Revision: July 18, 2023

The Bmad Reference Manual

David Sagan

Contents

List of Figures

List of Tables

Part I

Overview

Chapter 1

Overview and Introduction

1.1 acknowledgements

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

1.2 What is Bmad?

The original Bmad was developed as a subroutine library ("toolkit") for charged–particle and X-Ray simulations in accelerators and storage rings and has served as the calculational engine for many accelerator simulation programs including the *Tao* program which is widely used in the accelerator community. *Bmad* has been developed at the Cornell Laboratory for Accelerator-based ScienceS and Education (CLASSE) and has been in use since 1996. Eventually, in 2023, the organic growth of Bmad — leading to the code not being as well structured as it should be — pushed the decision that the code needed to be refactored. On top of this was the realization that, while the modern Fortran object-orientated language that *Bmad* was written in was a reasonable choice from a purely technical standpoint, the Fortran community had atrophied to the point where Fortran compiler maintenance was severely affected. The choice was made to use the *Julia* language for the rewrite.

The name "*Bmad* " thus has several meanings. Originally, the term only referred to the *Bmad* toolkit. As time went on, and *Bmad* was used in more and more programs, the term *Bmad* was applied to mean

the whole ecosystem of toolkit plus programs. Finally, there is refactored *Bmad*. This *Bmad* is a *Julia* package and as such has is more akin to *Bmad*-the-ecosystem as opposed to *Bmad*-the-toolkit in that *Bmad*-the-package is used both for constructing lattices (like *Bmad*-the-toolkit) and for simulation work without an intermediate program interfacing in between.

1.3 History

Bmad (Otherwise known as “Baby MAD" or “Better MAD" or just plain “Be MAD!") is a subroutine library for charged–particle and X-Ray simulations in accelerators and storage rings. *Bmad* has been developed at the Cornell Laboratory for Accelerator-based ScienceS and Education (CLASSE) and has been in use since 1996.

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

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

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

1.4 Why Julia?

The choice of *Julia* as the basis for the new *Bmad* was not an easy one. Other possibilities included C++ [], Python [], a combination of Python and C/C++, etc. If the *Bmad* refactoring project had been started before 2023 the choice probably would have been Python/C/C++. But in 2023 *Julia* development was mature enough, and the advantages of *Julia* over the alternatives was large enough, so that the decision was made to use *Julia*.

* Short history of *Julia*

* *Julia* was constructed for simulations/large data handling.

* Very active community (not Fortran)

But what is the compelling reason for using *Julia*? First of all, *Julia* is a scripting language which means that it is like Python.

1.5 Manual Organization

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

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

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

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

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

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

`David Sagan <dcs16@cornell.edu>`

The *Bmad* manual is organized as reference guide and so does not do a good job of instructing the beginner as to how to use *Bmad*. For that there is an introduction and tutorial on *Bmad* and *Tao* (§2.2) concepts that can be downloaded from the *Bmad* web page. Go to either the *Bmad* or *Tao* manual pages and there will be a link for the tutorial.

Chapter 2

Orientation

2.1 What is Bmad?

Bmad is an open-source software library (aka toolkit) for simulating charged particles and X-rays. *Bmad* is not a program itself but is used by programs for doing calculations. The advantage of *Bmad* over a stand-alone simulation program is that when new types of simulations need to be developed, *Bmad* can be used to cut down on the time needed to develop such programs with the added benefit that the number of programming errors will be reduced.

Over the years, *Bmad* has been used for a wide range of charged-particle and X-ray simulations. This includes:

`Lattice design`
`Spin tracking`
`Beam breakup (BBU) simulations in ERLs`
`Intra-beam scattering (IBS) simulations`
`Coherent Synchrotron Radiation (CSR)`
`X-ray simulations`
`Wakefields and HOMs`
`Touschek Simulations`
`Dark current tracking`
`Frequency map analysis`

2.2 Tao and Bmad Distributions

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

Besides the *Tao* program, an ecosystem of *Bmad* based programs has been developed. These programs, along with *Bmad*, are bundled together in what is called a *Bmad* `Distribution` which can be downloaded from the web. The following is a list of some of the more commonly used programs.

bmad_to_mad_sad_elegant
: The `bmad_to_mad_sad_elegant` program converts *Bmad* lattice format files to `MAD8`, `MADX`, `Elegant` and `SAD` format.

bbu
The `bbu` program simulates the beam breakup instability in Energy Recovery Linacs (ERLs).

dynamic_aperture
The `dynamic_aperture` program finds the dynamic aperture through tracking.

ibs_linac
The `ibs_linac` program simulates the effect of intra-beam scattering (ibs) for beams in a Linac.

ibs_ring
The `ibs_linac` program simulates the effect of intra-beam scattering (ibs) for beams in a ring.

long_term_tracking
The `long_term_tracking_program` is for long term tracking of a particle or beam possibly including tracking of the spin.

lux
The `lux` program simulates X-ray beams from generation through to experimental end stations.

mad8_to_bmad.py, madx_to_bmad.py
These python programs will convert `MAD8` and `MADX` lattice files to to *Bmad* format.

moga
The `moga` (multiobjective genetic algorithms) program does multiobjective optimization.

synrad
The `synrad` program computes the power deposited on the inside of a vacuum chamber wall due to synchrotron radiation from a particle beam. The calculation is essentially two dimensional but the vertical emittance is used for calculating power densities along the centerline. Crotch geometries can be handled as well as off axis beam orbits.

synrad3d
The `synrad3d` program tracks, in three dimensions, photons generated from a beam within the vacuum chamber. Reflections at the chamber wall is included.

tao
Tao is a general purpose simulation program.

2.3 Resources: More Documentation, Obtaining Bmad, etc.

More information and download instructions are readily available at the *Bmad* web site:

`www.classe.cornell.edu/bmad/`

Links to the most up-to-date *Bmad* and *Tao* manuals can be found there as well as manuals for other programs and instructions for downloading and setup.

The *Bmad* manual is organized as reference guide and so does not do a good job of instructing the beginner as to how to use *Bmad*. For that there is an introduction and tutorial on *Bmad* and *Tao* (§2.2) concepts that can be downloaded from the *Bmad* web page. Go to either the *Bmad* or *Tao* manual pages and there will be a link for the tutorial.

2.4 PTC: Polymorphic Tracking Code

The PTC/FPP library of Étienne Forest handles Taylor maps to any arbitrary order. This is also known as Truncated Power Series Algebra (TPSA). The core Differential Algebra (DA) package used by PTC/FPP was developed by Martin Berz[**?**]. The PTC/FPP libraries are interfaced to *Bmad* so that calculations that involve both *Bmad* and PTC/FPP can be done in a fairly seamless manner.

Basically, the FPP ("Fully Polymorphic Package") part of the PTC/FPP code handles Taylor map manipulation. This is purely mathematical. FPP has no knowledge of accelerators, magnetic fields, particle tracking etc. PTC ("Polymorphic Tracking Code") implements the physics and uses FPP to handle the Taylor map manipulation. Since the distinction between `FPP` and `PTC` is irrelevant to the non-programmer, "PTC" will be used to refer to the entire PTC/FPP package.

PTC is used by *Bmad* when constructing Taylor maps and when the `tracking_method` §**??**) is set to `symp_lie_ptc`. All Taylor maps above first order are calculated via PTC. No exceptions.

For more discussion of PTC see Chapter §**??**. For the programmer, also see Chapter §**??**.

For the purposes of this manual, PTC and FPP are generally considered one package and the combined PTC/FPP library will be referred to as simply "PTC".

Chapter 3

Bmad Concepts and Organization

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

3.1 Lattice Elements

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

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

In a lattice `branch` (§3.2), The ordered array of elements are assigned a number (the element index) starting from zero. The zeroth `beginning_ele` (§**??**) element, which is always named `BEGINNING`, is automatically included in every branch and is used as a marker for the beginning of the branch. Additionally, every branch will, by default, have a final marker element (§**??**) named `END`.

3.2 Lattice Branches

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

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

A `lattice` (§3.3), has an array of `branches`. Each `branch` in this array is assigned an index starting from 0. Additionally, each `branch` is assigned a name which is the `line` that defines the branch (§**??**).

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. 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.

3.3 Lattice

an array of `branches` that can be interconnected together to describe an entire machine complex. A `lattice` can include such things as transfer lines, dump lines, x-ray beam lines, colliding beam storage rings, etc. All of which can be connected together to form a coherent whole. In addition, a lattice may contain `controller elements` (Table **??**) 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 (§**??**).

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

```
branch_name>>element_name
```

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

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

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

3.4 Lord and Slave Elements

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

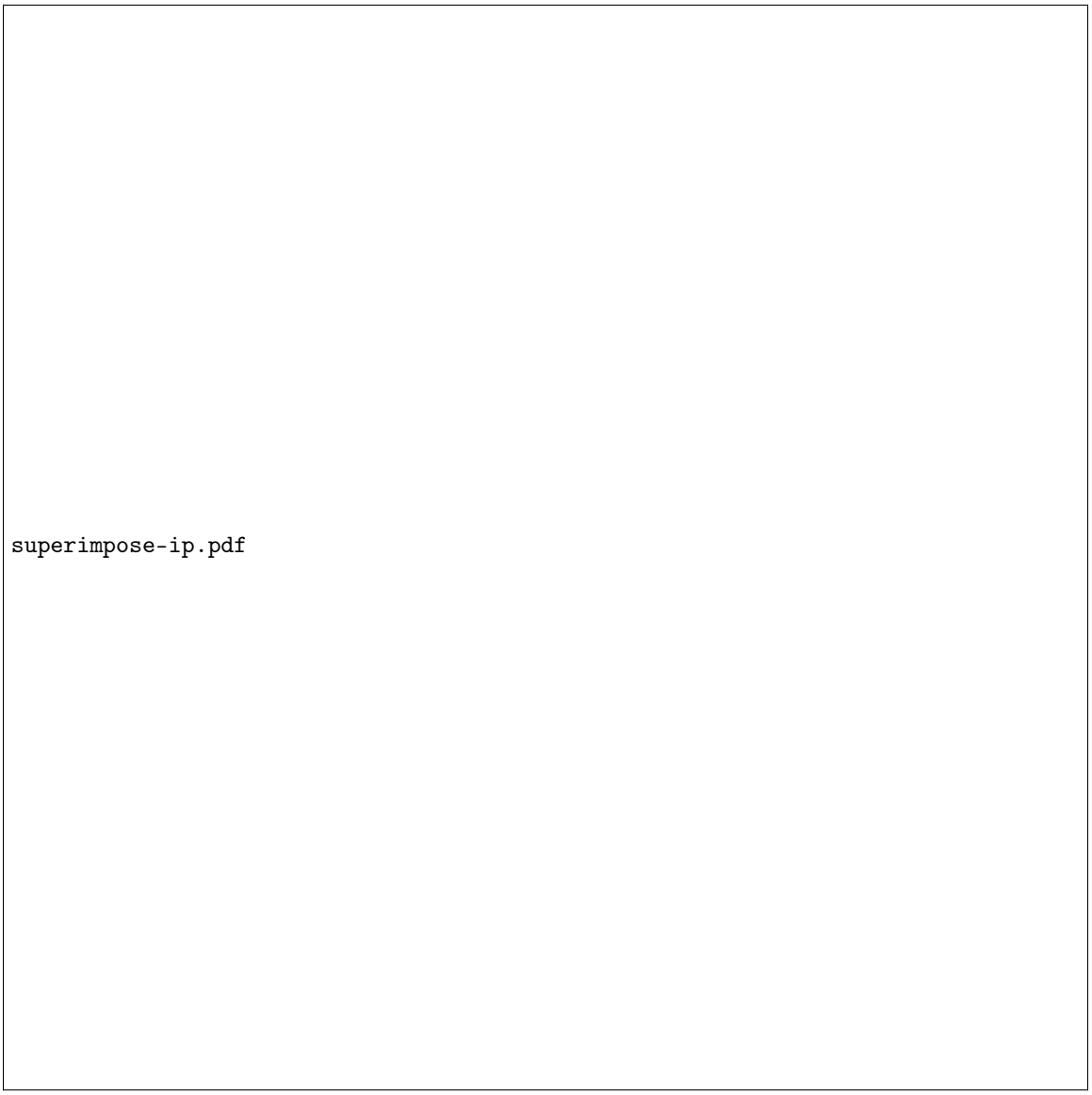

Figure 3.1: Superposition Example. A) Interaction region layout with quadrupoles overlapping a solenoid. B) The Bmad lattice representation has a list of split elements to track through and the undivided "lord" elements. Pointers (double headed arrows), keep track of the correspondence between the lords and their slaves.

Bmad tries to capture these interdependencies using what are referred to as `lord` and `slave` elements. The `lord` elements may be divided into two classes. In one class are the `controller` elements. These are `overlay` (§**??**), `group` (§**??**), `ramper` (§**??**), and `girder` (§**??**) elements that control the attributes of other elements which are their slaves.

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

An example will make this clear. `Superposition` (§??) is the ability to overlap lattice elements spatially. Fig. 3.1 shows an example which is a greatly simplified version of the IR region of Cornell's CESR storage ring when CESR was an e+/e– collider. As shown in Fig. 3.1A, two quadrupoles named `q1w` and `q1e` are partially inside and partially outside the interaction region solenoid named `cleo`. In the lattice file, the IR region layout is defined to be

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

The line named `cesr` ignores the solenoid and just contains the interaction point marker element named `ip` which is surrounded by two drifts named `dft1` which are, in turn, surrounded by the `q1w` and `q1e` quadrupoles. The solenoid is added to the layout on the second line by using superposition. The "ref = ip" indicates that the solenoid is placed relative to `ip`. The default, which is used here, is to place the center of the superimposed `cleo` element at the center of the `ip` reference element. The representation of the lattice in *Bmad* will contain two branch `sections` ("sections" is explained more fully later): One section, called the `tracking section`, contains the elements that are needed for tracking particles. In the current example, as shown in Fig. 3.1B, the first IR element in the tracking section is a quadrupole that represents the part of `q1e` outside of the solenoid. The next element is a combination solenoid/quadrupole, called a `sol_quad`, that represents the part of `q1e` inside `cleo`, etc. The other branch section that Bmad creates is called the `lord section` This section contain the undivided "physical" `super_lord` elements (§??) which, in this case are `q1e`, `q1w`, and `cleo`. Pointers are created between the lords and their `super_slave` elements in the tracking section so that changes in parameters of the lord elements can be transferred to their corresponding slaves.

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

Each lattice element is assigned a `slave_status` indicating what kind of slave it is and a `lord_status` indicating what kind of lord it is. Normally a user does not have to worry about this since these status attributes are handled automatically by *Bmad*. The possible `lord_status` settings are:

girder_lord
: A `girder_lord` element is a `girder` element (§??).

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

overlay_lord
: An `overlay_lord` is an `overlay` element (§??).

group_lord
: A `group_lord` is a `group` element (§??).

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

not_a_lord
: This element does not control anything.

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

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

The possible `slave_status` settings are

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

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

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

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

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

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

As a side note, Étienne Forest's PTC code (§2.4) uses separate structures to separate the physical element, which PTC calls an `element` from the particle track which PTC call a `fibre`. [Actually, PTC has two structures for the physical element, `element` and `elementp`. The latter being the "polymorph" version.] This `element` and `fibre` combination corresponds to *Bmad* `multipass_lord` and `multipass_slave` elements. PTC does not handle overlapping fields as *Bmad* does with `superposition` (§??).

Part II

Lattice Construction Reference

Part III

Simulations With Bmad

Part IV

Conventions and Physics

Part V

Bibliography