

**Holy traits** (named after Tim Holy) are a design pattern in Julia that behave similarly to **enums** (§??). A Holy trait group consists of a base abstract type with a set of values (traits) which are abstract types that inherit from the base abstract type.

The advantage of Holy traits is that they can be used with function dispatch. The disadvantage is that the same Holy trait value name cannot be used with multiple groups.

There is a convenience function **holy\_traits** which will define a traits group, export the names, and create a docstring for the group. Values can be added to an existing group by defining a new struct that inherits from the group abstract type.

Example: To extend the **EleGeometry** trait group to include the value **HELIX\_GEOMETRY** do

```
abstract type HELIX_GEOMETRY <: EleGeometry
```

### 4.2.1 ApertureShape Holy Trait Group

The shape of an aperture.

```
RECTANGULAR – Rectangular shape.
ELLIPTICAL  – Elliptical shape.
VERTEX      – Shape defined by set of vertices.
CUSTOM_SHAPE – Shape defined with custom function.
```

### 4.2.2 EleGeometry Holy Trait Group

The geometry of the reference orbit through an element.

STRAIGHT	– Straight line geometry.
CIRCULAR	– Circular "bend-like" geometry.
ZERO_LENGTH	– Zero longitudinal length geometry.
PATCH_GEOMETRY	– Patch element like geometry.
GIRDER_GEOMETRY	– Support girder-like geometry.
CRYSTAL_GEOMETRY	– Crystal geometry.
MIRROR_GEOMETRY	– Mirror geometry.



## Chapter 5

# Element Parameter Groups

Generally, element parameters are grouped into “**element parameter group**” types. How these groups are used in a lattice element is discussed in §2. This chapter discusses the groups in detail.

The parameter groups are:

<i>Group</i>	<i>Section</i>	<i>Group</i>	<i>Section</i>
ACKickerGroup	§5.1	InitParticleGroup	§5.13
AlignmentGroup	§5.2	LengthGroup	§5.14
ApertureGroup	§5.3	LordSlaveStatusGroup	§5.15
BMultipoleGroup	§5.4	MasterGroup	§5.16
BeamBeamGroup	§5.5	OriginEleGroup	§5.17
BendGroup	§5.6	PatchGroup	§5.18
DescriptionGroup	§5.7	RFGGroup	§5.19
DownstreamReferenceGroup	§5.8	RFAutoGroup	§5.20
EMultipoleGroup	§5.9	ReferenceGroup	§5.21
FloorPositionGroup	§5.10	SolenoidGroup	§5.22
ForkGroup	§5.11	TrackingGroup	§5.23
GirderGroup	§5.12	TwissGroup	§5.24

Table 5.1: Table of element parameter groups.

Element parameter groups inherit from the abstract type `EleParameterGroup` which in turn inherits from `BaseEleParameterGroup`. Some parameter groups have sub-group components. These sub-groups also inherit from `BaseEleParameterGroup`:

```
abstract type BaseEleParameterGroup end
abstract type EleParameterGroup <: BaseEleParameterGroup end
abstract type EleParameterSubGroup <: BaseEleParameterGroup end
```

To see which element types contain a given group, use the `info(::EleParameterGroup)` method. Example:

```
julia> info(AlignmentGroup)
ApertureGroup: Vacuum chamber aperture.
...
Found in:
  ACKicker
```

```

Bend
Kicker
...

```

To get information on a given element parameter, including what element group the parameter is in, use the `info(::Symbol)` function using the symbol corresponding to the parameter. For example, to get information on the multipole component `Ks2L` do:

```

julia> info(:Ks2L)
User name:      Ks2L
Stored in:      BMultipoleGroup.Ks
Parameter type: Number
Units:          1/m^2
Description:     Skew, length-integrated, momentum-normalized,
                  magnetic multipole of order 2.

```

Notes:

- All parameter groups have associated docstrings that can be accessed using the REPL help system.
- NaN denotes a real parameter that is not set.
- Parameters marked “dependent” are parameters calculated by *AcceleratorLattice.jl* and not settable by the User.
- There are several lattice element parameters that are not stored in a parameter group but are stored alongside of the parameter groups in the element Dict. Included is the element’s name, and information on lords and slaves of the element.

To simplify the structure of a lattice element, certain element parameters are not stored in the element structure but are calculated as needed. These “output” cannot be set. See §2.5 for details.

## 5.1 ACKickerGroup

The `ACKickerGroup` holds parameters associated with `ACKicker` elements.

The parameters of this group are:

```

amp_function::Function      - Amplitude function.

```



## 5.2 AlignmentGroup

The `AlignmentGroup` gives the alignment (position and angular orientation) of the physical element relative to the nominal position defined by the branch coordinates (§??). Alignment is specified with respect to the “alignment reference point” of an element as shown in Fig ?? . The `Bend` reference point is chosen to be the center of the chord connecting the two ends. This reference point was chosen over using the midpoint on the reference orbit arc since a common simulation problem is to simulate a bend with a `z_rot` keeping the entrance and exit endpoints fixed.

The parameters of the `AlignmentGroup` are:

```
offset::Vector - $[x, y, z]$ offset.
x_rot::Number - Rotation around the x-axis.
y_rot::Number - Rotation around the z-axis.
z_rot::Number - Rotation around the z-axis.
```

If the element is supported by a `Girder`, the alignment parameters are with respect to the orientation of the `Girder` position. If there is no supporting `Girder`, the alignment parameters are with respect to the branch reference coordinates. There are output alignment parameters:

```
q_align::Quaternion - Quaternion representation of x_rot, y_rot, z_rot.
offset_tot::Vector - $[x, y, z]$ offset.
x_rot_tot::Number - Rotation around the x-axis.
y_rot_tot::Number - Rotation around the z-axis.
z_rot_tot::Number - Rotation around the z-axis.
q_align_tot::Quaternion - Quaternion representation of tot rotations.
```

The “total alignment” parameters which have a `_tot` suffix are always the alignment of the element with respect to the branch coordinates. If there is no support `Girder`, the total alignment will be the same as the “relative” (non-tot) alignment.

The relative alignment can be set by the User. The total alignment is computed by `AcceleratorLattice.jl` based upon the relative alignment and the alignment of any `Girder`. `Girder` elements themselves also have both relative and total alignments since Girders can support other Girders.

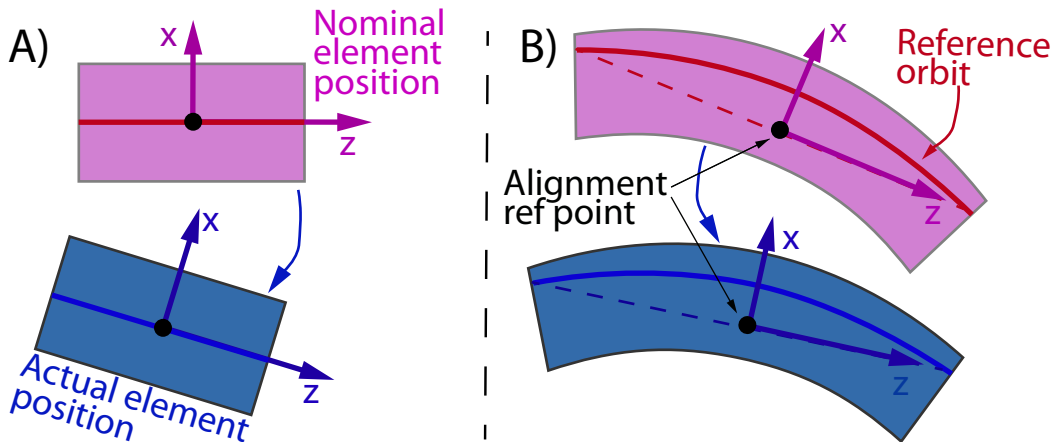


Figure 5.1: `AlignmentGroup` parameters The reference point is the origin about which the element alignment is calculated. A) For straight elements, the reference point is in the center of the element. For `Bend` elements, the reference point is at the midpoint of the chord connecting the entrance point to the exit point. The drawing for the bend is valid for a `ref_tilt` of zero. For non-zero `ref_tilt`, the outward direction from the bend center will not be the `x`-axis.

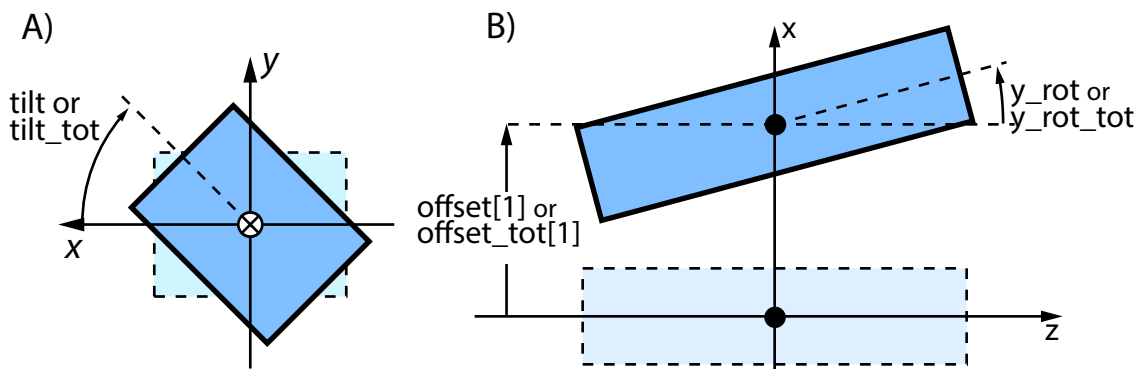


Figure 5.2: Alignment geometry. A)  $z\_rot$  (or  $z\_rot\_tot$ ) rotation. B) Combined  $offset[1]$  with  $y\_rot$  (or  $offset\_tot[1]$  with  $y\_rot\_tot$ ).

The `q_align` output parameter gives the quaternion representation of  $x\_rot$ ,  $y\_rot$  and  $z\_rot$ . Similarly, the `q_align_tot` output parameter gives the quaternion representation of  $x\_rot\_tot$ ,  $y\_rot\_tot$  and  $z\_rot\_tot$ .

### 5.3 ApertureGroup

The `ApertureGroup` stores information about apertures an element may have. The parameters of this group are:

```

x_limit::Vector{Number}    - Min/Max x-aperture limits. (m)
y_limit::Vector{Number}    - Min/Max y-aperture limits. (m)
aperture_shape::ApertureShape - Aperture shape. Default: ELLIPTICAL
aperture_at::BodyLoc.T     - Aperture location. Default: BodyLoc.ENTRANCE_END
wall::Wall2D               - Aperture defined by vertex array.
custom_aperture::Dict      - Custom aperture information.
aperture_shifts_with_alignment::Bool
                             - Alignment affects aperture? Default: false.

```

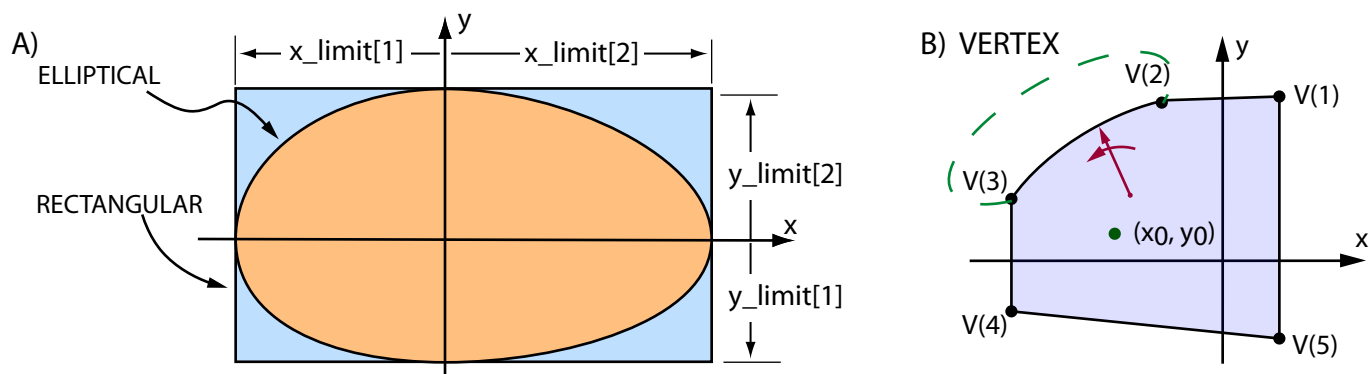


Figure 5.3: A) RECTANGULAR and ELLIPTICAL apertures. As drawn,  $x\_limit[1]$  and  $y\_limit[1]$  are negative and  $x\_limit[2]$  and  $y\_limit[2]$  are positive. B) The VERTEX aperture is defined by a set of vertices.

The aperture location is set by the `aperture_at` parameter. Possible values are given by the `BodyLoc` enum group (§4.1.2). The default is `BodyLoc.ENTRANCE_END`. The `.EVERYWHERE` location might be problematic for some types of particle tracking and so might not be always available.

The `aperture_shape` parameter selects the shape of the aperture. Possible values are given by the `ApertureShape` Holy trait group.

- `RECTANGULAR` - Rectangular shape.
- `ELLIPTICAL` - Elliptical shape.
- `VERTEX` - Shape defined by set of vertices.
- `CUSTOM_SHAPE` - Shape defined with a custom function.

For `RECTANGULAR` and `ELLIPTICAL` shapes the `x_limit` and `y_limit` parameters are used to calculate the aperture as shown in Fig. 5.3A. For an `ELLIPTICAL` aperture, a particle with position  $(x, y)$  is outside of the aperture if any one of the following four conditions is true:

- 1)  $x < 0$  and  $y < 0$  and  $(x/x\_limit[1])^2 + (y/y\_limit[1])^2 > 1$
- 2)  $x < 0$  and  $y > 0$  and  $(x/x\_limit[1])^2 + (y/y\_limit[2])^2 > 1$
- 3)  $x > 0$  and  $y < 0$  and  $(x/x\_limit[2])^2 + (y/y\_limit[1])^2 > 1$
- 4)  $x > 0$  and  $y > 0$  and  $(x/x\_limit[2])^2 + (y/y\_limit[2])^2 > 1$

For a `RECTANGULAR` aperture the corresponding four conditions are:

- 1)  $x < x\_limit[1]$
- 2)  $x > x\_limit[2]$
- 3)  $y < y\_limit[1]$
- 4)  $y > y\_limit[2]$

Default values for the limits are `-Inf` for `x_limit[1]` and `y_limit[1]` and `Inf` for `x_limit[2]` and `y_limit[2]`.

The `misalignment_moves_aperture` parameter determines whether misaligning an element (§5.2) affects the placement of the aperture. The default is `false`. A common case where `misalignment_moves_aperture` would be `false` is when a beam pipe, which incorporates the aperture, is not physically touching the surrounding magnet element. When tracking a particle, assuming that there are only apertures at the element ends, the order of computation with `misalignment_moves_aperture` set to `false` is

- 1) Start at upstream end of element
- 2) Check upstream aperture if there is one.
- 3) Convert from branch coordinates to body coordinates.
- 4) Track through the element body.
- 5) Convert from body coordinates to branch coordinates.
- 6) Check downstream aperture if there is one.
- 7) End at downstream end of element.

With `misalignment_moves_aperture` set to `true`, the computation order is

- 1) Start at upstream end of element
- 2) Convert from branch coordinates to body coordinates.
- 3) Check upstream aperture if there is one.
- 4) Track through the element body.
- 5) Check downstream aperture if there is one.
- 6) Convert from body coordinates to branch coordinates.
- 7) End at downstream end of element.

The `CUSTOM_SHAPE` setting for `aperture_shape` indicates whether a User supplied function is used to calculate whether the particle has hit the aperture. The function is stored in the `custom_aperture` parameter. The `custom_aperture` parameter is a Dict that stores the aperture function along with any data that the aperture calculation needs. The aperture function must be stored in `custom_aperture[:function]` and this function will be called with the signature

```
%
custom_aperture[:function](position::Vector, ele::Ele) -> ParticleState
```

where `position` is the phase space 6-vector of the particle, `ele` is the element with the aperture, and a `ParticleState` (§4.1.5) value is returned.

The `VERTEX` setting for `aperture_shape` is for defining an aperture using a set of vertex points as illustrated in Fig. ??B. Between vertex points, the aperture can follow a straight line or the arc of an ellipse. The vertex points are specified by setting the `section` parameter of `ApertureGroup`. Example:

```
wall = Wall2D([Vertex1([1.0, 4.0]), Vertex1([-1.0, 4.0, 6.0]),
                Vertex1([-5.0, 1.0]), Vertex1([-5.0, -1.0]),
                Vertex1([1.0, -1.5])], r0 = [-2.5, 0.5])
```

## 5.4 BMultipoleGroup

The `BMultipoleGroup` group stores magnetic multipole strengths. Also see `EMultipoleGroup`. The parameters of this group are:

```
vec::VectorBMultipole1
```

This group stores a vector of `BMultipole1` structs. The `BMultipole1` structure stores the values for a magnetic multipole of a given order. Only orders where there is a non-zero multipole are stored and there is no maximum limit to the order that can be stored. The multipoles will be stored in increasing order.

The `BMultipole1` structure has components:

```
Kn::Number      - Normal normalized component. EG: "Kn2", "Kn2L".
Ks::Number      - Skew multipole component. EG: "Ks2", "Ks2L".
Bn::Number      - Normal field component.
Bs::Number      - Skew field component.
tilt::Number     - Rotation of multipole around z-axis.
order::Int       - Multipole order.
integrated::UnionBool,Nothing - Integrated multipoles or not?
```

The `order` component gives the multipole order. There is storage for both normalized (`Kn` and `Ks`) and unnormalized (`Bn` and `Bs`) field strengths. The letter “n” designates the normal component and “s” designates the skew component. The *AcceleratorLattice.jl* bookkeeping code will take care of calculating the normalized field if the unnormalized field is set and vice versa. The reason why the structure has three components, normal, skew and tilt, that describe the field when only two would be sufficient is due to convenience. Having normal and skew components is convenient when a magnet has multiple windings that control both independently. A common case is combined horizontal and vertical steering magnets. On the other hand, being able to “misalign” the multipole using the `tilt` component is also useful.

The dot selection operator for an element (§2.5) is overloaded so that magnetic multipole parameters for order  $J$  can be accessed using the following notation:

Name	Stored In	Normalized	Integrated	Description
<code>KnJ</code>	<code>Kn</code>	Yes	No	Normal field.
<code>KsJ</code>	<code>Ks</code>	Yes	No	Skew field.
<code>KnJL</code>	<code>Kn</code>	Yes	Yes	Normal field.
<code>KsJL</code>	<code>Ks</code>	Yes	Yes	Skew field.
<code>BnJ</code>	<code>Bn</code>	No	No	Normal field.
<code>BsJ</code>	<code>Bs</code>	No	No	Skew field.
<code>BnJL</code>	<code>Bn</code>	No	Yes	Normal field.
<code>BsJL</code>	<code>Bs</code>	No	Yes	Skew field.
<code>tiltJ</code>	<code>tilt</code>	-	-	Field tilt.
<code>integratedJ</code>	<code>integrated</code>	-	-	Integrated fields?

Substitute the multipole order for  $J$  in the above table. For example, Ks2L is the normalized length-integrated skew field component of order 2.

Notice that both integrated and non-integrated fields are potentially stored in the same component of `BMultipole1`. Which type is stored is determined by the `integrated` logical. If `true`, the integrated value is stored and vice versa. The `integrated` setting can be different for different orders. The setting of `integrated` for a given order is determined by whether the first field component to be set for that order is an integrated quantity or not. After the value of `integrated` is set, an error will be thrown if a something that has the opposite sense in terms of integration is set. For example:

```
@ele qq = Quadrupole(l = 0.6, Ks0L = 1.0) # 0th order multipole is integrated
qq.Bn1 = 0.3                             # 1st order multipole is not integrated
qq.Ks1 = 0.5                             # This is OK.
println(qq.integrated0)                   # Will print 'true'
println(qq.Bn0)                           # Can use non-integrated component.
qq.Bn0 = 0.7                             # Cannot set non-integrated component! error thrown!
toggle_integrated!(qq, MAGNETIC, 0) # toggle integrated setting for order 0.
```

In the above example, the 0th order multipole is initialized using `Ks0L` so that multipole will have the `integrated` component set to `true` and non-integrated values cannot be set. However, independent of the setting of `integrated`, both integrated and non-integrated quantities can always be used in an equation. To change the value of `integrated`, use the `toggle_integrated!` function. This function also translates the values stored in the field components of the structure so that the field will stay constant.

The setting of `integrated` for a given multipole will also determine what stays constant of the length of the magnet changes. If `integrated` is `true`, the integrated values will be invariant and vice versa for `integrated` being `false`. Similarly, the setting of the `field_master` parameter (§5.16) will determine whether normalized or unnormalized quantities will stay constant if the reference energy is varied.

## 5.5 BeamBeamGroup

The parameters of this group are:

<code>n_slice::Number</code>	- Number of slices the Strong beam is divided into.
<code>n_particle::Number</code>	- Number of particle in the strong beam.
<code>species::Species</code>	- Strong beam species. Default is weak particle species.
<code>z0_crossing::Number</code>	- Weak particle phase space z when strong beam center passes the BeamBeam element.
<code>repetition_freq:: Number</code>	- Strong beam repetition rate.
<code>twiss::Twiss</code>	- Strong beam Twiss at IP.
<code>sig_x::Number</code>	- Strong beam horizontal sigma at IP.
<code>sig_y::Number</code>	- Strong beam vertical sigma at IP.
<code>sig_z::Number</code>	- Strong beam longitudinal sigma.
<code>bbi_constant::Number</code>	- BBI constant. Set by Bmad. See manual.

## 5.6 BendGroup

The `BendGroup` stores the parameters that characterize the shape of a `Bend` element §3.4. The only relevant shape parameter that is not in the `BendGroup` is the length  $L$  which is in the `LengthGroup`.

The parameters of this group are:

`bend_type::BendType.T` - Type of bend. Default: `BendType.SECTOR`.  
`angle::Number` - Reference bend angle.  
`g::Number` - Reference bend strength =  $1/\text{radius}$ .  
`bend_field_ref::Number` - Reference bend field.  
`L_chord::Number` - Chord length.  
`tilt_ref::Number` - Reference tilt.  
`e1::Number` - Entrance end pole face rotation.  
`e2::Number` - Exit end pole face rotation.  
`e1_rect::Number` - Entrance end pole face rotation.  
`e2_rect::Number` - Exit end pole face rotation.  
`edge_int1::Number` - Entrance end fringe field integral.  
`edge_int2::Number` - Exit end fringe field integral.  
`exact_multipoles::ExactMultipoles.T` - Default: `ExactMultipoles.OFF`

Associated output parameters:

`rho::Number` - Reference bend radius.  
`L_sagitta::Number` - Sagitta length.  
`bend_field::Number` - Actual dipole field in the plane of the bend.  
`norm_bend_field::Number` - Actual dipole strength in the plane of the bend.

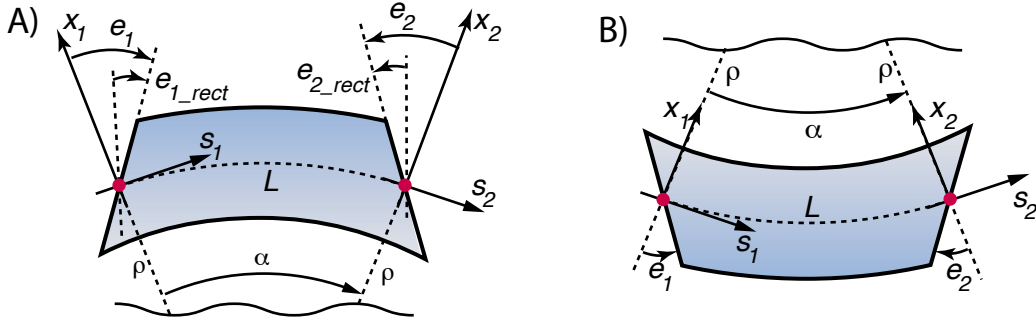


Figure 5.4: Bend geometry. Red dots are the entry and exit points that define the origin for the coordinate systems at the entry end ( $s_1, x_1$ ) and exit ends ( $s_2, x_2$ ) respectively. In the figure, the angle  $\alpha$  is denoted  $\alpha$  and the radius  $\rho$  is denoted  $\rho$ . A) Bend geometry with positive bend angle. For the geometry shown,  $g$ ,  $\text{angle}$ ,  $\rho$ ,  $e_1$ ,  $e_2$ ,  $e_{1\_rect}$ , and  $e_{2\_rect}$  are all positive. B) Bend geometry with negative bend angle. For the geometry shown,  $g$ ,  $\text{angle}$ ,  $\rho$ ,  $e_1$ ,  $e_2$ ,  $e_{1\_rect}$ , and  $e_{2\_rect}$  are all negative. Note: The figures are drawn for zero `ref_tilt` where the rotation axis is parallel to the  $y$ -axis.

In detail:

#### **angle**

The total Reference bend angle. A positive **angle** represents a bend towards negative  $x$  as shown in Fig. 5.4.

#### **bend\_field\_ref**

The `bend_field_ref` parameter is the reference magnetic bending field which is the field that is needed for the reference particle to be bent in a circle of radius  $\rho$  and the placement of lattice elements downstream from the bend. The actual (“total”) field is a vector sum of `bend_field_ref` plus the value of the `Bn0` and `Bs0` multipoles. If `tilt0` and `Bs0` are zero, the actual field is

$$\text{B-field (total)} = \text{bend\_field\_ref} + \text{Bn0}$$

See the discussion of  $g$  and  $\text{Kn0}$  below for more details.

**bend\_field (output param), norm\_bend\_field (output\_param)**

The actual dipole bend field ignoring any skew field component which is set by Bs0. The relation between this and bend\_field\_ref is

$$\text{bend\_field} = \text{bend\_field\_ref} + Bn0 * \cos(\text{tilt0}) + Bs0 * \sin(\text{tilt0})$$

**bend\_type**

The bend\_type parameter sets the “logical shape” of the bend. This parameter is of type BendType.T (§4.1.1) and can take values of

BendType.RECTANGULAR - or  
BendType.SECTOR - The default

The logical shape of a bend, in most situations, is irrelevant. The only case where the logical shape is used is when the bend angle is varied. In this case, for a SECTOR bend, the face angles e1 and e2 are held constant and e1\_rect and e2\_rect are varied to keep Eqs. (5.1) satisfied.

**e1, e2**

The values of e1 and e2 gives the rotation angle of the entrance and exit pole faces respectively with respect to the radial  $x_1$  and  $x_2$  axes as shown in Fig. 5.4. Zero e1 and e2 gives a wedge shaped magnet. Also see e1\_rect and e2\_rect. The relationship is

$$\begin{aligned} e1 &= e1\_rect + \text{angle}/2 \\ e2 &= e2\_rect + \text{angle}/2 \end{aligned} \quad (5.1)$$

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

$$\begin{aligned} e1(\text{AccelLattice}) &= e1(\text{SAD}) * \text{angle} + ae1(\text{SAD}) \\ e2(\text{AccelLattice}) &= e2(\text{SAD}) * \text{angle} + ae2(\text{SAD}) \end{aligned}$$

**e1\_rect, e2\_rect** Face angle rotations like e1 and e2 except angles are measured with respect to fiducial lines that are parallel to each other and rotated by angle/2 from the radial  $x_1$  and  $x_2$  axes as shown in Fig. ?? . Zero e1\_rect and e2\_rect gives a rectangular magnet shape.

**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  $k_1$  and  $k_2$  components, are corrected for the finite curvature of the reference orbit in a bend. See §12.3 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

$$\mathbf{B} = \sum_{n=0}^{\infty} \left[ \frac{a_n}{n+1} \nabla \phi_n^r + \frac{b_n}{n+1} \nabla \phi_n^i \right], \quad \mathbf{E} = \sum_{n=0}^{\infty} \left[ \frac{a_{en}}{n+1} \nabla \phi_n^i + \frac{b_{en}}{n+1} \nabla \phi_n^r \right] \quad (5.2)$$

and if exact\_multipoles is set to horizontally\_pure the horizontally pure solutions  $\psi_n^r$  and  $\psi_n^i$  are used instead of the vertically pure solutions  $\phi_n^r$  and  $\phi_n^i$ .

**edge\_int1, edge\_int2**

The field integral for the entrance pole face edge\_int1 is given by

$$\text{edge}_1 = \int_{\text{pole}} ds \frac{B_y(s) (B_{y0} - B_y(s))}{2 B_{y0}^2} \quad (5.3)$$

For the exit pole face there is a similar equation for edge\_int2

Note: In Bmad and MAD, these integrals are represented by the product of fint and hgap.

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

```
edge_int1 = (fb1 + f1) / 12
edge_int2 = (fb2 + f1) / 12
```

`edge_int1` and `edge_int2` can be related to the Enge function which is sometimes used to model the fringe field. The Enge function is of the form

$$B_y(s) = \frac{B_{y0}}{1 + \exp[P(s)]} \quad (5.4)$$

where

$$P(s) = C_0 + C_1 s + C_2 s^2 + C_3 s^3 + \dots \quad (5.5)$$

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

$$C_1 = \frac{1}{2 \text{field\_int}} \quad (5.6)$$

### **g, rho (output param)**

The Reference bending radius which determines the reference coordinate system is `rho` (see §??). `g = 1/rho` is the “bend strength” and is proportional to the Reference dipole magnetic field. `g` is related to the reference magnetic field `bend_field_ref` via

$$g = \frac{q}{p_0} \text{bend\_field\_ref} \quad (5.7)$$

where  $q$  is the charge of the reference particle and  $p_0$  is the reference momentum. It is important to keep in mind that changing `g` will change the Reference orbit (§??) and hence will move all downstream lattice elements in space.

The total bend strength felt by a particle is the vector sum of `g` plus the zeroth order magnetic multipole. If the multipole `tilt0` and `Ks0` is zero, the total bend strength is

$$\text{norm\_bend\_field} = g + \text{Kn0}$$

Changing the multipole strength `Kn0` or `Ks0` leaves the Reference orbit and the positions of all downstream lattice elements unchanged but will vary a particle’s orbit. One common mistake when designing lattices is to vary `g` and not `Kn0` which results in downstream elements moving around. See Sec. §?? for an example.

Note: A positive `g`, which will bend particles and the reference orbit in the  $-x$  direction represents a field of opposite sign as the field due a positive `hkick`.

### **h1, h2**

The attributes `h1` and `h2` are the curvature of the entrance and exit pole faces.

### **L, L\_arc, L\_chord, L\_sagitta (output param)**

The `L` parameter, which is in the `LengthGroup` and not the `BendGroup`, is the arc length of the reference trajectory through the bend.

`L_chord` is the chord length from entrance point to exit point. The `L_sagitta` parameter is the sagitta length (The sagitta is the distance from the midpoint of the arc to the midpoint of the chord). `L_sagitta` can be negative and will have the same sign as the `g` parameter.

### **L\_rectangle**

The `L_rectangle` parameter is the “rectangular” length defined to be the distance between the entrance and exit points. The coordinate system used for the calculation is defined by the setting of `fiducial_pt`. Fig. ?? shows `l_rectangle` for `fiducial_pt` set to `entrance_end` (the coordinate system corresponds to the entrance coordinate system of the bend). In this case, and in the case where `fiducial_pt` is set to `exit_end`, the rectangular length will be  $\rho \sin \alpha$ . If `fiducial_pt` is set to `none` or `center`, `l_rectangle` is the same as the chord length.



**ref\_tilt**

The `ref_tilt` attribute rotates a bend about the longitudinal axis at the entrance face of the bend. A bend with `ref_tilt` of  $\pi/2$  and positive `g` bends the element in the  $-y$  direction (“downward”). See Fig. 11.5. 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.

Important! Do not use `ref_tilt` when doing misalignment studies for a machine. Trying to misalign a dipole by setting `ref_tilt` will affect the positions of all downstream elements! Rather, use the `tilt` parameter.

The attributes `g`, `angle`, and `L` are mutually dependent. If any two are specified for an element *AcceleratorLattice.jl* will calculate the appropriate value for the third.

In the local coordinate system (§??), 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

$$e1 + e2 = \text{angle}$$

## 5.7 DescriptionGroup

The components of this group are element descriptive strings:

```
type::String
ID::String
class::String
```

For example

```
@ele q1 = Quadrupole(type = "rotating quad", ...)
```

These strings can be used to in element searching:

```
eles(lat, "type = "*rot*")      # Can use these strings in searching
```

In this example `lat` is the lattice that contains `q1` and the `eles` function will return a vector of all elements whose `type` string has the substring “rot” in it.

## 5.8 DownstreamReferenceGroup

The components of this group are:

```
species_ref_downstream::Species - Reference species.
pc_ref_downstream::Number      - Reference momentum*c.
E_tot_ref_downstream::Number    - Reference total energy.
```

Associated output parameters are:

```
 $\beta$ _ref_downstream::Number      - Reference v/c.
 $\gamma$ _ref_downstream::Number    - Reference relativistic gamma factor.
```

This group holds the reference energy and species at the downstream end of an element. Also see the `ReferenceGroup` (§5.21) documentation. This group and `ReferenceGroup` group are always paired. That is, these two are always both present or both not present in any given element.

For most elements, the values of the parameters in `DownstreamReferenceGroup` will be the same as the values in the corresponding `ReferenceGroup` parameters. That is, the value of `species_ref_downstream` in `DownstreamReferenceGroup` will be the same as the value of `species_ref` in `ReferenceGroup`, the value of `pc_ref_downstream` will be the same as `pc_ref`, etc. Elements where the reference energy (here "energy" refers to either `pc_ref`, `E_tot_ref`,  $\beta$ , or  $\gamma$ ) differs between upstream and downstream include `LCavity` and `Patch` elements. Elements where the reference energy and species differ between upstream and downstream include `Foil` and `Converter` elements.

Parameters of the `DownstreamReferenceGroup` are not user settable and are calculated by the *AcceleratorLattice.jl* bookkeeping routines. See the `ReferenceGroup` documentation for how these parameters are calculated.

## 5.9 EMultipoleGroup

The `EMultipoleGroup` group stores electric multipole strengths. Also see `BMultipoleGroup`. The parameters of this group are:

`vec::VectorEMultipole1`

This group stores a vector of `EMultipole1` structs. The `EMultipole1` structure stores the values for a electric multipole of a given order. Only orders where there is a non-zero multipole are stored and there is no maximum limit to the order that can be stored. The multipoles will be stored in increasing order.

The `EMultipole1` structure has components:

`En::Number` - Normal field component.  
`Es::Number` - Skew field component.  
`Etilt::Number` - Rotation of multipole around z-axis.  
`order::Int` - Multipole order.  
`Eintegrated::Union{Bool,Nothing}` - Integrated multipoles or not?

The `order` component gives the multipole order. There is storage for unnormalized (`En` and `Es`) field strengths however, unlike magnetic multipoles, there are no components for normalized field strengths. The letter “n” designates the normal component and “s” designates the skew component. There is also a `Etilt` component which will tilt the entire multipole §??. The reason why the structure has three components, normal, skew and tilt, that describe the field when only two would be sufficient is due convenience. Having normal and skew components is convenient when magnet has multiple windings that control both independently.

The dot selection operator for an element (§2.5) is overloaded so that electric multipole parameters for order  $J$  can be accessed using the following notation:

Name	Stored In	Integrated	Description
<code>EnJ</code>	<code>En</code>	No	Normal field.
<code>EsJ</code>	<code>Es</code>	No	Skew field.
<code>EnJL</code>	<code>En</code>	Yes	Normal field.
<code>EsJL</code>	<code>Es</code>	Yes	Skew field.
<code>EtiltJ</code>	<code>Etilt</code>	-	Field tilt.
<code>EintegratedJ</code>	<code>Eintegrated</code>	-	Integrated fields?

Substitute the multipole order for  $J$  in the above table. For example, `Es2L` is the normalized length-integrated skew field component of order 2.

Notice that both integrated and non-integrated fields are potentially stored in the same component of `EMultipole1`. Which type is stored is determined by the `Eintegrated` logical. If `true`, the integrated value is stored and vice versa. The `Eintegrated` setting can be different for different orders. The setting of `Eintegrated` for a given order is determined by whether the first field component to be set for that order is an integrated quantity or not. After the value of `Eintegrated` is set, an error will be thrown if a something that has the opposite sense in terms of integration is set. For example:

```
@ele qq = Quadrupole(l = 0.6, EsOL = 1.0) # 0th order is integrated
qq.En1 = 0.3                             # 1st order multipole is not integrated
qq.Es1 = 0.5                             # This is OK.
println(qq.Eintegrated0)                  # Will print "true"
println(qq.En0)                           # Can use non-integrated component.
toggle_integrated!(qq, ELECTRIC, 0)      # change integrated setting for order 0.
```

In the above example, the 0th order multipole is initialized using `EsOL` so that multipole will have the `Eintegrated` component set to `true` and non-integrated values cannot be set. However, independent of the setting of `Eintegrated`, both integrated and non-integrated quantities can always be used in an equation. To change the value of `Eintegrated`, use the `toggle_integrated!` function. This function also translates the values stored in the field components of the structure so that the field will stay constant.

The setting of `Eintegrated` for a given multipole will also determine what stays constant of the length of the magnet changes. If `Eintegrated` is `true`, the integrated values will be invariant with length changes and vice versa if `integrated` is `false`. Similarly, the setting of the `field_master` parameter (§5.16) will determine whether normalized or unnormalized quantities will stay constant if the reference energy is varied.

## 5.10 FloorPositionGroup

The `FloorPositionGroup` stores the nominal (calculated without alignment shifts) position and orientation in the floor coordinates of the upstream end of the element. system. The components of this group are:

```
r::Vector          - [x,y,z] position. Accessed using r_floor
q::Quaternion      - Quaternion orientation. Accessed using q_floor.
```

## 5.11 ForkGroup

The components of this group are:

```
to_line::Union{BeamLine,Branch} - Beam line to fork to
to_ele::Union{String,Element}    - Element forked to.
direction::Int                   - Longitudinal Direction of injected beam.
```

This group is used with a `Fork` element and specifies how the fork element attaches to another branch.

## 5.12 GirderGroup

The components of this group are:

```
supported::Vector{Element} - Elements supported by girder.
```

### 5.13 InitParticleGroup

The components of this group are:

```
orbit::VectorNumber    - Phase space 6-vector.
spin::VectorNumber     - Spin 3-vector.
```

### 5.14 LengthGroup

The components of this group are:

```
L::Number              - Length of element.
s::Number              - Starting s-position.
s_downstream::Number   - Ending s-position.
orientation::Int        - Longitudinal orientation. +1 or -1.
```

### 5.15 LordSlaveStatusGroup

The components of this group are:

```
lord_status::Lord.T    - Lord status.
slave_status::Slave.T   - Slave status.
```

The possible values of `lord_status` are:

```
Lord
  .NOT          - Not a lord
  .SUPER        - Is a Super lord (§8)
  .MULTIPASS    - Is a Multipass lord (§7)
```

The possible values of `slave_status` are:

```
Slave
  .NOT          - Not a slave
  .SUPER        - Is a Super slave (§8)
  .MULTIPASS    - Multipass slave (§7)
```

All elements in a tracking branch have this element group even if the type of element (For example, **Drift** elements) is such that the element will never be a lord or a slave.

Notice that elements that are supported by a **Girder** are not marked as slaves to the **Girder** although the supported elements will have a pointer to the supporting girder.

This group is used in lattice element lord/slave bookkeeping (§??). See this section for more details.

### 5.16 MasterGroup

The components of this group are:

```
is_on::Bool = true
field_master::Bool = false      # Does field or normalized field stay constant with energy changes?
```

## 5.17 OriginEleGroup

The components of this group are:

- `origin_ele::Ele` - Origin reference element. Default is `NULL_ELE`.
- `origin_ele_ref_pt::Loc.T` - Origin reference point. Default is `Loc.CENTER`.

The `OriginEleGroup` is used with `Fiducial`, `FloorShift`, and `Girder` elements. The `OriginEleGroup` is used to set the coordinate reference frame from which the orientation set by the `AlignmentGroup` is measured. To specify that the floor coordinates are to be used, set the `origin_ele` to `NULL_ELE`. Typically this is the same as using the beginning element of the first branch of a lattice as long as the first element does not have any orientation shifts.

## 5.18 PatchGroup

The components of this group are:

- `t_offset::Number` - Time offset.
- `E_tot_offset::Number` - Total energy offset. Default is `NaN` (not used).
- `E_tot_exit::Number` - Fix total energy at exit end. Default is `NaN` (not used).
- `pc_exit::Number` - Reference momentum\*c at exit end. Default is `NaN` (not used).
- `flexible::Bool` - Flexible patch? Default is `false`.
- `L_user::Number` - User set Length? Default is `NaN` (length calculated by bookkeeping code).
- `ref_coords::BodyLoc.T` - Reference coordinate system used inside the patch. Default is `BodyLoc.EXIT`.

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. 5.5A.

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 §11.4.4. This type of `patch` is called “rigid” or “inflexible” since the geometry of the `patch` is solely determined by the `patch`’s attributes as set in the lattice file and is independent of everything else. Example:

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

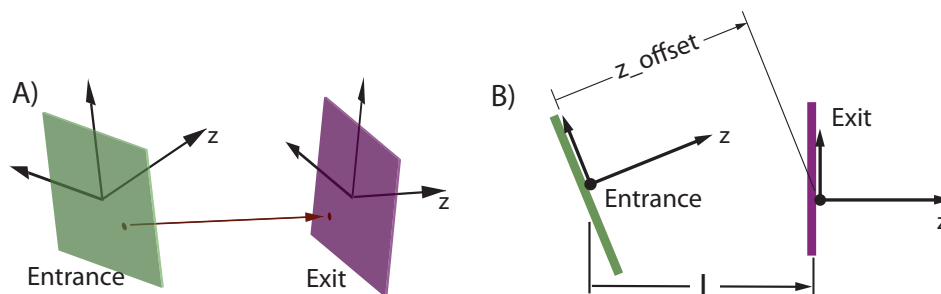


Figure 5.5: A) A `patch` element can align its exit face arbitrarily with respect to its entrance face. The red arrow illustrates a possible particle trajectory from entrance face to exit face. B) The reference length of a `patch` element, if `ref_coords` is set to the default value of `exit_end`, is the longitudinal distance from the entrance origin to the exit origin using the reference coordinates at the exit end as shown. If `ref_coords` is set to `entrance_end`, the length of the patch will be equal to the `z_offset`.

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 before the reference coordinates of the `patch` are computed. A `flexible patch` will have its offsets, pitches, and tilt as dependent parameters (§??) and these parameters will be computed appropriately. 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 §?? for an example.

The coordinates of the lattice element downstream of a `flexible patch` can be computed if there is a `fiducial` element (§3.10) somewhere downstream or if there is a `multipass_slave` (§7) element which is just downstream of the `patch` or at most separated by zero length elements from the `patch`. In this latter case, the `multipass_slave` must represent an  $N^{th}$  pass slave with  $N$  greater than 1. This works since the first pass slave will be upstream of the `patch` and so the first pass slave will have its coordinates already computed and the position of the downstream slave will be taken to be the same as the first pass slave. Notice that, without the `patch`, the position of multipass slave elements are independent of each other.

With `bmad_standard` tracking (§??) 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 (§??). 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` (§??) needs to be set to `custom`. In this case, custom code must be supplied for calculating the fields as a function of position (§??).

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. (??) and (??). 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 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.

A `patch` element can have an associated electric or magnetic field (§??). This can happen, for example, if a patch is used at the end of an injection line to match the reference coordinates of the injection line to the line being injected into (§??) and the patch element is within the field generated by an element in the line being injected into. In such a case, it can be convenient to set what the reference coordinates are since the orientation of any fields that are defined for a patch element will be oriented with respect to the patch element’s reference coordinates. For this, the `ref_coords` parameter of a patch can be used. Possible settings are: `ref_coords` are:

```
entrance_end !
exit_end     ! Default
```

The default setting of `ref_coords` is `exit_end` and with this the reference coordinates are set by the exit end coordinate system (see Fig. 5.5). If `ref_coords` is set to `entrance_end`, the reference coordinates are set by the entrance end coordinate system. Example:

```
p1: patch, x_offset = 1, x_pitch = 0.4    ! L = 0.289418 see below
p2: p1, ref_coords = entrance_end        ! L = 0
```

Here `p1` has `ref_coords` set to `exit_end` (the default). `p2` inherits the parameters of `p1` and sets `ref_coords` to `entrance_end`.

It is important to keep in mind that if there are multiple patches in a row, while two different configurations may be the same in a geometrical sense the total length may not be the same. For example:

```
pA: patch, x_offset = 1    ! L = 0
pB: patch, x_pitch = 0.4   ! L = 0
sum: line = (pA, pB)
```

The configuration of `pA` followed by `pB` is equivalent geometrically to the `p1` patch above but the total length of the (`pA`, `pB`) line is zero which is different from the length of `p1`.

Unfortunately, there is no intuitive way to define the “length” `L` of a patch. This is important since the transit time of the reference particle is the element length divided by the reference velocity. And the reference transit time will affect how the phase space  $z$  coordinate changes through the patch via Eq. (??). If the parameter `user_sets_length` is set to `True`, the value of `l` set in the lattice file will be used (default is zero). `user_sets_length` is set to `False` (the default), the length of a patch is calculated depending upon the setting of `ref_coords`. If `ref_coords` is set to `exit_end`, the length of the patch is calculated as the perpendicular distance between the origin of the patch’s entrance coordinate system and the exit face of the patch as shown in Fig. 5.5B. If `ref_coords` is set to `entrance_end`, the length is calculated as the perpendicular distance between the entrance face and the origin of the exit coordinate system. In this case, the length will be equal to `z_offset`.

To provide flexibility, the `t_offset` attribute can be used to offset the reference time. 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` 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. (??). The change in  $z$ ,  $\delta z$  will be

$$\delta z = \beta \cdot c \cdot t\_offset \quad (5.8)$$

where  $\beta$  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 (§11.3), a `patch`, or series of `patches`, which reflects the  $z$  direction must be placed in between. Such a `patch`, (or patches) is

called a **reflection patch**. See Section §11.4.5 for more details on how a reflection patch is defined. In order to avoid some confusing conceptual problems involving the coordinate system through a reflection patch, Runge-Kutta type tracking is prohibited with a reflection patch.<sup>1</sup>

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 §?? for more details.

## 5.19 RFGroup

The components of this group are:

<code>frequency::Number</code>	- RF frequency.
<code>harmon::Number</code>	- RF frequency harmonic number.
<code>voltage::Number</code>	- RF voltage.
<code>gradient::Number</code>	- RF gradient.
<code>phase::Number</code>	- RF phase.
<code>multipass_phase::Number</code>	- RF Phase added to multipass elements.
<code>cavity_type::Cavity.T</code>	- Cavity type. Default is <code>Cavity.STANDING_WAVE</code> .
<code>n_cell::Int</code>	- Number of cavity cells. Default is 1.

Whether `voltage` or `gradient` is kept constant with length changes is determined by the setting of `field_master` (§5.16). If `field_master` is true, the `gradient` is kept constant and vice versa.

## 5.20 RFAutoGroup

The components of this group are:

<code>do_auto_amp::Bool</code>	- Will autoscaling set <code>auto_amp</code> ? Default is true.
<code>do_auto_phase::Bool</code>	- Will autoscaling set <code>auto_phase</code> ? Default is true.
<code>auto_amp::Number</code>	- Auto RF field amplitude scale value.
<code>auto_phase::Number</code>	- Auto RF phase value.

## 5.21 ReferenceGroup

The components of this group are:

<code>species_ref::Species</code>	- Reference species entering end.
<code>pc_ref::Number</code>	- Reference momentum*c upstream end.
<code>E_tot_ref::Number</code>	- Reference total energy upstream end.
<code>time_ref::Number</code>	- Reference time upstream end.
<code>time_ref_downstream::Number</code>	- Reference time downstream end.
<code>extra_dtime_ref::Number</code>	- User set reference time change.
<code>dE_ref::Number</code>	- Sets change in reference energy.

Associated output parameters are:

<code><math>\beta</math>_ref::Number</code>	- Reference v/c upstream end.
<code><math>\gamma</math>_ref::Number</code>	- Reference relativistic gamma factor.

This group holds the reference energy, species, and time parameters at the upstream end of a lattice element. Also see the `DownstreamReferenceGroup` group documentation (§5.8). The `DownstreamReferenceGroup`

---

<sup>1</sup>In general, Runge-Kutta type tracking through a patch is a waste of time unless electric or magnetic fields are present.



group holds the reference energy and species at the downstream end of the element. This group and `DownstreamReferenceGroup` group are always paired. That is, these two are always both present or both not present in any given element.

For a `Beginning` element, parameters of this group are user settable except for the `dvoltage_ref` parameter. For all other element types, except for `dvoltage_ref` and `extra_dtime_ref`, the parameters of this group are calculated by the *AcceleratorLattice.jl* bookkeeping routines and are not user settable.

For most elements, the values of the parameters in `DownstreamReferenceGroup` will be the same as the values in the corresponding `ReferenceGroup` parameters. That is, the value of `species_ref_downstream` in `DownstreamReferenceGroup` will be the same as the value of `species_ref` in `ReferenceGroup`, the value of `pc_ref_downstream` will be the same as `pc_ref`, etc. Elements where the reference energy (here "energy" refers to either `pc_ref`, `E_tot_ref`,  $\beta$ , or  $\gamma$ ) differs between upstream and downstream are elements with a non-zero `dvoltage_ref` and include `LCavity` and `Patch` elements. Elements where the reference energy and species differ between upstream and downstream include `Foil` and `Converter` elements.

For elements where `dvoltage_ref` is nonzero the downstream reference energy `E_tot_ref_downstream` is calculated from the upstream `E_tot_ref` via the equation

$$E\_tot\_ref\_downstream = E\_tot\_ref + dvoltage\_ref * |Q\_ref|$$

where  $|Q\_ref|$  is the magnitude of the charge of the reference particle in units of the fundamental charge. Once `E_tot_ref_downstream` has been calculated, the downstream values of `pc`,  $\beta$ , and  $\gamma$  are calculated using the standard formulas. Notice that `dvoltage_ref` is completely independent from the actual voltage seen by a particle which is set by the `voltage` parameter of the `RFGGroup`.

The downstream reference time `time_ref_downstream` is calculated via

$$time\_ref\_downstream = time\_ref + transit\_time + extra\_dtime\_ref$$

where `transit_time` is the time to transit the element assuming a straight line trajectory and a linear energy change throughout the element. The general formula for the transit time is

$$transit\_time = L * (E\_tot\_ref + E\_tot\_ref\_downstream) / (c * (pc\_ref + pc\_ref\_downstream))$$

where `L` is the length of the element and `c` is the speed of light. For elements where there is no energy change (`dvoltage_ref = 0`), the transit time calculation simplifies to

$$transit\_time = L / (\beta\_ref * c)$$

The `extra_dtime_ref` parameter in the above is ment as a correction to take into account for particle motion that is not straight or acceleration that is not linear in energy. For example, in a wiggler, `extra_dtime_ref` can be used to correct for the oscillatory nature of the particle trajectories. Since *AcceleratorLattice.jl* does not do tracking (see the discussion in §15), `extra_dtime_ref` must be calculated by the User.

## 5.22 SolenoidGroup

The components of this group are:

```
Ksol::Number      - Normalized solenoid strength.
Bsol::Number      - Solenoid field.
```

## 5.23 TrackingGroup

The components of this group are:

```
num_steps::Int    - Number of steps.
ds_step::Number   - Step length.
```

## 5.24 TwissGroup

In development

The components of this group are:

## Chapter 6

# Constructing Lattices

Note:

```
@ele qq = Quadrupole()
bl = beamline([.., qq, ..., qq, ..], ...)
```

Here changing parameters of qq will affect the parameters of the qq's in any beamline. However, lattice expansion constructs the lattice with copies of qq so changing the parameters of qq will not affect any of the copies in the lattice. This is done so that parameters in the various qq's in the lattice are independent and an therefore differ from each other.

Branch geometry is inherited from the root line. To use a line with the "wrong" geometry, create a new line using the old line with the "correct" geometry. EG `ln2 = beamline(ln1.name, [ln1], geometry = CLOSED)` `lat = Lattice([ln2])`

Note: OPEN and CLOSED are aliases for BranchGeometry.OPEN and BranchGeometry.CLOSED

\* Show how to construct a Bmad replacement line using a function.

\* Show how to get the effect of a Bmad List by replacing elements after expansion.



# Chapter 7

## Multipass

### 7.1 Multipass Fundamentals

Multipass lines are a way to handle the bookkeeping when different elements being tracked through represent the same physical element. For example, consider the case where dual ring colliding beam machine is to be simulated. In this case the lattice file might look like:

```
ring1 = beamline("r1", [..., IR_region, ...])
ring2 = beamline("r2", [..., reverse(IR_region), ...])
IR_region = beamline("IR", [Q1, ...])
lat = Lattice("dual_ring", [ring1, ring2])
```

[The `reverse` construct means go through the line backwards (§??)] In this case, the Q1 element in `ring1` and the Q1 element in `ring2` represent the same physical element. Thus the parameters of both the Q1s should be varied in tandem. This can be done automatically using `multipass`. The use of `multipass` simplifies lattice and program development since the bookkeeping details are left to the *AcceleratorLattice.jl* bookkeeping routines.

To illustrate how `multipass` works, consider the example of an Energy Recovery Linac (ERL) where the beam will recirculate back through the LINAC section to recover the energy in the beam before it is dumped. In *AcceleratorLattice.jl*, this situation can be simulated by designating the LINAC section as `multipass`. The lattice file might look like:

```
@ele RF1 = LCavity(...)
linac = beamline["linac", [RF1, ...], multipass = true)
erl_line = beamline("erl", [linac, ..., linac])
lat = Lattice("erl", [erl_line])
rf1p2 = find_ele(lat, "RF1!mp1")
rf1p2.multipass_phase = pi
```

The beamline called `linac` is designated as `multipass`. This `linac` line appears twice in the line `erl_line` and `erl_line` is the root line for lattice expansion. In branch 1 of the lattice, which will be a tracking branch, there will be two elements derived from the RF1 element:

```
RF1!mp1, ..., RF1!mp2, ...
```

Since the two elements are derived from a `multipass` line, they are given unique names by adding an `!mpN` suffix where N is an integer. These types of elements are known as `multipass_slave` elements. In addition to the `multipass_slave` elements there will be a `multipass_lord` element (that doesn't get tracked through) called RF1 in the `multipass_lord` branch of the lattice (§??). Changes to the

parameters of the lord RF1 element will be passed to the slave elements by the *AcceleratorLattice.jl* bookkeeping routines. Assuming that the phase of RF1!mp1 gives acceleration, to make RF1!mp2 decelerate, the `multipass_phase` parameter of RF1!mp2 is set to `pi`. This is the one parameter that *AcceleratorLattice.jl*'s bookkeeping routines will not touch when transferring parameter values from RF1 to its slaves. Notice that the `multipass_phase` parameter had to be set after the lattice is formed using the `expand` function (§??). This is true since RF1!mp2 does not exist before the lattice is expanded. `multipass_phase` is useful with relative time tracking §??. However, `multipass_phase` is “unphysical” and is just a convenient 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 §??.

Multiple elements of the same name in a multipass line are considered physically distinct. Example:

```
m_line = beamline("m", [A, A, B], multipass = true)
u_line = beamline("u", [m_line, m_line])
lat = Lattice("lat", [u_line])
```

In this example, branch 1 of the lattice is:

```
A!mp1, A!mp1, B!mp1, A!mp2, A!mp2, B!mp2
```

In the `multipass_lord` branch of the lattice there will be two multipass lords called A and one another lord called B. That is, there are three physically distinct elements in the lattice. The first A lord controls the 1<sup>st</sup> and 4<sup>th</sup> elements in branch 1 and the second A lord controls the 2<sup>nd</sup> and 5<sup>th</sup> elements. If `m_line` was *not* marked `multipass`, branch 1 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 were substituted directly in place of the subline in the containing line. For example:

```
a_line = beamline("a", [A])
m_line = beamline("m", [a_line, a_line], multipass = true)
u_line = beamline("u", [m_line, m_line])
lat = Lattice("lat", [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 (§??). Example:

```
m_line = beamline("m", [A, B], multipass = true)
m2_line = beamline("m2", m_line)
@ele P = patch(...) # Reflection patch
u_line = beamline("u", [m_line, P, reverse(m2_line)])
lat = Lattice("lat", [u_line])
```

Here the tracking part of the lattice is

```
A!mp1, B!mp1, ..., B!mp2 (r), A!mp2 (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!mp n elements and similarly with B. This lattice represents the case where, when tracking, a particle goes through the `m_line` in the “forward” direction and, at the reflection patch element P, the coordinate system is reversed so that the particle is then tracked in the reverse direction through the elements of `m_line` twice. While it is possible to use reflection “—” (§??) instead of reversal (§??), reflection here does not make physical sense. Needed here is a reflection patch P (§3.24) between reversed and unreversed elements.

The procedure for how to group lattice elements into multipass slave groups which represent the same physical element is as follows. For any given element in the lattice, this element has some line it came from. Call this line  $L_0$ . The  $L_0$  line in turn may have been contained in some other line  $L_1$ , etc. The chain of lines  $L_0, L_1, \dots, L_n$  ends at some point and the last (top) line  $L_n$  will be one of the root lines listed in the `use` statement (§??) in the lattice file. 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, that element will not be a multipass slave. For elements that have an associated  $L_m$  multipass line, all elements that have a common  $L_m$  line and have the same element index when  $L_m$  is expanded are put into a multipass slave group (for a given line the element index with respect to that line is 1 for the first element in the expanded line, the second element has index 2, etc.). For example, using the example above, the first element of the lattice, `A!mp1`, has the chain:

```
m_line, u_line
```

The last element in the lattice, `(A!mp2)`, has the chain

```
m_line, m2_line, u_line
```

For both elements the  $L_m$  line is `m_line` and both elements are derived from the element with index 1 with respect to `m_line`. Therefore, 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 = beamline("a", [A], multipass = true)
m_line = beamline("m", [a_line, a_line, B], multipass = true)
u_line = beamline("u", [m_line, m_line])
lat = Lattice("lat", [u_line])
```

In this case, branch 1 of the lattice will be:

```
A!mp1, A!mp2, B!mp1, A!mp3, A!mp4, B!mp2
```

There will be two lord elements representing the two physically distinct elements A and B. The A lord element will control the four `A!mpN` elements in the tracking part of the lattice. The B lord will control the two `B!mpN` 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 = beamline("m", [A, A, B], multipass = true)
u_line = beamline([m_line])
lat = Lattice("lat", [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 floor coordinates (§11.4) of the slaves of a given multipass lord are not constrained by `AcceleratorLattice.jl` 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 The Reference Energy in a Multipass Line

Consider the lattice where the tracking elements are

```
A!mp1, C, A!mp2
```

where `A!mp1` and `A!mp2` are multipass slaves of element A and C is a `lcavity` element with some finite voltage. In this case, the reference energy calculation (§??) where the reference energy of an element is inherited from the previous element, assigns differing reference energies to `A!mp1` and `A!mp2`. In such a situation, what should be the assigned reference energy for the multipass lord element A? `AcceleratorLattice.jl` calculates the lord reference energy in one of two ways. If, in the lattice file, `e_tot_ref` or `pc_ref` is set for the multipass lord element, that setting will be used. If the reference energy (or reference momentum) is not set in the lattice file, the reference energy of the lord is set equal to the reference energy of the first pass slave element.





## Chapter 8

# Superposition

Superposition is the insertion of elements into a lattice after the lattice has been created by the `expand` function. Superposition is beneficial for various purposes. A common use of superposition is to insert **marker** elements within other elements. For example, placing a marker element in the middle of a quadrupole. Another use case is when the field in some region is due to the overlap of several elements. For example, a quadrupole magnet inside a larger solenoid magnet.

### 8.1 Superposition on a Drift

A simple example illustrates how superposition works (also see section §??):

```
using AcceleratorLattice
@ele dd = Drift(L = 12)
@ele bb = BeginningEle(species_ref = species("proton"), pc_ref = 1e11)
@ele ss = Solenoid(L = 1)
zline = beamline("z", [bb, dd])
lat = Lattice("lat", zline)

ref_ele = find_eles(lat, "dd")
superimpose!(ss, ref_ele, offset = 0.2)
```

Before superposition, branch 1 of lattice `lat` looks like

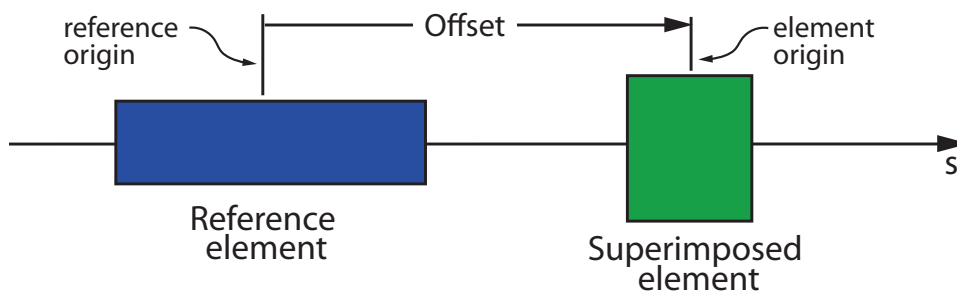


Figure 8.1: The superposition `offset` is the longitudinal  $s$ -distance from the origin point of the reference element to the origin point of the element being superimposed.

```

Branch 1: "z" geometry => open          L          s          s_downstream
1  "bb"          BeginningEle          0.000000    0.000000 ->    0.000000
2  "dd"          Drift                  12.000000    0.000000 ->   12.000000
3  "end_ele"     Marker                  0.000000   12.000000 ->   12.000000

```

The `superimpose!` function has the signature

```

function superimpose!(super_ele::Ele, ref::T;
    ele_origin::BodyLocationSwitch = b_center, offset::Real = 0,
    ref_origin::BodyLocationSwitch = b_center, wrap::Bool = true)
    where E <: Ele, T <: Union{Branch, Ele, VectorBranch, VectorE}

```

The `superimpose`

statement inserts a copy of the element `ss` in the lattice.

After insertion, branch 1 looks like:

```

Branch 1: "z" geometry => open          L          s          s_downstream
1  "bb"          BeginningEle          0.000000    0.000000 ->    0.000000
2  "dd!1"        Drift                  5.700000    0.000000 ->    5.700000
3  "ss"          Solenoid                1.000000    5.700000 ->    6.700000
4  "dd!2"        Drift                  5.300000    6.700000 ->   12.000000
5  "end_ele"     Marker                  0.000000   12.000000 ->   12.000000

```

The insertion of the `ss` copy is within the `drift` named `dd`. The position

With superpositions, `Drift` elements are handled differently from other elements. This is done to simplify the bookkeeping code. T

Rules:

- The `super_ele` element cannot be a `Drift`.
- The `bookkeeper!` function must be called after any superpositions or

```

Branch 1: "z" geometry => open L s s_downstream 1 "bb" BeginningEle 0.000000 0.000000 -> 0.000000
2 "dd" Drift 12.000000 0.000000 -> 12.000000 3 "end_ele" Marker 0.000000 12.000000 -> 12.000000
using AcceleratorLattice
@ele qq = Quadrupole(L = 4)
@ele dd = Drift(L = 12)
@ele ss = Solenoid(L = 1)
@ele bb = BeginningEle(species_ref = species("proton"), pc_ref = 1e11)
zline = beamline("z", [bb, qq, dd])
lat = Lattice("lat", zline)

```

```

ref_ele = find_eles (lat, "dd")
superimpose!(ss, ref_ele, offset = 0.2)

```

parameter 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 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 (§??) 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. (a single # has no special meaning other than the fact that *Bmad* uses it for mangling names. This is opposed to a double ## which is used to denote the  $N^{th}$  instance of an element (§??). 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. ??B.

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 (§??) 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 super_slave` 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. An `em_field super_slave` element will also be created if any of the superimposing elements have a non-zero orientation (§??) since it is not, in general, possible to construct a slave element that properly mimics the effect of a non-zero orientation.

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. ??.

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             ! Suppress superposition.
```

Superposition happens at the end of parsing so the last set of the `superimpose` for an element will override previous settings.

It is also possible to superimpose an element using the `superimpose` command which has the syntax:

```
superimpose, element = <ele-name>, ...
```

With the same optional superposition parameters (`ref`, `offset`, etc.) given below. Example:

```
superimpose, element = Q1, ref = B12, offset = 1.3,
              ele_origin = beginning, ref_origin = end
```

Note: Superposition using the `superimpose` statement allows superimposing the same element with multiple reference elements and/or multiple offsets. The drawback is that superposition using the `superimpose` statement may not be switched off later in the lattice file.

The placement of a superimposed element is illustrated in Fig. 8.1. 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 parameters that determine these three quantities are:

```
create_jumbo_slave = <Logical>      ! See §8.3
wrap_superimpose   = <Logical>      ! Wrap if element extends past lattice ends?
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 (§1.8)). Wild card characters (§??) can be used with `ref`. If `ref` matches to multiple elements (which may also happen without wild card characters if there are multiple elements with the name given by `ref`) in the lattice a superposition will be done, one for each match.

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 of the origin of the element being superimposed relative to the origin of the reference element. The default offset is zero. A positive offset moves the element being superimposed in the `downstream` direction if the reference element has a normal longitudinal orientation (§??) and vice versa for the reference element has a reversed longitudinal orientation.

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 parameters of exactly one element. In this case, the controlled element is used as the reference element.

By default, 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 = 4, superimpose, offset = 1
machine: line = (d)
use, machine
```

The lattice will have five elements in the tracking section:

	Element	Key	Length
0)	BEGINNING	Beginning_ele	0

```

1)   Q#2       Quadrupole    3   ! Slave representing beginning of Q element
2)   D#1       Drift        6
3)   Q#1       Quadrupole    1   ! Slave representing end of Q element
4)   END       Marker        0

```

And the lord section of the lattice will have the element Q.

To not wrap an element that is being superimposed, set the `wrap_superimpose` logical to `False`. Following the above example, if the definition of `q` is extended by adding `wrap_superimpose`:

```
q: quad, l = 4, superimpose, offset = 1, wrap_superimpose = F
```

In this instance there are four elements in the tracking section:

	Element	Key	Length
0)	BEGINNING	Beginning_ele	0
1)	Q	Quadrupole	4
2)	D#1	Drift	7
4)	END	Marker	0

And the lord section of the lattice will not have any elements.

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

```

The order of the elements in the lattice will be

S1, Z, S2

If `ref_origin` is not present or set to `center`, the ordering of the elements will be arbitrary.

If a zero length element is being superimposed at a spot where there are other zero length elements, the general rule is that the element will be placed as close as possible to the reference element. For example:

```

S1: marker, superimpose, offset = 1
S2: marker, superimpose, offset = 1

```

In this case, after S1 is superimposed at  $s = 1$  meter, the superposition of S2 will place it as close to the reference element, which in this case is the BEGINNING elements at  $s = 0$ , as possible. Thus the final order of the superimposed elements is:

S2, S1

To switch the order while still superimposing S2 second one possibility is to use:

```

S1: marker, superimpose, offset = 1
S2: marker, superimpose, ref = S1, ref_origin = end

```

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` (§??) 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 (§??) 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.

To simplify bookkeeping, a drift element may not be superimposed. Additionally, since drifts can disappear during superposition, to avoid unexpected behavior the superposition reference element may not be the  $N^{th}$  instance of a drift with a given name. For example, if there are a number of drift elements in the lattice named `a_drft`, the following is not allowed:

```
my_oct: octupole, ..., superimpose, ref = a_drft##2 ! This is an error
```

When the parameters of a `super_slave` are computed from the parameters of its `super_lords`, some types of parameters may be “missing”. For example, it is, in general, not possible to set appropriate aperture parameters (§??) of a `super_slave` if the lords of the slave have differing aperture settings. When doing calculations, *Bmad* will use the corresponding parameters stored in the lord elements to correctly calculate things.

When superposition is done in a line where there is `element reversal` (§??), 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

Superposition with `line reflection` (§??) works the same way as line reversal.

The `no_superposition` statement (§??) can be used to turn off superpositioning

## 8.2 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`.

## 8.3 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 §??, 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. ??C. 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 parameters of each lord will be set to show the position of the lord within the slave. These two parameters 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` (§??) will, by necessity, have to be `runge_kutta`, or `time_runge_kutta` and the `mat6_calc_method` (§??) 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 parameter as discussed in §??.

## 8.4 Changing Element Lengths when there is Superposition

When a program is running, if `group` (§??) or `overlay` (§??) 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 were 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 §??. 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 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 §??. 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}
```

\*) Difference from *Bmad*: Superposition is always done after lattice expansion.

\*) Superimposing using the same given Drift as a reference element multiple times is not allowed (unlike *Bmad*). Instead, superimpose a Null ele at the beginning or end of the drift and then use that as the reference. At the end, remove the Null element



## Chapter 9

# Customizing Lattices

### Custom Lattice Element Parameters

Custom parameters may be added to lattice elements but methods need to be created to tell *AcceleratorLattice.jl* how to handle these parameters.

- \* Define element parameter group

- \* Need to extend: `ELE_PARAM_INFO_DICT PARAM_GROUPS_LIST param_group_info`

### Custom Lattice Elements

- \* Need to extend: `ele_param_groups`



# Chapter 10

## Utilities

This chapter covers utility functions.

### 10.1 Iteration Over Elements in a Lattice

Iteration over a Region (see `traversal.jl`)

### 10.2 Searching for Lattice Elements

### 10.3 Particle Properties Conversion Functions

### 10.4 Math Utilities

### 10.5 Lattice Manipulation

### 10.6 Miscellaneous Utilities



## Part II

# Conventions and Physics



# Chapter 11

## Coordinates

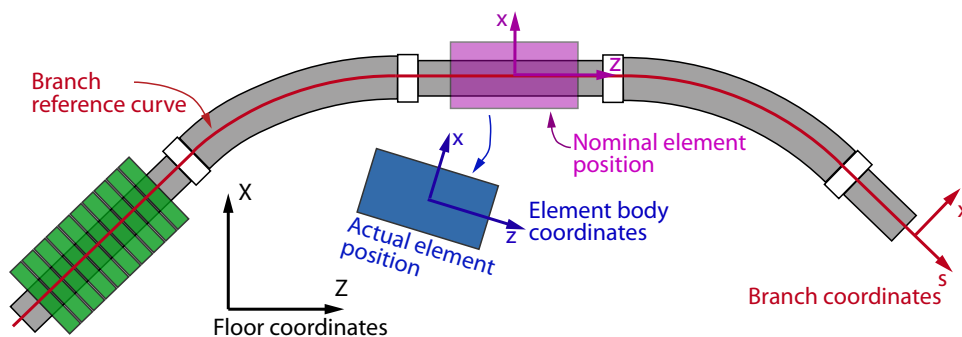


Figure 11.1: The **floor** coordinate system is independent of the accelerator. The **branch** curvilinear coordinate system follows the bends of the accelerator. The **branch reference curve** is the  $x = y = 0$  curve of the curvilinear coordinate system. Each lattice element has **element body** coordinates which, if the element has no alignment shifts (not “misaligned”), is the same as the **branch** coordinates.

### 11.1 Coordinate Systems

*AcceleratorLattice.jl* uses three coordinate systems as illustrated in Fig. 11.1. First, the **floor** coordinates are coordinates independent of the accelerator. The position of the accelerator itself as well as external objects like the building the accelerator is in may be described using **floor** coordinates.

It is inconvenient to describe the position lattice elements and the position of a particle beam using the **floor** coordinate system so, for each tracking lattice branch, a “**branch**” coordinate system is used (§??). This curvilinear coordinate system defines the nominal position of the lattice elements. The relationship between the **branch** and **floor** coordinate systems is described in §11.4. Non-tracking branches do not have an associated branch coordinate system.

The **branch reference curve** is the  $x = y = 0$  curve of the curvilinear coordinate system. The branch reference curve does not have to be continuous. A non-continuous reference curve is used, for example, to connect branches together.

The “nominal” position of a lattice element is the position of the element without any position and orientation shifts (§??) which are sometimes referred to as “misalignments”. Each lattice element has

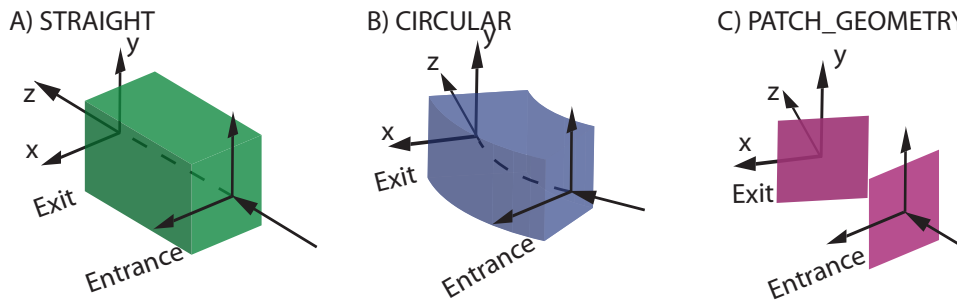


Figure 11.2: Lattice elements can be imagined as “LEGO blocks” which fit together to form the branch coordinate system. How elements join together is determined in part by their entrance and exit coordinate frames. A) For straight line elements the entrance and exit frames are colinear. B) For bends 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 frame.

“**element body**” (or just “**body**”) coordinates which are attached to the physical element and the electric and magnetic fields of an element are described with respect to **body** coordinates. If an element has no alignment shifts, the **body** coordinates of the element are aligned with the **branch** coordinates. The transformation between **branch** and **body** coordinates is given in §11.5.

## 11.2 Element Entrance and Exit Coordinates

As discussed in the next section, the branch coordinate system is constructed starting with the first element in a lattice tracking branch and then building up the coordinate system element-by-element. All elements in a tracking branch have an “**entrance**” and an “**exit**” coordinate frame as illustrated in Fig. 11.2. These coordinate frames are attached to the element and are part of the **element body coordinates**. The one element that does not have entrance and exit coordinate frames is the **Girder** (§3.14) element which supports other elements but these elements are stored in a non-tracking branch. There is also an exception where, if there is a **fiducial** element (or elements) in the branch line, the construction of the branch coordinate system starts at the **fiducial** elements.

Most elements have a “straight” geometry as shown in Fig. 11.2A. That is, the reference curve through the element is a straight line segment with the  $x$  and  $y$  axes always pointing in the same direction. For a **Bend** element (§3.4), the reference curve is a segment of a circular arc as shown in Fig. 11.2B. With the `ref_tilt` parameter of a bend set to zero, the rotation axis between the entrance and exit frames is parallel to the  $y$ -axis (§11.4). For **Patch** (§3.24) and **floor\_shift** (§3.11), and **fiducial** (§3.10) elements (Fig. 11.2C), the exit face can be arbitrarily oriented with respect to the entrance end. For the **FloorShift** and **fiducial** elements the interior reference curve between the entrance and exit faces is not defined. For the **Patch** element, the interior reference curve is dependent upon certain **Patch** element parameter settings (§3.24).

## 11.3 Branch Coordinates Construction

Assuming for the moment that there are no **Fiducial** elements present, the construction of the branch coordinate system starts at the **BeginningEle** element (§3.3) at the start of a branch. If the branch is a **root branch** (§1.9), The orientation of the beginning element within the floor coordinate system (§11.1)



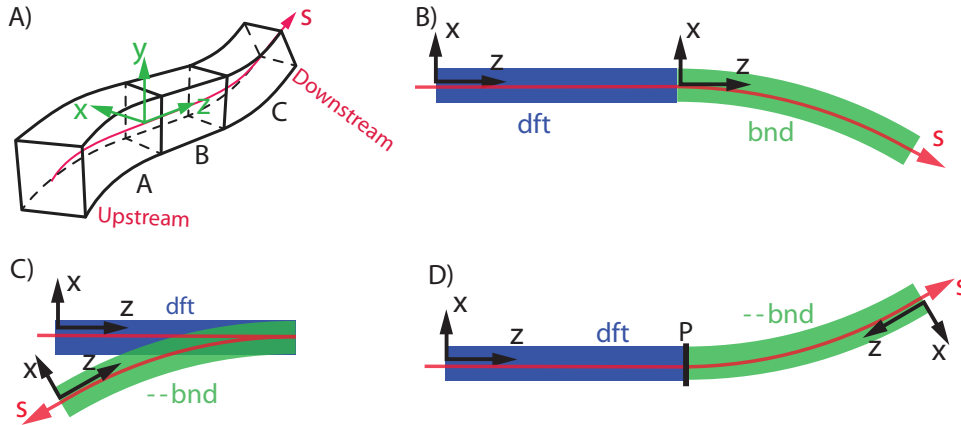


Figure 11.3: A) The branch coordinates are constructed by connecting the **downstream** reference frame of one element with the **upstream** reference frame of the next element in the branch. Coordinates shown is for the mating of element A to element B. B) Example with drift element **dft** followed by a bend **bnd**. Both elements are unreversed. C) Similar to (B) but in this case element **bnd** is reversed. D) Similar to (C) but in this case a reflection patch has been added in between **dft** and **bnd**. In (B), (C), and (D) the  $(x, z)$  coordinates are drawn at the **entrance** end of the elements. The  $y$  coordinate is always out of the page.

can be fixed by setting `FloorPositionGroup` parameters (§??) in the `BeginningEle` element. If the branch is not a **root** branch, the position of the beginning element is determined by the position of the `Fork` element from which the branch forks from. The default value of  $s$  at the `BeginningEle` element is zero for both root and non-root branches.

Once the beginning element in a branch is positioned, succeeding elements are concatenated together to form the branch coordinate system. All elements have an “**upstream**” and a “**downstream**” end as shown in Fig. 11.3A. The **downstream** end of an element is always farther (at greater  $s$ ) from the beginning element than the **upstream** end of the element. Normally, particles will travel in the  $+s$  direction, so particles will enter an element at the upstream end and exit at the downstream end.

If there are `Fiducial` elements, the branch coordinates are constructed beginning at these elements working both forwards and backwards along the branch.

If an element is not reversed (§??), the element’s **upstream** end is the same as the element’s **entrance** end (Fig. 11.2) and the **downstream** end is the same as the element’s **exit** end. If an element is reversed, the element’s **exit** end will be **upstream** end and the element’s **entrance** end will be the **downstream** end. That is, for a reversed element, particles traveling downstream will enter at the element’s **exit** end and will exit at the **entrance** end.

The procedure to connect elements together to form the branch coordinates is to ignore alignment shifts and mate the downstream reference frame of the element with the upstream reference frame of the next element in the branch so that the  $(x, y, z)$  axes coincide. If there are alignment shifts, the **entrance** and **exit** frames will move with the element. However, this does not affect the branch coordinate system. This is illustrated in Fig. 11.3. Fig. 11.3A shows the general situation with the downstream frame of element A mated to the upstream frame of element B. Figures 11.3B-C show branches constructed from the following lattice file:

```
@ele dft = Drift(L = 2)
@ele bnd = Bend(l = 2, g = pi/12)
@ele p = Patch(x_rot = pi)           ! Reflection patch.
```

```

BL = BeamLine([dft, bnd])           ! Fig. 11.3B. No reversal.
CL = BeamLine([dft, reverse(bnd)])  ! Fig. 11.3C. Illegal. Do not use!
DL = BeamLine([dft, p, reverse(bnd)]) ! Fig. 11.3D. Has reversal.

```

The  $(x, z)$  coordinates are drawn at the entrance end of the elements and  $z$  will always point towards the element's exit end. Fig. 11.3B shows the branch constructed from BL containing an unreversed drift named `dft` connected to an unreversed bend named `bnd`. Fig. 11.3C shows the branch constructed from CL. This is like BL except here element `bnd` is reversed. This gives an unphysical situation since a particle traveling through `dft` will “fall off” when it gets to the end. Fig. 11.3D shows the branch constructed from DL. Here a “reflection” patch P (§11.4.5) has been added to get a plausible geometry. In this case the patch rotates the coordinate system around the  $y$ -axis by  $180^\circ$  (leaving the  $y$ -axis invariant). It is always the case that a reflection patch is needed between reversed and unreversed elements.

Notes:

- If the first element after the `BeginningEle` element at the start of a branch is reversed, the `BeginningEle` element will be marked as reversed so that a reflection patch is not needed in this circumstance.
- Irrespective of whether elements are reversed or not, the branch  $(x, y, z)$  coordinate system at all  $s$ -positions will always be a right-handed coordinate system.
- Care must be taken when using reversed elements. For example, if the field of the `bnd` element in BL is appropriate for, say, electrons, that is, electrons will be bent in a clockwise fashion going through `bnd`, then an electron going through DL will be lost in the bend (the  $y$ -axis and hence the field is in the same direction for both cases so electrons will still be bent in a clockwise fashion but with DL a particle needs to be bent counterclockwise to get through the bend). To get a particle through the bend, positrons must be used.
- A reflection patch that rotated the coordinates, for example, around the  $x$ -axis by  $180^\circ$  (by setting `x_rot` to `pi`) would also produce a plausible geometry.

## 11.4 Floor Coordinates

The Cartesian `floor` coordinate system is the coordinate system “attached to the earth” that is used to describe the branch coordinate system. Following the *MAD* convention, the `floor` coordinate axes are labeled  $(X, Y, Z)$ . Conventionally,  $Y$  is the “vertical” coordinate and  $(X, Z)$  are the “horizontal” coordinates. To describe how the branch coordinate system is oriented within the floor coordinate system, each point on the  $s$ -axis of the branch 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 branch coordinate axes as shown in Fig. 11.4. These three angles are defined as follows:

- $\theta(s)$  **Azimuth (yaw) angle:** Angle in the  $(X, Z)$  plane between the  $Z$ -axis and the projection of the  $z$ -axis onto the  $(X, Z)$  plane. Corresponds to the `y_rot` element parameter (§??). A positive angle of  $\theta = \pi/2$  corresponds to the projected  $z$ -axis pointing in the negative  $X$ -direction.
- $\phi(s)$  **Pitch (elevation) angle:** Angle between the  $z$ -axis and the  $(X, Z)$  plane. Corresponds to the `x_rot` element parameter (§??). A positive angle of  $\phi = \pi/2$  corresponds to the  $z$ -axis pointing in the positive  $Y$  direction.
- $\psi(s)$  **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. Corresponds to the `tilt` element parameter (§??). A positive  $\psi$  forms a right-handed screw with the  $z$ -axis.

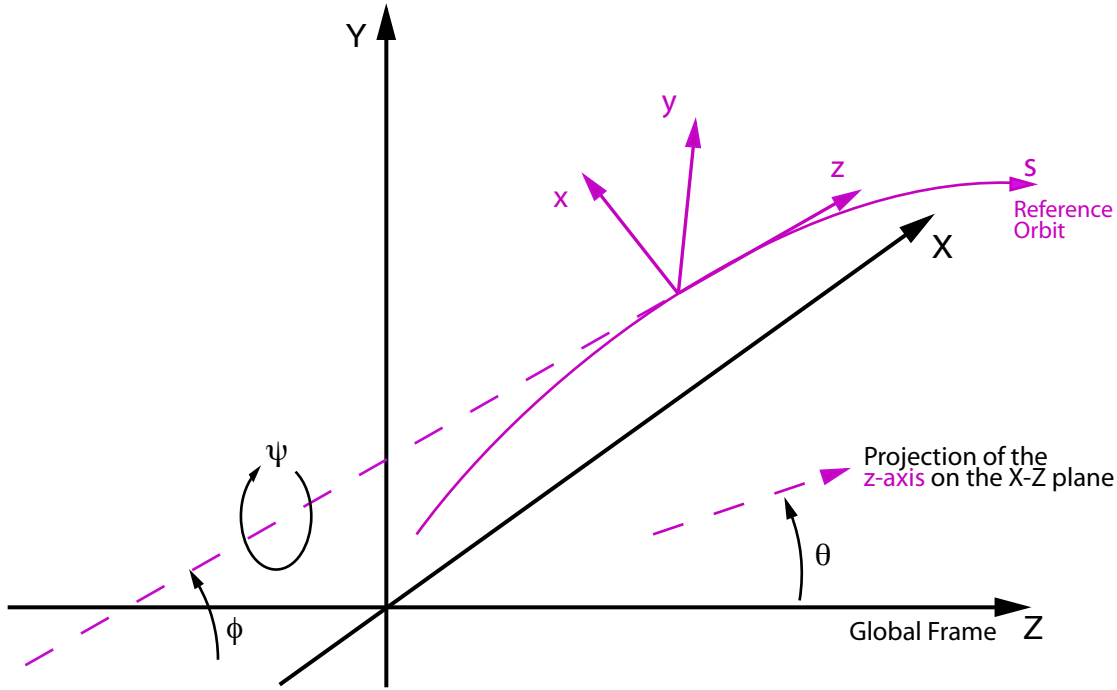


Figure 11.4: The branch coordinate system (purple), which is a function of  $s$  along the branch reference curve, is described in the floor coordinate system (black) by a position  $(X(s), Y(s), Z(s))$  and by angles  $\theta(s)$ ,  $\phi(s)$ , and  $\psi(s)$ .

By default, at  $s = 0$ , the reference curve'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. If the lattice has no vertical bends (the `ref_tilt` parameter (§3.4) of all bends are zero), the  $y$ -axis will always be in the vertical  $Y$  direction and the  $x$ -axis will lie in the horizontal  $(X, Z)$  plane. In this case,  $\theta$  decreases as one follows the reference curve 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  $\phi$  and  $\psi$  will both be zero. The `beginning` statement (§??) in a lattice file can be used to override these defaults.

Following *MAD*, the floor position of an element is characterized by a vector  $\mathbf{V}$

$$\mathbf{V} = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \quad (11.1)$$

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 branch coordinate axes in the order  $(x, y, z)$ .  $\mathbf{W}$  can be expressed in terms of the orientation angles  $\theta$ ,  $\phi$ , and  $\psi$  via the formula

$$\begin{aligned} \mathbf{W} &= \mathbf{R}_y(\theta) \mathbf{R}_x(-\phi) \mathbf{R}_z(\psi) \\ &= \begin{pmatrix} \cos \theta \cos \psi - \sin \theta \sin \phi \sin \psi & -\cos \theta \sin \psi - \sin \theta \sin \phi \cos \psi & \sin \theta \cos \phi \\ \cos \phi \sin \psi & \cos \phi \cos \psi & \sin \phi \\ -\cos \theta \sin \phi \sin \psi - \sin \theta \cos \psi & \sin \theta \sin \psi - \cos \theta \sin \phi \cos \psi & \cos \theta \cos \phi \end{pmatrix} \end{aligned} \quad (11.2)$$

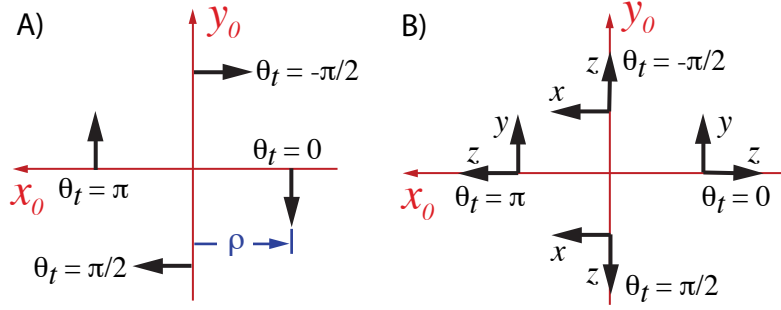


Figure 11.5: A) Rotation axes (bold arrows) for four different `ref_tilt` angles of  $\theta_t = 0, \pm\pi/2$ , and  $\pi$ .  $(x_0, y_0, z_0)$  are the branch coordinates at the entrance end of the bend with the  $z_0$  axis being directed into the page. Any rotation axis will be displaced by a distance 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  $\pi/2$ .

where

$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{pmatrix}, \quad \mathbf{R}_x(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \phi & -\sin \phi \\ 0 & \sin \phi & \cos \phi \end{pmatrix}, \quad \mathbf{R}_z(\psi) = \begin{pmatrix} \cos \psi & -\sin \psi & 0 \\ \sin \psi & \cos \psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (11.3)$$

Notice that these are Tait-Bryan angles and not Euler angles.

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  $\beta$

$$\mathbf{W} = \begin{pmatrix} \cos \beta + u_x^2 (1 - \cos \beta) & u_x u_y (1 - \cos \beta) - u_z \sin \beta & u_x u_z (1 - \cos \beta) + u_y \sin \beta \\ u_y u_x (1 - \cos \beta) + u_z \sin \beta & \cos \beta + u_y^2 (1 - \cos \beta) & u_y u_z (1 - \cos \beta) - u_x \sin \beta \\ u_z u_x (1 - \cos \beta) - u_y \sin \beta & u_z u_y (1 - \cos \beta) + u_x \sin \beta & \cos \beta + u_z^2 (1 - \cos \beta) \end{pmatrix} \quad (11.4)$$

### 11.4.1 Lattice Element Positioning

*AcceleratorLattice.jl*, again following *MAD*, computes  $\mathbf{V}$  and  $\mathbf{W}$  by starting at the first element of the lattice and iteratively using the equations

$$\mathbf{V}_i = \mathbf{W}_{i-1} \mathbf{L}_i + \mathbf{V}_{i-1}, \quad (11.5)$$

$$\mathbf{W}_i = \mathbf{W}_{i-1} \mathbf{S}_i \quad (11.6)$$

$\mathbf{L}_i$  is the displacement vector for the  $i^{th}$  element and matrix  $\mathbf{S}_i$  is the rotation of the branch coordinate system of the exit end with respect to the entrance end. For clarity, the subscript  $i$  in the equations below will be dropped. For all elements whose reference curve through them is a straight line, the corresponding  $\mathbf{L}$  and  $\mathbf{S}$  are

$$\mathbf{L} = \begin{pmatrix} 0 \\ 0 \\ L \end{pmatrix}, \quad \mathbf{S} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad (11.7)$$

Where  $L$  is the length of the element.

For a `bend`, the axis of rotation is dependent upon the bend's `ref_tilt` angle (§??) as shown in Fig. 11.5A. 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 branch 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. 11.5B 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  $\theta$ .

For a bend,  $\mathbf{S}$  is given using Eq. (11.4) with

$$\begin{aligned}\mathbf{u} &= (-\sin \theta_t, -\cos \theta_t, 0) \\ \beta &= \alpha_b\end{aligned}\tag{11.8}$$

where  $\theta_t$  is the `ref_tilt` angle. The  $\mathbf{L}$  vector for a bend is given by

$$\mathbf{L} = \mathbf{R}_z(\theta_t) \tilde{\mathbf{L}}, \quad \tilde{\mathbf{L}} = \begin{pmatrix} \rho(\cos \alpha_b - 1) \\ 0 \\ \rho \sin \alpha_b \end{pmatrix}\tag{11.9}$$

where  $\alpha_b$  is the bend `angle` (§3.4) and  $\rho$  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

$$\mathbf{S} = \mathbf{R}_z(\theta_t) \mathbf{R}_y(-\alpha_b) \mathbf{R}_z(-\theta_t)\tag{11.10}$$

The bend transformation above is so constructed that the transformation is equivalent to rotating the branch 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  $\pi$ ) the  $y$ -axis is the rotation axis and the  $y$ -axis of the branch coordinates before the bend will be parallel to the  $y$ -axis of the branch coordinates after the bend as shown in Fig. 11.5. That is, a lattice with only bends with  $\theta_t = 0$  or  $\pi$  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.

### 11.4.2 Position Transformation When Transforming Coordinates

A point  $\mathbf{Q}_g = (X, Y, Z)$  defined in the floor coordinate system, when expressed in the coordinate system defined by  $(\mathbf{V}, \mathbf{W})$  is

$$\mathbf{Q}_{VW} = \mathbf{W}^{-1} (\mathbf{Q}_g - \mathbf{V})\tag{11.11}$$

This is essentially the inverse of Eq. (11.5). That is, position vectors propagate inversely to the propagation of the coordinate system.

Using Eq. (11.11) with Eqs. (11.5), and (11.6), 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

$$\mathbf{q}_i = \mathbf{S}_i^{-1} (\mathbf{q}_{i-1} - \mathbf{L}_i),\tag{11.12}$$

$$\mathbf{P}_i = \mathbf{S}_i^{-1} \mathbf{P}_{i-1}\tag{11.13}$$

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

$$\mathbf{S}^{-1} = \mathbf{S}^T\tag{11.14}$$

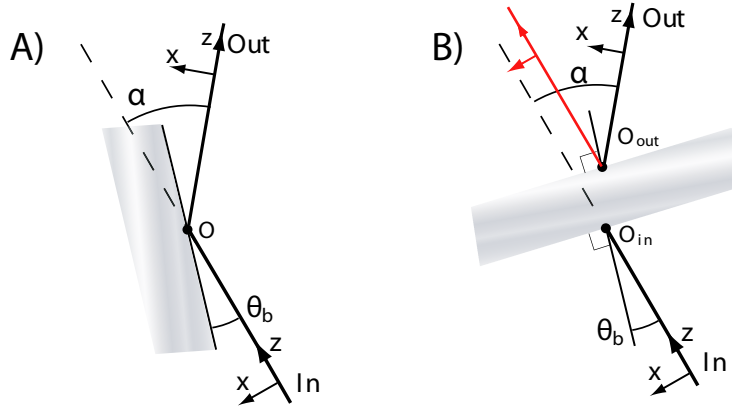


Figure 11.6: Mirror and crystal geometry. The geometry shown here is appropriate for a `ref_tilt` angle of  $\theta_t = 0$ .  $\theta_g$  is the bend angle of the incoming (entrance) ray, and  $\alpha_b$  is the total bend angle of the reference trajectory. A) Geometry for a mirror or a Bragg crystal. Point  $O$  is the origin of both the branch coordinates just before and just after the reflection/diffraction. B) Geometry for a Laue crystal. Point  $O_{out}$  is the origin of the coordinates just after diffraction is displaced from the origin  $O_{in}$  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.

### 11.4.3 Crystal and Mirror Entrance to Exit Coordinate Transformation

A `crystal` element (§??) diffracts photons and a `mirror` element (§??) reflects them. For a crystal setup for Bragg diffraction, and for a mirror, the reference curve is modeled as a zero length bend with  $\tilde{\mathbf{L}} = (0, 0, 0)$ , as shown in Fig. 11.6A. 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  $O$  in the figure) of the entrance and exit branch coordinates are the same. For Laue diffraction, the only difference is that  $\tilde{\mathbf{L}}$  is non-zero due to the finite thickness of the crystal as shown in Fig. 11.6B. This results in a separation between the entrance coordinate origin  $O_{in}$  and the exit coordinate origin  $O_{out}$ .

In all cases, the total bending angle is

$$\begin{aligned}
 \alpha_b &= \text{bragg\_angle\_in} + \text{bragg\_angle\_out} & ! \text{ Crystal, graze\_angle\_in} &= 0 \\
 \alpha_b &= \text{graze\_angle\_in} + \text{graze\_angle\_out} & ! \text{ Crystal, graze\_angle\_in} &\neq 0 \\
 \alpha_b &= 2 \text{ graze\_angle} & ! \text{ Mirror} &
 \end{aligned} \tag{11.15}$$

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 branch 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 branch  $z$ -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 §??.

#### 11.4.4 Patch and FloorShift Entrance to Exit Transformation

For **Patch** (§3.24) and **FloorShift** (§??) elements, the shift in the exit end reference coordinates is given by Eqs. (11.5) and (11.6) with

$$\mathbf{L} = \begin{pmatrix} x\_offset \\ y\_offset \\ z\_offset \end{pmatrix}$$

$$\mathbf{S} = \mathbf{R}_y(y\_rot) \mathbf{R}_x(y\_rot) \mathbf{R}_z(tilt) \quad (11.16)$$

The difference here between **Patch** and **SloorShift** elements is that, with a **Patch** element, the shift is relative to the exit end of the previous element while, for a **FloorShift** element, the shift is relative to the reference point on the origin element specified by the **origin\_ele** parameter of the **FloorShift**.

#### 11.4.5 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 **S** matrix (see Eq. (11.16)) satisfy:

$$S(3, 3) < 0 \quad (11.17)$$

Using Eq. (11.16) gives, after some simple algebra, this condition is equivalent to

$$\cos(x\_rot) \cos(y\_rot) < 0 \quad (11.18)$$

When there are a series of patches, The transformations of all the patches are concatenated together to form an effective **S** which can then be used with Eq. (11.17).

#### 11.4.6 Fiducial and Girder Coordinate Transformation

For **fiducial** and **girder** elements, the alignment of the reference coordinates with respect to “**origin**” coordinates is analogous to Eqs. (11.16). Explicitly:

$$\mathbf{L} = \begin{pmatrix} dx\_origin \\ dy\_origin \\ dz\_origin \end{pmatrix}$$

$$\mathbf{S} = \mathbf{R}_y(dtheta\_origin) \mathbf{R}_x(-dphi\_origin) \mathbf{R}_z(dpsi\_origin) \quad (11.19)$$

### 11.5 Transformation Between Branch and Element Body Coordinates

The **element body** coordinates are the coordinate system attached to an element. Without any alignment shifts, the **branch** coordinates (§??) and **element body** coordinates are the same. With alignment

shifts, the transformation between **branch** and **element body** coordinates depends upon whether the branch coordinate system is straight (§11.5.1) or bent (§11.5.2).

When tracking a particle through an element, the particle starts at the **nominal** (§11.1) upstream end of the element with the particle's position expressed in branch coordinates. Tracking from the the nominal upstream end to the actual upstream face of the element involves first transforming to element body coordinates (with  $s = 0$  in the equations below) and then propagating the particle as in a field free drift space from the particle's starting position to the actual element face. Depending upon the element's orientation, this tracking may involve tracking backwards. Similarly, after a particle has been tracked through the physical element to the actual downstream face, the tracking to the nominal downstream end involves transforming to branch coordinates (using  $s = L$  in the equations below) and then propagating the particle as in a field free drift space to the nominal downstream edge.

### 11.5.1 Straight Element Branch to Body Coordinate Transformation

For straight line elements, given a branch coordinate frame  $\Lambda_s$  with origin a distance  $s$  from the beginning of the element, alignment shifts will shift the coordinates to a new reference frame denoted  $E_s$ . Since alignment shifts are defined with respect to the middle of the element, the transformation between  $\Lambda_s$  and  $E_s$  is a three step process:

$$\Lambda_s \longrightarrow \Lambda_{\text{mid}} \longrightarrow E_{\text{mid}} \longrightarrow E_s \quad (11.20)$$

where  $\Lambda_{\text{mid}}$  and  $E_{\text{mid}}$  are the branch and element reference frames at the center of the element.

The first and last transformations from  $\Lambda_s$  to  $\Lambda_{\text{mid}}$  and from  $E_{\text{mid}}$  to  $E_s$  use Eqs. (11.5), (11.6), and (11.7) with the replacement  $L \rightarrow L/2 - s$  for the first transformation and  $L \rightarrow s - L/2$  for the third transformation. The middle transformation, by definition of the offset and rotation parameters is

$$\begin{aligned} \mathbf{L} &= \begin{pmatrix} x_{\text{offset}} \\ y_{\text{offset}} \\ z_{\text{offset}} \end{pmatrix} \\ \mathbf{S} &= \mathbf{R}_y(x_{\text{rot}}) \mathbf{R}_x(y_{\text{rot}}) \mathbf{R}_z(\text{tilt}) \end{aligned} \quad (11.21)$$

Notice that with this definition of how elements are shifted, the position of the center of a non-bend shifted element depends only on the offsets, and is independent of the rotations.

### 11.5.2 Bend Element Branch to Body Coordinate Transformation

For **Bend** elements there is a **ref\_tilt** as well as a **tilt** parameter. The former affects both the reference curve and the bend position (§??). Furthermore, **ref\_tilt** is calculated with respect to the coordinates at the beginning of the bend while **tilt**, **x\_rot**, **y\_rot**, and offsets are calculated with respect to the center of the chord connecting the ends of the bend (§??). The different reference frame used for **ref\_tilt** versus everything else means that five transformations are needed to get from the branch frame at point  $s$  to the corresponding element body frame (see Eq. (11.20)). Symbolically:

$$\Lambda_s \longrightarrow \Lambda_0 \longrightarrow \Xi_0 \longrightarrow \Xi_{\text{mid}} \longrightarrow \Omega_{c,\text{mid}} \longrightarrow \Omega_{c,0} \longrightarrow \Omega_0 \longrightarrow E_0 \longrightarrow E_s \quad (11.22)$$

1.  $\Lambda_s \longrightarrow \Lambda_0$

In branch coordinates, transform from  $\Lambda_s$ , the coordinates at a distance  $s$  from the beginning of the element, to  $\Lambda_0$  the branch coordinates at the beginning of the element. This is a rotation around the center of curvature of the bend and is given by Eqs. (11.5) and (11.6) with Eqs. (11.8) and (11.9) with the substitution  $\alpha_b \rightarrow -s/\rho$ .



2.  $\Lambda_0 \longrightarrow \Xi_0$ 

Transform from  $\Lambda_s$  to  $\Xi_0$  which is the “chord” coordinate system obtained by rotating  $\Lambda_0$  around the axis perpendicular to the bend plane such that the  $z$  axis of  $\Xi_0$  is parallel to the chord. The transformation is given by Eqs. (11.5) with  $L = (0, 0, 0)$  and  $S$  given by and (11.6) and (11.8) with  $\alpha_b \rightarrow \alpha_b/2$

3.  $\Xi_0 \longrightarrow \Xi_{\text{mid}}$ 

In chord coordinates, translate from the beginning of the chord to the end of the chord. The transformation is given by Eq. (11.7) with  $L \rightarrow L_c/2$  and  $L_c$  is the chord length.

4.  $\Xi_{\text{mid}} \longrightarrow \Omega_{c,\text{mid}}$ 

Transform from chord coordinates at the center of the chord to “shifted chord coordinates” at the same point. This is the same transform as used for straight elements:

$$\mathbf{L} = \begin{pmatrix} x\_offset \\ y\_offset \\ z\_offset \end{pmatrix}$$

$$\mathbf{S} = \mathbf{R}_y(x\_rot) \mathbf{R}_x(y\_rot) \mathbf{R}_z(tilt) \quad (11.23)$$

5.  $\Omega_{c,\text{mid}} \longrightarrow \Omega_{c,0}$ 

Transform in align shifted chord coordinates from the center of the chord back to the beginning of the chord. This is the reverse of  $\Xi_0 \longrightarrow \Xi_{\text{mid}}$ . In this case  $L \rightarrow -L_c/2$

6.  $\Omega_{c,0} \longrightarrow \Omega_0$ 

Rotate from aligned shifted chord coordinates at the entrance end to coordinates so that the  $z$ -axis is parallel to the body coordinates (tangent to the arc). This is the reverse of  $\Lambda_0 \longrightarrow \Xi_0$ . In this case  $\alpha_b \rightarrow -\alpha_b/2$

7.  $\Omega_0 \longrightarrow E_0$ 

Tilt (rotate around the  $z$ -axis) by an amount `ref_tilt` which brings the coordinate system to correspond to body coordinates at the entrance of the element.

$$\mathbf{L} = 0, \quad \mathbf{S} = \mathbf{R}_z(\theta_t) \quad (11.24)$$

8.  $E_0 \longrightarrow E_s$ 

Rotate around the center of the bend. Eqs. (11.8) and (11.9) are used with the substitutions  $\theta_t \rightarrow 0$  and  $\alpha_b \rightarrow L/\rho$ .



# Chapter 12

## Electromagnetic Fields

### 12.1 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

$$B_y(x, 0) = \sum_n B_n \frac{x^n}{n!} \quad (12.1)$$

Assuming that the reference orbit is locally straight (there are correction terms if the reference orbit is curved (§12.3)), the field is

$$\begin{aligned} B_x &= B_1 y + B_2 xy + \frac{1}{6} B_3 (3x^2 y - y^3) + \dots \\ B_y &= B_0 + B_1 x + \frac{1}{2} B_2 (x^2 - y^2) + \frac{1}{6} B_3 (x^3 - 3xy^2) + \dots \end{aligned} \quad (12.2)$$

The relation between the field  $B_n$  and the normalized field  $K_n$  is:

$$K_n \equiv \frac{q B_n}{P_0} \quad (12.3)$$

where  $q$  is the charge of the reference particle (in units of the elementary charge), 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  $\rho$  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 (§??) present.

The kicks  $\Delta p_x$  and  $\Delta p_y$  that a particle experiences going through a multipole field is

$$\Delta p_x = \frac{-q L B_y}{P_0} \quad (12.4)$$

$$= -K_0 L - K_1 L x + \frac{1}{2} K_2 L (y^2 - x^2) + \frac{1}{6} K_3 L (3xy^2 - x^3) + \dots$$

$$\Delta p_y = \frac{q L B_x}{P_0} \quad (12.5)$$

$$= K_1 L y + K_2 L xy + \frac{1}{6} K_3 L (3x^2 y - y^3) + \dots$$

A positive  $K_1L$  quadrupole component gives horizontal focusing and vertical defocusing. The general form is

$$\Delta p_x = \sum_{n=0}^{\infty} \frac{K_n L}{n!} \sum_{m=0}^{2m \leq n} \binom{n}{2m} (-1)^{m+1} x^{n-2m} y^{2m} \quad (12.6)$$

$$\Delta p_y = \sum_{n=0}^{\infty} \frac{K_n L}{n!} \sum_{m=0}^{2m \leq n-1} \binom{n}{2m+1} (-1)^m x^{n-2m-1} y^{2m+1} \quad (12.7)$$

where  $\binom{a}{b}$  (“a choose b”) denotes a binomial coefficient.

The above equations are for fields with a normal component only. If a given multipole field of order  $n$  has normal  $B_n$  and skew  $S_n$  components and is rotated in the  $(x, y)$  plane by an angle  $T_n$ , the magnetic field at a point  $(x, y)$  can be expressed in complex notation as

$$B_y(x, y) + iB_x(x, y) = \frac{1}{n!} (B_n + iS_n) e^{-i(n+1)T_n} e^{in\theta} r^n \quad (12.8)$$

where  $(r, \theta)$  are the polar coordinates of the point  $(x, y)$ .

Note that, for compatibility with MAD, the  $K_0L$  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  $\theta_n$ , Another representation is using normal  $\tilde{K}_n$  and skew  $\tilde{S}_n$ . The conversion between the two are

$$\begin{aligned} \tilde{K}_n &= K_n \cos((n+1)T_n) \\ \tilde{S}_n &= K_n \sin((n+1)T_n) \end{aligned} \quad (12.9)$$

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

$$\frac{qL}{P_0} (B_y + iB_x) = (b_n + ia_n) (x + iy)^n \quad (12.10)$$

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  $(K_nL, S_nL, T_n)$  is

$$b_n + ia_n = \frac{1}{n!} (K_nL + iS_nL) e^{-i(n+1)T_n} \quad (12.11)$$

In the case where  $S_nL = 0$

$$K_nL = n! \sqrt{a_n^2 + b_n^2} \quad (12.12)$$

$$\tan[(n+1)T_n] = \frac{-a_n}{b_n} \quad (12.13)$$

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

$$(n+1)T_n = \frac{\pi}{2} \quad (12.14)$$

For example, a normal quadrupole rotated by  $45^\circ$  becomes a skew quadrupole.

**Reference energy** scaling is applied if the **field\_master** attribute (§??) is True for an element so that the multipole values specified in the lattice file are not reference energy normalized

$$[a_n, b_n] \longrightarrow [a_n, b_n] \cdot \frac{q}{P_0} \quad (12.15)$$

## 12.2 Electrostatic Multipole Fields

Except for the `elseparator` element, *Bmad* specifies DC electric fields using normal  $b_{en}$  and skew  $a_{en}$  components (§??). The potential  $\phi_n$  for the  $n^{th}$  order multipole is

$$\phi_n = -\text{Re} \left[ \frac{b_{en} - ia_{en}}{n+1} \frac{(x+iy)^{n+1}}{r_0^n} \right] \quad (12.16)$$

where  $r_0$  is a “measurement radius” set by the `r0_elec` attribute of an element (§??).

The electric field for the  $n^{th}$  order multipole is

$$E_x - iE_y = (b_{en} - ia_{en}) \frac{(x+iy)^n}{r_0^n} \quad (12.17)$$

Notice that the magnetic multipole components  $a_n$  and  $b_n$  are normalized by the element length, reference charge, and reference momentum (Eq. (12.10)) while their electric counterparts are not.

Using the paraxial approximation, The kick given a particle due to the electric field is

$$\frac{dp_x}{ds} = \frac{q E_x}{\beta P_0 c}, \quad \frac{dp_y}{ds} = \frac{q E_y}{\beta P_0 c} \quad (12.18)$$

Where  $\beta$  is the normalized velocity.

## 12.3 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. (12.10) and (12.17). The analysis of the multipole fields in a bend here follows McMillan[McMillan:Multipoles].

In the rest of this section, normalized coordinates  $\tilde{r} = r/\rho$ ,  $\tilde{x} = x/\rho$ , and  $\tilde{y} = y/\rho$  will be used where  $\rho$  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  $\tilde{r} = 1 + \tilde{x}$ .

An electric or magnetic multipole can be characterized by a scalar potential  $\phi$  with the field given by  $-\nabla\phi$ . The potential is a solution to Laplace’s equation

$$\frac{1}{\tilde{r}} \frac{\partial}{\partial \tilde{r}} \left( \tilde{r} \frac{\partial \phi}{\partial \tilde{r}} \right) + \frac{\partial^2 \phi}{\partial \tilde{y}^2} = 0 \quad (12.19)$$

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

$$\phi_n^r = \frac{-1}{1+n} \sum_{p=0}^{2p \leq n+1} \binom{n+1}{2p} (-1)^p F_{n+1-2p}(\tilde{r}) \tilde{y}^{2p} \quad (12.20)$$

and in the form

$$\phi_n^i = \frac{-1}{1+n} \sum_{p=0}^{2p \leq n} \binom{n+1}{2p+1} (-1)^p F_{n-2p}(\tilde{r}) \tilde{y}^{2p+1} \quad (12.21)$$

where  $\binom{a}{b}$  (“a choose b”) denotes a binomial coefficient, and  $n$  is the order number which can range from 0 to infinity.<sup>1</sup>

In Eq. (12.21) the  $F_p(\tilde{r})$  are related by

$$F_{p+2} = (p+1)(p+2) \int_1^{\tilde{r}} \frac{d\tilde{r}}{\tilde{r}} \left[ \int_1^{\tilde{r}} d\tilde{r} \tilde{r} F_p \right] \quad (12.22)$$

with the “boundary condition”:

$$\begin{aligned} F_0(\tilde{r}) &= 1 \\ F_1(\tilde{r}) &= \ln \tilde{r} \end{aligned} \quad (12.23)$$

This condition ensures that the number of terms in the sums in Eqs. (12.20) and (12.21) are finite. With this condition, all the  $F_p$  can be constructed:

$$\begin{aligned} F_1 &= \ln \tilde{r} = \tilde{x} - \frac{1}{2}\tilde{x}^2 + \frac{1}{3}\tilde{x}^3 - \dots \\ F_2 &= \frac{1}{2}(\tilde{r}^2 - 1) - \ln \tilde{r} = \tilde{x}^2 - \frac{1}{3}\tilde{x}^3 + \frac{1}{4}\tilde{x}^4 - \dots \\ F_3 &= \frac{3}{2}[-(\tilde{r}^2 - 1) + (\tilde{r}^2 + 1) \ln \tilde{r}] = \tilde{x}^3 - \frac{1}{2}\tilde{x}^4 + \frac{7}{20}\tilde{x}^5 - \dots \\ F_4 &= 3\left[\frac{1}{8}(\tilde{r}^4 - 1) + \frac{1}{2}(\tilde{r}^2 - 1) - (\tilde{r}^2 + \frac{1}{2}) \ln \tilde{r}\right] = \tilde{x}^4 - \frac{2}{5}\tilde{x}^5 + \frac{3}{10}\tilde{x}^6 - \dots \\ &\text{Etc...} \end{aligned} \quad (12.24)$$

Evaluating these functions near  $\tilde{x} = 0$  using the exact  $\tilde{r}$ -dependent functions can be problematical due to round off error. For example, Evaluating  $F_4(\tilde{r})$  at  $\tilde{x} = 10^{-4}$  results in a complete loss of accuracy (no significant digits!) when using double precision numbers. In practice, *Bmad* uses a Padé approximant for  $\tilde{x}$  small enough and then switches to the  $\tilde{r}$ -dependent formulas for  $\tilde{x}$  away from zero.

For magnetic fields, the “real”  $\phi_n^r$  solutions will correspond to skew fields and the “imaginary”  $\phi_n^i$  solutions will correspond to normal fields

$$\mathbf{B} = -\frac{P_0}{qL} \sum_{n=0}^{\infty} \rho^n \left[ a_n \tilde{\nabla} \phi_n^r + b_n \tilde{\nabla} \phi_n^i \right] \quad (12.25)$$

where the gradient derivatives of  $\tilde{\nabla}$  are with respect to the normalized coordinates. In the limit of infinite bending radius  $\rho$ , the above equations converge to the straight line solution given in Eq. (12.10).

For electric fields, the “real” solutions will correspond to normal fields and the “imaginary” solutions are used for skew fields

$$\mathbf{E} = -\sum_{n=0}^{\infty} \rho^n \left[ a_{en} \tilde{\nabla} \phi_n^i + b_{en} \tilde{\nabla} \phi_n^r \right] \quad (12.26)$$

And this will converge to Eq. (12.17) in the straight line limit.

In the vertical plane, with  $\tilde{x} = 0$ , the solutions  $\phi_n^r$  and  $\phi_n^i$  have the same variation in  $\tilde{y}$  as the multipole fields with a straight geometry. For example, the field strength of an  $n = 1$  (quadrupole) multipole will be linear in  $\tilde{y}$  for  $\tilde{x} = 0$ . However, in the horizontal direction, with  $\tilde{y} = 0$ , the multipole field will vary like  $dF_2/d\tilde{x}$  which has terms of all orders in  $\tilde{x}$ . In light of this, the solutions  $\phi_n^r$  and  $\phi_n^i$  are called “vertically pure” solutions.

<sup>1</sup>Notice that here  $n$  is related to  $m$  in McMillian’s paper by  $m = n + 1$ . Also note that the  $\phi^r$  and  $\phi^i$  here have a normalization factor that is different from McMillian.

It is possible to construct “horizontally pure” solutions as well. That is, it is possible to construct solutions that in the horizontal plane, with  $\tilde{y} = 0$ , 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,  $\psi_n^r$  and  $\psi_n^i$ , as linear superpositions of the vertically pure solutions

$$\psi_n^r = \sum_{k=n}^{\infty} C_{nk} \phi_k^r, \quad \psi_n^i = \sum_{k=n}^{\infty} D_{nk} \phi_k^i \quad (12.27)$$

with the normalizations  $C_{nn} = D_{nn} = 1$ . The  $C_{nk}$  and  $D_{nk}$  are chosen, order by order, so that  $\psi_n^r$  and  $\psi_n^i$  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  $\tilde{x}^j$  term of  $(dF_i/d\tilde{x})/i$  when  $F_i$  is expressed as an expansion in  $\tilde{x}$  (Eq. (12.24)).  $C_{nk}$ ,  $k = 0, \dots, \infty$  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  $\tilde{x}$  for the functions  $F_i$ . To convert between field strength coefficients, Eqs. (12.25) and (12.26) and Eqs. (12.27) are combined

$$\begin{aligned} a_n &= \sum_{k=n}^{\infty} \frac{1}{\rho^{k-n}} C_{nk} \alpha_k, & a_{en} &= \sum_{k=n}^{\infty} \frac{1}{\rho^{k-n}} D_{nk} \alpha_{ek}, \\ b_n &= \sum_{k=n}^{\infty} \frac{1}{\rho^{k-n}} D_{nk} \beta_k, & b_{en} &= \sum_{k=n}^{\infty} \frac{1}{\rho^{k-n}} D_{nk} \beta_{ek} \end{aligned} \quad (12.28)$$

where  $\alpha_k$ ,  $\beta_k$ ,  $\alpha_{ek}$ , and  $\beta_{ek}$  are the corresponding coefficients for the horizontally pure solutions.

When expressed as a function of  $\tilde{r}$  and  $\tilde{y}$ , the vertically pure solutions  $\phi_n$  have a finite number of terms (Eqs. (12.20) and (12.21)). On the other hand, the horizontally pure solutions  $\psi_n$  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  $\phi_n^r$  and  $\phi_n^i$ . 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` (§3.4) 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.]

## 12.4 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 three parameterizations:

Cartesian Map	! §12.5.
Cylindrical Map	! §12.6
Generalized Gradient Map	! §12.7

These parameterizations are three of the four `field map` parameterizations that *Bmad* defines §??.

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 §12.5. for more details. The syntax for specifying the **Cartesian map** decomposition is discussed in §??.

The **cylindrical map** decomposition can be used for both DC and AC fields. See §12.6 for more details. The syntax for specifying the **cylindrical map** decomposition is discussed in §??.

The **generalized gradient map** start with the cylindrical map decomposition but then express the field using coefficients derived from an expansion of the scalar potential in powers of the radius (§12.7).

## 12.5 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 (§12.6).

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. §??.

The **Cartesian** decomposition is modeled using an extension of the method of Sagan, Crittenden, and Rubin[Sagan:wiggler]. 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:

$$\mathbf{B}(x, y, z) = \sum_i \mathbf{B}_i(x, y, z; A, k_x, k_y, k_z, x_0, y_0, \phi_z, family) \quad (12.29)$$

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:

$$\begin{aligned} B_x &= A \frac{k_x}{k_y} \cos(k_x(x + x_0)) \cosh(k_y(y + y_0)) \cos(k_z z + \phi_z) \\ B_y &= A \sin(k_x(x + x_0)) \sinh(k_y(y + y_0)) \cos(k_z z + \phi_z) \\ B_z &= -A \frac{k_z}{k_y} \sin(k_x(x + x_0)) \cosh(k_y(y + y_0)) \sin(k_z z + \phi_z) \end{aligned} \quad (12.30)$$

with  $k_y^2 = k_x^2 + k_z^2$



The **x family hyper-xy** form is:

$$\begin{aligned}
 B_x &= A \frac{k_x}{k_z} \cosh(k_x(x+x_0)) \cosh(k_y(y+y_0)) \cos(k_z z + \phi_z) \\
 B_y &= A \frac{k_y}{k_z} \sinh(k_x(x+x_0)) \sinh(k_y(y+y_0)) \cos(k_z z + \phi_z) \\
 B_s &= -A \sinh(k_x(x+x_0)) \cosh(k_y(y+y_0)) \sin(k_z z + \phi_z) \\
 &\quad \text{with } k_z^2 = k_x^2 + k_y^2
 \end{aligned} \tag{12.31}$$

And the **x family hyper-x** form is:

$$\begin{aligned}
 B_x &= A \cosh(k_x(x+x_0)) \cos(k_y(y+y_0)) \cos(k_z z + \phi_z) \\
 B_y &= -A \frac{k_y}{k_x} \sinh(k_x(x+x_0)) \sin(k_y(y+y_0)) \cos(k_z z + \phi_z) \\
 B_s &= -A \frac{k_z}{k_x} \sinh(k_x(x+x_0)) \cos(k_y(y+y_0)) \sin(k_z z + \phi_z) \\
 &\quad \text{with } k_z^2 = k_y^2 + k_x^2
 \end{aligned} \tag{12.32}$$

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  $B_{mad}$  by looking at the values of  $k_x$ ,  $k_y$ , and  $k_z$ .

Using a compact notation where  $\text{Ch} \equiv \cosh$ , 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	
$B_x$	$-A \frac{k_x}{k_y} \text{S}_x \text{Sh}_y \text{C}_z$	$A \frac{k_x}{k_z} \text{Sh}_x \text{Sh}_y \text{C}_z$	$A \text{Sh}_x \text{S}_y \text{C}_z$	
$B_y$	$A \text{C}_x \text{Ch}_y \text{C}_z$	$A \frac{k_y}{k_z} \text{Ch}_x \text{Ch}_y \text{C}_z$	$A \frac{k_y}{k_x} \text{Ch}_x \text{C}_y \text{C}_z$	(12.33)
$B_z$	$-A \frac{k_z}{k_y} \text{C}_x \text{Sh}_y \text{S}_z$	$-A \text{Ch}_x \text{Sh}_y \text{S}_z$	$-A \frac{k_z}{k_x} \text{Ch}_x \text{S}_y \text{S}_z$	
with	$k_y^2 = k_x^2 + k_z^2$	$k_z^2 = k_x^2 + k_y^2$	$k_x^2 = k_y^2 + k_z^2$	

the **qu family** of forms is:

Form	hyper-y	hyper-xy	hyper-x	
$B_x$	$A \frac{k_x}{k_y} \text{C}_x \text{Sh}_y \text{C}_z$	$A \frac{k_x}{k_z} \text{Ch}_x \text{Sh}_y \text{C}_z$	$A \text{Ch}_x \text{S}_y \text{C}_z$	
$B_y$	$A \text{S}_x \text{Ch}_y \text{C}_z$	$A \frac{k_y}{k_z} \text{Sh}_x \text{Ch}_y \text{C}_z$	$A \frac{k_y}{k_x} \text{Sh}_x \text{C}_y \text{C}_z$	(12.34)
$B_z$	$-A \frac{k_z}{k_y} \text{S}_x \text{Sh}_y \text{S}_z$	$-A \text{Sh}_x \text{Sh}_y \text{S}_z$	$-A \frac{k_z}{k_x} \text{Sh}_x \text{S}_y \text{S}_z$	
with	$k_y^2 = k_x^2 + k_z^2$	$k_z^2 = k_x^2 + k_y^2$	$k_x^2 = k_y^2 + k_z^2$	

the `sq` family of forms is:

Form	hyper-y	hyper-xy	hyper-x	
$B_x$	$-A \frac{k_x}{k_y} S_x \text{Ch}_y C_z$	$A \frac{k_x}{k_z} \text{Sh}_x \text{Ch}_y C_z$	$-A \text{Sh}_x C_y C_z$	
$B_y$	$A C_x \text{Sh}_y C_z$	$A \frac{k_y}{k_z} \text{Ch}_x \text{Sh}_y C_z$	$A \frac{k_y}{k_x} \text{Ch}_x S_y C_z$	(12.35)
$B_z$	$-A \frac{k_z}{k_y} C_x \text{Ch}_y S_z$	$-A \text{Ch}_x \text{Ch}_y S_z$	$A \frac{k_z}{k_x} \text{Ch}_x C_y S_z$	
with	$k_y^2 = k_x^2 + k_z^2$	$k_z^2 = k_x^2 + k_y^2$	$k_x^2 = k_y^2 + k_z^2$	

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.

Note: The vector potential for these fields is given in §??.

## 12.6 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 (§12.6).

In a lattice file, a `cylindrical` map is specified using the `cylindrical_map` attribute as explained in Sec. §??.

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. §12.6.1 while the RF decomposition is explained in Sec. §12.6.2.

### 12.6.1 DC Cylindrical Map Decomposition

The DC `cylindrical` parametrization used by *Bmad* essentially follows Venturini et al.[Venturini:LHC-Quads]. See Section §?? for details on the syntax used to cylindrical maps in *Bmad*. The electric and magnetic fields are both described by a scalar potential<sup>2</sup>

$$\mathbf{B} = -\nabla \psi_B, \quad \mathbf{E} = -\nabla \psi_E \quad (12.36)$$

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  $m$

$$\psi_B = \text{Re} \left[ \sum_{m=0}^{\infty} \psi_{Bm} \right] \quad (12.37)$$

---

<sup>2</sup>Notice the negative sign here and in Eq. (12.38) compared to Venturini et al.[Venturini:LHC-Quads]. This is to keep the definition of the electric scalar potential  $\psi_E$  consistent with the normal definition.

[Here and below, only equations for the magnetic field will be shown. The equations for the electric fields are similar.] The  $\psi_{Bm}$  are decomposed in  $z$  using a discrete Fourier sum.<sup>3</sup> Expressed in cylindrical coordinates the decomposition of  $\psi_{Bm}$  is

$$\psi_{Bm} = \sum_{n=-N/2}^{N/2-1} \psi_{Bmn} = \sum_{n=-N/2}^{N/2-1} \frac{-1}{k_n} e^{i k_n z} \cos(m\theta - \theta_{0m}) b_m(n) I_m(k_n \rho) \quad (12.38)$$

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. (12.38)  $k_n$  is given by

$$k_n = \frac{2\pi n}{N dz} \quad (12.39)$$

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. (12.38) 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 ( $z = 0, dz, 2 dz, \dots$ ) and has the virtue that the field is smoother in between.

The field associated with  $\psi_{Bm}$  is for  $m = 0$ :

$$\begin{aligned} B_\rho &= \text{Re} \left[ \sum_{n=-N/2}^{N/2-1} e^{i k_n z} b_0(n) I_1(k_n \rho) \right] \\ B_\theta &= 0 \\ B_z &= \text{Re} \left[ \sum_{n=-N/2}^{N/2-1} i e^{i k_n z} b_0(n) I_0(k_n \rho) \right] \end{aligned} \quad (12.40)$$

And for  $m \neq 0$ :

$$\begin{aligned} B_\rho &= \text{Re} \left[ \sum_{n=-N/2}^{N/2-1} \frac{1}{2} e^{i k_n z} \cos(m\theta - \theta_{0m}) b_m(n) \left[ I_{m-1}(k_n \rho) + I_{m+1}(k_n \rho) \right] \right] \\ B_\theta &= \text{Re} \left[ \sum_{n=-N/2}^{N/2-1} \frac{-1}{2} e^{i k_n z} \sin(m\theta - \theta_{0m}) b_m(n) \left[ I_{m-1}(k_n \rho) - I_{m+1}(k_n \rho) \right] \right] \\ B_z &= \text{Re} \left[ \sum_{n=-N/2}^{N/2-1} i e^{i k_n z} \cos(m\theta - \theta_{0m}) b_m(n) I_m(k_n \rho) \right] \end{aligned} \quad (12.41)$$

While technically  $\psi_{Bm0}$  is not well defined due to the  $1/k_n$  factor that is present, the field itself is well behaved. Mathematically, Eq. (12.38) can be corrected if, for  $n = 0$ , the term  $I_m(k_n \rho)/k_n$  is replaced by

$$\frac{I_m(k_0 \rho)}{k_0} \rightarrow \begin{cases} \rho & \text{if } m = 0 \\ \rho/2 & \text{if } m = 1 \\ 0 & \text{otherwise} \end{cases} \quad (12.42)$$

---

<sup>3</sup>Venturini uses a continuous Fourier transformation but *Bmad* uses a discrete transformation so that only a finite number of coefficients are needed.

The magnetic vector potential for  $m = 0$  is constructed such that only  $A_\theta$  is non-zero

$$\begin{aligned} A_\rho &= 0 \\ A_\theta &= \text{Re} \left[ \sum_{n=-N/2}^{N/2-1} \frac{i}{k_n} e^{i k_n z} b_0(n) I_1(k_n \rho) \right] \\ A_z &= 0 \end{aligned} \quad (12.43)$$

For  $m \neq 0$ , the vector potential is chosen so that  $A_\theta$  is zero.

$$\begin{aligned} A_\rho &= \text{Re} \left[ \sum_{n=-N/2}^{N/2-1} \frac{-i \rho}{2m} e^{i k_n z} \cos(m\theta - \theta_{0m}) b_m(n) \left[ I_{m-1}(k_n \rho) - I_{m+1}(k_n \rho) \right] \right] \\ A_\theta &= 0 \\ A_z &= \text{Re} \left[ \sum_{n=-N/2}^{N/2-1} \frac{-i \rho}{m} e^{i k_n z} \cos(m\theta - \theta_{0m}) b_m(n) I_m(k_n \rho) \right] \end{aligned} \quad (12.44)$$

Note: The description of the field using ‘‘generalized gradients’’[**Newton:map**] is similar to the above equations. The difference is that, with the generalized gradient formalism, terms in  $\theta$  and  $\rho$  are expanded in a Taylor series in  $x$  and  $y$ .

## 12.6.2 AC Cylindrical Map Decomposition

For RF fields, the `cylindrical` mode parametrization used by *Bmad* essentially follows Abell[**Abell:RF-maps**]. The electric field is the real part of the complex field

$$\mathbf{E}(\mathbf{r}) = \sum_{j=1}^M \mathbf{E}_j(\mathbf{r}) \exp[-2\pi i (f_j t + \phi_{0j})] \quad (12.45)$$

where  $M$  is the number of modes. Each mode satisfies the vector Helmholtz equation

$$\nabla^2 \mathbf{E}_j + k_{tj}^2 \mathbf{E}_j = 0 \quad (12.46)$$

where  $k_{tj} = 2\pi f_j/c$  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

$$\begin{aligned} E_\rho(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} -e^{i k_n z} i k_n e_0(n) \tilde{I}_1(\kappa_n, \rho) \\ E_\theta(\mathbf{r}) &= 0 \\ E_z(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} e^{i k_n z} e_0(n) \tilde{I}_0(\kappa_n, \rho) \end{aligned} \quad (12.47)$$

where  $\tilde{I}_m$  is

$$\tilde{I}_m(\kappa_n, \rho) \equiv \frac{I_m(\kappa_n \rho)}{\kappa_n^m} \quad (12.48)$$

with  $I_m$  being a modified Bessel function first kind, and  $\kappa_n$  is given by

$$\kappa_n = \sqrt{k_n^2 - k_t^2} = \begin{cases} \sqrt{k_n^2 - k_t^2} & |k_n| > k_t \\ -i \sqrt{k_t^2 - k_n^2} & k_t > |k_n| \end{cases} \quad (12.49)$$

with

$$k_n = \frac{2\pi n}{N dz} \quad (12.50)$$

$N$  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  $\kappa_n$  is imaginary,  $I_m(\kappa_n \rho)$  can be evaluated through the relation

$$I_m(-ix) = i^{-m} J_m(x) \quad (12.51)$$

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

$$e_0(n) = \frac{1}{\tilde{I}_0(\kappa_n, R)} \frac{1}{N} \sum_{p=0}^{N-1} e^{-2\pi i n p/N} E_z(R, p dz) \quad (12.52)$$

The non-accelerating  $m=0$  mode has an electric field in the form

$$\begin{aligned} E_\rho(\mathbf{r}) &= E_z(\mathbf{r}) = 0 \\ E_\theta(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} e^{i k_n z} b_0(n) \tilde{I}_1(\kappa_n, \rho) \end{aligned} \quad (12.53)$$

where the  $b_0$  coefficients can be obtained given knowledge of the field at some radius  $R$  via

$$b_0(n) = \frac{1}{\tilde{I}_1(\kappa_n, R)} \frac{1}{N} \sum_{p=0}^{N-1} e^{-2\pi i n p/N} E_\theta(R, p dz) \quad (12.54)$$

For positive  $m$ , the electric field is in the form

$$\begin{aligned} E_\rho(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} -i e^{i k_n z} \left[ k_n e_m(n) \tilde{I}_{m+1}(\kappa_n, \rho) + b_m(n) \frac{\tilde{I}_m(\kappa_n, \rho)}{\rho} \right] \cos(m\theta - \theta_{0m}) \\ E_\theta(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} -i e^{i k_n z} \left[ k_n e_m(n) \tilde{I}_{m+1}(\kappa_n, \rho) + \right. \\ &\quad \left. b_m(n) \left( \frac{\tilde{I}_m(\kappa_n, \rho)}{\rho} - \frac{1}{m} \tilde{I}_{m-1}(\kappa_n, \rho) \right) \right] \sin(m\theta - \theta_{0m}) \\ E_z(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} e^{i k_n z} e_m(n) \tilde{I}_m(\kappa_n, \rho) \cos(m\theta - \theta_{0m}) \end{aligned} \quad (12.55)$$

The  $\mathbf{e}_m$  and  $\mathbf{b}_m$  coefficients can be obtained given knowledge of the field at some radius  $R$  via

$$\begin{aligned} e_m(n) &= \frac{1}{\tilde{I}_m(\kappa_n, R)} \frac{1}{N} \sum_{p=0}^{N-1} e^{-2\pi i n p/N} E_{zc}(R, p dz) \\ b_m(n) &= \frac{R}{\tilde{I}_m(\kappa_n, R)} \left[ \frac{1}{N} \sum_{p=0}^{N-1} i e^{-2\pi i n p/N} E_{\rho c}(R, p dz) - k_n e_m(n) \tilde{I}_{m+1}(\kappa_n, R) \right] \end{aligned} \quad (12.56)$$

where  $E_{\rho c}$ ,  $E_{\theta s}$ , and  $E_{zc}$  are defined by

$$\begin{aligned} E_{\rho}(R, \theta, z) &= E_{\rho c}(R, z) \cos(m\theta - \theta_{0m}) \\ E_{\theta}(R, \theta, z) &= E_{\theta s}(R, z) \sin(m\theta - \theta_{0m}) \\ E_z(R, \theta, z) &= E_{zc}(R, z) \cos(m\theta - \theta_{0m}) \end{aligned} \quad (12.57)$$

The above mode decomposition was done in the gauge where the scalar potential  $\psi$  is zero. The electric and magnetic fields are thus related to the vector potential  $\mathbf{A}$  via

$$\mathbf{E} = -\partial_t \mathbf{A}, \quad \mathbf{B} = \nabla \times \mathbf{A} \quad (12.58)$$

Using Eq. (12.45), the vector potential can be obtained from the electric field via

$$\mathbf{A}_j = \frac{-i \mathbf{E}_j}{2\pi f_j} \quad (12.59)$$

Symplectic tracking through the RF field is discussed in Section §???. For the fundamental accelerating mode, the vector potential can be analytically integrated using the identity

$$\int dx \frac{x I_1(a \sqrt{x^2 + y^2})}{\sqrt{x^2 + y^2}} = \frac{1}{a} I_0(a \sqrt{x^2 + y^2}) \quad (12.60)$$

## 12.7 Generalized Gradient Map Field Modeling

*Bmad* has a number of `field map` models that can be used to model electric or magnetic fields (§??). One model involves what are called `generalized gradients`[Venturini:magmaps]. This model is restricted to modeling DC magnetic or electric fields. In a lattice file, the generalized gradient field model is specified using the `gen_grad_map` attribute as explained in Sec. §??.

The electric and magnetic fields are both described by a scalar potential<sup>4</sup>

$$\mathbf{B} = -\nabla \psi_B, \quad \mathbf{E} = -\nabla \psi_E \quad (12.61)$$

The scalar potential is then decomposed into azimuthal components

$$\psi = \sum_{m=1}^{\infty} \psi_{m,s} \sin(m\theta) + \sum_{m=0}^{\infty} \psi_{m,c} \cos(m\theta) \quad (12.62)$$

where the  $\psi_{m,\alpha}$  ( $\alpha = c, s$ ) are characterized by a using functions  $C_{m,\alpha}(z)$  which are functions along the longitudinal  $z$ -axis.

$$\psi_{m,\alpha} = \sum_{n=0}^{\infty} \frac{(-1)^{n+1} m!}{4^n n! (n+m)!} \rho^{2n+m} C_{m,\alpha}^{[2n]}(z) \quad (12.63)$$

The notation  $[2n]$  indicates the  $2n^{th}$  derivative of  $C_{m,\alpha}(z)$ .

---

<sup>4</sup>Notice the negative sign here and in Eq. (12.63) compared to Venturini et al.[Venturini:magmaps]. This is to keep the definition of the electric scalar potential  $\psi_E$  consistent with the normal definition.

From Eq. (12.63) the field is given by

$$\begin{aligned}
B_\rho &= \sum_{m=1}^{\infty} \sum_{n=0}^{\infty} \frac{(-1)^n m! (2n+m)}{4^n n! (n+m)!} \rho^{2n+m-1} \left[ C_{m,s}^{[2n]}(z) \sin m\theta + C_{m,c}^{[2n]}(z) \cos m\theta \right] + \\
&\quad \sum_{n=1}^{\infty} \frac{(-1)^n 2n}{4^n n! n!} \rho^{2n-1} C_{0,c}^{[2n]}(z) \\
B_\theta &= \sum_{m=1}^{\infty} \sum_{n=0}^{\infty} \frac{(-1)^n m! m}{4^n n! (n+m)!} \rho^{2n+m-1} \left[ C_{m,s}^{[2n]}(z) \cos m\theta - C_{m,c}^{[2n]}(z) \sin m\theta \right] \\
B_z &= \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(-1)^n m!}{4^n n! (n+m)!} \rho^{2n+m} \left[ C_{m,s}^{[2n+1]}(z) \sin m\theta + C_{m,c}^{[2n+1]}(z) \cos m\theta \right]
\end{aligned} \tag{12.64}$$

Even though the scalar potential only involves even derivatives of  $C_{m,\alpha}$ , the field is dependent upon the odd derivatives as well. The multipole index  $m$  is such that  $m = 0$  is for solenoidal fields,  $m = 1$  is for dipole fields,  $m = 2$  is for quadrupolar fields, etc. The **sin**-like generalized gradients represent normal (non-skew) fields and the **cos**-like one represent skew fields. The on-axis fields at  $x = y = 0$  are given by:

$$(B_x, B_y, B_z) = (C_{1,c}, C_{1,s}, -C_{0,c}^{[1]}) \tag{12.65}$$

The magnetic vector potential for  $m = 0$  is constructed such that only  $A_\theta$  is non-zero

$$\begin{aligned}
A_\rho &= 0 \\
A_\theta &= \sum_{n=1}^{\infty} \frac{(-1)^{n+1} 2n}{4^n n! n!} \rho^{2n-1} C_{0,c}^{[2n-1]} \\
A_z &= 0
\end{aligned} \tag{12.66}$$

For  $m \neq 0$ , the vector potential is chosen so that  $A_\theta$  is zero.

$$\begin{aligned}
A_\rho &= \sum_{m=1}^{\infty} \sum_{n=0}^{\infty} \frac{(-1)^n (m-1)!}{4^n n! (n+m)!} \rho^{2n+m+1} \left[ C_{m,s}^{[2n+1]} \cos(m\theta) - C_{m,c}^{[2n+1]} \sin(m\theta) \right] \\
A_\theta &= 0 \\
A_z &= \sum_{m=1}^{\infty} \sum_{n=0}^{\infty} \frac{(-1)^n (m-1)! (2n+m)}{4^n n! (n+m)!} \rho^{2n+m} \left[ -C_{m,s}^{[2n]} \cos(m\theta) + C_{m,c}^{[2n]} \sin(m\theta) \right]
\end{aligned} \tag{12.67}$$

The functions  $C_{m,\alpha}(z)$  are characterized by specifying  $C_{m,\alpha}(z_i)$  and derivatives at equally spaced points  $z_i$ , up to some maximum derivative order  $N_{m,\alpha}$  chosen by the user. Interpolation is done by constructing an interpolating polynomial (“non-smoothing spline”) for each GG of order  $2N_{m,\alpha} + 1$  for each interval  $[z_i, z_{i+1}]$  which has the correct derivatives from 0 to  $N_{m,\alpha}$  at points  $z_i$  and  $z_{i+1}$ . The coefficients of the interpolating polynomial are easily calculated by inverting the appropriate matrix equation.

The advantages of a generalized gradient map over a cylindrical or Cartesian map decomposition come from the fact that with generalized gradients the field at some point  $(x, y, z)$  is only dependent upon the value of  $C_{m,\alpha}(z)$  and derivatives at points  $z_i$  and  $z_{i+1}$  where  $z$  is in the interval  $[z_i, z_{i+1}]$ . This is in contrast to the cylindrical or Cartesian map decomposition where the field at any point is dependent upon *all* of the terms that characterize the field. This “**locality**” property of generalized gradients means that calculating coefficients is easier (the calculation of  $C_{m,\alpha}(z)$  at  $z_i$  can be done using only the field near  $z_i$  independent of other regions) and it is easier to ensure that the field goes to zero at the longitudinal ends of the element. Additionally, the evaluation is faster since only coefficients to

either side of the evaluation point contribute. The disadvantage of generalized gradients is that since the derivatives are truncated at some order  $N_{m,\alpha}$ , the resulting field does not satisfy Maxwell's equations with the error as a function of radius scaling with the power  $\rho^{m+N_{m,\alpha}}$ .

It is sometimes convenient to express the fields in terms of Cartesian coordinates. For sine like even derivatives  $C_{m,s}^{[2n]}$  the conversion is

$$\begin{aligned} (B_x, B_y) &= (\cos \theta B_\rho - \sin \theta B_\theta, \sin \theta B_\rho + \cos \theta B_\theta) \\ &= \frac{(-1)^n m!}{4^n n! (n+m)!} C_{m,s}^{[2n]} \left[ (n+m) (x^2 + y^2)^n (S_{xy}(m-1), C_{xy}(m-1)) + \right. \\ &\quad \left. n (x^2 + y^2)^{n-1} (S_{xy}(m+1), -C_{xy}(m+1)) \right] \end{aligned} \quad (12.68)$$

and for the sine like odd derivatives  $C_{m,s}^{[2n+1]}$

$$B_z = \frac{(-1)^n m!}{4^n n! (n+m)!} (x^2 + y^2)^n C_{m,s}^{[2n+1]}(z) S_{xy}(m) \quad (12.69)$$

where the last term in Eq. (12.68) is only present for  $n > 0$ .

$$\begin{aligned} S_{xy}(m) &\equiv \rho^m \sin m\theta = \sum_{r=0}^{2r \leq m-1} (-1)^r \binom{m}{2r+1} x^{m-2r-1} y^{2r+1} \\ C_{xy}(m) &\equiv \rho^m \cos m\theta = \sum_{r=0}^{2r \leq m} (-1)^r \binom{m}{2r} x^{m-2r} y^{2r} \end{aligned} \quad (12.70)$$

The conversion for the cosine like derivatives is:

$$\begin{aligned} (B_x, B_y) &= \frac{(-1)^n m!}{4^n n! (n+m)!} C_{m,c}^{[2n]} \left[ (n+m) (x^2 + y^2)^n (C_{xy}(m-1), -S_{xy}(m-1)) + \right. \\ &\quad \left. n (x^2 + y^2)^{n-1} (C_{xy}(m+1), S_{xy}(m+1)) \right] \\ B_z &= \frac{(-1)^n m!}{4^n n! (n+m)!} (x^2 + y^2)^n C_{m,c}^{[2n+1]}(z) C_{xy}(m) \end{aligned} \quad (12.71)$$

## 12.8 RF fields

The following describes the how RF fields are calculated when the `field_calc` attribute of an RF element is set to `bmad_standard`.<sup>5</sup> Also see Section §?? for how fringe fields are calculated.

With `cavity_type` set to `traveling_wave`, the setting of `longitudinal_mode` is ignored and the field is given by

$$\begin{aligned} E_s(r, \phi, s, t) &= G \cos(\omega t - k s + 2 \pi \phi) \\ E_r(r, \phi, s, t) &= -\frac{r}{2} G k \sin(\omega t - k s + 2 \pi \phi) \\ B_\phi(r, \phi, s, t) &= -\frac{r}{2c} G k \sin(\omega t - k s + 2 \pi \phi) \end{aligned} \quad (12.72)$$

<sup>5</sup>Notice that the equations here are only relevant with the `tracking_method` for an RF element set to a method like `runge_kutta` where tracking through the field of an element is done. For `bmad_standard` tracking, Equations for `lcavity` tracking are shown in §?? and `rfcavity` tracking in §??.



where  $G$  is the accelerating gradient,  $k = \omega/c$  is the wave number with  $\omega$  being the RF frequency.

For standing wave cavities, with `cavity_type` set to `standing_wave`, 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 (set by the `longitudinal_mode` element parameter). The  $p = 1$  longitudinal mode models the fields as a pillbox with the transverse wall at infinity as detailed in Chapter 3, Section VI of reference [Lee:Physics]

$$\begin{aligned} E_s(r, \phi, s, t) &= 2G \cos(k s) \cos(\omega t + 2\pi \phi) \\ E_r(r, \phi, s, t) &= r G k \sin(k s) \cos(\omega t + 2\pi \phi) \\ B_\phi(r, \phi, s, t) &= -\frac{r}{c} G k \cos(k s) \sin(\omega t + 2\pi \phi) \end{aligned} \tag{12.73}$$

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 (which is the default), a “pseudo TM<sub>010</sub>” mode is used that has the correct symmetry:

$$\begin{aligned} E_s(r, \phi, s, t) &= 2G \sin(k s) \sin(\omega t + 2\pi \phi) \\ E_r(r, \phi, s, t) &= -r G k \cos(k s) \sin(\omega t + 2\pi \phi) \\ B_\phi(r, \phi, s, t) &= \frac{r}{c} G k \sin(k s) \cos(\omega t + 2\pi \phi) \end{aligned} \tag{12.74}$$



# Part III

## Developer's Guide



## Chapter 13

# Defining New Lattice Elements

### 13.1 Defining new Element Parameters

\* Bookkeeping

### 13.2 Defining a New Element

To construct a new element type:

\* Define a new element type. Example:

```
construct_ele_type NewEleType
```

\* Extend EleGeometry Holy trait group (§4.2) if a new geometry is needed. Example:

```
abstract type CORKSCREW <: EleGeometry end
```

\* If the geometry is not STRAIGHT, Extend the `ele_geometry()` function to return the correct geometry for the new type of element. Example:

```
ele_geometry(ele::NewEleType) = CORKSCREW
```

\* If the element has a new type of geometry, extend the `propagate_ele_geometry()` function to handle the new type of geometry. Example:

```
function propagate_ele_geometry(::TypeCORKSCREW, fstart::FloorPositionGroup, ele::Ele)
    ...
    return floor_end # FloorPositionGroup at the downstream end of the element.
end
```



## Chapter 14

# Lattice Bookkeeping

Bookkeeping in *AcceleratorLattice.jl* mainly involves making sure that dependent parameters are updated as needed. This includes dependent parameters within a lattice element, propagating changes through the lattice, and lord/slave bookkeeping.

### 14.1 Lord/Slave Bookkeeping

There are two types of lords/slave groupings:

Superposition: Super lords / Super Slaves §8

Multipass: Multipass lords Multipass Slaves §7

The lord and slave status of a lattice element is contained in the `LordSlaveStatusGroup` parameter group. The components of this group are (§5.15):

`lord_status::Lord.T` - Lord status.  
`slave_status::Slave.T` - Slave status.

For a given element, some combinations of lord and slave status are not possible. The possibilities are:

slave_status	lord_status		
	.NOT	.SUPER	.MULTIPASS
.NOT	X	X	X
.SUPER	X		
.MULTIPASS	X	X	

Notice that the only possibility for an element to simultaneously be both a lord and a slave is for a super lord being a multipass slave.

### 14.2 Girders

`Girders` support a set of supported elements. A `Girder` may support other `Girders` and so a hierarchy

of **Girders** may be constructed. While a **Girder** may support many elements, any given element may only be supported by one **Girder**.

**Girder** elements may support super and multipass lord elements, a **Girder** will never support slave elements directly. This includes any super lord element that is also a multipass slave.

A **Girder** element will have a `Vector{Ele}` parameter of supported elements `.supported`. Supported elements will have a `.girder` parameter pointing to the supporting **Girder**. Elements that do not have a supporting **Girder** will not have this parameter.

### 14.3 Superposition

Super lords are formed when elements are superimposed on top of other elements (§8). The *AcceleratorLattice.jl* bookkeeping routines and take changes to lord element parameters and set the appropriate slave parameters.

When there is a set of lattice elements that are in reality the same physical element, a multipass lord can be used to represent the common physical element §7. The *AcceleratorLattice.jl* bookkeeping routines and take changes to lord element parameters and set the appropriate slave parameters.

**Girder** lords support other elements (possibly including other **Girder** lords). Alignment shifts of a **Girder** lord will shift the supported elements accordingly.

### 14.4 Lord/Slave Element Pointers

All three types of lord elements contain a `Vector{ele}` of elements called **slaves**.

### 14.5 Element Parameter Access

### 14.6 Changed Parameters and Auto-Bookkeeping

Importance of using `pop!`, `insert!`, `push!` and `set!` when modifying the `branch.ele` array.

The `ele.changed` parameter (which is actually `ele.pdict[:changed]`) is a dictionary. The keys of this dict will be either symbols of the changed parameters or will be an element parameter group. When the key is a symbol of a changed parameter, the dict value will be the old value of the parameter. These dict entries are set by the overloaded `Base.setProperty(ele, param_sym, value)` function. When the key is an element parameter group, the dict value will be the string `"changed"`. These dict entries are set by functions that do lord/slave bookkeeping.

When bookkeeping is done, entries from the `ele.changed` dict are removed when the corresponding parameter(s) are bookkept. If there are dict entries that remain after all bookkeeping is done, this is an indication of a problem and a warning message is printed.



# Chapter 15

## Design Decisions

This chapter discusses some of the design decisions that were made in the planning of *AcceleratorLattice.jl*. Hopefully this information will be useful as *AcceleratorLattice.jl* is developed in the future. The design of *AcceleratorLattice.jl* is heavily influenced by the decades of experience constructing and maintaining *Bmad*— both in terms of what works and what has not worked.

First a clarification. The name *Bmad* can be used in two senses. There is *Bmad* the Fortran software toolkit that can be used to create simulation programs. But *Bmad* can also be used to refer to the ecosystem of toolkit and *Bmad* based programs that have been developed over the years — the most heavily used program being Tao. In the discussion below, *Bmad* generally refers to the toolkit since it is the toolkit that defines the syntax for *Bmad* lattice files.

**Bmad history:** To understand *Bmad* it helps to understand some of the history of *Bmad*. The *Bmad* toolkit started out as a modest project for calculating Twiss parameters and closed orbits within online control programs for the Cornell CESR storage ring. As such, the lattice structure was simply an array of elements. That is, early *Bmad* did not have the concept of interlocking branches, and tracking was very simple — there was only one tracking method, symplecticity was ignored and ultra-relativistic and paraxial approximations were used. *Bmad* has come a long way from the early days but design decisions made early on still haunt the *Bmad* toolkit.

**Julia itself is the design language:** One of the main problems with *Bmad*— and many other simulation programs like MAD, Elegant, SAD, etc. — is that the design language is some custom construct with custom syntax put together by a team that never has enough manpower. This both greatly limits the versatility of language as well as adding the burden of developing and maintaining the language. Julia was chosen for *AcceleratorLattice.jl* due to the ability of using Julia as the design language.

There are many design decisions that flow from the fact that Julia is used for the design language so decisions are made to follow the “Julia way”. For example, case sensitivity of names, indexing of branch element arrays starting at 1 (*Bmad* uses 0), etc.

**Separation of tracking and lattice description:** One of the first *AcceleratorLattice.jl* design decisions was to separate particle tracking from the lattice description. This was done since experience with *Bmad* showed that properly doing lattice bookkeeping is vastly more complicated when tracking is

involved. This is especially true when the User can choose among multiple tracking methods for a given element and the User is free to vary the tracking method on-the-fly.

The decision to separate lattice and tracking was also inspired by the PTC code of Etienne Forest. The fact that *Bmad* did not make this separation complicated *Bmad*'s lattice element structure, the `ele_struct`, to the extent that the `ele_struct` is the most complicated structure in all of *Bmad*. And having complicated structures is an impediment to code sustainability. The lack of a separation in *Bmad* also made bookkeeping more complicated in cases where, for example, Twiss parameters were to be calculated under differing conditions (EG varying initial particle positions) but the `ele_struct` can only hold Twiss parameters for one specific condition.

**Lattice branches:** The organization of the lattice into branches with each branch having an array of elements has worked very well with *Bmad* and so is used with *AcceleratorLattice.jl*. The relatively minor difference is that with *AcceleratorLattice.jl* the organization of the branches is more logical with multiple lord branches with each lord branch containing only one type of lord.

**No Controllers:** *Bmad* has control element types called `groups` and `overlays`. Elements of these types can control the parameters of other elements. The ability to define controllers has been tremendously useful, for example, to simulate machine control from a control room. Nevertheless, controllers are not implemented with *AcceleratorLattice.jl*. The reason for this is that there is no need to define controllers since the Julia language provides all the necessary tools to construct control functions that have a versatility much greater than the ones in *Bmad*.

**Type stability:** Type stability is *not* a major concern with *AcceleratorLattice.jl*. The reason being that compared to the time needed for tracking and track analysis, lattice instantiation and manipulation does not take an appreciable amount of time. For tracking, where computation time is a huge consideration, an interface layer can be used to translate lattice parameters to a type stable form. Of much greater importance is the flexibility of *AcceleratorLattice.jl* to accomodate changing needs and software sustainability. Hence all element, branch, and lattice structures contain a Dict (always called `pdict`) which can store arbitrary information.

**Lattice element structure:** All lattice element structs are very simple: They contain a single Dict and all element information is stored within this Dict. This means that there is no restriction as to what can be stored in an element adding custom information to an element simple. And the ability to do customization easily is very important.

Within an element Dict, for the most part, parameters are grouped into “element group” structs. A flattened structure, that is, without the element group structs, would be the correct strategy if the number of possible parameters for a given element type was not as large as it is. However, the parameterization of an element can be complicated. For example, a field table describing the field in an element has a grid of field points plus parameters to specify the distance between points, the frequency (if the field is oscillating), etc. In such a case, where the number of parameters is large, and with the parameters falling into logical groups, using substructures if preferred. Another consideration is that parameter groups help remove the conflict that occurs when multiple parameters logically should have the same name. For example, if an element is made up of different parts and multipole parts can have independent misalignments, parameter groups help keep the offset parameters distinct.

**Defining multipoles using normal and skew strengths along with a tilt:** The reason why for any order multipole there are three components, normal, skew and tilt, that describe the field when only

two would be sufficient is due convenience. Having normal and skew components is convenient when magnet has multiple windings that control both independently. A common case is combined horizontal and vertical steering magnets. On the other hand, being able to “misalign” the multipole using the `tilt` component is also useful.



## Part IV

# Bibliography

