Date: 2024-10-3
Version: 0.5.5

The Accelerator Lattice.jl Reference Manual

David Sagan

Contents

List of Figures

List of Tables

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 documentationon. 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

SciBmad has it's origins in the "*Bmad* "[5] ecosystem of toolkits and programs developed over several decades at Cornell University. Although the two share a similar name, **the code of the two are completely separate**. That being said, the development of *AcceleratorLattice.jl* is heavily influenced by the experience — both good and bad — in the development and use of *Bmad* as well as experience with other accelerator simulation programs.

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[1, 4]. This is in marked contrast to many accelerator simulation programs such programs as MAD[3], Elegant[2], 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`. All 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[]`

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.

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` (§**??**).

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 `Lat`.

Essentially, a `lattice`, has an array 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` and `photon_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.

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 (§11.1.3). These parameters include the element's global 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 = Lat("simple_lat", [a_line])
```

1.12 Differences From Bmad

There are many differences between *AcceleratorLattice.jl* and *Bmad*. Many of 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 `Lat.branch[]` and `branch.ele[]` is from one conforming to the Julia standard.

The *Bmad* names for the coordinate systems (§11) was somewhat different and not always consistent. The `global` 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 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 though 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 `suptypes` function:

```
julia> subtypes(EleParameterGroup)
28-element VectorAny:
 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 -> Global 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.`
- `StringGroup -> Informational strings.`
- `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(EleParamterGroup)` 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(::Lat)` 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` = `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(::Lat)` 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(::Lat)` 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 be beginning element's reference energy (or equivilantly the referece momentum), or changes to the `voltage` of an `LCavity` element will result in the reference energy of all downstream elements changing.

Global position dependency: The position of a lattice element in the global coordinate system (§11.2) is affected by a) the lengths of all upstream elements, b) the bend angles of all upstream elements, and c) the position in global 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:

Element	*Section*	*Element*	*Section*
`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:

```
AlignmentGroup     -> Element position/orientation shift. §5.1
ApertureGroup      -> Vacuum chamber aperture. §5.2
BMultipoleGroup    -> Magnetic multipoles. §5.3
FloorPositionGroup -> Global floor position and orientation. §5.7
LengthGroup        -> Length and s-position parameters. §5.11
LordSlaveGroup     -> Element lord and slave status. §5.12
MasterGroup        -> Contains field_master parameter. §5.13
ReferenceGroup     -> Reference energy and species. §5.18
StringGroup        -> Informational strings. §5.20
TrackingGroup      -> Default tracking settings. §5.21
```

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

$$\omega \ll \frac{c}{r} \tag{3.1}$$

where ω 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 -> Global floor position and orientation. §5.7
LengthGroup        -> Length and s-position parameters. §5.11
LordSlaveGroup     -> Element lord and slave status. §5.12
ReferenceGroup     -> Reference energy and species. §5.18
StringGroup        -> Informational strings. §5.20
TrackingGroup      -> Default tracking settings. §5.21
```

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:

```
FloorPositionGroup -> Global floor position and orientation. §5.7
InitParticleGroup  -> Initial particle position and spin. §5.9
LengthGroup        -> Length and s-position parameters. §5.11
LordSlaveGroup     -> Element lord and slave status. §5.12
ReferenceGroup     -> Reference energy and species. §5.18
StringGroup        -> Informational strings. §5.20
TrackingGroup      -> Default tracking settings. §5.21
TwissGroup         -> Initial Twiss and coupling parameters. §5.22
```

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 §11.1. 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. §5.1
ApertureGroup       -> Vacuum chamber aperture. §5.2
BMultipoleGroup     -> Magnetic multipoles. §5.3
BendGroup           -> Bend element parameters. §5.5
EMultipoleGroup     -> Electric multipoles. §5.6
FloorPositionGroup -> Global floor position and orientation. §5.7
LengthGroup         -> Length and s-position parameters. §5.11
LordSlaveGroup      -> Element lord and slave status. §5.12
MasterGroup         -> Contains field_master parameter. §5.13
ReferenceGroup      -> Reference energy and species. §5.18
StringGroup         -> Informational strings. §5.20
TrackingGroup       -> Default tracking settings. §5.21
```

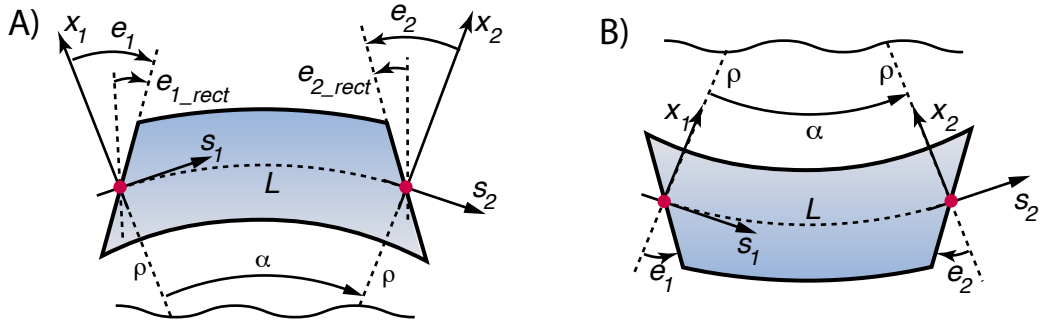

Figure 3.1: Bend geometry. Red dots are the entry and exit points that define the origin for the coordinate systems at the entry end (s_1, x_1) and exit ends (s_2, x_2) respectively. In the figure, the angle `alpha` is denoted α and the radius `rho` is denoted ρ. A) Bend geometry with positive bend angle. For the geometry shown, `g`, `angle`, `rho`, `e1`, `e2`, `e1_rect`, and `e2_rect` are all positive. B) Bend geometry with negative bend angle. For the geometry shown, `g`, `angle`, `rho`, `e1`, `e2`, `e1_rect`, and `e2_rect` are all negative. Note: The figures are drawn for zero `ref_tilt` where the rotation axis is parallel to the y-axis.

The `BendGroup` group (§5.5) 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 -> Global floor position and orientation. §5.7
LengthGroup        -> Length and s-position parameters. §5.11
LordSlaveGroup     -> Element lord and slave status. §5.12
ReferenceGroup     -> Reference energy and species. §5.18
StringGroup        -> Informational strings. §5.20
TrackingGroup      -> Default tracking settings. §5.21
```

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 -> Global floor position and orientation. §5.7
LengthGroup        -> Length and s-position parameters. §5.11
LordSlaveGroup     -> Element lord and slave status. §5.12
ReferenceGroup     -> Reference energy and species. §5.18
StringGroup        -> Informational strings. §5.20
TrackingGroup      -> Default tracking settings. §5.21
```

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 -> Global floor position and orientation. §5.7
LengthGroup        -> Length and s-position parameters. §5.11
LordSlaveGroup     -> Element lord and slave status. §5.12
ReferenceGroup     -> Reference energy and species. §5.18
```

```
StringGroup        -> Informational strings. §5.20
TrackingGroup      -> Default tracking settings. §5.21
```

3.8 Drift

A `Drift` is a field free element.

Element parameter groups associated with this element type are:

```
FloorPositionGroup -> Global floor position and orientation. §5.7
LengthGroup        -> Length and s-position parameters. §5.11
LordSlaveGroup     -> Element lord and slave status. §5.12
ReferenceGroup     -> Reference energy and species. §5.18
StringGroup        -> Informational strings. §5.20
TrackingGroup      -> Default tracking settings. §5.21
```

3.9 EGun

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

NOTE: This Element is in development and is incomplete.

Element parameter groups associated with this element type are:

```
FloorPositionGroup -> Global floor position and orientation. §5.7
LengthGroup        -> Length and s-position parameters. §5.11
LordSlaveGroup     -> Element lord and slave status. §5.12
ReferenceGroup     -> Reference energy and species. §5.18
StringGroup        -> Informational strings. §5.20
TrackingGroup      -> Default tracking settings. §5.21
```

3.10 Fiducial

A `Fiducial` element is used to fix the position and orientation of the reference orbit within the global coordinate system at the location of the `Fiducial` element. A `Fiducial` element will affect the global floor coordinates (§11.2) 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:

```
FloorPositionGroup -> Global floor position and orientation. §5.7
LengthGroup        -> Length and s-position parameters. §5.11
LordSlaveGroup     -> Element lord and slave status. §5.12
ReferenceGroup     -> Reference energy and species. §5.18
StringGroup        -> Informational strings. §5.20
TrackingGroup      -> Default tracking settings. §5.21
```

3.11 FloorShift

A `FloorShift` element shifts the reference orbit in the global 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 the start of an alternative `branch` for the beam (or X-rays or other particles generated by the beam) to follow.

NOTE: This Element is in development and is incomplete.

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.

NOTE: This Element is in development and is incomplete.

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 (§11.4.1) 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).

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 lines with respect to each other.

NOTE: This Element is in development and is incomplete.

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 (§11.4.2) 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 and 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 and 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 (§4.3) 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.

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
  ...
```

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*.

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 (§5.5). The `bend_type` parameter gives the "logical" shape of the bend. The setting of `bend_type` is only relavent when the bend curvature is varied. See §5.5 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 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 branche'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 global coordinates (§11.2) as the global 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 global 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 global 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 Lord Enum Group

Type of lord an element is. Possible values:

Lord
- .NOT — Not a lord
- .SUPER — Super lord
- .MULTIPASS — Multipass lord
- .GOVERNOR — Girder and other "minor" lords

4.1.6 Slave Enum Group

Type of slave an element is. Possible values:

Slave
- .NOT — Not a slave
- .SUPER — Super slave
- .MULTIPASS — Multipass slave

4.1.7 Loc Enum Group

Longitudinal location with respect to element's machine coordinates. Possible values:

Loc
- .UPSTREAM_END — Upstream element end
- .CENTER — center of element
- .INSIDE — Somewhere inside
- .DOWNSTREAM_END — Downstream element end

4.1.8 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

4.1.9 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.

4.2 FiducialPt Enum Group

Fiducial point location with respect to element's machine coordinates. Possible values:

FiducialPt
- .ENTRANCE_END — Entrance end of element
- .CENTER — Center of element
- .EXIT_END — Exit end of element
- .NONE — No point chosen

4.3 Holy Traits

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

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.

4.3.1 ApertureShape Holy Trait Group

The shape of an aperture.

- RECTANGULAR — Rectangular shape
- ELLIPTICAL — Elliptical shape

4.3.2 EleGeometry Holy Trait Group

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 parmeter groups are:

Group	*Section*	*Group*	*Section*
`AlignmentGroup`	§5.1	`LordSlaveGroup`	§5.12
`ApertureGroup`	§5.2	`MasterGroup`	§5.13
`BMultipoleGroup`	§5.3	`PatchGroup`	§5.14
`BeamBeamGroup`	§5.4	`RFCommonGroup`	§5.15
`BendGroup`	§5.5	`RFCavityGroup`	§5.16
`EMultipoleGroup`	§5.6	`RFAutoGroup`	§5.17
`FloorPositionGroup`	§5.7	`ReferenceGroup`	§5.18
`GirderGroup`	§5.8	`SolenoidGroup`	§5.19
`InitParticleGroup`	§5.9	`StringGroup`	§5.20
`LCavityGroup`	§5.10	`TrackingGroup`	§5.21
`LengthGroup`	§5.11	`TwissGroup`	§5.22

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
  ...
```

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.

5.1 AlignmentGroup

The `AlignmentGroup` gives the alignment (position and angular orientation) of the physical element relative to the nominal position defined by the machine 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 `tilt` keeping the entrance and exit endpoints fixed.

The parameters of the `AlignmentGroup` can be divided into two sub-groups. One group has a `_tot` suffix:

```
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.
tilt_tot::Number   - Rotation around the z-axis.
```

These "total alignment" parameters give the alignment of the element with with respect to the machine coordinates. The other sub-group of parameters do not have a `_tot` suffix:

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

These "relative alignment" parameters give the alignment of the element with respect to any `Girder` that is supporting the element. If there is no support `Girder`, the total alignment will be the same as the relative alignment. The relative alignment can be set

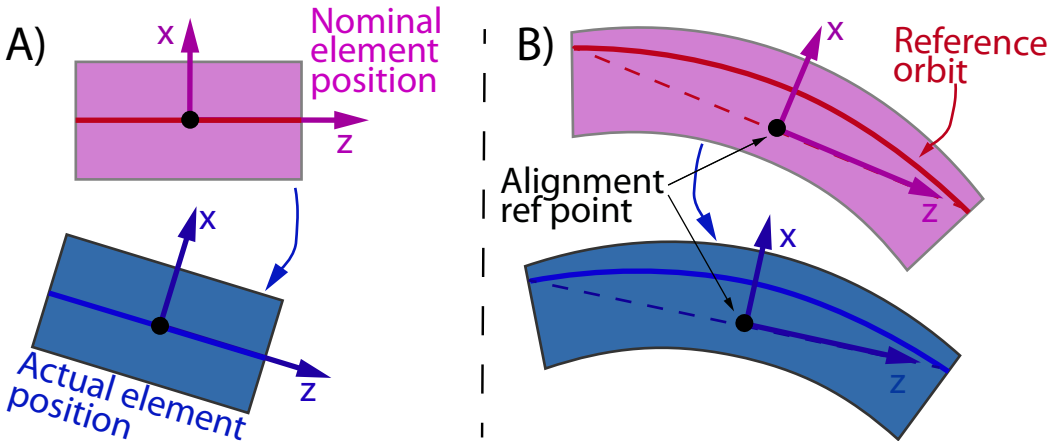

Figure 5.1: AlignmentGroup parameters The reference point is the origin about which the element alignment is calculated. A) For straight elements, the reference point is in the center of the element. For `Bend` elements, the reference point is at the midpoint of the chord connecting the entrance point 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.

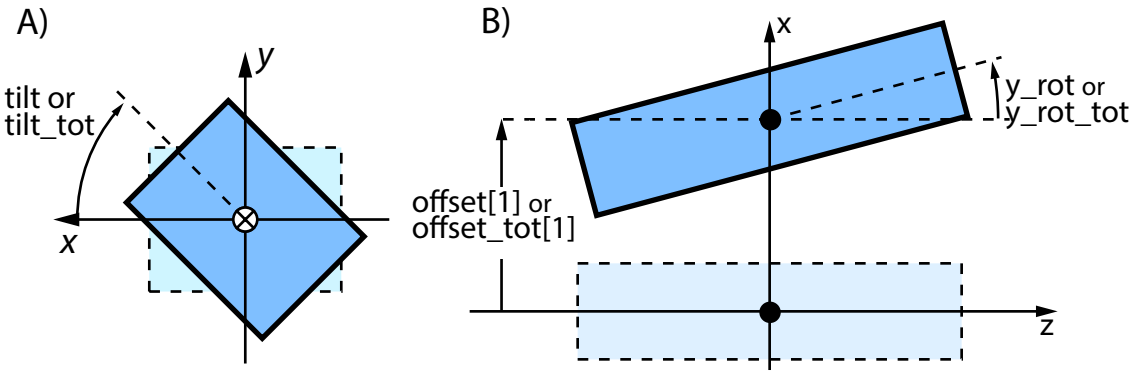

Figure 5.2: Alignment geometry. A) `tilt` (or `tilt_tot`) rotation. B) Combined `offset[1]` with `y_rot` (or `offset_tot[1]` with `y_rot_tot`).

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.

5.2 ApertureGroup

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

```
  x_limit::VectorNumber                    - 2-Vector of horizontal aperture limits. (m
  y_limit::VectorNumber                    - 2-Vector of vertical aperture limits. (m)
  vertex::VectorAcceleratorLattice.Vertex1  - Array of aperture vertexes. ()
  aperture_shape::ApertureShape              - Shape of aperture. Default is ELLIPTICAL
  aperture_at::AcceleratorLattice.BodyLoc.T   - Where the aperture is. Default is BodyLo
  misalignment_moves_aperture::Bool           - Does moving the element move the apertur
```

The aperture location of

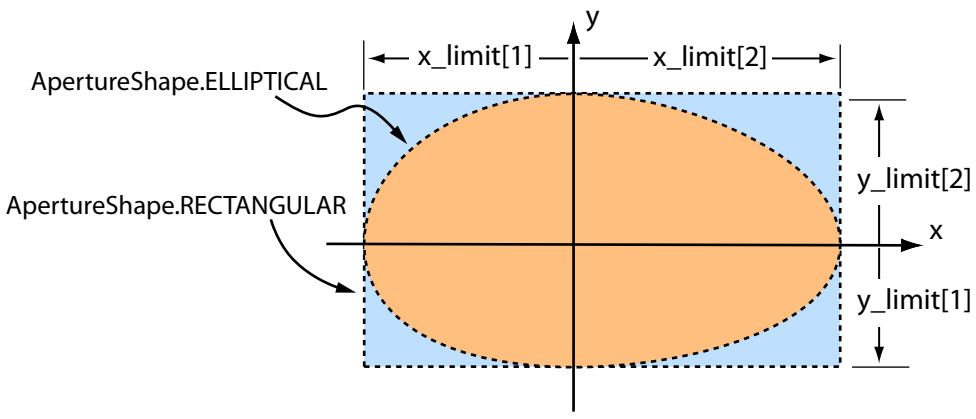

Figure 5.3: Rectangular and elliptical apertures. As drawn, `x_limit[1]` and `y_limit[1]` are negative and `x_limit[2]` and `y_limit[2]` are positive.

5.3 BMultipoleGroup

5.4 BeamBeamGroup

5.5 BendGroup

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

The parameters of this group are:

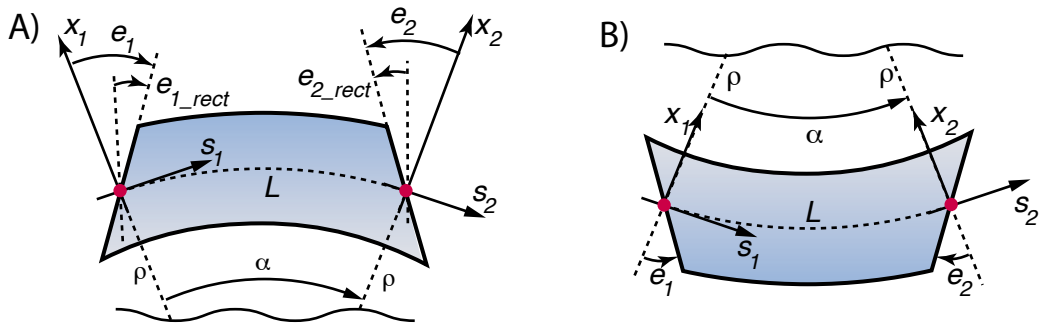

Figure 5.4: Bend geometry. Red dots are the entry and exit points that define the origin for the coordinate systems at the entry end (s_1, x_1) and exit ends (s_2, x_2) respectively. In the figure, the angle `alpha` is denoted α and the radius `rho` is denoted ρ. A) Bend geometry with positive bend angle. For the geometry shown, `g`, `angle`, `rho`, `e1`, `e2`, `e1_rect`, and `e2_rect` are all positive. B) Bend geometry with negative bend angle. For the geometry shown, `g`, `angle`, `rho`, `e1`, `e2`, `e1_rect`, and `e2_rect` are all negative. Note: The figures are drawn for zero `ref_tilt` where the rotation axis is parallel to the y-axis.

`BendGroup` parameters are:

angle
: The total design bend angle. A positive `angle` represents a bend towards negative x as shown in Fig. 5.4.

bend_field
: The `bend_field` parameter is the design magnetic bending field which determines the reference orbit and the placement of lattice elements downstream from the bend. The actual ("total") field is a vector sum of `bend_field` plus the value of the `Bn0` and `Bs0` multipoles. If `tilt0` and `Bs0` are zero, the actual field is
```
B-field (total) = bend_field + Bn0
```
See the discussion of `g` and `Kn0` below for more details.

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

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

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

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

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 `k1` and `k2` 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

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

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

fint1, fint2, hgap1, hgap2

The field integrals for the entrance pole face is given by the product of the `fint` and `hgap` parameters with `hgap` being the half gap between poles at the entrance face

$$F_{H1} \equiv F_{int1}\, H_{gap1} = \int_{pole} ds\, \frac{B_y(s)\,(B_{y0} - B_y(s))}{2\,B_{y0}^2} \tag{5.3}$$

For the exit pole face there is a similar equation using `fint2` and `hgap2` which defines F_{H2}. In the above equation B_{y0} is the field in the interior of the dipole. The values of `fint1`, `fint2`, `hgap1`, and `hgap2` are never used in isolation when tracking. Only the values for F_{H1} and F_{H2} matter. That is, to have an effect, both `fint` and `hgap` (or `fintx` and `hgapx`) must be non-zero.

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

F_{H1} = `fint * hgap = (fb1 + f1) / 12`
F_{H2} = `fintx * hgapx = (fb2 + f1) / 12`

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

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

where

$$P(s) = C_0 + C_1\, s + C_2\, s^2 + C_3\, s^3 + \ldots \tag{5.5}$$

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

$$C_1 = \frac{1}{2\, H_{gap}\, F_{int}} \tag{5.6}$$

fiducial_pt

The `fiducial_pt` parameter sets a fiducial point which can be used to keep the shape of the bend constant when, in a program, the parameters `rho`, `g`, `b_field` or `angle` are varied. Varying these parameters typically happens when doing machine design. Using a fiducial point can be helpful when designing a machine usin bend magnets that already exist.

The `fiducial_pt` parameter has four possible settings:

```
none          ! No fiducial point (default).
entrance_end  ! The entrance point is the fiducial point.
center        ! The center of the reference curve is the fiducial point.
exit_end      ! The exit point is the fiducial point.
```

With `fiducial_pt` set to `none` (the default). The bend shape is not held constant. With the other three settings, the bend shape will be held constant as discussed in §**??**. With `fiducial_pt` set to `entrance_end`, the reference trajectory at the entrance end is held fixed in both position and orientation with respect to the bend face and `g`, `l` and `e2`, along with the other depdendent parameters, are adjusted to both give the desired change in what was varied (which is one of `rho`, `g`, `b_field` or `angle`) and to keep the shape of the bend unchanged. See Fig. **??**. Similarly, if `fiducial_pt` is set to `center`, the center of the reference trajectory is held fixed in both position and orientation and if `fiducial_pt` is set to `exit_end`, the exit point is held fixed in both position and orientation.

g, rho

The design bending radius which determines the reference coordinate system is `rho` (see §11.1.1). `g` = `1/rho` is the "bend strength" and is proportional to the design dipole magnetic field. `g` is related to the design magnetic field `bend_field` via

$$\mathrm{g} = \frac{q}{p_0}\,\mathrm{bend\_field} \tag{5.7}$$

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 design 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

```
g_total = g + Kn0
```

Changing the multipole strength `Kn0` or `Ks0` leaves the design 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

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. 11.7. It is important to understand that

`ref_tilt`, unlike the `tilt` attribute of other elements, bends both the reference orbit along with the physical element. Note that the MAD `tilt` attribute for bends is equivalent to the *Bmad* `ref_tilt`. Bends in *Bmad* do not have a `tilt` attribute.

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 *Bmad* will calculate the appropriate value for the third.

In the local coordinate system (§11.1.1), looking from "above" (bend viewed from positive y), and with `ref_tilt` = 0, a positive `angle` represents a particle rotating clockwise. In this case. `g` will also be positive. For counterclockwise rotation, both `angle` and `g` will be negative but the length `l` is always positive. Also, looking from above, a positive `e1` represents a clockwise rotation of the entrance face and a positive `e2` represents a counterclockwise rotation of the exit face. This is true irregardless of the sign of `angle` and `g`. Also it is always the case that the pole faces will be parallel when

```
e1 + e2 = angle
```

5.6 EMultipoleGroup

5.7 FloorPositionGroup

5.8 GirderGroup

5.9 InitParticleGroup

5.10 LCavityGroup

5.11 LengthGroup

5.12 LordSlaveGroup

5.13 MasterGroup

5.14 PatchGroup

5.15 RFCommonGroup

5.16 RFCavityGroup

5.17 RFAutoGroup

5.18 ReferenceGroup

5.19 SolenoidGroup

5.20 StringGroup

5.21 TrackingGroup

5.22 TwissGroup

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 = Lat([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 = Lat("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 `Q1`s 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 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 = Lat("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 = Lat("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^{st} and 4^{th} elements in branch 1 and the second `A` lord controls the 2^{nd} and 5^{th} 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 where 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 = Lat("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 = Lat("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, ..., 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 = Lat("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 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 = Lat("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 global coordinates (§11.2) 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 = Lat("lat", zline)
```

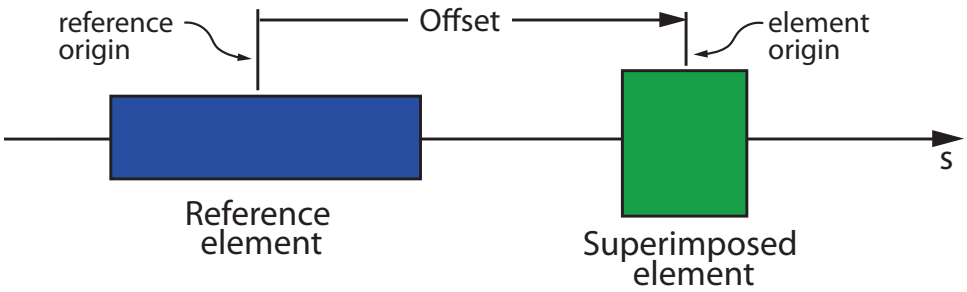

Figure 8.1: The superposition `offset` is the longitudinal s-distance from the origin point of the reference element to the origin point of the element being superimposed.

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

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

```
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 <: UnionBranch, 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 = Lat("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 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 where changed in the lattice file. There are two reasons for this. First, once the lattice file has been parsed, lattices can be "mangled" by adding or removing elements in a myriad of ways. This means that it is not possible to devise a general algorithm for adjusting superimposed element lengths that mirrors what the effect of changing the lengths in the lattice file.

Second, even if a lattice has not been mangled, an algorithm for varying lengths that is based on the superimpose information in the lattice file could lead to unexpected results. To see this consider the first example in Section §??. 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

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. To avoid confusion, *AcceleratorLattice.jl* will never be called *Bmad*, and *Bmad* always refers to the Fortran code (which is sometimes called Fortran *Bmad* for emphasis).

First a clarification. The name *Bmad* (referring to Fortran *Bmad*) can be used in two senses. There is *Bmad* the 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 *AcceleratorLattice.jl* 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.

Separation of tracking and lattice description: One of the first *AcceleratorLattice.jl* design decisions was to separate, as much as possible, particle tracking from the lattice description. This decision was 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 varing 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.

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 hugh consideration, an interface layer can be used to translate lattice parameters to a type stable form. Of much greater importance is 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.

Part II

Conventions and Physics

Chapter 11

Coordinates

Bmad uses three coordinate systems as illustrated in Fig. 11.1. First, the `global` (also called "`floor`") coordinates are independent of the placement accelerator. Things such as the building the accelerator is in may be described using `global` coordinates.

It is inconvenient to describe the position of a particle beam using the `global` coordinate system so a "`machine`" coordinate system is used (§11.1). This curvilinear coordinate system defines the nominal position of the lattice elements. The relationship between the `machine` and `global` coordinate systems is described in §11.2.

The "nominal" position of a lattice element is the position of the element without any position and orientation shifts (§5.1) which are sometimes referred to as "misalignments". Each lattice element has "`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 there are no misalignments, the `body` coordinates are aligned with the `machine` coordinates. The transformation between `machine` and `body` coordinates is given in §11.3. The $x = y = 0$ curved line of the machine coordinate system is known as the "`reference orbit`".

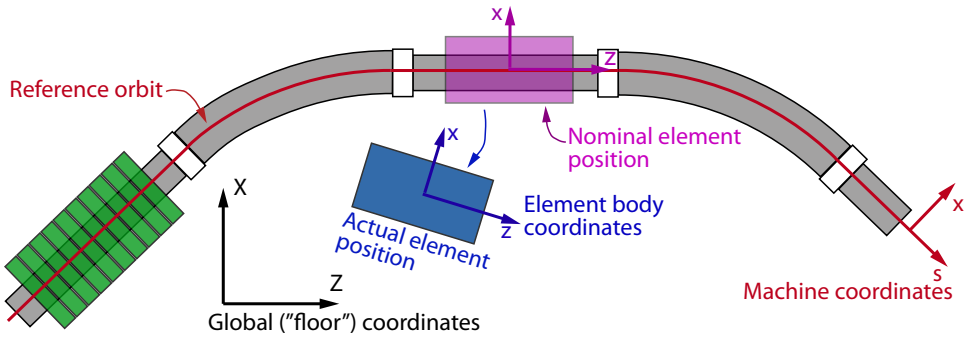

Figure 11.1: The `global` (or "`floor`") coordinate system is independent of the accelerator. The `machine` curvilinear coordinate system follows the bends of the accelerator. Each lattice element has `element body` coordinates which, if the element is not "misaligned" is the same as the `machine` coordinates. The $x = y = 0$ curved line of the machine coordinate system is known as the "reference orbit".

11.1 Machine Coordinates and Reference Orbit

11.1.1 The Reference Orbit

The `machine reference orbit` is the curved path used to define a coordinate system for describing a particle's position as shown in Fig. 11.2. The reference orbit is also used for orientating lattice elements in space. At a given time t, a particle's position can be described by a point $\mathcal{O}$ on the reference orbit a distance s relative to the reference orbit's zero position plus a transverse (x, y) offset. The point $\mathcal{O}$ on the reference orbit is used as the origin of the machine (x, y, z) coordinate system with the z–axis tangent to the reference orbit. The z–axis will generally be pointing in the direction of increasing s (Fig. 11.2A) but, as discussed below, will point counter to s for elements that are longitudinally `reversed` in orientation (Fig. 11.2B). The x and y–axes are perpendicular to the reference orbit and, by construction, the particle is always at $z = 0$. The coordinate system so constructed is called the "`machine coordinate system`" when there is need to distinguish it from the "`element body coordinate system`" (§11) which is attached to the physical element. There is a separate reference orbit for each branch (§1.8) of a lattice.

The reference orbit may not correspond to the orbit that any actual particle could travel. A common example is the `wiggler` element where particles always oscillate about the reference orbit which is a straight line.

Do not confuse this reference orbit (which defines the machine coordinate system) with the reference orbit about which the transfer maps are calculated (§**??**). The former is fixed by the lattice while the latter can be any arbitrary orbit.

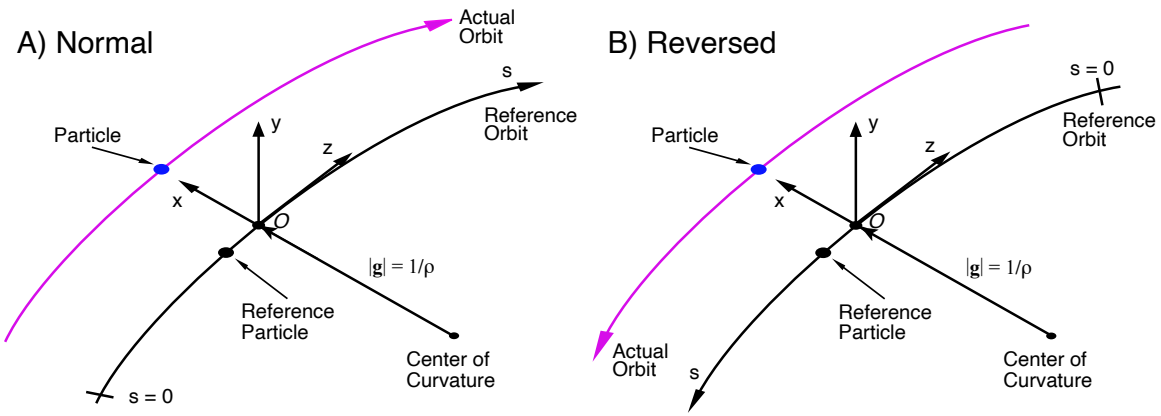

Figure 11.2: The reference coordinate system. By construction, a particle's z coordinate is zero. This is not to be confused with the phase space z coordinate (§11.4.2). The curvature vector $\mathbf{g}$ lies in the x-y plane and has a magnitude of $1/\rho$ where ρ is the bending radius. A) The z-axis will normally be parallel to the s-axis. B) For `reversed` elements it will be antiparallel. In both cases, the particle and reference particle are traveling in the direction of greater s.

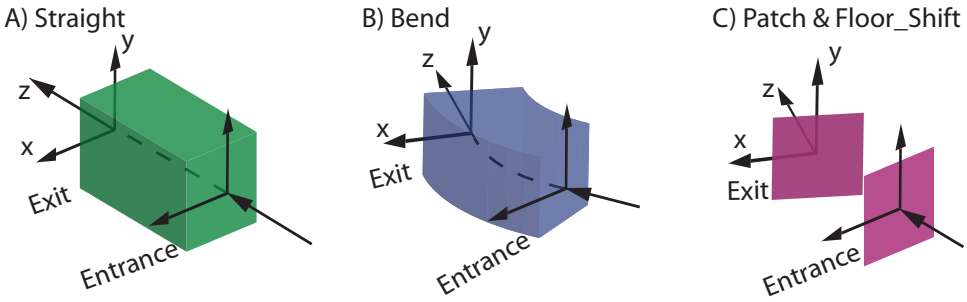

Figure 11.3: Lattice elements can be imagined as "LEGO blocks" which fit together to form the reference orbit along with the machine coordinate system. How elements join together is determined in part by their entrance and exit coordinate frames. A) For straight line elements the entrance and exit frames 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.

11.1.2 Element Entrance and Exit Coordinates

One way of thinking about the reference orbit and machine coordinates is to imagine that each element is like a LEGO block with an "`entrance`" and an "`exit`" coordinate frame as illustrated in Fig. 11.3[1]. These coordinate frames are attached to the element. that is, things like electric and magnetic fields, apertures, etc., are described with respect to the entrance and exit coordinates. Thus, for example, the `e1` edge of a `Bend` (§3.4) is always at the `entrance` face and the `e2` is always at the `exit` face. Most elements have a "straight" geometry as shown in Fig. 11.3A. That is, the reference orbit 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 orbit is a segment of a circular arc as shown in Fig. 11.3B. 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 (§11.2). For `Patch` (§3.24), and `floor_shift` (§3.11) elements, the exit face can can arbitrarily oriented with respect to the entrance end. For the `FloorShift` element the interior reference orbit between the entrance and exit faces is not defined. For the `Patch` element, the interior reference orbit is dependent upon certain `Patch` element parameter settings.

11.1.3 Reference Orbit and Machine Coordinates Construction

Assuming for the moment that there are no `Fiducial` elements present, the construction of the reference orbit 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 global coordinate system (§11) 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

[1] Thanks to Dan Abell for this analogy.

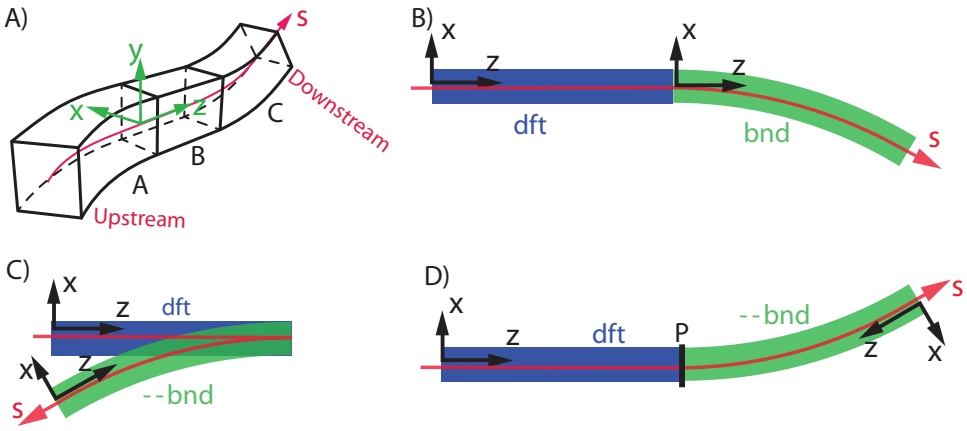

Figure 11.4: A) The machine coordinates are constructed by connecting the `downstream` reference frame of one element with the `upstream` reference frame of the next element in the branch. Coordinates shown is for the mating of element `A` to element B. B) Example with drift element `dft` followed by 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.

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 machine coordinates. All elements have an "`upstream`" and a "`downstream`" end as shown in Fig. 11.4A. The `downstream` end of an element is always farther (at greater s) from the beginning element than the `upstream` end of the element. Particles 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 machine coordinates are constructed beginning at these elements working both forwards and backwards along the branch.

Normally, the `upstream` end is the same as the element's `entrance` end (Fig. 11.3) and the `downstream` end is the same as the element's `exit` end. However, 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 machine coordinates is to mate the downstream reference frame of the element with the upstream reference frame of the next element in the branch so that, without misalignments, the (x, y, z) axes coincide[2]. This is illustrated in Fig. 11.4. Fig. 11.4A shows the general situation with the downstream frame of element `A` mated to the upstream frame of element `B`. Figures 11.4B-C

[2]If there are misalignments, the `entrance` and `exit` frames will move with the element. However, the `upstream` and `downstream` frames, along with the reference orbit and machine coordinates, will not move.

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])               ! No reversal.
CL = BeamLine([dft, reverse(bnd)])      ! Illegal. Do not use!
DL = BeamLine([dft, p, reverse(bnd)])   ! Valid.
```

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. 11.4B shows the branch constructed from `BL` containing an unreversed drift named `dft` connected to an unreversed bend named `bnd`. Fig. 11.4C 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. 11.4D shows the branch constructed from `DL`. Here a "`reflection`" patch `P` (§11.2.6) has been added to get a plausible geometry. The patch rotates the coordinate system around the y-axis by 180°(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 machine (x, y, z) coordinate system at all s-positions will always be a right-handed coordinate system.
- Care must be take 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°(by setting `x_rot` to `pi`) would also produce a plausible geometry.

11.1.4 Patch Element Coordinates

Generally, if a particle is reasonably near the reference orbit, there is a one-to-one mapping between the particle's position and (x, y, s) coordinates. A `Patch` (§3.24) elements with a non-zero `x_rot` or non-zero `y_rot` breaks the one-to-one mapping. This is illustrated in Fig. 11.5. The `Patch` element, shown schematically as an, irregular quadrilateral, is sandwiched between elements `ele_a` and `ele_b`. The machine coordinate system

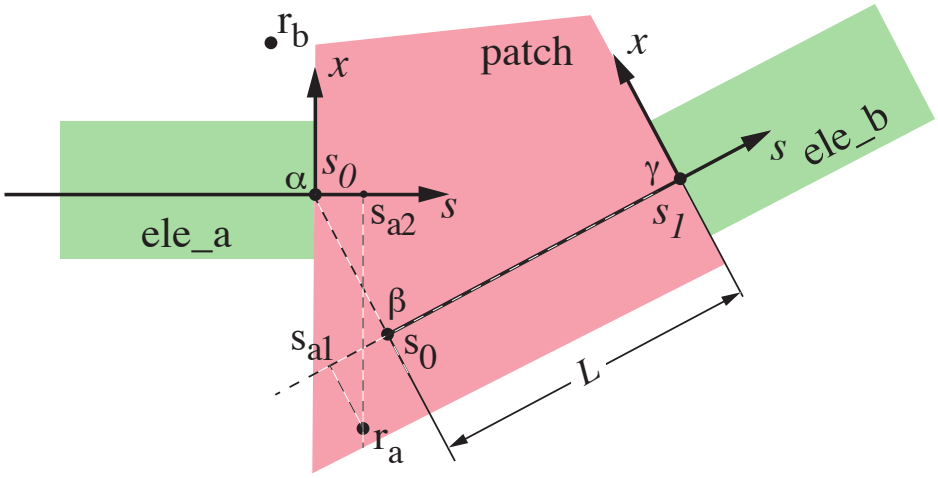

Figure 11.5: The machine reference coordinates in a `Patch` element. The `Patch` element, shown schematically as an irregular quadrilateral, is sandwiched between elements `ele_a` and `ele_b`. `L` is the length of the `Patch`. In this example, the `Patch` has a finite `y_rot`.

with origin at α are the coordinates at the end of `ele_a`. The coordinates at the end of the `Patch` has its origin labeled γ. By convention, the length of the patch `L` is taken to be the longitudinal distance from α to γ with the `Patch`'s exit coordinates defining the longitudinal direction. The "beginning" point of the `Patch` on the reference orbit a distance `L` from point γ is labeled β in the figure.

In the machine (x, y, s) coordinate system a particle at α will have some value $s = s_0$. A particle at point β will have the same value $s = s_0$ and a particle at γ will have $s = s_1 = s_0 + L$. A particle at point r_a in Fig. 11.5 illustrates the problem of assigning (x, y, s) coordinates to a given position. If the particle is considered to be within the region of `ele_a`, the particle's s position will be s_{a2} which is greater than the value s_0 at the exit end of the element. This contradicts the expectation that particles within `ele_a` will have $s \leq s_0$. If, on the other hand, the particle is considered to be within the `Patch` region, the particle's s position will be s_{a1} which is less than the value s_0 at the entrance to the patch. This contradicts the expectation that a particles within the `Patch` will have $s \geq s_0$.

To resolve this problem, *Bmad* considers a particle at position r_a to be within the `Patch` region. This means that there is, in theory, no lower limit to the s-position that a particle in the `Patch` region can have. This also implies that there is a discontinuity in the s-position of a particle crossing the exit face of `ele1`. Typically, when particles are translated from the exit face of one element to the exit face of the next, this `Patch` problem does not appear. It only appears when the track between faces is considered.

Notice that a particle at position r_b in Fig. 11.5 can simultaneously be considered to be in either `ele_a` or the `Patch`. While this creates an ambiguity it does not complicate tracking.

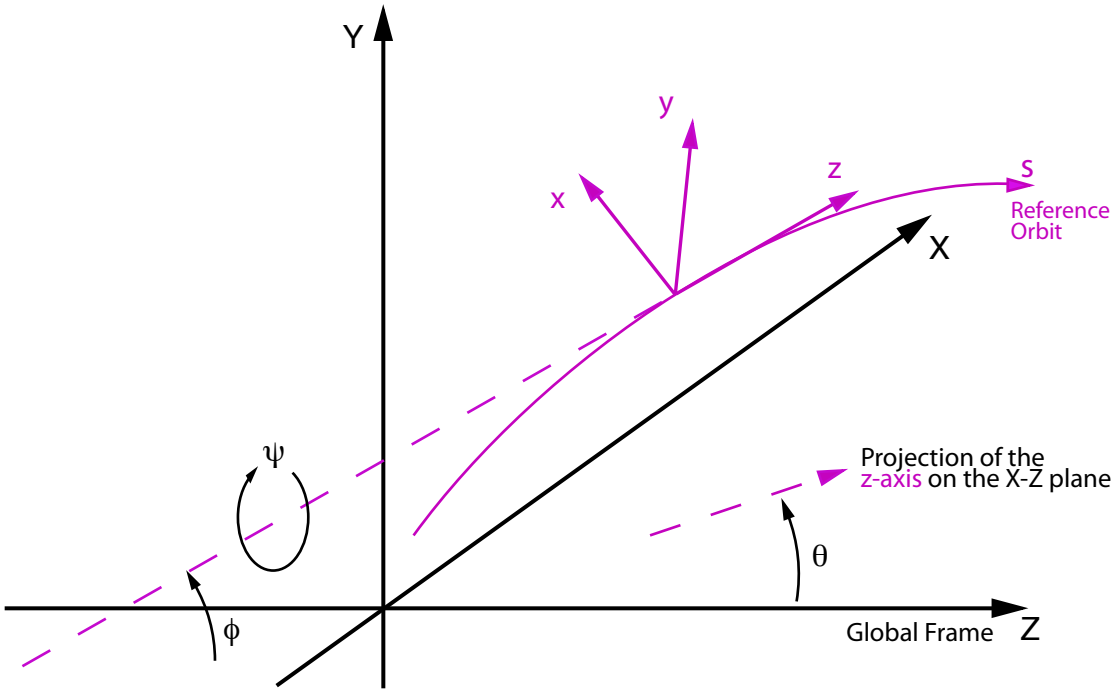

Figure 11.6: The machine (reference) coordinate system (purple), which is a function of s along the reference orbit, is described in the global coordinate system (black) by a position $(X(s), Y(s), Z(s))$ and and by angles $\theta(s)$, $\phi(s)$, and $\psi(s)$.

11.2 Global Coordinates

The Cartesian `global` coordinate system, also called the 'floor" coordinate system, is the coordinate system "attached to the earth" that is used to describe the machine coordinate system. Following the *MAD* convention, the `global` coordinate axis are labeled (X, Y, Z). Conventionally, Y is the "vertical" coordinate and (X, Z) are the "horizontal" coordinates. To describe how the machine coordinate system is oriented within the global coordinate system, each point on the s-axis of the machine 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 machine coordinate axes as shown in Fig. 11.6. 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 ψ forms a right–handed screw with the z–axis.

By default, at $s = 0$, the reference orbit's origin coincides with the (X, Y, Z) origin and the x, y, and z axes correspond to the X, Y, and Z axes respectively. 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, θ decreases as one follows the reference orbit when going through a horizontal bend with a positive bending angle. This corresponds to x pointing radially outward. Without any vertical bends, the Y and y axes will coincide, and ϕ and ψ will both be zero. The `beginning` statement (§??) in a lattice file can be use to override these defaults.

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

$$\mathbf{V} = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \tag{11.1}$$

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 machine coordinate axes in the order (x, y, z). $\mathbf{W}$ can be expressed in terms of the orientation angles θ, ϕ, and ψ via the formula

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

where

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

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 β

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

11.2.1 Lattice Element Positioning

Bmad, again following *MAD*, computes $\mathbf{V}$ and $\mathbf{W}$ by starting at the first element of the

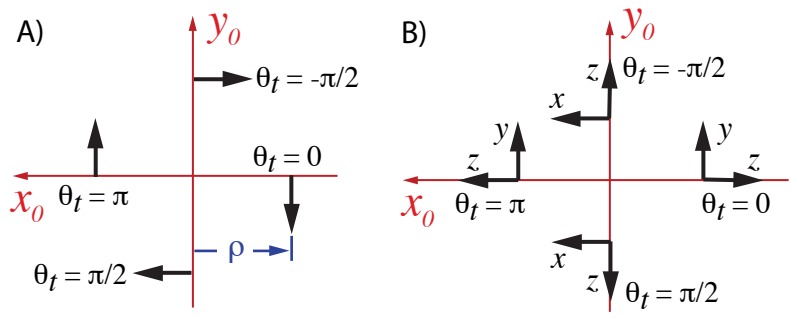

Figure 11.7: A) Rotation axes (bold arrows) for four different `ref_tilt` angles of $\theta_t = 0, \pm\pi/2$, and π. (x_0, y_0, z_0) are the machine coordinates at the entrance end of the bend with the z_0 axis being directed into the page. Any rotation axis will be displaced by a distance of the bend 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 $\pi/2$.

lattice and iteratively using the equations

$$\mathbf{V}_i = \mathbf{W}_{i-1}\,\mathbf{L}_i + \mathbf{V}_{i-1}, \tag{11.5}$$
$$\mathbf{W}_i = \mathbf{W}_{i-1}\,\mathbf{S}_i \tag{11.6}$$

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

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

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. 11.7A. The axis of rotation points in the negative y_0 direction for `ref_tilt` = 0 and is offset by the bend radius `rho`. Here (x_0, y_0, z_0) are the machine 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. 11.7B shows the exit coordinates for four different values of `ref_tilt` and for a bend angle `angle` of $\pi/2$. Notice that for a bend in the horizontal $X - Z$ plane, a positive bend `angle` will result in a decreasing azimuth angle θ.

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

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

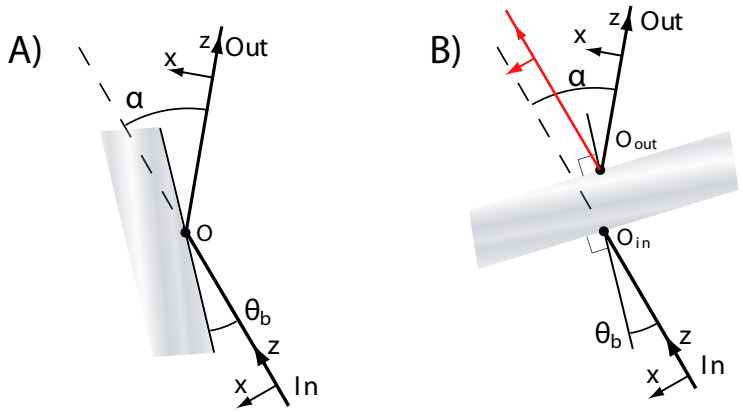

Figure 11.8: Mirror and crystal geometry. The geometry shown here is appropriate for a `ref_tilt` angle of $\theta_t = 0$. θ_g is the bend angle of the incoming (entrance) ray, and α_b is the total bend angle of the reference trajectory. A) Geometry for a mirror or a Bragg crystal. Point $\mathcal{O}$ is the origin of both the machine coordinates just before and just after the reflection/diffraction. B) Geometry for a Laue crystal. Point $\mathcal{O}_{out}$ is the origin of the coordinates just after diffraction is displaced from the origin $\mathcal{O}_{in}$ just before diffraction due to the finite thickness of the crystal. here the bend angles are measured with respect to the line that is in the plane of the entrance and exit coordinates and perpendicular to the surface. For Laue diffraction, the user has the option of using the undiffracted beam (shown in red) as the reference trajectory.

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

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

where α_b is the bend `angle` (§3.4) and ρ being the bend radius (`rho`). Notice that since $\mathbf{u}$ is perpendicular to z, the curvilinear reference coordinate system has no "torsion". That is, it is a Frenet-Serret coordinate system.

Note: An alternative equation for $\mathbf{S}$ for a bend is

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

The bend transformation above is so constructed that the transformation is equivalent to rotating the machine coordinate system around an axis that is perpendicular to the plane of the bend. This rotation axis is invariant under the bend transformation. For example, for $\theta_t = 0$ (or π) the y-axis is the rotation axis and the y-axis of the machine coordinates before the bend will be parallel to the y-axis of the machine coordinates after the bend as shown in Fig. 11.7. That is, a lattice with only bends with $\theta_t = 0$ or π will lie in the horizontal plane (this assuming that the y-axis starts out pointing along the Y-axis as it does by default). For $\theta_t = \pm\pi/2$, the bend axis is the x-axis. A value of $\theta_t = +\pi/2$ represents a downward pointing bend.

11.2.2 Position Transformation When Transforming Coordinates

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

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

This is essentially the inverse of Eq. (11.5). That is, 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

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

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

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

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

11.2.3 Crystal and Mirror Element 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 orbit is modeled as a zero length bend with $\widetilde{\mathbf{L}} = (0, 0, 0)$, as shown in Fig. 11.8A. Shown in the figure is the geometry appropriate for a `ref_tilt` angle of $\theta_t = 0$ (the rotation axis is here the y-axis). Since the mirror or crystal element is modeled to be of zero length, the origin points (marked $\mathcal{O}$ in the figure) of the entrance and exit machine coordinates are the same. For Laue diffraction, the only difference is that $\widetilde{\mathbf{L}}$ is non-zero due to the finite thickness of the crystal as shown in Fig. 11.8B. This results in a separation between the entrance coordinate origin $\mathcal{O}_{in}$ and the exit coordinate origin $\mathcal{O}_{out}$.

In all cases, the total bending angle is

$$\begin{aligned}
&\alpha_b = \text{bragg_angle_in} + \text{bragg_angle_out} && \text{! Crystal, graze_angle_in} = 0 \\
&\alpha_b = \text{graze_angle_in} + \text{graze_angle_out} && \text{! Crystal, graze_angle_in} \neq 0 \\
&\alpha_b = 2\,\text{graze_angle} && \text{! Mirror}
\end{aligned} \tag{11.15}$$

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 machine 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 machine 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 §**??**.

11.2.4 Patch and FloorShift Elements 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

$$\mathbf{L} = \begin{pmatrix} \text{x\_offset} \\ \text{y\_offset} \\ \text{z\_offset} \end{pmatrix}$$
$$\mathbf{S} = \mathbf{R}_y(\text{y\_rot})\ \mathbf{R}_x(\text{y\_rot})\ \mathbf{R}_z(\text{tilt}) \tag{11.16}$$

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`.

11.2.5 Fiducial and Girder Elements Origin Shift Transformation

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

$$\mathbf{L} = \begin{pmatrix} \text{dx\_origin} \\ \text{dy\_origin} \\ \text{dz\_origin} \end{pmatrix}$$
$$\mathbf{S} = \mathbf{R}_y(\text{dtheta\_origin})\ \mathbf{R}_x(\text{-dphi\_origin})\ \mathbf{R}_z(\text{dpsi\_origin}) \tag{11.17}$$

11.2.6 Reflection Patch

A `Patch` (or a series of patches) that reflects the direction of the `z`-axis is called a `reflection Patch`. By "reflected direction" it is meant that the dot product $\mathbf{z}_1 \cdot \mathbf{z}_2$ is negative where $\mathbf{z}_1$ is the z-axis vector at the `entrance` face and $\mathbf{z}_2$ is the z-axis vector at the `exit` face. This condition is equivalent to the condition that the associated $\mathbf{S}$ matrix (see Eq. (11.16)) satisfy:

$$S(3,3) < 0 \tag{11.18}$$

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

$$\cos(\text{x\_rot})\,\cos(\text{y\_rot}) < 0 \tag{11.19}$$

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. (11.18).

11.3 Transformation Between Machine and Element Body Coordinates

The `element body` coordinates are the coordinate system attached to an element. Without any misalignments, where "`misalignments`" are here defined to be any offset, or rotation (§**??**), the `machine` coordinates (§11.1.1) and `element body` coordinates are the same. With misalignments, the transformation between `machine` and `element body` coordinates depends upon whether the machine coordinate system is straight (§11.3.1) or bent (§11.3.2).

When tracking a particle through an element, the particle starts at the `nominal` (§11) upstream end of the element with the particle's position expressed in machine 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 machine 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.

11.3.1 Straight Element Misalignment Transformation

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

$$\Lambda_s \longrightarrow \Lambda_{\text{mid}} \longrightarrow E_{\text{mid}} \longrightarrow E_s \tag{11.20}$$

where Λ_{mid} and E_{mid} are the machine and element reference frames at the center of the element.

The first and last transformations from Λ_s to Λ_{mid} and from E_{mid} to E_s use Eqs. (11.5), (11.6), and (11.7) with the replacement $L \to L/2 - s$ for the first transformation and

$L \to s - L/2$ for the third transformation. The middle transformation, by definition of the offset and rotation parameters is

$$\mathbf{L} = \begin{pmatrix} \text{x\_offset} \\ \text{y\_offset} \\ \text{z\_offset} \end{pmatrix}$$
$$\mathbf{S} = \mathbf{R}_y(\text{x\_rot})\, \mathbf{R}_x(\text{y\_rot})\, \mathbf{R}_z(\text{tilt}) \tag{11.21}$$

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

11.3.2 Bend Element Misalignment Transformation

For `Bend` elements there is a `ref_tilt` as well as a `tilt` parameter. The former affects both the reference orbit 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 (§5.1). The different reference frame used for `ref_tilt` versus everything else means that five transformations are needed to get from the machine frame to the element body frame (see Eq. (11.20)). Symbolically:

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

1. $\Lambda_s \longrightarrow \Lambda_0$
   In machine coordinates, transform from Λ_s, the coordinates at a distance s from the beginning of the element, to Λ_0 the machine 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 $\alpha_b \to -s/\rho$.

2. $\Lambda_0 \longrightarrow \Xi_0$
   Transform from Λ_s to Ξ_0 which is the "chord" coordinate system obtained by rotating Λ_0 around the axis perpendicular to the bend plane such that the z axis of Ξ_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 $\alpha_b \to \alpha_b/2$

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 $L \to L_c/2$ and L_c is the chord length.

4. $\Xi_{\text{mid}} \longrightarrow \Omega_{\text{c,mid}}$
   Transform from chord coordinates at the center of the chord to "misaligned chord

coordinates" at the same point. This is the same transform as used for straight elements:

$$\mathbf{L} = \begin{pmatrix} \text{x\_offset} \\ \text{y\_offset} \\ \text{z\_offset} \end{pmatrix}$$
$$\mathbf{S} = \mathbf{R}_y(\text{x\_rot})\, \mathbf{R}_x(\text{y\_rot})\, \mathbf{R}_z(\text{tilt}) \tag{11.23}$$

5. $\Omega_{c,mid} \longrightarrow \Omega_{c,0}$
   Transform in misaligned 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 misaligned 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.

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

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 $\alpha_b \rightarrow L/\rho$.

11.4 Phase Space Coordinates

11.4.1 Reference Particle, Reference Energy, and Reference Time

The `reference energy` and `reference time` are needed in evaluating the phase space coordinates of charged particles (§11.4.2).

All lattice elements, except for controller elements, have an associated `reference energy` energy. The reference energy at the start of a lattice's `root branch` (§1.8) is set in the lattice file by setting the reference momentum (`p0c`) or total energy (`E_tot`) using a `parameter` (§**??**) or `beginning` (§**??**) statement. For other branches, the energy at the start of the branch is set using the appropriate line parameter (§**??**) statement.

Note that the reference momentum `p0c` is actually the reference momentum times the speed of light so that the reference momentum has the same unit (eV) as the reference energy.

For most elements, the reference energy is the same as the reference energy of the proceeding element. The following elements are exceptions:

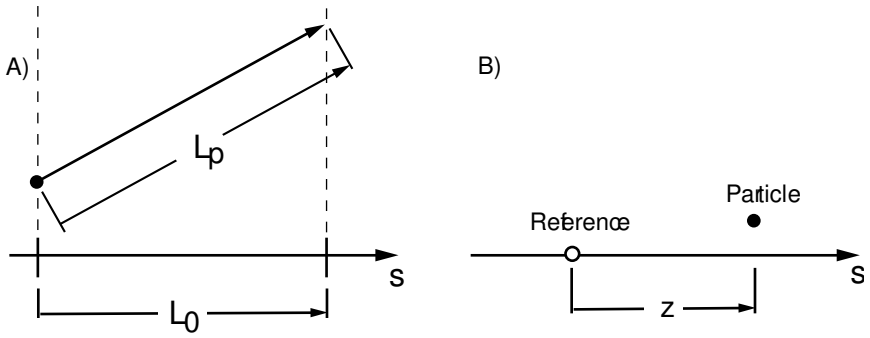

Figure 11.9: Interpreting phase space z at constant velocity: A) The change in z going through an element of length L_0 is $L_0 - L_p$. B) At constant time, z is the longitudinal distance between the reference particle and the particle.

```
custom
em_field
hybrid
lcavity
patch
```

The reference energy of these elements is determined by tracking a particle (the "`reference particle`") through the element with the particle starting on the reference orbit and whose energy is equal to the reference energy. The energy of the particle at the downstream end is the reference energy of the element. Note: Tracking through an element to determine the reference energy is always done with the element turned on independent of the setting of the element's `is_on` (§**??**) parameter. Reference energy tracking is also done ignoring any orientation parameters (§**??**) and errors like `voltage_err`.

Besides the reference energy, lattice elements have an associated `reference time` which is computed, for most elements, by the time-of-flight of the `reference particle` assuming that the reference particle is following the reference orbit. Exceptions are `wiggler` elements which uses the time-of-flight of the actual undulating trajectory. [Actually what is used in the computation of the z phase space coordinate (Eq. (11.27)) is the sum of reference time deltas of the elements that a particle has passed through. It is not possible to assign a unique reference time to an element when particles are recirculating through elements as in a storage ring.]

11.4.2 Charged Particle Phase Space Coordinates

For charged particles (more correctly, for everything but photons (§11.4.4)), *Bmad* uses the canonical phase space coordinates

$$\mathbf{r}(s) = (x, p_x, y, p_y, z, p_z) \tag{11.25}$$

The longitudinal position s is the independent variable instead of the time. x and y, are the transverse coordinates of the particle as shown in Fig. 11.2A. Note that x and y are

independent of the position of the reference particle.

The phase space momenta p_x and p_y are normalized by the reference (sometimes called the design) momentum P_0

$$p_x = \frac{P_x}{P_0}, \qquad p_y = \frac{P_y}{P_0} \tag{11.26}$$

where P_x and P_y are respectively the x and y momentums.

The phase space z coordinate is

$$\begin{aligned} z(s) &= -\beta(s)\, c\, (t(s) - t_0(s)) \\ &\equiv -\beta(s)\, c\, \Delta t(s) \end{aligned} \tag{11.27}$$

$t(s)$ is the time at which the particle is at position s, $t_0(s)$ is the time at which the reference particle is at position s, and β is v/c with v being the particle velocity (and not the reference velocity). The reference particle is, by definition, "synchronized" with elements whose fields are oscillating and therefore the actual fields a particle will see when traveling through such an element will depend upon the particle's phase space z. For example, the energy change of a particle traveling through an `lcavity` (§**??**) or `rfcavity` (§**??**) element is z dependent. Exception: With absolute time tracking (§**??**) fields are tied to the absolute time and not z.

If the particle's velocity is constant, and is the same as the velocity of the reference particle (for example, at high energy where $\beta = 1$ for all particles), then $\beta\, c\, t$ is just the path length. In this case, the change in z going through an element is

$$\Delta z = L_0 - L_p \tag{11.28}$$

where, as shown in Fig. 11.9A, L_0 is the path length of the reference particle (which is just the length of the element) and L_p is the path length of the particle in traversing the element. Another way of interpreting phase space z is that, at constant β, and constant time, z is the longitudinal distance between the particle and the reference particle as shown in Fig. 11.9B. with positive z indicating that the particle is ahead of the reference particle.

Do not confuse the phase space z with the z that is the particle's longitudinal coordinate in the machine reference frame as shown in Fig. 11.2. By construction, this latter z is always zero.

Notice that if a particle gets an instantaneous longitudinal kick so that β is discontinuous then, from Eq. (11.27), phase space z is discontinuous even though the particle itself does not move in space. In general, from Eq. (11.27), The value of z for a particle at s_2 is related to the value of z for the particle at s_1 by

$$z_2 = \frac{\beta_2}{\beta_1}\, z_1 - \beta_2\, c\, (\Delta t_2 - \Delta t_1) \tag{11.29}$$

$\Delta t_2 - \Delta t_1$ can be interpreted as the difference in transit time, between the particle and the reference particle, in going from s_1 to s_2.

The longitudinal phase space momentum p_z is given by

$$p_z = \frac{\Delta P}{P_0} \equiv \frac{P - P_0}{P_0} \tag{11.30}$$

where P is the momentum of the particle. For ultra–relativistic particles p_z can be approximated by

$$p_z = \frac{\Delta E}{E_0} \tag{11.31}$$

where E_0 is the reference energy (energy here always refers to the total energy) and $\Delta E = E - E_0$ is the deviation of the particle's energy from the reference energy. For an `Lcavity` element (§**??**) the reference momentum is *not* constant so the tracking for an `Lcavity` is not canonical.

MAD uses a different coordinate system where (z, p_z) is replaced by $(-c\Delta t, p_t)$ where $p_t \equiv \Delta E / P_0 c$. For highly relativistic particles the two coordinate systems are identical.

The relationship, between the phase space momenta and the slopes $x' \equiv dx/ds$ and $y' \equiv dy/ds$ is

$$x' = \frac{p_x}{\sqrt{(1+p_z)^2 - p_x^2 - p_y^2}}\,(1 + gx) \tag{11.32}$$

$$y' = \frac{p_y}{\sqrt{(1+p_z)^2 - p_x^2 - p_y^2}}\,(1 + gx) \tag{11.33}$$

$g = 1/\rho$ is the curvature function with ρ being the radius of curvature of the reference orbit and it has been assumed that the bending is in the x–z plane.

With the paraxial approximation, and in the relativistic limit, the change in z with position is

$$\frac{dz}{ds} = -g\,x - \frac{1}{2}(x'^2 + y'^2) \tag{11.34}$$

This shows that in a linac, without any bends, the z of a particle always decreases.

A particle can also have a spin. The spin is characterized by the spinor $\Psi = (\psi_1, \psi_2)^T$ where $\psi_{1,2}$ are complex numbers (§**??**).

11.4.3 Time-based Phase Space Coordinates

Some specialized routines (for example, time Runge Kutta tracking) use the time t as the independent variable for charged particle tracking. This is useful when particles can reverse direction since the normal z based tracking cannot handle this. Direction reversal can happen, for example, with low energy "dark current" electrons that are generated at the walls of the vacuum chamber.

When the tracking is time based the phase space coordinates are:

$$(x, c\,p_x, y, c\,p_y, z, c\,p_s) \tag{11.35}$$

The positions x, y, and z are the same as with phase space coordinates (§11.4.2). The momenta are defined as

$$\begin{aligned} cp_x &\equiv mc^2\gamma\beta_x \\ cp_y &\equiv mc^2\gamma\beta_y \\ cp_s &\equiv mc^2\gamma\beta_s, \end{aligned} \tag{11.36}$$

and internally are stored in units of eV.

11.4.4 Photon Phase Space Coordinates

The phase space coordinates discussed above implicitly assume that particles are traveling longitudinally in only one direction. That is, the sign of the s component of the momentum cannot be determined from the phase space coordinates. This is generally fine for tracking high energy beams of charged particles but for photon tracking this would oftentimes be problematical. For photons, therefore, a different phase space is used:

$$(x, \beta_x, y, \beta_y, z, \beta_z) \tag{11.37}$$

Here $(\beta_x, \beta_y, \beta_z)$ is the normalized photon velocity with

$$\beta_x^2 + \beta_y^2 + \beta_z^2 = 1 \tag{11.38}$$

and (x, y, z) are the reference orbit coordinates with z being the distance from the start of the lattice element the photon is in.

In *Bmad*, the information associated with a photon include its phase space coordinates and time along with the photon energy and four parameters E_x, ϕ_x, and E_y, ϕ_y specifying the intensity and phase of the field along the x and y axes transverse to the direction of propagation. the field in the vicinity of the photon is

$$\begin{aligned} E_x(\mathbf{r}, t) &\sim E_x \, e^{i(k\,(z-z_0)-\omega\,(t-t_{\rm ref})+\phi_x)} \\ E_y(\mathbf{r}, t) &\sim E_y \, e^{i(k\,(z-z_0)-\omega\,(t-t_{\rm ref})+\phi_y)} \end{aligned} \tag{11.39}$$

where z_0 is the photon z position and and $t_{\rm ref}$ is the reference time.

The normalization between field and intensity is dependent upon the particular parameters of any given simulation and so must be determined by the program using *Bmad*.

Chapter 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

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

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

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

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

$$K_n \equiv \frac{q\, B_n}{P_0} \tag{12.3}$$

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 ρ is the bending radius of a particle with the reference energy in a field of strength B. Notice that P_0 is the local reference momentum at the element which may not be the same as the reference energy at the beginning of the lattice if there are `lcavity` elements (§**??**) present.

The kicks Δp_x and Δp_y that a particle experiences going through a multipole field is

$$\begin{aligned}\Delta p_x &= \frac{-q\,L\,B_y}{P_0} \qquad (12.4)\\ &= -K_0 L - K_1 L\,x \;+\; \frac{1}{2}K_2 L(y^2 - x^2) + \frac{1}{6}K_3 L(3xy^2 - x^3) \;+\; \ldots\\ \Delta p_y &= \frac{q\,L\,B_x}{P_0} \qquad (12.5)\\ &= \qquad\quad K_1 L\,y + \quad K_2 L\,xy \qquad + \frac{1}{6}K_3 L(3x^2 y - y^3) \;+\; \ldots\end{aligned}$$

A positive $K_1 L$ quadrupole component gives horizontal focusing and vertical defocusing. The general form is

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

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

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

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

where (r, θ) are the polar coordinates of the point (x, y).

Note that, for compatibility with MAD, the $K0L$ component of a `Multipole` element rotates the reference orbit essentially acting as a zero length bend. This is not true for multipoles of any other type of element.

Instead of using magnitude K_n and rotation angle θ_n, Another representation is using normal $\widetilde{K}_n$ and skew $\widetilde{S}_n$. The conversion between the two are

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

Another representation of the magnetic field used by *Bmad* divides the fields into normal b_n and skew a_n components. In terms of these components the magnetic field for the n^{th} order multipole is

$$\frac{q\,L}{P_0}\,(B_y + iB_x) = (b_n + ia_n)\,(x + iy)^n \qquad (12.10)$$

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 $(K_n L, S_n L, T_n)$ is

$$b_n + i a_n = \frac{1}{n!} \left(K_n L + i\, S_n L\right) e^{-i(n+1)T_n} \tag{12.11}$$

In the case where $S_n L = 0$

$$K_n L = n! \sqrt{a_n^2 + b_n^2} \tag{12.12}$$

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

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

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

For example, a normal quadrupole rotated by 45° becomes a skew quadrupole.

The multipole fields can be "`reference energy`" scaled and/or "`element strength`" scaled. Scaling here means that the a_n and b_n values used in tracking are scaled from the input values given in the lattice file.

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

$$\left[a_n, b_n\right] \longrightarrow \left[a_n, b_n\right] \cdot \frac{q}{P_0} \tag{12.15}$$

`Element strength` scaling is applied when the multipoles are associated with a non `AB_Multipole` element and if the `scale_multipoles` attribute (§**??**) is `True`. This scaling uses a measurement radius r_0 and a scale factor F:

$$\left[a_n, b_n\right] \longrightarrow \left[a_n, b_n\right] \cdot F \cdot \frac{r_0^{n_{\rm ref}}}{r_0^n} \tag{12.16}$$

r_0 is set by the `r0_mag` attribute of an element. F and $n_{\rm ref}$ are set automatically depending upon the type of element as shown in Table 12.1. The γ_p term is

Element	F	$n_{\rm ref}$
Elseparator	$\sqrt{\texttt{Hkick}^2 + \texttt{Vkick}^2}$	0
Hkicker	Kick	0
Kicker,AC_Kicker	$\sqrt{\texttt{Hkick}^2 + \texttt{Vkick}^2}$	0
Rbend	G * L	0
Sbend	G * L	0
Vkicker	Kick	0
Wiggler	$\frac{2\,c\,\texttt{L\_pole}\,B_{max}}{\pi\,\texttt{p0c}}$	0
Quadrupole	K1 * L	1
Sol_Quad	K1 * L	1
Solenoid	KS * L	1
Sextupole	K2 * L	2
Octupole	K3 * L	3

Table 12.1: F and $n_{\rm ref}$ for various elements.

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 ϕ_n for the n^{th} order multipole is

$$\phi_n = -\,\mathrm{Re}\left[\frac{b_{en} - i a_{en}}{n+1}\,\frac{(x+iy)^{n+1}}{r_0^n}\right] \tag{12.17}$$

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

$$E_x - i E_y = (b_{en} - i a_{en})\,\frac{(x+iy)^n}{r_0^n} \tag{12.18}$$

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

$$\frac{dp_x}{ds} = \frac{q\,E_x}{\beta\,P_0\,c}, \qquad \frac{dp_y}{ds} = \frac{q\,E_y}{\beta\,P_0\,c} \tag{12.19}$$

Where β 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 (12.18). The analysis of the multipole fields in a bend here follows McMillan[**b:mcmillan**].

In the rest of this section, normalized coordinates $\widetilde{r} = r/\rho$, $\widetilde{x}/ = x/\rho$, and $\widetilde{y} = y/\rho$ will be used where ρ is the bending radius of the reference coordinate system, r is the distance, in the plane of the bend, from the bend center to the observation point, x is the distance in the plane of the from the reference coordinates to the observation point and y is the distance out-of-plane. With this convention $\widetilde{r} = 1 + \widetilde{x}$.

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

$$\frac{1}{\widetilde{r}}\frac{\partial}{\partial\widetilde{r}}\left(\widetilde{r}\,\frac{\partial\,\phi}{\partial\,\widetilde{r}}\right) + \frac{\partial^2\phi}{\partial\,\widetilde{y}^2} = 0 \tag{12.20}$$

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

$$\phi_n^r = \frac{-1}{1+n}\sum_{p=0}^{2p\le n+1}\binom{n+1}{2\,p}\,(-1)^p\,F_{n+1-2p}(\widetilde{r})\,\widetilde{y}^{2p} \tag{12.21}$$

and in the form

$$\phi_n^i = \frac{-1}{1+n}\sum_{p=0}^{2p\le n}\binom{n+1}{2p+1}\,(-1)^p\,F_{n-2p}(\widetilde{r})\,\widetilde{y}^{2p+1} \tag{12.22}$$

where $\binom{a}{b}$ ("a choose b") denotes a binomial coefficient, and n is the order number which can range from 0 to infinity.[1]

In Eq. (12.22) the $F_p(\widetilde{r})$ are related by

$$F_{p+2} = (p+1)\,(p+2)\int_1^{\widetilde{r}}\frac{d\widetilde{r}}{\widetilde{r}}\left[\int_1^{\widetilde{r}} d\widetilde{r}\,\widetilde{r}\,F_p\right] \tag{12.23}$$

with the "boundary condition":

$$\begin{aligned} F_0(\widetilde{r}) &= 1 \\ F_1(\widetilde{r}) &= \ln\,\widetilde{r} \end{aligned} \tag{12.24}$$

This condition ensures that the number of terms in the sums in Eqs. (12.21) and (12.22)

[1] Notice that here n is related to m in McMillian's paper by $m = n + 1$. Also note that the ϕ^r and ϕ^i here have a normalization factor that is different from McMillian.

are finite. With this condition, all the F_p can be constructed:

$$\begin{aligned}
F_1 &= \ln \widetilde{r} = \widetilde{x} - \frac{1}{2}\widetilde{x}^2 + \frac{1}{3}\widetilde{x}^3 - \dots \\
F_2 &= \frac{1}{2}(\widetilde{r}^2 - 1) - \ln \widetilde{r} = \widetilde{x}^2 - \frac{1}{3}\widetilde{x}^3 + \frac{1}{4}\widetilde{x}^4 - \dots \\
F_3 &= \frac{3}{2}[-(\widetilde{r}^2 - 1) + (\widetilde{r}^2 + 1)\ln \widetilde{r}] = \widetilde{x}^3 - \frac{1}{2}\widetilde{x}^4 + \frac{7}{20}\widetilde{x}^5 - \dots \\
F_4 &= 3[\frac{1}{8}(\widetilde{r}^4 - 1) + \frac{1}{2}(\widetilde{r}^2 - 1) - (\widetilde{r}^2 + \frac{1}{2})\ln \widetilde{r}] = \widetilde{x}^4 - \frac{2}{5}\widetilde{x}^5 + \frac{3}{10}\widetilde{x}^6 - \dots \\
&\text{Etc...}
\end{aligned} \tag{12.25}$$

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

For magnetic fields, the "real" ϕ_n^r solutions will correspond to skew fields and the "imaginary" ϕ_n^i solutions will correspond to normal fields

$$\mathbf{B} = -\frac{P_0}{q\,L}\sum_{n=0}^{\infty} \rho^n \left[a_n\,\widetilde{\nabla}\phi_n^r + b_n\,\widetilde{\nabla}\phi_n^i\right] \tag{12.26}$$

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

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

$$\mathbf{E} = -\sum_{n=0}^{\infty} \rho^n \left[a_{en}\,\widetilde{\nabla}\phi_n^i + b_{en}\,\widetilde{\nabla}\phi_n^r\right] \tag{12.27}$$

And this will converge to Eq. (12.18) in the straight line limit.

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

It is possible to construct "horizontally pure" solutions as well. That is, it is possible to construct solutions that in the horizontal plane, with $\widetilde{y} = 0$, behave the same as the corresponding multipole fields with a straight geometry. A straight forward way to do this, for some given multipole of order n, is to construct the horizontally pure solutions,

ψ_n^r and ψ_n^i, as linear superpositions of the vertically pure solutions

$$\psi_n^r = \sum_{k=n}^{\infty} C_{nk}\,\phi_k^r, \qquad \psi_n^i = \sum_{k=n}^{\infty} D_{nk}\,\phi_k^i \tag{12.28}$$

with the normalizations $C_{nn} = D_{nn} = 1$. The C_{nk} and D_{nk} are chosen, order by order, so that ψ_n^r and ψ_n^i are horizontally pure. For the real potentials, the C_{nk}, are obtained from a matrix $\mathbf{M}$ where M_{ij} is the coefficient of the $\widetilde{x}^j$ term of $(dF_i/d\widetilde{x})/i$ when F_i is expressed as an expansion in $\widetilde{x}$ (Eq. (12.25)). C_{nk}, $k = 0, \ldots, \infty$ are the row vectors of the inverse matrix $\mathbf{M}^{-1}$. For the imaginary potentials, the D_{nk} are constructed similarly but in this case the rows of $\mathbf{M}$ are the coefficients in $\widetilde{x}$ for the functions F_i. To convert between field strength coefficients, Eqs. (12.26) and (12.27) and Eqs. (12.28) are combined

$$\begin{aligned} a_n &= \sum_{k=n}^{\infty} \frac{1}{\rho^{k-n}}\, C_{nk}\,\alpha_k, \quad a_{en} = \sum_{k=n}^{\infty} \frac{1}{\rho^{k-n}}\, D_{nk}\,\alpha_{ek}, \\ b_n &= \sum_{k=n}^{\infty} \frac{1}{\rho^{k-n}}\, D_{nk}\,\beta_k, \quad b_{en} = \sum_{k=n}^{\infty} \frac{1}{\rho^{k-n}}\, D_{nk}\,\beta_{ek} \end{aligned} \tag{12.29}$$

where α_k, β_k, α_{ek}, and β_{ek} are the corresponding coefficients for the horizontally pure solutions.

When expressed as a function of $\widetilde{r}$ and $\widetilde{y}$, the vertically pure solutions ϕ_n have a finite number of terms (Eqs. (12.21) and (12.22)). On the other hand, the horizontally pure solutions ψ_n have an infinite number of terms.

The vertically pure solutions form a complete set. That is, any given field that satisfies Maxwell's equations and is independent of z can be expressed as a linear combination of ϕ_n^r and ϕ_n^i. Similarly, the horizontally pure solutions form a complete set. [It is, of course, possible to construct other complete sets in which the basis functions are neither horizontally pure nor vertically pure.]

This brings up an important point. To properly simulate a machine, one must first of all understand whether the multipole values that have been handed to you are for horizontally pure multipoles, vertically, pure multipoles, or perhaps the values do not correspond to either horizontally pure nor vertically pure solutions! Failure to understand this point can lead to differing results. For example, the chromaticity induced by a horizontally pure quadrupole field will be different from the chromaticity of a vertically pure quadrupole field of the same strength. With *Bmad*, the `exact_multipoles` (§3.4) 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[**b: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:

$$\mathbf{B}(x,y,z) = \sum_i \mathbf{B}_i(x,y,z;A,k_x,k_y,k_z,x_0,y_0,\phi_z,family) \tag{12.30}$$

Each term B_i is specified using seven numbers $(A, k_x, k_y, k_z, x_0, y_0, \phi_z)$ and a switch called `family` which can be one of:

```
x,  qu
y,  sq
```

Roughly, taking the offsets x_0 and y_0 to be zero (see the equations below), the `x family` gives a field on-axis where the y component of the field is zero. that is, the `x` family is useful for simulating, say, magnetic vertical bend dipoles. The `y family` has a field that on-axis has no x component. The `qu family` has a magnetic quadrupole like (which for electric fields is skew quadrupole like) field on-axis and the `sq family` has a magnetic skew quadrupole like field on-axis. Additionally, assuming that the x_0 and y_0 offsets are zero, the `sq` family, unlike the other three families, has a nonzero on-axis z field component.

Each family has three possible forms These are designated as "`hyper-y`", "`hyper-xy`", and "`hyper-x`".

For the `x family` the `hyper-y` form is:

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

The `x family hyper-xy` form is:

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

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

$$\begin{aligned} B_x &= \quad A \quad \cosh(k_x(x+x_0)) \cos(k_y(y+y_0)) \cos(k_z z + \phi_z) \\ B_y &= -A \frac{k_y}{k_x} \sinh(k_x(x+x_0)) \sin(k_y(y+y_0)) \cos(k_z z + \phi_z) \\ B_s &= -A \frac{k_z}{k_x} \sinh(k_x(x+x_0)) \cos(k_y(y+y_0)) \sin(k_z z + \phi_z) \\ &\qquad \text{with } k_x^2 = k_y^2 + k_z^2 \end{aligned} \tag{12.33}$$

The relationship between k_x, k_y, and k_z ensures that Maxwell's equations are satisfied. Notice that which form `hyper-y`, `hyper-xy`, and `hyper-x` a particular $\mathbf{B}_i$ belongs to can be computed by *Bmad* by looking at the values of k_x, k_y, and k_z.

Using a compact notation where Ch $\equiv$ cosh, subscript x is $k_x(x + x_0)$, subscript z is

$k_z z + \phi_z$, etc., the `y family` of forms is:

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

(12.34)

the `qu family` of forms is:

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

(12.35)

the `sq family` of forms is:

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

(12.36)

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.[**b:vent.map**]. 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[2]

$$\mathbf{B} = -\nabla \psi_B, \qquad \mathbf{E} = -\nabla \psi_E \tag{12.37}$$

The scalar potentials both satisfy the Laplace equation $\nabla^2 \psi = 0$. The scalar potentials are decomposed as a sum of modes indexed by an integer m

$$\psi_B = \text{Re}\left[\sum_{m=0}^{\infty} \psi_{Bm}\right] \tag{12.38}$$

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

$$\psi_{Bm} = \sum_{n=-N/2}^{N/2-1} \psi_{Bmn} = \sum_{n=-N/2}^{N/2-1} \frac{-1}{k_n} e^{i\,k_n\,z} \cos(m\,\theta - \theta_{0m})\, b_m(n)\, I_m(k_n\,\rho) \tag{12.39}$$

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. (12.39) k_n is given by

$$k_n = \frac{2\pi\, n}{N\, dz} \tag{12.40}$$

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

[2] Notice the negative sign here and in Eq. (12.39) compared to Venturini et al.[**b:vent.map**]. This is to keep the definition of the electric scalar potential ψ_E consistent with the normal definition.

[3] Venturini uses a continuous Fourier transformation but *Bmad* uses a discrete transformation so that only a finite number of coefficients are needed.

$(N-1)\,dz$. Note: Typically the sum in Eq. (12.39) and other equations below runs from 0 to $N-1$. Using a sum from $-N/2$ to $N/2-1$ gives exactly the same field at the sample points $(z = 0, dz, 2\,ds, \ldots)$ and has the virtue that the field is smoother in between.

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

$$\begin{aligned} B_\rho &= \mathrm{Re}\left[\sum_{n=-N/2}^{N/2-1} e^{i\,k_n\,z}\,b_0(n)\,I_1(k_n\,\rho)\right] \\ B_\theta &= 0 \\ B_z &= \mathrm{Re}\left[\sum_{n=-N/2}^{N/2-1} i\,e^{i\,k_n\,z}\,b_0(n)\,I_0(k_n\,\rho)\right] \end{aligned} \tag{12.41}$$

And for $m \neq 0$:

$$\begin{aligned} B_\rho &= \mathrm{Re}\left[\sum_{n=-N/2}^{N/2-1} \frac{1}{2}\,e^{i\,k_n\,z}\,\cos(m\,\theta-\theta_{0m})\,b_m(n)\left[I_{m-1}(k_n\,\rho)+I_{m+1}(k_n\,\rho)\right]\right] \\ B_\theta &= \mathrm{Re}\left[\sum_{n=-N/2}^{N/2-1} \frac{-1}{2}\,e^{i\,k_n\,z}\,\sin(m\,\theta-\theta_{0m})\,b_m(n)\left[I_{m-1}(k_n\,\rho)-I_{m+1}(k_n\,\rho)\right]\right] \\ B_z &= \mathrm{Re}\left[\sum_{n=-N/2}^{N/2-1} i\,e^{i\,k_n\,z}\,\cos(m\,\theta-\theta_{0m})\,b_m(n)\,I_m(k_n\,\rho)\right] \end{aligned} \tag{12.42}$$

While technically ψ_{Bm0} is not well defined due to the $1/k_n$ factor that is present, the field itself is well behaved. Mathematically, Eq. (12.39) can be corrected if, for $n = 0$, the term $I_m(k_n\,\rho)/k_n$ is replaced by

$$\frac{I_m(k_0\,\rho)}{k_0} \rightarrow \begin{cases} \rho & \text{if } m = 0 \\ \rho/2 & \text{if } m = 1 \\ 0 & \text{otherwise} \end{cases} \tag{12.43}$$

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

$$\begin{aligned} A_\rho &= 0 \\ A_\theta &= \mathrm{Re}\left[\sum_{n=-N/2}^{N/2-1} \frac{i}{k_n}\,e^{i\,k_n\,z}\,b_0(n)\,I_1(k_n\,\rho)\right] \\ A_z &= 0 \end{aligned} \tag{12.44}$$

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

$$\begin{aligned} A_\rho &= \text{Re}\left[\sum_{n=-N/2}^{N/2-1} \frac{-i\,\rho}{2\,m}\, e^{i\,k_n\,z}\, \cos(m\,\theta - \theta_{0m})\, b_m(n)\, \Big[I_{m-1}(k_n\,\rho) - I_{m+1}(k_n\,\rho)\Big]\right] \\ A_\theta &= 0 \\ A_z &= \text{Re}\left[\sum_{n=-N/2}^{N/2-1} \frac{-i\,\rho}{m}\, e^{i\,k_n\,z}\, \cos(m\,\theta - \theta_{0m})\, b_m(n)\, I_m(k_n\,\rho)\right] \end{aligned} \tag{12.45}$$

Note: The description of the field using ‘‘`generalized gradients`’’[**b:newton**] is similar to the above equations. The difference is that, with the generalized gradient formalism, terms in θ and ρ are expanded in a Taylor series in x and y.

12.6.2 AC Cylindrical Map Decomposition

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

$$\mathbf{E}(\mathbf{r}) = \sum_{j=1}^{M} \mathbf{E}_j(\mathbf{r})\, \exp[-2\pi i\,(f_j\,t + \phi_{0j})] \tag{12.46}$$

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

$$\nabla^2 \mathbf{E}_j + k_{tj}^2\, \mathbf{E}_j = 0 \tag{12.47}$$

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

$$\begin{aligned} E_\rho(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} -e^{i\,k_n\,z}\, i\,k_n\, e_0(n)\, \widetilde{I}_1(\kappa_n, \rho) \\ E_\theta(\mathbf{r}) &= 0 \\ E_z(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} e^{i\,k_n\,z}\, e_0(n)\, \widetilde{I}_0(\kappa_n, \rho) \end{aligned} \tag{12.48}$$

where $\widetilde{I}_m$ is

$$\widetilde{I}_m(\kappa_n, \rho) \equiv \frac{I_m(\kappa_n\,\rho)}{\kappa_n^m} \tag{12.49}$$

with I_m being a modified Bessel function first kind, and κ_n is given by

$$\kappa_n = \sqrt{k_n^2 - k_t^2} = \begin{cases} \sqrt{k_n^2 - k_t^2} & |k_n| > k_t \\ -i\sqrt{k_t^2 - k_n^2} & k_t > |k_n| \end{cases} \tag{12.50}$$

with

$$k_n = \frac{2\pi\, n}{N\, dz} \tag{12.51}$$

N is the number of points where E_{zc} is evaluated, and dz is the distance between points. The length of the field region is $(N-1)\, dz$. When κ_n is imaginary, $I_m(\kappa_n\, \rho)$ can be evaluated through the relation

$$I_m(-i\, x) = i^{-m}\, J_m(x) \tag{12.52}$$

where J_m is a Bessel function of the first kind. The e_0 coefficients can be obtained given knowledge of the field at some radius R via

$$e_0(n) = \frac{1}{\widetilde{I}_0(\kappa_n, R)}\, \frac{1}{N} \sum_{p=0}^{N-1} e^{-2\pi i n p/N}\, E_z(R, p\, dz) \tag{12.53}$$

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

$$\begin{aligned} E_\rho(\mathbf{r}) &= E_z(\mathbf{r}) = 0 \\ E_\theta(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} e^{ik_n z}\, b_0(n)\, \widetilde{I}_1(\kappa_n, \rho) \end{aligned} \tag{12.54}$$

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

$$b_0(n) = \frac{1}{\widetilde{I}_1(\kappa_n, R)}\, \frac{1}{N} \sum_{p=0}^{N-1} e^{-2\pi i\, n\, p/N}\, E_\theta(R, p\, dz) \tag{12.55}$$

For positive m, the electric field is in the form

$$\begin{aligned} E_\rho(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} -i\, e^{i\, k_n\, z} \left[k_n\, e_m(n)\, \widetilde{I}_{m+1}(\kappa_n, \rho) + b_m(n)\, \frac{\widetilde{I}_m(\kappa_n, \rho)}{\rho} \right] \cos(m\, \theta - \theta_{0m}) \\ E_\theta(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} -i\, e^{i\, k_n\, z} \left[k_n\, e_m(n)\, \widetilde{I}_{m+1}(\kappa_n, \rho) + \right. \\ &\qquad \left. b_m(n) \left(\frac{\widetilde{I}_m(\kappa_n, \rho)}{\rho} - \frac{1}{m}\, \widetilde{I}_{m-1}(\kappa_n, \rho) \right) \right] \sin(m\, \theta - \theta_{0m}) \\ E_z(\mathbf{r}) &= \sum_{n=-N/2}^{N/2-1} e^{i\, k_n\, z}\, e_m(n)\, \widetilde{I}_m(\kappa_n, \rho) \cos(m\, \theta - \theta_{0m}) \end{aligned} \tag{12.56}$$

The `e_m` and `b_m` coefficients can be obtained given knowledge of the field at some radius R via

$$\begin{aligned} e_m(n) &= \frac{1}{\widetilde{I}_m(\kappa_n, R)} \frac{1}{N} \sum_{p=0}^{N-1} e^{-2\pi i n p/N} E_{zc}(R, p\,dz) \\ b_m(n) &= \frac{R}{\widetilde{I}_m(\kappa_n, R)} \left[\frac{1}{N} \sum_{p=0}^{N-1} i\, e^{-2\pi i n p/N} E_{\rho c}(R, p\,dz) - k_n\, e_m(n)\, \widetilde{I}_{m+1}(\kappa_n, R) \right] \end{aligned} \tag{12.57}$$

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

$$\begin{aligned} E_\rho(R, \theta, z) &= E_{\rho c}(R, z)\, \cos(m\,\theta - \theta_{0m}) \\ E_\theta(R, \theta, z) &= E_{\theta s}(R, z)\, \sin(m\,\theta - \theta_{0m}) \\ E_z(R, \theta, z) &= E_{zc}(R, z)\, \cos(m\,\theta - \theta_{0m}) \end{aligned} \tag{12.58}$$

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

$$\mathbf{E} = -\partial_t\, \mathbf{A}, \qquad \mathbf{B} = \nabla \times \mathbf{A} \tag{12.59}$$

Using Eq. (12.46), the vector potential can be obtained from the electric field via

$$\mathbf{A}_j = \frac{-i\, \mathbf{E}_j}{2\,\pi\, f_j} \tag{12.60}$$

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

$$\int dx\, \frac{x\, I_1(a\,\sqrt{x^2+y^2})}{\sqrt{x^2+y^2}} = \frac{1}{a}\, I_0(a\,\sqrt{x^2+y^2}) \tag{12.61}$$

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`[**b:gen.grad**]. 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[4]

$$\mathbf{B} = -\nabla\, \psi_B, \qquad \mathbf{E} = -\nabla\, \psi_E \tag{12.62}$$

[4]Notice the negative sign here and in Eq. (12.64) compared to Venturini et al.[**b:gen.grad**]. This is to keep the definition of the electric scalar potential ψ_E consistent with the normal definition.

The scalar potential is then decomposed into azimuthal components

$$\psi = \sum_{m=1}^{\infty} \psi_{m,s} \sin(m\theta) + \sum_{m=0}^{\infty} \psi_{m,c} \cos(m\theta) \tag{12.63}$$

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.

$$\psi_{m,\alpha} = \sum_{n=0}^{\infty} \frac{(-1)^{n+1} m!}{4^n\, n!\, (n+m)!}\, \rho^{2n+m}\, C_{m,\alpha}^{[2n]}(z) \tag{12.64}$$

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

From Eq. (12.64) the field is given by

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

$$B_\theta = \sum_{m=1}^{\infty}\sum_{n=0}^{\infty} \frac{(-1)^n\, m!\, m}{4^n\, n!\, (n+m)!} \rho^{2n+m-1} \left[C_{m,s}^{[2n]}(z)\, \cos m\theta - C_{m,c}^{[2n]}(z)\, \sin m\theta\right] \tag{12.65}$$

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

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:

$$(B_x, B_y, B_z) = (C_{1,c}, C_{1,s}, -C_{0,c}^{[1]}) \tag{12.66}$$

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

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

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

$$\begin{aligned} A_\rho &= \sum_{m=1}^{\infty}\sum_{n=0}^{\infty} \frac{(-1)^n\,(m-1)!}{4^n\,n!\,(n+m)!}\rho^{2n+m+1}\,\left[C_{m,s}^{[2n+1]}\cos(m\theta) - C_{m,c}^{[2n+1]}\,\sin(m\theta)\right] \\ A_\theta &= 0 \\ A_z &= \sum_{m=1}^{\infty}\sum_{n=0}^{\infty} \frac{(-1)^n\,(m-1)!\,(2n+m)}{4^n\,n!\,(n+m)!}\rho^{2n+m}\,\left[-C_{m,s}^{[2n]}\cos(m\theta) + C_{m,c}^{[2n]}\,\sin(m\theta)\right] \end{aligned} \tag{12.68}$$

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

$$\begin{aligned} (B_x, B_y) &= (\cos\theta\,B_\rho - \sin\theta\,B_\theta,\ \sin\theta\,B_\rho + \cos\theta\,B_\theta) \\ &= \frac{(-1)^n\,m!}{4^n n!\,(n+m)!}\,C_{m,s}^{[2n]}\,\Big[(n+m)\,(x^2+y^2)^n\ (S_{xy}(m-1),\,C_{xy}(m-1)) + \\ &\qquad\qquad n\,(x^2+y^2)^{n-1}\ (S_{xy}(m+1),\,-C_{xy}(m+1))\Big] \end{aligned} \tag{12.69}$$

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

$$B_z = \frac{(-1)^n\,m!}{4^n n!\,(n+m)!}(x^2+y^2)^n\,C_{m,s}^{[2n+1]}(z)\,S_{xy}(m) \tag{12.70}$$

where the last term in Eq. (12.69) is only present for $n > 0$.

$$S_{xy}(m) \equiv \rho^m \sin m\theta = \sum_{r=0}^{2r \le m-1} (-1)^r \binom{m}{2r+1} x^{m-2r-1} y^{2r+1}$$
$$C_{xy}(m) \equiv \rho^m \cos m\theta = \sum_{r=0}^{2r \le m} (-1)^r \binom{m}{2r} x^{m-2r} y^{2r} \tag{12.71}$$

The conversion for the cosine like derivatives is:

$$(B_x, B_y) = \frac{(-1)^n m!}{4^n n!\,(n+m)!} C_{m,c}^{[2n]} \Big[(n+m)\,(x^2+y^2)^n\ (C_{xy}(m-1),\ -S_{xy}(m-1)) + n\,(x^2+y^2)^{n-1}\ (C_{xy}(m+1),\ S_{xy}(m+1)) \Big] \tag{12.72}$$
$$B_z = \frac{(-1)^n m!}{4^n n!\,(n+m)!} (x^2+y^2)^n\, C_{m,c}^{[2n+1]}(z)\, C_{xy}(m)$$

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`.[5] 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

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

where G is the accelerating gradient, $k = \omega/c$ is the wave number with ω 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

[5] Notice that the equations here are only relavent 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 §??.

models the fields as a pillbox with the transverse wall at infinity as detailed in Chapter 3, Section VI of reference [**b:lee**]

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

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_{010}" mode is used that has the correct symmetry:

$$\begin{aligned} E_s(r,\phi,s,t) &= 2\,G\,\sin(k\,s)\,\sin(\omega\,t+2\,\pi\,\phi) \\ E_r(r,\phi,s,t) &= -r\,G\,k\,\cos(k\,s)\,\sin(\omega\,t+2\,\pi\,\phi) \\ B_\phi(r,\phi,s,t) &= \frac{r}{c}\,G\,k\,\sin(k\,s)\,\cos(\omega\,t+2\,\pi\,\phi) \end{aligned} \tag{12.75}$$

Part III

Developer's Guide

Chapter 13

Defining New Lattice Elements

Part IV

Bibliography

Chapter 14

Bibliography

[1] Robert B. Appleby et al. "Merlin++, a flexible and feature-rich accelerator physics and particle tracking library". In: *Comput. Phys. Commun.* 271 (2022), p. 108204. DOI: 10.1016/j.cpc.2021.108204. arXiv: 2011.04335 [physics.acc-ph].

[2] M. Borland. "elegant: A Flexible SDDS-Compliant Code for Accelerator Simulation". In: *6th International Computational Accelerator Physics Conference (ICAP 2000)*. 2000. DOI: 10.2172/761286.

[3] H. Grote et al. "The MAD program". In: *Proceedings of the 1989 IEEE Particle Accelerator Conference, . 'Accelerator Science and Technology*. 1989, 1292–1294 vol.2. DOI: 10.1109/PAC.1989.73426.

[4] G. Iadarola et al. "Xsuite: an integrated beam physics simulation framework". In: (Sept. 2023). arXiv: 2310.00317 [physics.acc-ph].

[5] D. Sagan. "Bmad: A relativistic charged particle simulation library". In: *Nucl. Instrum. Meth.* A558.1 (2006). Proceedings of the 8th International Computational Accelerator Physics Conference, pp. 356–359. ISSN: 0168-9002. DOI: https://doi.org/10.1016/j.nima.2005.11.001.