Revision: March 24, 2024

The Accelerator Lattice.jl Reference Manual

David Sagan

Contents

List of Figures

List of Tables

Part I

Lattice Construction and Manipulation

Chapter 1

Introduction, Overview, and Concepts

This chapter is an overview of, and an introduction to, the *AcceleratorLattice.jl* package which is part of the greater *Bmad-Julia* ecosystem of toolkits and programs for accelerator simulations. With *AcceleratorLattice.jl*, lattices, which can be used to describe such things as LINACs, storage rings, X-ray beam lines, can be constructed and manipulated. Tracking and lattice analysis (for example, calculating closed orbits and Twiss functions) is left to other packages in the *Bmad-Julia* ecosystem.

The *Julia* language itself is used as the basis for constructing lattices. Other simulation programs have similarly utilized the underlying programming language for constructing lattices[1, 4], but this is in marked contrast to such programs as MAD[3], Elegant[2], and *Bmad* [5].

1.1 Acknowledgements

It is my pleasure to express appreciation to people who have contributed to this effort, and without whom, *Bmad-Julia* would only be a shadow of what it is today:

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

1.2 Lattice Elements

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

Lattice elements are structs that inherit from the abstract type `Lat`.

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

In a lattice `branch` (§1.3), The ordered array of elements are assigned an `element index` starting from one. The first element is called `beginning_ele` (§**??**). This element is always included in every `branch` §1.3 and is used as a marker for the beginning of the `branch`. Additionally, every branch will have a final marker element (§??) named `end_ele`.

1.3 Lattice Branches

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

Chapter §?? shows how a `branch` can be defined using `lines`.

Branches can be interconnected using `fork` elements (§??). This is used to simulate forking beam lines such as a connections to a transfer line, dump line, or an X-ray beam line. A `branch` from which other `branches` fork but is not forked to by any other `branch` is called a `root` branch. A branch that is forked to by some other branch is called a `downstream` branch.

There are two types of `branches`: `LordBranches` and `TrackingBranches`, Branches whose `Branch.type` are set to `LordBranch` hol

1.4 Lattice

A `lattice` (§1.4), has an array of `branches`. Each `branch` in this array has a name an is assigned an index starting from one. Additionally, each `branch` is assigned a name which is the `line` that defines the branch (§??).

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

Branches can be interconnected using `fork` and `photon_fork` elements (§??). 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. The `use` statement (§??) in a lattice file will list the `root branches` of a lattice. To connect together lattice elements that are physically shared between branches, for example, the interaction region in colliding beam machines, `multipass` lines (§??) can be used.

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

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

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

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

Chapter 2

Constructing Lattices

2.1 Defining a Lattice Element

Chapter §?? gives a list of lattice elements defined by *AcceleratorLattice.jl*. Lattice elements are instantiated from structs which inherit from the abstract type `Lat`.

Elements are defined using the `@ele` macro. The general syntax 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, K1 = 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, a single `@eles` macro can be used instead of multiple `@ele` macros. Example:

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

2.1.1 Anatomy of an Element

The structs for all elements types contain exactly one component which is a Dict called `pdict` (short for "parameter dict"). With

To copy an element use the `deepcopy` constructor.

2.2 Defining a Lattice Element Type

All lattice element types like `Quadrupole`, `Marker`, etc. are subtypes of the abstract type `Ele`. To construct a new type, use the `construct_ele_type` macro. Example:

```
@construct_ele_type MyEleType
```

2.3 Lattice Element Internals

All element types have a single component called `pdict` ("parameter dict") which is of type `Dict{Symbol,Any}`. Using a `Dict` has advantages and disadvantages. The advantage is that an element is not restricted as to what can be stored in it. The disadvantage is that it is not type stable (§**??**). This is generally acceptable when lattices are constructed but is undesirable during tracking. To regain type stability during tracking, element parameters are put into immutable structs called `element parameter` groups and these structs are stored in `pdict`. During tracking, the tracking code can access element parameters via the struct which makes the code type stable as will be illustrated below.

The `element parameter` group structures are all subtypes of the abstract type `EleParameterGroup`. For example, the `LengthGroup` holds the length and s-positions of the element:

```
@kwdef struct LengthGroup <: EleParameterGroup
  L::Float64 = 0
  s::Float64 = 0
  s_downstream::Float64 = 0
end
```

The `kwdef` macro automatically defines a keyword-based constructor for `LengthGroup`. When a parameter group is stored in an element's `pdict`, the key will be the symbol associated with the struct which in this case is `:LengthGroup`. For example, an element's length can be accessed via `ele.pdict[:LengthGroup].L`.

Chapter 3

Lattice Elements

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

Element	*Section*	*Element*	*Section*
BeamBeam	??	Marker	??
BeginningEle	??	Mask	??
Bend	??	Multipole	??
Collimator	??	NullEle	??
Converter	??	Octupole	??
CrabCavity	??	Patch	??
Custom	??	Pipe	??
Drift	??	Quadrupole	??
EGun	??	RFbend	??
ElSeparator	??	RFcavity	??
EMField	??	SadMult	??
Fiducial	??	Sextupole	??
FloorShift	??	Solenoid	??
Foil	??	Taylor	??
Fork	??	ThickMultipole	??
Instrument	??	Undulator	??
Kicker	??	UnionEle	??
Lcavity	??	Wiggler	??

Table 3.1: Table of element types suitable for use with charged particles. Also see Table 3.3

The list of element types known to *Bmad* is shown in Table 3.1, 3.2, and 3.3. Table 3.1 lists the elements suitable for use with charged particles, Table 3.2 which lists the elements suitable for use with photons, and finally Table 3.3 lists the `controller` element types that can be used for parameter control of other elements. Note that some element types are suitable for both particle and photon use.

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

Element	*Section*	*Element*	*Section*
Beginning_Ele	??	Lens	??
Capillary	??	Marker	??
Crystal	??	Mask	??
Custom	??	Match	??
Detector	??	Monitor	??
Diffraction_Plate	??	Mirror	??
Drift	??	Multilayer_Mirror	??
Ecollimator	??	Patch	??
Fiducial	??	Photon_Fork	??
Floor_Shift	??	Photon_Init	??
Fork	??	Pipe	??
GKicker	??	Rcollimator	??
Instrument	??	Sample	??

Table 3.2: Table of element types suitable for use with photons. Also see Table 3.3

Element	*Section*	*Element*	*Section*
Controller	??	Ramper	??
Girder	??		

Table 3.3: Table of controller elements.

3.1 Lattice Element Parameters

Before discussing lattice elements themselves, the element parameters need to be discussed first. Element parameters are divided into immutable struct groups which inherit from the abstract type `EleParameterGroup`. A list of parameter groups can be seen using the command

For example, the position of the element with respect

Element parameters are listed in

3.2 Anatomy of a Lattice Element

All lattice elements inherit from the abstract type `Ele`. There is a macro `construct_ele_type` that is used to construct a new type of element. For example:

```
@construct_ele_type Bend
```

this defines the immutable `Bend` struct which inherits from `Ele`.

All element structs have a single `Dict{Symbol,Any}` field called `param`. The dot selection operator has been overloaded so that something like `ele.name` is mapped to `ele.param[:name]`. Except!

Chapter 4

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 ele_param_groups

`Custom Lattice Elements`

* Need to extend: ele_param_groups

Chapter 5

Design Decisions

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

First a clarification. The name *Bmad* can be used in two senses. There is *Bmad* the 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, multiple branches could not be instantiated. 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 and 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.

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 makes 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.

Part II

Conventions and Physics

Part III

Bibliography

Chapter 6

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.