

Revision: 12.0  
February 8, 2010

The  
*Bmad*  
Reference Manual

David Sagan

## Overview

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

Prior to the development of *Bmad*, simulation programs at Cornell were written almost from scratch to perform calculations that were beyond the capability of existing, generally available software. This practice was inefficient, leading to much duplication of effort. Since the development of simulation programs was time consuming, needed calculations were not being done. As a response, the *Bmad* subroutine library, using an object oriented approach and written in Fortran90, were developed. The aim of the *Bmad* project was to:

- Cut down on the time needed to develop programs.
- Minimize computation times.
- 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* has a wide range of routines to do many things. *Bmad* can be used to study both single and multi-particle beam dynamics. It has routines to track both particles and macroparticles. *Bmad* has various tracking algorithms including Runge–Kutta and symplectic (Lie algebraic) integration. Wake fields, and radiation excitation and damping can be simulated. *Bmad* has routines for calculating transfer matrices, emittances, Twiss parameters, dispersion, coupling, etc. The elements that *Bmad* knows about include quadrupoles, RF cavities (both storage ring and LINAC accelerating types), solenoids, dipole bends, 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.

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

The strength of *Bmad* is that, as a subroutine library, it provides a flexible framework from which sophisticated simulation programs may easily be developed. The weakness of *Bmad* comes from its strength: *Bmad* cannot be used straight out of the box. Someone must put the pieces together into a program. To partially remedy this problem, the *Tao* program[1] has been developed at Cornell. *Tao*, which uses *Bmad* as its simulation engine, is a general purpose program for simulating high energy 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.

## Introduction

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

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

### Part I

Part I discusses the *Bmad* lattice input standard. The *Bmad* lattice input standard was developed using the *MAD* lattice input standard as a starting point. *MAD* (Methodical Accelerator Design) is a widely used stand-alone program developed at CERN by Christoph Iselin for charged-particle optics calculations. Since it can be convenient to do simulations with both *MAD* and *Bmad*, differences and similarities between the two input formats are noted.

### 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[4]. 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 <dcsl6@cornell.edu>

It is my pleasure to express appreciation to people who have contributed to this effort: To David Rubin for his support, to Etienne Forest for use of his remarkable PTC/FPP library not to mention his patience in explaining everything to me, to Mark Palmer for all his work porting *Bmad* to different platforms, to Hans Grote for granting the adaptation of figures in the *MAD* manual for use in this one, and to Joel Brock, Sarah Buchan, Joseph Choi, Gerry Dugan, Michael Ehrlichman, Mike Forster, Richard Helms, Georg Hoffstaetter, Chris Mayes, Karthik Narayan, David Rubin, Michael Saelim, Jeff Smith, Jeremy Urban, and Mark Woodley for their help.



# Contents

<b>I</b>	<b>Language Reference</b>	<b>15</b>
<b>1</b>	<b>Lattice File Overview</b>	<b>17</b>
1.1	Lattice Files	17
1.2	File Example and Syntax	18
1.3	Digested Files	18
1.4	Element Sequence Definition	19
1.5	Lattice Elements	19
1.6	Element Attributes	20
1.7	Variable Types	20
1.8	Units and Constants	20
1.9	Arithmetic Expressions	21
1.10	Intrinsic functions	22
<b>2</b>	<b>Elements</b>	<b>23</b>
2.1	AB_Multipole	24
2.2	BeamBeam	24
2.3	Bend_Sol_Quad	25
2.4	Bends: Rbend and Sbend	26
2.5	Branch and Photon_Branch	29
2.6	Collimators: Ecollimator and Rcollimator	29
2.7	Crystal	30
2.8	Custom	30
2.9	Drift	30
2.10	Elseparator	31
2.11	Hkicker and Vkicker	31
2.12	Hybrid	31
2.13	Girder	31
2.14	Instrument, Monitor, and Pipe	32
2.15	Kicker	33
2.16	Lcavity	33
2.17	Marker	36
2.18	Match	36
2.19	Mirror	38
2.20	Multipole	38
2.21	Null_Ele	39
2.22	Octupole	39
2.23	Patch	39
2.24	Quadrupole	40
2.25	RFcavity	41

2.26	Sextupole . . . . .	41
2.27	Solenoid . . . . .	42
2.28	Sol_Quad . . . . .	42
2.29	Taylor . . . . .	42
2.30	Wiggler . . . . .	43
<b>3</b>	<b>Overlays, Groups, Superpositions, and Multipass</b>	<b>45</b>
3.1	Overlay Elements . . . . .	45
3.2	Group Elements . . . . .	46
3.3	Superposition . . . . .	47
3.3.1	Changing Element Lengths when there is Superposition . . . . .	49
3.4	Multipass . . . . .	50
3.4.1	The Reference Orbit in a Multipass Line . . . . .	51
3.4.2	Using Patch elements to Vary the Reference Orbit in a Multipass Line . . . . .	53
<b>4</b>	<b>Element Attributes</b>	<b>55</b>
4.1	Dependent and Independent Attributes . . . . .	55
4.2	Type, Alias and Descrip Attributes . . . . .	56
4.3	Beam_Energy and POC Attributes . . . . .	57
4.4	Offset, Pitch, Tilt, and Roll Attributes . . . . .	57
4.5	Hkick, Vkick, and Kick Attributes . . . . .	58
4.6	Aperture and Limit Attributes . . . . .	59
4.7	Length Attributes . . . . .	61
4.8	Is_on Attribute . . . . .	61
4.9	Multipole Attributes: An, Bn, KnL, Th . . . . .	61
4.10	Instrumental Measurement Attributes . . . . .	62
<b>5</b>	<b>Tracking and Transfer Matrix Calculation Methods</b>	<b>63</b>
5.1	tracking_method Switches . . . . .	63
5.2	mat6_calc_method Switches . . . . .	66
5.3	Integration Methods . . . . .	68
5.4	Symplectify Attribute . . . . .	69
5.5	Map_with_offsets Attribute . . . . .	69
<b>6</b>	<b>Beam Lines, Replacement Lists, and Branching</b>	<b>71</b>
6.1	Use Statement . . . . .	71
6.2	Beam Lines without Arguments . . . . .	71
6.3	Beam Lines with Replaceable Arguments . . . . .	72
6.4	Replacement Lists . . . . .	73
6.5	Line and List Tags . . . . .	73
6.6	Branching . . . . .	74
<b>7</b>	<b>Miscellaneous Statements</b>	<b>75</b>
7.1	Parameter Statement . . . . .	75
7.2	Beam_start Statement . . . . .	76
7.3	Beam Statement . . . . .	76
7.4	Beginning Statement . . . . .	77
7.5	Title Statement . . . . .	77
7.6	Call Statement . . . . .	77
7.7	Return and End_File statements . . . . .	78
7.8	Expand_lattice Statement . . . . .	78
7.9	No_superimpose Statement . . . . .	79

7.10	Debugging Statements . . . . .	79
<b>8</b>	<b>Bmad Parameter Structures</b>	<b>81</b>
8.1	Bmad Global Parameters . . . . .	81
8.2	Beam Initialization Parameters . . . . .	82
8.3	CSR Parameters . . . . .	84
<b>II</b>	<b>Conventions and Physics</b>	<b>87</b>
<b>9</b>	<b>Coordinates</b>	<b>89</b>
9.1	Reference Orbit . . . . .	89
9.2	Global Coordinates . . . . .	90
9.2.1	Mirror Element . . . . .	92
9.2.2	Patch Element . . . . .	93
9.3	Phase Space Coordinate System . . . . .	93
<b>10</b>	<b>Physics</b>	<b>97</b>
10.1	Units . . . . .	97
10.2	Magnetic Fields . . . . .	97
10.3	Taylor Maps and Symplectic Integration . . . . .	99
10.4	Symplectification . . . . .	102
10.5	LINAC Accelerating Cavities (Lcavity) . . . . .	103
10.6	Wigglers . . . . .	104
10.7	Synchrotron Radiation Damping and Excitation . . . . .	105
10.8	Coupling and Normal Modes . . . . .	106
10.9	Dispersion Calculation . . . . .	107
10.10	Instrumental Measurements . . . . .	108
10.10.1	Orbit Measurement . . . . .	108
10.10.2	Dispersion Measurement . . . . .	109
10.10.3	Coupling Measurement . . . . .	109
10.10.4	Phase Measurement . . . . .	110
10.11	Bunch Initialization . . . . .	111
10.11.1	Elliptical Phase Space Distribution . . . . .	111
10.11.2	Kapchinsky-Vladimirsky Phase Space Distribution . . . . .	112
10.12	Macroparticles . . . . .	113
10.13	Wake fields . . . . .	114
10.13.1	Short-Range Wakes . . . . .	114
10.13.2	Long-Range Wakes . . . . .	115
10.14	Synchrotron Radiation Integrals . . . . .	117
10.15	Spin Dynamics . . . . .	120
10.16	Coherent Synchrotron Radiation . . . . .	121
<b>III</b>	<b>Programmer's Guide</b>	<b>123</b>
<b>11</b>	<b>Bmad Programming Overview</b>	<b>125</b>
11.1	The Bmad Libraries . . . . .	125
11.2	getf and listf . . . . .	126
11.3	Precision . . . . .	127
11.4	Programming Conventions . . . . .	128
11.5	Manual Notation . . . . .	128

<b>12 Introduction to Bmad programming</b>	<b>131</b>
12.1 A First Program . . . . .	131
12.2 Explanation of the Simple_Program . . . . .	131
<b>13 The Ele_struct</b>	<b>135</b>
13.1 Initialization and Pointers . . . . .	135
13.2 String Components . . . . .	136
13.3 Element Key . . . . .	136
13.4 The %value(:) array . . . . .	138
13.5 Limits . . . . .	138
13.6 Twiss Parameters, etc. . . . .	138
13.7 Element Lords and Element Slaves . . . . .	139
13.8 Coordinates, Offsets, etc. . . . .	139
13.9 Transfer Maps: Linear and Non-linear (Taylor) . . . . .	140
13.10 Wake fields . . . . .	140
13.11 Wiggler Types . . . . .	142
13.12 Multipoles . . . . .	142
13.13 Tracking Methods . . . . .	143
13.14 General Use Components . . . . .	143
13.15 Bmad Reserved Variables . . . . .	143
<b>14 The lat_struct</b>	<b>145</b>
14.1 Pointers . . . . .	145
14.2 Branches in the lat_struct . . . . .	146
14.3 Param_struct Component . . . . .	147
14.4 Elements Controlling Other Elements . . . . .	148
14.5 Lattice Bookkeeping . . . . .	153
14.6 Finding Elements and Changing Attribute Values . . . . .	154
14.7 Beam_start Component . . . . .	155
14.8 Miscellaneous . . . . .	156
<b>15 Reading and Writing Lattices</b>	<b>157</b>
15.1 Reading in Lattices . . . . .	157
15.2 Digested Files . . . . .	157
15.3 Writing Lattice files . . . . .	158
15.4 Accelerator Markup Language . . . . .	158
<b>16 Twiss Parameters, Coupling, Chromaticity, Etc.</b>	<b>161</b>
16.1 Ele_struct Components . . . . .	161
16.2 Twiss Parameter Calculations . . . . .	162
16.3 Tune Calculation . . . . .	163
16.4 Chromaticity Calculation . . . . .	163
<b>17 Tracking and Transfer Maps</b>	<b>165</b>
17.1 The coord_struct . . . . .	165
17.2 Tracking Through the Elements . . . . .	166
17.3 Closed Orbit . . . . .	166
17.4 Apertures . . . . .	167
17.5 Tracking Methods . . . . .	168
17.6 Taylor Maps . . . . .	168
17.7 Reverse Tracking . . . . .	169
17.8 Particle Distribution Tracking . . . . .	169



17.9	Spin Tracking . . . . .	170
17.10	Custom Field Calculations . . . . .	170
<b>18</b>	<b>Miscellaneous Programming</b>	<b>171</b>
18.1	Custom Elements . . . . .	171
18.2	Physical Constants . . . . .	171
18.3	Common Structures . . . . .	171
<b>19</b>	<b>Etienne Forest's PTC/FPP</b>	<b>173</b>
19.1	Phase Space . . . . .	173
19.2	Initialization . . . . .	174
19.3	Taylor Maps . . . . .	174
<b>20</b>	<b>C++ Interface</b>	<b>175</b>
20.1	C++ Classes . . . . .	176
20.2	Fortran calling C++ . . . . .	177
20.3	C++ calling Fortran . . . . .	177
<b>21</b>	<b>Quick_Plot Plotting</b>	<b>179</b>
21.1	An Example . . . . .	179
21.2	Plotting Coordinates . . . . .	182
21.3	Length and Position Units . . . . .	182
21.4	Y2 and X2 axes . . . . .	183
21.5	Text . . . . .	184
21.6	Styles . . . . .	184
21.7	Structures . . . . .	187
<b>22</b>	<b>Helper Routines</b>	<b>189</b>
22.1	Nonlinear Optimization . . . . .	189
22.2	Matrix Manipulation . . . . .	189
<b>23</b>	<b>Bmad Library Subroutine List</b>	<b>191</b>
23.1	Beam: Low Level Routines . . . . .	192
23.2	Beam: Tracking and Manipulation . . . . .	193
23.3	Branch Handling Routines . . . . .	194
23.4	C++ Interface . . . . .	194
23.5	Coherent Synchrotron Radiation (CSR) . . . . .	196
23.6	Collective Effects . . . . .	196
23.7	Electro-Magnetic Fields . . . . .	196
23.8	General Helper Routines . . . . .	197
23.8.1	General Helper: File, System, and IO Routines . . . . .	197
23.8.2	General Helper: Math (Except Matrix) Routines . . . . .	198
23.8.3	General Helper: Matrix Routines . . . . .	199
23.8.4	General Helper: Misc Routines . . . . .	199
23.8.5	General Helper: String Manipulation Routines . . . . .	200
23.9	Inter-Beam Scattering (IBS) . . . . .	201
23.10	Lattice: Informational . . . . .	201
23.11	Lattice: Element Manipulation . . . . .	203
23.12	Lattice: Geometry . . . . .	204
23.13	Lattice: Low Level Stuff . . . . .	204
23.14	Lattice: Manipulation . . . . .	205
23.15	Lattice: Miscellaneous . . . . .	206

23.16	Reading and Writing Lattice Files	207
23.17	Matrices	207
23.18	Matrix: Low Level Routines	209
23.19	Measurement Simulation Routines	209
23.20	Multipass	209
23.21	Multipoles	210
23.22	Nonlinear Optimizers	210
23.23	Overloading the equal sign	211
23.24	Particle Coordinate Stuff	211
23.25	Photon Routines	212
23.26	Interface to PTC	212
23.27	Quick Plot Routines	213
23.27.1	Quick Plot Page Routines	213
23.27.2	Quick Plot Computational Routines	214
23.27.3	Quick Plot Drawing Routines	214
23.27.4	Quick Plot Set Routines	216
23.27.5	Informational Routines	217
23.27.6	Conversion Routines	218
23.27.7	Miscellaneous Routines	218
23.27.8	Low Level Routines	218
23.28	Spin Tracking	220
23.29	Transfer Maps: Routines Called by make_mat6	220
23.30	Transfer Maps: Taylor Maps	221
23.31	Tracking and Closed Orbit	222
23.32	Tracking: Low Level Routines	224
23.33	Tracking: Macroparticle	224
23.34	Tracking: Mad Routines	225
23.35	Tracking: Routines called by TRACK1	226
23.36	Twiss and Other Calculations	227
23.37	Twiss: 6 Dimensional	228
23.38	Wake Fields	228
23.39	Deprecated	229
<b>Bibliography</b>		<b>231</b>
<b>Routine Index</b>		<b>233</b>
<b>Index</b>		<b>239</b>

# List of Figures

2.1	Coordinate systems for (a) <b>Rbend</b> and (b) <b>Sbend</b> elements. . . . .	27
3.1	Superposition Illustration. . . . .	48
3.2	The reference orbit with a multipass bend. . . . .	51
3.3	Using patch elements to vary the reference orbit in a multipass line. . . . .	53
4.1	Geometry of Pitch and Offset attributes . . . . .	58
4.2	Geometry of a Tilt . . . . .	58
4.3	Geometry of a Roll . . . . .	59
4.4	Apertures for <b>ecollimator</b> and <b>rcollimator</b> elements . . . . .	60
9.1	The Local Reference System. . . . .	89
9.2	The Global Coordinate System . . . . .	91
9.3	Reflection by a mirror . . . . .	92
9.4	Interpreting Canonical $z$ at constant velocity. . . . .	93
10.1	CSR Calculation . . . . .	122
12.1	Example Bmad program . . . . .	132
12.2	Output from the example program . . . . .	134
13.1	The <b>ele_struct</b> . Part 1 . . . . .	136
13.2	The <b>ele_struct</b> . part 2. . . . .	137
14.1	Definition of the <b>lat_struct</b> . . . . .	145
14.2	Definition of the <b>param_struct</b> . . . . .	147
14.3	Example of multipass combined with superposition . . . . .	151
17.1	Condensed <b>track_all</b> code. . . . .	167
20.1	Example Fortran routine calling a $C++$ routine. . . . .	176
20.2	Example $C++$ routine callable from a Fortran routine. . . . .	176
20.3	Example $C++$ routine calling a Fortran routine. . . . .	177
20.4	Example Fortran routine callable from a $C++$ routine. . . . .	177
21.1	<i>Quick Plot</i> example program. . . . .	180
21.2	Output of <b>plot_example.f90</b> . . . . .	181
21.3	A Graph within a Box within a Page. . . . .	182



# List of Tables

1.1	Physical and mathematical constants recognized by <i>Bmad</i> . . . . .	21
2.1	Table of element classes suitable for use with relativistic particles. . . . .	23
2.2	Table of element classes suitable for use with photons. . . . .	24
2.3	Table of element classes used for parameter control of other elements. . . . .	24
4.1	Table of dependent variables. . . . .	55
4.2	Dependent variables that can be set in a primary lattice file. . . . .	55
4.3	Field and Strength Attributes. . . . .	56
5.1	Table of available <b>tracking_method</b> switches for a given element class. . . . .	65
5.2	Table of available <b>mat6_calc_method</b> switches for a given element class. . . . .	67
10.1	Physical units used by <i>Bmad</i> . . . . .	97
10.2	$F$ and $n_{\text{ref}}$ for various elements. . . . .	99
14.1	Bounds of the tracking and control parts of the main lattice <b>% branch(0)% ele(:)array</b> .147	
14.2	Possible element <b>%lord_status/%slave_status</b> combinations. . . . .	150
20.1	Bmad structures and their corresponding C++ classes. . . . .	175
21.1	Plotting Symbols at Height = 40.0 . . . . .	185
21.2	PGPLOT Escape Sequences. . . . .	186
21.3	Roman to Greek Character Conversion . . . . .	187



## Part I

# Language Reference





# Chapter 1

## Lattice File Overview

### 1.1 Lattice Files

A lattice file (or files) defines an accelerator: That is, a lattice file defines the sequence of elements that a particle will travel through along with the attributes of the elements (their lengths, their strengths, etc.). The *Bmad* software library comes with routines to read in (parse) lattice files. There are two input formats that *Bmad* understands: The *Bmad* standard format and XSIF[14]. XSIF stands for “Extended Standard Input Format.” XSIF is essentially a subset of the MAD input format. XSIF has its own documentation which can be found at:

```
http://www-project.slac.stanford.edu/  
lc/ilc/TechNotes/LCCNotes/PDF/LCC-0060%20rev.1.pdf
```

Since XSIF does not have a `lattice_type` statement (§7.1), the type of the lattice (whether circular or linear) is determined by the presence or absence of any `LCavity` elements in the XSIF file. This is independent of whether `LCavity` elements are actually used in the lattice.

An XSIF lattice file may be called from within a *Bmad* lattice file. See Section §7.6 for more details.

Since XSIF is separately documented it will not be discussed in detail here and throughout this manual, unless explicitly stated otherwise, the lattice format under discussion will always pertain to the *Bmad* standard format. One feature of the *Bmad* implementation of XSIF Note: One point that is not covered in the XSIF documentation is that for a `MATRIX` element, unlike *MAD*, the `Rii` terms (the diagonal terms of the linear matrix) are not unity by default. Thus

```
m: matrix
```

in an XSIF file will give a matrix with all elements being zero.

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

## 1.2 File Example and Syntax

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

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

A *Bmad* lattice file consists of a sequence of statements. Normally a statement occupies a single line in the file. Several statements may be placed on the same line by inserting a semicolon (;) between them. A long statement can occupy multiple lines by putting an ampersand (&) at the end of each line of a statement except for the last line. An exclamation mark (!) denotes a comment and the exclamation mark and everything after the exclamation mark on a line are ignored. *Bmad* is case insensitive. All names are converted to uppercase except strings (§1.7).

Names of constants, elements, lines, etc. are limited to 40 characters. The first character must be a letter (A — Z). The other characters may be a letter, a digit (0 — 9) or an underscore (\_). Other characters may appear but should be avoided since they are used by *Bmad* for various purposes. For example, the backslash (\) character is used to by *Bmad* when forming the names of superposition slaves (§3.3) and dots (.) are used by *Bmad* when creating names of **tagged** elements (§6.5). Also use of special characters may make the lattice files less portable to non-*Bmad* programs.

| The following example constructs a linear lattice with two elements:

```
parameter[lattice_type] = LINEAR_LATTICE
parameter[e_tot] = 2.7389062E9
parameter[particle] = POSITRON
beginning[beta_a] = 14.5011548
beginning[alpha_a] = -0.53828197
beginning[beta_b] = 31.3178048
beginning[alpha_b] = 0.25761815
q: quadrupole, l = 0.6, b1_gradient = 9.011
d: drift, l = 2.5
t: line = (q, d)
use, t
```

here `parameter[lattice_type]` (§7.1) is set to `LINEAR_LATTICE` which specifies that the lattice is not circular. In this case, the beginning Twiss parameters need to be specified and this is done by the **beginning** statements (§7.4). A quadrupole named **q** and a drift element named **d** are specified and the entire lattice consists of element **q** followed by element **d**.

## 1.3 Digested Files

Normally the *Bmad* parser routine will create what is called a “digested file” after it has parsed a lattice file so that when a program is run and the same lattice file is to be read in again, to save time, the

digested file can be used to load in the lattice information. This digested file is in binary format and is not human readable. The digested file will contain the transfer maps for all the elements. Using a digested file can save considerable time if some of the elements in the lattice need to have Taylor maps computed. (this occurs typically with map-type wigglers).

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

Digested files contain the names of the lattice files used to create them. If a lattice file has been modified since the digested file has been created then the the lattice files will be reread and a new digested file will be generated. Additionally, if any of the random number functions are used in the process of creating the lattice the digested file will be ignored.

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

## 1.4 Element Sequence Definition

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

```
13: line = (l1, l2)    ! Concatenate two lines
11: line = (a, b, c)   ! Line with 3 elements
12: line = (a, z)      ! Another line
use, 13                ! Use 13 as the lattice definition.
```

In this case the lattice would be

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

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

The **superimpose** construct allows elements to be placed in a lattice at a definite longitudinal position. What happens is that after a lattice is expanded, there is a reshuffling of the elements to accommodate any new superimpose elements. See §3.3 for more details.

## 1.5 Lattice Elements

The syntax for defining a lattice element roughly follows *MAD*:

```
label: keyword [, attributes]
```

Overlay and Group elements have a slightly different syntax

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

and Girder elements have the syntax

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

For example:

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

Chapter 2 gives a list of elements with their attributes.

## 1.6 Element Attributes

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

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

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

```
element-name[attribute-name]
```

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

```
b01w[roll] = 6.5           ! Set an attribute value.
b01w[l] = 6.5             ! Change an attribute value.
b01w[l] = b01w[rho] / 12   ! OK to reset an attribute value.
my_const = b01[rho] / b01[l] ! Use of attribute values in an expression.
```

Notice that there can be no space between the element name and the [ opening bracket. When setting an attribute value, if more than one element has the `element-name` then *all* such elements will be set. When setting an attribute value, if `element-name` is the name of a type of element, all elements of that type will be set. Additionally, "\*" can be used to denote all elements. For example

```
rfcavity[voltage] = 3.7           ! Set all Rfcavity elements.
*[tracking_method] = bmad_standard ! Set for all elements
```

Chapter 2 gives a list of elements with their attributes.

## 1.7 Variable Types

There are five types of variables in *Bmad*: Reals, Integers, Switches, Logicals (booleans), and Strings. Acceptable logical values are

```
true   false
t       f
```

For example

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

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

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

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

Switches are variables that take discrete values. For example:

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

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

## 1.8 Units and Constants

*Bmad* uses SI (Système International) units as shown in Table 10.1.

*Bmad* defines commonly used physical and mathematical constants shown in Table 1.1. All symbols use straight SI units except for `e_mass` and `p_mass` which are provided for compatibility with *MAD*.

<i>Symbol</i>	<i>Value</i>	<i>Units</i>	<i>Name</i>
pi	3.14159265359		
twopi	2 * pi		
fourpi	4 * pi		
e_log	2.718281828		
sqrt_2	1.4142135623731		
degrees	180 / pi		
m_electron	$0.51099906 \cdot 10^6$	eV	Electron mass
m_proton	$0.938271998 \cdot 10^9$	eV	Proton mass
c_light	$2.99792458 \cdot 10^8$	m/sec	Speed of light
r_e	$2.8179380 \cdot 10^{-15}$	m	Electron radius
r_p	$1.5346980 \cdot 10^{-18}$	m	Proton radius
e_charge	$1.6021892 \cdot 10^{-19}$	C	Electron charge
h_planck	$4.13566733 \cdot 10^{-15}$	eV*sec	Planck's constant
h_bar_planck	$6.58211899 \cdot 10^{-16}$	eV*sec	Planck / $2\pi$
e_mass	$0.51099906 \cdot 10^{-3}$	GeV	Electron mass
p_mass	0.938271998	GeV	Proton mass

Table 1.1: Physical and mathematical constants recognized by *Bmad*.

## 1.9 Arithmetic Expressions

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

$a + b$  Addition

$a - b$  Subtraction

$a * b$  Multiplication *Bmad* also has a set of intrinsic functions. A list of these is given in §1.10.

$a / b$  Division

$a \wedge b$  Exponentiation

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

1, 10.35, 5E3, 314.159E-2

Symbolic constants can be defined using the syntax

```
parameter_name = expression
```

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

```
parameter_name := expression
```

Examples:

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

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

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

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

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

## 1.10 Intrinsic functions

The following intrinsic functions are recognized by *Bmad*:

<code>sqrt(x)</code>	Square Root
<code>log(x)</code>	Logarithm
<code>exp(x)</code>	Exponential
<code>sin(x)</code>	Sine
<code>cos(x)</code>	Cosine
<code>tan(x)</code>	Tangent
<code>asin(x)</code>	Arc sine
<code>acos(x)</code>	Arc cosine
<code>atan(x)</code>	Arc Tangent
<code>abs(x)</code>	Absolute Value
<code>ran()</code>	Random number between 0 and 1
<code>ran_gauss()</code>	Gaussian distributed random number

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

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

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

If an element is used multiple times in a lattice, and if `ran` or `gauss_ran` is used to set an attribute value of this element, then to have all instances of the element have different attribute values the setting of the attribute must be after the lattice has been expanded (§7.8). For example:

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

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

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

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

# Chapter 2

## Elements

A lattice is made up of a collection of elements — quadrupoles, bends, etc. This chapter discusses the various classes of elements available in *Bmad* except for **groups** and **overlays** which are discussed Chapter 3

Most element classes available in *MAD* are provided in *Bmad*. Additionally, *Bmad* provides a number of element classes that are not available in *MAD*. A word of caution: In some cases where both *MAD* and *Bmad* provide the same element class, there will be an overlap of the attributes available but the two sets of attributes will not be the same. The list of element classes known to *Bmad* is shown in Table 2.1, 2.2, and 2.3. Table 2.1 lists the elements suitable for use with relativistic particles, Table 2.2 which lists the elements suitable for use with photons, and finally Table 2.3 lists the element classes that can be used for parameter control of other elements. Note that some classes are suitable for both particle and photon use.

<i>Element</i>	<i>Section</i>	<i>Element</i>	<i>Section</i>
AB_Multipole	2.1	Null_Ele	2.21
BeamBeam	2.2	Octupole	2.22
Bend_Sol_Quad	2.3	Patch	2.23
Branch	2.5	Photon_Branch	2.5
Custom	2.8	Pipe	2.14
Drift	2.9	Quadrupole	2.24
Ecollimator	2.6	Rbend	2.4
ElSeparator	2.10	Rcollimator	2.6
HKicker	2.11	RFcavity	2.25
Hybrid	2.12	Sbend	2.4
Instrument	2.14	Sextupole	2.26
Kicker	2.15	Solenoid	2.27
Lcavity	2.16	Sol_Quad	2.28
Marker	2.17	Taylor	2.29
Match	2.18	VKicker	2.11
Monitor	2.14	Wiggler	2.30
Multipole	2.20		

Table 2.1: Table of element classes suitable for use with relativistic particles.

<i>Element</i>	<i>Section</i>	<i>Element</i>	<i>Section</i>
Crystal	<a href="#">2.7</a>	Match	<a href="#">2.18</a>
Custom	<a href="#">2.8</a>	Monitor	<a href="#">2.14</a>
Drift	<a href="#">2.9</a>	Mirror	<a href="#">2.19</a>
Instrument	<a href="#">2.14</a>	Patch	<a href="#">2.23</a>
Marker	<a href="#">2.17</a>	Pipe	<a href="#">2.14</a>

Table 2.2: Table of element classes suitable for use with photons.

<i>Element</i>	<i>Section</i>	<i>Element</i>	<i>Section</i>
Group	<a href="#">3.2</a>	Overlay	<a href="#">3.1</a>
Girder	<a href="#">2.13</a>		

Table 2.3: Table of element classes used for parameter control of other elements.

## 2.1 AB\_Multipole

<i>Attribute Class</i>	<i>§</i>	<i>Attribute Class</i>	<i>§</i>
<i>an</i> , <i>bn</i> multipoles	<a href="#">4.9</a>	Offsets and tilt	<a href="#">4.4</a>
Description strings	<a href="#">4.2</a>	Is_on	<a href="#">4.8</a>
Reference energy	<a href="#">4.3</a>	Tracking & transfer map	<a href="#">5</a>
Aperture Limits	<a href="#">4.6</a>	Length	<a href="#">4.7</a>

An `AB_Multipole` is a thin multipole lens up to 20th order. The only difference between this and a `Multipole` is the input format. See [§4.9](#) for more details. For *an* and *bn*, *n* is in the range 0 through 20.

The length *l* is a fictitious length that is used for synchrotron radiation computations and affects the longitudinal position of the next element but does not affect any tracking or transfer map calculations. The *x\_pitch* and *y\_pitch* attributes are not used in tracking.

Like a `MAD` multipole, An `AB_Multipole` will affect the reference orbit if there is a dipole component.

Example:

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

## 2.2 BeamBeam

<i>Attribute Class</i>	<i>§</i>	<i>Attribute Class</i>	<i>§</i>
Aperture Limits	<a href="#">4.6</a>	Offsets, pitches, and tilt	<a href="#">4.4</a>
Description strings	<a href="#">4.2</a>	Is_on	<a href="#">4.8</a>
Reference energy	<a href="#">4.3</a>	Tracking & transfer map	<a href="#">5</a>

Attributes specific to a `BeamBeam` element are:

```
sig_x   = <Real>      ! Horizontal strong beam sigma
sig_y   = <Real>      ! Vertical strong beam sigma
sig_z   = <Real>      ! Strong beam length
charge  = <Real>      ! Strong beam charge
n_slice = <Integer>   ! Number of strong beam slices
bbi_constant = <Real> ! Dependent attribute (§4.1).
```



A **BeamBeam** element simulates an interaction with an opposing (“strong”) beam traveling in the opposite direction. The strong beam is assumed to be Gaussian in shape. In the **bmad\_standard** calculation the beam–beam kick is computed using the Bassetti–Erskine complex error function formula[15]

**n\_part** is the nominal number of particles of the strong beam. **n\_part** is set using the **parameter** command (§7.1) and is thus common to all **BeamBeam** elements. To vary the number of particles in an individual **BeamBeam** element use the **charge** attribute. The default is **charge** = -1 which indicates that the strong beam has the opposite charge of the weak beam.

**sig\_x**, **sig\_y**, **sig\_z** are the strong beam’s sigmas. **x\_offset** and **y\_offset** are used to offset the **BeamBeam** element. Note that in **MAD** the attributes used to offset the strong beam are called **xma** and **yma**. Since the offsets might not be known until run time (they, of course, depend upon the particular orbits), often **x\_offset** and **y\_offset** will be set by a program rather than from the lattice file.

**x\_pitch** and **y\_pitch** gives the beam–beam interaction a crossing angle. This is the full crossing angle, not the half-angle.

The strong beam is divided up into **n\_slice** equal charge (not equal thickness) slices. The default for **n\_slice** is 1. Propagation through the strong beam involves a kick at the charge center of each slice with drifts in between the kicks. The kicks are calculated using the standard Bassetti–Erskine formula. Even though the strong beam can have a finite **sig\_z** the length of the element is always considered to be zero. This is achieved by adding drifts at either end of any tracking so that the longitudinal starting point and ending point are identical. The longitudinal *s*-position of the **BeamBeam** element is at the center of the strong bunch. For example, with **n\_slice** = 2 the calculation would proceed as follows:

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

**bbi\_constant**:  $C_{bbi} = N m_e r_e / (2 \pi \gamma (\sigma_x + \sigma_y))$  is a measure of the beam–beam interaction strength. For example, in the linear region near  $x = y = 0$  the horizontal component of the beam–beam kick is approximately  $k_x = -4 \pi x C_{bbi} / \sigma_x$  and the horizontal beam–beam tune shift is  $dQ_x = C_{bbi} \beta_a / \sigma_x$ .

Example:

```
bbi: beambeam, sig_x = 3e-3, sig_y = 3e-4, x_offset = 0.05
```

## 2.3 Bend\_Sol\_Quad

Attribute Class	§	Attribute Class	§
symplectify	5.4	Offsets, pitches, and tilt	4.4
Description strings	4.2	Is_on	4.8
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
Hkick & Vkick	4.5	an, bn multipoles	4.9
Integration settings	5.3		

Attributes specific to a **Bend\_Sol\_Quad** element are:

```
g          = <Real>      ! Bend strength 1/rho
angle      = <Real>      ! Bend angle. A settable dependent variable (§4.1)
```

```

rho      = <Real>      ! Bend radius. A settable dependent variable (§4.1)
bend_tilt = <Real>      ! Bend tilt angle. See §4.4.
k1       = <Real>      ! Quad strength.
x_quad   = <Real>      ! Quad horizontal offset.
y_quad   = <Real>      ! Quad vertical offset.
quad_tilt = <Real>      ! Quad tilt. See §4.4.
ks       = <Real>      ! Solenoid strength.
dks_ds   = <Real>      ! Solenoid field variation.
tilt     = <Real>      ! Overall tilt. See §4.4

```

**Bend\_Sol\_Quad** is a combination Bend, Solenoid, and Quadrupole with the solenoid strength varying linearly with longitudinal position. This enables the simulation of solenoid edge fields. The magnetic field is:

$$\begin{aligned}
\frac{q B_x}{P_0} &= -g_y + k_{1n}(y - y_q) - k_{1s}(x - x_q) - \frac{dks/ds}{2} x \\
\frac{q B_y}{P_0} &= g_x + k_{1n}(x - x_q) + k_{1s}(y - y_q) - \frac{dks/ds}{2} y \\
\frac{q B_s}{P_0} &= k_s + dks/ds
\end{aligned} \tag{2.1}$$

The reference trajectory is along the solenoid centerline. The quadrupole field is offset from the solenoid by (**x\_quad**, **y\_quad**). The quadrupole and bend have individual tilts **quad\_tilt** and **bend\_tilt** respectively. **tilt** gives an overall tilt. Thus the normal and skew quadrupole components  $k_{1n}$ , and  $k_{1s}$  are given by

```

k_1n = k1 * cos (2*(tilt + quad_tilt))
k_1s = k1 * sin (2*(tilt + quad_tilt))

```

and the dipole bend components ( $g_x$ ,  $g_y$ ) are given by

```

g_x = g * cos (tilt + bend_tilt)
g_y = g * sin (tilt + bend_tilt)

```

Dipole edge fields have not been implemented since it is not clear where the entrance and exit faces of the bend should be and how they are aligned with the solenoid.

To simulate a real solenoid you will need at least three **bend\_sol\_quad** elements: The middle element is the body of the solenoid with the linear solenoid strength **dks\_ds** = 0 and the two end elements have nonzero **dks\_ds** to simulate the solenoid edges.

Currently, tracking through a **Bend\_Sol\_Quad** is via symplectic integration only. **bmad\_standard** tracking is not an option since there is a possibility in the future to implement tracking via a closed formula. Example:

```
bsq: bend_sol_quad, l = 3.7, ks = -2.3, dks_ds = 4.7, g = 1/87
```

## 2.4 Bends: Rbend and Sbend

Attribute Class	§	Attribute Class	§
Symplectify	5.4	Offsets, pitches, and tilt	4.4
Description strings	4.2	Is_on	4.8
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
Hkick & Vkick	4.5	an, bn multipoles	4.9
Integration settings	5.3		

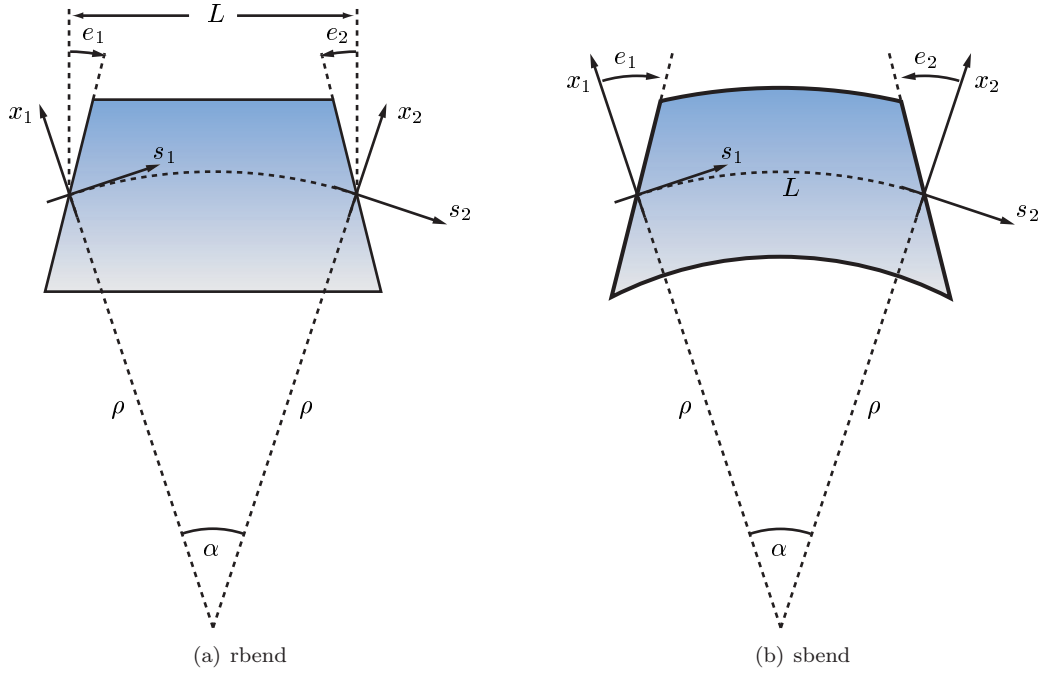


Figure 2.1: Coordinate systems for (a) Rbend and (b) S bend elements.

Attributes specific to a Rbend and S bend elements are:

<code>g</code>	= <Real>	! Design bend strength (= 1/rho).
<code>g_err</code>	= <Real>	! Bend strength error (§4.1).
<code>b_field</code>	= <Real>	! Field strength (= $P_0 g / q$ ) (§4.1).
<code>b_field_err</code>	= <Real>	! Field strength error (§4.1).
<code>angle</code>	= <Real>	! Bend angle. A settable dependent variable (§4.1).
<code>rho</code>	= <Real>	! Bend radius. A settable dependent variable (§4.1).
<code>e1, e2</code>	= <Real>	! Face angles.
<code>fint, fintx</code>	= <Real>	! Face field integrals.
<code>hgap, hgapx</code>	= <Real>	! Pole half gap.
<code>h1, h2</code>	= <Real>	! Face curvature.
<code>roll</code>	= <Real>	! See 4.4.
<code>k1</code>	= <Real>	! Quadrupole strength.
<code>k2</code>	= <Real>	! Sextupole strength (§4.1).
<code>b1_gradient</code>	= <Real>	! Quadrupole field strength (§4.1).
<code>b2_gradient</code>	= <Real>	! Sextupole field strength (§4.1).
<code>n_ref_pass</code>	= <Int>	! Multipass reference turn (§3.4).
<code>ref_orbit</code>	= <Switch>	! Multipass reference geometry switch (§3.4).
<code>l_chord</code>		! Chord length. Dependent attribute. See §4.7.

Rbend and S bend are dipole bends. The difference between the two is the way the `l`, `e1`, and `e2` attributes are interpreted. To ease the bookkeeping burden, after reading in a lattice, *Bmad* will internally convert all Rbends into Sbends. this is done on using the following transformation on Rbends:

```

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

```

**l, l\_chord**

For a **Rbend**, **l** is the chord length and not the arc length as it is for a **Sbend**. However, after reading in a lattice, *Bmad* will internally convert all **Rbends** into **Sbends**, additionally, the **l\_chord** attribute will be set to the input **l**, and **l** will be set to the true path length (see above).

**h1, h2**

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

**e1, e2**

the rotation angle of the entrance pole face is **e1** and at the exit face it is **e2**. Zero **e1** and **e2** for an **Rbend** gives a rectangular magnet (Figure 2.1a). Zero **e1** and **e2** for a **Sbend** gives a wedge shaped magnet (Figure 2.1b). An **Sbend** with an  $\mathbf{e1} = \mathbf{e2} = \mathbf{angle}/2$  is equivalent to an **Rbend** with  $\mathbf{e1} = \mathbf{e2} = 0$  (see above).

**angle**

The total design bend angle. A positive **angle** represents a bend towards negative  $x$  values (see Figure 9.1).

**k1, b1\_gradient**

The normalized and unnormalized quadrupole strength.

**k2, b2\_gradient**

The normalized and unnormalized sextupole strength.

**g, g\_err, rho**

The design bending radius which determines the reference coordinate system is **rho** (see §9.1).  $\mathbf{g} = 1/\mathbf{rho}$  is the curvature function and is proportional to the design dipole magnetic field. The true field strength is given by  $\mathbf{g} + \mathbf{g\_err}$  so changing **g\_err** leaves the design orbit unchanged but varies a particle's orbit.

**fint, fintx, hgap, hgapx**

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

$$F_{int} = \int_{pole} ds \frac{B_y(s)(B_0 - B_y(s))}{2H_{gap}B_0^2} \quad (2.2)$$

with a similar equation for **fintx**. **hgap** and **hgapx** are the half gaps at the entrance and exit faces. If **fint** or **fintx** is given without a value then a value of 0.5 is used. If **fint** or **fintx** is not present then the default value of 0 is used. Note: *MAD* does not have the **fintx** and **hgapx** attributes. *MAD* just assumes that the values are the same for the entrance and exit faces. For compatibility with *MAD*, if **fint** is given but **fintx** is not, then **fintx** is set equal to **fint**. Similarly, **hgapx** will be set to **hgap** if **hgapx** is not given.

**tilt**

The roll angle about the longitudinal axis at the entrance face of the bend is given by **tilt**. **tilt** = 0 bends the reference trajectory in the  $-x$  direction. If the **tilt** attribute is given without any value then the value  $\pi/2$  will be used. This makes for a **downward** pointing vertical bend.

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

Since internally all **Rbends** are converted to **Sbends**, if one wants to vary the **g** attribute of a bend and still keep the bend rectangular, an overlay (§3.1) can be constructed to maintain the proper face angles. For example:

```

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

```

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

`n_ref_pass`, and `ref_orbit` attributes are only used when a bend is part of a `multipass` line and is used to set the reference geometry of the bend. See section §3.4 for more details.

In the local coordinate system (§9.1), looking from “above” (bend viewed from positive  $y$ ), and with `tilt` = 0, a positive `angle` represents a particle rotating clockwise. In this case, `g` will also be positive. For counterclockwise rotation, both `angle` and `g` will be negative but the length `l` is always positive. Also, looking from above, a positive `e1` represents a clockwise rotation of the entrance face and a positive `e2` represents a counterclockwise rotation of the exit face. This is true irregardless of the sign of `angle` and `g`. Also it is always the case that the pole faces will be parallel when

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

Example bend specification:

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

## 2.5 Branch and Photon\_Branch

Attribute Class	§	Attribute Class	§
Description strings	4.2	Offsets	4.4
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
		Is_on	4.8

Attributes specific to a `Branch` and `Photon_Branch` element are:

```

direction = <Integer> ! +1 for outgoing branch (default). -1 for an incoming branch.
to = <line_name>      ! Name of line connected at the branch point.

```

A `Branch` element marks the start of an alternative line for the particle beam. A `Photon_Branch` element marks the start of a photon beam line. See §6.6 for more details. Example:

```

b_mark: branch, to = b_line
b_line: line = (...)

```

## 2.6 Collimators: Ecollimator and Rcollimator

Attribute Class	§	Attribute Class	§
Description strings	4.2	Offsets	4.4
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
Symplectify	5.4	Integration settings	5.3
Hkick & Vkick	4.5		

An `Ecollimator` is a drift with elliptic collimation. A `Rcollimator` is a drift with rectangular collimation. Note: Collimators are the exception to the rule that the aperture is independent of any `tilts`. See §4.6 for more details. Example:

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

## 2.7 Crystal

A `crystal` element is used for photon diffraction. Coding for tracking through a `crystal` is now under development and this section will be filled out as the code is developed.

## 2.8 Custom

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
Symplectify	5.4	Offsets, pitches, and tilt	4.4
Description strings	4.2	Is_on	4.8
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
Integration settings	5.3		

Attributes specific to a `Custom` element are

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

A `custom` element is an element whose properties are defined outside of the standard *Bmad* subroutine library. That is, to use a custom element, some programmer must write the appropriate custom routines which are then linked with the *Bmad* subroutines into a program. *Bmad* will call the custom routines at the appropriate time to do tracking, transfer matrix calculations, etc. See the programmer who wrote the custom routines for more details! See §18.1 on how to write custom routines.

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

Example:

```
c1: custom, l = 3, val4 = 5.6, val12 = 0.9, ds_step = 0.2, tracking_method = boris
```

## 2.9 Drift

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
Description strings	4.2	Symplectify	5.4
Integration settings	5.3	Hkick & Vkick	4.5

A `Drift` element is a space free and clear of any fields. Example:

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

## 2.10 Elseparator

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
Symplectify	5.4	Offsets, pitches, and tilt	4.4
Description strings	4.2	Is_on	4.8
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
Hkick & Vkick	4.5	$a_n$ , $b_n$ multipoles	4.9
Integration settings	5.3		

Attributes specific to a Bend\_Sol\_Quad element are:

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

A ElSeparator is an electrostatic separator.

For an Elseparator, the kick is determined by `hkick` and `vkick`. The `gap` for an Elseparator is used to compute the electric field for a given kick. The voltage is a dependent attribute determined by:

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

Example:

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

## 2.11 Hkicker and Vkicker

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
Symplectify	5.4	tilt	4.4
Description strings	4.2	Is_on	4.8
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
Kick	4.5	$a_n$ , $b_n$ multipoles	4.9
Integration settings	5.3		

A Hkicker is a horizontal bend and a Vkicker is a vertical bend. Note that Hkicker and Vkicker elements use the kick attribute while a kicker uses the `hkick` and `vkick` attributes. Example:

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

## 2.12 Hybrid

A Hybrid element is an element that is formed by concatenating other element together. Hybrid elements are not part of the input lattice file but are created by a program, usually for speed purposes.

## 2.13 Girder

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
Description strings	4.2	Offsets, pitches, and tilt	4.4

Attributes specific to an **Girder** are:

```
girder = {<List>}    ! List of elements on the Girder
```

An **Girder** is a support structure that orients the elements that are attached to it in space.

When an **Girder** overlays an element, then that elements orientation attributes (**x\_offset**, **y\_pitch**, **tilt**, etc.) give the orientation of the element with respect to the **Girder**. An example will make this clear:

```
q1: quad, l = 10
q2: quad, l = 5, x_offset = 0.2, x_pitch = 0.01
ib: girder = {q1, q2}, x_pitch = 0.1, x_offset = 0.3
this_line: line = (q1, q2)
use, this_line
```

In this example **ib** supports elements **q1** and **q2**. The center of **ib** is at  $s = 7.5$  (**ib** starts at  $s = 0$  which is the beginning of **q1** and ends at  $s = 15$  which is the end of **q2**). Like other elements, pitch is calculated from the center of an **Girder** element (see Sec. 4.4). The center of **q2** is at  $s = 12.5$  so the distance between the center of **ib** and **q2** is  $ds = 5$ . The pitch of **ib** produces an offset at the center of **q2** of  $0.5 = 0.1 * 5$ . This, added to the offsets of **ib** and **q2**, give the total offset of **q2** to be  $1.0 = 0.5 + 0.3 + 0.2$ . The total **x\_pitch** of **q2** is  $0.11 = 0.1 + 0.01$ . From the above example it can be seen that an **Girder** looks similar to an **Overlay** (see Sec. 3.1). It would, however, take six **Overlays** to simulate the effect of a single **Girder**.

The **Girder** statement syntax is:

```
beam_name: GIRDER = {ele1, ele2, ... }, ...
```

An **Girder** element will be created for each **ele1** element in the lattice. The elements **ele2**, **ele3**, etc. do not have to be consecutive but, if more than one **Girder** is to be created, need to be in order of increasing **s**. For example:

```
q1: quad
q2: quad
s0: sextupole
s1: sextupole
ib: girder = {q1, s1, q2}
this_line: line = (q1, s0, s1, q2, ..., q1, s0, s1, q2)
use, this_line
```

In this example two **Girder** elements will be created.

Note to programmers: The total horizontal offset of any element is stored in the element component `%value(x_offset_tot$)`. Similarly the total tilt is stored in `%value(tilt_tot$)`, etc.

## 2.14 Instrument, Monitor, and Pipe

Attribute Class	§	Attribute Class	§
Symplectify	5.4	Offsets, pitches, and tilt	4.4
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
Description strings	4.2	Is_on	4.8
Integration settings	5.3	Hkick & Vkick	4.5
Instrumental variables	4.10		

*Bmad* treats **instrument**, **monitor**, and **pipe** elements like a **drift**. There is a difference, however, when superimposing elements (§3.3). For example, a **quadrupole** superimposed on top of a **drift** results in a



free `quadrupole` element in the tracking part of the lattice and no `lord` elements are created. On the other hand, a `quadrupole` superimposed on top of a `monitor` results in a `quadrupole` element in the tracking part of the lattice and this `quadrupole` element will have two `lords`: A `quadrupole` superposition `lord` and a `monitor` superposition `lord`.

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

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

## 2.15 Kicker

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
Symplectify	5.4	tilt	4.4
Description strings	4.2	Is_on	4.8
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
Hkick & Vkick	4.5	<i>an</i> , <i>bn</i> multipoles	4.9
Integration settings	5.3		

A `Kicker` can deflect a beam in both planes. Note that a `Kicker` uses the `hkick` and `vkick` attributes while `Hkicker` and `Vkicker` elements use the `kick` attribute. In addition a `Kicker` can apply a displacement to a particle using the `h_displace` and `v_displace` attributes. Example:

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

## 2.16 Lcavity

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
Symplectify	5.4	Offsets, and pitches	4.4
Description strings	4.2	Is_on	4.8
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
Hkick & Vkick	4.5	Integration settings	5.3

An `Lcavity` is a LINAC accelerating cavity. The transverse trajectory through an `Lcavity` is modeled using equations developed by Rosenzweig and Serafini[16] modified to give the correct phase-space area at non ultra-relativistic energies. See Section §10.5 for more details.

The attributes specific to an `Lcavity` are

```
coupler_at      = <Switch> ! Location of a coupler giving a transverse kick.
coupler_strength = <Real>  ! Strength of the coupler.
coupler_angle   = <Real>  ! Angle of the kick.
coupler_phase   = <Real>  ! Phase of the kick.
gradient        = <Real>  ! Accelerating gradient (V/m).
gradient_err     = <Real>  ! Accelerating gradient error (V/m).
phi0            = <Real>  ! Phase (rad/2 $\pi$ ) of the reference particle with
                        ! respect to the RF. phi0 = 0 is on crest.
dphi0           = <Real>  ! Phase with respect to a multipass lord (rad/2 $\pi$ ).
```

```

phi0_err      = <Real>    ! Phase error (rad/2 $\pi$ )
e_loss        = <Real>    ! Loss parameter for short range wake fields (V/Coul).
rf_frequency   = <Real>    ! Rf frequency (Hz).
delta_e        = <Real>    ! Change in energy of an on-crest particle.
                        ! Dependent attribute (§4.1).
sr_wake_file   = <String> ! Short range wake field definition file.
lr_wake_file   = <String> ! Long range wake field definition file.
lr_freq_spread = <Real>    ! Frequency spread of the LR wake fields.

```

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

```
delta_e = gradient * L
```

The energy kick felt by a particle is

```

dE = gradient_tot * L * cos(twopi * (phi_ref + phi_err + phi(z))) -
      e_loss * n_part * e_charge

```

where

```

gradient_tot = gradient + gradient_err
phi_ref = phi0 + dphi0
phi_err = phi0_err
phi(z) = -z * rf_frequency / c_light

```

`phi0` is only to be used to shift the phase with respect to a `multipass` lord. See §3.4.

The `e_loss` attribute in the formula for `dE` represents the energy loss due to short-range wake fields. `n_part` is set using the `parameter` statement (§7.1) and represents the number of particles in a bunch. `e_charge` is the charge on an electron (Table 1.1). Notice that the energy kick is independent of the sign of the charge of the particle

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

```
dE(reference) = gradient * L * cos(twopi * phi_ref) - e_loss * n_part * e_charge
```

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

```
dE = gradient * L * cos(twopi * (phi_ref - phi(z))) ! WRONG
```

This is incorrect.

Note: The transfer matrix for an `Lcavity` with finite `gradient` is never symplectic. See §9.3.

The attributes that characterize the dipole transverse kick due to a coupler port are:

```

coupler_at      = <Switch> ! What end the coupler is at
coupler_strength = <Real>   ! Normalized strength
coupler_angle    = <Real>   ! Polarization angle (rad/2 $\pi$ )
coupler_phase    = <Real>   ! Phase angle with respect to the RF (rad/2 $\pi$ )

```

The possible `coupler_at` settings are:

```

entrance_end
exit_end ! default
both_ends

```

For `coupler_angle = 0` the transverse kick due to the coupler is

```

dp_x = gradient * coupler_strength *
      cos(twopi * (phi_ref + coupler_phase + phi(z))) / (c * P_0)
dp_y = 0

```

The formulas used to compute the wake field are given in §10.13. The input file name for the short-range wake fields is specified using the `sr_wake_file` attribute. The file gives both monopole longitudinal and dipole transverse wakes. Comment lines may be included by starting a line with an exclamation mark (!). Blank lines are also ignored. An example input file is:

```

!      z      Wz      Wt
!      [m]      [V/C/m]      [V/C/m^2]
0.000E+00  1.61125E+15  0.00000E+00    1
-1.000E-05  1.44516E+15 -1.30560E+15    2
-2.000E-05  1.38148E+15 -2.50665E+15    3
.. etc ..
-1.970E-03  3.49958E+14 -7.95507E+16   198
-1.980E-03  3.48606E+14 -7.97253E+16   199
-1.990E-03  3.47263E+14 -7.98989E+16   200
END_SECTION

! Pseudo Wake modes:
!      Amp      damp      k      phase
! Longitudinal:  [V/C/m]      [1/m]      [1/m]      [rad]
! Transverse:    [V/C/m^2]    [1/m]      [1/m]      [rad]

&short_range_modes
longitudinal(1) = 3.23e14      1.23e3      3.62e3      0.123
longitudinal(2) = 6.95e13      5.02e2      1.90e3     -1.503
.. etc ..
transverse(1) = 4.23e14      2.23e3      5.62e3      0.789
transverse(2) = 8.40e13      5.94e2      1.92e3      1.455
.. etc ..
z_max = -1.3e-3
/

```

The file is divided into two sections with a line containing the word `END_SECTION` marking the division between the sections. Wakes can be specified via a table of wake versus longitudinal position  $z$  and/or using a set of “pseudo” modes (§10.13). The first section gives the wake vs  $z$  table, and the second section gives the longitudinal monopole and transverse dipole pseudo modes. The range of the table is from 0 to  $z_{cut}$  where  $z_{cut}$  is the  $z$  value in the last line of the table. If the longitudinal distance  $dz$  between two particles is within the range of the table then the table will be used to calculate the wake kick for this pair. If  $dz$  is larger than  $z_{cut}$  the pseudo modes will be used. The pseudo modes are valid from  $z_{cut}$  to  $z_{max}$ .

In the first section with the table of wake vs.  $z$ , the first column is the longitudinal distance  $z$ .  $z$  must start at 0 and must increment by the a constant amount from row to row.  $z$  is negative since the wake extends behind a particle. The second column is the longitudinal wake function in  $V/C/m$ . The third column is the transverse wake in  $V/C/m^2$ . Any additional columns are ignored. Wake field formulas are to be found in §10.13. The wake field file is only used with macroparticle and particle distribution tracking. When the short-range wake field file is used with either of these the `e_loss` attribute is ignored. However, even in this case, a finite `e_loss` value will affect the reference energy. Since the quantities like quadrupole  $k_1$  strengths and bend strengths are referenced to the reference energy, The value of `e_loss` will affect the results even with a short-range wake field file.

The input file name for the long-range wake fields is specified using the `lr_wake_file` attribute. The file gives the wake modes by specifying the frequency (in Hz),  $R/Q$  (in  $\Omega/\text{meter}^{2m}$ ),  $Q$ , and  $m$  (order number), and optionally the polarization angle (in radians/2pi) for each cavity mode. The input uses

Fortran90 namelist syntax: The data begins with the string `&long_range_modes` and ends with a `/`. Everything outside is ignored. Each mode is labeled `lr(i)` where `i` is the mode index. An example input file is:

```

      Freq      R/Q      Q      m      Polar      b_sin  b_cos a_sin  a_cos  t_ref
      [Hz]      [Ohm/      Angle
                  m^(2m)]      [Rad/2pi]
&long_range_modes
  lr(1) = 1.650e9    0.76    7.0e4    1    unpol
  lr(2) = 1.699e9   11.21    5.0e4    1    0.15
/
```

If “unpolarized” is used for the polarization angle, the mode is taken to be unpolarized.

`lr_freq_spread` is used to randomly spread out the long range mode frequencies among different cavities. The spread is Gaussian in shape with an RMS of `lr_freq_spread * F` where  $F$  is the frequency of a mode. After the long-range modes have been defined they can be referenced or redefined using the notation

```

  lr(n)%freq      ! Frequency
  lr(n)%r_over_q  ! R/Q
  lr(n)%q         ! Q
  lr(n)%angle     ! Polarization Angle
```

Example:

```
lcav[lr(2)%freq] = 1.1 * lcav[lr(2)%freq] ! Raise frequency by 10%
```

Example:

```
rf1: lcav, l = 4.5, gradient = 1.2e6, sr_wake_file = "sr1.dat"
```

## 2.17 Marker

Attribute Class	§	Attribute Class	§
Description strings	4.2	Is_on	4.8
Reference energy	4.3	Offsets and tilt	4.4
Aperture Limits	4.6	Tracking & transfer map	5
Instrumental variables	4.10		

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

Example:

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

## 2.18 Match

Attribute Class	§	Attribute Class	§
Description strings	4.2	Is_on	4.8
Aperture Limits	4.6	Length	4.7
Reference energy	4.3	Integration settings	5.3

Attributes specific to a **Match** element are:

```

beta_a0 = <Real>, beta_b0 = <Real>    ! Beginning betas
beta_a1 = <Real>, beta_b1 = <Real>    ! Ending betas
alpha_a0 = <Real>, alpha_b0 = <Real>  ! Beginning alphas
alpha_a1 = <Real>, alpha_b1 = <Real>  ! Ending alphas
eta_x0 = <Real>, eta_y0 = <Real>      ! Beginning etas
eta_x1 = <Real>, eta_y1 = <Real>      ! Ending etas
etap_x0 = <Real>, etap_y0 = <Real>    ! Beginning eta'
etap_x1 = <Real>, etap_y1 = <Real>    ! Ending eta'
dphi_a = <Real>, dphi_b = <Real>      ! Phase advances
match_end = <Logical>                 ! See below. Default is False.
x0, px0, y0, py0, z0, pz0 = <Real>    ! Beginning coordinates
x1, px1, y1, py1, z1, pz1 = <Real>    ! Ending coordinates
match_end_orbit = <Logical>           ! See below. Default is False.

```

A `Match` element is used to match the Twiss parameters between two points. That is, the transfer map for a match element, which is just a linear matrix, is calculated such that if the Twiss parameters at the exit end of the element preceding the `match` element are given by `beta_a0`, `beta_b0`, etc., then the computed Twiss parameters at the exit end of the `match` element will be `beta_a1`, `beta_b1`, etc. `dphi_a` and `dphi_b` are the phase advances in radians.

If the `match_end` attribute is set to `True` then the beginning Twiss parameters are ignored and the transfer matrix is calculated using the Twiss parameters from the exit end of the previous element. That is, the actual Twiss parameters at the exit end of the match element will be the Twiss parameters as set in the element. The `match_end` attribute may only be used with `linear_lattice` lattices (§7.1).

When running a program, if a `match` element has its `match_end` attribute is set to `True`, the beginning Twiss and dispersion parameters (`beta_a0`, `eta_x0`, etc.) of the `match` element are continuously updated to be equal to the Twiss and dispersion parameters at the end of the the element before it. As a consequence, if `match_end` for the `match` element is toggled to `False`, the values of the elements in the transfer matrix through the `match` element are frozen in place.

The coordinate parameters (`x0`, `x1`, etc.) add a constant term (a “kick”) to the transfer map through a `match` element:

$$r_1 = \mathbf{M} r_0 + \mathbf{V} \quad (2.3)$$

where  $r_1$  is the output coordinates,  $r_0$  are the input coordinates,  $\mathbf{M}$  is the transfer matrix determined by the settings of the beginning and ending twiss and dispersion parameters, and the kick term,  $\mathbf{V}$  is given by

$$\mathbf{V} = \begin{pmatrix} x1 \\ px1 \\ y1 \\ py1 \\ z1 \\ pz1 \end{pmatrix} - \mathbf{M} \begin{pmatrix} x0 \\ px0 \\ y0 \\ py0 \\ z0 \\ pz0 \end{pmatrix} \quad (2.4)$$

If `match_end_orbit` is set to `True`, a particle tracked through the `match` element will have its ending coordinates equal to (`x1`, `px1`, `y1`, `py1`, `z1`, `pz1`) and the the beginning coordinate parameters (`x0`, `px0`, `y0`, `py0`, `z0`, `pz0`) will be set to the particle’s coordinates at the beginning of the match element. Similar to `match_end`, if `match_end_orbit` for the `match` element is toggled to `False`, the  $BfV$  vector will become fixed.

The attribute `l` is not used in the transfer matrix calculation. It is sometimes needed by a program for other computations. For example, to compute the time it takes to go through a match element.

Example:

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

## 2.19 Mirror

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
Description strings	4.2	Offsets, pitches, and tilt	4.4
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6		

Attributes specific to a Mirror element are:

```

graze_angle      = <Real>      ! Angle between incoming beam and mirror surface.
graze_angle_err  = <Real>      ! Error in the graze angle.
critical_angle   = <Real>      ! Critical angle.
tilt_err         = <Real>      ! Error in the tilt angle.
g_graze          = <Real>      ! 1/Radius-of-curvature in the bend plane.
g_trans          = <Real>      ! 1/Radius-of-curvature transverse to the bend plane.
```

A Mirror reflects photons. The reference trajectory for a mirror is that of a zero length bend (§9.2.1) and hence the length (1) parameter of a mirror is fixed at zero. The reference trajectory is determined by the values of the `graze_angle` and `tilt` parameters. A positive `graze_angle` bends the reference trajectory in the same direction as a positive `g` for a bend element. The `graze_angle_err` and `tilt_err` parameters affect the orientation of the mirror in the global reference system but do not affect the reference trajectory.

The `g_graze` parameter is equal to  $1/R_{\text{graze}}$  where  $R_{\text{graze}}$  is the radius of curvature of the mirror surface in the bend plane as shown in Fig. 9.3. A positive `g_graze` indicates that the mirror is concave from the point of view of the photons. Similarly, the `g_trans` parameter is equal to  $1/R_{\text{trans}}$  where  $R_{\text{trans}}$  is the radius of curvature of the mirror surface in the plane transverse to the bend plane. Again, a positive `g_trans` indicates that the mirror is concave from the point of view of the photons.

A mirror may be offset and pitched (4.4). The incoming local reference coordinates are used for these misalignments. The `x_pitch` and `y_pitch` parameters can be related to the `graze_angle_err` and `tilt_err` parameters. For example, with zero tilt, a positive `x_pitch` is equivalent to a negative `graze_angle_err` of the same magnitude, and a positive `y_pitch` is equivalent to a negative `tilt_err` of the same magnitude.

## 2.20 Multipole

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
<i>KnL, Tn multipoles</i>	4.9	Offsets, pitches, and tilt	4.4
Description strings	4.2	Is_on	4.8
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6		

A Multipole is a thin multipole lens up to 20th order. The only difference between this and an `AB_Multipole` is the input format. See the Magnetic fields section 4.9 for more details.

The length 1 is a fictitious length that is used for synchrotron radiation computations and affects the longitudinal position of the next element but does not affect any tracking or transfer map calculations.

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

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

## 2.21 Null\_Ele

A `Null_Ele` is a special type of element. It is like a `Marker` but it has the property that when the lattice is expanded (§6.2) all `Null_Ele` elements are removed. The primary use of a `Null_Ele` is in computer generated lattices where it can be used to serve as a reference point for element superpositions (§3.3). It is not generally useful otherwise.

## 2.22 Octupole

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
Symplectify	5.4	Offsets, pitches, and tilt	4.4
Description strings	4.2	Is_on	4.8
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
Hkick & Vkick	4.5	an, bn multipoles	4.9
Integration settings	5.3		

An `Octupole` is a magnetic element with a cubic field dependence with transverse offset (§10.2). The `Bmad_Standard` calculation treats an octupole using a kick-drift-kick model.

Attributes specific to an `Octupole` element are:

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

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

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

## 2.23 Patch

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
is_on	4.8	Offsets, pitches, and tilt	4.4
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Description strings	4.2
Length	4.7		

Attributes specific to a `Patch` elements are:

```
x_offset      = <Real>
y_offset      = <Real>
z_offset      = <Real>
x_pitch       = <Real>
y_pitch       = <Real>
tilt          = <Real>
pz_offset     = <Real>
translate_after = <Logical> ! Default: False.
```

```
patch_end     = <Logical> ! Default: False.
x_position    = <Real>
y_position    = <Real>
```

```

z_position      = <Real>
theta_position  = <Real>
phi_position    = <Real>
psi_position    = <Real>

```

A **patch** element shifts the reference orbit. This is a generalization of *MAD*'s **yrot** and **srot** elements. For example, a positive **x\_offset** offsets the reference orbit after the **patch** in the positive  $x$ -direction relative to the reference orbit before the **patch**. Hence, the  $x$  coordinate of a particle going through a patch with a positive **x\_offset** will be decreased. See §9.2.2 for formulas for the reference orbit transformation with a **patch**.

Normally, for computing the change in the reference orbit, the offsets are applied before the pitches and tilts. If reverse is desired, the **translate\_after** logical can be set to True.

If the **patch\_end** logical is set to True, the **patch** element will change its behavior. In this case, the setting of the offsets, pitches and tilt will be ignored, and the global position of the reference coordinates at the exit end of the **patch** will be fixed by the setting of **x\_position**, **y\_position**, **z\_position**, **theta\_position**, **phi\_position** and **psi\_position**. Here, the transfer map through the **patch** will be the unit map. That is, the phase space coordinates of a particle will not change when tracking through such an element.

The **l** length attribute is used to set the longitudinal  $s$  distance between the previous and next elements and a program can, for example, use **l** to compute the time it takes to go through the element. Otherwise, the value of **l** is not used tracking or transfer matrix calculations.

Example:

```
pt: patch, x_offset = 3.2
```

## 2.24 Quadrupole

Attribute Class	§	Attribute Class	§
Symplectify	5.4	Offsets, pitches, and tilt	4.4
Description strings	4.2	Is_on	4.8
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
Hkick & Vkick	4.5	$a_n$ , $b_n$ multipoles	4.9
Integration settings	5.3		

An **Octupole** is a magnetic element with a linear field dependence with transverse offset (§10.2).

Attributes specific to a **Quadrupole** element are:

```

k1          = <Real>    ! Quadrupole strength.
b1_gradient = <Real>    ! Field strength. (§4.1).

```

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

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



## 2.25 RFcavity

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
Symplectify	5.4	Offsets, & pitches	4.4
Description strings	4.2	Is_on	4.8
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
Hkick & Vkick	4.5		

Attributes specific to an RFcavity are:

```

rf_frequency = <Real>    ! Frequency
harmon       = <Real>    ! Harmonic number
voltage      = <Real>    ! Cavity voltage
phi0         = <Real>    ! Cavity phase
dphi0        = <Real>    ! Phase variation with multipass

```

An RFcavity is an RF cavity without acceleration generally used in a storage ring.

The phi0 attribute here is identical to the lag attribute of MAD. The energy kick felt by a particle is

$$dE = e\_charge * voltage * \sin(2\pi * (\phi\_ref - \phi(z)))$$

where

```

phi_ref = phi0 + dphi0
phi(z) = -z * rf_frequency / c_light

```

dphi0 is only to be used to shift the phase with respect to a multipass lord. See §3.4. e\_charge is the charge on an electron (Table 1.1). Notice that the energy kick is independent of the sign of the charge of the particle

If harmon is non-zero then rf\_frequency is a dependent attribute calculated by

$$rf\_frequency = harmon * c\_light / L\_lattice$$

where L\_lattice is the total lattice length.

Example:

```
rf1: rfcav, l = 4.5, harmon = 1281, voltage = 5e6
```

## 2.26 Sextupole

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
Symplectify	5.4	Offsets, pitches, and tilt	4.4
Description strings	4.2	Is_on	4.8
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
Hkick & Vkick	4.5	an, bn multipoles	4.9
Integration settings	5.3		

A Sextupole is a magnetic element with a quadratic field dependence with transverse offset (§10.2).

Attributes specific to an Sextupole element are:

```

k2          = <Real>    ! Sextupole strength.
b2_gradient = <Real>    ! Field strength. (§4.1).

```

The Bmad\_Standard calculation treats a sextupole using a kick-drift-kick model.

If the tilt attribute is present without a value then a value of  $\pi/6$  is used. Example:

```
q03w: sext, l = 0.6, k2 = 0.3, tilt ! same as tilt = pi/6
```

## 2.27 Solenoid

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
Symplectify	5.4	Offsets & pitches	4.4
Description strings	4.2	Is_on	4.8
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
Hkick & Vkick	4.5	$a_n, b_n$ multipoles	4.9
Integration settings	5.3		

Attributes specific to an Solenoid element are:

```
ks          = <Real>    ! Solenoid strength.
bs_field    = <Real>    ! Field strength. (§4.1).
```

A Solenoid is an element with a longitudinal magnetic field. Example:

```
cleo_sol: solenoid, l = 2.6, ks = 1.5*beam[energy]Q
```

## 2.28 Sol\_Quad

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
Symplectify	5.4	Offsets, pitches, and tilt	4.4
Description strings	4.2	Is_on	4.8
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.7
Hkick & Vkick	4.5	$a_n, b_n$ multipoles	4.9
Integration settings	5.3		

Attributes specific to a Sol\_Quad element are:

```
k1          = <Real>    ! Quadrupole strength.
ks          = <Real>    ! Solenoid strength.
bs_field    = <Real>    ! Solenoid Field strength.
bl_gradient = <Real>    ! Quadrupole Field strength.
```

A Sol\_Quad is a combination solenoid/quadrupole. Example:

```
sq02: sol_quad, l = 2.6, k1 = 0.632, ks = 1.5*beam[energy]
```

## 2.29 Taylor

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
Description strings	4.2	Is_on	4.8
Aperture Limits	4.6	Symplectify	5.4
Length	4.7	Tracking & transfer map	5
Reference energy	4.3		

A Taylor is a Taylor map (§10.3). This can be used in place of the *MAD* matrix element.

Attributes specific to a Taylor element are:

```
{<out>: <coef>, <e1> <e2> <e3> <e4> <e5> <e6> } ! Taylor coefficient.
```

A term in a Taylor map is of the form

$$x_j(\text{out}) = C \cdot \prod_{i=1}^6 x_i^{e_i}(\text{in}) \quad (2.5)$$

where  $\mathbf{x} = (x, p_x, y, p_y, z, p_z)$ . For example a term in a Taylor map that was

$$p_y(\text{out}) = 2.73 \cdot y^2(\text{in}) p_z(\text{in}) \quad (2.6)$$

would be written as

```
{4: 2.73, 0 0 2 0 0 1}
```

By default a Taylor element starts out as the unit map. That is, a Taylor element starts with the following 6 terms

```
{1: 1.0, 1 0 0 0 0 0}
{2: 1.0, 0 1 0 0 0 0}
{3: 1.0, 0 0 1 0 0 0}
{4: 1.0, 0 0 0 1 0 0}
{5: 1.0, 0 0 0 0 1 0}
{6: 1.0, 0 0 0 0 0 1}
```

A term in a Taylor element will override any previous term with the same out and e1 through e6 indexes. For example the term:

```
tt: Taylor, {1: 4.5, 1 0 0 0 0 0}
```

will override the default {1: 1.0, 1 0 0 0 0 0} term.

The 1 length attribute is not in any map calculation. 1 can be used to set the longitudinal  $s$  distance between the previous and next elements and a program can, for example, use 1 to compute the time it takes to go through the element.

Example Taylor element definition:

```
tt: Taylor, {4: 2.73, 0 0 2 0 0 1}, &
          {2: .2.73, 2 0 0 0 0 1}
```

## 2.30 Wiggler

Attribute Class	§	Attribute Class	§
Symplectify	<a href="#">5.4</a>	Offsets, pitches, and tilt	<a href="#">4.4</a>
Description strings	<a href="#">4.2</a>	Is_on	<a href="#">4.8</a>
Reference energy	<a href="#">4.3</a>	Tracking & transfer map	<a href="#">5</a>
Aperture Limits	<a href="#">4.6</a>	Length	<a href="#">4.7</a>
Hkick & Vkick	<a href="#">4.5</a>	an, bn multipoles	<a href="#">4.9</a>
Integration settings	<a href="#">5.3</a>		

A Wiggler is basically a periodic array of alternating bends. There are two types of wigglers. Those that are described using a magnetic field map (“map type”) and those that are described assuming a periodic field (“periodic type”). See [§10.6](#) for more details. Tracking a particle through a wiggler is always done so that if the particle starts on-axis with no momentum offsets, there is no change in the  $z$  coordinate even though the actual trajectory through the wiggler does not follow the straight line reference trajectory.

Attributes specific to a map type Wiggler element are:

```
term(i) = {<Wig_Term>}
polarity = <Real>      ! Used to scale the field strength.
x_ray_line_len = <Real>
```

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

A `<Wig_Term>` is of the form  $C, k_x, k_y, k_z, \phi_z$  as explained in §10.6. `Polarity` is used to scale the magnetic field. By default, `Polarity` has a value of 1.0. Example:

```
wig1: wiggler, l = 1.6, &
      term(1) = {0.03, 3.00, 4.00, 5.00, 0.63}, &
      term(2) = ...
...
wig1[polarity] = -1 ! Reverse the polarity of the wiggler
```

For the `periodic` type wigglers the attributes are:

```
b_max    = <Real> ! Maximum magnetic field (in T) on the wiggler centerline.
polarity = <Real> ! Used to scale the field strength.
l_pole   = <Real> ! Wiggler pole length. The period is then 2 * l_pole.
n_pole   = <Real> ! The number of poles (L / L_POLE).
           ! A settable Dependent attribute (§4.1).
k1        ! Vertical focusing strength. Dependent attribute (§4.1).
rho       ! Bending radius. Dependent attribute (§4.1).
```

Example:

```
wig2: wiggler, l = 1.6, b_max = 2.1, n_pole = 7 ! periodic type wiggler
```

The type of wiggler is determined by whether there are `term(i)` terms. If present, the wiggler is classed as a `map` type.

Note: When using Taylor maps and symplectic tracking with a `periodic` type wiggler, the number of poles must be even as explained in section §10.6.

## Chapter 3

# Control Elements: Overlays, Groups, Superpositions, and Multipass

It is possible to have elements controlling the attributes of other elements. These **lord** elements are meant, for example, to mimic the effect of changing a knob in the control room. For this purpose the lattice is split into two sections: The first section is the list of elements that *Bmad* uses for any analysis (tracking, Twiss parameter calculations, etc.). This first part is called the **tracking** part of the lattice. Elements in the **tracking** part are either **slave** elements if they have a controlling lord or **free** elements if they do not. The second section consists solely of **lord** elements. **Lord** elements can control other **lord** elements and a hierarchy of **lord** elements may be established.

There are five types of lord elements:

- **Group** lord elements are used to make variations in attributes. For example, to simulate the action of a control room knob that changes the beam tune in a storage ring, a **Group** can be used to vary the strength of selected quads in a specified manner. **Groups** are covered in §3.2.
- An **Overlay** lord is like a **group** except that **overlays** set the value (not a change in) the attributes they control. **Overlays** are covered in §3.1.
- A **superposition** lord is created when elements are superimposed on top of other elements. This is covered in §3.3.
- **Girder** elements mimic the effect of a support girder. This is covered in §2.13.
- **Multipass** elements are used when the beam recirculates through the same element multiple times. This is covered in §3.4.

### 3.1 Overlay Elements

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

```
over1: overlay = {a_ele, b_ele/2.0}, hkick = 0.003
over2: overlay = {b_ele}, hkick
over2[hkick] = 0.9
a_ele: quad, hkick = 0.05, ...
b_ele: rbend, ...
this_line: line = ( ... a_ele, ... b_ele, ... )
use, this_line
```

In the example the overlay `over1` controls the `hkick` attribute of the "slave" elements `a_ele` and `b_ele`. `over2` controls the `hkick` attribute of just `b_ele`. `over1` has a `hkick` value of 0.003 and `over2` has been assigned a value for `hkick` of 0.9.

There are coefficients associated with the control of a slave element. The default coefficient is 1.0. To specify a coefficient use a slash "/" after the element name followed by the coefficient. In the above example the coefficient for the control of `b_ele` from `over1` is 2.0 and for the others the default 1.0 is used. thus

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

An `overlay` will control all elements of a given name. Thus, in the above example, if there are multiple elements in `this_line` with the name `b_ele` then the `over1` and `over2` overlays will control the `hkick` attribute of all of them.

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

The default value for an overlay is 0 so for example

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

will make `c_ele[k1] = 0`. Overlays can also control more than one type of attribute as the following example shows

```
over4: overlay = {this_quad[k1]/3.4, this_sextupole[k2], ...}, hkick
```

## 3.2 Group Elements

`group` is like `overlay` in that a `group` element controls the attribute values of other "slave" elements. A `group` element is used to make changes in value. This is unlike an `overlay` which sets a specific value directly. An example will make this clear

```
gr: group = {q1}, k1
gr[command] = 0.34
q1, quad, 1 = ...
q1[k1] = 0.57
```

In this example the group `gr` controls the `k1` attribute of the element `q1`. Unlike overlays, values are assigned to group elements using the `command` attribute. When a lattice file is read in then command values for any groups are always applied last. This is independent of the order that they appear in the file. Thus in this example the value of `q1[k1]` would be  $0.91 = 0.57 + 0.34$ . When the changes are made to the slave attributes the value of `command` is stored in the `group's old_command` attribute. After the lattice is read in a program can change the `gr[command]` attribute and this change will be added to the value of `q1[k1]`. The bookkeeping routine that transfers the change from `gr[command]` to `q1[k1]` doesn't care what the current value of `q1[k1]` is. It only knows it has to change it by the change in `gr[command]`.

A `group` can be used to control an elements position and length using the attributes

```
accordion_edge ! Element grows or shrinks symmetrically
start_edge     ! Varies element's starting edge s-position
end_edge       ! Varies element's ending edge s-position
symmetric_edge ! Varies element's overall s-position. Constant length.
s_offset       ! Similar to symmetric_edge
```

With `accordion_edge`, `start_edge`, `end_edge`, and `symmetric_edge` the longitudinal position of an elements edges are varied. This is done by appropriate control of the element's length and the lengths of the elements to either side. With `s_offset` the physical element is offset from its reference position (§4.4) and the elements on either side are untouched. In all cases the total length of the lattice is kept invariant.

As an example, consider `accordion_edge` which varies the edges of an element so that the center of the element is fixed but the length varies. With `accordion_edge` a change of, say, 0.1 in a `group`'s `command` attribute moves both edges of the element by 0.1 meters so that the length of the element changes by 0.2 meters. To keep the total lattice length invariant the lengths of the elements to either side are varied accordingly. For example

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

This last line that defines `gr2` is just a shorthand notation for

```
gr2: group = {q10}, start_edge
gr2[command] = 0.1
```

The effect will be to lengthen the length of `q10` and shorten the length of `d1`.

The full list of attributes of a group are

```
command
old_command
coef
type          ! See section 4.2
alias         ! See section 4.2
descrip       ! See section 4.2
```

The `coef` attribute is not used by any *Bmad* routine. It is defined for individual programs to store, say, a needed conversion factor.

Like `overlays`, coefficients can be specified for the individual elements under a `group`'s control and `groups` can control more than one type of attribute. For example

```
gr3: group = {q1[k1]/-1.0, q2[tilt], oct1/-2.0}, k3
gr3[command] = 2.0
gr3[old_command] = 1.5
```

In this example `gr3` controls 3 attributes of 3 different elements. The change in `gr3` when the lattice is read in is  $0.5 = 2.0 - 1.5$ . this 0.5 change will change `q1[k1]` by  $-0.5 = -1 \times 0.5$ , `q2[tilt]` will change by 0.5 and `oct1[k3]` will change by  $-1.0 = -2.0 \times 0.5$ .

### 3.3 Superposition

In practice the field at a particular point in the lattice may be due to more than one physical element. A common example is a quadrupole inside a solenoid. *Bmad* has a mechanism to handle some of these cases using what is called “superposition”. A simple example shows how this works:

```
Q: quad, l = 10
D: drift, l = 10
S: solenoid, l = 6, superimpose, ref = q, ref_end, offset = -1
lat: line = (Q, D)
use, lat
```

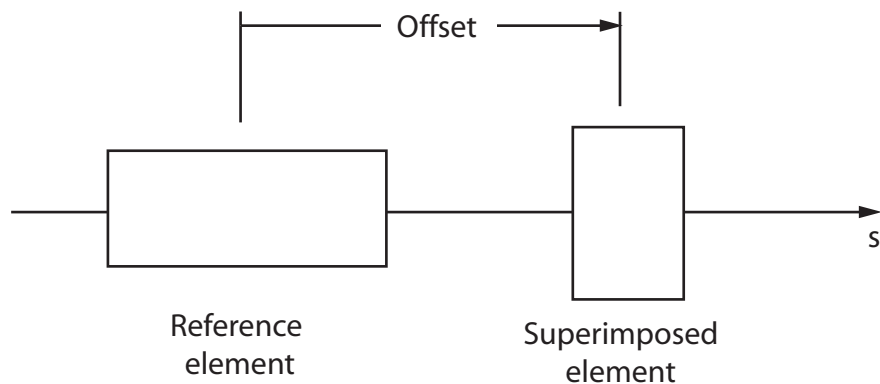


Figure 3.1: Illustration of a superposition.

The `superimpose` attribute of element `S` superimposes `S` over the lattice (`Q`, `D`). The placement of `S` is such that the center of `S` is offset -1 meter from the end of `Q` (more on how superimposed elements get placed later). The tracking part of the lattice list (the part that one does tracking through) Looks like:

	Element	Key	Length	Total
1)	Q#1	Quadrupole	6	6
2)	Q\S	Sol_quad	4	10
3)	S#1	Solenoid	2	12
4)	D#2	Drift	8	20

What *Bmad* has done is to split the original elements (`Q`, `D`) at the edges of `S`. The first element in the lattice, `Q#1`, is the part of `Q` that is outside of `S`. Since this is only part of `Q`, *Bmad* has put a `#1` in the name so that there will be no confusion. (`#` has no special meaning other than the fact that *Bmad* uses it for mangling names). The next element, `Q\S`, is the part of `Q` that is inside `S`. `Q\S` is a combination solenoid/quadrupole element as one could expect. `S#1` is the part of `S` that is outside `Q` so this element is just a solenoid. Finally, `D#2` is the rest of the drift outside `S`.

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).

Superpositions are restricted in that *Bmad* may not have an appropriate element for the superimposed fields. For example, a solenoid can not be superimposed over an octupole. If a zero length element, such as a marker, is superimposed with some other element (or vice versa) the element is just split in two. For example:

```
Q: quad, l = 10
M: marker, superimpose, offset = 6
lat: line = (Q)
use, lat
```

The resulting lattice (first part) would be

	Element	Key	Length
1)	Q#1	Quadrupole	6
2)	M	Marker	0
3)	Q#2	Quadrupole	4



and the second part of the lattice would have the Q element.

A superposition is illustrated in Figure 3.1 The placement of a superimposed element is determined by three factors: A reference point on the superimposed element, a reference point in the lattice line, and an offset between the points. The attributes that determine these three quantities are:

```
ref = <element name in lattice>
offset = <length>          (default = 0)
ref_beginning
ref_center                  (default)
ref_end
ele_beginning
ele_center                  (default)
ele_end
```

`ref` sets the reference element. If `ref` is not present then the start of the lattice is used. `ref_beginning`, `ref_center` or `ref_end` can be used to indicate where on the reference element the reference point is. Default is `ref_center`. Similarly, `ele_beginning`, `ele_center`, or `ele_end` can be used to indicate the reference point on the superimposed element at the beginning (entrance) edge, the center, or the end (exit) edge respectively. If neither of these attributes are given the default is to use the element center. `offset` is the longitudinal offset between the reference point on the reference element and the reference point on the superimposed element. The default if not present is zero.

Superposition may be done with any element except `Drift`, `Group`, `Overlay`, and `Girder` control elements. A superimposed element that extends beyond either end of the lattice will be wrapped around. For consistency's sake, this is done even if the `lattice_type` is set to `linear_lattice` (for example, It is sometimes convenient to treat a circular lattice as linear).

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. Thus, it is an error if a second superposition uses the drift as the reference element. For example

```
dft1: drift, l = 10
q1: quad, l = 1, superposition, ref = dft1    ! OK
q2: quad, l = 2, superposition, ref = dft1    ! ERROR!
```

### 3.3.1 Changing Element Lengths when there is Superposition

When the lattice is constructed, superposition of elements is done before the addition of any `group` or `overlay` elements. This is done since `overlays` and `groups` are allowed to refer to elements that are superimposed. This can lead to some unexpected results. For example:

```
q1: quad, l = 10
q2: quad, l = 10
lat: line = (q1, q2)
use, lat
o: overlay = q1, l = 12
m: marker, superimpose, offset = 15
```

In this example, the marker is initially positioned at 15 meters. The application of the overlay will increase the length of `q1` by 2 meters which will push the marker `m` to 17 meters which is probably not what was intended. To avoid this problem, an `expand_lattice` statement (§7.8) can be placed after the overlay, but before the `superimpose`, statement

```
...
o: overlay = q1, l = 12
expand_lattice
m: marker, superimpose, offset = 15
```

The length of a superimposed element must necessarily be equal to the sum of the lengths of its slave elements. For example, the element **S** in the first example of Section §3.3 has a length of 6 meters which is equal to the sum of the length of **Q\S** (3 meters) plus the length of **S#1** (also 3 meters).

If a program varies the length of a superimposed element after the lattice has been parsed, the length of the slaves will be adjusted accordingly. For example, if element **S** in the first example of Section §3.3 is increased from 6 meters to 9 meters, The lattice would look like

	Element	Key	Length	Total
1)	<b>Q#1</b>	Quadrupole	4	4
2)	<b>Q\S</b>	Sol_quad	6	10
3)	<b>S#1</b>	Solenoid	3	13
4)	<b>D#2</b>	Drift	8	21

The length of **Q\S** has been increased from 4 meters to 6 meters and the length of **S#1** has been increased from 2 meters to 3 meters to give the proper length for **S**. Additionally, to keep the length of **Q** at 10 meters, the length of **Q#1** has been decreased to 4 meters. Notice that the overall length of the lattice has increased by 1 meter.

Notice that this result is *not* what would be obtained if the length of the element **S** is increased to 9 meters in the lattice file. The reason for this incompatibility stems from the fact that, in general, it is a complex problem to have length changes after parsing mirror what would happen if the lengths were changed in the lattice file itself. As a result, *Bmad* chooses to use a relatively simple algorithm. In practice, lattice layouts can be designed without superimpose, using the `no_superimpose` command (§7.9), and when the element lengths are fixed, superposition can be used.

## 3.4 Multipass

Some lattices have the beam recirculating through the same element multiple times. For example, an Energy Recovery Linac (ERL) will circulate the beam back through the LINAC part to retrieve the energy in the beam. In *Bmad* this situation can simulated using the `multipass` attribute. A simple example shows how this works.

```
A: lcavity
linac_part: line[multipass] = (A, ...)
my_line: line = (linac_part, ..., linac_part)
use, my_line
expand_lattice
A\2[dphi0] = 0.5
```

The tracking part of the lattice consists of two slave elements

```
A\1, ..., A\2, ...
```

Since the two elements are derived from a `multipass` line they are given unique names by adding a `\n` suffix. In addition there is a lord element (that doesn't get tracked through) called **A** in the lord part of the lattice. Changes to attributes of the lord **A** element will be passed to the slave elements by *Bmad*'s bookkeeping routines. Assuming **A\1** is an accelerating cavity, to make **A\2** a decelerating cavity the `dphi0` attribute of **A\2** is set to 0.5. This is the one attribute that *Bmad*'s bookkeeping routines will not touch when transferring attribute values from **A** to its slaves. Notice that the `dphi0` attribute had to be set after `expand_lattice` (§7.8) is used to expand the lattice since *Bmad* does immediate evaluation and **A\2** does not exist before the lattice is expanded.

Sublines of a `multipass` line are automatically `multipass`:

```
a_line: line = (...)
m_line: line[multipass] = (... , a_line, ...)
```

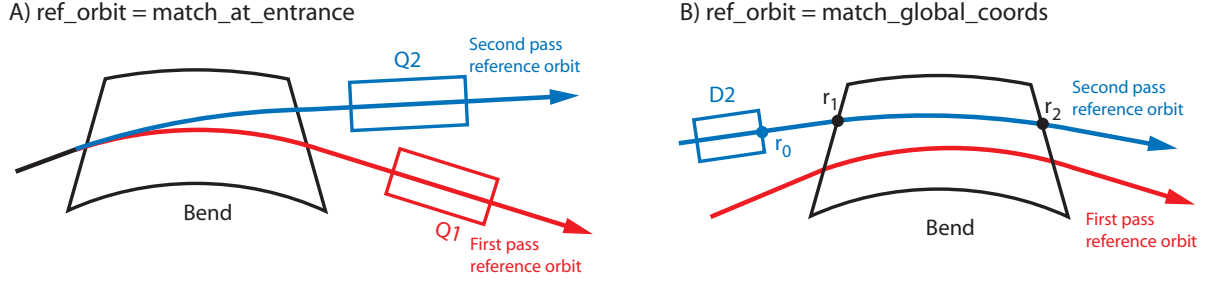


Figure 3.2: A) With `ref_orbit = match_at_entrance`, the reference orbits of different pass will be taken to be the same at the entrance end of the magnet. If there is a variation of the reference energy through a bend from pass to pass, the reference orbit through the bend will vary from pass to pass. B) With `ref_orbit = match_global_coords`, the reference orbit on the first pass establishes the position of the bend in the global coordinate system and the reference orbit on other passes is calculated with respect to this.

In this example `a_line` is implicitly multipass.

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

```
m_line: line[multipass] = (A, A, B)
u_line: line = (m_line, m_line)
use, u_line
```

In this example the tracking part of the lattice is

```
A\1, A\1, B\1, A\2, A\2, B\2
```

In the control section of the lattice there will be two multipass lords called A and one called B. The first A lord controls the 1<sup>st</sup> and 4<sup>th</sup> elements in the tracking part of the lattice and the second A lord controls the 2<sup>nd</sup> and 5<sup>th</sup> elements.

If a multipass line is reversed then the elements are considered to be transversed backwards:

```
m_line: line[multipass] = (A, A, B)
u_line: line = (m_line, -m_line)
use, u_line
```

In this example the tracking part of the lattice is

```
A\1, A\1, B\1, B\2, A\2, A\2
```

Here the 1<sup>st</sup> and 6<sup>th</sup> elements are connected to a multipass lord and the 2<sup>nd</sup> and 5<sup>th</sup> elements are connected to a different multipass lord.

### 3.4.1 The Reference Orbit in a Multipass Line

If there are `lcavity` elements in the lattice then the reference energy at a given element may differ from pass to pass. In this case, the normalized strength (`k1`, `kick`, etc.) for magnetic and electric elements will not be the same from pass to pass. To avoid an ambiguity, all magnetic and electric elements that are used in a multipass line must have their magnetic or electric field strength set as the independent attribute (§4.1), or a reference energy (§4.3) must be defined. A reference energy is defined by setting `e_tot` or `p0c`, or by setting `n_ref_pass` as described below.

For a bend in a multipass line, there is the added complication of how to define the reference orbit when the reference energy is different from pass to pass. If, say, a different reference orbit is used for each pass

there is a problem of how to interpret multipole values like `k1`. On the other hand, it is sometimes very convenient to be able to use separate reference orbits.

A bend element has a `ref_orbit` attribute that, when combined with the reference energy, define the reference orbit. `ref_orbit` is a switch that can take the values:

```
single_ref          ! Default.
match_global_coords ! Must be used with n_ref_pass = 1
match_at_entrance
match_at_exit
patch_in            ! Used with patch elements.
patch_out           ! And must be used with n_ref_pass = 1.
```

A value of `single_ref` (the default) means that the *same* reference orbit will be used for all passes. The `match_global_coords`, `match_at_entrance` and `match_at_exit` values are used when *separate* reference orbits are desired as described below.

To set the reference energy, one (and only one) of the attributes `n_ref_pass`, `e_tot` or `p0c` needs to be set. `n_ref_pass` is an integer indicating which pass is used to define the reference energy for the bend element. Note: If `ref_orbit` is set to `match_global_coords`, `n_ref_pass` must be used and must be set to 1.

An example will make this clear. Consider a bend defined by

```
B: sbend, l = 1, b_field = 1.0, n_ref_pass = 1, ref_orbit = single_ref
```

Here the same reference orbit will be used for all passes. Assume that the reference momentum `p0c` on the first pass is 5 GeV (this is determined by the reference momentum at the beginning of the lattice and any `lcavity` elements in the lattice). Since `n_ref_pass` is one, this fixes the momentum at which the reference orbit is calculated to be 5 GeV. The bending radius `rho` of the reference orbit is related to the field and momentum by

$$\text{rho} = \text{p0c} / (\text{c\_light} * \text{b\_field})$$

In this case, this translates into a bending radius of 16.7 m. In the tracking lattice, the first pass element `B\1` will have a value for `b_field` of 1 Tesla just like its lord element. Also, like its lord element, `B\1` will have an error field, `b_field_err`, of zero. The total field

$$\text{b\_field\_tot} = \text{b\_field} + \text{b\_field\_err}$$

will be 1 Tesla. Additionally, the edge face angles `e1` and `e2` will be the same as `B`. In this example both are zero. This is true for all passes.

Now assume that on the second pass, the reference momentum is 10 GeV. Since, on the second pass, `p0c` is a factor of 2 larger, to keep the bending radius invariant, the value of `b_field` for the second pass element `B\2` will be a factor of 2 larger. That is, it will be 2 Tesla. However, the total field in the element must be the same on all passes (`B\1` and `B\2` represent the same physical element after all). Thus, `B\2` must have a value for `b_field_err` of -1 Tesla.

Consider the case where the value of `ref_orbit` is set to `match_at_entrance` and the reference momentum is set instead of `n_ref_pass`:

```
B: sbend, l = 1, b_field = 1.0, p0c = 5e9, ref_orbit = match_at_entrance
```

Here the reference orbit will be different on the two passes. The two reference orbits will coincide at the entrance edge of the magnet. This is illustrated in Figure 3.2A. For both `B\1` and `B\2`, the value of `b_field` will be 1 Tesla giving bending radii of 16.7 m and 33.4 m respectively. With `B\2`, the element length will be a slight bit larger than `B\1` with a value of 1.0000045. The entrance face angle `e1` will always be the same on all passes (zero in this example) but `e2` will vary. In this case `e2` for `B\2` will be 0.003. With the different reference orbit for the second pass, there is no good way to handle a non-zero multipole attribute so *Bmad* disallows them in this case.

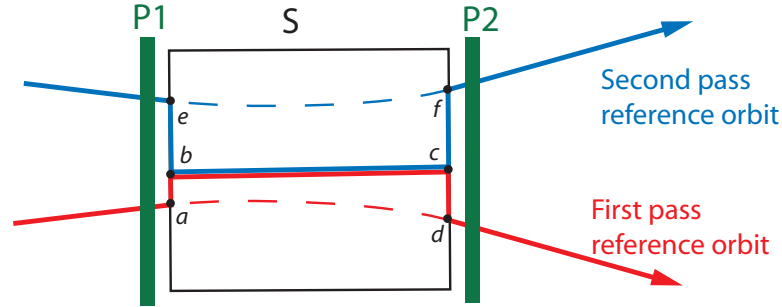


Figure 3.3: Using patch elements to vary the reference orbit in a multipass line. The gap between patch elements P1 and P2 and the lattice section S is for illustration purposes only.

`match_at_exit` is similar to `match_at_entrance` except that the different reference orbits will coincide at the exit edge of the magnet. If `match_at_exit` is used with the parameters in the above example, the only change for `B\2` is that `e1` will now be 0.003 and `e2` will be zero.

The `match_at_entrance` value for `ref_orbit` is useful for simulating a bend at the end of a multipass line that is used to separate the beams on the various passes. Notice that the next elements downstream (labeled Q1 and Q2 in Figure 3.2A) will be physically separated in space. That is, it does not make sense to use `match_at_entrance` for a bend that is *not* at the end of a multipass line. Similarly, the `match_at_exit` value is useful for simulating a bend that combines beams of different energies. Such a bend should be placed at the beginning of a multipass line.

With `ref_orbit` set to `match_global_coords` the reference orbit on a given pass is determined by the global coordinates (§9.2) of the reference trajectory. Example:

```
B: sbend, l = 1, b_field = 1.0, n_ref_pass = 1, ref_orbit = match_global_coords
```

As will be explained, when using `match_global_coords`, `n_ref_pass` must be present and set to one. With `match_global_coords`, `B\1` will have the same parameters as `B` and this will determine the reference orbit for `B\1`. See Figure 3.2B). For the second pass, the reference trajectory at the entrance end of the bend is determined as follows: The calculation starts with the global coordinates of the reference orbit at the exit end of the element just before `B\2` (marked  $r_0$  in Figure 3.2B)). To be physically correct, this point must lie on the entrance face of `B\2`. However, it is possible to have a lattice where this is not so and the calculation does not demand this. If  $r_0$  does not lie on the entrance face, the reference orbit is extended (either forward or backward) in a straight line to a point where it intersects the entrance face (marked  $r_1$  in Figure 3.2B)). The radius of curvature is calculated by scaling the radius as calculated from the first pass scaled by the change in reference momentum between the first and second passes. From the entrance point and the radius of curvature, the exit point (marked  $r_2$  in Figure 3.2B)) can be calculated. The curve from  $r_1$  to  $r_2$  defines the reference trajectory through `B\2`. The face angles `e1` and `e2`, and the path length `l` for `B\2` can be calculated from the geometry.

Since the calculation with `match_global_coords` is rather complicated, the use of `match_global_coords` is restricted to lattices that lie horizontally in the global  $X-Z$  plane. Also, the restriction to `n_ref_pass = 1` is necessary since the calculation becomes nondeterministic otherwise.

### 3.4.2 Using Patch elements to Vary the Reference Orbit in a Multipass Line

There exist situations where it is desirable to have a different reference orbits for each pass through a non-bend elements. Such a situation is illustrated in Figure 3.3. In this example, beams of differing energy pass through an a section of the lattice labeled S. This section may contain one or more elements.

The desired reference orbit for each pass follows the beam trajectories. This is achieved by placing patch elements, called P1 and P2, just before and just after S. The corresponding lattice file would look like

```
p1: patch, ref_orbit = patch_in, n_ref_pass = 1, translate_after = True
p2: patch, ref_orbit = patch_out, n_ref_pass = 1, ref_patch = p1

m_line: line[multipass] = (p1, S, p2)
all_line: line = (... , m_line, ... , m_line, ...)
use, all_line

expand_lattice
p1\1[x_offset] = 0.05
p1\1[x_pitch] = -0.0034
```

The `ref_orbit` parameter for `p1` and `p2` indicate whether the patch is just before (`patch_in`) or just after (`patch_out`) the section of interest. In this example, `x_offset` and `x_pitch` parameters for the first pass slave `p1\1` are set in the lattice file. This determines the orientation of the reference orbit through S (labeled *b-c* in the figure) with respect to the incoming first pass reference orbit. Point *a* in the figure is the first pass reference orbit just before `p1\1` and point *b* is the reference orbit just after. On the second pass, *Bmad* will calculate the patch parameters for `p1\2` so that the global coordinates of the reference orbit just after `p1\2` is the same as the reference orbit just after `p1\1`. That is, point *b*.

The patch parameters for the `p2\1` first pass slave of `p2`, which has `ref_orbit` set to `patch_out`, the calculation is as follows: In the definition of `p2` the `ref_patch` parameter is set to `p1`. *Bmad* uses this as the starting point for tracking. *Bmad* starts with a particle on the reference trajectory just *before* `p1\1` and tracks it to the end of S (curve *a-d* in the figure). *Bmad* then sets the parameters of `p2\1` so that the reference orbit after the patch (point *d*) coincides with the particle. The calculation of the second pass slave `p2\2` is analogously computed.

The above procedure of defining the reference orbit using patches with `ref_orbit` set to `patch_in` and `patch_out` is helpful in designing lattices. However, once a lattice is designed, there will be a problem for simulations of lattice errors since particle trajectories, and hence the reference orbit, will be affected by such errors. To avoid this, once the layout of the lattice has been settled on, the `patch_in` and `patch_out` patches should be removed and four new patches, to replace the four multipass slave patches, should be introduced outside of the multipass section with the patch parameters set to the appropriate values.

# Chapter 4

## Element Attributes

### 4.1 Dependent and Independent Attributes

For convenience, *Bmad* computes the values of some attributes based upon the values of other attributes. These dependent variables are listed in Table 4.1. Also shown in Table 4.1 are the independent variables they are calculated from. In the table `n_part` and `l_lattice` (lattice length) are lattice attributes, not element attributes. The first two are set by the `parameter` statement (See §7.1). `l_lattice` is calculated when the lattice is read in.

<i>Element</i>	<i>Dependent Variables</i>	<i>Independent Variables</i>
BeamBeam	bbi_constant	charge, sig_x, sig_y, e_tot, n_part
Elseparator	e_field, voltage	hkick, vkick, gap, l, e_tot
Lcavity	e_loss, delta_e	gradient, l
Rbend, Sbend	rho, angle, l_chord	g, l
Wiggler (periodic type)	k1, rho	b_max, e_tot
All elements	num_steps	ds_step

Table 4.1: Table of dependent variables and the independent variables they are calculated from.

No attempt should be made to set or vary within a program dependent attributes. It should be remarked that this is not an iron clad rule. If a program properly bypasses *Bmad*'s attribute bookkeeping routine then anything is possible. In a lattice file, before lattice expansion (§7.8), *Bmad* allows the setting of a select group of dependent attributes if the appropriate independent attributes are not set. The list of settable dependent variables is given in Table 4.2. After reading in the lattice *Bmad* will set the appropriate independent variable based upon the value of the dependent variable. `harmon` is the exception in that it will never be set by the bookkeeping routine.

<i>Element</i>	<i>Dependent Variable Set</i>	<i>Independent Variables Not Set</i>
Lcavity	delta_e	gradient
Rbend, Sbend	rho	g
Rbend, Sbend	angle	g, or l
RFcavity	rf_frequency	harmon
Wiggler (periodic type)	n_pole	l_pole

Table 4.2: Dependent variables that can be set in a primary lattice file.

The normal attribute used to vary the strength of, say, a **quadrupole** is **k1**. It is sometimes convenient to be able to vary the magnetic field strength directly instead. To do this *Bmad* has a rule that if the appropriate field attribute appears in the primary lattice file then it becomes an independent variable and the normalized strength attribute (the strength attribute normalized by the reference energy) becomes a dependent variable as tabulated in Table 4.3. Using both field strength and normalized strength

<i>Element</i>	<i>Normalized Strength</i>	<i>Field Attribute</i>
Sbend, Rbend	g	b_field
Sbend, Rbend	g_err	b_field_err
Solenoid, Sol_quad	ks	bs_field
Quadrupole, Sol_quad, Sbend, Rbend	k1	b1_gradient
Sextupole, Sbend, Rbend	k2	b2_gradient
Octupole	k3	b3_gradient
HKicker, VKicker	kick	bl_kick
Most	hkick	bl_hkick
Most	vkick	bl_vkick

Table 4.3: Field and Strength Attributes.

as the independent variable for a given element is not permitted. For example, for a quadrupole the normalized strengths **k1**, **hkick**, and **vkick** can be used as the independent variable or the field strengths **b1\_gradient**, **bl\_hkick** and **bl\_vkick**. but the mixing of the two is not valid

```
Q1: quadrupole, k1 = 0.6, bl_hkick = 37.5 ! NO. Not VALID.
```

To define an element with the field strength as the independent attribute without setting the strength just set the strength to zero or, alternatively, the **field\_master** logical can be set. For example

```
Q1: quadrupole, b1_gradient = 0 ! Field strengths now the independent variables
Q1: quadrupole, field_master = T ! Same as above
```

The same effect can be obtained by setting the field or **field\_master** attributes after the element has been defined.

```
q1: quadrupole ! Define q1.
q1[b1_gradient] = 0 ! Field strengths now the independent variables.
q1[field_master] = T ! Same as above.
```

## 4.2 Type, Alias and Descrip Attributes

There are three string labels associated with any element:

```
type    = <String>
alias   = <String>
descrip = <String>
```

*Bmad* routines do not use these labels except when printing element information. **type** and **alias** can be up to 16 characters in length and **descrip** can be up to 200 characters. The attribute strings can be enclosed in double quotation marks ("). The attribute strings may contain blanks. If the attribute string does not contain a blank then the quotation marks may be omitted. In this case the first comma (,) or the end of the line marks the end of the string. Example:

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



### 4.3 Beam\_Energy and P0C Attributes

The attributes that define the reference energy and momentum at an element are:

```
e_tot = <Real> ! Total energy in eV.
p0c   = <Real> ! Momentum in eV.
```

The energy and momentum are defined at the exit end of the element. For ultra-relativistic particles these two values are the same (§9.3). Except for multipass elements (§3.4), `e_tot` and `p0c` are dependent attributes and, except for multipass elements, any setting of `e_tot` and `p0c` in the lattice input file will be ignored. The value of `e_tot` and `p0c` for an element is calculated by *Bmad* to be the same as the previous element except for `Lcavity` and `patch` elements. To set the `e_tot` or `p0c` at the start of the lattice use the `beginning` or `parameter` statements. See §7.1. Since the energy changes from the start to the end of an `Lcavity`, An `Lcavity` has the dependent attributes

```
e_tot_start and
p0c_start
```

which are just the reference energy and momentum at the start of the element.

For multipass elements, the reference energy is set by specifying one of `e_tot`, `p0c`, or `n_ref_pass` as described in §3.4.

### 4.4 Offset, Pitch, Tilt, and Roll Attributes

There are up to 7 attributes that can offset a physical element from the reference orbit. They are

```
x_offset = <Real>
y_offset = <Real>
s_offset = <Real>
x_pitch  = <Real>
y_pitch  = <Real>
tilt     = <Real>
roll     = <Real>
```

The exception here is the `Patch` element which uses these attributes to modify the reference orbit itself.

`x_offset` translates an element in the local  $x$ -direction as shown in Figure 4.1. Similarly, `y_offset` and `s_offset` translate an element along the local  $y$  and  $z$ -directions respectively. For a bend it is assumed that the bend angle is small and the rotation of the local reference axes through the bend is ignored.

The `x_pitch` attribute rotates an element about the  $y$ -axis so that the exit face of the element is displaced in the  $+x$ -direction as shown in figure 4.1. Similarly the `y_pitch` attribute rotates an element about the  $x$ -axis so that the exit face of the element is displaced in the  $+y$ -direction. The rotations are about the center of the element which is in contrast to the `dtheta` and `dphi` misalignments of *MAD* which rotate around the entrance point. In terms of rotation angle

```
x_pitch = dtheta
y_pitch = -dphi
```

The `tilt` attribute rotates the element in the  $(x, y)$  plane as shown in figure 4.2. The rotation axis is the  $z$ -axis at the entrance face. The reference orbit is also rotated for any element who's exit coordinates are not collinear with the entrance coordinates. For example, a `bend` or `mirror` with a tilt of  $\pi/2$  will bend a beam vertically upward. The `hkick` and `vkick` attributes are not affected by `tilt` except for `Kicker` and `ElSeparator` elements. Like *MAD*, *Bmad* allows the use of the `tilt` attribute without a value to designate a skew element. For example

```
q1: quad, 1 = 0.6, x_offset = 0.03, y_pitch = 0.001, tilt
```

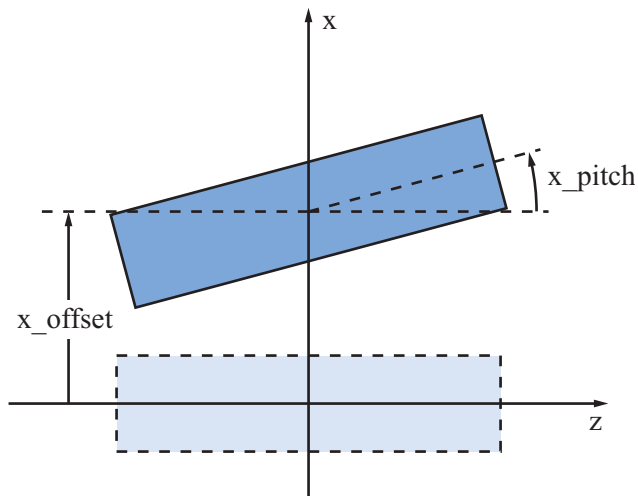


Figure 4.1: Geometry of Pitch and Offset attributes

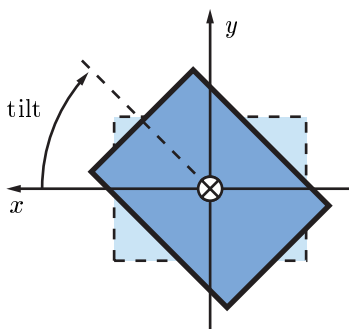


Figure 4.2: Geometry of a Tilt

Default tilts can be used for `rbend`, `sbend`, `sol_quad`, `quadrupole`, `sextupole`, and `octupole` elements. The default tilt is  $\pi/(2(n+1))$  where  $n$  is the order of the element ( $n = 0$  for bends,  $n = 1$  for quadrupoles etc.)

For all elements, offsets, pitches, and tilts are with respect to the entrance coordinates (the local coordinates just before the element).

The `roll` attribute is only used for bends and rotates the bend, along an axis that runs through the entrance point and exit point as shown in figure 4.3. A `roll` does not affect the reference orbit. The major effect of a `roll` is to give a vertical kick to the beam. A positive `roll` is similar to a positive `tilt`. That is, with a bend with positive bend angle, a positive `roll` will move the outside portion ( $+x$  side) of the bend upward and the inside portion ( $-x$  side) downward. Much like car racetracks which are typically slanted towards the inside of a turn.

## 4.5 Hkick, Vkick, and Kick Attributes

The kick attributes that an element may have are:

`kick`, `bl_kick` = <Real> ! Used only with a `Hkicker` or `Vkicker`

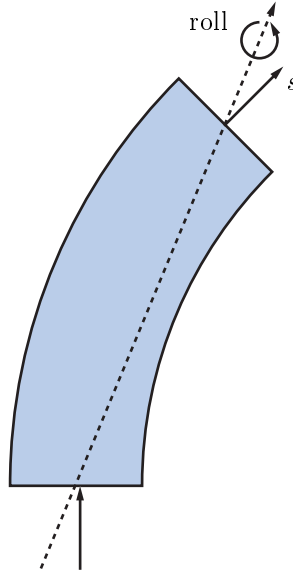


Figure 4.3: Geometry of a Roll

```

hkick, bl_hkick = <Real>
vkick, bl_vkick = <Real>

```

`kick`, `hkick`, and `vkick` attributes are the integrated kick of an element in radians. `kick` is only used for `Hkicker` and `Vkicker` elements. All other elements that can kick use `hkick` and `vkick`. The `tilt` attribute will only rotate a kick for `Hkicker`, `Vkicker`, `Elseparator` and `Kicker` elements. This rule was implemented so that, for example, the `hkick` attribute for a skew quadrupole would represent a horizontal steering. The `bl_kick`, `bl_hkick`, and `bl_vkick` attributes are the integrated field kick in meters-Tesla. Normally these are dependent attributes except if they appear in the lattice file (§4.1).

## 4.6 Aperture and Limit Attributes

The aperture attributes are:

```

x1_limit      = <Real>      ! Horizontal, negative side, aperture limit
x2_limit      = <Real>      ! Horizontal, positive side, aperture limit
y1_limit      = <Real>      ! Vertical, negative side, aperture limit
y2_limit      = <Real>      ! Vertical, positive side, aperture limit
x_limit       = <Real>      ! Alternative to specifying x1_limit and x2_limit
y_limit       = <Real>      ! Alternative to specifying y1_limit and y2_limit
aperture       = <Real>      ! Alternative to specifying x_limit and y_limit
aperture_at    = <Switch>    ! What end aperture is at.
aperture_type  = <Switch>    ! What type of aperture it is.
offset_moves_aperture = <Logical> ! Element offsets affect aperture position

```

`x1_limit`, `x2_limit`, `y1_limit`, and `y2_limit` specify the half-width of the aperture of an element as shown in figure 4.4. A zero `x1_limit`, `x2_limit`, `y1_limit`, or `y2_limit` is interpreted as no aperture in the appropriate plane.

By default, apertures are assumed to be rectangular except that an `Ecollimator` has a elliptical aperture. This can be changed by setting the `aperture_type` attribute. The possible values of this attribute are:

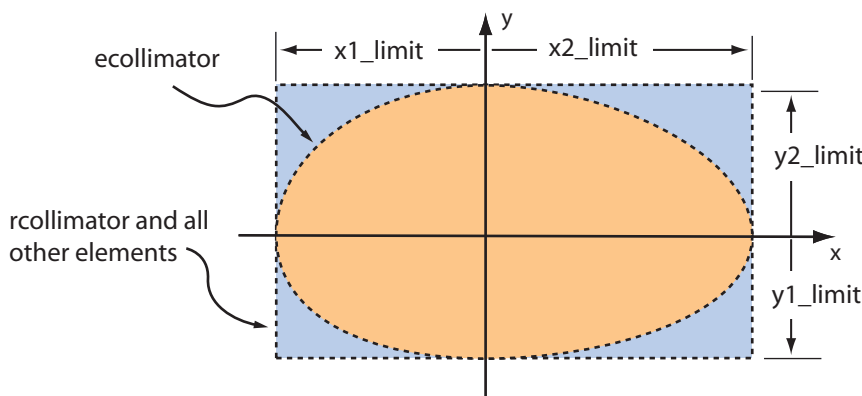


Figure 4.4: Apertures for ecollimator and rcollimator elements

```
rectanular
elliptical
```

To avoid numerical overflow and other errors in tracking, a particle will be considered to have hit an aperture in an element, even if there are no apertures set for that element, if its orbit exceeds 1000 meters. Additionally, there are other situations where a particle will be considered lost. For example, if a particle's trajectory does not intersect the output face in a bend.

For convenience, `x_limit` can be used to set `x1_limit` and `x2_limit` to a common value. Similarly, `y_limit` can be used to set `y1_limit` and `y2_limit`. The `aperture` attribute can be used to set all four `x1_limit`, `x2_limit`, `y1_limit` and `y2_limit` to a common value. Internally, the *Bmad* code does *not* store `x_limit`, `y_limit`, or `aperture`. This means that using `x_limit`, `y_limit` or `aperture` in arithmetic expressions is an error:

```
q2: quad, aperture = q1[aperture]    ! THIS IS AN ERROR!
q2: quad, aperture = q1[x1_limit]    ! Correct
```

Normally, whether a particle hits an aperture or not is evaluated independent of any element offsets (§4.4). This is equivalent to the situation where a beam pipe containing an aperture is independent of the element the beam pipe passes through. This can be changed by setting the `offset_moves_aperture` attribute to `True`. In this case any offsets or pitches will be considered to have shifted the aperture boundary. The exception here is that the default for `rcollimator` and `ecollimator` elements is for `offset_moves_aperture` to be `True`.

Even with `offset_moves_aperture` set to `True`, tilts will not affect the aperture calculation. This is done, for example, so that the tilt of a skew quadrupole does not affect the aperture. The exception here is that tilting an `rcollimator` or `ecollimator` element will tilt the aperture.

By default the aperture is evaluated at the exit face only of the element. This can be changed by setting the `aperture_at` attribute. Possible settings for `aperture_at` are:

```
entrance_end
exit_end    ! default
both_ends
```

Note that the entrance and exit ends of an element are independent of which direction particles are tracked through an element. Thus if a particle is tracked backwards it enters an element at the “exit end” and exits at the “entrance end”.

Examples:

```
q2: quad, aperture_type = elliptical
```

```

q1: quad, l = 0.6, x1_limit = 0.045, offset_moves_aperture = T
q1[y1_limit] = 0.03
q1[y2_limit] = 0.03
q1[y_limit] = 0.03 ! equivalent to the proceeding 2 lines.
q1[aperture_at] = both_ends

```

## 4.7 Length Attributes

The length attributes are

```

l = <Real> !
l_chord      ! Chord length of a bend. Dependent attribute.

```

The length `l` is the path length of the reference particle. The one exception is that for an `Rbend` the length `l` is the chord length (§2.4). Internally, *Bmad* converts all `Rbends` to `Sbends` and stores the chord length under the `l_chord` attribute.

Note that for `Wigglers` the length `l` is not the same as the path length for a particle with the reference energy starting on the reference orbit.

## 4.8 Is\_on Attribute

The `is_on` attribute

```
is_on = <Logical>
```

is used to turn an element off. Turning an element off essentially converts it into a drift. Example

```

q1: quad, l = 0.6, k1 = 0.95
q1[is_on] = False

```

`is_on` does not affect any apertures that are set. Additionally, `is_on` does not affect the reference orbit. Therefore, turning off an `lcavity` will not affect the reference energy.

## 4.9 Multipole Attributes: An, Bn, KnL, Tn

A `Multipole` element specifies its multipole components using an Amplitude (`KnL`) and a tilt (`Tn`)

```

KnL = <Real>
Tn = <Real> ! Default is  $\pi/(2n + 2)$ 

```

`AB_Multipole` and all other elements that have multipole attributes specify the multipoles using normal (`Bn`) and skew (`An`) components

```

An = <Real>
Bn = <Real>

```

Here `n` ranges from 0 (dipole component) through 20. Example:

```
q1: quadrupole, b0 = 0.12, a20 = 1e7, radius = 0.045
```

Multipole formulas are given in §10.2. Note that for `Multipole` and `AB_multipole` (but not any other element) a non-zero dipole component will affect the reference orbit (just like a normal dipole will).

The `Tn` tilt component without a value takes a default of  $\pi/(2n + 2)$  which makes the component **skew**. Example:

```
m: multipole, k1l = 0.45, t1 ! Skew quadrupole
```

For everything other than a `Multipole` and `AB_multipole`, the multipole strength is scaled by a factor  $F r_0^{n_{\text{ref}}}/r_0^n$  (cf. Eq. (10.15)) where  $F$  is the strength of the element (for example  $F$  is  $K1 \cdot L$  for a quadrupole), and  $r_0$  is the “measurement radius” and is set by the `radius` attribute. The default value of  $r_0$ , if the `radius` is not given, is 1.0.

## 4.10 Instrumental Measurement Attributes

**Instrument**, **Monitor**, and **Marker** elements have special attributes to describe orbit, betatron phase, dispersion and coupling measurements. These attributes are:

<i>Attribute</i>		<i>! Symbol (§10.10)</i>	
<code>tilt</code>	= <Real>	$\theta_t$	See §4.4
<code>x_offset</code>	= <Real>	$x_{\text{err}}$	See §4.4
<code>y_offset</code>	= <Real>	$y_{\text{err}}$	See §4.4
<code>x_gain_err</code>	= <Real>	$dg_{x,\text{err}}$	Horizontal gain error
<code>y_gain_err</code>	= <Real>	$dg_{y,\text{err}}$	Vertical gain error
<code>crunch</code>	= <Real>	$\psi_{\text{err}}$	Crunch angle
<code>tilt_calib</code>	= <Real>	$\theta_{\text{err}}$	tilt angle calibration
<code>x_offset_calib</code>	= <Real>	$x_{\text{cal}}$	Horizontal offset calibration
<code>y_offset_calib</code>	= <Real>	$y_{\text{cal}}$	Vertical offset calibration
<code>x_gain_calib</code>	= <Real>	$dg_{x,\text{cal}}$	Horizontal gain calibration
<code>y_gain_calib</code>	= <Real>	$dg_{y,\text{cal}}$	Vertical gain calibration
<code>crunch_calib</code>	= <Real>	$\psi_{\text{cal}}$	Crunch angle calibration
<code>noise</code>	= <Real>	$n_f$	Noise factor
<code>de_eta_meas</code>	= <Real>	$dE/E$	Percent change in energy
<code>n_sample</code>	= <Real>	$N_s$	Number of sampling points
<code>osc_amplitude</code>	= <Real>	$A_{\text{osc}}$	Oscillation amplitude

A program can use these quantities to calculate “measured” values from the “laboratory” values. Here, “laboratory” means as calculated from some model lattice. See §10.10 for the conversion formulas.

## Chapter 5

# Tracking and Transfer Matrix Calculation Methods

Typically, there are several ways to do tracking and transfer matrix calculations for a given element class within *Bmad*. What method is used is selected on an element-by-element basis using the attributes

```
tracking_method = Switch    ! Method for tracking.  
mat6_calc_method = Switch  ! Method for 6x6 transfer matrix calculation.
```

These switches are discussed in more detail in sections §5.1 and §5.2. Example:

```
q2: quadrupole, tracking_method = boris  
q2[tracking_method] = boris  
quadrupole[tracking_method] = boris
```

The first two lines of this example have exactly the same effect in terms of setting the `tracking_method`. The third line shows how to set the `tracking_method` for an entire class of elements.

Two element attributes that can affect the way the transfer matrix is calculated are `symplectify` and `map_with_offsets`. These are discussed in §5.4 and §5.5 respectively.

### 5.1 tracking\_method Switches

The `tracking_method` attribute of an element sets the algorithm that is used for single particle tracking through that element. A table giving which methods are available with which element classes is given in Table 5.1.

A note on terminology: Adaptive step size control used with the `adaptive_boris` and `Runge_Kutta` integrators means that instead of taking fixed step sizes the integrator chooses the proper step size so that the error in the tracking is below the maximum allowable error set by `rel_tol` and `abs_tol` tolerances. The advantage of step size control is that the integrator uses a smaller step size when needed (the fields are rapidly varying), but makes larger steps when it can. The disadvantage is that a step is more computationally intensive since the error in a step is estimated by repeating a step using two mini steps. If the fields are rather uniform and you know what a good step size is you can save time by using a fixed step size.

`adaptive_boris` Second order Boris integration[17] with adaptive step size control. This should be nearly symplectic but slow.

**Boris!tracking method** Second order Boris Integration[17]. Like `Runge_Kutta`, `Boris` does tracking by integrating the equation of motion. `Boris` handles both electric and magnetic fields and does not assume that the particle is ultra-relativistic. `Boris` preserves conserved quantities more accurately than `Runge_Kutta`.

**Bmad\_Standard!tracking method** Uses formulas for tracking. The emphasis here is on speed. Note: If an element has non-zero multipole values, `Bmad_Standard` tracking will generally put half the multipole kick at the beginning of the element and half at the end. This is generally a good approximation but in it can result in differences between this and tracking methods like `Symp_Lie_PTC` which model multipoles as distributed evenly throughout an element.

**Custom!tracking method** This method will call a routine `track1_custom` which must be supplied by the programmer implementing the custom tracking. The default `track1_custom` supplied with the *Bmad* release will print an error message and stop the program if it is called which probably indicates a program linking problem.

**Linear** Linear just uses the 0th order vector with the 1st order 6x6 transfer matrix for an element. Very simple. Depending upon how the transfer matrix was generated this may or may not be symplectic.

**MAD** This uses the MAD 2nd order transfer map.

**None!tracking method** This prevents the transfer matrix from being recomputed. Using `None` in the input file is generally not a good idea since it prevents the matrix from being computed in the first place. Typically `None` is used internally in a program to prevent recomputation.

**runge\_kutta!tracking method** This uses a 4<sup>th</sup> order Runge Kutta integration algorithm with adaptive step size control. This is essentially the `ODEINT` subroutine from Numerical Recipes[18]. This may be slow but it should be accurate. This method is non-symplectic.

**Symp\_Lie\_Bmad!tracking method** Symplectic tracking using a Hamiltonian with Lie operation techniques. This is similar to `Symp_Lie_PTC` (see below) except this uses a *Bmad* routine. By bypassing some of the generality inherent in PTC's routines `Symp_Lie_Bmad` achieves about a factor of 10 improvement in speed over `Symp_Lie_PTC`. However, `Symp_Lie_Bmad` is currently only implemented for Wigglers.

**Symp\_Lie\_PTC** Symplectic tracking using a Hamiltonian with Lie operator techniques. This uses Etienne's PTC software for the calculation. This method is symplectic but can be slow.

**Symp\_Map** This uses a partially inverted, implicit Taylor map. The calculation uses Etienne's PTC software. Since the map is implicit, a Newton search method must be used. This will slow things down from the Taylor method but this is guaranteed symplectic. Note: Due to memory limitations in PTC, the number of elements using `symp_map` is limited to be of order 50.

**Taylor** This uses a Taylor map generated from Etienne's PTC package. Generating the map may take time but once you have it it should be very fast. One possible problem with using a Taylor map is that you have to worry about the accuracy if you do tracking at points that are far from the expansion point about which the map was made. This method is non-symplectic away from the expansion point. Whether the Taylor map is generated taking into account the offset an element has is governed by the `map_with_offsets` attribute (§5.5). Note: Taylor maps for `lcavity`, `match`, and `patch` elements are limited to first order.



<i>Element Class</i>	Adaptive_Boris	Bmad_Standard	Boris	Custom	Linear	MAD	Runge_Kutta	Symp_Lie_Bmad	Symp_Lie_PTC	Symp_Map	Taylor
ab_multipole		D		X	X				X	X	X
beambeam		D		X	X						
bend_sol_quad				X				D			
custom	X		X	D	X		X				
drift	X	D	X	X	X	X	X		X	X	X
ecollimator	X	D	X	X	X		X		X	X	X
elseparator	X	D	X	X	X	X	X		X	X	X
hkicker	X	D	X	X	X		X		X	X	X
instrument	X	D	X	X	X		X		X	X	X
kicker	X	D	X	X	X		X		X	X	X
lcavity		D		X	X						*
marker		D		X	X				X	X	X
match		D		X							*
monitor	X	D	X	X	X		X		X	X	X
multipole		D		X	X				X	X	X
octupole	X	D	X	X	X		X		X	X	X
patch		D		X							*
quadrupole	X	D	X	X	X	X	X		X	X	X
rbend		D		X	X	X			X	X	X
rcollimator	X	D	X	X	X		X		X	X	X
rfcavity		D		X	X	X			X	X	X
sbend		D		X	X	X			X	X	X
sextupole	X	D	X	X	X	X	X		X	X	X
solenoid	X	D	X	X	X	X	X		X	X	X
sol_quad	X	D	X	X	X	X	X		X	X	X
taylor		D		X	X						
vkicker	X	D	X	X	X		X		X	X	X
wiggler (map type)	X	D	X	X	X		X	X	X	X	X
wiggler (periodic type)		D		X	X		X <sup>a</sup>	X <sup>a</sup>	X <sup>a</sup>	X <sup>a</sup>	X <sup>a</sup>

<sup>a</sup>See §10.6 for more details

Table 5.1: Table of available **tracking\_method** switches for a given element class. “D” denotes the default method. “X” denotes an available method. “\*” denotes that the Taylor map will only be first order.

## 5.2 `mat6_calc_method` Switches

The `mat6_calc_method` attribute sets how the 6x6 Jacobian transfer matrix for a given element is computed. A table giving which methods are available with which element classes is give in Table 5.2.

For methods that do not necessarily produce a symplectic matrix the `symplectify` attribute of an element can be set to `True` to solve the problem. See §10.4.

Symplectic integration is like ordinary integration of a function  $f(x)$  but what is integrated here is a Taylor map. Truncating the map to  $0^{th}$  order gives the particle trajectory and truncating to  $1^{st}$  order gives the transfer matrix (Jacobian). The order at which a Taylor series is truncated at is set by `taylor_order` (see §7.1. Like ordinary integration there are various formulas that one can use to do symplectic integration. In *Bmad* (or more precisely Etienne’s PTC) you can use one of 3 methods. This is set by `integrator_order`. `integrator_order = n` where  $n$  is allowed by PTC to be 2, 4, or 6. With an integration order of  $n$  the error in an integration step scales as  $dz^n$  where  $dz$  is step size. The step size  $dz$  is set by the length of the element and the value of `ds_step`. Remember, as in ordinary integration, higher integration order does not necessarily imply higher accuracy.

**Bmad\_Standard** Uses formulas for the calculation. The emphasis here is on speed

**Custom** This method will call a routine `make_mat6_custom` which must be supplied by the programmer implementing the custom transfer matrix calculation. The default `make_mat6_custom` supplied with the *Bmad* release will print an error message and stop the program if it is called which probably indicates a program linking problem.

**MAD** This uses the MAD 2nd transfer map.

**None** This prevents the transfer matrix from being recomputed. Using `None` in the input file is generally not a good idea since it prevents the matrix from being computed in the first place. Typically `None` is used internally in a program to prevent recomputation.

**Symp\_Lie\_Bmad** A Symplectic calculation using a Hamiltonian with Lie operator techniques. This is similar to `Symp_Lie_PTC` (see below) except this uses a *Bmad* routine. By bypassing some of the generality inherent in PTC’s routines `Symp_Lie_Bmad` achieves about a factor of 10 improvement in speed over `Symp_Lie_PTC`. However, `Symp_Lie_Bmad` is currently only implemented for Wigglers.

**Symp\_Lie\_PTC** Symplectic integration using a Hamiltonian and Lie operators. This uses Etienne’s PTC software for the calculation. This method is symplectic but can be slow.

**Taylor** This uses a Taylor map generated from Etienne’s PTC package. Generating the map may take time but once you have it it should be very fast. One possible problem with using a Taylor map is that you have to worry about the accuracy if you do a calculation at points that are far from the expansion point about which the map was made. This method is non-symplectic away from the expansion point. Whether the Taylor map is generated taking into account the offset an element has is governed by the `map_with_offsets` attribute (§5.5). Note: Taylor maps for `lcavity`, `match`, and `patch` elements are limited to first order.

**Tracking** This uses the tracking method set by `tracking_method` to track 6 particles around the central orbit. This method is susceptible to inaccuracies caused by nonlinearities. Furthermore this method is almost surely slow. While non-symplectic, the advantage of this method is that it is directly related to any tracking results.

	Bmad_Standard	Custom	MAD	None	Symp_Lie_Bmad	Symp_Lie_PTC	Taylor	Tracking
ab_multipole	D	X		X		X	X	
beambeam	D	X		X				X
bend_sol_quad		X		X	D			X
custom		D		X				X
drift	D	X	X	X		X	X	X
ecollimator	D	X		X		X	X	X
elseparator	D	X	X	X		X	X	X
hkicker	D	X		X		X	X	X
instrument	D	X		X		X	X	X
kicker	D	X		X		X	X	X
lcavity	D	X		X				X
marker	D	X		X		X	X	X
match	D	X		X				
monitor	D	X		X		X	X	X
multipole	D	X		X		X	X	
octupole	D	X		X		X	X	X
patch	D	X		X				
quadrupole	D	X	X	X		X	X	X
rbend	D	X	X	X		X	X	X
rcollimator	D	X		X		X	X	X
rfcavity	D	X	X	X		X	X	X
sbend	D	X	X	X		X	X	X
sextupole	D	X	X	X		X	X	X
solenoid	D	X	X	X		X	X	X
sol_quad	D	X	X	X		X	X	X
taylor	D	X		X				
vkicker	D	X		X		X	X	X
wiggler (map type)	D	X		X	X	X	X	X
wiggler (periodic type)	D	X		X	X <sup>a</sup>	X <sup>a</sup>	X <sup>a</sup>	X

<sup>a</sup>See §10.6 for more details

Table 5.2: Table of available `mat6_calc_method` switches for a given element class. “D” denotes the default method. “X” denotes an available method.

### 5.3 Integration Methods

Integration methods are split into two classes: Those that involve Taylor maps and those that simply track a particle's position. The Taylor map methods are

```
symp_lie_bmad
symp_lie_ptc
taylor
```

See section §10.3 for more information on Taylor maps and symplectic integration. The methods that do not involve Taylor maps are

```
adaptive_boris
boris
runge_kutta
```

there are a number of attributes that can affect the calculation. They are

```
ds_step = <Real>           ! Integration step length
num_steps = <Integer>      ! Number of integration steps.
integrator_order = <Integer> ! Integrator order
rel_tol = <Real>           ! Relative tolerance
abs_tol = <Real>           ! Absolute tolerance
field_calc = Switch        ! How the field is calculated
```

Example:

```
q1: quadrupole, l = 0.6, tracking_method = bmad_standard, &
    mat6_calc_method = symp_lie_ptc, ds_step = 0.2, field_calc = custom
```

One way to create a transfer map through an element is to divide the element up into slices and then to propagate the transfer map slice by slice. There are several ways to do this integration. The `boris` and `runge_kutta` methods integrate the equations of motion to give the 0<sup>th</sup> order Taylor map which just represents a particle's orbit. Symplectic integration using Lie algebraic techniques, on the other hand, can generate Taylor maps to any order. The `ds_step` attribute determines the slice thickness. Alternatively, `num_steps` attribute can be used in place of `ds_step` to specify the number of slices. This is applicable to Boris, `symp_lie_bmad`, and `symp_lie_ptc` integration.

`integrator_order` is the order of the integration formula for `Symp_Lie_PTC`. Possible values are

```
integrator_order = 2 (default), 4, or 6
```

Essentially, an integration order of  $n$  means that the error in an integration step scales as  $dz^{n+1}$  where  $dz$  is the slice thickness. For a given number of steps a higher order will give more accurate results but a higher order integrator will take more time per step. It turns out that for wigglers, after adjusting `ds_step` for a given accuracy, the order 2 integrator is the fastest. This is not surprising given the highly nonlinear nature of a wiggler. Note that `symp_lie_bmad` always uses an order 2 integrator independent of the setting of `integrator_order`.

`adaptive_boris` and `runge_kutta` use adaptive step control independent of `ds_step`. These methods use the `rel_tol` and `abs_tol` attributes to try to keep the estimated error of the integration such that

```
error < abs_tol + |orbit| * rel_tol
```

lowering the error bounds makes for greater accuracy (as long as round-off doesn't hurt) but for slower tracking.

The `boris`, `adaptive_boris`, and `runge_kutta` tracking all use as input the electric and magnetic fields of an element. How the EM fields are calculated is determined by the `field_calc` attribute for an element. Possible values for `field_calc` are:

```
Bmad_Standard    ! This is the default
Custom
```

**Custom** means that the field calculations are done outside of the *Bmad* software. A program doing **Custom** field calculations will need the appropriate custom routine. See §17.10 for more details.

Both **boris** and **adaptive\_boris** tracking do not assume that a particle is relativistic so these tracking methods can be used with non-relativistic particles. The phase space coordinates used in these tracking methods are not the usual *Bmad* coordinates Rather what is used is

$$(x, p_x/p_0, y, p_y/p_0, s-ct, dE/(cP_0))$$

At high energy  $s - ct = z$  which is the distance of the particle from the reference particle and  $cP_0 = v E_0/C = E_0$  so that  $dE/cP_0 = dE/E$  giving the standard *Bmad* coordinates.

## 5.4 Symplectify Attribute

The **symplectify** attribute

```
symplectify = <Logical>
```

is used to make the transfer matrix for an element symplectic. The linear transport matrix may be non-symplectic for a number of reasons. For example, the linear matrix that comes from expanding a Taylor Map around any point that is not the origin of the map is generally not symplectic. The transfer matrix for an element can be symplectified by setting the **symplectify** attribute to **True**. See section 10.4 for details on how a matrix is symplectified. The default value of **symplectify**, if it is not present, is **False**. If it is present without a value then it defaults to **true**. Examples:

```
s1: sextupole, l = 0.34                ! symplectify = False
s1: sextupole, symplectify = True, l = 0.34 ! symplectify = True
s1: sextupole, symplectify, l = 0.34      ! symplectify = True
```

## 5.5 Map\_with\_offsets Attribute

The **map\_with\_offsets** attribute sets whether the Taylor map generated for an element includes the affect due to the elements (mis)orientation in space. That is, the affect of any pitches, offsets or tilt (§4.4). The default is **True** which means that the Taylor map will include such effects.

How **map\_with\_offsets** is set will not affect the results of tracking or the Jacobian matrix calculation. What is affected is the speed of the calculations. With **map\_with\_offsets** set to **True** the Taylor map will have to be recalculated each time an element is reoriented in space. On the other hand, with **map\_with\_offsets** set to **False** each tracking and Jacobian matrix calculation will include the extra computation involving the effect of the orientation. Thus if an element's orientation is fixed it is faster to set **map\_with\_offsets** to **True** and if the orientation is varying it is faster to set **map\_with\_offsets** to **False**.

If the global parameter **bmad\_com%conserve\_taylor\_maps** (§8.1) is set to **True** (the default), then, if an element is offset within a program, and if **map\_with\_offset** is set to **True** for that element, *Bmad* will toggle **map\_with\_offset** to **False** to conserve the map.



## Chapter 6

# Beam Lines, Replacement Lists, and Branching

### 6.1 Use Statement

To *Bmad*, a “lattice” is the sequence of physical elements that is to be studied. The lattice is constructed in the input lattice file using what are known as Beam Lines and Replacement Lists. Beam Lines are further subdivided into Lines with and without replacement arguments. This essentially corresponds to the *MAD* definition of Lines and Lists. There can be multiple Beam Lines and Replacement Lists defined in a lattice file and Lines and Lists can be nested inside other Lines and Lists. The particular Line that defines the lattice to be analyzed by *Bmad* is selected by the `use` statement. For example

```
use, my_line
```

would pick the Line `my_line` for analysis. If there are multiple `use` statements in a lattice file, the last `use` statement defines which line to use.

*Bmad* associates a number with each element starting at 1 for the first element in the lattice, 2 for the second element, etc. Additionally, *Bmad* always automatically creates a 0<sup>th</sup> element to mark the beginning of the lattice. The name of this element is always `BEGINNING`.

### 6.2 Beam Lines without Arguments

A Beam Line without arguments has the format

```
label: line = (member1, member2, ...)
```

where `member1`, `member2`, etc. are either elements, other Beam Lines or Replacement Lists, or sublines enclosed in parentheses. Example:

```
line1: line = (a, b, c)
line2: line = (d, line1, e)
use, line2
```

This example shows how an Line member can refer to another Beam Line. This is helpful if the same sequence of elements appears repeatedly in the lattice. When `line2` is expanded to form the lattice the definition of `line1` will be inserted in to produce the following lattice for analysis

```
(d, a, b, c, e)
```

Note: In the expanded lattice, any `Null_Ele` type elements (§2.21) will be discarded. For example, if element `b` in the above example is a `Null_Ele` then the actual expanded lattice will be:

```
(d, a, c, e)
```

A member that is a Line or List can be reflected (elements taken in reverse order) if a negative sign is put in front of it. For example:

```
line1: line = (a, b, c)
line2: line = (d, -line1, e)
```

`line2` when expanded gives

```
(d, c, b, a, e)
```

Reflecting a subline will also reflect any sublines of the subline. For example:

```
line0: line = (y, z)
line1: line = (line0, b, c)
line2: line = (d, -line1, e)
```

`line2` when expanded gives

```
(d, c, b, z, y, e)
```

A repetition count, which is an integer followed by an asterisk, means that the member is repeated. For example

```
line1: line = (a, b, c)
line2: line = (d, 2*line1, e)
```

`line2` when expanded gives

```
(d, a, b, c, a, b, c, e)
```

Repetition count can be combined with reflection. For example

```
line1: line = (a, b, c)
line2: line = (d, -2*line1, e)
```

`line2` when expanded gives

```
(d, c, b, a, c, b, a, e)
```

Instead of the name of a line, subline members can also be given as an explicit list using parentheses. For example, the previous example could be rewritten as

```
line2: line = (d, -2*(a, b, c), e)
```

A line can have the `multipass` attribute. This is covered in §3.4.

## 6.3 Beam Lines with Replaceable Arguments

Beam lines can have an argument list using the following syntax

```
line_name(dummy_arg1, dummy_arg2, ...): LINE = (member1, member2, ...)
```

The dummy arguments are replaced by the actual arguments when the Line is used elsewhere. For example:

```
line1(DA1, DA2): line = (a, DA2, b, DA1)
line2: line = (h, line1(y, z), g)
```

When `line2` is expanded the actual arguments of `line1`, in this case `(y, z)`, replaces the dummy arguments `(DA1, DA2)` to give for `line2`

```
h, a, z, b, y, g
```

Unlike *MAD*, Beam Line actual arguments can only be elements or Beam Lines. Thus the following is not allowed

```
line2: line = (h, line1(2*y, z), g)    ! NO: 2*y NOT allowed as an argument.
```



## 6.4 Replacement Lists

When a lattice is expanded, all the lattice members that correspond to a name of a Replacement List are replaced successively, by the members in the Replacement List. The general syntax is

```
label: LIST = (member1, member2, ...)
```

For example:

```
list1: list = (a, b, c)
line1: line = (z1, list1, z2, list1, z3, list1, z4, list1)
use, line1
```

When the lattice is expanded the first instance of `list1` in `line1` is replaced by `a` (which is the first element of `list1`), the second instance of `list1` is replaced by `b`, etc. If there are more instances of `list1` in the lattice then members of `list1`, the replacement starts at the beginning of `list1` after the last member of `list1` is used. In this case the lattice would be:

```
z1, a, z2, b, z3, c, z4, a
```

Unlike *MAD* members of a replacement list can only be simple elements without reflection or repetition count and not other Lines or Lists. For example the following is not allowed:

```
list1: list = (2*a, b) ! NO: No repetition count allowed.
```

## 6.5 Line and List Tags

When a lattice has repeating lines, it can be desirable to differentiate between repeated elements. This can be done by tagging lines with a *tag*. An example will make this clear:

```
line1: line = (a, b)
line2: line = (line1, line1)
use, line2
```

When expanded the lattice would be:

```
a, b, a, b
```

The first and third elements have the same name “a” and the second and fourth elements have the same name “b”. Using tags the lattice elements can be given unique names. lines or lists are tagged using brackets [...]. The general syntax is:

```
line_name[tag_name]           ! Syntax for lines
list_name[tag_name]           ! Syntax for lists
replacement_line[tag_name](arg1, arg2, ...) ! Syntax for replacement lines.
```

Thus to differentiate the lattice elements in the above example `line2` needs to be modified using tags:

```
line1: line = (a, b)
line2: line = (line1[t1], line1[t2])
use, line2
```

In this case the lattice elements will have names of the form:

```
tag_name.element_name
```

In this particular example, the lattice with tagging will be:

```
t1.a, t1.b, t2.a, t2.b
```

Of course with this simple example one could have just as easily not used tags:

```
t1.a: a;   t2.a: a
t1.b: b;   t2.b: b
line1: line = (t1.a, t1.b, t2.a, t2.b)
use, line2
```

But in more complicated situations tagging can make for compact lattice files.

When lines are nested, the name of an element is formed by concatenating the tags together with dots in between in the form:

```
tag_name1.tag_name2. ... tag_name_n.element_name
```

An example will make this clear:

```
list1 = (g, h)
line1(y, z) = (a, b)
line2: line = (line1[t1](a, b))
line3: line = (line2, list1[hh])
line4: line = (line3[z1], line3[z2])
use, line4
```

The lattice elements in this case are:

```
z1.t1.a, z1.t1.b, z1.hh.g, z2.t1.a, z2.t1.b, z1.hh.h
```

To modify a particular tagged element the lattice must be expanded first (§7.8). For example:

```
line1: line = (a, b)
line2: line = (line1[t1], line1[t2])
use, line2
expand_lattice
t1.b[k1] = 1.37
b[k1] = 0.63          ! This statement does not have any effect
```

After the lattice has been expanded there is no connection between the original **a** and **b** elements and the elements in the lattice like **t1.b**. Thus the last line in the example where the **k1** attribute of **b** is modified do not have any effect on the lattice elements.

## 6.6 Branching

Lines that branch off from the main lattice can be defined in *Bmad*. There are two types of branch elements. **Branch** elements define where the particle beam can branch off, say to a beam dump. **Photon\_Branch** elements define the source point for X-ray beams. Example:

```
erl: line = (... , b_mark, ...)          ! Define the main lattice with a branch
use, erl
b_mark: branch, to = b_line, &           ! Define the branch point
        direction = -1, alias = dump
b_line: line = (... , q3d, ...)          ! Define the branch line
```

Branch lines can themselves have branch elements. A branch line always starts out tangential to the line it is branching from. The **direction** attribute of the branch element indicates whether the branch line is outgoing in the forward direction (**direction** = +1) or incoming (**direction** = -1).

Branches are named after the setting of the **alias** attribute of the branch element. In the above example, the branch line would be named **DUMP**. If the **alias** attribute is not set, the name of the branch line (**B\_LINE** in the above example) is used instead. Like the main lattice, *Bmad* always automatically creates a zeroth element at the beginning of the branch called **BEGINNING**. The longitudinal **s** position of an element in a branch is determined by the distance from the beginning of the branch.

## Chapter 7

# Miscellaneous Statements

This chapter deals with the statements not covered in the previous chapters.

### 7.1 Parameter Statement

The `parameter` statement is used to set the `lattice` name and other variables. The variables that can be set by `parameter` is

```
parameter[lattice]      = <String>      ! Lattice name
parameter[lattice_type] = <Switch>      !
parameter[taylor_order] = <Integer>     ! Default: 3
parameter[e_tot]        = <Real>        ! Reference total Energy.
                                !      Default: 1000 * rest_energy.
parameter[p0c]          = <Real>        ! Reference momentum.
parameter[n_part]       = <Real>        ! Number of particles in a bunch.
parameter[ran_seed]     = <Integer>     ! Random number generator init.
parameter[particle]     = <particle_type> ! Eg: positron, antiproton, etc.
parameter[aperture_limit_on] = <Logical> ! Use aperture limits in tracking.
```

Examples

```
parameter[lattice]      = "L9A19C501.FD93S_4S_15KG"
parameter[lattice_type] = circular_lattice
parameter[taylor_order] = 5
parameter[E_tot]        = 5.6e9      ! eV
```

For more information on `parameter[ran_seed]` see §1.10.

Valid particle switches are:

```
positron ! default
electron
proton
antiproton
```

The `parameter[e_tot]` and `parameter[p0c]` are the reference total energy and momentum at the start of the lattice. Each element in a lattice has an individual reference `e_tot` and `p0c` attributes which are dependent parameters. The reference energy and momentum will only change between `LCavity` or `Patch` elements. The starting reference energy, if not set, will be set to 1000 time the particle rest energy. Note: `beginning[e_tot]` and `beginning[p0c]` are equivalent to `parameter[e_tot]` and `parameter[p0c]`.

The `parameter[n_part]` is the number of particle in a bunch. it is used with `BeamBeam` elements and is used to calculate the change in energy through an `Lcavity`. See §2.16 for more details.

Aperture limits may be set for elements in the lattice (§4.6). Setting `aperture_limit_on` to `False` will disable all set apertures. `True` is the default.

The `lattice` name is stored by *Bmad* for use by a program but it does not otherwise effect any *Bmad* routines. Historically it is possible to set the lattice name using the syntax

```
lattice = <String>    ! DO NOT USE THIS SYNTAX
```

This syntax is obsolete since a typographical error cannot be caught.

Valid `lattice_type` switches are

```
circular_lattice    ! Default w/o LCavity element present.
linear_lattice      ! Default if LCavity elements present.
```

a `circular_lattice` is for a closed lat where one expects a periodic solution for the Twiss parameters. A `linear_lattice` is not closed so that the initial Twiss parameters need to be given, not computed. Although `circular_lattice` is the nominal default, If there are `Lcavity` elements present, `linear_lattice` will be used as the default.

The Taylor order (§10.3) is set by `parameter[taylor_order]` and is the maximum order for a Taylor map. Historically it is possible to set the Taylor order using the syntax

```
taylor_order = <Integer>    ! DO NOT USE THIS SYNTAX
```

This syntax is obsolete since a typographical error is not easily caught.

## 7.2 Beam\_start Statement

The `beam_start` statement is used to set the starting coordinates for particle tracking

```
beam_start[x]      = <Real> ! Horizontal position.
beam_start[px]     = <Real> ! Horizontal momentum.
beam_start[y]      = <Real> ! Vertical position.
beam_start[py]     = <Real> ! Vertical momentum.
beam_start[z]      = <Real> ! Longitudinal position.
beam_start[pz]     = <Real> ! Longitudinal momentum (energy deviation).
beam_start[emittance_a] = <Real> ! A-mode emittance
beam_start[emittance_b] = <Real> ! B-mode emittance
beam_start[emittance_z] = <Real> ! Z-mode emittance
```

Examples

```
beam_start[y] = 2 * beam_start[x]
```

## 7.3 Beam Statement

The `beam` statement is provided for compatibility with *MAD*. The syntax is

```
beam, energy = GeV, pc = GeV, particle = <Switch>, n_part = <Real>
```

For example

```
beam, energy = 5.6    ! Note: GeV to be compatible with MAD
beam, particle = electron, n_part = 1.6e10
```

Setting the reference energy using the `energy` attribute is the same as using `parameter[e_tot]`. Similarly, setting `pc` is equivalent to setting `parameter[p0c]`. Valid `particle` switches are the same as `parameter[particle]`.

## 7.4 Beginning Statement

For non-circular lattices the `beginning` statement can be used to set the Twiss parameters and beam energy at the beginning of the lat

```
beginning[beta_a] = <Real> ! "a" mode beta
beginning[alpha_a] = <Real> ! "a" mode alpha
beginning[phi_a] = <Real> ! "a" mode phase
beginning[eta_x] = <Real> ! x-axis dispersion
beginning[etap_x] = <Real> ! x-axis dispersion derivative.
beginning[beta_b] = <Real> ! "b" mode beta
beginning[alpha_b] = <Real> ! "b" mode alpha
beginning[phi_b] = <Real> ! "b" mode phase
beginning[eta_y] = <Real> ! y-axis dispersion
beginning[etap_y] = <Real> ! y-axis dispersion derivative.
beginning[cmat_ij] = <Real> ! C coupling matrix. i, j = '1', or '2'
beginning[s] = <Real> ! Longitudinal starting position.
beginning[e_tot] = <Real> ! Reference total energy in eV.
beginning[p0c] = <Real> ! Reference momentum in eV.
```

The `gamma_a`, `gamma_b`, and `gamma_c` (the coupling gamma factor) will be kept consistent with the values set. If not set the default values are all zero. `beginning[e_tot]` and `parameter[e_tot]` are equivalent and one or the other may be set but not both. Similarly, `beginning[p0c]` and `parameter[p0c]` are equivalent.

For any lattice the `beginning` statement can be used to set the starting floor position (see 9.2). The syntax is

```
beginning[x_position] = <Real> ! X position
beginning[y_position] = <Real> ! Y position
beginning[z_position] = <Real> ! Z position
beginning[theta_position] = <Real> ! Angle on floor
beginning[phi_position] = <Real> ! Angle of attack
beginning[psi_position] = <Real> ! Roll angle
```

## 7.5 Title Statement

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

```
title, <String>
```

or the statement can be split into two lines

```
title
<String>
```

For example

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

## 7.6 Call Statement

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

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

Example:

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

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

For filenames that are relative, the called file will be searched for in two different locations:

- 1) Relative to the directory of the calling file.
- 2) Relative to the current directory.

The first instance where a file is found is used. Thus, in the above example, the first call will search for:

- 2) ../layout/my\_layout.bmad (relative to the calling file directory)
- 1) ../layout/my\_layout.bmad (relative to the current directory)

An XSIF (§1.1) lattice file may be called from within an *Bmad* lattice file by prepending "xsif::" to the file name. Example:

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

This statement must be the first statement in the *Bmad* lattice file except for any `no_digested`, `parser_debug`, or `no_superimpose` statements. The XSIF lattice file must define a complete lattice and cannot contain any *Bmad* specific statements. The call to the XSIF file automatically expands the lattice (§7.8) and any additional statements in the *Bmad* lattice file operate on the expanded lattice.

## 7.7 Return and End\_File statements

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

## 7.8 Expand\_lattice Statement

At some point in parsing a lattice file, the ordered sequence of elements that form a lattice must be constructed. This process is called `lattice expansion` since the element sequence can be built up from sub-sequences (§6). Normally lattice expansion happens automatically at the end of the parsing of the lattice file but an explicit `expand_lattice` statement in a lattice file will cause immediate expansion. The reason why this can be important is that there are restrictions, on some types of operations which must come either before or after lattice expansion:

- The `ran` and `ran_gauss` functions, when used with elements that show up multiple times in a lattice, generally need to be used after lattice expansion. See §1.10.
- Some dependent variables may be set as if they are independent variables but only if done before lattice expansion. See §4.1.
- Setting the `dphi0` attribute for an `Lcavity` or `RFcavity` multipass slave may only be done after lattice expansion (§3.4).
- Setting individual element attributes for tagged elements can only be done after lattice expansion (§6.5).

## 7.9 No\_superimpose Statement

The `no_superimpose` statement is used to suppress superpositions (§3.3). This is useful for debugging purposes.

## 7.10 Debugging Statements

There are two statements, `parser_debug` and `no_digested`, which can help in debugging the *Bmad* lattice parser itself. That is, these statements are generally only used by programmers.

The `no_digested` statement if present, will prevent *Bmad* from creating a digested file.

The `parser_debug` statement will cause information about the lattice to be printed out at the terminal. It is recommended that this statement be used with small test lattices since it can generate a lot of output. The syntax is

```
parser_debug <switches>
```

Valid <switches> are

```
beam_start      ! Print the beam_start information.
ele <n1> <n2> ... ! Print full info on selected elements.
lattice         ! Print a list of lattice element information.
lord            ! Print full information on all lord elements.
seq            ! Print sequence information.
slave          ! Print full information on all slave elements.
var            ! Print variable information.
```

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

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





## Chapter 8

# Bmad Parameter Structures

*Bmad* has various parameters which affect various calculations that *Bmad* performs. A given program may give the user access to some of these parameters so, in order to allow the intelligent setting of these parameters, this chapter gives an in-depth description.

A set of parameters are grouped that affect a particular type of calculation are grouped into “**structures**”. Each structure has a “**structure name**” (also called a “**type name**”) which identifies the list of parameters in the structure. Additionally, there will be an “**instance name**” which is what the user uses to refer to this **structure**. For global parameters there will be a unique instance name. For non-global parameters the instance name will be program specific. It is possible to have multiple instance names. For example, in the situation where a program is simulating multiple particle beams, there could be multiple `beam_init_struct` (§8.2) instances. To refer to a particular parameter use the syntax

```
instance_name%parameter_name
```

For example, To refer to the `max_aperture_limit` parameter in Section §8.1 the syntax is

```
bmad_com%max_aperture_limit
```

## 8.1 Bmad Global Parameters

Some overall parameters are stored in the `bmad_common_struct` structure. The instance name here is `bmad_com`. The parameters of this structure are:

```
type bmad_common_struct
  real(rp) :: max_aperture_limit = 1e3      ! Max Aperture.
  real(rp) :: d_orb(6)           = 1e-5     ! for the make_mat6_tracking routine.
  real(rp) :: grad_loss_sr_wake  = 0        ! Internal var for Cavities.
  real(rp) :: default_ds_step    = 0.2      ! Integration step size.
  real(rp) :: significant_longitudinal_length = 1e-10 ! meter
  real(rp) :: rel_tolerance      = 1e-5     ! Runge-Kutta: Relative tolerance.
  real(rp) :: abs_tolerance      = 1e-8     ! Runge-Kutta: Absolute tolerance.
  real(rp) :: rel_tol_adaptive_tracking = 1e-6 ! Tracking relative tolerance.
  real(rp) :: abs_tol_adaptive_tracking = 1e-7 ! Tracking absolute tolerance.
  integer :: taylor_order = 3              ! 3rd order is default
  integer :: default_integ_order = 2       ! PTC integration order
  logical :: sr_wakes_on = .true.         ! Short range wake fields?
  logical :: lr_wakes_on = .true.         ! Long range wake fields
```

```

logical :: mat6_track_symmetric = .true.    ! symmetric offsets
logical :: auto_bookkeeper = .true.        ! Automatic bookkeeping?
logical :: trans_space_charge_on = .false.  ! Space charge switch
logical :: coherent_synch_rad_on = .false.  ! csr
logical :: spin_tracking_on = .false.       ! spin tracking?
logical :: radiation_damping_on = .false.   ! Damping toggle.
logical :: radiation_fluctuations_on = .false. ! Fluctuations toggle.
logical :: compute_ref_energy = .true.      ! Enable recomputation?
logical :: conserve_taylor_maps = .true.    ! Enable bookkeeper to set
                                           ! ele%map_with_offsets = F?

end type

```

`max_aperture_limit` is the maximum amplitude a particle can have during tracking. If this amplitude is exceeded, the particle is lost even if there is no element aperture set. Having a maximum aperture limit helps prevent numerical overflow in the tracking calculations.

`d_orb` is the orbit displacement used in the routine that calculates the transfer matrix through an element via tracking.

## 8.2 Beam Initialization Parameters

Beams of particles are used for simulating inter-bunch intra-bunch effects. The `beam_init_struct` structure holds parameters which are used to initialize the beam. The parameters of this structure are:

```

type beam_init_struct
  character(16) distribution_type(3)          ! "ELLIPSE", "KV", "GRID", "" (default).
  type(ellipse_beam_init_struct) ellipse(3) ! For ellipse beam distribution
  type(kv_beam_init_struct) KV               ! For KV beam distribution
  type(grid_beam_init_struct) grid(3)        ! For grid beam distribution
  !!! The following are for Random distributions
  character(16) :: random_engine              ! "pseudo" (default) or "quasi".
  character(16) :: random_gauss_converter ! "exact" (default) or "limited".
  real(rp) :: random_sigma_cutoff = 4.0      ! Used with "limited" converter.
  real(rp) :: center_jitter(6) = 0.0        ! Bunch center rms jitter
  real(rp) :: emitt_jitter(2) = 0.0         ! %RMS a and b mode bunch emittance jitter
  real(rp) :: sig_z_jitter = 0.0            ! bunch length RMS jitter
  real(rp) :: sig_e_jitter = 0.0            ! energy spread RMS jitter
  integer :: n_particle = 0                 ! Number of simulated particles per bunch.
  logical :: renorm_center = .true.         ! Renormalize centroid?
  logical :: renorm_sigma = .true.         ! Renormalize sigma?
  !!! The following are used by all distribution types
  type(beam_spin_struct) spin              ! Spin
  real(rp) a_norm_emitt                    ! a-mode emittance
  real(rp) b_norm_emitt                    ! b-mode emittance
  real(rp) :: dPz_dz = 0                   ! Correlation of Pz with long position.
  real(rp) :: center(6) = 0                ! Bench center offset.
  real(rp) dt_bunch                        ! Time between bunches.
  real(rp) sig_z                           ! Z sigma in m.
  real(rp) sig_e                           ! dE/E (pz) sigma.
  real(rp) bunch_charge                    ! Charge in a bunch.
  integer :: n_bunch = 1                   ! Number of bunches.
  logical :: init_spin = .false.           ! initialize beam spinors

```

end type

The number of bunches in the beam is set by `n_bunch`. The `%distributeion_type(:)` array determines what algorithms are used to generate the particle distribution for a bunch. `%distributeion_type(1)` sets the distribution type for the  $(x, p_x)$  2D phase space, etc. Possibilities for `%distributeion_type(:)` are:

```
""           ! Random distribution (default).
"ELLIPSE"    ! Ellipse distribution (§10.11.1)
"KV"         ! Kapchinsky-Vladimirsky distribution (§10.11.2)
"GRID"       ! Uniform distribution.
```

Currently, if a random distribution is used, it must be used in all 2D phase spaces. Since the Kapchinsky-Vladimirsky distribution is for a 4D phase space, if the Kapchinsky-Vladimirsky distribution is used, "KV" must appear exactly twice in the `%distributeion_type(:)` array.

The parameters common to all the distribution types are marked in the `beam_init_struct` above. The parameters for the random distribution are also indicated. These random distribution parameters are:

#### `%random_engine`

This component sets the algorithm to use in generating a uniform distribution of random numbers in the interval  $[0, 1]$ . "pseudo" is a pseudo random number generator and "quasi" is a quasi random generator. "quasi random" is a misnomer in that the distribution generated is fairly uniform.

#### `%random_gauss_converter, %random_sigma_cutoff`

To generate Gaussian random numbers, a conversion algorithm from the flat distribution generated according to `%random_engine` is needed. `%random_gauss_converter` selects the algorithm. The "exact" conversion uses an exact conversion. The "limited" limits the maximum sigma generated to the value of `%random_sigma_cutoff`. The "limited" method is also somewhat faster than the "exact" method. Note that the sigma cutoff is ignored in the "exact" method.

#### `%n_paricle`

This component sets the number of particles generated per bunch.

#### `%renorm_center, %renorm_sigma`

If set to True, these components will ensure that the actual beam center and sigmas will correspond to the input values. Otherwise, there will be fluctuations due to the finite number of particles generated.

#### `%center_jitter, %emitt_jitter, %sig_z_jitter, %sig_e_jitter`

These components can be used to provide a bunch-to-bunch random variation in the emittance and bunch center.

The `%ellipse(:)` array sets the parameters for the `ellipse` distribution (§10.11.1). Each component of this array looks like

```
type ellipse_beam_init_struct
  integer part_per_ellipse ! number of particles per ellipse.
  integer n_ellipse       ! number of ellipses.
  real(rp) sigma_cutoff   ! sigma cutoff of the representation.
end type
```

The `%kv` component of the `beam_init_struct` sets the parameters for the Kapchinsky-Vladimirsky distribution (§10.11.2)

```
type kv_beam_init_struct
  integer part_per_phi(2) . ! number of particles per angle variable.
  integer n_I2             ! number of I2
  real(rp) A               ! A = I1/e
end type
```

The `%grid` component of the `beam_init_struct` sets the parameters for a uniformly spaced grid of particles.

```
type grid_beam_init_struct
  integer n_x      ! number of columns.
  integer n_px     ! number of rows.
  real(rp) x_min   ! Lower x limit.
  real(rp) x_max   ! Upper x limit.
  real(rp) px_min  ! Lower px limit.
  real(rp) px_max  ! Upper px limit.
end type
```

### 8.3 CSR Parameters

The Coherent Synchrotron Radiation (CSR) calculation is discussed in Section §10.16. Besides the parameters discussed below, the `coherent_synch_rad_on` parameter in Section §8.1 must be set True to enable the CSR calculation.

The CSR parameter structure has a `type` name of `csr_parameter_struct` and an instance name of `csr_param`. This structure has components

```
type csr_parameter_struct
  real(rp) :: ds_track_step = 0      ! Tracking step size
  real(rp) :: beam_chamber_height = 0 ! Used in shielding calculation.
  real(rp) :: sigma_cutoff = 0.1     ! Cutoff for the lsc calc. If a bin sigma
                                     ! is < cutoff * sigma_ave then ignore.

  integer :: n_bin = 0                ! Number of bins used
  integer :: particle_bin_span = 2    ! Longitudinal particle length / dz_bin
  integer :: n_shield_images = 0      ! Chamber wall shielding. 0 = no shielding.
  logical :: lcsr_component_on = .true. ! Longitudinal csr component
  logical :: lsc_component_on = .true. ! Longitudinal space charge component
  logical :: tsc_component_on = .true. ! Transverse space charge component
  logical :: small_angle_approx = .true. ! Use lcsr small angle approximation?
end type
```

The values for the various quantities shown above are their default values.

`ds_track_step` is the nominal longitudinal distance traveled by the bunch between CSR kicks. The actual distance between kicks within a lattice element is adjusted so that there is an integer number of steps from the element entrance to the element exit. This parameter must be set to something positive otherwise an error will result. Larger values will speed up the calculation at the expense of accuracy.

`beam_chamber_height` is the height of the beam chamber in meters. This parameter is used when shielding is taken into account. See also the description of the parameter `n_shield_images`.

`sigma_cutoff` is used in the longitudinal space charge (LSC) calculation and is used to prevent bins with only a few particles in them to give a large contribution to the kick when the computed transverse sigmas are abnormally low.

`n_bin` is the number of bins used. The bin width is dynamically adjusted at each kick point so that the bins will span the bunch length. This parameter must be set to something positive. Larger values will slow the calculation while smaller values will lead to inaccuracies and loss of resolution. `n_bin` should also not be set so large that the average number of particles in a bin is too small. “Typical” values are in the range 100 — 1000.

`particle_bin_span` is the width of a particle's triangular density distribution (cf. §10.16) in multiples of the bin width. A larger span will give better smoothing of the computed particle density with an attendant loss in resolution.

`n_shield_images` is the number of shielding current layers used in the shielding calculation. A value of zero results in no shielding. See also the description of the parameter `beam_chamber_height`. The proper setting of this parameter depends upon how strong the shielding is. Larger values give better accuracy at the expense of computation speed. “Typical” values are in the range 0 — 5.

`lcsr_component_on` toggles on or off the (longitudinal) CSR kick.

`lsc_component_on` toggles on or off the transverse space charge kick. Currently this calculation is not implemented so this parameter does not have any affect.

`small_angle_approx` toggles whether the small angle approximation is used in the calculation. This is generally an excellent approximation.



## Part II

# Conventions and Physics





## Chapter 9

# Coordinates

### 9.1 Reference Orbit

The **reference orbit** is the curved path used to define a coordinate system for particles as shown in Figure 9.1. At a given time  $t$ , a particle's position can be described by a point on the reference orbit a distance  $s$  relative to the reference orbit's zero position plus a transverse offset. This point on the reference orbit is used as the origin of the local  $(x, y, z)$  coordinate system with the  $z$ -axis tangent to the reference orbit and pointing in the direction of increasing  $s$ . The  $x$  and  $y$ -axes are perpendicular to the reference orbit. If the lattice has no vertical bends, the  $y$ -axis is in the vertical direction and the  $x$ -axis is in the horizontal plane. Notice that by construction, the particle is always at  $z = 0$ .

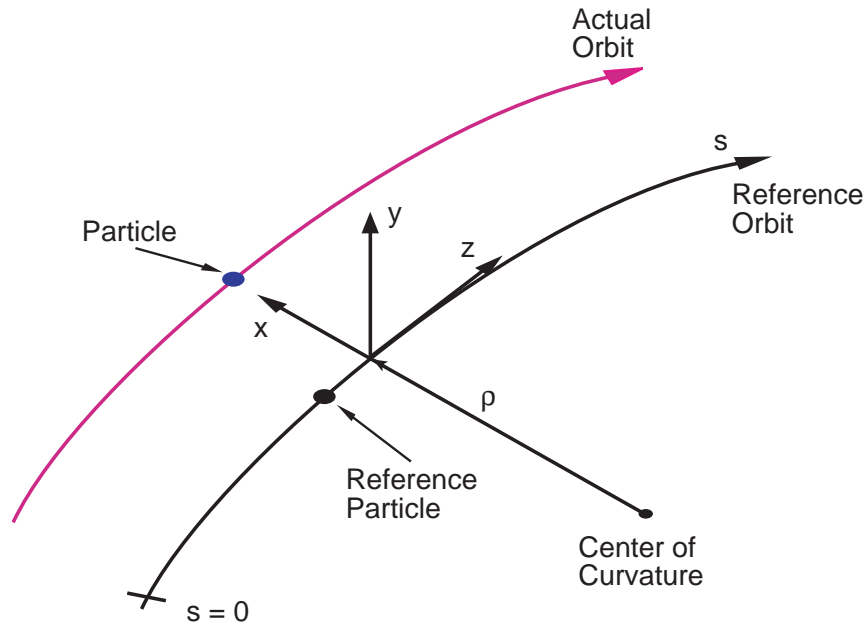


Figure 9.1: The Local Reference System. By construction,  $z = 0$  for the particle coordinates in the local reference system.

In *Bmad*, a lattice is comprised of a sequence of elements such as quadrupoles, bends, RF cavities, etc. Each element has an entrance point, an exit point, and a reference curve between them. For a **bend**, the reference curve is a segment of a circular arc. For all other elements, the reference curve is a straight line segment. The reference orbit itself is constructed by arranging the elements so that the exit point of one element coincides with the entrance point of the next with the reference curves forming an arc with no kinks. The reference orbit is then the sum of the reference curves. Exceptions to this construction method may be made by using **Patch** elements which can arbitrarily offset the entrance point of an element with respect to the exit point of the previous element. See §2.23. If not specified otherwise, the  $s = 0$  position is the entrance point of the first element.

Notice that, in a **Wiggler**, the reference orbit, which is a straight line, does *not* correspond to the orbit that any actual particle could travel. Typically the physical entity of an element is centered about the reference curve. However, by specifying offsets and pitches (See §4.4), the physical element may be arbitrarily offset with respect to its reference curve. Shifting a physical magnet with respect to its reference curve generally means that the reference curve does *not* correspond to the orbit that any actual particle could travel.

Do not confuse this reference orbit (which defines the local coordinate system) with the reference orbit about which the transfer maps are calculated (§16.2). The former is fixed by the lattice while the latter can be any arbitrary orbit.

## 9.2 Global Coordinates

The global coordinate system, also called the ‘floor’ coordinates, describes the orientation of the reference orbit with respect to the laboratory coordinate system. *Bmad*, following the *MAD* convention, uses a Cartesian coordinate system  $(X, Y, Z)$  for the global coordinate system, along with three angles  $\theta, \phi, \psi$  used to define the reference orbit’s orientation as shown in Figure 9.2. Conventionally,  $Y$  is the ‘vertical’ coordinate and  $(X, Z)$  are the ‘horizontal’ coordinates. The three angles are defined as follows:

**$\theta$  Azimuth angle:** Angle in the  $(X, Z)$  plane between the  $Z$ -axis and the projection of the  $z$ -axis onto the  $(X, Z)$  plane. A positive angle of  $\theta = \pi/2$  corresponds to the projected  $z$ -axis pointing in the positive  $X$  direction.

**$\phi$  Pitch (elevation) angle:** Angle between the  $z$ -axis and the  $(X, Z)$  plane. A positive angle of  $\phi = \pi/2$  corresponds to the  $z$ -axis pointing in the positive  $Y$  direction.

**$\psi$  Roll angle:** Angle of the  $x$ -axis with respect to the line formed by the intersection of the  $(X, Z)$  plane with the  $(x, y)$  plane. A positive  $\psi$  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.  $\theta$  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  $\phi$  and  $\psi$  will both be zero. The **beginning** statement (§7.4) in a lattice file can be used to override these defaults.

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

$$\mathbf{V} = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \quad (9.1)$$

The orientation of an element is described by a unitary matrix  $\mathbf{W}$ . The column vectors of  $\mathbf{W}$  are the unit vectors spanning the local coordinate axes in the order  $(x, y, s)$ .  $\mathbf{W}$  can be expressed in terms of

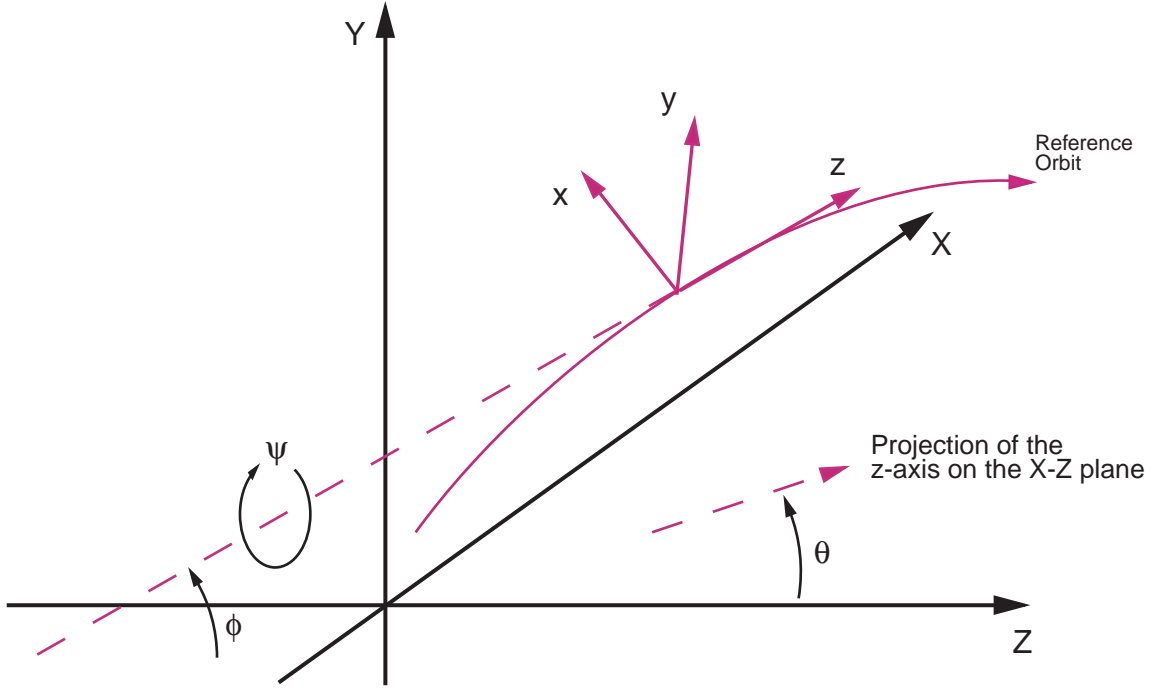


Figure 9.2: The Global Coordinate System

the angles  $\theta$ ,  $\phi$ , and  $\psi$  via the formula

$$\mathbf{W} = \mathbf{W}_\Theta \mathbf{W}_\Phi \mathbf{W}_\Psi \quad (9.2)$$

where

$$\mathbf{W}_\Theta = \begin{pmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{pmatrix}, \quad \mathbf{W}_\Phi = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \phi & \sin \phi \\ 0 & -\sin \phi & \cos \phi \end{pmatrix}, \quad \mathbf{W}_\Psi = \begin{pmatrix} \cos \psi & -\sin \psi & 0 \\ \sin \psi & \cos \psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (9.3)$$

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

$$\mathbf{V}_i = \mathbf{W}_{i-1} \mathbf{L}_i + \mathbf{V}_{i-1}, \quad (9.4)$$

$$\mathbf{W}_i = \mathbf{W}_{i-1} \mathbf{S}_i \quad (9.5)$$

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

$$\mathbf{L} = \begin{pmatrix} 0 \\ 0 \\ L \end{pmatrix}, \quad \mathbf{S} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad (9.6)$$

Where  $L$  is the length of the element. For a **bend**,  $\mathbf{L}$  and  $\mathbf{S}$  are given by

$$\mathbf{L} = \mathbf{T} \tilde{\mathbf{L}}, \quad \mathbf{S} = \mathbf{T} \tilde{\mathbf{S}} \mathbf{T}^{-1} \quad (9.7)$$

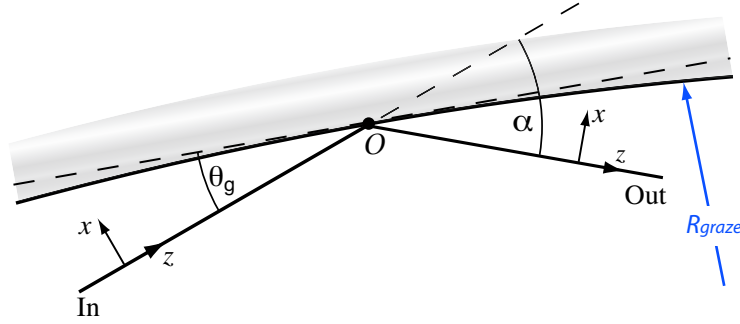


Figure 9.3: Reflection by a mirror. The geometry shown here is appropriate for a tilt angle of  $\theta_t = 0$ .  $\theta_g$  is the graze angle,  $\alpha$  is the bend angle of the coordinates. Point  $O$  is the origin of both the local coordinates just before and just after the reflection.

where

$$\tilde{\mathbf{L}} = \begin{pmatrix} \rho(\cos \alpha - 1) \\ 0 \\ \rho \sin \alpha \end{pmatrix}, \quad \tilde{\mathbf{S}} = \begin{pmatrix} \cos \alpha & 0 & -\sin \alpha \\ 0 & 1 & 0 \\ \sin \alpha & 0 & \cos \alpha \end{pmatrix}, \quad \mathbf{T} = \begin{pmatrix} \cos \theta_t & -\sin \theta_t & 0 \\ \sin \theta_t & \cos \theta_t & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (9.8)$$

with  $\rho$  being the bend radius (`rho`),  $\alpha$  is the bend **angle** (§2.4), and  $\theta_t$  is the **tilt** angle (§4.4). Without a tilt,  $\mathbf{T}$  is the unit matrix resulting in  $\mathbf{L} = \tilde{\mathbf{L}}$  and  $\mathbf{S} = \tilde{\mathbf{S}}$ . Notice that for a bend in the horizontal  $X - Z$  plane, a positive bend **angle** will result in a decreasing azimuth angle  $\theta$ .

The bend transformation (Eq. (9.7)) is so constructed that the transformation is equivalent to rotating the local coordinate system around an axis that is perpendicular to the plane of the bend. This rotation axis is invariant under the bend transformation. For example, for  $\theta_t = 0$  (or  $\pi$ ) the  $y$ -axis is the rotation axis and the  $y$ -axis of the local coordinates before the bend will be parallel to the  $y$ -axis of the local coordinates after the bend. That is, a lattice with only bends with  $\theta_t = 0$  or  $\pi$  will lie in the horizontal plane (this assuming that the  $y$ -axis starts out pointing along the  $Y$ -axis as it does by default). For  $\theta_t = \pm\pi/2$ , the bend axis is the  $x$ -axis. A value of  $\theta_t = +\pi/2$  represents a downward pointing bend.

### 9.2.1 Mirror Element

A **mirror** element (§2.19) reflects photons. A mirror can be thought of as a zero length bend ( $\tilde{\mathbf{L}} = (0, 0, 0)$ ) with the total bend angle being twice the graze angle. The orientation of the exit coordinates (the local coordinates after the reflection) are only affected by the mirror's tilt and graze angle parameters and is independent of all other mirror parameters such as the radius of curvature of the mirror surface, etc. The local  $z$ -axis of the entrance coordinates along with the  $z$ -axis of the exit coordinates form a plane which is called the mirror's **bend plane**.

An example is shown in Fig. 9.3. This example is appropriate for a tilt angle of  $\theta_t = 0$  (the rotation axis is here the  $y$ -axis). Since the mirror is modeled to be of zero length, the origin (marked  $O$  in the figure) of the entrance and exit local coordinates is the same.

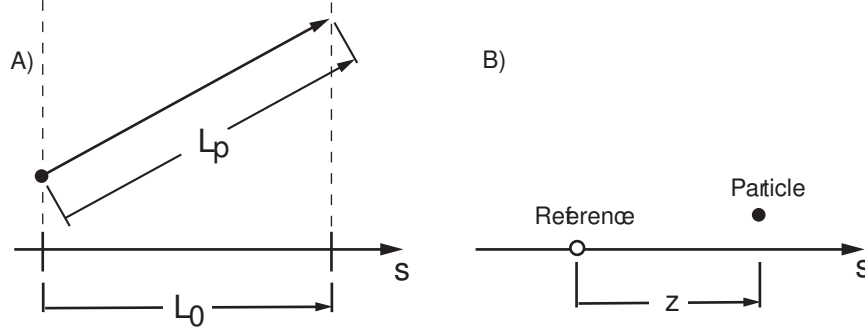


Figure 9.4: Interpreting Canonical  $z$  at constant velocity: A) The change in  $z$  going through an element of length  $L_0$  is  $L_0 - L_p$ . B) At constant time,  $z$  is the longitudinal distance between the reference particle and the particle.

### 9.2.2 Patch Element

A **patch** element shifts the reference orbit (§2.23). Eqs. (9.4) and (9.5) are used to propagate the **W** matrix through a **patch**. The **L** vector for a patch is

$$\mathbf{L} = (x_{\text{offset}} \ y_{\text{offset}} \ z_{\text{offset}}) \quad (9.9)$$

The **S** matrix corresponding to a non-zero **tilt** for a **patch** element is

$$\mathbf{S} = \begin{pmatrix} \cos \theta_t & -\sin \theta_t & 0 \\ \sin \theta_t & \cos \theta_t & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (9.10)$$

The **S** matrix corresponding to a non-zero **x\_pitch** of a **patch** element is

$$\mathbf{S} = \begin{pmatrix} \cos \theta_x & 0 & -\sin \theta_x \\ 0 & 1 & 0 \\ \sin \theta_x & 0 & \cos \theta_x \end{pmatrix} \quad (9.11)$$

where  $\theta_x$  is the value of **x\_pitch**. Finally, the **S** matrix corresponding to a non-zero **y\_pitch** of a **patch** element is

$$\mathbf{S} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta_y & \sin \theta_y \\ 0 & -\sin \theta_y & \cos \theta_y \end{pmatrix} \quad (9.12)$$

where  $\theta_y$  is the value of **y\_pitch**.

If a **patch's** **translate\_after** is set to **True**, the offsets are applied after the pitches and tilts. In this case, Eq. (9.4) is replaced by

$$\mathbf{V}_i = \mathbf{W}_i \mathbf{L}_i + \mathbf{V}_{i-1}, \quad (9.13)$$

## 9.3 Phase Space Coordinate System

*Bmad* uses the canonical phase space coordinates

$$\mathbf{r}(s) = (x, p_x, y, p_y, z, p_z) \quad (9.14)$$

The longitudinal position  $s$  is the independent variable instead of the time.  $x$  and  $y$ , are the reference orbit coordinates given in §9.1. The canonical momenta  $p_x$  and  $p_y$  are normalized by the reference (sometimes called the design) momentum  $P_0$

$$p_x = \frac{P_x}{P_0} \quad (9.15)$$

$$p_y = \frac{P_y}{P_0} \quad (9.16)$$

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

The canonical  $z$  coordinate is

$$\begin{aligned} z(s) &= -\beta(s) c (t(s) - t_0(s)) \\ &\equiv -\beta(s) c \Delta t(s) \end{aligned} \quad (9.17)$$

$t$  is the time at which the particle is at position  $s$ ,  $t_0$  is the time at which the reference particle is at position  $s$ , and  $\beta$  is  $v/c$  with  $v$  being the particle velocity (and not the reference velocity). 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),  $\beta c t$  is just the path length. Thus, the change in  $z$  going through an element is

$$\Delta z = L_0 - L_p \quad (9.18)$$

where, as shown in Figure 9.4A,  $L_0$  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. At constant  $\beta$ , another way of interpreting canonical  $z$  is that, at constant time,  $z$  is the longitudinal distance between the particle and the reference particle as shown in Figure 9.4B. Positive  $z$  indicates that the particle is ahead of the reference particle.

Do not confuse the canonical  $z$  with the  $z$  that is the particle's longitudinal coordinate in the local reference frame as shown in Figure 9.1. By construction, this latter  $z$  is always zero.

Notice that if a particle gets an instantaneous longitudinal kick so that  $\beta$  is discontinuous then, from Eq. (9.17), canonical  $z$  is discontinuous even though the particle itself does not move in space. In general, from Eq. (9.17), The value of  $z$  for a particle at  $s_2$  is related to the value of  $z$  for the particle at  $s_1$  by

$$z_2 = \frac{\beta_2}{\beta_1} z_1 - \beta_2 c (\Delta t_2 - \Delta t_1) \quad (9.19)$$

$\Delta t_2 - \Delta t_1$  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 canonical momentum  $p_z$  is given by

$$p_z = \frac{\Delta P}{P_0} \equiv \frac{P - P_0}{P_0} \quad (9.20)$$

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

$$p_z = \frac{\Delta E}{E_0} \quad (9.21)$$

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 (§2.16) the reference momentum is *not* constant so the tracking for an `Lcavity` is not canonical.

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

**Bmad\_standard** (§5) tracking and transfer matrix calculations use the small angle (paraxial) approximation where it is assumed that  $p_x, p_y \ll 1$ . With this approximation, the relationship, between the canonical momenta and the slopes  $x' \equiv dx/ds$  and  $y' \equiv dy/ds$  is

$$x' \approx \frac{p_x - a_x}{1 + p_z} (1 + gx) \quad (9.22)$$

$$y' \approx \frac{p_y - a_y}{1 + p_z} (1 + gx) \quad (9.23)$$

$\mathbf{a} = q A/c P_0$  is the normalized vector potential,  $g = 1/\rho$  is the curvature function with  $\rho$  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

$$\frac{dz}{ds} = -g x - \frac{1}{2}(x'^2 + y'^2) \quad (9.24)$$

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

For those programmers using the PTC software package directly (ignore this if you don't know what is being talked about here) Étienne Forest uses, by default, a different coordinate system where  $(z, p_z)$  is replaced by  $(p_z, -z)$ . However, PTC also has the ability to switch to the  $(p_t, c\Delta t)$  coordinate system.





# Chapter 10

## Physics

This chapter describes some of the physics and various conventions *Bmad* uses. This chapter is by no means a comprehensive treatment of accelerator physics and the interested reader is invited to consult any one of the number of good texts that deal with the subject.

### 10.1 Units

*Bmad* uses SI (Système International) units as shown in Table 10.1. Note that *MAD* uses different units. For example, *MAD*'s unit of Particle Energy is GeV not eV.

<i>Quantity</i>	<i>Units</i>
Angles	radians
Betatron Phase	radians
Charge	Coulombs
Current	Amps
Frequency	Hz
Kick	radians
Length	meters
Magnetic Field	Tesla
Particle Energy	eV
Phase Angles (RF)	radians/ $2\pi$
Voltage	Volts

Table 10.1: Physical units used by *Bmad*.

### 10.2 Magnetic 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*, the vertical magnetic field along the  $y = 0$  axis is expanded in a Taylor series

$$B_y(x, 0) = \sum_n B_n \frac{x^n}{n!} \quad (10.1)$$

This is not the most general form for the magnetic field. Essentially all of the skew components have been ignored here. Assuming that the reference orbit is locally straight (there are correction terms if the Reference Orbit is locally curved), the field up to 3<sup>rd</sup> order is

$$B_x = B_1 y + B_2 xy + \frac{1}{6} B_3 (3x^2 y - y^3) + \dots \quad (10.2)$$

$$B_y = B_0 + B_1 x + \frac{1}{2} B_2 (x^2 - y^2) + \frac{1}{6} B_3 (x^3 - 3xy^2) + \dots \quad (10.3)$$

The normalized integrated multipole  $K_n L$  is used when specifying magnetic multipole components

$$K_n L \equiv \frac{q L B_n}{P_0} \quad (10.4)$$

$L B_n$  is the integrated multipole component over a length  $L$ , and  $P_0$  is the reference momentum. Note that  $P_0/q$  is sometimes written as  $B\rho$ . This is just an old notation where  $\rho$  is the bending radius of a particle with the reference energy in a field of strength  $B$ . The kicks  $\Delta p_x$  and  $\Delta p_y$  that a particle experiences going through a multipole field is

$$\Delta p_x = \frac{-q L B_y}{P_0} \quad (10.5)$$

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

$$\Delta p_y = \frac{q L B_x}{P_0} \quad (10.6)$$

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

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

$$\Delta p_x = \sum_{n=0}^{\infty} \frac{K_n L}{n!} \sum_{m=0}^{\lfloor \frac{n}{2} \rfloor} \binom{n}{2m} (-1)^{m+1} x^{n-2m} y^{2m} \quad (10.7)$$

$$\Delta p_y = \sum_{n=0}^{\infty} \frac{K_n L}{n!} \sum_{m=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n}{2m+1} (-1)^m x^{n-2m-1} y^{2m+1} \quad (10.8)$$

So far only the normal components of the field have been considered. If the fields associated with a particular  $B_n$  multipole component are rotated in the  $(x, y)$  plane by an angle  $\theta_n$ , the magnetic field at a point  $(x, y)$  can be expressed in complex notation as

$$B_y(x, y) + iB_x(x, y) = \frac{1}{n!} B_n e^{-i(n+1)\theta_n} e^{in\theta} r^n \quad (10.9)$$

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

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  $n^{\text{th}}$  order multipole is

$$\frac{q L}{P_0} (B_y + iB_x) = (b_n + ia_n) (x + iy)^n \quad (10.10)$$

The conversion between  $(a_n, b_n)$  and  $(K_n L, \theta_n)$  is

$$b_n + ia_n = \frac{1}{n!} K_n L e^{-i(n+1)\theta_n} \quad (10.11)$$

or

$$K_n L = n! \sqrt{a_n^2 + b_n^2} \quad (10.12)$$

$$\tan[(n+1)\theta_n] = \frac{-a_n}{b_n} \quad (10.13)$$

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

$$(n+1)\theta_n = \frac{\pi}{2} \quad (10.14)$$

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

When the  $a_n$  and  $b_n$  are associated with a physical element (as opposed to the  $a_n$  and  $b_n$  associated with an `AB_Multipole` element), a measurement radius  $r_0$  and a scale factor  $F$  are used to scale the  $a_n$  and  $b_n$  according to the formula

$$[a_n(\text{actual}), b_n(\text{actual})] = [a_n(\text{input}), b_n(\text{input})] \cdot F \cdot \frac{r_0^{n_{\text{ref}}}}{r_0^n} \quad (10.15)$$

$a_n(\text{input})$  and  $b_n(\text{input})$  are the multipole values as given in the lattice file.  $a_n(\text{actual})$  and  $b_n(\text{actual})$  are the multipole values that are used in any simulation calculations.  $r_0$  is set by the `radius` attribute of an element.  $F$  and  $n_{\text{ref}}$  are set automatically depending upon the type of element as shown in Table 10.2.

Note that the  $n = 0$  component of an `AB_Multipole` or `Multipole` element rotates the reference orbit essentially acting as a zero length bend. This is not true for multipoles that are associated with non-multipole elements.

<i>Element</i>	<i>F</i>	<i>n<sub>ref</sub></i>
Kicker	$\sqrt{H\text{kick}^2 + V\text{kick}^2}$	0
Hkicker	Kick	0
Vkicker	Kick	0
Rbend	G * L	0
Sbend	G * L	0
Elseparator	$\sqrt{H\text{kick}^2 + V\text{kick}^2}$	0
Quadrupole	K1 * L	1
Solenoid	KS * L	1
Sol_Quad	K1 * L	1
Sextupole	K2 * L	2
Octupole	K3 * L	3

Table 10.2:  $F$  and  $n_{\text{ref}}$  for various elements.

## 10.3 Taylor Maps and Symplectic Integration

A transport map  $\mathcal{M} : \mathcal{R}^6 \rightarrow \mathcal{R}^6$  through an element or a section of a lattice is a function that maps the starting phase space coordinates  $\mathbf{r}(\text{in})$  to the ending coordinates  $\mathbf{r}(\text{out})$

$$\mathbf{r}(\text{out}) = \mathcal{M} \mathbf{r}(\text{in}) \quad (10.16)$$

$\mathcal{M}$  is made up of six functions  $\mathcal{M}_i : \mathcal{R}^6 \rightarrow \mathcal{R}$ . Each of these functions maps to one of the  $r(\text{out})$  coordinates. These functions can be expanded in a Taylor series and truncated at some order. Each

Taylor series is in the form

$$r_i(\text{out}) = \sum_{j=1}^N C_{ij} \prod_{k=1}^6 r_k^{e_{ijk}}(\text{in}) \quad (10.17)$$

Where the  $C_{ij}$  are coefficients and the  $e_{ijk}$  are integer exponents. The order of the map is

$$\text{order} = \max_{i,j} \left( \sum_{k=1}^6 e_{ijk} \right) \quad (10.18)$$

The standard *Bmad* routine for printing a Taylor map might produce something like this:

Taylor Terms:										
Out	Coef	Exponents							Order	Reference
-----										
1:	-0.6000000000000	0	0	0	0	0	0	0	0.200000000	
1:	1.0000000000000	1	0	0	0	0	0	1		
1:	0.1450000000000	2	0	0	0	0	0	2		
-----										
2:	-0.1850000000000	0	0	0	0	0	0	0	0.000000000	
2:	1.3000000000000	0	1	0	0	0	0	1		
2:	3.8000000000000	2	0	0	0	0	1	3		
-----										
3:	1.0000000000000	0	0	1	0	0	0	1	0.100000000	
3:	1.6000000000000	0	0	0	1	0	0	1		
3:	-11.138187077310	1	0	1	0	0	0	2		
-----										
4:	1.0000000000000	0	0	0	1	0	0	1	0.000000000	
-----										
5:	0.0000000000000	0	0	0	0	0	0	0	0.000000000	
5:	0.000001480008	0	1	0	0	0	0	1		
5:	1.0000000000000	0	0	0	0	1	0	1		
5:	0.0000000000003	0	0	0	0	0	1	1		
5:	0.0000000000003	2	0	0	0	0	0	2		
-----										
6:	1.0000000000000	0	0	0	0	0	1	1	0.000000000	

Each line in the example represents a single Taylor term. The Taylor terms are grouped into 6 Taylor series, one each output phase space coordinate. The first column in the example, labeled “out”, (corresponding to the  $i$  index in Eq. (10.17)) indicates the Taylor series: 1 =  $x(\text{out})$ , 2 =  $p_x(\text{out})$ , etc. The 6 exponent columns give the  $e_{ijk}$  of Eq. (10.17). In this example, the second Taylor series (out = 2), when expressed as a formula, would read:

$$p_x(\text{out}) = -0.185 + 1.3 p_x(\text{in}) + 3.8 x^2(\text{in}) p_z(\text{in}) \quad (10.19)$$

The reference column in the above example shows the input coordinates around which the Taylor map is calculated. In this case, the reference coordinates where

$$(x, p_x, y, p_y, z, p_z)_{ref} = (0.2, 0, 0.1, 0, 0, 0) \quad (10.20)$$

The choice of the reference point will affect the values of the coefficients of the Taylor map. For example, suppose that the exact map through an element looks like

$$x(\text{out}) = A \sin(k x(\text{in})) \quad (10.21)$$

Then a Taylor map to 1<sup>st</sup> order is

$$x(out) = c_0 + c_1 x(in) \quad (10.22)$$

where

$$\begin{aligned} c_1 &= A k \cos(k x_{\text{ref}}) \\ c_0 &= A \sin(k x_{\text{ref}}) - c_1 x_{\text{ref}} \end{aligned} \quad (10.23)$$

Notice that once the coefficient values are determined the reference point does not play any role when the Taylor map is evaluated to determine the output coordinates as a function of the input coordinates.

Of importance in working with Taylor maps is the concept of **feed-down**. This is best explained with an example. To keep the example simple, the discussion is limited to one phase space dimension so that the Taylor maps are a single Taylor series. Take the map  $M_1$  from point 0 to point 1 to be

$$M_1 : x_1 = x_0 + 2 \quad (10.24)$$

and the map  $M_2$  from point 1 to point 2 to be

$$M_2 : x_2 = x_1^2 + 3x_1 \quad (10.25)$$

Then concatenating the maps to form the map  $M_3$  from point 0 to point 2 gives

$$M_3 : x_2 = (x_0 + 2)^2 + 3(x_0 + 2) = x_0^2 + 7x_0 + 10 \quad (10.26)$$

However if we are evaluating our maps to only 1<sup>st</sup> order the map  $M_2$  becomes

$$M_2 : x_2 = 3x_1 \quad (10.27)$$

and concatenating the maps now gives

$$M_3 : x_2 = 3(x_0 + 2) = 3x_0 + 6 \quad (10.28)$$

Comparing this to Eq. (10.26) shows that by neglecting the 2<sup>nd</sup> order term in Eq. (10.25) leads to 0<sup>th</sup> and 1<sup>st</sup> order errors in Eq. (10.28). These errors can be traced to the finite 0<sup>th</sup> order term in Eq. (10.24). This is the principal of feed-down: Given  $M_3$  which is a map produced from the concatenation of two other maps,  $M_1$ , and  $M_2$

$$M_3 = M_2(M_1) \quad (10.29)$$

Then if  $M_1$  and  $M_2$  are correct to  $n^{th}$  order,  $M_3$  will also be correct to  $n^{th}$  order as long as  $M_1$  has no constant (0<sup>th</sup> order) term. [Notice that a constant term in  $M_2$  does not affect the argument.] What happens if we know there are constant terms in our maps? One possibility is to go to a coordinate system where the constant terms vanish. In the above example that would mean using the coordinate  $\tilde{x}_0$  at point 0 given by

$$\tilde{x}_0 = x_0 + 2 \quad (10.30)$$

The other possibility is to use symplectic integration. By its nature, symplectic integration never has problems with feed-down.

The subject of symplectic integration is too large to be covered in this guide. The reader is referred to the book “Beam Dynamics: A New Attitude and Framework” by Etienne Forest[20]. A brief synopsis: Symplectic integration uses as input 1) The Hamiltonian that defines the equations of motion, and 2) a Taylor map  $M_1$  from point 0 to point 1. Symplectic integration from point 1 to point 2 produces a Taylor map  $M_3$  from point 0 to point 2. Symplectic integration can produce maps to arbitrary order. In any practical application the order  $n$  of the final map is specified and in the integration procedure all

terms of order higher than  $n$  are ignored. If one is just interested in knowing the final coordinates of a particle at point 2 given the initial coordinates at point 1 then  $M_1$  is just the constant map

$$M_1 : x_1 = c_i \quad (10.31)$$

where  $c_i$  is the initial starting point. The order of the integration is set to 0 so that all non-constant terms are ignored. The final map is also just a constant map

$$M_3 : x_2 = c_f \quad (10.32)$$

If the map from point 1 to point 2 is desired then the map  $M_1$  is just set to the identity map

$$M_1 : x_1 = x_0 \quad (10.33)$$

In general it is impossible to exactly integrate any non-linear system. In practice, the symplectic integration is achieved by slicing the interval between point 1 and point 2 into a number of (generally equally spaced) slices. The integration is performed, slice step by slice step. This is analogous to integrating a function by evaluating the function at a number of points. Using more slices gives better results but slows down the calculation. The speed and accuracy of the calculation is determined by the number of slices and the **order** of the integrator. The concept of integrator order can best be understood by analogy by considering the trapezoidal rule for integrating a function of one variable:

$$\int_{y_a}^{y_b} f(y) dy = h \left[ \frac{1}{2} f(y_a) + \frac{1}{2} f(y_b) \right] + o(h^3 f^{(2)}) \quad (10.34)$$

In the formula  $h = y_b - y_a$  is the slice width.  $o(h^3 f^{(2)})$  means that the error of the trapezoidal rule scales as the second derivative of  $f$ . Since the error scales as  $f^{(2)}$  this is an example of a second order integrator. To integrate a function between points  $y_1$  and  $y_N$  we slice the interval at points  $y_2 \dots y_{N-1}$  and apply the trapezoidal rule to each interval. Examples of higher order integrators can be found, for example, in Numerical Recipes[18]. The concept of integrator order in symplectic integration is analogous.

The optimum number of slices is determined by the smallest number that gives an acceptable error. The slice size is given by the **ds\_step** attribute of an element (§5.3). Integrators of higher order will generally need a smaller number of slices to achieve a given accuracy. However, since integrators of higher order take more time per slice step, and since it is computation time and not number of slices which is important, only a measurement of error and calculation time as a function of slice number and integrator order will unambiguously give the optimum integrator order and slice width. In doing a timing test, it must be remembered that since the magnitude of any non-linearities will depend upon the starting position, the integration error will be dependent upon the starting map  $M_1$ . *Bmad* has integrators of order 2, 4, and 6 (§5.3). Timing tests performed for some wiggler elements (which have strong nonlinearities) showed that, in this case, the 2<sup>nd</sup> order integrator gave the fastest computation time for a given accuracy. However, the higher order integrators may give better results for elements with weaker nonlinearities.

## 10.4 Symplectification

If the evolution of a system can be described using a Hamiltonian then it can be shown that the linear part of any transport map (the Jacobian) must obey the symplectic condition. If a matrix  $\mathbf{M}$  is not symplectic, Healy[5] has provided an elegant method for finding a symplectic matrix that is “close” to  $\mathbf{M}$ . The procedure is as follows: From  $\mathbf{M}$  a matrix  $\mathbf{V}$  is formed via

$$\mathbf{V} = \mathbf{S}(\mathbf{I} - \mathbf{M})(\mathbf{I} + \mathbf{M})^{-1} \quad (10.35)$$

where  $\mathbf{S}$  is the matrix

$$\mathbf{S} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix} \quad (10.36)$$

$\mathbf{V}$  is symmetric if and only if  $\mathbf{M}$  is symplectic. In any case, a symmetric matrix  $\mathbf{W}$  near  $\mathbf{V}$  can be formed via

$$\mathbf{W} = \frac{\mathbf{V} + \mathbf{V}^t}{2} \quad (10.37)$$

A symplectic matrix  $\mathbf{F}$  is now obtained by inverting (10.35)

$$\mathbf{F} = (\mathbf{I} + \mathbf{S}\mathbf{W})(\mathbf{I} - \mathbf{S}\mathbf{W})^{-1} \quad (10.38)$$

## 10.5 LINAC Accelerating Cavities (Lcavity)

The transverse trajectory through an Lcavity is modeled using equations developed by Rosenzweig and Serafini[16] with

$$\begin{aligned} b_0 &= 1 \\ b_{-1} &= 1 \end{aligned}$$

and all other  $b_n$  set to zero.

The transport through the body (R&S Eq. (9)) has been modified to give the correct phase-space area at non ultra-relativistic energies:

$$\begin{pmatrix} x \\ x' \end{pmatrix}_2 = \begin{pmatrix} m_{11} & \beta_1 m_{12} \\ \frac{1}{\beta_2} m_{21} & \frac{\beta_1}{\beta_2} m_{22} \end{pmatrix} \begin{pmatrix} x \\ x' \end{pmatrix}_1 \quad (10.39)$$

where the  $m_{ij}$  are the matrix elements from R&S Eq. (9) and the  $\beta$  are the standard relativistic factors. With this, the determinate of the matrix is  $\beta_1 \gamma_1 / \beta_2 \gamma_2$ .

The change in  $z$  going through a cavity is calculated by first calculating the particle transit time  $\Delta t$

$$\begin{aligned} \Delta t &= \int_{s_1}^{s_2} ds \beta(s) \\ &= \int_{s_1}^{s_2} ds \frac{E}{\sqrt{E^2 - (mc^2)^2}} \\ &= \frac{P_{z2} - P_{z1}}{G} \end{aligned} \quad (10.40)$$

where it has been assumed that the accelerating gradient  $G$  is constant through the cavity. In this equation  $\beta = v/c$ ,  $E$  is the energy, and  $P_{z2}$  and  $P_{z1}$  are the entrance and exit momenta. Using Eq. (9.17), the change in  $z$  is thus

$$z_2 = \frac{\beta_2}{\beta_1} z_1 - \beta_2 \left( \frac{c P_{z2} - c P_{z1}}{G} - \frac{c \overline{P}_{z2} - c \overline{P}_{z1}}{\overline{G}} \right) \quad (10.41)$$

where  $\overline{P}$  and  $\overline{G}$  are the momentum and gradient of the reference particle.

## 10.6 Wigglers

As discussed in §2.30, *Bmad* wiggler elements are split into two classes: **map type** and **periodic type**. The **map type** wigglers are modeled using the method of Sagan, Crittenden, and Rubin[6]. In this model the magnetic field is written as a sum of terms  $B_i$

$$B(x, y, s) = \sum_i B_i(x, y, s; C, k_x, k_y, k_s, \phi_s) \quad (10.42)$$

Each term  $B_i$  is specified using five numbers:  $(C, k_x, k_y, k_s, \phi_s)$ . A term can take one of three forms: The first form is

$$\begin{aligned} B_x &= -C \frac{k_x}{k_y} \sin(k_x x) \sinh(k_y y) \cos(k_s s + \phi_s) \\ B_y &= C \cos(k_x x) \cosh(k_y y) \cos(k_s s + \phi_s) \\ B_s &= -C \frac{k_s}{k_y} \cos(k_x x) \sinh(k_y y) \sin(k_s s + \phi_s) \\ &\text{with } k_y^2 = k_x^2 + k_s^2. \end{aligned} \quad (10.43)$$

The second form is

$$\begin{aligned} B_x &= C \frac{k_x}{k_y} \sinh(k_x x) \sinh(k_y y) \cos(k_s s + \phi_s) \\ B_y &= C \cosh(k_x x) \cosh(k_y y) \cos(k_s s + \phi_s) \\ B_s &= -C \frac{k_s}{k_y} \cosh(k_x x) \sinh(k_y y) \sin(k_s s + \phi_s) \\ &\text{with } k_y^2 = k_x^2 - k_s^2, \end{aligned} \quad (10.44)$$

The third form is

$$\begin{aligned} B_x &= C \frac{k_x}{k_y} \sinh(k_x x) \sin(k_y y) \cos(k_s s + \phi_s) \\ B_y &= C \cosh(k_x x) \cos(k_y y) \cos(k_s s + \phi_s) \\ B_s &= -C \frac{k_s}{k_y} \cosh(k_x x) \sin(k_y y) \sin(k_s s + \phi_s) \\ &\text{with } k_y^2 = k_x^2 - k_s^2. \end{aligned} \quad (10.45)$$

The relationship between  $k_x$ ,  $k_y$ , and  $k_s$  ensures that Maxwell's equations are satisfied. Since the field is given by analytic equations, Lie algebraic techniques can be used to construct Taylor maps to arbitrary order.

**Periodic type** wigglers use a simplified model where the magnetic field components are

$$\begin{aligned} B_y &= B_{\max} \cosh(k_s y) \cos(k_s s) \\ B_s &= -B_{\max} \sinh(k_s y) \sin(k_s s) \end{aligned} \quad (10.46)$$

where  $B_{\max}$  is the maximum field on the centerline and  $k$  is given in terms of the pole length (**l\_pole**) by

$$k_s = \frac{\pi}{l_{\text{pole}}} \quad (10.47)$$

This type of wiggler has infinitely wide poles. With **bmad\_standard** tracking and transfer matrix calculations the vertical focusing is assumed small so averaged over a period the horizontal motion looks like a drift and the vertical motion is modeled as a combination focusing quadrupole and focusing octupole giving a kick[7]

$$\frac{dp_y}{ds} = k_1 \left( y + \frac{2}{3} k_s^2 y^3 \right) \quad (10.48)$$



where

$$k_1 = \frac{-1}{2} \left( \frac{c B_{\max}}{P_0 (1 + p_s)} \right)^2 \quad (10.49)$$

with  $k_1$  being the linear focusing constant. For radiation calculations the true horizontal trajectory with  $y = 0$  is needed

$$x = \frac{\sqrt{2|k_1|}}{k_s^2} \cos(k_s s) \quad (10.50)$$

With `periodic type` wigglers and `bmad_standard` tracking, the phase  $\phi_s$  in Eqs. (10.46) is irrelevant. When the tracking involves Taylor maps and symplectic integration, the phase is important. Here the phase is chosen so that  $B_y$  is symmetric about the center of the wiggler

$$\phi_s = \frac{-k_y L}{2} \quad (10.51)$$

With this choice, a particle that enters the wiggler on-axis will leave the wiggler on-axis provided there is an even number of poles.

## 10.7 Synchrotron Radiation Damping and Excitation

Emission of synchrotron radiation by a particle can be decomposed into two parts. The deterministic average radiation emitted produces damping while the stochastic fluctuating part produces excitation[8].

The treatment of radiation damping by *Bmad* essentially follows *MAD*. The average change in energy  $\Delta E$  of a particle going through a section of magnet due to synchrotron radiation is

$$\frac{\Delta E}{E_0} = -k_d (1 + p_z) \quad (10.52)$$

where

$$k_d \equiv \frac{2 r_e}{3} \gamma_0^3 \langle g_0^2 \rangle L_p (1 + p_z) \quad (10.53)$$

$r_e$  is the classical electron radius,  $L_p$  is the actual path length,  $\gamma_0$  is the energy factor of an on-energy particle,  $1/g_0$  is the bending radius of an on-energy particle, and  $\langle g_0^2 \rangle$  is an average of  $g_0^2$  over the actual path.

The energy lost is given by

$$\frac{\Delta E}{E_0} = -k_f (1 + p_z) \quad (10.54)$$

where

$$k_f \equiv \left( \frac{55 r_e \hbar c}{24 \sqrt{3} m_e} L_p \gamma_0^5 \langle g_0^3 \rangle \right)^{1/2} (1 + p_z) \xi \quad (10.55)$$

$\xi$  is a Gaussian distributed random number with unit sigma and zero mean.

Using Eqs. (10.53) and (10.55) The total change in  $p_z$  can be written as

$$\Delta p_z = \frac{\Delta E}{E_0} = -k_E (1 + p_z) \quad (10.56)$$

where

$$k_E = k_d + k_f \quad (10.57)$$

Since the radiation is emitted in the forward direction the angles  $x'$  and  $y'$  are invariant which leads to the following equations for the changes in  $p_x$  and  $p_y$

$$\begin{aligned}\Delta p_x &= -k_E p_x \\ \Delta p_y &= -k_E p_y\end{aligned}\tag{10.58}$$

The above formalism does not take into account the fact that radiation is emitted with a  $1/\gamma$  angular distribution. This means that the calculated vertical emittance for a lattice with bends only in the horizontal plane and without any coupling elements such as skew quadrupoles will be zero. Typically, in practice, the vertical emittance will be dominated by coupling so this approximation is generally a good one.

## 10.8 Coupling and Normal Modes

The coupling formalism used by *Bmad* is taken from the paper of Sagan and Rubin[9]. The main equations are reproduced here. A one-turn map  $\mathbf{T}(s)$  for the transverse two-dimensional phase space  $\mathbf{x} = (x, x', y, y')$  starting and ending at some point  $s$  can be written as

$$\mathbf{T} = \mathbf{V} \mathbf{U} \mathbf{V}^{-1},\tag{10.59}$$

where  $\mathbf{V}$  is symplectic, and  $\mathbf{U}$  is of the form

$$\mathbf{U} = \begin{pmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{B} \end{pmatrix}.\tag{10.60}$$

Since  $\mathbf{U}$  is uncoupled the standard Twiss analysis can be performed on the matrices  $\mathbf{A}$  and  $\mathbf{B}$ . The normal modes are labeled  $a$  and  $b$  and if the one-turn matrix  $\mathbf{T}$  is uncoupled then  $a$  corresponds to the horizontal mode and  $b$  corresponds to the vertical mode.

$\mathbf{V}$  is written in the form

$$\mathbf{V} = \begin{pmatrix} \gamma \mathbf{I} & \mathbf{C} \\ -\mathbf{C}^+ & \gamma \mathbf{I} \end{pmatrix},\tag{10.61}$$

where  $\mathbf{C}$  is a 2x2 matrix and  $+$  superscript denotes the symplectic conjugate:

$$\mathbf{C}^+ = \begin{pmatrix} C_{22} & -C_{12} \\ -C_{21} & C_{11} \end{pmatrix}.\tag{10.62}$$

Since we demand that  $\mathbf{V}$  be symplectic we have the condition

$$\gamma^2 + \|\mathbf{C}\| = 1,\tag{10.63}$$

and  $\mathbf{V}^{-1}$  is given by

$$\mathbf{V}^{-1} = \begin{pmatrix} \gamma \mathbf{I} & -\mathbf{C} \\ \mathbf{C}^+ & \gamma \mathbf{I} \end{pmatrix}.\tag{10.64}$$

$\mathbf{C}$  is a measure of the coupling.  $\mathbf{T}$  is uncoupled if and only if  $\mathbf{C} = \mathbf{0}$ .

It is useful to normalize out the  $\beta(s)$  variation in the the above analysis. Normalized quantities being denoted by a bar above them. The normalized normal mode matrix  $\bar{\mathbf{U}}$  is defined by

$$\bar{\mathbf{U}} = \mathbf{G} \mathbf{U} \mathbf{G}^{-1},\tag{10.65}$$

Where  $\mathbf{G}$  is given by

$$\mathbf{G} \equiv \begin{pmatrix} \mathbf{G}_a & \mathbf{0} \\ \mathbf{0} & \mathbf{G}_b \end{pmatrix}, \quad (10.66)$$

with

$$\mathbf{G}_a = \begin{pmatrix} \frac{1}{\sqrt{\beta_a}} & 0 \\ \frac{\alpha_a}{\sqrt{\beta_a}} & \sqrt{\beta_a} \end{pmatrix}, \quad (10.67)$$

with a similar equation for  $\mathbf{G}_b$ . With this definition, the corresponding  $\overline{\mathbf{A}}$  and  $\overline{\mathbf{B}}$  (cf. Eq. (10.60)) are just rotation matrices. The relationship between  $\mathbf{T}$  and  $\overline{\mathbf{U}}$  is

$$\mathbf{T} = \mathbf{G}^{-1} \overline{\mathbf{V}} \overline{\mathbf{U}} \overline{\mathbf{V}}^{-1} \mathbf{G}, \quad (10.68)$$

where

$$\overline{\mathbf{V}} = \mathbf{G} \mathbf{V} \mathbf{G}^{-1}. \quad (10.69)$$

Using Eq. (10.66),  $\overline{\mathbf{V}}$  can be written in the form

$$\overline{\mathbf{V}} = \begin{pmatrix} \gamma \mathbf{I} & \overline{\mathbf{C}} \\ -\overline{\mathbf{C}}^+ & \gamma \mathbf{I} \end{pmatrix}, \quad (10.70)$$

with the normalized matrix  $\overline{\mathbf{C}}$  given by

$$\overline{\mathbf{C}} = \mathbf{G}_a \mathbf{C} \mathbf{G}_b^{-1}. \quad (10.71)$$

The normal mode coordinates  $\mathbf{a} = (a, a', b, b')$  are related to the laboratory frame via

$$\mathbf{a} = \mathbf{V}^{-1} \mathbf{x}. \quad (10.72)$$

In particular the normal mode dispersion  $\eta_a = (\eta_a, \eta'_a, \eta_b, \eta'_b)$  is related to the laboratory frame dispersion  $\eta_x = (\eta_x, \eta'_x, \eta_y, \eta'_y)$  via

$$\eta_a = \mathbf{V}^{-1} \eta_x. \quad (10.73)$$

When there is no coupling ( $\mathbf{C} = 0$ ),  $\eta_a$  and  $\eta_x$  are equal to each other.

## 10.9 Dispersion Calculation

The dispersion ( $\eta$ ) and the dispersion derivative ( $\eta'$ ) are defined by the equations

$$\begin{aligned} \eta_x(s) &\equiv \left. \frac{dx}{dp_z} \right|_s, & \eta'_x(s) &\equiv \left. \frac{d\eta_x}{ds} \right|_s = \left. \frac{dx'}{dp_z} \right|_s \\ \eta_y(s) &\equiv \left. \frac{dy}{dp_z} \right|_s, & \eta'_y(s) &\equiv \left. \frac{d\eta_y}{ds} \right|_s = \left. \frac{dy'}{dp_z} \right|_s \\ \eta_z(s) &\equiv \left. \frac{dz}{dp_z} \right|_s \end{aligned} \quad (10.74)$$

Given the dispersion at a given point, the dispersion at some other point is calculated as follows: Let  $\mathbf{r} = (x, p_x, y, p_y, z, p_z)$  be the reference orbit, around which the dispersion is to be calculated. Let  $\mathbf{V}$  and  $\mathbf{M}$  be the zeroth and first order components of the transfer map between two points labeled 1 and 2. Thus

$$\mathbf{r}_2 = \mathbf{M} \mathbf{r}_1 + \mathbf{V} \quad (10.75)$$

Define the dispersion vector  $\eta$  by

$$\eta = (\eta_x, \eta'_x (1 + p_z), \eta_y, \eta'_y (1 + p_z), \eta_z, 1) \quad (10.76)$$

Differentiating Eq. (10.75) with respect to energy, the dispersion at point 2 in terms of the dispersion at point 1 is

$$\eta_2 = \frac{dp_{z1}}{dp_{z2}} [\mathbf{M} \eta_1] + \mathbf{V}_\eta \quad (10.77)$$

where

$$\mathbf{V}_\eta = \frac{dp_{z1}}{dp_{z2}} \frac{1}{1 + p_{z1}} \begin{pmatrix} M_{12} p_{x1} + M_{14} p_{y1} \\ M_{22} p_{x1} + M_{24} p_{y1} \\ M_{32} p_{x1} + M_{34} p_{y1} \\ M_{42} p_{x1} + M_{44} p_{y1} \\ M_{52} p_{x1} + M_{54} p_{y1} \\ M_{62} p_{x1} + M_{64} p_{y1} \end{pmatrix} - \begin{pmatrix} 0 \\ \frac{p_{x2}}{1 + p_{z2}} \\ 0 \\ \frac{p_{y2}}{1 + p_{z2}} \\ 0 \\ 0 \end{pmatrix} \quad (10.78)$$

The sixth row of the matrix equation gives  $dp_{z1}/dp_{z2}$ . Explicitly

$$\left[ \frac{dp_{z1}}{dp_{z2}} \right]^{-1} = M_{66} + M_{61} \eta_{x1} + M_{62} \eta'_{x1} (1 + p_{z1}) + M_{63} \eta_{y1} + M_{64} \eta'_{y1} (1 + p_{z1}) + M_{62} p_{x1} + M_{64} p_{y1} \quad (10.79)$$

For everything except `Rfcavity` and `Lcavity` elements,  $dp_{z1}/dp_{z2}$  is 1.

## 10.10 Instrumental Measurements

*Bmad* has the ability to simulate instrumental measurement errors for orbit, dispersion, betatron phase, and coupling measurements. The appropriate attributes are listed in §4.10 and the conversion formulas are outlined below.

### 10.10.1 Orbit Measurement

For orbits, the relationship between measured position  $(x, y)_{\text{meas}}$  and true position  $(x, y)_{\text{true}}$  is

$$\begin{pmatrix} x \\ y \end{pmatrix}_{\text{meas}} = n_f \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} + \mathbf{M}_m \left[ \begin{pmatrix} x \\ y \end{pmatrix}_{\text{true}} - \begin{pmatrix} x \\ y \end{pmatrix}_0 \right] \quad (10.80)$$

with

$$\begin{pmatrix} x \\ y \end{pmatrix}_0 = \begin{pmatrix} x_{\text{err}} - x_{\text{cal}} \\ y_{\text{err}} - y_{\text{cal}} \end{pmatrix} \quad (10.81)$$

and

$$\mathbf{M}_m = \begin{pmatrix} g_x \cos(d\theta + d\psi) & g_x \sin(d\theta + d\psi) \\ -g_y \sin(d\theta - d\psi) & g_y \cos(d\theta - d\psi) \end{pmatrix} \quad (10.82)$$

where

$$\begin{aligned} d\psi &= \psi_{\text{err}} - \psi_{\text{cal}} \\ d\theta &= \theta_{\text{err}} - \theta_{\text{cal}} \\ g_x &= 1 + dg_{x,\text{err}} - dg_{x,\text{cal}} \\ g_y &= 1 + dg_{y,\text{err}} - dg_{y,\text{cal}} \end{aligned} \quad (10.83)$$

$r_1$  and  $r_2$  are Gaussian random numbers whose distribution is centered at zero and has unit width.  $n_f$  is the noise factor inherent in the measurement,  $(x, y)_{\text{err}}$  are monitor offset errors and  $(x, y)_{\text{cal}}$  are the offset calibration factors.  $\theta_{\text{err}}$  and  $\phi_{\text{err}}$  are error tilt and “crunch” angles, and  $\theta_{\text{cal}}$  and  $\phi_{\text{cal}}$  are the corresponding calibration angles. Finally,  $dg_{x,\text{err}}$  and  $dg_{y,\text{err}}$  are the horizontal and vertical gain errors, and  $dg_{x,\text{cal}}$  and  $dg_{y,\text{cal}}$  are the corresponding calibration gains.

The calibration variables are useful for simulating the process where a measurement or series of measurements is analyzed to find the values of the error parameters. In this case, the measured position  $(x, y)_m$  represents the beam position corrected for “known” offsets, tilts, and gain errors.

### 10.10.2 Dispersion Measurement

A dispersion measurement is considered to be the result of measuring the orbit at two different energies. The measured values are then

$$\begin{pmatrix} \eta_x \\ \eta_y \end{pmatrix}_{\text{meas}} = \frac{\sqrt{2} n_f}{dE/E} \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} + \mathbf{M}_m \begin{pmatrix} \eta_x \\ \eta_y \end{pmatrix}_{\text{true}} \quad (10.84)$$

The factor of  $\sqrt{2}$  comes from the fact that there are two measurements.

### 10.10.3 Coupling Measurement

The coupling measurement is considered to be the result of measuring the beam at a detector over  $N_s$  turns while the beam oscillates at a normal mode frequency with some amplitude  $A_{\text{osc}}$ . The measured coupling is computed as follows. First, consider excitation of the  $a$ -mode which can be written in the form:

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix}_{\text{true}} = A_{\text{osc}} \begin{pmatrix} \cos \phi_i \\ K_{22a} \cos \phi_i + K_{12a} \sin \phi_i \end{pmatrix}_{\text{true}} \quad (10.85)$$

$i$  is the turn number and  $\phi_i$  is the oscillation phase on the  $i^{\text{th}}$  turn. The coefficients  $K_{22a}$  and  $K_{12a}$  are related to the coupling  $\overline{\mathbf{C}}$  via David and Rubin[9] Eq. 54:

$$\begin{aligned} K_{22a} &= \frac{-\sqrt{\beta_b}}{\gamma \sqrt{\beta_a}} \overline{\mathbf{C}}_{22} \\ K_{12a} &= \frac{-\sqrt{\beta_b}}{\gamma \sqrt{\beta_a}} \overline{\mathbf{C}}_{12} \end{aligned} \quad (10.86)$$

To apply the measurement errors, consider the general case where the beam’s oscillations are split into two components: One component being in-phase with some reference oscillator (which is oscillating with the same frequency as the beam) and a component oscillating out-of-phase:

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix}_{\text{true}} = \begin{pmatrix} q_{a1x} \\ q_{a1y} \end{pmatrix}_{\text{true}} A_{\text{osc}} \cos(\phi_i + d\phi) + \begin{pmatrix} q_{a2x} \\ q_{a2y} \end{pmatrix}_{\text{true}} A_{\text{osc}} \sin(\phi_i + d\phi) \quad (10.87)$$

where  $d\phi$  is the phase of the reference oscillator with respect to the beam. Comparing Eq. (10.85) with Eq. (10.87) gives the relation

$$\begin{aligned} K_{22a} &= \frac{q_{a1x} q_{a1y} + q_{a2x} q_{a2y}}{q_{a1x}^2 + q_{a2x}^2} \\ K_{12a} &= \frac{q_{a1x} q_{a2y} - q_{a2x} q_{a1y}}{q_{a1x}^2 + q_{a2x}^2} \end{aligned} \quad (10.88)$$

This equation is general and can be applied in either the true or measurement frame of reference. Eq. (10.80) can be used to transform  $(x_i, y_i)_{\text{true}}$  in Eq. (10.85) to the measurement frame of reference. Only the oscillating part is of interest. Averaging over many turns gives

$$\begin{pmatrix} q_{a1x} \\ q_{a1y} \end{pmatrix}_{\text{meas}} = \mathbf{M}_m \begin{pmatrix} q_{a1x} \\ q_{a1y} \end{pmatrix}_{\text{true}}, \quad \begin{pmatrix} q_{a2x} \\ q_{a2y} \end{pmatrix}_{\text{meas}} = \mathbf{M}_m \begin{pmatrix} q_{a2x} \\ q_{a2y} \end{pmatrix}_{\text{true}} \quad (10.89)$$

This neglects the measurement noise. A calculation shows that the noise gives a contribution to the measured  $K_{22a}$  and  $K_{12a}$  of

$$K_{22a} \rightarrow K_{22a} + r_1 \frac{n_f}{N_s A_{\text{osc}}}, \quad K_{12a} \rightarrow K_{12a} + r_2 \frac{n_f}{N_s A_{\text{osc}}} \quad (10.90)$$

Using the above equations, the transformation from the true coupling to measured coupling is as follows: From a knowledge of the true  $\overline{\mathbf{C}}$  and Twiss values, the true  $K_{22a}$  and  $K_{12a}$  can be calculated via Eq. (10.86). Since the value of  $d\phi$  does not affect the final answer,  $d\phi$  in Eq. (10.87) is chosen to be zero. Comparing this to Eq. (10.85) gives

$$\begin{pmatrix} q_{a1x} \\ q_{a1y} \end{pmatrix}_{\text{true}} = \begin{pmatrix} 1 \\ K_{22a} \end{pmatrix}_{\text{true}}, \quad \begin{pmatrix} q_{a2x} \\ q_{a2y} \end{pmatrix}_{\text{true}} = \begin{pmatrix} 0 \\ K_{12a} \end{pmatrix}_{\text{true}} \quad (10.91)$$

Now Eq. (10.89) is used to convert to the measured  $q$ 's and Eq. (10.88) then gives the measured  $K_{22a}$  and  $K_{12a}$ . Finally, Applying Eq. (10.90) and then Eq. (10.86) gives the measured  $\overline{\mathbf{C}}_{22}$  and  $\overline{\mathbf{C}}_{12}$ .

A similar procedure can be applied to  $b$ -mode oscillations to calculate values for the measured  $\overline{\mathbf{C}}_{11}$  and  $\overline{\mathbf{C}}_{12}$ .  $K_{11b}$  and  $K_{12b}$  are defined by

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix}_{\text{true}} = A_{\text{osc}} \begin{pmatrix} K_{11b} \cos \phi_i + K_{12b} \sin \phi_i \\ \cos \phi_i \end{pmatrix}_{\text{true}} \quad (10.92)$$

Comparing this to David and Rubin[9] Eq. 55 gives

$$\begin{aligned} K_{11b} &= \frac{\sqrt{\beta_a}}{\gamma \sqrt{\beta_b}} \overline{\mathbf{C}}_{11} \\ K_{12b} &= \frac{-\sqrt{\beta_a}}{\gamma \sqrt{\beta_b}} \overline{\mathbf{C}}_{12} \end{aligned} \quad (10.93)$$

The  $q_{x1b}$ ,  $q_{y1b}$ ,  $q_{x2b}$  and  $q_{y2b}$  are defined by using Eq. (10.87) with the “a” subscript replaced by “b”. The relationship between  $K$  and  $q$  is then

$$\begin{aligned} K_{11b} &= \frac{q_{b1y} q_{b1x} + q_{b2y} q_{b2x}}{q_{b1y}^2 + q_{b2y}^2} \\ K_{12b} &= \frac{q_{b1y} q_{b2x} - q_{b2y} q_{b1x}}{q_{b1y}^2 + q_{b2y}^2} \end{aligned} \quad (10.94)$$

#### 10.10.4 Phase Measurement

Like the coupling measurement, the betatron phase measurement is considered to be the result of measuring the beam at a detector over  $N_s$  turns while the beam oscillates at a normal mode frequency with some amplitude  $A_{\text{osc}}$ . Following the analysis of the previous subsection, the phase  $\phi$  is

$$\begin{pmatrix} \phi_a \\ \phi_b \end{pmatrix}_{\text{meas}} = \begin{pmatrix} \phi_a \\ \phi_b \end{pmatrix}_{\text{true}} + \frac{n_f}{N_s A_{\text{osc}}} \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} - \begin{pmatrix} \tan^{-1} \left( \frac{q_{a2x}}{q_{a1x}} \right) \\ \tan^{-1} \left( \frac{q_{b2y}}{q_{b1y}} \right) \end{pmatrix}_{\text{meas}} \quad (10.95)$$

## 10.11 Bunch Initialization

*Developed by Michael Saelim*

To better visualize the evolution of a particle beam, it is sometimes convenient to initialize the beam with the particles regularly spaced. The following two algorithms are implemented in *Bmad* for such a purpose.

### 10.11.1 Elliptical Phase Space Distribution

To observe nonlinear effects on the beam, it is sometimes convenient to initialize a bunch of particles in a way that puts more particles in the tails of the bunch than one would normally have with the standard method of seeding particles using a Gaussian distribution. In order to preserve the emittance, a distribution with more particles in the tail needs to decrease the charge per tail particle relative to the core. This feature, along with a regular distribution, are contained in the following “**ellipse**” distribution algorithm.

Consider the two dimensional phase space  $(x, p_x)$ . The transformation to action-angle coordinates,  $(J, \phi)$ , is

$$J = \frac{1}{2}[\gamma x^2 + 2\alpha x x' + \beta x'^2] \quad (10.96)$$

$$\tan \phi = \frac{-\beta(x' + \alpha x)}{x} \quad (10.97)$$

The inverse is

$$\begin{pmatrix} x \\ x' \end{pmatrix} = \sqrt{2J} \begin{pmatrix} \sqrt{\beta} & 0 \\ -\frac{\alpha}{\sqrt{\beta}} & -\frac{1}{\sqrt{\beta}} \end{pmatrix} \begin{pmatrix} \cos \phi \\ \sin \phi \end{pmatrix}. \quad (10.98)$$

In action-angle coordinates, the normalized Gaussian phase space distribution,  $\rho(J, \phi)$ , is

$$\rho(J, \phi) = \frac{1}{2\pi\varepsilon} e^{-\frac{J}{\varepsilon}}. \quad (10.99)$$

where the emittance  $\varepsilon$  is just the average of  $J$  over the distribution

$$\varepsilon = \langle J \rangle \equiv \int dJ d\phi J \rho(J, \phi). \quad (10.100)$$

The beam sizes  $\sigma$  and  $\sigma'$  are

$$\sigma = \sqrt{\langle x^2 \rangle} = \sqrt{\varepsilon\beta} \quad (10.101)$$

$$\sigma' = \sqrt{\langle x'^2 \rangle} = \sqrt{\varepsilon\gamma}, \quad (10.102)$$

and the covariance is

$$\langle xx' \rangle = -\varepsilon\alpha. \quad (10.103)$$

The **ellipse** algorithm starts by partitioning phase space into regions bounded by ellipses of constant  $J = B_n$ ,  $n = 0, \dots, N_J$ . The boundary values  $B_n$  are chosen so that, except for the last boundary, the  $\sqrt{B_n}$  are equally spaced

$$B_n = \begin{cases} \frac{\varepsilon}{2} \left( \frac{n_\sigma n}{N} \right)^2 & \text{for } 0 \leq n < N_J \\ \infty & \text{for } n = N_J \end{cases} \quad (10.104)$$

where  $n_\sigma$  is called the “**boundary sigma cutoff**”. Within each region, an elliptical shell of constant  $J_n$  is constructed with  $N_\phi$  particles equally spaced in  $\phi$ . The charge  $q_n$  of each particle of the  $n^{th}$  ellipse is

chosen so that the total charge of all the particles of the ellipse is equal to the total charge within the region

$$N_\phi q_n = \int_{B_{n-1}}^{B_n} dJ \int_0^{2\pi} d\phi \rho(J, \phi) = \exp\left(-\frac{B_{n-1}}{\varepsilon}\right) - \exp\left(-\frac{B_n}{\varepsilon}\right) \quad (10.105)$$

The value of  $J_n$  is chosen to coincide with the average  $J$  within the region

$$N_\phi q_n J_n = \int_{B_{n-1}}^{B_n} dJ \int_0^{2\pi} d\phi J \rho(J, \phi) = \varepsilon(\xi + 1) e^{-\xi} \left| \frac{B_{n-1}}{\varepsilon} \right| \quad (10.106)$$

The **ellipse** phase space distribution is thus

$$\rho_{model}(J, \phi) = q_{tot} \sum_{n=1}^{N_J} q_n \delta(J - J_n) \sum_{m=1}^{N_\phi} \delta(\phi - 2\pi \frac{m}{N_\phi}) \quad (10.107)$$

where  $q_{tot}$  is the total charge. At a given point in the lattice, where the Twiss parameters are known, the input parameters needed to construct the **ellipse** phase space distribution is  $n_\sigma$ ,  $N_J$ ,  $N_\phi$ , and  $q_{tot}$ .

The **ellipse** distribution is two dimensional in nature but can easily be extended to six dimensions.

### 10.11.2 Kapchinsky-Vladimirsky Phase Space Distribution

The Kapchinsky-Vladimirsky (KV) distribution can be thought of as a four dimensional analog of the **ellipse** distribution with only one elliptical shell. Consider a 4D phase space  $(x, x', y, y')$ . Using this framework, a 4D Gaussian distribution is

$$\rho(J_x, \phi_x, J_y, \phi_y) = \frac{1}{(2\pi)^2 \varepsilon_x \varepsilon_y} \exp\left(-\frac{J_x}{\varepsilon_x}\right) \exp\left(-\frac{J_y}{\varepsilon_y}\right) \quad (10.108)$$

$$= \frac{1}{(2\pi)^2 \varepsilon_x \varepsilon_y} \exp\left(-\frac{I_1}{\varepsilon}\right), \quad (10.109)$$

where the orthogonal action coordinates are:

$$I_1 = \left( \frac{J_x}{\varepsilon_x} + \frac{J_y}{\varepsilon_y} \right) \varepsilon \quad (10.110)$$

$$I_2 = \left( -\frac{J_x}{\varepsilon_y} + \frac{J_y}{\varepsilon_x} \right) \varepsilon \quad (10.111)$$

with  $\varepsilon = \left( \frac{1}{\varepsilon_x^2} + \frac{1}{\varepsilon_y^2} \right)^{-1/2}$ . The reverse transformation is:

$$J_x = \left( \frac{I_1}{\varepsilon_x} - \frac{I_2}{\varepsilon_y} \right) \varepsilon \quad (10.112)$$

$$J_y = \left( \frac{I_1}{\varepsilon_y} + \frac{I_2}{\varepsilon_x} \right) \varepsilon. \quad (10.113)$$

The KV distribution is

$$\rho(I_1, I_2, \phi_x, \phi_y) = \frac{1}{A} \delta(I_1 - \xi), \quad (10.114)$$

where  $A = \frac{\varepsilon_x \varepsilon_y}{\varepsilon^2} \xi (2\pi)^2$  is a constant which normalizes the distribution to 1. By choosing a particular  $\xi$ , and iterating over the domain of the three remaining coordinates, one can populate a 3D subspace of constant density.



The range in  $I_2$  to be iterated over is constrained by  $J_x, J_y \geq 0$ . Thus  $I_2$  is in the range  $[-\frac{\epsilon_x}{\epsilon_y} I_1, \frac{\epsilon_y}{\epsilon_x} I_1]$ . This range is divided into  $N$  regions of equal size, with a ring of particles placed in the middle of each region. The angle variables are also constrained to  $\phi_x, \phi_y \in [0, 2\pi]$ , with each range divided into  $M_x$  and  $M_y$  regions, respectively. Each of these regions will have a particle placed in its center.

The weight of a particle is determined by the total weight of the region of phase space it represents. Because the density  $\rho$  is only dependent on  $I_1$ ,

$$q = \int_0^\infty dI_1 \int_{I_2}^{I_2+\Delta I_2} dI_2 \int_{\phi_x}^{\phi_x+\Delta\phi_x} d\phi_x \int_{\phi_y}^{\phi_y+\Delta\phi_y} d\phi_y \frac{1}{A} \delta(I_1 - \xi) \quad (10.115)$$

$$= \frac{1}{A} \Delta I_2 \Delta \phi_x \Delta \phi_y. \quad (10.116)$$

To represent the distribution with particles of equal weight, we must partition  $(I_2, \phi_x, \phi_y)$ -space into regions of equal volume.

The weight of each particle is

$$q = \frac{1}{NM_x M_y} = \frac{1}{\text{number of particles}} \quad (10.117)$$

## 10.12 Macroparticles

*Note: The macroparticle tracking code is not currently maintained in favor of tracking an ensemble of particles.*

A macroparticle[10] is represented by a centroid position  $\bar{\mathbf{r}}$  and a  $6 \times 6$   $\sigma$  matrix which defines the shape of the macroparticle in phase space.  $\sigma_i = \sqrt{\sigma(i, i)}$  is the RMS sigma for the  $i^{th}$  phase space coordinate. For example  $\sigma_z = \sqrt{\sigma(5, 5)}$ .

$\sigma$  is a real, non-negative symmetric matrix. The equation that defines the ellipsoid at a distance of  $n$ -sigma from the centroid is

$$(\mathbf{r} - \bar{\mathbf{r}})^t \sigma^{-1} (\mathbf{r} - \bar{\mathbf{r}}) = n \quad (10.118)$$

where the  $t$  superscript denotes the transpose. Given the sigma matrix at some point  $s = s_1$ , the sigma matrix at a different point  $s_2$  is

$$\sigma_2 = \mathbf{M}_{21} \sigma_1 \mathbf{M}_{21}^t \quad (10.119)$$

where  $\mathbf{M}_{21}$  is the Jacobian of the transport map between points  $s_1$  and  $s_2$ .

The Twiss parameters can be calculated from the sigma matrix. The dispersion is given by

$$\begin{aligned} \sigma(1, 6) &= \eta_x \sigma(6, 6) \\ \sigma(2, 6) &= \eta'_x \sigma(6, 6) \\ \sigma(3, 6) &= \eta_y \sigma(6, 6) \\ \sigma(4, 6) &= \eta'_y \sigma(6, 6) \end{aligned} \quad (10.120)$$

Ignoring coupling for now the betatron part of the sigma matrix can be obtained from the linear equations of motion. For example, using

$$x = \sqrt{2\beta_x \epsilon_x} \cos \phi_x + \eta_x p_z \quad (10.121)$$

Solving for the first term on the RHS, squaring and averaging over all particles gives

$$\beta_x \epsilon_x = \sigma(1, 1) - \frac{\sigma^2(1, 6)}{\sigma(6, 6)} \quad (10.122)$$

It is thus convenient to define the betatron part of the sigma matrix

$$\sigma_\beta(i, j) \equiv \sigma(i, j) - \frac{\sigma(i, 6) \sigma(j, 6)}{\sigma(6, 6)} \quad (10.123)$$

and in terms of the betatron part the emittance is

$$\epsilon_x^2 = \sigma_\beta(1, 1) \sigma_\beta(2, 2) - \sigma_\beta(1, 2)^2 \quad (10.124)$$

and the Twiss parameters are

$$\epsilon_x \begin{pmatrix} \beta_x & -\alpha_x \\ -\alpha_x & \gamma_x \end{pmatrix} = \begin{pmatrix} \sigma_\beta(1, 1) & \sigma_\beta(1, 2) \\ \sigma_\beta(1, 2) & \sigma_\beta(2, 2) \end{pmatrix} \quad (10.125)$$

If there is coupling the transformation between the  $4 \times 4$  transverse normal mode sigma matrix  $\sigma_a$  and the  $4 \times 4$  laboratory matrix  $\sigma_x$  is

$$\sigma_x = \mathbf{V} \sigma_a \mathbf{V}^t \quad (10.126)$$

The sigma matrix is the same for all macroparticles and is determined by the local Twiss parameters:

$$\begin{aligned} \sigma(1, 1) &= \epsilon_x \beta_x \\ \sigma(1, 2) &= -\epsilon_x \alpha_x \\ \sigma(2, 2) &= \epsilon_x \gamma_x = \epsilon_x (1 + \alpha_x^2) / \beta_x \\ \sigma(3, 3) &= \epsilon_y \beta_y \\ \sigma(3, 4) &= -\epsilon_y \alpha_b \\ \sigma(4, 4) &= \epsilon_y \gamma_y = \epsilon_y (1 + \alpha_b^2) / \beta_y \\ \sigma(i, j) &= 0 \quad \text{otherwise} \end{aligned} \quad (10.127)$$

The centroid energy of the  $k^{th}$  macroparticle is

$$E_k = E_b + \frac{(n_{mp} - 2k + 1) \sigma_E N_{\sigma E}}{n_{mp}} \quad (10.128)$$

where  $E_b$  is the central energy of the bunch,  $n_{mp}$  is the number of macroparticles,  $\sigma_E$  is the energy sigma, and  $N_{\sigma E}$  is the number of sigmas in energy that the range of macroparticle energies cover. The charge of each macroparticle is, within a constant factor, the charge contained within the energy region  $E_k - dE_{mp}/2$  to  $E_k + dE_{mp}/2$  assuming a Gaussian distribution where the energy width  $dE_{mp}$  is

$$dE_{mp} = \frac{2 \sigma_E N_{\sigma E}}{n_{mp}} \quad (10.129)$$

## 10.13 Wake fields

### 10.13.1 Short-Range Wakes

Wake field effects are divided into short-range (within a bunch) and long-range (between bunches).

Only the transverse dipole and longitudinal monopole components of the short-range wake field are modeled. The longitudinal monopole energy kick  $dE$  for the  $i^{th}$  macroparticle is computed from the equation

$$\Delta p_z(i) = \frac{-e L}{v P_0} \left( \frac{1}{2} W_{\parallel}^{SR}(0) |q_i| + \sum_{j \neq i} \widetilde{W}_{\parallel}^{SR}(dz_{ij}) |q_j| \right) \quad (10.130)$$

where  $v$  is the particle velocity,  $e$  is the charge on an electron,  $q$  is the macroparticle charge,  $L$  is the cavity length,  $dz_{ij}$  is the longitudinal distance between the  $i^{th}$  and  $j^{th}$  macroparticles,  $W_{\parallel}^{SR}$  is the short-range longitudinal wake field function, and  $\widetilde{W}_{\parallel}^{SR}$  is a modified wake field function

$$\widetilde{W}_{\parallel}^{SR}(dz) = \begin{cases} W_{\parallel}^{SR}(0) \cdot \frac{dz + \sigma_{zij}}{2\sigma_{zij}} & -\sigma_{zij} < dz < \sigma_{zij} \\ W_{\parallel}^{SR}(dz) & \text{otherwise} \end{cases} \quad (10.131)$$

$\widetilde{W}_{\parallel}^{SR}$  is used instead of  $W_{\parallel}^{SR}$  to avoid simulation artifacts is due to the discontinuity of  $W_{\parallel}^{SR}$  at  $dz = 0$ .  $\sigma_{zij}$  is the combined macroparticle length

$$\sigma_{zij} = \sigma_{zi} + \sigma_{zj} + \sigma_{z0} \quad (10.132)$$

where  $\sigma_{zi}$  and  $\sigma_{zj}$  are the longitudinal sizes for the  $i^{th}$  and  $j^{th}$  particles respectively, and  $\sigma_{z0}$  is a small fudge factor needed when  $\sigma_{zi} = \sigma_{zj} = 0$ .

The transverse kick  $\Delta p_x(i)$  for the  $i^{th}$  macroparticle due to the dipole short-range transverse wake field is modeled with the equation

$$\Delta p_x(i) = \frac{-e L \sum_j |q_j| x_j W_{\perp}^{SR}(dz_{ij})}{v P_0} \quad (10.133)$$

There is a similar equation for  $\Delta p_y(i)$ .  $W_{\perp}^{SR}$  is the transverse short-range wake function.

The wake field functions  $W_{\parallel}^{SR}$  and  $W_{\perp}^{SR}$  can be specified in a *Bmad* lattice file using a table of wake vs  $z$  or by using “pseudo” modes where

$$W(z) = \sum_i A_i e^{d_i z} \sin(k_i z + \phi_i) \quad (10.134)$$

The set of mode parameters  $(A_i, d_i, k_i, \phi_i)$  are chosen to fit the calculated wake potential. The advantage of the mode approach is that the calculation time scales as the number of particles  $N$  while the calculation based upon a table scales as  $N^2$ . The disadvantage is that initially there is an extra step in that a fit to the wake potential must be performed to generate the mode parameter values. Furthermore, since the transverse wake typically has a form that looks something like  $\sqrt{-z}$  for small  $z$  (small here means small in magnitude,  $z$  is always negative) a set of pseudo modes may not fit the wake function well at small  $z$ . *Bmad* implements a hybrid scheme where for small  $z$  below some cutoff  $z_{\text{cut}}$  the wake vs  $z$  table is used and for larger  $z$  the pseudo modes are used.

### 10.13.2 Long-Range Wakes

Following Chao[11] Eq. 2.88, The long-range wake fields are characterized by a set of cavity modes. The wake function  $W$  for a mode is given by

$$W(z) = c \left( \frac{R}{Q} \right) e^{kz/2Q} \sin(kz) \quad (10.135)$$

where the wake number  $k$  is related to the mode frequency  $\omega$  by

$$k = \omega/c \quad (10.136)$$

Here  $z$  is the distance behind the particle generating the wake.  $z$  is negative for all particles affected by the wake so  $W(z)$  is a decaying exponential as expected.

Assuming that the macroparticle generating the wake is offset a distance  $r_w$  along the  $x$ -axis, a trailing macroparticle will see a kick

$$\Delta \mathbf{p}_\perp = -C I_m W_m(z) m r^{m-1} (\hat{\mathbf{r}} \cos m\theta - \hat{\theta} \sin m\theta) \quad (10.137)$$

$$= -C I_m W_m(z) m r^{m-1} (\hat{\mathbf{x}} \cos[(m-1)\theta] - \hat{\mathbf{y}} \sin[(m-1)\theta])$$

$$\Delta p_z = -C I_m W'_m(z) r^m \cos m\theta \quad (10.138)$$

where  $m$  is the order of the mode,  $C$  is given by

$$C = \frac{e}{v P_0} \quad (10.139)$$

and

$$I_m = q_w r_w^m \quad (10.140)$$

with  $q_w$  being the charge on the particle. Generalizing the above, a macroparticle at  $(r_w, \theta_w)$  will generate a wake

$$-\Delta p_x + i\Delta p_y = C I_m W(z) m r^{m-1} e^{-im\theta_w} e^{i(m-1)\theta} \quad (10.141)$$

$$\Delta p_z = C I_m W'(z) r^m \cos[m(\theta - \theta_w)] \quad (10.142)$$

Comparing Eq. (10.141) to (10.9), and using the relationship between kick and field as given by (10.5) and (10.6), shows that the form of the wake field transverse kick is the same as for a multipole of order  $n = m - 1$ .

The wake field felt by a particle is due to the wake fields generated by all the particles ahead of it. If the wake field kicks are computed by summing over all particle pairs, the computation will scale as  $N^2$  where  $N$  is the number of particles. This quickly becomes computationally exorbitant. A better solution is to keep track of the wakes in a cavity. When a particle comes through, the wake it generates is simply added to the existing wake. This computation scales as  $N$  and makes simulations with large number of particles practical.

To add wakes together, A wake must be decomposed into its components. Spatially, there are normal and skew components and temporally there are sin and cosine components. This gives 4 components which will be labeled  $a_{\cos}$ ,  $a_{\sin}$ ,  $b_{\cos}$ , and  $b_{\sin}$ . For a mode of order  $m$ , a particle passing through at a distance  $z_w$  with respect to the reference particle will produce wake components

$$\begin{aligned} \delta a_{\sin, m} &= c \left( \frac{R}{Q} \right) e^{-kz_w/2Q} \cos(kz_w) I_m \sin(m\theta_w) \\ \delta a_{\cos, m} &= -c \left( \frac{R}{Q} \right) e^{-kz_w/2Q} \sin(kz_w) I_m \sin(m\theta_w) \\ \delta b_{\sin, m} &= c \left( \frac{R}{Q} \right) e^{-kz_w/2Q} \cos(kz_w) I_m \cos(m\theta_w) \\ \delta b_{\cos, m} &= -c \left( \frac{R}{Q} \right) e^{-kz_w/2Q} \sin(kz_w) I_m \cos(m\theta_w) \end{aligned} \quad (10.143)$$

These are added to the existing wake components. The total is

$$a_{\sin, m} = \sum_{\text{particles}} \delta a_{\sin, m} \quad (10.144)$$

with similar equations for  $a_{\cos, m}$  etc. Here the sum is over all particles that cross the cavity before the kicked particle. To calculate the kick due to wake, the normal and skew components are added together

$$\begin{aligned} a_m &= e^{kz/2Q} (a_{\cos, m} \cos(kz) + a_{\sin, m} \sin(kz)) \\ b_m &= e^{kz/2Q} (b_{\cos, m} \cos(kz) + b_{\sin, m} \sin(kz)) \end{aligned} \quad (10.145)$$

Here  $z$  is the distance of the particle with respect to the reference particle. In analogy to Eq. (10.141) and (10.142), the kick is

$$-\Delta p_x + i\Delta p_y = C m (b_m + ia_m) r^{m-1} e^{i(m-1)\theta} \quad (10.146)$$

$$\Delta p_z = C r^m ((b'_m + ia'_m)e^{im\theta} + (b'_m - ia'_m)e^{-im\theta}) \quad (10.147)$$

where  $a' \equiv da/dz$  and  $b' \equiv db/dz$ .

When simulating trains of bunches, the exponential factor  $-kz_w/2Q$  in Eq. (10.143) can become very large. To prevent numerical overflow, *Bmad* uses a reference time  $z_{\text{ref}}$  so that all longitudinal positions  $z$  in the above equations are replaced by

$$z \longrightarrow z - z_{\text{ref}} \quad (10.148)$$

The above equations were developed assuming cylindrical symmetry. With cylindrical symmetry, the cavity modes are actually a pair of degenerate modes. When the symmetry is broken, the modes no longer have the same frequency. In this case, one has to consider a mode's polarization angle  $\phi$ . Equations (10.145) and (10.146) are unchanged. In place of Eq. (10.143), the contribution of a particle to a mode is

$$\begin{aligned} \delta a_{\text{sin},m} &= c \left( \frac{R}{Q} \right) e^{-kz_w/2Q} \cos(kz_w) I_m [\sin(m\theta_w) \sin^2(m\phi) + \cos(m\theta_w) \sin(m\phi) \cos(m\phi)] \\ \delta a_{\text{cos},m} &= -c \left( \frac{R}{Q} \right) e^{-kz_w/2Q} \sin(kz_w) I_m [\sin(m\theta_w) \sin^2(m\phi) + \cos(m\theta_w) \sin(m\phi) \cos(m\phi)] \\ \delta b_{\text{sin},m} &= c \left( \frac{R}{Q} \right) e^{-kz_w/2Q} \cos(kz_w) I_m [\cos(m\theta_w) \cos^2(m\phi) + \sin(m\theta_w) \sin(m\phi) \cos(m\phi)] \\ \delta b_{\text{cos},m} &= -c \left( \frac{R}{Q} \right) e^{-kz_w/2Q} \sin(kz_w) I_m [\cos(m\theta_w) \cos^2(m\phi) + \sin(m\theta_w) \sin(m\phi) \cos(m\phi)] \end{aligned} \quad (10.149)$$

Each mode is characterized by an  $R/Q$ ,  $Q$ ,  $\omega$ , and  $m$ . Notice that  $R/Q$  is defined so that it includes the cavity length. Thus the long-range wake equations, as opposed to the short-range ones, do not have any explicit dependence on  $L$ .

To make life more interesting, different people define  $R/Q$  differently. A common practice is to define an  $R/Q$  “at the beam pipe radius”. In this case the above equations must be modified to include factors of the beam pipe radius. Another convention uses a “linac definition” which makes  $R/Q$  twice as large and adds a factor of 2 in Eq. (10.135) to compensate.

## 10.14 Synchrotron Radiation Integrals

The synchrotron radiation integrals are used to compute emittances, the energy spread, etc. The standard formulas assume no coupling between the horizontal and vertical planes [12, 8]. With coupling, the equations need to be generalized and this is detailed below.

In the general case the curvature function  $\mathbf{G} = (G_x, G_y)$ , which points away from the center of curvature of the particle's orbit (see Figure 9.1), does not lie in the horizontal plane.  $\mathbf{G}$  has a magnitude  $G = 1/\rho$  and a positive  $G_y$  indicates a downward bend in the negative  $y$  direction. Similarly, the dispersion  $\eta = (\eta_x, \eta_y)$  will not lie in the horizontal plane. With this notation, the synchrotron Integrals for

coupled motion are:

$$I_0 = \oint ds \gamma_0 G \quad (10.150)$$

$$I_1 = \oint ds \mathbf{G} \cdot \boldsymbol{\eta} \equiv \oint ds (G_x \eta_x + G_y \eta_y) \quad (10.151)$$

$$I_2 = \oint ds G^2 \quad (10.152)$$

$$I_3 = \oint ds G^3 \quad (10.153)$$

$$I_{4a} = \oint ds [G^2 \mathbf{G} \cdot \boldsymbol{\eta}_a + \nabla G^2 \cdot \boldsymbol{\eta}_a] \quad (10.154)$$

$$I_{4b} = \oint ds [G^2 \mathbf{G} \cdot \boldsymbol{\eta}_b + \nabla G^2 \cdot \boldsymbol{\eta}_b] \quad (10.155)$$

$$I_{4z} = \oint ds [G^2 \mathbf{G} \cdot \boldsymbol{\eta} + \nabla G^2 \cdot \boldsymbol{\eta}] \quad (10.156)$$

$$I_{5a} = \oint ds G^3 \mathcal{H}_a \quad (10.157)$$

$$I_{5b} = \oint ds G^3 \mathcal{H}_b \quad (10.158)$$

where  $\gamma_0$  is that usual relativistic factor and  $\mathcal{H}_a$  is

$$\mathcal{H}_a = \gamma_a \eta_a^2 + 2 \alpha_a \eta_a \eta'_a + \beta_a \eta_a'^2 \quad (10.159)$$

with a similar equation for  $\mathcal{H}_b$ . Here  $\boldsymbol{\eta}_a = (\eta_{ax}, \eta_{ay})$ , and  $\boldsymbol{\eta}_b = (\eta_{bx}, \eta_{by})$  are the dispersion vectors for the  $a$  and  $b$  modes respectively in  $x$ - $y$  space (these 2-vectors are not to be confused with the dispersion 4-vectors used in the previous section). The position dependence of the curvature function is:

$$\begin{aligned} G_x(x, y) &= G_x + x k_1 + y s_1 \\ G_y(x, y) &= G_y + x s_1 - y k_1 \end{aligned} \quad (10.160)$$

where  $k_1$  is the quadrupole moment and  $s_1$  is the skew-quadrupole moment. Using this gives on-axis ( $x = y = 0$ )

$$\nabla G^2 = 2 (G_x k_1 + G_y s_1, G_x s_1 - G_y k_1) \quad (10.161)$$

Note:  $I_0$  is not a standard radiation integral. It is useful, though, in calculating the average number of photons emitted.

In a dipole a non-zero  $e_1$  or  $e_2$  gives a contribution to  $I_4$  via the  $\nabla G^2 \cdot \boldsymbol{\eta}$  term. The edge field is modeled as a thin quadrupole of length  $\delta$  and strength  $k = -\tan(e)/\delta$ . It is assumed that  $\mathbf{G}$  rises linearly within the edge field from zero on the outside edge of the edge field to its full value on the inside edge of the edge field. Using this in Eq. (10.161) and integrating over the edge field gives the contribution to  $I_4$  from a non-zero  $e_1$  as

$$I_{4z} = -\tan(e_1) G^2 (\cos(\theta) \eta_x + \sin(\theta) \eta_y) \quad (10.162)$$

With an analogous equation for a finite  $e_2$ . The extension to  $I_{4a}$  and  $I_{4b}$  involves using  $\boldsymbol{\eta}_a$  and  $\boldsymbol{\eta}_b$  in place of  $\boldsymbol{\eta}$ . In Eq. (10.162)  $\theta$  is the tilt angle which is non-zero if the bend is not in the horizontal plane.

The above integrals are invariant under rotation of the  $(x, y)$  coordinate system and reduce to the standard equations when  $G_y = 0$  as they should.

There are various parameters that can be expressed in terms of these integrals. The  $I_1$  integral can be related to the momentum compaction  $\alpha_p$  via

$$I_1 = \alpha_p L \quad (10.163)$$

where  $L$  is the storage ring circumference. The energy loss per turn is

$$U_0 = \frac{2 r_e E_0^4}{3 m c^2} I_2 \quad (10.164)$$

where  $E_0$  is the nominal energy and  $r_e$  is the classical electron radius (electrons are assumed here but the formulas are easily generalized).

The damping partition numbers are

$$J_a = 1 - \frac{I_{4a}}{I_2}, \quad J_b = 1 - \frac{I_{4b}}{I_2}, \quad \text{and} \quad J_z = 2 + \frac{I_{4z}}{I_2}. \quad (10.165)$$

Since

$$\eta_a + \eta_b = \eta, \quad (10.166)$$

Robinson's theorem,  $J_a + J_b + J_z = 4$ , is satisfied. Alternatively, the exponential damping coefficients per turn are

$$\alpha_a = \frac{U_0 J_a}{2E_0}, \quad \alpha_b = \frac{U_0 J_b}{2E_0}, \quad \text{and} \quad \alpha_z = \frac{U_0 J_z}{2E_0}. \quad (10.167)$$

The energy spread is given by

$$\sigma_{pz}^2 = \left( \frac{\sigma_E}{E_0} \right)^2 = C_q \gamma_0^2 \frac{I_3}{2I_2 + I_{4z}} \quad (10.168)$$

where  $\gamma_0$  is the usual energy factor and

$$C_q = \frac{55}{32\sqrt{3}} \frac{\hbar}{mc} = 3.84 \times 10^{-13} \text{ meter for electrons} \quad (10.169)$$

If the synchrotron frequency is not too large the bunch length is given by

$$\sigma_z = \frac{I_1}{M(6,5)} \sigma_{pz} \quad (10.170)$$

where  $M(6,5)$  is the  $(6,5)$  element for the 1-turn transfer matrix of the storage ring. Finally the emittances are given by

$$\begin{aligned} \epsilon_a &= C_q \gamma_0^2 \frac{I_{5a}}{I_2 - I_{4a}} \\ \epsilon_b &= C_q \gamma_0^2 \frac{I_{5b}}{I_2 - I_{4b}} \end{aligned} \quad (10.171)$$

For a non-circular machine, radiation integrals are still of interest if there are bends or steering elements but, in this case, the appropriate energy factors must be included to take account any changes in energy due to any **LCavity** elements. The  $I_1$  integral is not altered and the  $I_4$  integrals are not relevant. The other integrals become

$$L_2 = \int ds G^2 \gamma_0^4 \quad (10.172)$$

$$L_3 = \int ds G^3 \gamma_0^7 \quad (10.173)$$

$$L_{5a} = \int ds G^3 \mathcal{H}_a \gamma_0^6 \quad (10.174)$$

$$L_{5b} = \int ds G^3 \mathcal{H}_b \gamma_0^6 \quad (10.175)$$

In terms of these integrals, the energy loss through the lattice is

$$U_0 = \frac{2r_e mc^2}{3} L_2 \quad (10.176)$$

The energy spread assuming  $\sigma_E$  is zero at the start and neglecting any damping is

$$\sigma_E^2 = \frac{4}{3} C_q r_e (mc^2)^2 L_3 \quad (10.177)$$

and, again neglecting any initial beam width, the transverse beam size at the end of the lattice is

$$\begin{aligned} \epsilon_a &= \frac{2}{3} C_q r_e \frac{L_{5a}}{\gamma_f} \\ \epsilon_b &= \frac{2}{3} C_q r_e \frac{L_{5b}}{\gamma_f} \end{aligned} \quad (10.178)$$

Where  $\gamma_f$  is the final gamma.

## 10.15 Spin Dynamics

The matrix representation of the observable corresponding to the measurement of spin along the unit vector  $\mathbf{u}$  in the  $|+\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ ,  $|-\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$  basis is

$$S_{\mathbf{u}} = \mathbf{S} \cdot \mathbf{u} \quad (10.179)$$

$$= \frac{\hbar}{2} \sigma_x \sin \theta \cos \phi + \frac{\hbar}{2} \sigma_y \sin \theta \sin \phi + \frac{\hbar}{2} \sigma_z \cos \theta \quad (10.180)$$

$$= \frac{\hbar}{2} \begin{pmatrix} \cos \theta & \sin \theta e^{-i\phi} \\ \sin \theta e^{i\phi} & -\cos \theta \end{pmatrix} \quad (10.181)$$

where  $\theta$  and  $\phi$  are the polar angles characterizing the vector  $\mathbf{u}$  and  $\sigma_{x,y,z}$  are the three Pauli matrices. The expectation value of this operator representing the spin of a particle satisfies the equation of motion of a classical spin vector in the particle's instantaneous rest frame. The classical spin vector  $\mathbf{s}$  is described for time  $t$  in the laboratory frame by the Thomas-Bargmann-Michel-Telegdi (T-BMT) equation,

$$\frac{d}{dt} \mathbf{s} = \boldsymbol{\Omega}_{BMT}(\mathbf{r}, \mathbf{p}, \mathbf{t}) \times \mathbf{s}, \quad (10.182)$$

$$\boldsymbol{\Omega}_{BMT}(\mathbf{r}, \mathbf{p}, \mathbf{t}) = -\frac{q}{m\gamma} \left[ (1 + G\gamma) \mathbf{B} - \frac{G\mathbf{p} \cdot \mathbf{B}}{(\gamma + 1)m^2 c^2} \mathbf{p} - \frac{1}{mc^2} \left( G + \frac{1}{1 + \gamma} \right) \mathbf{p} \times \mathbf{E} \right], \quad (10.183)$$

where  $\mathbf{E}(\mathbf{r}, t)$  and  $\mathbf{B}(\mathbf{r}, t)$  are the electric and magnetic fields in the laboratory frame,  $\mathbf{p}$  and  $\gamma$  are the particle's momentum and relativistic gamma factor in the laboratory frame,  $q$  and  $m$  are the particle's charge and mass, and  $G = \frac{(g-2)}{2}$  is the particle's anomalous gyro-magnetic g-factor which is 1.793 for protons and 0.00116 for electrons and positrons.

For a distribution of spins the polarization  $P$  along the unit vector  $\mathbf{u}$  is defined as the absolute value of the average expectation value of the spin over all  $N$  particles times  $\frac{2}{\hbar}$ ,

$$P = \frac{2}{\hbar} \left\| \frac{1}{N} \sum_{j=1}^N \langle + | S_{\mathbf{u}} | + \rangle \right\| \quad (10.184)$$

$$= \left\| \frac{1}{N} \sum_{j=1}^N \cos \theta \right\|. \quad (10.185)$$



It is more efficient to use the SU(2) representation rather than SO(3) when describing rotations of spin. In the SU(2) representation, a spin  $\mathbf{s}$  is written as a spinor  $\Psi = (\psi_1, \psi_2)^T$  where  $\psi_{1,2}$  are complex numbers and

$$\mathbf{s} = \Psi^\dagger \boldsymbol{\sigma} \Psi \quad (10.186)$$

$$\leftrightarrow \Psi = \frac{1}{\sqrt{2(s_3 + 1)}} \begin{pmatrix} 1 + s_3 \\ s_1 + i s_2 \end{pmatrix}, \quad (10.187)$$

or in polar coordinates,

$$\Psi = \begin{pmatrix} \psi_1 \\ \psi_2 \end{pmatrix} = e^{i\xi} \begin{pmatrix} \cos \frac{\phi}{2} \\ \sin \frac{\phi}{2} e^{i\phi} \end{pmatrix} \quad (10.188)$$

$$\leftrightarrow \mathbf{s} = \begin{pmatrix} \sin \theta \cos \phi \\ \sin \theta \sin \phi \\ \cos \theta \end{pmatrix}. \quad (10.189)$$

The  $\xi$  being an extra phase factor. Due to the unitarity of the spin vector,  $|\psi_1|^2 + |\psi_2|^2 = 1$ .

In spinor notation the T-BMT equation can be written

$$\frac{d}{dt} \Psi = -\frac{i}{2} (\boldsymbol{\sigma} \cdot \boldsymbol{\Omega}) \Psi. \quad (10.190)$$

The solution leads to a rotation of the spin vector by an angle  $\alpha$  around a unit vector  $\mathbf{e}$  represented as

$$\Psi = e^{-i\frac{\alpha}{2}\mathbf{e} \cdot \boldsymbol{\sigma}} \Psi_i \quad (10.191)$$

$$= (a_0 \mathbf{1}_2 - i \mathbf{a} \cdot \boldsymbol{\sigma}) \Psi_i \quad (10.192)$$

$$= \mathbf{A} \Psi_i. \quad (10.193)$$

The SU(2) matrix  $\mathbf{A}$  is called a *quaternion* and has the normalization condition  $a_0^2 + \mathbf{a}^2 = 1$ . The three components  $(a_1, a_2, a_3)$ , along with the normalization condition, describes the transport of spin between any two points in an accelerator.

## 10.16 Coherent Synchrotron Radiation

First a definition of some terms to avoid confusion: **space charge** (SC) and **Coherent Synchrotron Radiation** (CSR) deal with the same problem which is the direct interaction between the particles of a bunch. This is to be differentiated from the indirect **wake field** effects which are mediated by the vacuum chamber within which a particle beam moves. The difference between SC and CSR is that a CSR calculation takes into account the fact that there is a delay time, due to the finite velocity of the speed of light, so that the field felt by some particle at time  $t$  due to another (source) particle is based on the source particle's position at some retarded time  $t' \neq t$ . A SC calculation, on the other hand, assumes that the field produced by a particle at time  $t$  can be computed from that particle's positions at time  $t$ . Which type of calculation is better depends upon what is to be simulated. Generally, the SC calculation is preferred at lower energies where a particle's velocity is significantly different from the speed of light.

*Bmad* simulates coherent synchrotron radiation (CSR) using the formalism developed by Sagan[13]. This formalism divides the total kick received by a particle due to another particle into two parts: One part is called the **longitudinal space charge** (LSC) kick and the other component is the **coherent synchrotron radiation** (CSR) kick. By definition, the LSC component is the kick that would result if

both particles were traveling in a straight line. The CSR component is what is left when the LSC kick is subtracted off from the total kick. Generally, the LSC kick is negligible compared to the CSR kick at large enough particle energies.

Transport through a lattice element involves a beam of particles. The lattice element is divided up into a number of slices. Transport through a slice is a two step process. The first step is to give all the particles a kick due to the CSR. The second step is transport of all particles without any interaction between particles. Note that only the longitudinal CSR kick is implemented and transverse kicks are ignored.

The particle-particle kick is calculated by dividing the bunch longitudinally into a number of bins. To smooth the computed bin densities, each particle of the bunch is considered to have a triangular density distribution as shown in Figure 10.1. The particle density of a bin is calculated by summing the contribution from all the particles. The contribution of a given particle to a given bin is calculated from the overlap of the particle's triangular density distribution with the bin. For the CSR kick, the density is actually calculated for a second set of staggered bins that have been offset by  $1/2$  the bin width with respect to the first set. This gives the density at the edges of the original set of bins. The density is considered to vary linearly between the computed density points. For a description of the parameters that affect the CSR calculation see Section §8.3.

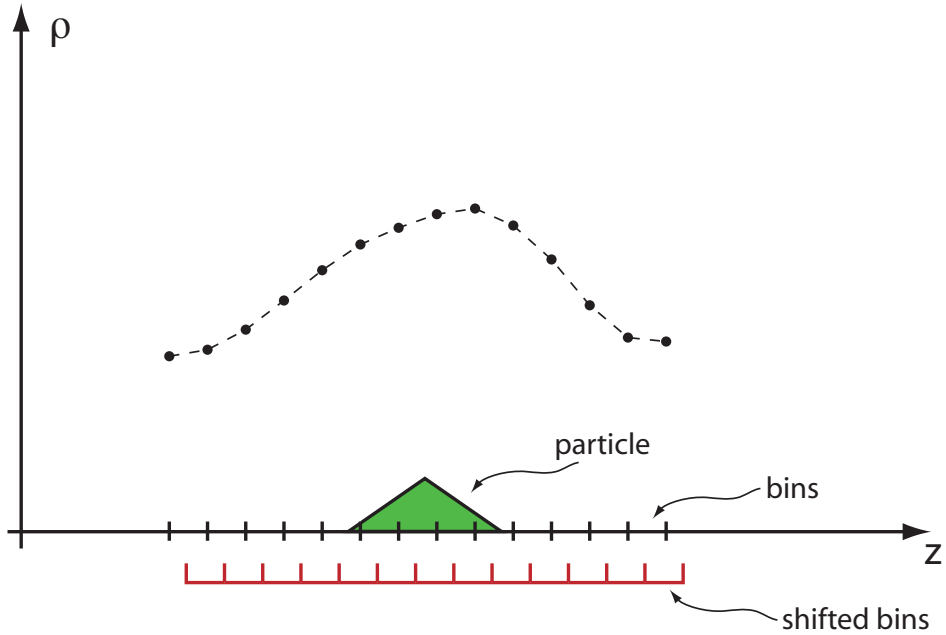


Figure 10.1: The Coherent Synchrotron Radiation kick is calculated by dividing longitudinally a bunch into a number of bins. To smooth the computed densities, each particle of the bunch is considered to have a triangular density distribution.

## Part III

# Programmer's Guide



## Chapter 11

# Bmad Programming Overview

### 11.1 The Bmad Libraries

The code that goes into a program based upon *Bmad* is divided up into a number of libraries. The *Bmad* web site has general information on the organization of these libraries including information on obtaining and compiling programs. The *Bmad* web site is at:

<http://www.lepp.cornell.edu/~dcs/bmad>

The *Bmad* libraries are divided into two groups. One group of libraries contains the Cornell developed code. The “**package**” libraries consist of non-Cornell code that *Bmad* relies upon.

The Cornell developed code can further be divided into Cesr storage ring specific code and non Cesr specific code. The Cesr specific code will not be discussed in this Manual.

The Cornell non Cesr specific code libraries are: subsidiary libraries are:

**sim\_utils** The **sim\_utils** library contains a set of miscellaneous helper routines. Included are routines for string manipulation, file manipulation, matrix manipulation, spline fitting, Gaussian random number generation, etc.

**bmad** The **bmad** library contains the routines for relativistic charged particle simulation including particle tracking, Twiss calculations, symplectic integration, etc., etc.

The **package** libraries are:

**forest** This is the PTC/FPP (Polymorphic Tracking Code / Fully Polymorphic Package) library of Etienne Forest that handles Taylor maps to any arbitrary order (this is also known as Truncated Power Series Algebra (TPSA)). See Chapter 19 for more details. FPP/PTC is a very general package and *Bmad* only makes use of a small part of its features. For more information see the FPP/PTC web site at

<http://acc-physics.kek.jp/forest/PTC/Introduction.html>

**recipes** Numerical Recipes is a set of subroutines for doing scientific computing including Runge–Kutta integration, FFTs, interpolation and extrapolation, etc., etc. The documentation for this library is the books “Numerical Recipes, in Fortran, The Art of Scientific Computing” and “Numerical Recipes in Fortran90, the Art of Parallel Scientific Computing”[18]. The first book explains how the subroutines work and the second book explains what the argument lists for the Fortran90

version of the subroutines are. You do need both books if you want to use Numerical Recipes. For *Bmad* this library has been modified to handle both single or double precision reals as needed in the other libraries (See §11.3).

**PGPLOT** The PGPLOT Graphics Subroutine Library is a Fortran or C-callable, device-independent graphics package for making simple scientific graphs. Documentation including a user’s manual may be obtained from the PGPLOT web site at

<http://www.astro.caltech.edu/~tjp/pgplot>.

One disadvantage of PGPLOT for the programmer is that it is not the most user friendly. To remedy this, there is a set of Fortran90 wrapper subroutines called `quick_plot`. The `quick_plot` suite is part of the `sim_utils` library and is documented in Chapter 21.

**XSIF** XSIF is a library from SLAC to read in XSIF format files. See §1.1 for more details. The only *Bmad* routine to use this library is `xsif_parser`.

## 11.2 getf and listf

As can be seen from the program example in Chapter 12 there is a lot going on behind the scenes even for this simple program. This shows that programming with *Bmad* can be both easy and hard. Easy in the sense that a lot can be done with just a few lines. The hard part comes about since there are many details that have to be kept in mind in order to make sure that the subroutines are calculating what you really want them to calculate.

To help with the details, all *Bmad* subroutines have in their source (.f90) files a comment block that explains the arguments needed by the subroutines and explains what the subroutine does. To help quickly access these comments, there are two Perl scripts that are supplied with the *Bmad* distribution that are invoked with the commands `listf` and `getf`.

The `getf` command is used to locate routines and structures, and to type out information on them. The form of the command is

```
getf <name>
```

This searches for any routine or structure with the name <name>. <name> may contain the wild-cards “\*” and “.” where “\*” matches to any number of characters and “.” matches to any single character. For example:

```
getf bmad_parser
getf lat_struct
getf twiss_at_.
```

The third line in this example will match to the routine `twiss_at_s` but not the routine `twiss_at_start`. You may or may not have to put quotation marks if you use wild card characters. As an example, the command `getf twiss_struct` produces:

```
/home/cesrplib/cesr_libs/devel/cvssrc/bmad/modules/twiss_mod.f90
type twiss_struct
  real(rp) beta, alpha, gamma, phi, eta, etap
  real(rp) sigma, emit
end type
```

The first line shows the file where the structure is located (This is system and user dependent so don’t be surprised if you get a different directory when you use `getf`). The rest of the output shows the definition of the `twiss_struct` structure. The result of issuing the command `getf element_locator` is:

```

/home/cesrplib/cesr_libs/devel/cvssrc/bmad/code/element_locator.f90
!+
! Subroutine element_locator (ele_name, lat, ix_ele)
!
! Subroutine to locate an element in a lat.
!
! Modules Needed:
!   use bmad
!
! Input:
!   ele_name -- Character*16: Name of the element to find.
!   lat      -- Lat_struct: Lat to search through
!
! Output:
!   ix_ele -- Integer: Index of element in lat%ele(:) array.
!           ix_ele set to -1 if not found.
!-

```

The first line again shows in what file the subroutine is located. The rest of the output explains what the routine does and how it can be called.

The `listf` command is like the `getf` command except that only the file name where a routine or structure is found is printed. The `listf` command is useful if you want to just find out where a routine or structure definition lives. For example, the `listf *locator*` command would produce

```

/home/cesrplib/cesr_libs/devel/cvssrc/bmad/code/element_locator.f90
/home/cesrplib/cesr_libs/devel/cvssrc/bmad/code/elements_locator.f90
/home/cesrplib/cesr_libs/devel/cvssrc/cesr_utils/cesr_locator.f90

```

The way `getf` and `listf` work is that they search a list of directories to find the `bmad`, `sim_utils`, and `tao` libraries. Currently the libraries in the *Bmad* distribution that were not developed at Cornell are not searched. This is primarily due to the fact that, to save time, `getf` and `listf` make assumptions about how documentation is arranged in a file and the non-Cornell libraries do not follow this format.

## 11.3 Precision

*Bmad* comes in two flavors: One version where the real numbers are single precision and a second version with double precision reals. Which version you are working with is controlled by the kind parameter `rp` (Real Precision) which is defined in the `precision_def` module. [Note: For compatibility with older programs the kind parameter `rdef` is defined to be equal to `rp`.] On most platforms single precision translates to `rp = 4` and double precision to `rp = 8`. The double precision version is used by default since round-off errors can be significant in some calculations. Long-term tracking is an example where the single precision version is not adequate. Changing the precision means recompiling all the libraries except PTC and PGPLOT. You cannot mix and match. Either you are using the single precision version or you are using the double precision version.

To define floating point variables in Fortran with the correct precision, use the syntax “`real(rp)`”. For example:

```
real(rp) var1, var2, var3
```

When you want to define a literal constant, for example to pass an argument to a subroutine, add the suffix `_rp` to the end of the constant. For example

```
var1 = 2.0_rp * var2
call my_sub (var1, 1.0e6_rp)
```

Note that `2_rp` is different from `2.0_rp`. `2_rp` is an integer of kind `rp`, not a real.

Independent of the setting of `rp`, the parameters `sp` and `dp` are defined to give single and double precision numbers respectively.

## 11.4 Programming Conventions

*Bmad* subroutines follow the following conventions:

**A “\$” suffix denotes a parameter:** A “\$” at the end of a name denotes an integer parameter. For example, in the above program, to check whether an element is a quadrupole one would write:

```
if (lat%ele(i)%key == quadrupole$) ...
```

Checking the source code one would find in the module `bmad_struct`

```
integer, parameter :: drift$ = 1, sbend$ = 2, quadrupole$ = 3, group$ = 4
```

One should always use the parameter name instead of the integer it represents. That is, one should never write

```
if (lat%ele(i)%key == 3) ... ! DO NOT DO THIS!
```

For one, using the name makes the code clearer. However, more importantly, the integer value of the parameters may at times be shuffled for practical internal reasons. The use of the integer value could thus lead to disastrous results.

**Structure names have a “\_struct” suffix:** For example: `lat_struct`, `ele_struct`, etc. Structures without a `_struct` are usually part of Etienne’s PTC/FPP package.

## 11.5 Manual Notation

*Bmad* defines a number of structures and these structures may contain components which are structures, etc. In order to keep the text in this manual succinct when referring to components, the enclosing structure name may be dropped. For example, the `lat_struct` structure looks like

```
type lat_struct
  character(40) name
  type (mode_info_struct) a, b, z
  type (lat_param_struct) param
  type (ele_struct), pointer :: ele(:)
  type (branch_struct), allocatable :: branch(:)
  ... etc. ...
end type
```

In this example, “`%a`” could be used to refer to, the `a` component of the `lat_struct`. To make it explicit that this is a component of a `lat_struct`, “`lat_struct%a`” is an alternate possibility. Since the vast majority of structures have the “\_struct” suffix, this may be shortened to “`lat%a`”. A similar notation works for subcomponents. For example, a `branch_struct` looks like



```
type branch_struct
  character(40) name
  integer ix_from_ele          ! Index of branching element
  integer, pointer :: n_ele_track ! Number of tracking elements
  integer, pointer :: n_ele_max
  type (ele_struct), pointer :: ele(:) ! Element array
  ... etc. ...
end type
```

The `ele` component of the `branch` component of the `lat_struct` can be referred to using “`lat%branch%ele`”, “`%branch%ele`”, or “`%ele`”. Potentially, the last of these could be confused with the “`lat%ele`” component so “`%ele`” would only be used if the meaning is unambiguous in the context.



## Chapter 12

# Introduction to Bmad programming

To get the general feel for how *Bmad* works before getting into the nitty-gritty details in subsequent chapters, this chapter analyzes an example test program.

### 12.1 A First Program

Consider the example program shown in Figure 12.1. This program is provided with the *Bmad* distribution in a file called `simple_program.f90`. `simple_program.f90` is in the `bmad` library area in a directory called `bmad/simple_program`.

### 12.2 Explanation of the Simple\_Program

A line by line explanation of the example program follows. The `use bmad` statement at line 3 defines the *Bmad* structures and defines the interfaces (argument lists) for the *Bmad* subroutines. In particular, the `lat_struct` (§14.2) structure (line 5) holds all of the lattice information: The list of elements, their attributes, etc. `bmad_parser` (line 13) is the routine which parses a lattice file and transfers the information to a `lat_struct` variable. To get a listing of the `lat_struct` components or to find out more about `bmad_parser` use the `getf` command as discussed in §11.2.

After `bmad_parser` is called, the program checks if the lattice is circular (§7.1) and, if so, uses the routine `twiss_at_start` (line 15) to multiply the transfer matrices of the individual elements together to form the 1-turn matrix from the start of the lat back to the start. From this matrix `twiss_at_start` calculates the Twiss parameters at the start of the lattice and puts the information into `lat%ele(0)` (§16.2). The next call, to `twiss_propagate_all`, takes the starting Twiss parameters and, using the transfer matrices of the individual elements, calculates the Twiss parameters at all the elements. Notice that if the lattice is not circular, The starting Twiss parameters will need to have been defined in the lattice file.

The program is now ready to print out some information on the first 11 elements in the lattice which it does on lines 20 through 24 of the program. The do-loop is over the array `lat%ele(:)`. Each element of the array holds the information about an individual lattice element as explained in Chapter 14. The `lat%ele(0)` element is basically a marker element to denote the beginning of the array (§6). Using the pointer `ele` to point to the individual elements (line 22) makes for a cleaner syntax and reduces typing. The table that is produced is shown in lines 1 through 12 of Figure 12.2. The first column is

```

1  program test
2
3      use bmad                ! Define the structures we need to know about.
4      implicit none
5      type (lat_struct), target :: lat    ! This structure holds the lattice info
6      type (ele_struct), pointer :: ele
7      type (ele_pointer_struct), allocatable :: eles(:)
8      integer i, ix, n_loc
9      logical err
10
11     ! Read in a lattice and calculate the twiss parameters.
12
13     call bmad_parser ("simple_program/lat.bmad", lat) ! Read in a lattice.
14     if (lat%param%lattice_type == circular_lattice$) &
15         call twiss_at_start (lat) ! Calculate starting Twiss params.
16     call twiss_propagate_all (lat) ! Propagate Twiss parameters
17
18     ! Print info on the first 11 elements
19
20     print *, ' Ix  Name                Ele_type                S      Beta_a'
21     do i = 0, 10
22         ele => lat%ele(i)
23         print '(i4,2x,a16,2x,a,2f12.4)', i, ele%name, key_name(ele%key), ele%s, ele%a%beta
24     enddo
25
26     ! Find the CLEO_SOL element and print information on it.
27
28     call lat_ele_locator ('CLEO_SOL', lat, eles, n_loc, err)
29     print *
30     print *, '!-----'
31     print *, '! Information on element: CLEO_SOL'
32     print *
33     call type_ele (eles(1)%ele, .false., 0, .false., 0, .true., lat)
34
35     deallocate (eles)
36
37 end program

```

Figure 12.1: Example Bmad program

the element index  $i$ . The second column, `ele%name`, is the name of the element. The third column, `key_name`(`eleinteger` denoting what type of element (quadrupole, wiggler, etc.) it is. `key_name` is an array that translates the integer key of an element to a printable string. The fourth column, `ele%s`, is the longitudinal position at the exit end of the element. Finally, the last column, `ele%x%beta`, is the  $a$ -mode (nearly horizontal mode) beta function.

The final section of the program, lines 28 through 33, uses the routine `lat_ele_locator` (§14.6) to find the element in the lattice with the name `CLEO_SOL`. `type_ele` is used to type out the element's attributes and other information as shown on lines 14 through 41 of the output (more on this later).

This brings us to the lattice file used for the input to the program. The call to `bmad_parser` shows that this file is called `simple_program/lat.bmad`. In this file there is a call to another file

```
call, file = "layout.bmad"
```

It is in this second file that the layout of the lattice is defined. In particular, the line used to define the element order looks like

```
cesr: line = (IP_L0, d001, DET_00W, d002, Q00W, d003, ...)
use, cesr
```

If you compare this to the listing of the elements in Figure 12.2 you will find differences. For example, element #2 in the program listing is named `CLEO_SOL\3`. From the definition of the `cesr` line this should be `d001` which, if you look up its definition in `layout.bmad` is a drift. The difference between lattice file and output is due to the presence the `CLEO_SOL` element which appears in `lat.bmad`:

```
ks_solenoid      := -1.0e-9 * clight * solenoid_tesla / beam[energy]
cleo_sol: solenoid, l = 3.51, ks = ks_solenoid, superimpose
```

The solenoid is 3.51 meters long and it is superimposed upon the lattice with its center at  $s = 0$  (this is the default if the position is not specified). When `bmad_parser` constructs the lattice list of elements the superposition of `IP_L0`, which is a zero-length marker, with the solenoid does not modify `IP_L0`. The superposition of the `d001` drift with the solenoid gives a solenoid with the same length as the drift. Since this is a “new” element, `bmad_parser` makes up a name that reflects that it is basically a section of the solenoid it came from. Next, since the `CLEO_SOL` element happens to only cover part of the `Q00W` quadrupole, `bmad_parser` breaks the quadrupole into two pieces. The piece that is inside the solenoid is a `sol_quad` and the piece outside the solenoid is a regular quadrupole. See §3.3 for more details. Since the center of the `CLEO_SOL` is at  $s = 0$ , half of it extends to negative  $s$ . In this situation, `bmad_parser` will wrap this half back and superimpose it on the elements at the end of the lattice list near  $s = s_{lat}$  where  $s_{lat}$  is the length of the lattice. As explained in Chapter 14, the lattice list that is used for tracking extends from `lat%ele(0)` through `lat%ele(n)` where  $n = \text{lat}\%n\_ele\_track$ . The `CLEO_SOL` element is put in the section of `lat%ele(n)` with  $n > \text{lat}\%n\_ele\_track$  since it is not an element to be tracked through. The `Q00W` quadrupole also gets put in this part of the list. The bookkeeping information that the `cleo_sol\3` element is derived from the `cleo_sol` is put in the `cleo_sol` element as shown in lines 33 through 41 of the output. It is now possible in the program to vary, say, the strength of the `ks` attribute of the `CLEO_SOL` and have the `ks` attributes of the dependent (“`super_slave`”) elements updated with one subroutine call. For example, the following code increases the solenoid strength by 1%

```
call lat_ele_locator ('CLEO_SOL', lat, eles, n_loc, err)
eles(1)%ele(ix)%value(ks$) = eles(1)%ele%value(ks$) * 1.01
call lattice_bookkeeper (lat)
```

`Bmad` takes care of the bookkeeping. In fact `control_bookkeeper` is automatically called when transfer matrices are remade so the direct call to `control_bookkeeper` may not be necessary.

Making `bmad` the working directory, the `gmake -f M.simple_program` command will compile and link the program. The executables, `simple_program` and `simple_program_g`, are to be found in `../bin/`. Running the program with the command `../bin/simple_program` gives the output as shown in Figure 12.2.

```

1  Ix Name           Ele_type           S      Beta_a
2  0 BEGINNING      INIT_ELEMENT      0.0000   0.9381
3  1 IP_LO          MARKER            0.0000   0.9381
4  2 CLEO_SOL#3     SOLENOID          0.6223   1.3500
5  3 DET_OOW        MARKER            0.6223   1.3500
6  4 CLEO_SOL#4     SOLENOID          0.6380   1.3710
7  5 QOOW\CLEO_SOL  SOL_QUAD          1.7550   7.8619
8  6 QOOW#1         QUADRUPOLE        2.1628  16.2350
9  7 D003           DRIFT            2.4934  27.4986
10 8 DET_O1W        MARKER            2.4934  27.4986
11 9 D004           DRIFT            2.9240  46.6018
12 10 Q01W         QUADRUPOLE        3.8740  68.1771
13
14 !-----
15 ! Information on element: CLEO_SOL
16
17 Element #        871
18 Element Name: CLEO_SOL
19 Key: SOLENOID
20 S:                1.7550
21 Ref_time:        0.0000E+00
22
23 Attribute values [Only non-zero values shown]:
24   1  L              = 3.5100000E+00
25   7  KS             = -8.5023386E-02
26  32  POC            = 5.2890000E+09
27  33  E_TOT          = 5.2890000E+09
28  34  BS_FIELD       = -1.5000000E+00
29  50  DS_STEP        = 2.0000000E-01
30
31      TRACKING_METHOD = Bmad_Standard
32      MAT6_CALC_METHOD = Bmad_Standard
33      FIELD_CALC      = Bmad_Standard
34      APERTURE_AT     = Exit_End
35      OFFSET_MOVES_APERTURE = F
36      INTEGRATOR_ORDER: = 2
37      NUM_STEPS       = 18
38      SYMPLECTIFY     = F
39      FIELD_MASTER    = F
40      CSR_CALC_ON     = T
41
42 Lord_status: SUPER_LORD
43 Slave_status: FREE
44 Slaves: Number: 6
45   Name              Lat_index  Attribute      Coefficient
46   QOOW\CLEO_SOL     865  -----      3.182E-01
47   CLEO_SOL#1        866  -----      4.460E-03
48   CLEO_SOL#2        868  -----      1.773E-01
49   CLEO_SOL#3         2  -----      1.773E-01
50   CLEO_SOL#4         4  -----      4.460E-03
51   QOOW\CLEO_SOL     5  -----      3.182E-01

```

Figure 12.2: Output from the example program

# Chapter 13

## The Ele\_struct

This chapter describes the `ele_struct` which is the structure that holds all the information about an individual lattice element: quadrupoles, separators, wigglers, etc. The `ele_struct` structure is shown in Figures 13.1 and 13.2. This structure is somewhat complicated, however, in practice, a lot of the complexity is generally hidden by the *Bmad* bookkeeping routines.

As a general rule, for variables like the Twiss parameters that are not constant along the length of an element, the value stored in the corresponding component in the `ele_struct` is the value at the exit end of the element.

For printing information about an element, the `type_ele` or `type2_ele` routines can be used (§12.1). The difference between the two is that `type_ele` will print to the terminal window while `type2_ele` will return an array of strings containing the element information.

### 13.1 Initialization and Pointers

The `ele_struct` has a number of components and subcomponents that are pointers and this raises a deallocation issue. Generally, most `ele_struct` elements are part of a `lat_struct` variable (§14.1) and such elements in a `lat_struct` are handled by the `lat_struct` allocation/deallocation routines. In the case where a local `ele_struct` variable is used within a subroutine or function, the `ele_struct` variable must either be defined with the `save` attribute

```
type (ele_struct), save :: ele           ! Use the save attribute
logical, save :: init_needed = .false.
...
if (init_needed) then
  call init_ele (ele)                   ! Initialize element once
  init_needed = .false.
endif
```

or the pointers within the variable must be deallocated with a call to `deallocate_ele_pointers`:

```
type (ele_struct) ele
...
call init_ele (ele)                   ! Initialize element each time
...
call deallocate_ele_pointers (ele) ! And deallocate.
```

```

type ele_struct
  character(40) name           ! name of element \sref{c:ele.string}.
  character(40) type           ! type name \sref{c:ele.string}.
  character(40) alias          ! Another name \sref{c:ele.string}.
  character(40) component_name ! Used by overlays, multipass patch, etc.
  character(200), pointer :: descrip => null() ! Description string.
  type (twiss_struct) a, b, z   ! Twiss parameters at end of element \sref{c:twiss}.
  type (xy_disp_struct) x, y    ! Projected dispersions \sref{c:twiss}.
  type (floor_position_struct) floor ! Global floor position at end of ele.
  type (mode3_struct), pointer :: mode3 => null()
  type (coord_struct) map_ref_orb_in ! Ref orbit at entrance of element.
  type (coord_struct) map_ref_orb_out ! Ref orbit at exit of element.
  type (genfield), pointer :: gen_field => null() ! For symp_map$
  type (taylor_struct) :: taylor(6) ! Taylor terms
  type (wake_struct), pointer :: wake => null() ! Wakefields
  type (wig_term_struct), pointer :: wig_term(:) => null() ! Wiggler Coefs
  type (trans_space_charge_struct), pointer :: trans_sc => null()
  real(rp) value(n_attrib_maxx) ! attribute values.
  real(rp) old_value(n_attrib_maxx) ! Used to see if %value(:) array has changed.
  real(rp) gen0(6) ! constant part of the genfield map.
  real(rp) vec0(6) ! 0th order transport vector.
  real(rp) mat6(6,6) ! 1st order transport matrix.
  real(rp) c_mat(2,2) ! 2x2 C coupling matrix
  real(rp) gamma_c ! gamma associated with C matrix
  real(rp) s ! longitudinal position at the exit end.
  real(rp) ref_time ! Time ref particle passes exit end.
  real(rp), pointer :: r(:, :) => null() ! For general use. Not used by Bmad.
  real(rp), pointer :: a_pole(:) => null() ! multipole
  real(rp), pointer :: b_pole(:) => null() ! multipoles
  real(rp), pointer :: const(:) => null() ! Working constants.
  ele_struct definition continued on next figure...

```

Figure 13.1: The `ele_struct`. structure definition. The complete structure is shown in this and the following figure.

In the normal course of events, the pointers of an `ele_struct` variable should not be pointing to the same memory locations as the pointers of any other `ele_struct` variable. To make sure of this, the equal sign in the assignment `ele1 = ele2` is overloaded by the routine `ele_equal_ele` and this routine will allocate as necessary.

## 13.2 String Components

The `%name`, `%type`, `%alias`, and `%descrip` components of the `ele_struct` all have a direct correspondence with the `name`, `type`, `alias`, and `descrip` element attributes in an input lattice file (§4.2). On input (§15.1), from a lattice file, `name`, `type`, and `alias` attributes will be converted to uppercase before being loaded into an `ele_struct`. To save memory, since `%descrip` is not frequently used, `%descrip` is a pointer that is only allocated if `descrip` is set for a given element.

## 13.3 Element Key

The `%key` integer component gives the class of element (`quadrupole`, `rfcavity`, etc.). In general, to get the corresponding integer parameter for an element class, just add a “\$” character to the class name. For example `quadrupole$` is the integer parameter for `quadrupole` elements. The `key_name` array converts



```

... ele_struct definition continued from previous figure.
integer key                ! key value
integer sub_key            ! For wigglers: map_type$, periodic_type$
integer ix_ele             ! Index in lat%branch(n)%ele(:) array [n = 0 <==> lat%ele(:)].
integer ix_branch          ! Index in lat%branch(:) array [0 => In lat%ele(:)].
integer ix_value           ! Overlays: Index of control attribute.
integer slave_status       ! super_slave$, etc.
integer n_slave            ! Number of slaves
integer ix1_slave          ! Start index for slave elements
integer ix2_slave          ! Stop index for slave elements
integer lord_status        ! overlay_lord$, etc.
integer n_lord             ! Number of lords
integer ic1_lord           ! Start index for lord elements
integer ic2_lord           ! Stop index for lord elements
integer ix_pointer         ! For general use. Not used by Bmad.
integer ixx               ! Index for Bmad internal use
integer mat6_calc_method   ! bmad_standard$, taylor$, etc.
integer tracking_method    ! bmad_standard$, taylor$, etc.
integer field_calc         ! Used with Boris, Runge-Kutta integrators.
integer num_steps          ! number of slices for DA_maps
integer integrator_order   ! For Etiennes' PTC: 2, 4, or 6.
integer ref_orbit          ! Multipass ref orb: single_ref$, match_global_coords$, etc.
integer taylor_order       ! Order of the taylor series.
integer aperture_at        ! Aperture location: exit_end$, ...
integer aperture_type      ! Type of aperture: rectangular$, or elliptical$.
integer coupler_at         ! Lcavity coupler location: exit_end$, ...
logical symplectify        ! Symplectify mat6 matrices.
logical mode_flip          ! Have the normal modes traded places?
logical multipoles_on      ! For turning multipoles on/off
logical map_with_offsets   ! Taylor map calculated with element offsets?
logical field_master       ! Calculate strength from the field value?
logical is_on              ! For turning element on/off.
logical old_is_on          ! For saving the element on/off state.
logical logic              ! For general use. Not used by Bmad.
logical bmad_logic         ! For Bmad internal use only.
logical on_a_girder        ! Have an Girder overlay_lord?
logical csr_calc_on        ! Coherent synchrotron radiation calculation
logical offset_moves_aperture ! element offsets affects aperture?
end type

```

Figure 13.2: The `ele_struct`. The complete structure is shown in this and the preceeding figure.

from integer to the appropriate string. For example:

```

type (ele_struct) ele
if (ele%key == wiggler$) then      ! Test if element is a wiggler.
print *, 'This element: ', key_name(ele%key) ! Prints, for example, 'WIGGLER'

```

Note: The call to `init_ele` is needed for any `ele_struct` defined outside of a `lat_struct` structure.

The `%sub_key` component is only used for Wiggler, Rbend and Sband elements. For Wiggler elements, `%sub_key` is either set to

```

map_type$ or
periodic_type$

```

depending upon the type of wiggler. For bend elements, when a lattice file is parsed (§15.1), all `rbend` elements are converted into `sband` elements (§2.4). To keep track of what the original definition of the element was, the `%sub_key` component will be set to `sband$` or `rbend$` whatever is appropriate. In the case of bends, the `%sub_key` component does not affect any calculations and is only used in the routines that recreate lattice files from a `lat_struct` (§15.3).

## 13.4 The %value(:) array

Most of the real valued attributes of an element are held in the %value(:) array. For example, the value of the k1 attribute for a quadrupole element is stored in %value(k1\$) where k1\$ is an integer parameter that *Bmad* defines. In general, to get the correct index in %value(:) for a given attribute, add a "\$" as a suffix. To convert from an attribute name to its index in the %value array use the `attribute_index` routine. To go back from an index in the %value array to a name use the `attribute_name` routine. Example:

```
type (ele_struct) ele
call init_ele (ele)      ! Initialize element
ele%key = quadrupole$    ! Set element to be a quadrupole
ele%value(k1$) = 0.3      ! Set K1 value
print *, 'Index for Quad K1: ', attribute_index(ele, 'K1') ! prints: '4' (= k1$)
print *, 'Name for Quad k1$: ', attribute_name (ele, k1$)   ! prints: 'K1'
```

The list of attributes for a given element type is given in the writeup for the different element in Chapter 2.

There are also 5 slots in the %value(:) array for general use. they have indexes labeled `general1$` through `general5$`. These slots are not used by *Bmad* so a program can take advantage of them. The index names can be redefined to fit a particular need. For example, suppose a program needs to store a time stamp number. The code to do this could look like:

```
integer, parameter :: time_stamp$ = general1$
...
lat%ele(i)%value(time_stamp$) = ...
```

The %field\_master logical within an element sets whether it is the normalized strength or field strength that is the independent variable. See §4.1 for more details.

The %old\_value(:) component of the `ele_struct` is used by the `r:attribute.bookkeeper|attribute_bookkeeper` routine to check for changes for changes in the %value(:) array since the last time the `attribute_bookkeeper` routine had been called. If nothing has been changed, the `attribute_bookkeeper` routine knows not to waste time recalculating dependent values. Essentially what this means is that the %old\_value(:) array should not be modified outside of `attribute_bookkeeper`.

## 13.5 Limits

The aperture limits (§4.6) in the `ele_struct` are:

```
%value(x1_limit$)
%value(x2_limit$)
%value(y1_limit$)
%value(y2_limit$)
```

The values of these limits along with the %aperture\_at, %aperture\_type, and %offset\_moves\_aperture components are used in tracking to determine if a particle has hit the vacuum chamber wall. See Section §17.4 for more details.

## 13.6 Twiss Parameters, etc.

The components %a, %b, %z, %x, %y, %c\_mat, %gamma\_c, %mode\_flip, and mode3 hold information on the Twiss parameters, dispersion, and coupling at the exit end of the element. See Chapter 16 for more details.

## 13.7 Element Lords and Element Slaves

In *Bmad*, elements in a lattice can control other elements. The components that determine this control are:

```
%slave_status
%n_slave
%ix1_slave
%ix2_slave
%lord_status
%n_lord
%ic1_lord
%ic2_lord
%component_name
```

This is explained fully in the chapter on the `lat_struct` (§14).

## 13.8 Coordinates, Offsets, etc.

The `%floor` component gives the global “floor” coordinates (§9.2) at the exit end of the element. The components of the `%floor` structure are

```
type floor_position_struct
  real(rp) x, y, z          ! offset from origin
  real(rp) theta, phi, psi  ! angular orientation
end type
```

The routine `ele_geometry` will calculate an element’s floor coordinates given the floor coordinates at the beginning of the element. In a lattice, the `lat_geometry` routine will calculate the floor coordinates for the entire lattice using repeated calls to `ele_geometry`.

The positional offsets (§4.4) for an element from the reference orbit are stored in

```
%value(x_offset$)
%value(y_offset$)
%value(x_pitch$)
%value(y_pitch$)
%value(tilt$)
```

If the element is supported by an `girder` element (§2.13) then the `girder` offsets are added to the element offsets and the total offset with respect to the reference coordinate system is stored in:

```
%value(x_offset_tot$)
%value(y_offset_tot$)
%value(x_pitch_tot$)
%value(y_pitch_tot$)
%value(tilt_tot$)
```

If there is no `girder`, the values for `%value(x_offset_tot$)`, etc. are set to the corresponding values in `%value(x_offset$)`, etc. Thus, to vary the position of an individual element the values of `%value(x_offset$)`, etc. are changed and to read the position of an element a program should look at `%value(x_offset_tot$)`, etc.

The longitudinal position at the exit end of an element is stored in `%s` and the reference time is stored in `%ref_time`. See §9.2 for more details.

## 13.9 Transfer Maps: Linear and Non-linear (Taylor)

The routine `make_mat6` computes the linear transfer matrix (Jacobian) along with the zeroth order transfer vector. This matrix is stored in `%mat6(6,6)` and the zeroth order vector is stored in `%vec0(6)`. The reference orbit at the entrance end of the element about which the transfer matrix is computed is stored in `%map_ref_orb_in` and the reference orbit at the exit end is stored in `%map_ref_orb_out`. In the calculation of the transfer map, the vector `%vec0` is set so that

$$\text{map\_ref\_orb\_out} = \text{\%mat6} * \text{map\_ref\_orbit\_in} + \text{\%vec0}$$

The reason redundant information is stored in the element is to save computation time.

To compute the transfer maps for an entire lattice use the routine `lat_make_mat6`.

The Taylor map (§5) for an element is stored in `%taylor(1:6)`. Each `%taylor(i)` is a `taylor_struct` structure that defines a Taylor series:

```
type taylor_struct
  real (rp) ref
  type (taylor_term_struct), pointer :: term(:) => null()
end type
```

Each Taylor series has an array of `taylor_term_struct` terms defined as

```
type taylor_term_struct
  real(rp) :: coef
  integer :: exp(6)
end type
```

The coefficient for a Taylor term is stored in `%coef` and the six exponents are stored in `%exp(6)`.

To see if there is a Taylor map associated with an element the association status of `%taylor(1)%term` needs to be checked. As an example the following finds the order of a Taylor map.

```
type (ele_struct) ele
...
if (associated(ele%taylor(1)%term) then ! Taylor map exists
  taylor_order = 0
  do i = 1, 6
    do j = 1, size(ele%taylor(i)%term)
      taylor_order = max(taylor_order, sum(ele%taylor(i)%term(j)%exp)
    enddo
  enddo
else ! Taylor map does not exist
  taylor_order = -1 ! flag non-existence
endif
```

The Taylor map is made up around some reference phase space point corresponding to the coordinates at the entrance of the element. This reference point is saved in `%taylor(1:6)%ref`. Once a Taylor map is made, the reference point is not needed in subsequent calculations. However, The Taylor map itself will depend upon what reference point is chosen (§10.3).

When using the `symp_map$` tracking method (§5.1), the pointer to the partially inverted Taylor map is stored in the `%gen_field` component of the `ele_struct`. The actual storage of the map is handled by the PTC library (§11.1). The PTC partially inverted map does not have any zeroth order terms so the zeroth order terms are stored in the `%gen0(6)` vector.

## 13.10 Wake fields

See §10.13 for the equations used in wake field calculations. Wake fields are stored in the `%wake` struct:

```

type wake_struct
  character(200) :: sr_file = ' '
  character(200) :: lr_file = ' '
  type (sr_table_wake_struct), pointer :: sr_table(0:) => null()
  type (sr_mode_wake_struct), pointer :: sr_mode_long(:) => null()
  type (sr_mode_wake_struct), pointer :: sr_mode_trans(:) => null()
  type (lr_wake_struct), pointer :: lr(:) => null()
  real(rp) :: z_sr_mode_max = 0
end type

```

Since %wake is a pointer its association status must be tested before any of its sub-components are accessed.

```

type (ele_struct) ele
...
if (associated(ele%wake)) then
...

```

*Bmad* observes the following rule: If %wake is associated then it is assumed that all the sub-components (%wake%sr\_table, etc.) are associated. This simplifies programming in that you do not have to test directly the association status of the sub-components.

The short-range wake can be parameterized in either of two ways. One parameterization uses a table of wake verses z position. If this parameterization utilizes the %wake%sr\_table(0:) array The structure of each element in this array is:

```

type sr_table_wake_struct ! Tabular short-Range Wake struct
  real(rp) z               ! Distance behind the leading particle
  real(rp) long            ! Longitudinal wake in V/C/m
  real(rp) trans           ! Transverse wake in V/C/m^2
end type

```

All %wake%sr\_table(0)%z must be negative except %wake%sr\_table(0)%z = 0. Wake field kicks are applied using Eqs. (10.130) and (10.133).

The alternative short-range wake parameterization uses pseudo-modes (Eq. (10.134)). This parameterization utilizes the %wake%sr\_mode\_long, and %wake%sr\_mode\_trans arrays for the longitudinal and transverse modes respectively. The structure used for the elements of these arrays are:

```

type sr_mode_wake_struct ! Pseudo-mode short-range wake struct
  real(rp) amp             ! Amplitude
  real(rp) damp            ! Damping factor.
  real(rp) freq            ! Frequency in Hz
  real(rp) phi             ! Phase in radians/2pi
  real(rp) norm_sin        ! non-skew sin-like component of the wake
  real(rp) norm_cos        ! non-skew cos-like component of the wake
  real(rp) skew_sin        ! skew sin-like component of the wake
  real(rp) skew_cos        ! skew cos-like component of the wake
end type

```

The wake field kick is calculated from Eq. (10.134). %amp, %damp, %freq, and %phi are the input parameters from the lattice file. the last four components (%norm\_sin, etc.) store the accumulated wake: Before the bunch passes through these are set to zero and as each particle passes through the cavity the contribution to the wake due to the particle is calculated and added the components.

%wake%z\_sr\_mode\_max is the maximum z value beyond which the pseudo mode representation is not valid. This is set in the input lattice file.

The %wake%lr array stores the long-range wake modes. The structure definition is:

```

type lr_wake_struct      ! Long-Range Wake struct
  real(rp) freq          ! Actual Frequency in Hz
  real(rp) freq_in       ! Input frequency in Hz
  real(rp) R_over_Q      ! Strength in V/C/m^2
  real(rp) Q             ! Quality factor
  real(rp) angle         ! polarization angle (radians/2pi).
  integer m              ! Order (1 = dipole, 2 = quad, etc.)
  real(rp) norm_sin      ! non-skew sin-like component of the wake
  real(rp) norm_cos      ! non-skew cos-like component of the wake
  real(rp) skew_sin      ! skew sin-like component of the wake
  real(rp) skew_cos      ! skew cos-like component of the wake
  logical polarized      ! Polarized mode?
end type

```

This is similar to the `sr_mode_wake_struct`. `%freq_in` is the actual frequency in the input file. `bmad_parser` will set `%freq` to `%freq_in` except when the `lr_freq_spread` attribute is non-zero in which case `bmad_parser` will vary `%freq` as explained in §2.16. `%polarized` is a logical that indicates whether the the mode has a polarization angle. If so, then `%angle` is the polarization angle.

## 13.11 Wiggler Types

The `%sub_key` component of the `ele_struct` is used to distinguish between `map` type and `periodic` type wigglers (§13.3):

```

if (ele%key == wiggler$ .and. ele%sub_key == map_type$) ...
if (ele%key == wiggler$ .and. ele%sub_key == periodic_type$) ...

```

For a `map` type wiggler, the wiggler field terms (§10.6) are stored in the `%wig_term(:)` array of the `element_struct`. This is an array of `wig_term_struct` structure. A `wig_term_struct` looks like:

```

type wig_term_struct
  real(rp) coef
  real(rp) kx, ky, kz
  real(rp) phi_z
  integer type      ! hyper_y$, hyper_xy$, or hyper_x$
end type

```

A `periodic` wiggler will have a single `%wig_term(:)` term that can be used for tracking purposes, etc.

The setting for this `wig_term` element is

```

ele%wig_term(1)%ky      = pi / ele%value(l_pole$)
ele%wig_term(1)%coef    = ele%value(b_max$)
ele%wig_term(1)%kx      = 0
ele%wig_term(1)%kz      = ele%wig_term(1)%ky
ele%wig_term(1)%phi_z   = (ele%value(l_pole$) - ele%value(l$)) / 2
ele%wig_term(1)%type    = hyper_y$

```

## 13.12 Multipoles

The multipole components of an element (See §10.2) are stored in the pointers `%a(:)` and `%b(:)`. If `%a` and `%b` are allocated they always have a range `%a(0:n_pole_maxx)` and `%b(0:n_pole_maxx)`. Currently `n_pole_maxx = 20`. For a `Multipole` element, the `%a(n)` array stores the integrated multipole strength  $KnL$ , and the `%b(n)` array stores the tilt  $T_n$ .

A list of *Bmad* routines for manipulating multipoles can be found in §23.21.

## 13.13 Tracking Methods

A number of `ele_struct` components control tracking and transfer map calculations. These are

```
%mat6_calc_method
%tracking_method
%num_steps
%integrator_order
%taylor_order
%symplectify
%multipoles_on
%map_with_offsets
%is_on
%csr_calc_on
%offset_moves_aperture
```

See Chapter §17 for more details.

## 13.14 General Use Components

There are three components of an `ele_struct` that are guaranteed to never be used by any *Bmad* routine and so are available for use by someone writing a program. These components are

```
real(rp), pointer :: r(:) => null()
integer ix_pointer
logical logic
```

## 13.15 Bmad Reserved Variables

A number of `ele_struct` components are reserved for use by *Bmad* routines only. These are

```
%ixx
%old_is_on
%bmad_logic
%const(:)
```





## Chapter 14

# The lat\_struct

The `lat_struct` is the structure that holds of all the information about a lattice including any `branch` and `photon_branch` lines (§2.5). The components of a `lat_struct` are listed in Figure 14.1.

```
type lat_struct
  character(16) name                ! Name in USE statement
  character(40) lattice              ! Lattice name
  character(80) input_file_name      ! Lattice input file name
  character(80) title                ! From TITLE statement
  type (mode_info_struct) a, b, z    ! tunes, etc.
  type (lat_param_struct) param      ! Lattice parameters
  type (ele_struct), pointer :: ele(:) ! Array of lattice elements
  type (ele_struct) ele_init         ! For use by any program
  type (branch_struct), allocatable :: branch(:) ! Branch arrays
  type (control_struct), allocatable :: control(:) ! control list
  type (coord_struct) beam_start     ! Starting coordinates.
  integer version                    ! Digested file version number
  integer n_ele_track                ! elements in tracking lattice
  integer n_ele_max                  ! Index of last element used
  integer n_control_array            ! last index used in control(:)
  integer n_ic_max                   ! last index used in ic(:) array
  integer input_taylor_order         ! As set in the input file
  integer, allocatable :: ic(:)      ! index to %control(:)
end type
```

Figure 14.1: Definition of the `lat_struct`.

### 14.1 Pointers

Since the `lat_struct` has pointers within it, there is an extra burden on the programmer to make sure that allocation and deallocation is done properly. To this end, the equal sign has been overloaded by the routine `lat_equal_lat` so that when one writes

```
type (lat_struct) lattice1, lattice2
```

```
! ... some calculations ...
lattice1 = lattice2
```

the pointers in the `lat_struct` structures will be handled properly. The result will be that `lattice1` will hold the same information as `lattice2` but the pointers in `lattice1` will point to different locations in physical memory so that changes to one lattice will not affect the other.

Initial allocation of the pointers in a `lat_struct` variable is generally handled by the `bmad_parser` and `lat_equal_lat` routines. Once allocated, local `lat_struct` variables must have the `save` attribute or the pointers within must be appropriately deallocated before leaving the routine.

```
type (lat_struct), save :: lattice      ! Either do this at the start or ...
...
call deallocate_lat_pointers (lattice) ! ... Do this at the end.
```

Using the `save` attribute will generally be faster but will use more memory. Typically using the `save` attribute will be the best choice.

## 14.2 Branches in the lat\_struct

The lattice is divided up into the “main branch” (also called the “main lattice”) and, if there are `branch` or `photon_branch` elements, a number of other branches. To simplify the bookkeeping, the main lattice is itself considered a branch.

The branch information is contained in the `%branch(0:)` array. The `%branch` array is always indexed from 0 with the 0 branch being the main lattice. The upper bound of this array will be equal to the number of `branch` plus `photon_branch` elements present. The definition of the `branch_struct` structure is

```
type branch_struct
  character(40) name
  integer key                ! photon_branch$, branch$, etc.
  integer ix_branch          ! Index in lat%branch(:) array.
  integer ix_from_branch     ! 0 => Main lattice
  integer ix_from_ele        ! Index of branching element
  integer, pointer :: n_ele_track ! Number of tracking elements
  integer, pointer :: n_ele_max
  type (ele_struct), pointer :: ele(:) ! Element array
  type (lat_param_struct), pointer :: param
end type
```

The `branch(i)%key` component is set to either `photon_branch$` or `branch$`. The `key_name` function will convert the integer key value to a string for printing. The value of the `%branch(i)%ix_branch` component is the branch index and will thus have the value `i`. This can be useful when passing a branch to a subroutine. The `%branch(i)%ix_from_branch` component gives the branch index of the branch that the  $i^{th}$  branch branched off from. `%branch(i)%ix_from_ele` gives the index in the `%branch(j)%ele(:)` array of the `branch` or `photon_branch` element that marks the beginning of the  $i^{th}$  branch. Example:

```
type (lat_struct) lat
...
ib = lat%branch(3)%ix_from_branch
ie = lat%branch(3)%ix_from_ele
!! lat%branch(ib)%ele(ie) is the branch or photon_branch element for lat%branch(3)
```

The `%branch%ele(:)` array holds the array of elements in the branch. Historically, the `lat_struct` was developed at the start of the *Bmad* project and branches were implemented well after that. To

<i>section</i>	<i>Element index</i>	
	<i>min</i>	<i>max</i>
tracking	0	%n_ele_track
control	%n_ele_track+1	%n_ele_max

Table 14.1: Bounds of the tracking and control parts of the main lattice %branch(0)%ele(:) array.

```

type lat_param_struct
  real(rp) n_part           ! Particles/bunch (for beambeam elements).
  real(rp) total_length     ! total_length of lattice
  real(rp) growth_rate      ! growth rate/turn if not stable
  real(rp) t1_with_RF(6,6)  ! Full 1-turn 6x6 matrix
  real(rp) t1_no_RF(6,6)    ! Transverse 1-turn 4x4 matrix (RF off).
  integer particle           ! +1 = positrons, -1 = electrons
  integer ix_lost            ! If lost at what element?
  integer end_lost_at        ! entrance_end$ or exit_end$
  integer plane_lost_at      ! x_plane$, y_plane$, or z_plane$
  integer lattice_type       ! linear_lattice$, circular_lattice$, etc...
  integer ixx                ! Integer for general use
  logical stable             ! is closed lattice stable?
  logical aperture_limit_on  ! use apertures in tracking?
  logical lost               ! for use in tracking
end type

```

Figure 14.2: Definition of the param\_struct.

maintain compatibility with older code, the lat%ele(:) array, which is used by older code, points to the same memory block as the lat%branch(0)%ele array. Similarly, %branch(0)%n\_ele\_track, %branch(0)%n\_ele\_max, and %branch(0)%param point to lat%n\_ele\_track, lat%n\_ele\_max, lat%param respectively.

The %branch%ele(:) array is always allocated with zero as the lower bound. %ele(0) is a marker element with its %name component set to “BEGINNING”. %ele(0)%mat6 is always the unit matrix. For the main branch, the %branch(0)%ele(0:) array is divided up into two parts: The “tracking” part and a “control” part (also called the “lord” part). The tracking part of this array holds the elements that are tracked through. The control part holds elements that control attributes of other elements (§14.4). The bounds of these two parts is given in Table 14.2. Only the main branch has a lord section so %branch%n\_ele\_track and %branch%n\_ele\_max are the same for all other branches. Since the main branch can also be accessed via the lat%ele(:) array, code that deals with the lord section of the lattice may use lat%ele(:) in place of lat%branch(0)%ele(:).

### 14.3 Param\_struct Component

The %param component within each lat%branch(:) is a lat\_param\_struct structure whose definition is shown in Figure 14.2

%param%total\_length is the length of the lattice that a beam tracks through defined by

```
%param%total_length = %ele(n_ele_track)%s - %ele(0)%s
```

Normally `%ele(0)%s = 0` so `%param%total_length = %ele(n_ele_track)%s` but this is not always the case.

`%param%n_part` is the number of particles in a bunch and is used by `beambeam` element to determine the strength of the beambeam interaction. `%param%n_part` is also used by `lcavity` elements for wake field calculations.

When tracking particles through a lattice the variable `%param%aperture_limit_on` determines if apertures are checked. `%param%lost` is used to signal if a particle is lost and `%param%ix_lost` gives the index of the element at which a particle is lost. Additionally, `%param%end_lost_at` is used to indicate at which end the particle was lost at. Possible values for `%end_lost_at` are

```
%entrance_end
%exit_end
```

The `%plane_lost_at` component records whether the lost particle was lost horizontally, vertically or longitudinally. Possible values for `%plane_lost_at` are

```
%x_plane
%y_plane
%z_plane
```

See Chapter 17 for more details.

`%param%t1_with_RF` and `%param%t1_no_RF` are the 1-turn transfer matrices from the start of the lattice to the end. `%param%t1_with_RF` is the full transfer matrix with RF on. `%param%t1_no_RF` is the transverse transfer matrix with RF off. `%param%t1_no_RF` is used to compute the Twiss parameters. When computing the Twiss parameters `%param%stable` is set according to whether the matrix is stable or not. If the matrix is not stable the Twiss parameters cannot be computed. If unstable, `%param%growth_rate` will be set to the growth rate per turn of the unstable mode.

The particle type for a branch is stored in the integer variable `%param%particle`. The value of this variable will correspond to one of the constants:

```
positron$
electron$
proton$
antiproton$
photon$
```

To print the name of the particle use the function `particle_name`. A particles mass and charge can be obtained from the functions `mass_of` and `charge_of` respectively. `charge_of` returns the particle's charge in units of the fundamental electron charge. Example:

```
type (lat_struct) lat
...
print *, 'Beam Particles are: ', particle_name(lat%param%particle)
if (lat%param%particle == proton$) print *, 'I do not like protons!'
print *, 'Particle mass (eV): ', mass_of(lat%param%particle)
print *, 'Particle charge (Coul):', e_charge * charge_of(lat%param%particle)
```

## 14.4 Elements Controlling Other Elements

In the `lat_struct` structure, certain elements in the `%ele(:)` array (equivalent to the `%branch(0)%ele(:)` array), called **lord** elements, can control the attributes (component values) of other `%branch(:)%ele(:)` elements. Elements so controlled are called **slave** elements. The situation is complicated by the fact that a given element may simultaneously be a **lord** and a **slave**. For example, an `overlay` element (§3)

is a lord since it controls attributes of other elements but an `overlay` can itself be controlled by other `overlay` and `group` elements. In all cases, circular lord/slave chains are not permitted.

The lord and slave elements can be divided up into classes. What type of lord an element is, is set by the value of the element's `ele%lord_status` component. Similarly, what type of slave an element is is set by the value of the element's `ele%slave_status` component. Nomenclature note: An element may be referred to by its `%lord_status` or `%slave_status` value. For example, an element with `ele%lord_status` set to `super_lord$` can be referred to as a “`super_lord`” element.

The value of the `ele%lord_status` component can be one of:

#### **`super_lord$`**

A `super_lord` element is created when elements are superimposed on top of other elements (§3.3).

#### **`girder_lord$`**

A `girder_lord` element is a `girder` element (§2.13). That is, the element will have `ele%key = girder$`. In this case, the slave elements are marked by setting the `slave_ele%on_a_girder` logical to `True`.

#### **`multipass_lord$`**

`multipass_lord` elements are created when multipass lines are present (§3.4).

#### **`overlay_lord$`**

An `overlay_lord` is an `overlay` element (§3.1). That is, such an element will have `ele%key = overlay$`.

#### **`group_lord$`**

A `group_lord` is a `group` element (§3.2). That is, such an element will have `ele%key = group$`.

#### **`not_a_lord$`**

This element does not control anything.

Any element whose `%lord_status` is something other than `not_a_lord$` is called a lord element. In the tracking part of the lattice (§14.2), `%lord_status` will always be `not_a_lord$`. In the lord section of the lattice, under normal circumstances, there will never be any `not_a_lord` elements. However, it is permissible, and sometimes convenient, for programs to set the `%lord_status` of a lord element to `not_a_lord$`.

`indexfree$` The possible values for the `ele%slave_status` component are:

#### **`super_slave$`**

A `super_slave` element is an element in the tracking part of the lattice that has one or more `super_lord` lords (§3.3).

#### **`multipass_slave$`**

A `multipass_slave` element is the slave of a `multipass_lord` (§3.4).

#### **`patch_in_slave$`**

A `patch_in_slave` is a `patch` element that is also a `multipass_lord` with `%ref_orbit` set to `patch_out$` (§3.4).

#### **`overlay_slave$`**

An `overlay_slave` is an element that has one or more `overlay_lord` or `girder_lord` lords. Note that an `overlay_slave` can have zero associated `overlay_lords` as long as it has a `girder_lord`. Thus the term `overlay_slave` is a bit misleading. An `overlay_slave` can also have associated `group_lord` elements.

ele%slave_status	ele%lord_status1					
	not_a_lord\$	group_lord\$	girder_lord\$	overlay_lord\$	multipass_lord\$	super_lord\$
free\$	X	X	X	X	X	X
group_slave\$	X	X	X	X	X	X
overlay_slave\$	X	X	X	X	X	X
patch_in_slave\$					X	
multipass_slave\$	X					X
super_slave\$	X					

(a) Possible ele%lord\_status and ele%slave\_status combinations within an individual element.

slave%slave_status	lord%lord_status					
	not_a_lord\$	group_lord\$	girder_lord\$	overlay_lord\$	multipass_lord\$	super_lord\$
free\$						
group_slave\$		X				
overlay_slave\$		X	X	X		
patch_in_slave\$					Y	
multipass_slave\$		1		X	X	
super_slave\$						X

(b) Possible %lord\_status and %slave\_status combinations for any lord/slave pair.

Table 14.2: Possible %lord\_status/%slave\_status combinations. “X” marks a possible combination. “1” indicates that the slave will have exactly one lord of the type given in the column. “Y” indicates the combination is not bidirectional.

### group\_slave\$

A **group\_slave** is an element that has one or more **group\_lord** elements controlling the slave’s attributes. A **group\_slave** will not have any other types of lords.

### free\$

Free elements do not have any lord elements controlling them.

Any element whose %slave\_status is something other than **free\$** is called a **slave** element. **super\_slave** elements always appear in the tracking part of the lattice. The other types can be in either the tracking or control parts of the lattice.

Only some combinations of %lord\_status values and %slave\_status values are permissible for a given element. Table 14.2(a) lists the valid combinations. Thus, for example, it is *not* possible for an element to be simultaneously a **super\_lord** and a **super\_slave**.

For lord/slave pairs, Table 14.2(b) lists the valid combinations of %lord\_status values in the lord element and %slave\_status values in the slave element. Thus, for example, a **super\_slave** may only be controlled by a **super\_lord**. In the example in Section §3.4, element A would be a **multipass\_lord** and A\1 and A\2 would be **multipass\_slaves**. When superposition is combined with multipass, the elements in the tracking part of the lattice will be **super\_slaves**. These elements will be controlled by **super\_lords** which will also be **multipass\_slaves** and these **super\_lord/multipass\_slave** elements will be controlled by **multipass\_lords**. This is illustrated in Figure 14.3.

The number of slave elements that a lord controls is given by the value of the lord’s %n\_slave component. Additionally, the number of lord elements that the slave has is given by the value of the slave’s. %n\_lord component. To find the slaves and lords of a given element, use the routines **pointer\_to\_slave** and **pointer\_to\_lord**. Example:

```

type (lat_struct), target :: lat
type (ele_struct), pointer :: this_ele, lord_ele, slave_ele
...
this_ele => lat%ele(321)      ! this_ele points to a given element in the lattice

do i = 1, this_ele%n_lord    ! Loop over all lords of this_ele
```

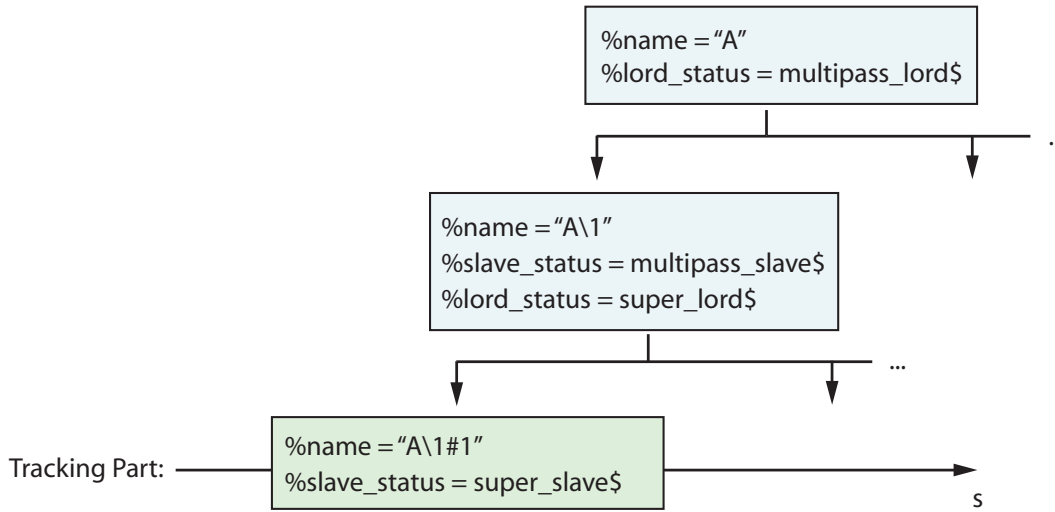


Figure 14.3: Example of multipass combined with superposition. A `multipass_lord` element named A controls a set of `multipass_slaves` (only one shown). The `multipass_slave` elements are also `super_lord` elements and they will control `super_slave` elements in the tracking part of the lattice.

```

! lord_ele points to the ith lord element of this_ele
lord_ele => pointer_to_lord (lat, this_ele, i)
...
enddo

do i = 1, this_ele%n_slave ! Loop over all slaves of this_ele
! slave_ele points to the ith slave element of this_ele
slave_ele => pointer_to_slave (lat, this_ele, i)
...
enddo

```

The lord/slave bookkeeping is bidirectional. That is, for any given element, call it `this_ele`, consider the  $i^{\text{th}}$  lord:

```
lord_ele_i => pointer_to_lord (lat, this_ele, i)
```

then there will always be some index  $j$  such that the element pointed to by

```
pointer_to_slave(lat, lord_ele_i, j)
```

is the original element `this_ele`. The same is true for the slaves of any given element. That is, for the  $i^{\text{th}}$  slave

```
slave_ele_i => pointer_to_slave (lat, this_ele, i)
```

there will always be some index  $j$  such that the element pointed to by

```
pointer_to_lord(lat, slave_ele_i, j)
```

will be the original element `this_ele`. The one exception here is that the lord of a `patch_in_slave` does not have a pointer back to the `patch_in_slave`.

The following ordering of slaves and lords is observed:

#### Slaves of a `super_lord`:

The associated `super_slave` elements of a given `super_lord` element are ordered from the entrance end of the `super_lord` to the exit end. That is, in the code snippet above, `pointer_to_slave (lat, this_ele, 1)` will point to the slave at the start of the `super_lord` and `pointer_to_slave (lat, this_ele, this_ele%n_lord)` will point to the slave at the exit end of the `super_lord`.

**Slaves of a multipass\_lord:**

The associated `multipass_slave` elements of a `multipass_lord` element are ordered by pass number. That is, in the code snippet above, `pointer_to_slave (lat, this_ele, i)` will point to the slave of the  $i^{th}$  pass.

**Lord of a multipass\_slave:**

A `multipass_slave` will have exactly one associated `multipass_lord` and this lord will be the first one. That is, `pointer_to_slave (lat, this_ele, 1)`.

**Patch\_in\_slave:**

When a lattice input file has a `patch` element that is used in a `multipass` line and the `patch` element has its `ref_orbit` attribute set to `patch_out` (§3.4.2), the `multipass_lord` element corresponding to this element is also a `patch_in_slave`. In this case, the element will have exactly one lord and that lord will be the `multipass_lord` element corresponding to the element specified by the `ref_patch` attribute of the `patch_in_slave`. Using the example given in §3.4.2, the `multipass_lord` corresponding to the `p1` `patch` element will have a lord that is the corresponding element to the `p2` `patch` element. The reason for this exception is so that the slaves pointed to by `pointer_to_slave` of a `multipass_lord` will always be `multipass_slaves`.

The element control information is stored in the `lat%control(:)` array. Each element of this array is a `control_struct` structure

```
type control_struct
  real(rp) coef                ! control coefficient
  integer ix_lord              ! index to lord element
  integer ix_slave            ! index to slave element
  integer ix_branch           ! Branch index of the slave element.
  integer ix_attrib           ! index of attribute controlled
end type
```

Each element in the `lat%control(:)` array holds the information on one lord/slave pair. The `%ix_lord` component gives the index of the lord element which is always in the main branch — branch 0. The `%ix_slave` and `%ix_branch` components give the element index and branch index of the slave element. The `%coef` and `%ix_attrib` components are used to store the coefficient and attribute index for `overlay` and `group` control. The appropriate `control_struct` for a given lord/slave pair can be obtained from the optional fourth argument of the `pointer_to_lord` and `pointer_to_slave` functions. Example: The following prints a list of the slaves, along with the attributes controlled and coefficients, on all group elements in a lattice.

```
type (lat_struct), target :: lat
type (ele_struct), pointer :: lord, slave
type (control_struct), pointer :: con
...
do i = lat%n_ele_track+1, lat%n_ele_max ! loop over all lords
  lord => lat%ele(i)
  if (lord%lord_status = group_lord$) then
    print *, 'Slaves for group lord: ', lord%name
    do j = 1, lord%n_slave
      slave => pointer_to_slave (lat, lord, j, ix_con)
      con => lat%control(ix_con)
      attrib_name = attribute_name (slave, con%ix_attrib)
      print *, i, slave%name, attrib_name, con%coef
    enddo
  endif
enddo
```



The elements in the `lat%control(:)` array associated with the slaves of a given lord are in the same order as the slaves and the index of the associated `lat%control(:)` element of the first slave is given by the `%ix1_slave` component of the lord and the last slave is given by the `%ix2_slave` component of the lord. Example:

```
type (lat_struct), target :: lat
type (ele_struct), pointer :: lord, slave
...
lord => lat%ele(i)                ! Point to some element
if (lord%n_slave > 0) then
  slave => pointer_to_slave (lat, lord, 1, ix_con)
  print *, lord%ix1_slave == ix_con  ! Will print "T" for True
  slave => pointer_to_slave (lat, lord, lord%n_slave, ix_con)
  print *, lord%ix2_slave == ix_con  ! Will print "T" for True
endif
```

This fact can be used to determine where a slave is in the list of slaves for a lord. The following example prints the pass number of a `multipass_slave` taking advantage of the fact that the pass number

```
type (lat_struct), target :: lat
type (ele_struct), pointer :: lord, slave
...
slave => lat%ele(i)                ! Point to some element
if (slave%slave_status == multipass_slave$) then
  ! The multipass_lord of this element is the first lord.
  lord => pointer_to_lord(lat, slave, 1, ix_con)
  print *, 'Multipass_slave: ', slave$name
  print *, 'Is in pass number:', ix_con - lord%ix1_slave + 1
endif
```

The `%ic1_lord` and `%ic2_lord` components of a given slave element, along with the `lat%ic(:)` array, can be used to find the lords of the slave. In virtually all practical cases, it is simpler to use the `pointer_to_lord` function to find the lords and so it is recommended that the use of `%ic1_lord` and `%ic2_lord` be avoided. For the interested reader, the entire `pointer_to_lord` function (minus some error checking) is:

```
function pointer_to_lord (lat, slave, ix_lord, ix_control) result (lord_ptr)
  implicit none
  type (lat_struct), target :: lat
  type (ele_struct) slave
  type (ele_struct), pointer :: lord_ptr
  integer, optional :: ix_control
  integer ix_lord, icon
  !
  icon = lat%ic(slave%ic1_lord + ix_lord - 1)
  lord_ptr => lat%ele(lat%control(icon)%ix_lord)
  if (present(ix_control)) ix_control = icon
end function
```

## 14.5 Lattice Bookkeeping

When an attribute of an element is changed, the entire lattice can be affected. For example, changing the length of an element will affect the global position of all subsequent elements in the lattice. To do the necessary bookkeeping, the `lattice_bookkeeper` routine may be used.

```

type (lat_struct) lat
...
lat%ele(i)%value(gradient$) = 1.05e6 ! Change the gradient of an RFCavity
call lattice_bookkeeper (lat)         ! And update everything.

```

The advantage of using `lattice_bookkeeper` is that a complete job is done in one call. The disadvantage is that, since `lattice_bookkeeper` does not know exactly what has changed, it generally does more calculations than is necessary. This can be a problem if `lattice_bookkeeper` is being called many times. Additionally, routines like `lat_make_mat6` will also call bookkeeping routines and this will add to the execution time of a program.

If program execution time is a problem, it may be necessary to fine tune the bookkeeping process. The first step is to turn off automatic bookkeeping in routines like `lat_make_mat6` by setting the global variable `bmad_com%auto_bookkeeper` (§8.1) to False

```
bmad_com%auto_bookkeeper = .false.
```

The bookkeeping routines that can be used are in place of `lattice_bookkeeper` are:

```

attribute_bookkeeper ! Bookkeeping of attributes in a given element
control_bookkeeper   ! Lord/slave control bookkeeping
s_calc               ! Longitudinal element position calc.
lat_geometry         ! Element global (floor) positions.
compute_reference_energy

```

See the individual routines for more details.

## 14.6 Finding Elements and Changing Attribute Values

The routine `lat_ele_locator` can be used to search for an element in a lattice by name or key type or a combination of both. Example:

```

type (lat_struct) lat
type (ele_pointer_struct), allocatable :: eles(:)
integer n_loc; logical err
...
call lat_ele_locator ("quad:skew*", lat, eles, n_loc, err)
print *, 'Quadrupole elements whose name begins with the string "SKEW":'
print *, 'Name          Branch_index      Element_index'
do i = 1, n_loc ! Loop over all elements found to match the search string.
  print *, eles(i)%ele$name, eles(i)%ele%ix_branch, eles(i)%ele%ix_ele
enddo

```

This example finds all elements where `ele%key` is `quadrupole$` and `ele$name` starts with “skew”. See the documentation on `lat_ele_locator` for more details on the syntax of the search string.

The `ele_pointer_struct` array returned by `lat_ele_locator` is an array of pointers to `ele_struct` elements

```

type ele_pointer_struct
  type (ele_struct), pointer :: ele => null()
end type

```

The `n_loc` argument is the number of elements found and the `err` argument is set True on a decode error of the search string.

Once an element (or elements) is identified in the lattice, it’s attributes can be altered. However, care must be taken that an element’s attribute can be modified (§4.1). The function `attribute_free` will check if an attribute is free to vary.

```

type (lat_struct) lat
integer ix_ele
...
call lat_ele_locator ('Q10W', lat, eles, n_loc, err) ! look for a element 'Q10W'
free = attribute_free (eles(i)%ele, 'K1', lat, .false.)
if (.not. free) print *, 'Cannot vary k1 attribute of element Q10W'

```

With user input the routine `pointer_to_attribute` is a convenient way to obtain from an input string a pointer that points to the appropriate attribute. For example:

```

type (lat_struct) lat
character(40) attrib_name, ele_name
real(rp), pointer :: attrib_ptr
real(rp) set_value
logical err_flag
integer ix_attrib, ie
...
write (*, '(a)', advance = 'no') ' Name of element to vary: '
accept '(a)', ele_name
write (*, '(a)', advance = 'no') ' Name of attribute to vary: '
accept '(a)', attrib_name
write (*, '(a)', advance = 'no') ' Value to set attribute at: '
accept *, set_value
do ie = 1, lat%n_ele_max
  if (lat%ele(ie)%name == ele_name) then
    call pointer_to_attribute (lat%ele(ie), attrib_name, &
                             .false., attrib_ptr, ix_attrib, err_flag)
    if (err_flag) exit ! Do nothing on an error
    attrib_ptr = set_value ! Set the attribute
  endif
enddo

```

changing an element attribute generally involves changing values in the `%ele(i)%value(:)` array. This is done using the `set_ele_attribute` routine. For example:

```

type (lat_struct) lat
logical err_flag, make_xfer_mat
...
call element_locator ('Q01W', lat, ix_ele)
call set_ele_attribute (lat, ix_ele, 'K1', 0.1_rp, err_flag, make_xfer_mat)

```

This example sets the K1 attribute of an element named Q01W. `set_ele_attribute` checks whether this element is actually free to be varied and sets the `err_flag` logical accordingly. An element's attribute may not be freely varied if, for example, the attribute is controlled via an Overlay.

## 14.7 Beam\_start Component

The `lat%beam_start` component is a `coord_struct` structure for holding the information obtained from `beam_start` statements (§7.2) in a *Bmad* lattice file.

This component is not used in any standard *Bmad* calculation. It is up to an individual program to use as desired.

## 14.8 Miscellaneous

The `%ele_init` component within the `lat_struct` is not used by *Bmad* and is available for general program use.

## Chapter 15

# Reading and Writing Lattices

### 15.1 Reading in Lattices

*Bmad* has routines for reading XSIF (§1.1) and *Bmad* formatted lattice files. The subroutine to read in an XSIF lattice file is `xsif_parser`. There are two subroutines in *Bmad* to read in a *Bmad* standard lattice file: `bmad_parser` and `bmad_parser2`. `bmad_parser` is used to initialize a `lat_struct` (§14) structure from scratch using the information from a lattice file. Unless told otherwise, after reading in the lattice, `bmad_parser` will compute the 6x6 transfer matrices for each element and this information will be stored in the digested file (§1.3) that is created. Notice that `bmad_parser` does *not* compute any Twiss parameters.

`bmad_parser2` is typically used after `bmad_parser` if there is additional information that needs to be added to the lattice. For example, consider the case where the aperture limits for the elements is stored in a file that is separate from the main lattice definition file and it is undesirable to put a `call` statement in one file to reference the other. To read in the lattice information along with the aperture limits, there are two possibilities: One possibility is to create a third file that calls the first two:

```
! This is a file to be called by bmad_parser
call, file = 'lattice_file'
call, file = 'aperture_file'
```

and then just use `bmad_parser` to parse this third file. The alternative is to use `bmad_parser2` so that the program code looks like:

```
! program code to read in everything
call bmad_parser ('lattice_file', lat)      ! read in a lattice.
call bmad_parser2 ('aperture_file', lat)    ! read in the aperture limits.
```

### 15.2 Digested Files

Since parsing can be slow, once the `bmad_parser` routine has transferred the information from a lattice file into the `lat_struct` it will make what is called a digested file. A digested file is an image of the `lat_struct` in binary form. When `bmad_parser` is called, it first looks in the same directory as the lattice file for a digested file whose name is of the form:

```
'digested_' // LAT_FILE    ! for single precision BMAD
'digested8_' // LAT_FILE    ! for double precision BMAD
```

where `LAT_FILE` is the lattice file name. If `bmad_parser` finds the digested file, it checks that the file is not out-of-date (that is, whether the lattice file(s) have been modified after the digested file is made). `bmad_parser` can do this since the digested file contains the names and the dates of all the lattice files that were involved. Also stored in the digested file is the “*Bmad* version number”. The *Bmad* version number is a global parameter that is increased (not too frequently) each time a code change involves modifying the structure of the `lat_struct` or `ele_struct`. If the *Bmad* version number in the digested file does not agree with the number current when `bmad_parser` was compiled, or if the digested file is out-of-date, a warning will be printed, and `bmad_parser` will reparse the lattice and create a new digested file.

Since computing Taylor Maps can be very time intensive, `bmad_parser` tries to reuse Taylor Maps it finds in the digested file even if the digested file is out-of-date. To make sure that everything is OK, `bmad_parser` will check that the attribute values of an element needing a Taylor map are the same as the attribute values of a corresponding element in the digested file before it reuses the map. Element names are not a factor in this decision.

This leads to the following trick: If you want to read in a lattice where there is no corresponding digested file, and if there is another digested file that has elements with the correct Taylor Maps, then, to save on the map computation time, simply make a copy of the digested file with the digested file name corresponding to the first lattice.

The digested file is in binary format and is not human readable but it can provide a convenient mechanism for transporting lattices between programs. For example, say you have read in a lattice, changed some parameters in the `lat_struct`, and now you want to do some analysis on this modified `lat_struct` using a different program. One possibility is to have the first program create a digested file

```
call write_digested_bmad_file ('digested_file_of_mine', lat)
```

and then read the digested file in with the second program

```
call read_digested_bmad_file ('digested_file_of_mine', lat)
```

An alternative to writing a digested file is to write a lattice file using `write_bmad_lattice_file`

### 15.3 Writing Lattice files

To create a *Bmad* lattice file from a `lat_struct` instance, use the routine `write_bmad_lattice_file`. *MAD-8*, *MAD-X*, or *XSIF* compatible lattice files can be created from a `lat_struct` variable using the routine `bmad_to_mad_or_xsif`:

```
type (lat_struct) lat          ! lattice
...
call bmad_parser (bmad_lat_file, lat)      ! Read in a lattice
call bmad_to_mad_or_xsif ('lat.mad', 'MAD-8', lat) ! create MAD file
```

Information can be lost when creating a *MAD* or *XSIF* file. For example, neither *MAD* nor *XSIF* has the concept of things such as overlays and groups.

### 15.4 Accelerator Markup Language

The Accelerator Markup Language (AML) / Universal Accelerator Parser (UAP) project<sup>[22]</sup> is a collaborative effort with the aim of 1) creating a lattice format (the AML part) that can be used to fully

describe accelerators and storage rings, and 2) producing software (the **UAP** part) that can parse **AML** lattice files. A side benefit of this project is that the **UAP** code has been extended to be able to translate between **AML**, *Bmad*, *MAD-8*, *MAD-X*, and *XSIF*. See the **AML/UAP** project home web page for more details.





## Chapter 16

# Twiss Parameters, Coupling, Chromaticity, Etc.

### 16.1 Ele\_struct Components

The `ele_struct` (§13) has a number of components that hold information on the Twiss parameters, dispersion, and coupling at the exit end of the element. The Twiss parameters of the three normal modes (§10.8) are contained in the `ele%a`, `ele%b`, and `ele%z` components which are of type `twiss_struct`:

```
type twiss_struct
  real(rp) beta      ! Twiss Beta function
  real(rp) alpha     ! Twiss Alpha function
  real(rp) gamma     ! Twiss gamma function
  real(rp) phi       ! Normal mode Phase advance
  real(rp) eta       ! Normal mode dispersion
  real(rp) etap      ! Normal mode dispersion derivative
  real(rp) sigma     ! Normal mode beam size
  real(rp) sigma_p   ! Normal mode beam size derivative
  real(rp) emit      ! Geometric emittance
  real(rp) norm_emit ! Normalized emittance
end type
```

The projected horizontal and vertical dispersions in an `ele_struct` are contained in the `%x` and `%y` components. These components are of type `xy_disp_struct`:

```
type xy_disp_struct
  real(rp) eta      ! Projected dispersion
  real(rp) etap     ! Projected dispersion derivative.
end type
```

The relationship between the projected and normal mode dispersions are given by Eq. (10.72). The 2x2 coupling matrix  $\mathbf{C}$  (Eq. (10.61)) is stored in the `%c(2,2)` component of the `ele_struct` and the  $\gamma$  factor of Eq. (10.61) is stored in the `ele%gamma_c` component. There are several routines to manipulate the coupling factors. For example:

```
c_to_cbar(ele, cbar_mat)      ! Form Cbar(2,2) matrix
make_v_mats(ele, v_mat, v_inv_mat) ! Form V matrices.
```

See §23.17 for a complete listing of such routines.

Since the normal mode and projected dispersions are related, when one is changed within a program the appropriate change must be made to the other. To make sure everything is consistent, the `changed_attribute_bookkeeper` routine can be used.

The `%mode_flip` logical component of an `ele_struct` indicates whether the  $a$  and  $b$  normal modes have been flipped relative to the beginning of the lattice. See Sagan and Rubin[9] for a discussion of this. The convention adopted by *Bmad* is that the `%a` component of all the elements in a lattice will all correspond to the same physical normal mode. Similarly, the `%b` component of all the elements will all correspond to some (other) physical normal mode. That is, at an element where there is a mode flip (with `%mode_flip` set to True), the `%a` component actually corresponds to the **B** matrix element in Eq. (10.60) and vice versa. The advantage of this convention is that routines that calculate properties of the modes (for example the emittance), can ignore whether the modes are flipped or not.

The normal mode analysis of Sagan and Rubin, while it has the benefit of simplicity, is strictly only applicable to lattices where the RF cavities are turned off. The full 6-dimensional analysis is summarized by Wolski[24]. The `normal_mode3_calc` routine perform the full analysis. The results are put in the `%mode3` component of the `ele_struct` which is of type `mode3_struct`:

```
type mode3_struct
  real(rp) v(6,6)
  type (twiss_struct) a, b, c
  type (twiss_struct) x, y
end type
```

The 6-dimensional `mode3%v(6,6)` component is the analog of the 4-dimensional **V** matrix appearing in Eq. (10.59).

## 16.2 Twiss Parameter Calculations

A calculation of the Twiss parameters starts with the Twiss parameters at the beginning of the lattice. For linear machines, these Twiss parameters are generally set in the input lattice file (§7.4). For circular machines, the routine `twiss_at_start` may be used (§7.4)

```
type (lat_struct) lat
...
if (lat%param%lattice_type == circular_lattice$) call twiss_at_start(lat)
```

In either case, the initial Twiss parameters are placed in `lat%ele(0)`.

To propagate the Twiss, coupling and dispersion parameters from the start of the lattice to the end, the routine `twiss_propagate_all` can be used. This routine works by repeated calls to `twiss_propagate1` which does a single propagation from one element to another. The Twiss propagation depends upon the transfer matrices having already computed (§17). `twiss_propagate_all` also computes the twiss parameters for all the lattice branches.

Before any Twiss parameters can be calculated the transfer matrices stored in the lattice elements must be computed. `bmad_parser` does this automatically about the zero orbit. If, to see nonlinear effects, a different orbit needs to be used for the reference, The routine `lat_make_mat6` can be used. For example

```
type (lat_struct) lat
type (coord_struct), allocatable :: orbit(:)
call bmad_parser ('my_lattice', lat)
call closed_orbit_calc (lat, orbit, 4)
call lat_make_mat6 (lat, -1, orbit)
```

This example reads in a lattice, finds the closed orbit which may be non-zero due to, say, kicks due to a separator, and then remakes the transfer matrices (which are stored in `lat%ele(i)%mat6` around the closed orbit.

Once the transfer matrices are calculated the Twiss parameters at the start of the lattice need to be defined. The Twiss parameters at the start are in `lat%ele(0)`. If the lattice is open then generally the Twiss parameters are set in the lattice file or may easily be set in a program. For example

```
lat%ele(0)%x%beta = 1.2
lat%ele(0)%x%alpha = 0.1
lat%ele(0)%x%gamma = (1 + lat%ele(0)%x%alpha**2) / lat%ele(0)%beta
lat%ele(0)%x%eta = 0
```

Note that `%beta`, `%alpha`, and `%gamma` all must be specified.

If the lattice is circular, the routine `twiss_at_start` may be used to calculate the self-consistent starting Twiss parameters:

```
type (lat_struct) lat
call bmad_parser ('my_lattice', lat)
call twiss_at_start (lat)
```

Once the starting Twiss parameters are set, `twiss_propagate_all` can be used to propagate the Twiss parameters to the rest of the elements

```
call twiss_propagate_all (lat)
```

It is important to keep in mind that `lat%ele(i)%x%eta` is the *a*-mode dispersion, not the dispersion along the *x*-axis (§13.6).

The routine `twiss_and_track_at_s` can be used to calculate the Twiss parameters at any given longitudinal location. Alternatively, to propagate the Twiss parameters partially through a given element use the the routine `twiss_and_track_partial`.

## 16.3 Tune Calculation

For a circular lattice, the tune is calculated via the routine `twiss_at_start`. This routine multiplies the transfer matrices of the elements together to form the one-turn transfer matrix. From this the routine extracts the initial Twiss parameters and the tune. The tune is placed in the variables

```
type (lat_struct) lat
lat%a%tune      ! a-mode tune
lat%b%tune      ! b-mode tune
```

The routine `set_tune` can be used to set the transverse tunes:

```
set_tune (phi_a_set, phi_b_set, dk1, lat, orb_, ok)
```

`set_tune` varies quadrupole strengths until the desired tunes are achieved. As input, `set_tune` takes an argument `dk1(:)` which is an array that specifies the relative change to be made to the quadrupoles in the lattice.

To set the longitudinal (synchrotron) tune, the routine `set_z_tune` can be used. `set_z_tune` works by varying rf cavity voltages until the desired tune is achieved.

## 16.4 Chromaticity Calculation

For a circular lattice, `chrom_calc` calculates the chromaticity by calculating the tune change with change in beam energy.

`chrom_tune` sets the chromaticity by varying the sextupoles. This is a very simple routine that simply divides the sextupoles into two families based upon the local beta functions at the sextupoles.

## Chapter 17

# Tracking and Transfer Maps

*Bmad* can do two types of tracking. “**particle tracking**” involves tracking particles which are characterized by its phase space coordinates and spin. The other type of tracking involves “**macroparticles**” (§10.12) which are characterized by a centroid position and a  $6 \times 6$  “sigma” matrix characterizing the size of the macroparticle.

Macroparticle tracking was implemented in *Bmad* in order to simulate particle bunches. The idea was that far fewer macroparticles than particles would be needed to characterize a bunch. In practice, it was found that the complexity of handling the macroparticle sigma matrix more than offset the reduction in the number of particles needed. Hence, while the basic macroparticle tracking routines still exist, macroparticle tracking is not currently maintained and the use of this code is discouraged. Macroparticle tracking could be revived if there is a demonstrated need for it however.

Particle tracking can be divided into “single particle” tracking and “beam” tracking. Single particle tracking is simply tracking a single particle. Beam tracking is tracking an ensemble of particles divided up into a number of bunches that make up the entire beam. Both types particle tracking are covered in this chapter.

### 17.1 The coord\_struct

For particle tracking, the starting point is the `coord_struct` which defines a particle. The definition of the `coord_struct` is

```
type coord_struct
  real(rp) vec(6)      ! (x, px, y, py, z, pz)
  complex(rp) spin(2) ! particle spin in spinor notation
end type
```

To get an orbit, that is, the particle position at every element in a lattice, you will need an array of `coord_struct`s. Since the number of elements in the lattice is not known in advance the array must be declared to be allocatable.

```
type (coord_struct), allocatable :: orbit(:)
```

An example of how to do multi-turn tracking (assuming a circular lattice) is

```
type (lat_struct) lat           ! lattice to track through
type (coord_struct), allocatable :: orbit(:)
...
```

```

call bmad_parser ('this_lattice', lat)
...
call reallocate_coord (orbit, lat%n_ele_max)
orbits(0)%vec = (/ 0.01, 0.2, 0.3, 0.4, 0.0, 0.0 /) ! initialization
do i = 1, n_turns
  call track_all (lat, orbit)
  orbit(0) = orbit(lat%n_ele_track)
end do

```

`orbit(n)` holds the particle's position at the exit end of the  $n^{th}$  element. Since `super_lord` (§13.7) elements have an associated particle position, the upper bound of the `orbit(:)` array must be at least `lat%n_ele_max` (§14.2). The call to `reallocate_coord` does the allocation for the `orbit(:)` array. Since `lat%ele(0)` is essentially a marker element `orbit(0)` is the orbit at the start of the lattice. `track_all` takes `orbit(0)` and tracks through the list of lattice elements until it gets to the last trackable element `lat%n_ele_track` (§14.2).

If you are writing a routine where the `coord_struct` array is local (not passed as an argument to the routine) then you have to decide how to cleanup the allocated `coord_struct` memory at the end of the routine. In general you have two choices: 1) Deallocate the array. This is the cleanest solution but it can be slow since you have to allocate afresh each time the routine is called. 2) Use the `save` attribute so that the array stays around until the next time the routine is called

```

type (coord_struct), allocatable, save :: orb(:)

```

Saving the `coord_struct` is faster but leaves memory tied up.

## 17.2 Tracking Through the Elements

The routine `track1` is the routine that tracks through one element in the lattice. The routine `track_all` calls `track1` in a loop over all elements to track through the entire lattice. Alternatively the routine `track_many` can be used to track through a selective number of elements or to track backwards (See §17.7). The routines used for tracking and closed orbit calculations are listed in Section §23.31.

The `track_all` routine serves as a good example of how tracking works. `track_all` tracks a particle through a lattice from beginning to end. Its code, condensed slightly, is shown in Figure 17.2. The `reallocate_coord` call (line 13) is done in case the number of elements in the lattice has changed. The call to `track1` (line 18) tracks through one element from the exit end of the  $n - 1^{st}$  element to the exit end of the  $n^{th}$  particle. `lat%param%lost` is a logical that signals the calling routine whether a particle has been lost. This generally happens when the particle's position is larger than the aperture. When a particle is lost `lat%param%ix_lost` is used to record in what element the loss occurred.

## 17.3 Closed Orbit

For a circular lattice the closed orbit may be calculated using `closed_orbit_calc`. By default this routine will track in the forward direction which is acceptable unless the particle you are trying to simulate is traveling in the reverse direction and there is radiation damping on. In this case you must tell `closed_orbit_calc` to do backward tracking. This routine works by iteratively converging on the closed orbit using the 1-turn matrix to calculate the next guess. On rare occasions if the nonlinearities are strong enough, this can fail to converge. An alternative routine is `closed_orbit_from_tracking` which tries to do things in a more robust way but with a large speed penalty.

```

1  subroutine track_all (lat, orbit)
2      use bmad_struct
3      use bmad_interface
4      implicit none
5      type (lat_struct) lat
6      type (coord_struct), allocatable :: orbit(:)
7      integer n
8
9      ! Init
10
11      lat%param%lost = .false.
12      if (size(orbit) < lat%n_ele_max+1) &
13          call reallocate_coord (orbit, lat%n_ele_max)
14
15      ! Track through the elements and check for lost particles.
16
17      do n = 1, lat%n_ele_track
18          call track1 (orbit(n-1), lat%ele(n), lat%param, orbit(n))
19          if (lat%param%lost) then
20              lat%param%ix_lost = n
21              return
22          endif
23      enddo
24  end subroutine

```

n

Figure 17.1: Condensed track\_all code.

## 17.4 Apertures

The routine `check_aperture_limit` checks the aperture at a given element. The `%aperture_type` component determines whether the ap

The logical `lat%param%aperture_limit_on` determines if element apertures (See §4.6) are used to determine if a particle has been lost in tracking. The default `lat%param%aperture_limit_on` is True. Even if this is False there is a “hard” aperture limit set by `bmad_com%max_aperture_limit`. This hard limit is used to prevent floating point overflows. The default hard aperture limit is 1000 meters. Additionally, even if a particle is within the hard limit, some routines will mark a particle as lost if the tracking calculation will result in an overflow.

`lat%param%lost` is the logical to check to see if a particle has been lost. `lat%param%ix_lost` is set by `track_all` and gives the index of the element at which a particle is lost. `%param%end_lost_at` gives which end the particle was lost at. The possible values for `lat%param%end_lost_at` are:

```

entrance_end$
exit_end$

```

When tracking forward, if a particle is lost at the exit end of an element then the place where the orbit was outside the aperture is at `orbit(ix)` where `ix` is the index of the element where the particle is lost (given by `lat%param%ix_lost`). If the particle is lost at the entrance end then the appropriate index is one less (remember that `orbit(i)` is the orbit at the exit end of an element).

To tell how a particle is lost, check the `lat%param%plane_lost_at` parameter. Possible values for this are:

```

x_plane$

```

```

y_plane$
z_plane$

```

`x_plane$` and `y_plane$` indicate that the particle was lost either horizontally, or vertically. `z_plane$` indicates that the particle was turned around in an `lcavity` element. That is, the cavity was deaccelerating the particle and the particle did not have enough energy going into the cavity to make it to the exit.

## 17.5 Tracking Methods

For each element the method of tracking may be set either via the input lattice file (see §5.1) or directly in the program by setting the `%tracking_method` attribute of an element

```

type (ele_struct) ele
...
ele%tracking_method = boris$ ! for boris tracking

```

To form the corresponding parameter to a given tracking method just put “\$” after the name. For example, the `bmad_standard` tracking method is specified by the `bmad_standard$` parameter.

It should be noted that except for `linear` tracking, none of the *Bmad* tracking routines make use of the `ele%mat6` transfer matrix. The reverse, however, is not true. The transfer matrix routines (`lat_make_mat6`, etc.) will do tracking.

*Bmad* simulates radiation damping and excitation by applying a kick just before and after each element. To turn on radiation damping and/or excitation use the `setup_radiation_tracking` routine.

## 17.6 Taylor Maps

A list of routines for manipulating Taylor maps is given in §23.30. The order of the Taylor maps is set in the lattice file using the `parameter` statement (§7.1). In a program this can be overridden using the routine `set_taylor_order`. The routine `taylor_coef` can be used to get the coefficient of any given term.

Transfer Taylor maps for an element are generated as needed when the `ele%tracking_method` or `ele%mat6_calc_method` is set to `Symp_Lie_Bmad`, `Symp_Lie_PTC`, `Symp_Map`, or `Taylor`. Since generating a map can take an appreciable time, *Bmad* follows the rule that once generated, these maps are never regenerated unless an element attribute is changed. To generate a Taylor map within an element irregardless of the `ele%tracking_method` or `ele%mat6_calc_method` settings use the routine `ele_to_taylor`. This routine will kill any old Taylor map before making any new one. To kill a Taylor map (which frees up the memory it takes up) use the routine `kill_taylor`.

To test whether a `taylor_struct` variable has an associated Taylor map. That is, to test whether memory has been allocated for the map, use the Fortran associated function:

```

type (bmad_taylor) taylor(6)
...
if (associated(taylor(1)%term)) then ! If has a map ...
...

```

To concatenate the Taylor maps in a set of elements the routine `concat_taylor` can be used

```

type (lat_struct) lat ! lattice
type (taylor_struct) taylor(6) ! taylor map

```



```

...
call taylor_make_unit (taylor) ! Make a unit map
do i = i1+1, i2
  call concat_taylor (taylor, lat%ele(i)%taylor, taylor)
enddo

```

The above example forms the transfer Taylor map starting at the end of element `i1` to the end of element `i2`. Note: This example assumes that all the elements have a Taylor map. The problem with concatenating maps is that if there is a constant term in the map “feed down” can make the result inaccurate (§10.3). To get around this one can “track” a Taylor map through an element using symplectic integration.

```

type (lat_struct) lat          ! lattice
type (taylor_struct) taylor(6) ! Taylor map
...
call taylor_make_unit (taylor) ! Make a unit map
do i = i1+1, i2
  call call taylor_propagate1 (taylor, lat%ele(i), lat%param)
enddo

```

Symplectic integration is typically much slower than concatenation. The width of an integration step is given by `%ele%value(ds_step$)`. The attribute `%ele%value(num_steps$)`, which gives the number of integration steps, is a dependent variable (§4.1) and should not be set directly. The order of the integrator (§10.3) is given by `%ele%integrator_order`. PTC (§19) currently implements integrators of order 2, 4, or 6.

## 17.7 Reverse Tracking

There are two ways to do reverse tracking in which the particle goes in the direction of decreasing `s`. The first way is to use the `track_many` routine. See the `track_many` routine for more details. The advantage of using `track_many` is that it is simple. The disadvantage is that it can slow things down some since each element goes through a reversal process every time it is tracked through. If a program is doing a lot of tracking the other option is to form a reversed lattice with the elements in the reverse order and track through that. The routine `lat_reverse` will do this. One must be somewhat careful since the reversed lattice uses a reversed coordinate system. The transformation between the reversed and unreversed lattices is

$$(x, p_x, y, p_y, z, p_z) \rightarrow (x, -p_x, y, -p_y, -z, p_z) \quad (17.1)$$

See the `lat_reverse` routine for more details.

Generally tracking backwards is simply the reverse of tracking forwards (time reversal symmetry). That is, if you start at some place, track forward for some distance and then track back to the starting place the ending orbit will be equal to the starting orbit. However, it should always be kept in mind that radiation damping or excitation breaks this symmetry.

## 17.8 Particle Distribution Tracking

Initializing a distribution of particles to conform to some initial set of Twiss parameters and emittances can be done using the routine `init_beam_distribution`.

## 17.9 Spin Tracking

Spin tracking has been implemented for `bmad_standard`, `boris` and `adaptive_boris` tracking methods. To turn spin tracking on use the `bmad_com%spin_tracking_on` flag. Then, after properly initializing the spin in the `coord_struct`, calls to `track1` will track both the particle orbit and the spin.

## 17.10 Custom Field Calculations

Custom Electric and Magnetic field calculations are used with `runge_kutta` and `boris` tracking (See §5.3). To implement custom field calculations the `ele%field_calc` component of an element must be set to `custom$`. This can be done either through the lattice input file or within a program. Additionally a routine `em_field_custom` must be linked with any program using the custom calculations.

## Chapter 18

# Miscellaneous Programming

### 18.1 Custom Elements

Routines to handle custom elements (§2.8) must be supplied by the creator of the elements. There are potentially four routines that must be written to implement a custom element:

```
radiation_integrals_custom
em_field_custom
make_mat6_custom
track1_custom
```

[Use `getf` for more details about the argument lists for these routines.] The *Bmad* library has dummy routines of the same name to keep the linker happy when custom routines are not implemented. These dummy routines, if called, will print an error message and stop the program. The custom routines are called by their corresponding regular routines. By “regular” routine it is meant the routine without the “\_custom” suffix. For example, `radiation_integrals` will call `radiation_integrals_custom`. Thus if a program using the custom code does not call a particular regular routine, the program does not have to implement the corresponding custom routine.

```
track1_bunch_custom
```

### 18.2 Physical Constants

Common physical constants that can be used in any expression are defined in the file `sim_utils/interfaces/physical_constants.f90`

This includes the constants given in Section §1.8 along with

```
mu_0_vac          ! Permeability of free space
eps_0_vac          ! Permittivity of free space
g_factor_electron ! Electron anomalous gyro-magnetic moment
g_factor_proton    ! Proton anomalous gyro-magnetic moment
```

### 18.3 Common Structures

**NOTE: THIS CHAPTER IS UNDER CONSTRUCTION!**

There are two common structures used by Bmad for communication between routines. These are `bmad_com` which is a `bmad_common_struct` structure and `bmad_status` which is a `bmad_status_struct` structure.

The `bmad_status_struct` structure is:

```
type bmad_status_struct
  integer :: status      = ok$      ! Computation status
  logical :: ok          = .true.   ! Error flag
  logical :: type_out    = .true.   ! Print error messages?
  logical :: sub_type_out = .true.   !
  logical :: exit_on_error = .true. ! Exit program on error?
end type
```

## Chapter 19

# Etienne Forest's PTC/FPP

Etienne Forest[20] has written what is actually two software libraries: FPP and PTC. The software and a manual can be obtained at

`<http://acc-physics.kek.jp/forest/PTC/Introduction.html>`

FPP stands for “Fully Polymorphic Package.” What this library does is implement Taylor maps (aka Truncated Power Series Algebra or TPSA) and Lie algebraic operations. Thus in FPP you can define a Hamiltonian and then generate the Taylor map for this Hamiltonian. FPP is very general. It can work with an arbitrary number of dimensions. FPP, however, is a purely mathematical package in the sense that it knows nothing about accelerator physics. That is, it does not know about bends, quadrupoles or any other kind of element, it has no conception of a lattice (a string of elements), it doesn’t know anything about Twiss parameters, etc.

This is where PTC (Polymorphic Tracking Code) comes in. PTC implements the high energy physics stuff and uses FPP as the engine to do the Lie algebraic calculations. PTC is a very general package and *Bmad* only makes use of a small part of its features. Essentially the part that *Bmad* uses is the part that creates Taylor maps and does symplectic integration. Not used is, for example, the Computational part that does normal form analysis Twiss parameters (*Bmad* does its own), beam envelope tracking, etc. PTC also has the ability, which *Bmad* does not take advantage of, to define “knobs” which means that the Taylor map can be a function of the phase space coordinates plus other variables (for example the strength of a quadrupole). The list goes on.

*Bmad* has been structured so that “normally” a programmer will not have to deal with PTC (in general when PTC is written it is meant PTC in conjunction with FPP) subroutines directly.

### 19.1 Phase Space

PTC uses different longitudinal phase space coordinates compared to *Bmad*. *Bmad*’s phase space coordinates are

$$(x, p_x, y, p_y, z, p_z) \tag{19.1}$$

PTC uses

$$(x, p_x, y, p_y, p_z, ct \sim -z) \tag{19.2}$$

`vec_bmad_to_ptc` and `vec_ptc_to_bmad` are conversion routines that translate between the two. Actually there are a number of conversion routines that translate between *Bmad* and PTC structures. See §23.26 for more details.

## 19.2 Initialization

One important parameter in PTC is the order of the Taylor maps. By default *Bmad* will set this to 3. The order can be set within a lattice file using the `parameter[taylor_order]` attribute. In a program the order can be set using `set_ptc`. In fact `set_ptc` must be called by a program before PTC can be used. `bmad_parser` will do this when reading in a lattice file. That is, if a program does not use `bmad_parser` then to use PTC it must call `set_ptc`. Note that resetting PTC to a different order reinitializes PTC's internal memory so one must be careful if one wants to change the order in mid program.

## 19.3 Taylor Maps

FPP stores its `real_8` Taylor maps in such a way that it is not easy to access them directly to look at the particular terms. To simplify life, Etienne has implemented the `universal_taylor` structure:

```
type universal_taylor
  integer, pointer :: n      ! Number of coefficients
  integer, pointer :: nv     ! Number of variables
  real(dp), pointer :: c(:)  ! Coefficients C(N)
  integer, pointer :: j(:, :) ! Exponents of each coefficients J(N,NV)
end type
```

*Bmad* always sets `nv = 6`. *Bmad* overloads the equal sign to call routines to convert between Etienne's `real_8` Taylor maps and `universal_taylor`:

```
type (real_8) tlr(6)      ! Taylor map
type (universal_taylor) ut(6) ! Taylor map
...
tlr = ut                  ! Convert universal_taylor -> real_8
ut = tlr                  ! Convert real_8 -> universal_taylor
```

## Chapter 20

# C++ Interface

To ease the task of using C or C++ routines with *Bmad*, the *Bmad* library defines a set of C++ classes in one-to-one correspondence with the major *Bmad* structures as given in table 20. In addition to the C++ classes, the *Bmad* library defines a set of conversion routines to transfer data values between the *Bmad* Fortran structures and the corresponding C++ classes.

<i>Bmad structure</i>	<i>C++ Class</i>
amode_struct	C_amode
bmad_com_struct	C_bmad_com
control_struct	C_control
coord_struct	C_coord
ele_struct	C_ele
em_field_struct	C_em_field
floor_position_struct	C_floor_position
linac_mode_struct	C_linac_mode
lr_wake_struct	C_lr_wake
modes_struct	C_modes
mode_info_struct	C_mode_info
orbit_struct	C_orbit
param_struct	C_param
lat_struct	C_lat
sr_wake_struct	C_sr_wake
taylor_struct	C_taylor
taylor_term_struct	C_taylor_term
twiss_struct	C_twiss
wake_struct	C_wake
wig_term_struct	C_wig_term

Table 20.1: Bmad structures and their corresponding C++ classes.

```

1  subroutine test
2      use bmad
3      type (lat_struct) lattice    // lattice on Fortran side
4      ! ... setup lattice ...
5      call c_wrapper (lattice)    // Pass lattice to C++ routine
6  end subroutine

```

Figure 20.1: Example Fortran routine calling a C++ routine.

```

1  #include "cpp_and_bmad.h"
2
3  external "C" c_wrapper(lat_struct* f_lat) {
4      using namespace Bmad;
5      C_lat c_lat;           // Lattice on \cpp side
6      f_lat >> c_lat;        // Transfer info: F -> \cpp
7      // ... do calculations ...
8      cout << c_lat.name << " " << c_lat.ele[1].value[K1] << endl;
9      c_lat >> f_lat;        // Transfer back: C++ -> F
10 }

```

Figure 20.2: Example C++ routine callable from a Fortran routine.

## 20.1 C++ Classes

The C++ classes are defined in a header file `cpp_and_bmad.h`. Generally, The C++ classes have been set up to simply mirror the corresponding *Bmad* structures. For example, the `C_lat` class has a string component named `.version` that mirrors the `%version` component of the `lat_struct` structure. There are some exceptions here and `getf` (Section §11.2) can be used to quickly examine the definition of any C++ class. Note that while generally the same component name is used for both the *Bmad* structures and the C++ classes in the case where a component of a structure has a trailing underscore `_` the corresponding C++ structure does not. For example, the analog on the C++ side of the `%ele_` component of a `lat_struct` is `.ele`

All of the C++ arrays and matrices are zero based so that, for example, the index of the `.vec[i]` array in a `C_coord` runs from 0 through 5 and not 1 through 6 as on the Fortran side. Notice that for a `lat_struct` the `%ele(0:)` component has a starting index of zero so there is no off-by-one problem here.

A header file `bmad_parameters.h` defines corresponding *Bmad* parameters for all C++ routine. The *Bmad* parameters are in a namespace called `Bmad`. The convention is that the name of a corresponding C++ parameter is obtained by dropping the ending `$` (if there is one) and converting to uppercase. For example, `electron$` on the Fortran side converts to `Bmad::ELECTRON` in C++. To keep the conversion of the of parameters like `k1$` which are used with the `ele%value(0:n_attrib_maxx)` array, the corresponding `ele.value[]` array has goes from 0 to `Bmad::N_ATTRIB_MAXX` with the 0th element being unused.



```

1  #include "cpp_and_bmad.h"
2  external "C" void f_wrapper(C_lat&);
3
4  void test () {
5      C_lat c_lat;          // lattice on C++ side
6      f_wrapper(c_lat);     // pass lattice to Fortran side
7  }

```

Figure 20.3: Example C++ routine calling a Fortran routine.

```

1  subroutine f_wrapper (c_lat)
2      type (c_dummy_struct) c_lat      ! C++ lattice
3      type (lat_struct) f_lat          ! lattice on Fortran side
4      call lat_to_f (c_lat, f_lat)     ! Transfer info: C++ -> F
5      ! ... do some calculations ...
6      call lat_to_c (f_lat, c_lat)     ! Transfer back: F -> C++
7  end subroutine

```

Figure 20.4: Example Fortran routine callable from a C++ routine.

## 20.2 Fortran calling C++

A simple example of a Fortran routine calling a C++ routine is shown in figures 20.1 and 20.2. The Fortran calling routine simply passes the `lat_struct` structure to the C++ routine. On the C++ side the C++ routine must be declared as `external "C"` to make sure the name mangling between C++ and Fortran is consistent and to make sure the arguments are passed correctly. Notice that on the C++ side the routine name has a trailing underscore `_` while on the Fortran side it does not. In the C++ routine the `lat_struct` argument is declared as a pointer. The `lat_struct` class on the C++ side holds no data and is simply meant as a place holder. There are corresponding “dummy” classes for each *Bmad* structure. These classes should never be instantiated like:

```
lat_struct lattice; // Wrong!
```

The C++ compiler would not allocate any storage for this variable and the use of this variable will (hopefully) lead to the program bombing.

The `>>` operator on the C++ side has been overloaded to do the Fortran to C++ conversion. The only exception is with the `bmad_common_struct` structure where the data on the Fortran side comes directly from the *Bmad* `bmad_common_struct` common block. In this case no structure variable is passed to the C++ routine and the syntax on the C++ side for transferring this common block data back and forth looks like:

```

C_bmad_com c_com;          // Variable to hold Bmad common block data
bmad_com_to_c(c_com); // Load C++ variable
bmad_com_to_f(c_com); // Transfer back to common block

```

## 20.3 C++ calling Fortran

An example of a C++ routine calling a Fortran routine is shown in figures 20.4 and 20.3. `lat_to_f` and `lat_to_c` converts between a `lat_struct` and a `C_lat`. In general the routines names to convert between a `xyz_struct` structure and a `C_xyz` class are called `xyz_to_f` and `xyz_to_c` on the Fortran side. On the C++ side these routines get a trailing underscore `_` and are overloaded by the `>>` operator.

In general it is not possible to call an arbitrary Fortran routine directly from C++ or vice versa. There are certain restrictions that Fortran and C++ routines must observe to be callable from the other language. In practice this means that to use bmad from a C++ main program the interface routine (in this example `f_wrapper`) will be written in Fortran and the appropriate *Bmad* routines are called directly by the interface.

## Chapter 21

# Quick\_Plot Plotting

The plotting package included in the *Bmad* distribution is PGPLOT (see §11.1). One drawback of PGPLOT is that the arguments to PGPLOT's subroutines are not always conveniently structured. To remedy this a suite of wrapper routines have been developed which can be used to drive PGPLOT. This suite is called *Quick Plot* and lives in the `sim_utils` library which comes with the Bmad distribution. A quick reference guide can be seen online by using the command `getf quick_plot`. For quick identification in a program, all *Quick Plot* subroutines start with a `qp_` prefix. Also, by convention, all PGPLOT subroutines start with a `pg` prefix.

While *Quick Plot* covers most of the features of PGPLOT, *Quick Plot* is still a work in progress. For example, contour plots have not yet been implemented in *Quick Plot*. If you see a feature that is lacking in *Quick Plot* please do not hesitate to make a request to `dcsl6@cornell.edu`.

Note: PGPLOT uses single precision `real(4)` numbers while *Quick Plot* uses `real(rp)` numbers. If you use any PGPLOT subroutines directly be careful of this.

### 21.1 An Example

An example of how *Quick Plot* can be used in a program is shown in Figure 21.1. In the *Bmad* distribution a copy of this program is in the file

```
sim_utils/plot_example/plot_example.f90
```

The `plot_example.f90` program generates the figure shown in Figure 21.2 from the input file named `plot.dat`. The first few lines of the data file are

```
&parameters
  title = "A Tale of Two Graphs"
/
```

Any junk here...

Col1	Col2	Col3	Col4	Col5
0	0.0000	0.1000	0.0000	-0.0125
1	0.0001	0.0995	0.0101	-0.0127
2	0.0004	0.0980	0.0203	-0.0130
3	0.0009	0.0955	0.0304	-0.0132
...				

```

1  program example_plot
2      use quick_plot
3      integer id
4      character(1) ans
5
6      ! Generate PS and X-windows plots.
7      call qp_open_page ("PS-L") ! Tell \quickplot to generate a PS file.
8      call plot_it              ! Generate the plot
9      call qp_close_page       ! quick_plot.ps is the file name
10     call qp_open_page ("X", id, 600.0_rp, 470.0_rp, "POINTS")
11     call plot_it
12     write (*, "(a)", advance = "NO") " Hit any class to end program: "
13     accept "(a)", ans
14
15     !-----
16     contains
17     subroutine plot_it              ! This generates the plot
18         real(rp), allocatable :: x(:), y(:), z(:), t(:)
19         real(rp) x_axis_min, x_axis_max, y_axis_min, y_axis_max
20         integer x_places, x_divisions, y_places, y_divisions
21         character(80) title
22         logical err_flag
23         namelist / parameters / title
24
25         ! Read in the data
26         open (1, file = "plot.dat", status = "old")
27         read (1, nml = parameters) ! read in the parameters.
28         call qp_read_data (1, err_flag, x, 1, y, 3, z, 4, t, 5) ! read in the data.
29         close (1)
30
31         ! Setup the margins and page border and draw the title
32         call qp_set_page_border (0.01_rp, 0.02_rp, 0.2_rp, 0.2_rp, "%PAGE")
33         call qp_set_margin (0.07_rp, 0.05_rp, 0.05_rp, 0.05_rp, "%PAGE")
34         call qp_draw_text (title, 0.5_rp, 0.85_rp, "%PAGE", "CT")
35
36         ! draw the left graph
37         call qp_set_box (1, 1, 2, 1)
38         call qp_calc_and_set_axis ("X", minval(x), maxval(x), 4, 8, "ZERO_AT_END")
39         call qp_calc_and_set_axis ("Y", minval(z), maxval(z), 4, 8, "GENERAL")
40         call qp_draw_axes ("X\dlab\u", "\gb(\A)")
41         call qp_draw_data (x, y, symbol_every = 0)
42
43         call qp_save_state (.true.)
44         call qp_set_symbol_attrib (times$, color = blue$, height = 20.0_rp)
45         call qp_set_line_attrib ("PLOT", color = blue$, style = dashed$)
46         call qp_draw_data (x, z, symbol_every = 5)
47         call qp_restore_state
48
49         ! draw the right graph. star5_filled$ is a five pointed star.
50         call qp_save_state (.true.)
51         call qp_set_box (2, 1, 2, 1)
52         call qp_set_graph_attrib (draw_grid = .false.)
53         call qp_set_symbol_attrib (star5_filled$, height = 10.0_rp)
54         call qp_set_axis ("Y", -0.1_rp, 0.1_rp, 4, 2)
55         call qp_set_axis ('Y2', 1.0_rp, 100.0_rp, label = 'Y2 axis', &
56             draw_numbers = .true., ax_type = 'LOG')
57         call qp_draw_axes ("\"m1 \m2 \m3 \m4 \m5 \m6 \m7", "\fsLY\fn", &
58             title = "That Darn Graph")
59         call qp_draw_data (x, t, draw_line = .false., symbol_every = 4)
60         call qp_restore_state
61     end subroutine
62 end program

```

Figure 21.1: *Quick Plot* example program.

## A Tale of Two Graphs

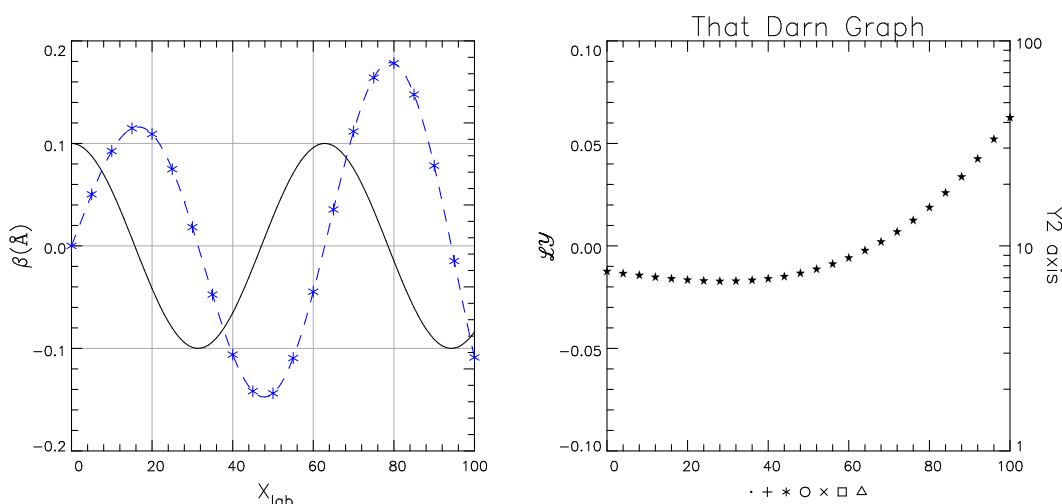


Figure 21.2: Output of plot\_example.f90.

The program first creates a PostScript file for printing on lines 7 through 9 and then makes an X-windows plot on lines 10 and 11. The write/accept lines 12 and 13 are to pause the program to prevent the X-window from immediately closing upon termination of the program.

The heart of the plotting is in the subroutine `plot_it` beginning on line 17. The namelist read on line 27 shows how both parameters and data can be stored in the same file so that a plotting program can be automatically told what the appropriate plot labels are. The `qp_draw_text` call on line 34 draws the title above the two graphs.

The `qp_read_data` call on line 28 will skip any “header” lines (lines that do not begin with something that looks like a number) in the data file. In this instance `qp_read_data` will read the first, third fourth and fifth data columns and put them into the `x`, `y`, `z`, and `t` arrays.

`qp_set_page_border`, `qp_set_box`, and `qp_set_margin` sets where the graph is going to be placed. `qp_set_box(1, 1, 2, 1)` on line 37 tells *Quick Plot* to put the first graph in the left box of a 2 box grid. The `qp_set_margin` on line 33 sets the margins between the box and the graph axes.

`qp_calc_and_set_axis` on lines 38 and 39 are used to scale the axes. “ZERO\_AT\_END” ensures that the  $x$ -axis starts (or stops) at zero. `qp_calc_and_set_axis` is told to restrict the number of major divisions to be between 4 and 8. For the horizontal axis, as can be seen in Figure 21.2, it chooses 5 divisions.

After drawing the first data curve (the solid curve) in the left graph, the routines `qp_set_symbol_attrib` and `qp_set_line_attrib` are called on lines 44 and 45 to plot the next data curve in blue with a dashed line style. By default, this curve goes where the last one did: in the left graph. To keep the setting of the line and symbol attributes from affecting other plots the routines `qp_save_state` and `qp_restore_state` on lines 43 and 47 are used. `qp_save_state` saves the current attributes in a attribute stack. `qp_restore_state` restores the saved attributes from the attribute stack. `qp_draw_axes` is called on line 40 to draw the  $x$  and  $y$ -axes along, and `qp_draw_data` is called on lines 41 and 46 to draw the two data curves.

Lines 50 through 60 draw the third curve in the right hand graph. The `qp_set_axis` call on lines 55/56 sets a log scale for the  $y_2$  (right hand) axis. The syntax of the string arguments of `qp_draw_axes` in lines 40 and 57/58 comes from PGPLOT and allows special symbols along with subscripts and superscripts.

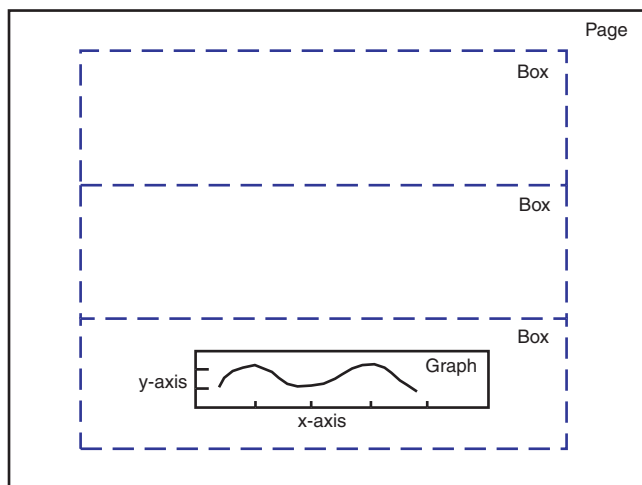


Figure 21.3: A Graph within a Box within a Page.

## 21.2 Plotting Coordinates

*Quick Plot* uses the following concepts as shown in Figure 21.3

- PAGE -- The entire drawing surface.
- BOX -- The area of the page that a graph is placed into.
- GRAPH -- The actual plotting area within the bounds of the axes.

In case you need to refer to the PGPLOT routines the correspondence between this and PGPLOT is:

QUICK_PLOT	PGPLOT
-----	-----
PAGE	VIEW SURFACE
BOX	No corresponding entity.
GRAPH	VIEWPORT and WINDOW

Essentially the VIEWPORT is the region outside of which lines and symbols will be clipped (if clipping is turned on) and the WINDOW defines the plot area. I'm not sure why PGPLOT makes a distinction, but VIEWPORT and WINDOW are always the same region.

`qp_open_page` determines the size of the **page** if it is settable (like for X-windows). The page is divided up into a grid of boxes. For example, in Figure 21.3, the grid is 1 box wide by 3 boxes tall. The border between the grid of boxes and the edges of the page are set by `qp_set_page_border`. The box that the graph falls into is set by `qp_set_box`. The default is to have no margins with 1 box covering the entire page. The `qp_set_margin` routine sets the distance between the box edges and the axes (See the PGPLOT manual for more details).

## 21.3 Length and Position Units

Typically there is an optional **units** argument for *Quick Plot* routines that have length and/or position arguments. For example, using `getf` one can see that the arguments for `qp_draw_rectangle` are

```
Subroutine qp_draw_rectangle (x1, x2, y1, y2, units, color, width, style, clip)
```

The **units** argument is a character string which is divided into three parts. The syntax of the **units** argument is

`unit_type/ref_object/corner`

The first part `unit_type` gives the type of units

```
"%"      -- Percent.
"DATA"   -- Data units. (Draw default)
"MM"     -- millimeters.
"INCH"   -- Inches. (Set default)
"POINTS" -- Printers points (72 points = 1 inch, 1pt is about 1pixel).
```

The second and third parts give the reference point for a position. The second part specifies the reference object

```
"PAGE" -- Relative to the page (Set default).
"BOX"  -- Relative to the box.
"GRAPH" -- Relative to the graph (Draw default).
```

The third part gives corner of the reference object that is the reference point

```
"LB" -- Left Bottom (Set and Draw default).
"LT" -- Left Top.
"RB" -- Right Bottom.
"RT" -- Right Top.
```

Notes:

- The DATA unit type, by definition, always uses the lower left corner of the GRAPH as a reference point.
- For the % `unit_type` the / between `unit_type` and `ref_object` can be omitted.
- If the `corner` is specified then the `ref_object` must appear also.
- Everything must be in upper case.
- For some routines (`qp_set_margin`, etc.) only a relative distance is needed. In this case the `ref_object/corner` part, if present, is ignored.
- The `units` argument is typically an optional argument. If not present the default units will be used. There are actually two defaults: The draw default is used for drawing text, symbols, or whatever. The set default is used for setting margins, and other lengths. Initially the draw default is DATA/GRAPH/LB and the set default is INCH/PAGE/LB. Use `qp_set_parameters` to change this.

Examples:

```
"DATA"      -- This is the draw default.
"DATA/GRAPH/LB" -- Same as above.
"DATA/BOX/RT" -- ILLEGAL: DATA must always go with GRAPH/LB.
"%PAGE/LT"   -- Percentage of page so (0.0, 1.0) = RT of page.
"%BOX"       -- Percentage of box so (1.0, 1.0) = RT of box.
"INCH/PAGE"  -- Inches from LB of page.
```

## 21.4 Y2 and X2 axes

The top and right axes of a graph are known as X2 and Y2 respectively as shown in Figure 21.3. Normally the X2 axis mirrors the X axis and the Y2 axis mirrors the Y axis in that the tick marks and axis numbering for the X2 and Y2 axes are the same as the X and Y axes respectively. `qp_set_axis` can be used to disable mirroring. For example:

call `qp_set_axis ("Y2", mirror = .false.)` ! y2-axis now independent of y.  
`qp_set_axis` can also be used to set Y2 axis parameters (axis minimum, maximum, etc.) and setting the Y2 or X2 axis minimum or maximum will, by default, turn off mirroring.

Note that the default is for the X2 and Y2 axis numbering not to be shown. To enable or disable axis numbering again use `qp_set_axis`. For example:

```
call qp_set_axis ("Y2", draw_numbers = .true.) ! draw y2 axis numbers
```

To plot data using the X2 or Y2 scale use the `qp_use_axis` routine. For example:

```
call qp_save_state (.true.)
call qp_use_axis (y = 'Y2')
! ... Do some data plotting here ...
call qp_restore_state
```

## 21.5 Text

## 21.6 Styles

Symbolic constants have been defined for *Quick Plot* subroutine arguments that are used to choose various styles. As an example of this is in lines 44 and 45 of Figure 21.1. The numbers in the following are the PGPLOT equivalents.

The *Quick Plot* line styles are:

1 -- solid\$	Solid
2 -- dashed\$	Dashed
3 -- dash_dot\$	Dash--dot
4 -- dotted\$	Dotted
5 -- dash_dot3\$	Dash--dot--dot--dot

The color styles in *Quick Plot* are:

0 -- White\$	(actually the background color)
1 -- Black\$	(actually the foreground color)
2 -- Red\$	
3 -- Green\$	
4 -- Blue\$	
5 -- Cyan\$	
6 -- Magenta\$	
7 -- Yellow\$	
8 -- Orange\$	
9 -- Yellow_Green\$	
10 -- Light_Green\$	
11 -- Navy_Blue\$	
12 -- Purple\$	
13 -- Reddish_Purple\$	
14 -- Dark_Grey\$	
15 -- Light_Grey\$	

The fill styles are:

1 -- solid_fill\$
2 -- no_fill\$
3 -- hatched\$
4 -- cross_hatched\$




























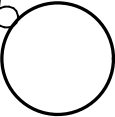
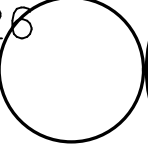
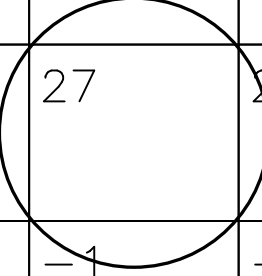












0 	1 	2 	3 	4 
5 	6 	7 	8 	9 
10 	11 	12 	13 	14 
15 	16 	17 	18 	19 
20 	21 	22 	23 	24 
25 	26 	27 	28 	29 
30 	31 	-1 	-2 	-3 
-4 	-5 	-6 	-7 	-8 

Table 21.1: Plotting Symbols at Height = 40.0

<code>\u</code>	Start a superscript, or end a subscript
<code>\d</code>	Start a subscript, or end a superscript (note that <code>\u</code> and <code>\d</code> must always be used in pairs)
<code>\b</code>	Backspace (i.e., do not advance text pointer after plotting the previous character)
<code>\fn</code>	Switch to Normal font (1)
<code>\fr</code>	Switch to Roman font (2)
<code>\fi</code>	Switch to Italic font (3)
<code>\fs</code>	Switch to Script font (4)
<code>\\</code>	Backslash character ( <code>\</code> )
<code>\x</code>	Multiplication sign ( $\times$ )
<code>\.</code>	Centered dot ( $\cdot$ )
<code>\A</code>	Angstrom symbol ( $\text{\AA}$ )
<code>\gx</code>	Greek letter corresponding to roman letter x
<code>\mn \mnn</code>	Graph marker number $n$ or $nn$ (1-31)
<code>\(nnnn)</code>	Character number $nnnn$ (1 to 4 decimal digits) from the Hershey character set; the closing parenthesis may be omitted if the next character is neither a digit nor “ <code>)</code> ”. This makes a number of special characters (e.g., mathematical, musical, astronomical, and cartographical symbols) available.

Table 21.2: PGPLOT Escape Sequences.

The symbol types are:

```

0 -- square$
1 -- dot$
2 -- plus$
3 -- times$
4 -- circle$
5 -- x_symbol$
7 -- triangle$
8 -- circle_plus$
9 -- circle_dot$
10 -- square_concave$
11 -- diamond$
12 -- star5$
13 -- triangle_filled$
14 -- red_cross$
15 -- star_of_david$
16 -- square_filled$
17 -- circle_filled$
18 -- star5_filled$

```

Beside this list, PGPLOT maps other numbers onto symbol types. The PGPLOT list of symbols is:

```

-3 ... -31 - a regular polygon with abs(type) edges.
          -2 - Same as -1.
          -1 - Dot with diameter = current line width.
0 ... 31 - Standard marker symbols.
32 ... 127 - ASCII characters (in the current font).
          E.G. to use letter F as a marker, set type = ICHAR("F").
> 127 - A Hershey symbol number.

```

Roman	a	b	g	d	e	z	y	h	i	k	l	m
Greek	$\alpha$	$\beta$	$\gamma$	$\delta$	$\epsilon$	$\zeta$	$\eta$	$\theta$	$\iota$	$\kappa$	$\lambda$	$\mu$
Roman	n	c	o	p	r	s	t	u	f	x	q	w
Greek	$\nu$	$\xi$	$\omicron$	$\pi$	$\rho$	$\sigma$	$\tau$	$\upsilon$	$\phi$	$\chi$	$\psi$	$\omega$
Roman	A	B	G	D	E	Z	Y	H	I	K	L	M
Greek	A	B	$\Gamma$	$\Delta$	E	Z	H	$\Theta$	I	K	$\Lambda$	M
Roman	N	C	O	P	R	S	T	U	F	X	Q	W
Greek	N	$\Xi$	$\O$	$\Pi$	P	$\Sigma$	T	$\Upsilon$	$\Phi$	X	$\Psi$	$\Omega$

Table 21.3: Conversion for the string " $\backslash g<r>$ " where "<r>" is a Roman character to the corresponding Greek character.

Table 21.1 shows some of the symbols and there associated numbers. Note: At constant height PGPLOT gives symbols of different size. To partially overcome this, *Quick Plot* scales some of the symbols to give a more uniform appearance. Table 21.1 was generated using a height of 40 via the call

```
call qp_draw_symbol (0.5_rp, 0.5_rp, "%BOX", k, height = 40.0_rp)
```

Table 21.3 shows how the character string " $\backslash g<r>$ ", where "<r>" is a Roman letter, map onto the Greek character set.

PGPLOT defines certain escape sequences that can be used in text strings to draw Greek letters, etc. These escape sequences are given in Table 21.2.

PGPLOT defines a text background index:

- 1 - Transparent background.
- 0 - Erase underlying graphics before drawing text.
- 1 to 255 - Opaque with the number specifying the color index.

## 21.7 Structures

*Quick Plot* uses several structures to hold data. The structure that defines a line is a `qp_line_struct`

```
type qp_line_struct
  integer width      ! Line width. Default = 1
  integer color      ! Line color. Default = black$
  integer style      ! Line style. Default = solid$
end type
```

The `qp_symbol_struct` defines how symbols are drawn

```
type qp_symbol_struct
  integer type        ! Default = circle_dot$
  real(rp) height     ! Default = 6.0 (points)
  integer color       ! Default = black$
  integer fill        ! Default = solid_fill$
  integer line_width  ! Default = 1
end type
```

The `qp_axis_struct` defines how axes are drawn

```

type qp_axis_struct
  character(80) label      ! Axis label.
  real(rp) min            ! Axis range left/bottom number.
  real(rp) max            ! Axis range right/top number.
  real(rp) number_offset  ! Offset in inches of numbering from the axis line.
                          ! Default = 0.05
  real(rp) label_offset   ! Offset in inches of the label from the numbering.
                          ! Default = 0.05
  real(rp) major_tick_len ! Length of the major ticks in inches. Def = 0.10
  real(rp) minor_tick_len ! Length of the minor ticks in inches. Def = 0.06
  integer major_div        ! Number of major divisions. Default = 5
  integer minor_div        ! Number of minor divisions. 0 = auto-choose. Default = 0
  integer minor_div_max    ! Maximum number for auto choose. Default = 5
  integer places           ! Places after the decimal point. Default = 0
  character(16) type       ! For log scales. Not yet implemented.
  integer tick_side        ! +1 = draw to the inside, 0 = both, -1 = outside.
                          ! Default = +1
  integer number_side      ! +1 = draw to the inside, -1 = outside.
                          ! Default = -1
  logical draw_label       ! Draw the label? Default = True.
  logical draw_numbers     ! Draw the numbering? Default = True.
end type

```

## Chapter 22

# Helper Routines

This chapter gives an overview of various computational helper routines.

### 22.1 Nonlinear Optimization

Nonlinear optimization is the process of finding a minimum (or maximum) of a nonlinear function (the "merit" function). Nonlinear optimization is frequently used for lattice design or matching of data to a model. For more information on this see the *Tao* manual.

In terms of routines for implementing nonlinear optimization the Numerical Recipes library (§11.1 that is distributed along with *Bmad* contains several. In particular, the routine `super_mrqmin` which implements the Levenberg–Marquardt is an excellent routine for finding local minimum when the merit function can be expressed as the sum of quadratic terms. Another routine, `frprmn`, which is an implementation of the Fletcher–Reeves algorithm, is also good at finding local minimum and has the advantage that as input it does not need a derivative matrix as does Levenberg–Marquardt. The disadvantage of Fletcher–Reeves is that it is slower than Levenberg–Marquardt.

A second implementation of Levenberg–Marquardt available with *Bmad* is `opti_lmdif` which is Fortran90 version of the popular `lmdif` routine. Also available is `opti_de` which implements the Differential Evolution algorithm of Storn and Price[23]. This routine is good for finding global minima but can be slow.

Another routine that should be mentioned is the `amoeba` routine from Numerical Recipes that implements the downhill simplex method of Neider and Mead. This routine is robust but slow but is easily parallelized so it is a good routine for parallel processing.

### 22.2 Matrix Manipulation

There are a number of *Bmad* routines for matrix manipulation as listed in §23.17. In fact, Fortran90 has a number of intrinsic matrix routines as well but this is outside the scope of this manual. The following example shows some of the *Bmad* matrix routines

```
real(rp) mat(6,6), mat_inv(6,6)
call mat_make_unit (mat)      ! make a unit matrix
call mat_inv (mat, mat_inv) ! Compute the inverse matrix.
```



## Chapter 23

# Bmad Library Subroutine List

Below are a list of *Bmad* and *sim\_utils* routines sorted by their functionality. Use the `getf` and `listf` (§11.2) scripts for more information on individual routines. This list includes low level routines that are not generally used in writing code for a program but may be useful in certain unique situations. Excluded from the list are very low level routines that are solely meant for *Bmad* internal use.

<i>Routine Type</i>	<i>Section</i>
Beam: Low Level Routines	<a href="#">23.1</a>
Beam: Tracking and Manipulation	<a href="#">23.2</a>
Branch Handling	<a href="#">23.3</a>
C++ Interface	<a href="#">23.4</a>
Coherent Synchrotron Radiation (CSR)	<a href="#">23.5</a>
Collective Effects	<a href="#">23.6</a>
Electro-Magnetic Fields	<a href="#">23.7</a>
General Helper Routines	<a href="#">23.8</a>
Inter-Beam Scattering (IBS)	<a href="#">23.9</a>
Lattice: Element Informational	<a href="#">23.10</a>
Lattice: Element Manipulation	<a href="#">23.11</a>
Lattice: Geometry	<a href="#">23.12</a>
Lattice: Low Level Stuff	<a href="#">23.13</a>
Lattice: Manipulation	<a href="#">23.14</a>
Lattice: Miscellaneous	<a href="#">23.15</a>
Lattice: Reading and Writing Files	<a href="#">23.16</a>
Matrices	<a href="#">23.17</a>
Matrix: Low Level Routines	<a href="#">23.18</a>
Measurement Simulation Routines	<a href="#">23.19</a>
Multipass	<a href="#">23.20</a>
Multipoles	<a href="#">23.21</a>
Optimizers (Nonlinear)	<a href="#">23.22</a>
Overload Equal Sign	<a href="#">23.23</a>
Particle Coordinate Stuff	<a href="#">23.24</a>
Photon Routines	<a href="#">23.25</a>
PTC Interface	<a href="#">23.26</a>
Quick Plot	<a href="#">23.27</a>
Spin	<a href="#">23.28</a>
Transfer Maps: Routines Called by make_mat6	<a href="#">23.29</a>
Transfer Maps: Taylor Maps	<a href="#">23.30</a>
Tracking and Closed Orbit	<a href="#">23.31</a>
Tracking: Low Level Routines	<a href="#">23.32</a>
Tracking: Macroparticle	<a href="#">23.33</a>
Tracking: Mad Routines	<a href="#">23.34</a>
Tracking: Routines Called by track1	<a href="#">23.31</a>
Twiss and Other Calculations	<a href="#">23.36</a>
Twiss: 6-Dimensional	<a href="#">23.37</a>
Wake Fields	<a href="#">23.38</a>
Deprecated	<a href="#">23.39</a>

## 23.1 Beam: Low Level Routines

The following helper routines are generally not useful for general use.

**add\_sr\_long\_wake (ele, bunch, num\_in\_front, follower)**

Adds the longitudinal wake for all particles in front of the follower.



**find\_bunch\_sigma\_matrix (particle, ave, sigma)**

Routine to find the sigma matrix elements of a particle distribution.

**init\_spin\_distribution (beam\_init, bunch)**

Initializes a spin distribution according to init\_beam%spin

**order\_particles\_in\_z (bunch)**

Subroutine to order the particles longitudinally The ordering uses the centroid of the particles:

**track1\_beam (beam\_start, lat, ix\_ele, beam\_end, err)**

Subroutine to track a beam of particles through a single element. Overloaded by track1\_beam.

**track1\_bunch (bunch\_start, lat, ix\_ele, bunch\_end, err)**

Subroutine to track a bunch of particles through an element.

**track1\_bunch\_hom (bunch\_start, ele, param, bunch\_end)**

Subroutine to track a bunch of particles through an element.

**track1\_particle (start, ele, param, end)**

Subroutine to track a particle through an element.

## 23.2 Beam: Tracking and Manipulation

See §17.8 for a discussion of using a collection of particles to simulate a bunch.

**angle\_to\_canonical\_coords (particle, energy0)**

Subroutine to convert particle coords from (x, x', y, y', z, E)

**bbl\_kick (x, y, r, kx, ky)**

Subroutine to compute the normalized kick due to the beam-beam interaction using the normalized position for input.

**calc\_bunch\_params (bunch, ele, params)**

Finds all bunch parameters defined in bunch\_params\_struct, both normal-mode and projected

**calc\_bunch\_params (bunch, ele, params, plane, slice\_center, slice\_spread)**

Finds all bunch parameters for a slice through the beam distribution.

**canonical\_to\_angle\_coords (particle, energy0)**

Subroutine to convert particle coords from (x, px, y, py, z, pz)

**init\_beam\_distribution (ele, beam\_init, beam)**

Subroutine to initialize a distribution of particles matched to the Twiss parameters, centroid position, and Energy - z correlation

**init\_bunch\_distribution (ele, param, beam\_init, bunch)**

Subroutine to initialize either a random or tail-weighted distribution of particles.

**ion\_kick(x, y, x\_kicker, y\_kicker, s\_kicker)**

subroutine to return the kick felt by an ion due to the passage of a bunch. Can also be used for beam-beam simulations.

**reallocate\_beam (beam, n\_bunch, n\_particle)**

Subroutine to reallocate memory within a beam\_struct.

**track1\_bunch\_custom** (**bunch\_start**, **lat**, **ix\_ele**, **bunch\_end**)

Dummy routine for custom bunch tracking.

**track\_beam** (**lat**, **beam**, **ix1**, **ix2**)

Subroutine to track a beam of particles from the end of lat%ele(ix1) Through to the end of lat%ele(ix2).

## 23.3 Branch Handling Routines

**allocate\_branch\_array** (**branch**, **upper\_bound**, **lat**)

Subroutine to allocate or re-allocate an branch array. The old information is saved.

**deallocate\_branch** (**branch**)

Subroutine to deallocate a branch array and everything in it.

**transfer\_branch** (**branch1**, **branch2**)

Subroutine to set branch2 = branch1. This is a plain transfer of information not using the over-loaded equal.

**transfer\_branches** (**branch1**, **branch2**)

Subroutine to set branch2 = branch1. This is a plain transfer of information not using the over-loaded equal.

## 23.4 C++ Interface

**amode\_to\_c** (**f\_amode**, **c\_amode**)

Subroutine to convert a Bmad amode\_struct to a C++ C\_amode.

**arr2mat** (**arr**, **n1**, **n2**) **result** (**mat**)

Function to take a an array and turn it into a matrix.

**bmad\_com\_to\_c** (**c\_bmad\_com**)

Subroutine to convert the Bmad bmad\_com\_struct common block to a C++ C\_bmad\_com.

**c\_logic** (**logic**) **result** (**c\_log**)

Function to convert from a Fortran logical to a C logical.

**c\_str** (**str**) **result** (**c\_string**)

Function to append a null (0) character at the end of a string (trimmed of trailing blanks) so it will look like a C character array.

**control\_to\_c** (**f\_control**, **c\_control**)

Subroutine to convert a Bmad control\_struct to a C++ C\_control.

**coord\_to\_c** (**f\_coord**, **c\_coord**)

Subroutine to convert a Bmad coord\_struct to a C++ C\_coord.

**ele\_to\_c** (**f\_ele**, **c\_ele**)

Subroutine to convert a Bmad ele\_struct to a C++ C\_ele.

**em\_field\_to\_c** (**f\_em\_field**, **c\_em\_field**)

Subroutine to convert a Bmad em\_field\_struct to a C++ C\_em\_field.

**f\_logic (logic) result (f\_log)**

Function to convert from a Fortran logical to a C logical.

**floor\_position\_to\_c (f\_floor\_position, c\_floor\_position)**

Subroutine to convert a Bmad floor\_position\_struct to a C++ C\_floor\_position.

**linac\_mode\_to\_c (f\_linac\_mode, c\_linac\_mode)**

Subroutine to convert a Bmad linac\_mode\_struct to a C++ C\_linac\_mode.

**lr\_wake\_to\_c (f\_lr\_wake, c\_lr\_wake)**

Subroutine to convert a Bmad lr\_wake\_struct to a C++ C\_lr\_wake.

**mat2arr (mat) result (arr)**

Function to take a matrix and turn it into an array.

**modes\_to\_c (f\_modes, c\_modes)**

Subroutine to convert a Bmad modes\_struct to a C++ C\_modes.

**mode\_info\_to\_c (f\_mode\_info, c\_mode\_info)**

Subroutine to convert a Bmad mode\_info\_struct to a C++ C\_mode\_info.

**param\_to\_c (f\_param, c\_param)**

Subroutine to convert a Bmad param\_struct to a C++ C\_param.

**lat\_to\_c (f\_lat, c\_lat)**

Subroutine to convert a Bmad lat\_struct to a C++ C\_lat.

**sr\_table\_wake\_to\_c (f\_sr\_table\_wake, c\_sr\_wake)**

Subroutine to convert a Bmad sr\_table\_wake\_struct to a C++ C\_sr\_table\_wake.

**sr\_mode\_wake\_to\_c (f\_sr\_mode\_wake, c\_sr\_wake)**

Subroutine to convert a Bmad sr\_mode\_wake\_struct to a C++ C\_sr\_mode\_wake.

**twiss\_to\_c (f\_twiss, c\_twiss)**

Subroutine to convert a Bmad twiss\_struct to a C++ C\_twiss.

**taylor\_term\_to\_c (f\_taylor\_term, c\_taylor\_term)**

Subroutine to convert a Bmad taylor\_term\_struct to a C++ C\_taylor\_term.

**taylor\_to\_c (f\_taylor, c\_taylor)**

Subroutine to convert a Bmad taylor\_struct to a C++ C\_taylor.

**wake\_to\_c (f\_wake, c\_wake)**

Subroutine to convert a Bmad wake\_struct to a C++ C\_wake.

**wig\_term\_to\_c (f\_wig\_term, c\_wig\_term)**

Subroutine to convert a Bmad wig\_term\_struct to a C++ C\_wig\_term.

**xy\_disp\_to\_c (f\_xy\_disp, c\_xy\_disp)**

Subroutine to convert a Bmad xy\_disp\_struct to a C++ C\_xy\_disp.

## 23.5 Coherent Synchrotron Radiation (CSR)

**csr\_bin\_particles (particle, bin)**

Routine to bin the particles longitudinally in s.

**csr\_bin\_kicks (lat, ix\_ele, s\_travel, bin)**

Routine to cache intermediate values needed for the csr calculations.

**csr\_kick\_calc (bin, particle)**

Routine to calculate the longitudinal coherent synchrotron radiation kick.

**i\_csr (z, d, val, bin) result (i\_this)**

Routine to calculate the CSR kick integral.

**kick\_csr\_lsc (kick1, k\_factor, bin)**

Routine to calculate the CSR kick integral.

**z\_calc\_csr (d, val, bin, dz\_dd) result (z\_this)**

Routine to calculate the distance between the source particle and the kicked particle.

**d\_calc\_csr (dz\_particles, val, bin) result (d\_this)**

Routine to calculate the distance between source and kick points.

## 23.6 Collective Effects

**setup\_trans\_space\_charge\_calc (calc\_on, lattice, mode, closed\_orb)**

Subroutine to initialize constants needed by the transverse space charge tracking routine track1\_space\_charge. This routine must be called if

**touschek\_lifetime (mode, lifetime, lat, orb)**

Subroutine to calculate the Touschek lifetime for a lat.

**ibs\_rates (lat, mode, rates, formula)**

Subroutine to calculate the IBS rates for a lat.

**ibs\_equilibrium(lat, inmode, ibsmode, formula, coupling)**

Subroutine to calculate the equilibrium mode of a lat due to IBS effects by iterating over derivatives of the equilibrium equations.

**ibsequilibrium2(lat, inmode, ibsmode, formula, ratio, initial\_blow\_up)**

Subroutine to calculate the equilibrium mode of a lat due to IBS effects by iterating over the equilibrium equations.

## 23.7 Electro-Magnetic Fields

**em\_field\_calc (ele, param, s\_pos, here, local\_ref\_frame, field, calc\_dfield)**

Subroutine to calculate the E and B fields for an element.

**em\_field\_custom(ele, param, s, here, local\_ref\_frame, field, calc\_dfield)**

Custom routine for calculating fields.

**em\_field\_kick (ele, param, s, r, local\_ref\_frame, dr\_ds, dkick)**

Subroutine to essentially calculate the kick felt by a particle in a element.

## 23.8 General Helper Routines

### 23.8.1 General Helper: File, System, and IO Routines

#### **append\_subdirectory (dir, sub\_dir, dir\_out)**

Subroutine to combine a directory specification with a subdirectory specification to form a complete directory

#### **cesr\_iargc ()**

Platform independent function to return the number of command line arguments. Use this with cesr\_getarg.

#### **cesr\_getarg (i\_arg, arg)**

Platform independent function to return the i'th command line argument. Use this with cesr\_iargc.

#### **dir\_close ()**

Routine to close a directory that was opened with dir\_open. Also see dir\_read.

#### **dir\_open (dir\_name) result (opened)**

Routine to open a directory to obtain a list of its files. Use this routine with dir\_read and dir\_close.

#### **dir\_read (file\_name) result (valid)**

Routine to get the names of the files in a directory. Use this routine with dir\_open and dir\_close.

#### **file\_suffixer (in\_file\_name, out\_file\_name, suffix, add\_switch)**

Routine to add/replace a suffix to a file name.

#### **get\_tty\_char (this\_char, wait, flush)**

Subroutine for getting a single character from the terminal. Also see: get\_a\_char

#### **get\_a\_char (this\_char, wait, ignore\_this)**

Subroutine for getting a single character from the terminal. Also see: get\_tty\_char

#### **get\_file\_time\_stamp (file, time\_stamp)**

Routine to get the "last modified" time stamp for a file.

#### **lunget()**

Function to return a free file unit number to be used with an open statement.

#### **milli\_sleep (milli\_sec)**

Routine to pause the program for a given number of milli-seconds.

#### **output\_direct (file\_unit, do\_print, min\_level, max\_level)**

Subroutine to set where the output goes when out\_io is called. Output may be sent to the terminal screen, written to a file, or both.

#### **out\_io (...)**

Subroutine to print to the terminal for command line type programs. The idea is that for programs with a gui this routine can be easily replaced with another routine.

#### **read\_a\_line (prompt, line\_out, trim\_prompt)**

Subroutine to read a line of input from the terminal. The line is also add to the history buffer so that the up-arrow

#### **skip\_header (unit, error\_flag)**

Subroutine to find the first line of data in a file.

**splitfilename(filename, path, basename, is\_relative) result (ix\_char)**

Routine to take filename and splits it into its constituent parts, the directory path and the base file name.

**system\_command (line)**

Routine to execute an operating system command from within the program.

**type\_this\_file (filename)**

Subroutine to type out a file to the screen.

## 23.8.2 General Helper: Math (Except Matrix) Routines

**abs\_sort (array, index, n)**

Subroutine to sort by absolute value.

**complex\_error\_function (wr, wi, zr, zi)**

This routine evaluates the function  $w(z)$  in the first quadrant of the complex plane.

**linear\_fit (x, y, n\_data, a, b, sig\_a, sig\_b)**

Subroutine to fit to  $y = A + Bx$

**modulo2 (x, amp)**

Function to return  $y = x + 2 * n * \text{amp}$ ,  $n$  is an integer, such that  $y$  is in the interval  $[-\text{amp}, \text{amp}]$ .

**ran\_engine (set, get)**

Subroutine to set what random number generator algorithm is used. If this routine is never called then `pseudo_random$` is used.

**ran\_gauss (harvest)**

Subroutine to return a Gaussian distributed random number with unit sigma.

**ran\_gauss\_converter (set, get, sigma\_cut)**

Subroutine to set what conversion routine is used for converting uniformly distributed random numbers to Gaussian distributed random numbers.

**ran\_seed\_put (seed)**

Subroutine to seed the random number generator.

**ran\_seed\_get (seed)**

Subroutine to return the seed used for the random number generator.

**ran\_uniform (harvest)**

Subroutine to return a random number uniformly distributed in the interval  $[0, 1]$ . This routine uses the same algorithm as `ran` from

**spline\_akima (spline, stat)**

Given a set of  $(x,y)$  points we want to interpolate between the points. This subroutine computes the semi-hermite cubic spline developed by akima

**spline\_evaluate (spline, x, ok, y, dy)**

Subroutine to evaluate a spline at a set of points.

**super\_ludcmp (a, indx, d, err)**

This routine is essentially `ludcmp` from Numerical Recipes with the added feature that an error flag is set instead of bombing the program when there is a problem.

### 23.8.3 General Helper: Matrix Routines

**mat\_eigen (mat, eval\_r, eval\_i, evec\_r, evec\_i, error)**

Routine for determining the eigen vectors and eigen values of a matrix.

**mat\_inverse (mat, mat\_inv)**

Subroutine to take the inverse of a square matrix.

**mat\_make\_unit (mat)**

routine to create a unit matrix.

**mat\_rotation (mat, angle, bet\_1, bet\_2, alph\_1, alph\_2)**

Subroutine to construct a 2x2 rotation matrix for translation from point 1 to point 2.

**mat\_symplectify (mat\_in, mat\_symp)**

Subroutine to form a symplectic matrix that is approximately equal to the input matrix.

**mat\_symp\_error (mat) result (error)**

Routine to check the symplecticity of a square matrix

**mat\_symp\_conj (mat1, mat2)**

Subroutine to take the symplectic conjugate of a square matrix.

**mat\_symp\_decouple (t0, tol, stat, u, v, ubar, vbar, g, twiss1, twiss2, type\_out)**

Subroutine to find the symplectic eigen-modes of the one turn 4x4 coupled transfer matrix T0.

**mat\_type (mat, nunit, header)**

Subroutine to output matrices to the terminal or to a file

### 23.8.4 General Helper: Misc Routines

**date\_and\_time\_stamp (string, numeric\_month)**

Subroutine to return the current date and time in a character string.

**err\_exit()**

Subroutine to first show the stack call list before exiting. This routine is typically used when a program detects an error condition.

**integer\_option (integer\_default, opt\_integer)**

Function to return True or False depending upon the state of an optional integer.

**logic\_option (logic\_default, opt\_logic)**

Function to return True or False depending upon the state of an optional logical.

**re\_allocate (ptr\_to\_array, n)**

Function to reallocate a pointer to an array of strings, integers, reals, or logicals.

**re\_associate (array, n)**

Function to reassociate an allocatable array of strings, integers, reals, or logicals.

**real\_option (real\_default, opt\_real)**

Function to return True or False depending upon the state of an optional real.

**string\_option (string\_default, opt\_string)**

Subroutine to return True or False depending upon the state of an optional string.

### 23.8.5 General Helper: String Manipulation Routines

**downcase\_string (string)**

Routine to convert a string to lowercase:

**indexx\_char (arr, index)**

Subroutine to sort a character array. This subroutine is used to overload the generic name indexx.

**index\_nocase (string, match\_str) result (indx)**

Function to look for a sub-string of string that matches match\_str. This routine is similar to the fortran INDEX function

**is\_integer (string)**

Function to tell if the first word in a string is a valid integer.

**is\_logical (string, ignore) result (good)**

Function to test if a string represents a logical. Accepted possibilities are (individual characters can be either case):

**is\_real (string, ignore) result (good)**

Function to test if a string represents a real number.

**match\_reg (str, pat)**

Function for matching with regular expressions. Note: strings are trimmed before comparison.

**match\_wild (string, template) result (this\_match)**

Function to do wild card matches. Note: trailing blanks will be discarded before any matching is done.

**match\_word (string, names, ix, exact\_case, matched\_name)**

Subroutine to match the first word in a string against a list of names. Abbreviations are accepted.

**on\_off\_logic (logic) result (name)**

Function to return the string "ON" or "OFF".

**str\_match\_wild(str, pat) result (a\_match)**

Function to match a character string against a regular expression pattern. This is a replacement for the VMS function str\$match\_wild.

**string\_to\_int (line, default, value, err\_flag)**

Subroutine to convert a string to an integer.

**string\_to\_real (line, default, value, err\_flag)**

Subroutine to convert a string to an real.

**string\_trim(in\_string, out\_string, word\_len)**

Subroutine to trim a string of leading blanks and/or tabs and also to return the length of the first word.

**string\_trim2 (in\_str, delimiters, out\_str, ix\_word, delim, ix\_next)**

Subroutine to trim a string of leading delimiters and also to return the length of the first word.

**str\_downcase (destination, source)**

Subroutine to convert a string to down case.

**str\_substitute (string, str\_match, str\_replace, do\_trim)**

Routine to substitute all instances of one sub-string for another in a string



**upcase\_string (string)**

Routine to convert a string to uppercase:

## 23.9 Inter-Beam Scattering (IBS)

**ibs\_lifetime(lat, mode, lifetime, formula)**

This module computes the beam lifetime due to the diffusion process according to equation 12

**bjmt(lat, mode, rates)**

This is a private subroutine. To access this subroutine, call `ibs_rates`.

**bane(lat, mode, rates)**

This is a private subroutine. To access this subroutine, call `ibs_rates`.

**cimp(lat, mode, rates)**

This is a private subroutine. To access this subroutine, call `ibs_rates`.

**g(u)**

This is an 13-degree piecewise polynomial interpolation of the integral for the CIMP ibs formulation.

**mtto(lat, mode, rates)**

NOTE: The Mtingwa-Tollerstrup formula gives different from the other formulations in this module.

## 23.10 Lattice: Informational

**attribute\_free (ix\_ele, attrib\_name, lat, err\_print\_flag, except\_overlay) result (free)**  
**attribute\_free (ele, attrib\_name, lat, err\_print\_flag, except\_overlay) result (free)**  
**attribute\_free (ix\_ele, ix\_branch, attrib\_name, lat, err\_print\_flag, except\_overlay)**  
**result (free)**

Overloaded function to check if an attribute is free to vary.

**attribute\_index (key, name)**

Function to return the index of an attribute for a given element type and the name of the attribute

**attribute\_name (key, index)**

Function to return the name of an attribute for a particular type of element.

**attribute\_type (attrib\_name) result (attrib\_type)**

Routine to return the type (logical, integer, real, or named) of an attribute.

**attribute\_value\_name (attrib\_name, attrib\_value, ele, is\_default) result (val\_name)**

Routine to return the name corresponding to the value of a given attribute.

**check\_lat\_controls (lat, exit\_on\_error)**

Subroutine to check if the control links in a lat structure are valid.

**ele\_at\_s (lat, s, ix\_ele)**

Subroutine to return the index of the element at position s.

**ele\_loc\_to\_string (ele, show\_branch0) result (str)**

Routine to encode an element's location into a string.

**ele\_to\_lat\_loc (ele) result (ele\_loc)**

Function to return an `lat_ele_loc_struct` identifying where an element is in the lattice.

**equivalent\_taylor\_attributes (ele1, ele2) result (equiv)**

Subroutine to see if two elements are equivalent in terms of their attributes so that their Taylor Maps, if they existed, would be the same.

**find\_element\_ends (lat, ix\_ele, ix\_start, ix\_end)**

Subroutine to find the end points of an element.

**get\_element\_slave\_list (lat, ix\_lord, slave\_list, n\_slave)**

Subroutine to get the list of slaves for an element.

**key\_name (key\_index)**

Translate an element key index (EG: quadrupole\$, etc.) to a character string.

**key\_name\_to\_key\_index (key\_str, abbrev\_allowed) result (key\_index)**

Function to convert a character string (eg: "drift") to an index (eg: drift\$).

**lat\_ele\_locator (loc\_str, lat, eles, n\_loc, err)**

Routine to locate all the elements in a lattice that corresponds to `loc_str`.

**n\_attr\_string\_max\_len () result (max\_len)**

Routine to return the the maximum number of characters in any attribute name known to `bmad`.

**pointer\_to\_indexed\_attribute (ele, ix\_attr, do\_allocation,  
ptr\_attr, err\_flag, err\_print\_flag)**

Returns a pointer to an attribute of an element `ele` with attribute index `ix_attr`.

**pointer\_to\_lord (lat, slave, ix\_lord, ix\_control) result (lord\_ptr)**

Function to point to a lord of a slave.

**pointer\_to\_multipass\_lord (ele, lat, ix\_pass, super\_lord) result (multi\_lord)**

Routine to find the multipass lord of a lattice element. A `multi_lord` will be found for:

**pointer\_to\_slave (lat, lord, ix\_slave, ix\_ctrl) result (slave\_ptr)**

Function to point to a slave of a lord.

**type\_ele (ele, type\_zero\_attr, type\_mat6,  
type\_twiss, type\_control, type\_wake, type\_floor\_coords)**

Subroutine to print the contents of an element at the terminal.

**type2\_ele (ele, lines, n\_lines, type\_zero\_attr, type\_mat6,  
type\_twiss, type\_control, type\_wake, type\_floor\_coords)**

Like `type_ele` but the output is stored in a string array.

**type\_twiss (ele, frequency\_units)**

Subroutine to type out the Twiss parameters from an element.

**type2\_twiss (ele, frequency\_units, lines, n\_lines)**

Like `type_twiss` but the output is stored in a string array.

## 23.11 Lattice: Element Manipulation

These routine are for adding elements, moving elements, etc.

**add\_lattice\_control\_structs (lat, ix\_ele)**

Subroutine to adjust the control structure of a lat so that extra control elements can be added.

**add\_superimpose (lat, super\_ele\_in, ix\_branch, super\_ele\_out)**

Subroutine to make a superimposed element.

**attribute\_bookkeeper (ele, param)**

Subroutine to make sure the attributes of an element are self-consistent.

**changed\_attribute\_bookkeeper (lat, a\_ptr)**

Subroutine to do bookkeeping when a particular attribute has been altered.

**create\_group (lat, ix\_ele, contrl)**

Subroutine to create a group control element.

**create\_girder (lat, ix\_girder, ix\_slave)**

Subroutine to add the controller information to slave elements of an girder\_lord.

**create\_overlay (lat, ix\_overlay, attrib\_name, , contrl)**

Subroutine to add the controller information to slave elements of an overlay\_lord.

**create\_wiggler\_model (wiggler, lat)**

Routine to create series of bend and drift elements to serve as a model for a wiggler. This routine uses the mrqmin nonlinear optimizer to vary the parameters in the wiggler

**insert\_element (lat, insert\_ele, insert\_index)**

Subroutine to Insert a new element into the tracking part of the lat structure.

**makeup\_super\_slave1 (slave, lord, offset, param, at\_entrance\_end, at\_exit\_end)**

Routine to transfer the %value, %wig\_term, and %wake%lr information from a superposition lord to a slave when the slave has only one lord.

**make\_hybrid\_lat (lat\_in, use\_ele, remove\_markers, lat\_out, ix\_out)**

Subroutine to concatenate together elements to make a hybrid lat

**new\_control (lat, ix\_ele)**

Subroutine to create a new control element.

**pointer\_to\_attribute (ele, attrib\_name, do\_allocation,  
ptr\_attrib, ix\_attrib, err\_flag, err\_print\_flag)**

Returns a pointer to an attribute of an element with name attrib\_name.

**pointers\_to\_attribute (lat, ele\_name, attrib\_name, do\_allocation,  
ptr\_array, err\_flag, err\_print\_flag, ix\_eles, ix\_attrib)**

Returns an array of pointers to an attribute with name attrib\_name within elements with name ele\_name.

**pointer\_to\_ele (lat, ix\_ele, ix\_branch) result (ele\_ptr)**

**pointer\_to\_ele (lat, ele\_loc\_id) result (ele\_ptr)**

Subroutine to point to a given element.

**remove\_eles\_from\_lat (lat)**

Subroutine to remove an elements from the lattice.

**split\_lat (lat, s\_split, ix\_branch, ix\_split, split\_done, add\_suffix, check\_controls)**

Subroutine to split a lat at a point.

**update\_hybrid\_list (lat, n\_in, use\_ele)**

Subroutine used to specify a list of element that should not be hybridized by `make_hybrid_lat`.

## 23.12 Lattice: Geometry

**ele\_geometry (ele0, ele, param)**

Subroutine to calculate the physical (floor) placement of an element given the placement of the preceding element. This is the same as the MAD convention.

**floor\_angles\_to\_w\_mat (theta, phi, psi, w\_mat)**

Routine to construct the W matrix that specifies the orientation of an element in the global "floor" coordinates. See the Bmad manual for more details.

**floor\_w\_mat\_to\_angles (w\_mat, theta0, theta, phi, psi)**

Routine to construct the angles that define the orientation of an element in the global "floor" coordinates from the W matrix. See the Bmad manual for more details.

**init\_floor (floor)**

Routine to initialize a floor\_position\_struct to zero.

**lat\_geometry (lat)**

Subroutine to calculate the physical placement of all the elements in a lattice. That is, the physical machine layout on the floor.

**s\_calc (lat)**

Subroutine to calculate the longitudinal distance S for the elements in a lat.

**theta\_floor (s, lat, theta\_base) result (theta\_fl)**

Returns the angle of the reference coordinate system in the global reference system.

## 23.13 Lattice: Low Level Stuff

**adjust\_super\_lord\_s\_position (lat, ix\_lord)**

Subroutine to adjust the positions of the slaves of a super\_lord due to changes in the lord's s\_offset.

**bracket\_index (s\_arr, i\_min, i\_max, s, ix)**

Subroutine to find the index ix so that  $s(ix) \leq s < s(ix+1)$ . If  $s < s(1)$  then  $ix = 0$

**check\_controller\_controls (ctrl, name, err)**

Routine to check for problems when setting up group or overlay controllers.

**deallocate\_ele\_pointers (ele)**

Subroutine to deallocate the pointers in an element.

**dispersion\_to\_orbit (ele, disp\_orb)**

Subroutine to make an orbit vector proportional to the dispersion.

**makeup\_super\_slave (lat, ix\_slave)**

Subroutine to calculate the attributes of overlay slave elements.

**orbit\_to\_dispersion (orb\_diff, ele)**

Subroutine to take an orbit vector difference and calculate the dispersion.

**re\_allocate\_eles (eles, n, save, exact)**

Routine to allocate an array of ele\_pointer\_structs.

**twiss1\_propagate (twiss1, mat2, length, twiss2)**

Subroutine to propagate the twiss parameters of a single mode.

## 23.14 Lattice: Manipulation

**allocate\_ele\_array (ele, upper\_bound)**

Subroutine to allocate or re-allocate an element array.

**allocate\_lat\_ele\_array (lat, upper\_bound, ix\_branch)**

Subroutine to allocate or re-allocate an element array.

**control\_bookkeeper (lat, ix\_ele)**

Subroutine to calculate the combined strength of the attributes for controlled elements.

**deallocate\_ele\_array\_pointers (eles)**

Routine to deallocate the pointers of all the elements in an element array and the array itself.

**deallocate\_lat\_pointers (lat)**

Subroutine to deallocate the pointers in a lat.

**init\_ele (ele, ix\_ele, ix\_branch)**

Subroutine to initialize an element.

**init\_lat (lat, n)**

Subroutine to initialize a Bmad lat.

**lattice\_bookkeeper (lat)**

Subroutine to do bookkeeping for the entire lattice.

**lat\_reverse (lat\_in, lat\_rev)**

Subroutine to construct a lat structure with the elements in reversed order. This may be used for backward tracking through the lat.

**reallocate\_coord (coord, n\_coord)**

Subroutine to reallocate an allocatable coord\_struct array to at least: coord(0:n\_coord).

**reverse\_ele (ele)**

Subroutine to "reverse" an element for backward tracking.

**set\_design\_linear (lat)**

Subroutine to set only those elements on that constitute the "design" lattice. That is, only quadrupoles, bends and wigglers will be set on.

**set\_on\_off (key, lat, switch, orb)**

Subroutine to turn on or off a set of elements (quadrupoles, RF cavities, etc.) in a lat.

**transfer\_ele (ele1, ele2)**

Subroutine to set ele2 = ele1. This is a plain transfer of information not using the overloaded equal.

**transfer\_eles (ele1, ele2)**

Subroutine to set  $\text{ele2}(:) = \text{ele1}(:)$ . This is a plain transfer of information not using the overloaded equal.

**transfer\_ele\_taylor (ele\_in, ele\_out, taylor\_order)**

Subroutine to transfer a Taylor map from one element to another.

**transfer\_lat (lat1, lat2)**

Subroutine to set  $\text{lat2} = \text{lat1}$ . This is a plain transfer of information not using the overloaded equal.

**transfer\_lat\_parameters (lat\_in, lat\_out)**

Subroutine to transfer the lat parameters (such as  $\text{lat}\%name$ ,  $\text{lat}\%param$ , etc.) from one lat to another.

**transfer\_lat\_taylors (lat\_in, lat\_out, type\_out, transfered\_all)**

Subroutine to transfer the taylor maps from the elements of one lat to the elements of another.

**zero\_ele\_offsets (ele)**

Subroutine to zero the offsets, pitches and tilt of an element.

## 23.15 Lattice: Miscellaneous

**cross\_product (vec1, vec2)**

Returns the cross product of  $\text{vec1} \times \text{vec2}$

**c\_multi (n, m)**

Subroutine to compute multipole factors:  $c\_multi(n, m) = +/- ("n \text{ choose } m")/n!$

**compute\_reference\_energy (lat)**

Subroutine to compute the reference energy for each element in a lattice.

**convert\_total\_energy\_to (E\_tot, particle, gamma, kinetic, beta, pc, brho)**

Subroutine to calculate the momentum, etc. from a particle's total energy.

**convert\_pc\_to (pc, particle, E\_tot, gamma, kinetic, beta, brho)**

Subroutine to calculate the energy, etc. from a particle's momentum.

**field\_interpolate\_3d (position, field\_mesh, deltas)**

Function to interpolate a 3d field.

**name\_to\_list (lat, ele\_names, use\_ele)**

Subroutine to make a list of the elements in a lat whose name matches the names in the  $\text{ele\_names}$  list.

**order\_super\_lord\_slaves (lat, ix\_lord)**

Subroutine to make the slave elements of a  $\text{super\_lord}$  in order.

**release\_rad\_int\_cache (ix\_cache)**

Subroutine to release the memory associated with caching wiggler values.

**wiggler\_vec\_potential (ele, energy, here, vec\_pot)**

Subroutine to calculate the normalized vector potential at a point for a wiggler.

## 23.16 Reading and Writing Lattice Files

**aml\_parser** (lat\_file, lat, make\_mats6, digested\_read\_ok, use\_line)

Subroutine to parse an AML input file and put the information in a lat\_struct.

**bmad\_parser** (in\_file, lat, make\_mats6, digested\_read\_ok, use\_line)

Subroutine to parse (read in) a Bmad input file.

**bmad\_parser2** (in\_file, lat, orbit, make\_mats6)

Subroutine to parse (read in) a Bmad input file to modify an existing lattice.

**bmad\_to\_mad\_or\_xsif** (out\_type, file\_name, lat, use\_matrix\_model,  
ix\_start, ix\_end, converted\_lat, err)

Subroutine to write a mad or xsif lattice file using the information in a lat\_struct.

**combine\_consecutive\_elements** (lat)

Routine to combine consecutive elements in the lattice that have the same name. This allows simplification, for example, of lattices where elements have been split to compute the beta function at the center.

**create\_sol\_quad\_model** (sol\_quad, lat)

Routine to create series of solenoid and quadrupole elements to serve as a replacement model for a sol\_quad element.

**create\_unique\_ele\_names** (lat, key, suffix)

Routine to give elements in a lattice unique names.

**read\_digested\_bmad\_file** (in\_file\_name, lat, version)

Subroutine to read in a digested file.

**write\_bmad\_lattice\_file** (lattice\_name, lat)

Subroutine to write a Bmad lattice file using the information in a lat\_struct.

**write\_digested\_bmad\_file** (digested\_name, lat, n\_files, file\_names)

Subroutine to write a digested file.

**xsif\_parser** (xsif\_file, lat, make\_mats6, use\_line)

Subroutine to parse an XSIF (extended standard input format) lattice file.

## 23.17 Matrices

**c\_to\_cbar** (ele, cbar\_mat)

Subroutine to compute Cbar from the C matrix and the Twiss parameters.

**cbar\_to\_c** (cbar\_mat, ele)

Subroutine to compute C coupling matrix from the Cbar matrix and the Twiss parameters.

**clear\_lat\_1turn\_mats** (lat)

Clear the 1-turn matrices in the lat structure.

**determinant** (mat) **result** (det)

Routine to take the determinant of a square matrix This routine is adapted from Numerical Recipes.

**do\_mode\_flip** (ele, ele\_flip)

Subroutine to mode flip the Twiss parameters of an element

**make\_g2\_mats (twiss, g\_mat, g\_inv\_mat)**

Subroutine to make the matrices needed to go from normal mode coords to coordinates with the beta function removed.

**make\_g\_mats (ele, g\_mat, g\_inv\_mat)**

Subroutine to make the matrices needed to go from normal mode coords to coordinates with the beta function removed.

**make\_mat6 (ele, param, c0, c1)**

Subroutine to make the 6x6 transfer matrix for an element.

**make\_v\_mats (ele, v\_mat, v\_inv\_mat)**

Subroutine to make the matrices needed to go from normal mode coords to X-Y coords and vice versa.

**mat6\_calc\_at\_s (lat, mat6, vec0, s1, s2, one\_turn, unit\_start)**

Subroutine to calculate the transfer map between longitudinal positions s1 to s2.

**mat6\_to\_taylor (mat6, vec0, bmad\_taylor)**

Subroutine to form a first order Taylor map from the 6x6 transfer matrix and the 0th order transfer vector.

**match\_ele\_to\_mat6 (ele, mat6, vec0)**

Subroutine to make the 6 x 6 transfer matrix from the twiss parameters.

**multi\_turn\_tracking\_to\_mat (track, i\_dim, mat1, track0, chi)**

Subroutine to analyze 1-turn tracking data to find the 1-turn transfer matrix and the closed orbit offset.

**transfer\_matrix\_calc (lat, rf\_on, mat6, ix1, ix2)**

Subroutine to calculate the transfer matrix between two elements. If ix1 and ix2 are not present the full 1-turn matrix is calculated.

**one\_turn\_mat\_at\_ele (ele, phi\_a, phi\_b, mat4)**

Subroutine to form the 4x4 1-turn coupled matrix with the reference point at the end of an element.

**lat\_make\_mat6 (lat, ix\_ele, coord, ix\_branch)**

Subroutine to make the 6x6 linear transfer matrix for an element

**taylor\_to\_mat6 (a\_taylor, c0, mat6, c1)**

Subroutine to calculate the linear (Jacobian) matrix about some trajectory from a Taylor map.

**transfer\_mat2\_from\_twiss (twiss1, twiss2, mat)**

Subroutine to make a 2 x 2 transfer matrix from the Twiss parameters at the end points.

**transfer\_mat\_from\_twiss (ele1, ele2, m)**

Subroutine to make a 6 x 6 transfer matrix from the twiss parameters at the beginning and end of the element.

**twiss\_from\_mat2 (mat, det, twiss, stat, tol, type\_out)**

Subroutine to extract the Twiss parameters from the one-turn 2x2 matrix

**twiss\_from\_mat6 (mat6, ele, stable, growth\_rate)**

Subroutine to extract the Twiss parameters from the one-turn 6x6 matrix

**twiss\_to\_1\_turn\_mat (twiss, phi, mat2)**

Subroutine to form the 2x2 1-turn transfer matrix from the Twiss parameters.



## 23.18 Matrix: Low Level Routines

Listed below are helper routines that are not meant for general use.

**drift\_mat6\_calc (mat6, length, start, end)**

Subroutine to calculate a drift transfer matrix with a possible kick.

**mat6\_dispersion (mat6, e\_vec)**

Subroutine to put the dispersion into ele%mat6 given the dispersion vector E\_VEC

**sol\_quad\_mat6\_calc (ks, k1, length, mat6, orb)**

Subroutine to calculate the transfer matrix for a combination solenoid/quadrupole element.

**tilt\_mat6 (mat6, tilt)**

Subroutine to transform a 6x6 transfer matrix to a new reference frame that is tilted in (x, Px, y, Py) with respect to the old reference frame.

## 23.19 Measurement Simulation Routines

Routines to simulate errors in orbit, dispersion, betatron phase, and coupling measurements

**check\_if\_ele\_is\_monitor (ele, err)**

Routine to check that the element is either an instrument, monitor, or marker. This routine is private and not meant for general use.

**compute\_bpm\_transformation\_numbers (ele)**

Routine to compute the numbers associated with the transformation between the actual orbit, phase, eta, and coupling, and what the measured values.

**to\_eta\_reading (eta, ele, axis, reading, err)**

Compute the measured dispersion reading given the true dispersion and the monitor offsets, noise, etc.

**to\_orbit\_reading (orb, ele, axis, reading, err)**

Calculate the measured reading on a bpm given the actual orbit and the BPM's offsets, noise, etc.

**to\_phase\_and\_coupling\_reading (ele, mon, err)**

Find the measured coupling values given the actual ones

## 23.20 Multipass

**multipass\_all\_info (lat, info)**

Subroutine to put multipass to a multipass\_all\_info\_struct structure.

**multipass\_chain (ele, lat, ix\_pass, n\_links, chain\_ele)**

Routine to return the chain of elements that represent the same physical element when there is multipass.

**pointer\_to\_multipass\_lord (ele, lat, ix\_pass, super\_lord) result (multi\_lord)**

Routine to find the multipass lord of a lattice element. A multi\_lord will be found for:

## 23.21 Multipoles

**ab\_multipole\_kick (a, b, n, coord, kx, ky)**

Subroutine to put in the kick due to an ab\_multipole.

**multipole\_kicks (knl, tilt, coord, ref\_orb\_offset)**

Subroutine to put in the kick due to a multipole.

**mexp (x, m) result (this\_exp)**

Returns  $x**m$  with  $0**0 = 0$ .

**multipole\_ab\_to\_kt (an, bn, knl, tn)**

Subroutine to convert ab type multipoles to kt (MAD standard) multipoles.

**multipole\_ele\_to\_ab (ele, particle, a, b, use\_ele\_tilt)**

Subroutine to put the scaled element multipole components (normal and skew) into 2 vectors.

**multipole\_ele\_to\_kt (ele, particle, knl, tilt, use\_ele\_tilt)**

Subroutine to put the scaled element multipole components (strength and tilt) into 2 vectors.

**multipole\_init(ele, zero)**

Subroutine to initialize the multipole arrays within an element.

**multipole\_kick (knl, tilt, n, coord)**

Subroutine to put in the kick due to a multipole.

**multipole\_kt\_to\_ab (knl, tn, an, bn)**

Subroutine to convert kt (MAD standard) multipoles to ab type multipoles.

## 23.22 Nonlinear Optimizers

**opti\_lmdif (vec, n, merit, eps) result(this\_opti)**

Function which tries to get the merit function(s) as close to zero as possible by changing the values in vec. Multiple merit functions can be used.

**initial\_lmdif()**

Subroutine that clears out previous saved values of the optimizer.

**suggest\_lmdif (xv,fv,eps,itermx,iend,reset\_flag)**

Reverse communication subroutine.

**super\_mrqrmin (y, weight, a, covar, alpha, chisq, funcs,  
alamda, status, maska)**

Routine to do non-linear optimizations. This routine is essentially mrqrmin from Numerical Recipes with some added features.

**opti\_de (v\_best, generations, population, merit\_func, v0, v\_del)**

Differential Evolution for Optimal Control Problems. This optimizer is based upon the work of Storn and Price.

## 23.23 Overloading the equal sign

These routines are overloaded by the equal sign so should not be called explicitly.

**mp\_beam\_equal\_mp\_beam (beam1, beam2)**

Subroutine that is used to set one macroparticle beam to another. This routine takes care of the pointers in beam1.

**branch\_equal\_branch (branch1, branch2)**

Subroutine that is used to set one branch equal to another.

**bunch\_equal\_bunch (bunch1, bunch2)**

Subroutine that is used to set one macroparticle bunch to another. This routine takes care of the pointers in bunch1.

**coord\_equal\_coord (coord1, coord2)**

Subroutine that is used to set one coord\_struct equal to another.

**ele\_equal\_ele (ele1, ele2)**

Subroutine that is used to set one element equal to another. This routine takes care of the pointers in ele1.

**ele\_vec\_equal\_ele\_vec (ele1, ele2)**

Subroutine that is used to set one element vector equal to another. This routine takes care of the pointers in ele1.

**mp\_beam\_equal\_mp\_beam (beam1, beam2)**

Subroutine to set one macroparticle beam equal to another taking care of pointers so that they don't all point to the same place.

**real\_8\_equal\_taylor (y8, bmad\_taylor)**

Subroutine to overload "=" in expressions `real_8 (PTC) = bmad_taylor`.

**lat\_equal\_lat (lat1, lat2)**

Subroutine that is used to set one lat equal to another. This routine takes care of the pointers in lat1.

**lat\_vec\_equal\_lat\_vec (lat1, lat2)**

Subroutine that is used to set one lat array equal to another. This routine takes care of the pointers in lat1(:).

**taylor\_equal\_real\_8 (bmad\_taylor, y8)**

Subroutine to overload "=" in expressions `bmad_taylor = real_8 (PTC)`

**universal\_equal\_universal (universal1, universal2)**

Subroutine that is used to set one PTC universal\_taylor structure equal to another.

## 23.24 Particle Coordinate Stuff

**convert\_coords (in\_type\_str, coord\_in, ele, out\_type\_str, coord\_out)**

Subroutine to convert between lab frame, normal mode, normalized normal mode, and action-angle coordinates.

**init\_coord (orb, vec)**  
 Subroutine to initialize a coord\_struct.

**type\_coord (coord)**  
 Subroutine to type out a coordinate.

## 23.25 Photon Routines

**photon\_init (g\_bend, gamma, orbit)**  
 Routine to initialize a photon

**photon\_vert\_angle\_init (e\_rel, gamma\_phi, r\_in)**  
 Routine to convert a "random" number in the interval [0,1] to a photon vertical emission angle for a simple bend.

**photon\_energy\_init (e\_rel, r\_in)**  
 Routine to convert a random number in the interval [0,1] to a photon energy.

## 23.26 Interface to PTC

**concat\_real\_8 (y1, y2, y3)**  
 Subroutine to concatenate two real\_8 taylor series.

**ele\_to\_fibre (ele, fiber, param, integ\_order, steps)**  
 Subroutine to convert a Bmad element to a PTC fibre element.

**map\_coef (y, i, j, k, l, style)**  
 Function to return the coefficient of the map y(:) up to 3rd order.

**kill\_gen\_field (gen\_field)**  
 Subroutine to kill a gen\_field.

**kind\_name (this\_kind)**  
 Function to return the name of a PTC kind.

**real\_8\_equal\_taylor (y8, bmad\_taylor)**  
 Subroutine to overload "=" in expressions real\_8 = bmad\_taylor

**real\_8\_to\_taylor (y8, bmad\_taylor, switch\_z)**  
 Subroutine to convert from a real\_8 taylor map in Etienne's PTC to a taylor map in Bmad.

**real\_8\_init (y, set\_taylor)**  
 Subroutine to allocate a PTC real\_8 variable.

**remove\_constant\_taylor (taylor\_in, taylor\_out, c0, remove\_higher\_order\_terms)**  
 Subroutine to remove the constant part of a taylor series.

**lat\_to\_layout (lat, ptc\_layout)**  
 Subroutine to create a PTC layout from a Bmad lat.

**set\_ptc (param, taylor\_order, integ\_order, n\_step, no\_cavity, exact\_calc)**  
 Subroutine to initialize PTC.

**set\_taylor\_order (order, override\_flag)**

Subroutine to set the taylor order.

**sort\_universal\_terms (ut\_in, ut\_sorted)**

Subroutine to sort the taylor terms from "lowest" to "highest".

**taylor\_equal\_real\_8 (bmad\_taylor, y8)**

Subroutine to overload "=" in expressions bmad\_taylor = y8

**taylor\_to\_real\_8 (bmad\_taylor, y8, switch\_z)**

Subroutine to convert from a taylor map in Bmad to a real\_8 taylor map in Etienne's PTC.

**type\_layout (lay)**

Subroutine to print the global information in a PTC layout.

**type\_map1 (y, type0, n\_dim, style)**

Subroutine to type the transfer map up to first order.

**type\_fibre (fb)**

Subroutine to print the global information in a fibre.

**type\_map (y)**

Subroutine to type the transfer maps of a real\_8 array.

**type\_real\_8\_taylors (y, switch\_z)**

Subroutine to type out the taylor series from a real\_8 array.

**taylor\_to\_genfield (bmad\_taylor, gen\_field, c0)**

Subroutine to construct a genfield (partially inverted map) from a taylor map.

**universal\_to\_bmad\_taylor (u\_taylor, bmad\_taylor, switch\_z)**

Subroutine to convert from a universal\_taylor map in Etienne's PTC to a taylor map in Bmad.

**vec\_bmad\_to\_ptc (vec\_bmad, vec\_ptc)**

Subroutine to convert from Bmad to PTC coordinates.

**vec\_ptc\_to\_bmad (vec\_ptc, vec\_bmad)**

Subroutine to convert from PTC to Bmad coordinates.

## 23.27 Quick Plot Routines

### 23.27.1 Quick Plot Page Routines

**qp\_open\_page (page\_type, i\_chan, x\_len, y\_len, units)**

Subroutine to Initialize a page (window) for plotting.

**qp\_select\_page (iw)**

Subroutine to switch to a particular page for drawing graphics.

**qp\_close\_page()**

Subroutine to finish plotting on a page.

### 23.27.2 Quick Plot Computational Routines

**qp\_axis\_niceness (imin, imax, divisions) result (score)**

Routine to calculate how “nicely” an axis will look. The higher the score the nicer.

**qp\_calc\_and\_set\_axis (axis, data\_min, data\_max, div\_min, div\_max,  
                          bounds, axis\_type, slop\_factor)**

Subroutine to calculate a "nice" plot scale given the minimum and maximum of the data.

**qp\_calc\_axis\_params (data\_min, data\_max, div\_min,  
                          div\_max, how, places, axis\_min, axis\_max, divisions)**

Subroutine to calculate a "nice" plot scale given the minimum and maximum of the data. This is similar to calc\_axis\_scale.

**qp\_calc\_axis\_divisions (axis\_min, axis\_max, div\_min, div\_max, divisions)**

Routine to calculate the best (gives the nicest looking drawing) number of major divisions for fixed axis minimum and maximum.

**qp\_calc\_axis\_places (axis\_min, axis\_max, divisions, places)**

Subroutine to calculate the number of decimal places needed to display the axis numbers.

**qp\_calc\_axis\_scale (data\_min, data\_max, divisions, how,  
                          places, axis\_min, axis\_max, niceness\_score)**

Subroutine to calculate a "nice" plot scale given the minimum and maximum of the data.

**qp\_calc\_minor\_div (delta, div\_max, divisions)**

Subroutine to calculate the number of minor divisions an axis should have.

**qp\_convert\_rectangle\_rel (rect1, rect2)**

Subroutine to convert a "rectangle" (structure of 4 points) from one set of relative units to another

### 23.27.3 Quick Plot Drawing Routines

**qp\_clear\_box**

Subroutine to clear the current box on the page.

**qp\_clear\_page()**

Subroutine to clear all drawing from the page.

**qp\_draw\_circle (x0, y0, r, angle0, del\_angle,  
                          units, width, color, style, clip)**

Subroutine to plot a section of an ellipse.

**qp\_draw\_ellipse (x0, y0, r\_x, r\_y, theta\_xy,  
                          angle1, angle2, units, width, color, style, clip)**

Subroutine to plot a section of an ellipse.

**qp\_draw\_axes()**

Subroutine to plot the axes, title, etc. of a plot.

**qp\_draw\_data (x, y, draw\_line, symbol\_every, clip)**

Subroutine to plot data, axes with labels, a grid, and a title.

**qp\_draw\_graph (x, y, x\_lab, y\_lab, title,  
                          draw\_line, draw\_symbol, clip, symbol\_every)**

Subroutine to plot data, axes with labels, a grid, and a title.

- qp\_draw\_graph\_title (title)**  
Subroutine to draw the title for a graph.
- qp\_draw\_grid()**  
Subroutine to draw a grid on the current graph.
- qp\_draw\_histogram (x\_dat, y\_dat, x\_lab, y\_lab, title, draw\_axes)**  
Subroutine to plot data, axes with labels, a grid, and a title.
- qp\_draw\_curve\_legend (origin, text\_offset, line\_length,  
line, symbol, text, draw\_line, draw\_symbol, draw\_text)**  
Subroutine to draw a legend with each line in the legend having a line, a symbol, some text.
- qp\_draw\_text\_legend (lines, x, y, units)**  
Subroutine to draw a legend of lines of text.
- qp\_draw\_main\_title (lines, justify)**  
Subroutine to plot the main title at the top of the page.
- qp\_draw\_polyline (x, y, units, width, color, style, clip)**  
Subroutine to draw a polyline.
- qp\_draw\_polyline\_no\_set (x, y, units)**  
Subroutine to draw a polyline. This is similar to qp\_draw\_polyline except qp\_set\_line\_attrib is not called.
- qp\_draw\_polyline\_basic (x, y, units)**  
Subroutine to draw a polyline. See also qp\_draw\_polyline
- qp\_draw\_line (x1, x2, y1, y2, units, width, color, style, clip)**  
Subroutine to draw a line.
- qp\_draw\_rectangle (x1, x2, y1, y2, units, color, width, style, clip)**  
Subroutine to draw a rectangular box.
- qp\_draw\_symbol (x, y, units, type, height, color, fill, line\_width, clip)**  
Draws a symbol at (x, y)
- qp\_draw\_symbols (x, y, units, type, height, color,  
fill, line\_width, clip, symbol\_every)**  
Draws a symbol at the (x, y) points.
- qp\_draw\_text (text, x, y, units, justify, height, color, angle, ...)**  
Subroutine to draw text.
- qp\_draw\_text\_no\_set (text, x, y, units, justify, angle)**  
Subroutine to display on a plot a character string. See also: qp\_draw\_text.
- qp\_draw\_text\_basic (text, x, y, units, justify, angle)**  
Subroutine to display on a plot a character string. See also: qp\_draw\_text.
- qp\_draw\_x\_axis (who, y\_pos)**  
Subroutine to draw a horizontal axis.
- qp\_draw\_y\_axis (who, x\_pos)**  
Subroutine to draw a horizontal axis.

**qp\_paint\_rectangle (x1, x2, y1, y2, units, color)**

Subroutine to paint a rectangular region a specified color. The default color is the background color (white\$).

**qp\_to\_axis\_number\_text (axis, ix\_n, text)**

Subroutine to form the text string for an axis number.

**23.27.4 Quick Plot Set Routines****qp\_calc\_and\_set\_axis (axis, data\_min, data\_max,  
div\_min, div\_max, bounds, axis\_type, slop\_factor)**

Subroutine to calculate a "nice" plot scale given the minimum and maximum of the data.

**qp\_eliminate\_xy\_distortion()**

This subroutine will increase the x or y margins so that the conversion between data units and page units is the same for the x and y axes.

**qp\_set\_axis (axis, a\_min, a\_max, div, places, label, draw\_label,  
draw\_numbers, minor\_div, minor\_div\_max, mirror,  
number\_offset, label\_offset, major\_tick\_len, minor\_-****tick\_len, ax\_type)**

Subroutine to set (but not plot) the min, max and divisions for the axes of the graph.

**qp\_set\_box (ix, iy, ix\_tot, iy\_tot)**

Subroutine to set the box on the physical page. This routine divides the page into a grid of boxes.

**qp\_set\_graph (title)**

Subroutine to set certain graph attributes.

**qp\_set\_graph\_limits()**

Subroutine to calculate the offsets for the graph. This subroutine also sets the PGPLOT window size equal to the graph size.

**qp\_set\_graph\_placement (x1\_marg, x\_graph\_len, y1\_marg, y\_graph\_len, units)**

Subroutine to set the placement of the current graph inside the box. This routine can be used in place of qp\_set\_margin.

**qp\_set\_layout (x\_axis, y\_axis, x2\_axis, y2\_axis, ...)**

Subroutine to set various attributes. This routine can be used in place of other qp\_set\_\* routines.

**qp\_set\_line (who, line)**

Subroutine to set the default line attributes.

**qp\_set\_margin (x1\_marg, x2\_marg, y1\_marg, y2\_marg, units)**

Subroutine to set up the margins from the sides of the box (see QP\_SET\_BOX) to the edges of the actual graph.

**qp\_set\_page\_border (x1\_b, x2\_b, y1\_b, y2\_b, units)**

Subroutine to set the border around the physical page.

**qp\_set\_page\_border\_to\_box ()**

Subroutine to set the page border to correspond to the region of the current box. This allows qp\_set\_box to subdivide the current box.



**qp\_set\_clip (clip)**

Subroutine to set the default clipping state.

**qp\_set\_parameters (text\_scale)**

Subroutine to set various quick plot parameters.

**qp\_subset\_box (ix, iy, ix\_tot, iy\_tot, x\_marg, y\_marg)**

Subroutine to set the box for a graph. This is the same as qp\_set\_box but the boundaries of the page are taken to be the box boundaries.

**qp\_set\_symbol (symbol)**

Subroutine to set the type and size of the symbols used in plotting data. See the pgplot documentation for more details.

**qp\_set\_symbol\_attrib (type, height, color, fill, line\_width, clip)**

Subroutine to set the type and size of the symbols used in plotting data.

**qp\_set\_line\_attrib (who, width, color, style, clip)**

Subroutine to set the default line attributes.

**qp\_set\_graph\_attrib (draw\_grid, draw\_title)**

Subroutine to set attributes of the current graph.

**qp\_set\_text\_attrib (who, height, color, background, uniform\_spacing, spacing\_factor)**

Subroutine to set the default text attributes.

**qp\_use\_axis (x, y)**

Subroutine to set what axis to use: X or X2, Y or Y2.

**23.27.5 Informational Routines****qp\_get\_axis (axis, a\_min, a\_max, div, ... )**

Subroutine to get the min, max, divisions etc. for the X and Y axes.

**qp\_get\_layout\_attrib (who, x1, x2, y1, y2, units)**

Subroutine to get the attributes of the layout.

**qp\_get\_line (who, line)**

Subroutine to get the default line attributes.

**qp\_get\_parameters (text\_scale)**

Subroutine to get various quick\_plot parameters.

**qp\_get\_symbol (symbol)**

Subroutine to get the symbol parameters used in plotting data. Use qp\_set\_symbol or qp\_set\_symbol\_attrib to set symbol attributes.

**qp\_text\_len (text)**

Function to find the length of a text string.

### 23.27.6 Conversion Routines

**qp\_from\_inch\_rel (x\_inch, y\_inch, x, y, units)**

Subroutine to convert from a relative position (an offset) in inches to other units.

**qp\_from\_inch\_abs (x\_inch, y\_inch, x, y, units)**

Subroutine to convert to absolute position (x, y) from inches referenced to the Left Bottom corner of the page

**qp\_text\_height\_to\_inches(height\_pt) result (height\_inch)**

Function to convert from a text height in points to a text height in inches taking into account the text\_scale.

**qp\_to\_inch\_rel (x, y, x\_inch, y\_inch, units)**

Subroutine to convert a relative (x, y) into inches.

**qp\_to\_inch\_abs (x, y, x\_inch, y\_inch, units)**

Subroutine to convert an absolute position (x, y) into inches referenced to the Left Bottom corner of the page.

**qp\_to\_inches\_rel (x, y, x\_inch, y\_inch, units)**

Subroutine to convert a relative (x, y) into inches.

**qp\_to\_inches\_abs (x, y, x\_inch, y\_inch, units)**

Subroutine to convert an absolute position (x, y) into inches referenced to the left bottom corner of the page.

### 23.27.7 Miscellaneous Routines

**qp\_read\_data (iu, err\_flag, x, ix\_col, y, iy\_col, z, iz\_col, t, it\_col)**

Subroutine to read columns of data.

### 23.27.8 Low Level Routines

**qp\_clear\_box\_basic (x1, x2, y1, y2, page\_type)**

Subroutine to clear all drawing from a box. That is, white out the box region.

**qp\_clear\_page\_basic**

Subroutine to clear all drawing from the page.

**qp\_close\_page\_basic**

Subroutine to finish plotting on a page. For X this closes the window.

**qp\_convert\_point\_rel (x\_in, y\_in, units\_in, x\_out, y\_out, units\_out)**

Subroutine to convert a (x, y) point from from one set of relative units to another.

**qp\_convert\_point\_abs (x\_in, y\_in, units\_in, x\_out, y\_out, units\_out)**

Subroutine to convert a (x, y) point from from one set of absolute units to another.

**qp\_draw\_symbol\_basic (x, y, symbol)**

Subroutine to draw a symbol.

**qp\_init\_com\_struct**

Subroutine to initialize the common block qp\_state\_struct. This subroutine is not for general use.

**qp\_join\_units\_string (u\_type, region, corner, units)**

Subroutine to form a units from its components.

**qp\_justify (justify)**

Function to convert a justify character string to a real value representing the horizontal justification.

**qp\_open\_page\_basic (page\_type, x\_len, y\_len, plot\_file**

x\_page, y\_page, i\_chan) Subroutine to Initialize a page (window) for plotting.

**qp\_paint\_rectangle\_basic (x1, x2, y1, y2, color, page\_type)**

Subroutine to fill a rectangle with a given color. A color of white essentially erases the rectangle.

**qp\_pointer\_to\_axis (axis, axis\_ptr)**

Subroutine to return a pointer to an common block axis.

**qp\_restore\_state**

Subroutine to restore saved attributes. Use qp\_save\_state to restore the saved state.

**qp\_restore\_state\_basic ()**

Subroutine to restore the print state.

**qp\_save\_state (buffer)**

Subroutine to save the current attributes. Use qp\_restore\_state to restore the saved state.

**qp\_save\_state\_basic**

Subroutine to save the print state.

**qp\_select\_page\_basic (iw)**

Subroutine to switch to a particular page for drawing graphics.

**qp\_set\_char\_size\_basic (height)**

Subroutine to set the character size.

**qp\_set\_clip\_basic (clip)**

Subroutine to set the clipping state. Note: This affects both lines and symbols.

**qp\_set\_color\_basic (ix\_color, page\_type)**

Subroutine to set the color taking into account that GIF inverts the black for white.

**qp\_set\_graph\_position\_basic (x1, x2, y1, y2)**

Subroutine to set the position of a graph. Units are inches from lower left of page.

?

**qp\_set\_line\_width\_basic (line\_width)**

Subroutine to set the line width.

**qp\_set\_line\_style\_basic (style)**

Subroutine to set the line style.

**qp\_set\_symbol\_fill\_basic (fill)**

Subroutine to set the symbol fill style.

**qp\_set\_symbol\_size\_basic** (height, symbol\_type, page\_type, uniform\_size)

Subroutine to set the symbol\_size

**qp\_set\_text\_background\_color\_basic** (color)

Subroutine to set the character text background color.

**qp\_split\_units\_string** (u\_type, region, corner, units)

Subroutine to split a units string into its components.

**qp\_text\_len\_basic** (text, len\_text)

Function to find the length of a text string.

**qp\_translate\_to\_color\_index** (name, index)

Subroutine to translate from a string to a color index.

## 23.28 Spin Tracking

**spinor\_to\_polar** (coord, polar)

Subroutine to convert a spinor into polar coordinates.

**polar\_to\_vec** (polar, vec)

Subroutine to convert a spin vector from polar coordinates to cartesian coordinates.

**polar\_to\_spinor** (polar, coord)

Subroutine to convert a spin vector in polar coordinates to a spinor.

**vec\_to\_polar** (vec, polar, phase)

Subroutine to convert a spin vector from cartesian coordinates to polar coordinates preserving the complex phase.

**spinor\_to\_vec** (coord, vec)

Subroutine to convert a spinor to a spin vector in cartesian coordinates.

**vec\_to\_spinor** (vec, coord, phase)

Subroutine to convert a spin vector in cartesian coordinates to a spinor using the specified complex phase.

**angle\_between\_polars** (polar1, polar2)

Function to return the angle between two spin vectors in polar coordinates.

**quaternion\_track** (a, start, end)

Subroutine to track the spin with the Euler four-vector quaternion a.

**track1\_spin** (start, ele, param, end)

Subroutine to track the particle spin through one element.

## 23.29 Transfer Maps: Routines Called by make\_mat6

**Make\_mat6** is the routine for calculating the transfer matrix (Jacobin) through an element. The routines listed below are used by **make\_mat6**. In general a program should call **make\_mat6** rather than using these routines directly.

**make\_mat6\_bmad (ele, param, c0, c1)**

Subroutine to make the 6x6 transfer matrix for an element using closed formulas.

**make\_mat6\_custom (ele, param, c0, c1)**

Dummy routine for making the 6x6 transfer matrices.

**make\_mat6\_symp\_lie\_ptc (ele, param, c0, c1)**

Subroutine to make the 6x6 transfer matrix for an element using the PTC symplectic integrator.

**make\_mat6\_taylor (ele, param, c0, c1)**

Subroutine to make the 6x6 transfer matrix for an element from a Taylor map.

**make\_mat6\_tracking (ele, param, c0, c1)**

Subroutine to make the 6x6 transfer matrix for an element by tracking 7 particle with different starting conditions.

## 23.30 Transfer Maps: Taylor Maps

**add\_taylor\_term (bmad\_taylor, coef, expn, replace)**

**add\_taylor\_term (bmad\_taylor, coef, i1, i2, i3, i4, i5, i6, i7, i8, i9, replace)**

Overloaded routine to add a Taylor term to a Taylor series.

**concat\_ele\_taylor (taylor1, ele, taylor3)**

Routine to concatenate two taylor maps.

**concat\_taylor (taylor1, taylor2, taylor3)**

Subroutine to concatenate two taylor series:  $\text{taylor3}(x) = \text{taylor2}(\text{taylor1}(x))$

**ele\_to\_taylor (ele, param, orb0)**

Subroutine to make a Taylor map for an element. The order of the map is set by `set_ptc`.

**equivalent\_taylor\_attributes (ele1, ele2) result (equiv)**

Subroutine to see if to elements are equivalent in terms of attributes so that their Taylor Maps would be the same.

**init\_taylor\_series (bmad\_taylor, n\_term)**

Subroutine to initialize a Bmad Taylor series.

**kill\_taylor (bmad\_taylor)**

Subroutine to deallocate a Bmad Taylor map.

**mat6\_to\_taylor (mat6, vec0, bmad\_taylor)**

Subroutine to form a first order Taylor map from the 6x6 transfer matrix and the 0th order transfer vector.

**set\_taylor\_order (order, override\_flag)**

Subroutine to set the taylor order.

**sort\_taylor\_terms (taylor\_in, taylor\_sorted)**

Subroutine to sort the taylor terms from "lowest" to "highest" of a Taylor series.

**taylor\_coef (bmad\_taylor, exp)**

Function to return the coefficient for a particular taylor term from a Taylor Series.

**taylor\_equal\_taylor (taylor1, taylor2)**

Subroutine to transfer the values from one taylor map to another:  $\text{Taylor1} \leq \text{Taylor2}$

**taylor\_minus\_taylor (taylor1, taylor2) result (taylor3)**

Routine to add two taylor maps.

**taylor\_plus\_taylor (taylor1, taylor2) result (taylor3)**

Routine to add two taylor maps.

**taylors\_equal\_taylors (taylor1, taylor2)**

Subroutine to transfer the values from one taylor map to another.

**taylor\_make\_unit (bmad\_taylor)**

Subroutine to make the unit Taylor map

**taylor\_to\_mat6 (a\_taylor, c0, mat6, c1)**

Subroutine to calculate the linear (Jacobian) matrix about some trajectory from a Taylor map.

**taylor\_inverse (taylor\_in, taylor\_inv)**

Subroutine to invert a taylor map.

**taylor\_propagate1 (tlr, ele, param)**

Subroutine to track a real\_8 taylor map through an element. The alternative routine, if ele has a taylor series, is concat\_taylor.

**track\_taylor (start, bmad\_taylor, end)**

Subroutine to track using a Taylor map.

**transfer\_ele\_taylor (ele\_in, ele\_out, taylor\_order)**

Subroutine to transfer a Taylor map from one element to another.

**transfer\_lat\_taylors (lat\_in, lat\_out, type\_out, transfered\_all)**

Subroutine to transfer the taylor maps from the elements of one lat to the elements of another.

**truncate\_taylor\_to\_order (taylor\_in, order, taylor\_out)**

Subroutine to throw out all terms in a taylor map that are above a certain order.

**type\_taylors (bmad\_taylor)**

Subroutine to print in a nice format a Bmad taylor map at the terminal.

**type2\_taylors (bmad\_taylor, lines, n\_lines)**

Subroutine to write a Bmad taylor map in a nice format to a character array.

## 23.31 Tracking and Closed Orbit

The following routines perform tracking and closed orbit calculations.

**check\_aperture\_limit (orb, ele, param)**

Subroutine to check if an orbit is outside the aperture.

**closed\_orbit\_calc (lat, closed\_orb, i\_dim, direction)**

Subroutine to calculate the closed orbit at the beginning of the lat.

**closed\_orbit\_from\_tracking (lat, closed\_orb, i\_dim, eps\_rel, eps\_abs, init\_guess)**

Subroutine to find the closed orbit via tracking.

**compute\_even\_steps** (ds\_in, length, ds\_default, ds\_out, n\_step)

Subroutine to compute a step size ds\_out, close to ds\_in, so that an integer number of steps spans the length.

**dynamic\_aperture** (lat, track\_input, aperture)

Subroutine to determine the dynamic aperture of a lattice via tracking.

**multi\_turn\_tracking\_analysis** (track, i\_dim, track0, ele, stable, growth\_rate, chi)

Subroutine to analyze multi-turn tracking data to get the Twiss parameters etc.

**multi\_turn\_tracking\_to\_mat** (track, i\_dim, mat1, track0, chi)

Subroutine to analyze 1-turn tracking data to find the 1-turn transfer matrix and the closed orbit offset.

**offset\_particle** (ele, param, coord, set, set\_canonical,  
set\_tilt, set\_multipoles, set\_hvkicks, s\_pos)

Subroutine to effectively offset an element by instead offsetting the particle position to correspond to the local element coordinates.

**offset\_photon** (ele, param, coord, set)

Routine to effectively offset an element by instead offsetting the photon position to correspond to the local crystal or mirror coordinates.

**orbit\_amplitude\_calc** (ele, orb, amp\_a, amp\_b, amp\_na, amp\_nb, particle)

Routine to calculate the "invariant" amplitude of a particle at a particular point in its orbit.

**setup\_radiation\_tracking** (lat, closed\_orb, fluctuations\_on, damping\_on)

Subroutine to compute synchrotron radiation parameters prior to tracking.

**tilt\_coords** (tilt\_val coord, set)

Subroutine to effectively tilt (rotate in the x-y plane) an element by instead rotating the particle position with negative the angle.

**track1** (start, ele, param, end)

Subroutine to track through a single element.

**track1\_beam\_simple** (beam\_start, ele, param, beam\_end)

Routine to track a beam of particles through a single element. This routine does *\*not\** include multiparticle effects.

**track1\_bunch\_csr** (bunch\_start, lat, ix\_ele, bunch\_end)

Routine to track a bunch of particles through the element lat%ele(ix\_ele) with csr radiation effects.

**track\_all** (lat, orbit, ix\_branch)

Subroutine to track through the lat.

**track\_many** (lat, orbit, ix\_start, ix\_end, direction, ix\_branch)

Subroutine to track from one element in the lat to another.

**twiss\_and\_track** (lat, orb)

See the Twiss section for more details.

**twiss\_and\_track\_at\_s** (lat, s, ele, orb, orb\_at\_s, ix\_branch, err)

Subroutine to calculate the Twiss parameters and orbit at a particular longitudinal position.

**twiss\_and\_track\_intra\_ele** (ele, param, l\_start, l\_end, track\_entrance,

track\_exit, orbit\_start, orbit\_end, ele\_start, ele\_end, err) Routine to track a particle within an element.

**twiss\_and\_track\_partial** (ele1, ele2, param, del\_s, ele3, start, end)

Subroutine to calculate the Twiss parameters and orbit at a particular position inside an element.

**twiss\_from\_tracking** (lat, closed\_orb, d\_orb, error)

Subroutine to compute from tracking the Twiss parameters and the transfer matrices for every element in the lat.

## 23.32 Tracking: Low Level Routines

**odeint\_bmad** (start, ele, param, end, s1, s2, rel\_tol, abs\_tol, h1, hmin)

Subroutine to do Runge Kutta tracking.

**slice\_ele\_calc** (ele, param, i\_slice, n\_slice\_tot, sliced\_ele)

Routine to create an element that represents a slice of another element. This routine can be used for detailed tracking through an element.

**track1\_boris\_partial** (start, ele, param, s, ds, end)

Subroutine to track 1 step using boris tracking. This subroutine is used by track1\_boris and track1\_adaptive\_boris.

**track\_a\_drift** (orb, length)

Subroutine to track through a drift.

**track\_a\_bend** (start, ele, param, end)

Particle tracking through a bend element.

## 23.33 Tracking: Macroparticle

*Note: The macroparticle tracking code is not currently maintained in favor of tracking an ensemble of particles.*

**calc\_macro\_bunch\_params** (bunch, ele, params)

Subroutine to calculate various beam characteristics from a bunch.

**init\_macro\_distribution** (beam, init, canonical\_out)

Subroutine to initialize a macroparticle distribution.

**mat\_to\_mp\_sigma** (mat, sigma)

Subroutine to convert a sigma matrix. to a linear array of macroparticle sigmas.

**mp\_sigma\_to\_mat** (sigma, mat)

Subroutine to convert a linear array of macroparticle sigmas to a sigma matrix.

**mp\_to\_angle\_coords** (mp, energy0)

Subroutine to convert macroparticle coords from (x, px, y, py, z, pz) to (x, x', y, y', z, E).

**mp\_to\_canonical\_coords** (mp, energy0)

Subroutine to convert macroparticle coords from (x, x', y, y', z, E) to (x, px, y, py, z, pz).

**reallocate\_macro\_beam** (beam, n\_bunch, n\_macro)

Subroutine to reallocate memory within a beam\_struct.



**track1\_macro\_beam (start, ele, param, end)**

Subroutine to track a beam of macroparticles through an element.

**track\_macro\_beam (lat, beam, ix1, ix2)**

Subroutine to track a beam of macroparticles from the end of lat%ele(ix1) Through to the end of lat%ele(ix2).

**track1\_macroparticle (start, ele, param, end)**

Subroutine to track a macroparticle through an element.

## 23.34 Tracking: Mad Routines

**make\_mat6\_mad (ele, param, map, c0, c1)**

Subroutine to make the 6x6 transfer matrix for an element from the 2nd order MAD transport map. The map is stored in ele%taylor.

**make\_mad\_map (ele, particle, map)**

Subroutine to make a 2nd order transport map a la MAD.

**mad\_add\_offsets\_and\_multipoles (ele, energy, map)**

Subroutine to add in the effect of element offsets and/or multipoles on the 2nd order transport map for the element.

**mad\_drift (ele, energy, map)**

Subroutine to make a transport map for a drift space. The equivalent MAD-8 routine is: TMDRF

**mad\_elseif (ele, energy, map)**

Subroutine to make a transport map for an electric separator. The equivalent MAD-8 routine is: TMSEP

**mad\_sextupole (ele, energy, map)**

Subroutine to make a transport map for an sextupole. The equivalent MAD-8 routine is: TMSEXT

**mad\_sbend (ele, energy, map)**

Subroutine to make a transport map for a sector bend element. The equivalent MAD-8 routine is: TMBEND

**mad\_sbend\_fringe (ele, energy, into, map)**

Subroutine to make a transport map for the fringe field of a dipole. The equivalent MAD-8 routine is: TMFRNG

**mad\_sbend\_body (ele, energy, map)**

Subroutine to make a transport map for the body of a sector dipole. The equivalent MAD-8 routine is: TMSECT

**mad\_tmfoc (el, sk1, c, s, d, f)**

Subroutine to compute the linear focussing functions. The equivalent MAD-8 routine is: TMFOC

**mad\_quadrupole (ele, energy, map)**

Subroutine to make a transport map for an quadrupole element. The equivalent MAD-8 routine is: TMSEXT

**mad\_rfcavity (ele, energy, map)**

Subroutine to make a transport map for an rf cavity element. The equivalent MAD-8 routine is: TMRF

**mad\_solenooid (ele, energy, map)**

Subroutine to make a transport map for an solenoid. The equivalent MAD-8 routine is: TMSEXT

**mad\_sol\_quad (ele, energy, map)**

Subroutine to make a transport map for a combination solenoid/quadrupole. Note: There is no equivalent MAD-8 routine.

**mad\_tmsymm (te)**

subroutine to symmetrize the 2nd order map t. The equivalent MAD-8 routine is: tmsymm

**mad\_tmtilt (map, tilt)**

Subroutine to apply a tilt to a transport map. The equivalent MAD-8 routine is: TMTILT

**mad\_concat\_map2 (map1, map2, map3)**

Subroutine to concatenate two 2nd order transport maps. map3 = map2(map1)

**mad\_track1 (c0, map, c1)**

Subroutine to track through a 2nd order transfer map. The equivalent MAD-8 routine is: TM-TRAK

**track1\_mad (start, ele, param, end)**

Subroutine to track through an element using a 2nd order transfer map. Note: If map does not exist then one will be created.

**mad\_map\_to\_taylor (map, taylor)**

Subroutine to convert a mad order 2 map to a taylor map.

**taylor\_to\_mad\_map (taylor, map)**

Subroutine to convert a Taylor map to a mad order 2 map. If any of the Taylor terms have order greater than 2 they are ignored.

**make\_unit\_mad\_map (map)**

Subroutine to initialize a 2nd order transport map to unity.

## 23.35 Tracking: Routines called by TRACK1

Note: Generally you don't call these routines directly.

**symp\_lie\_bmad (ele, param, start, end, calc\_mat6)**

Symplectic integration through an element to 0th or 1st order.

**track1\_adaptive\_boris (start, ele, param, end, s\_start, s\_end)**

Subroutine to do Boris tracking with adaptive step size control.

**track1\_boris (start, ele, param, end, s\_start, s\_end)**

Subroutine to do Boris tracking.

**track1\_bmad (start, ele, param, end)**

Particle tracking through a single element BMAD\_standard style.

**track1\_custom (start, ele, param, end)**

Dummy routine for custom tracking.

**track1\_linear (start, ele, param, end)**

Particle tracking through a single element using the transfer matrix..

**track1\_radiation (start, ele, param, end, edge)**

Subroutine to put in radiation damping and/or fluctuations.

**track1\_runge\_kutta (start, ele, param, end)**

Subroutine to do tracking using Runge-Kutta integration.

**track1\_symp\_lie\_ptc (start, ele, param, end)**

Particle tracking through a single element using a Hamiltonian and a symplectic integrator.

**track1\_symp\_map (start, ele, param, end)**

Particle tracking through a single element using a partially inverted taylor map (In PTC/FPP this is called a genfield).

**track1\_taylor (start, ele, param, end)**

Subroutine to track through an element using the elements taylor series.

## 23.36 Twiss and Other Calculations

**calc\_z\_tune (lat)**

Subroutine to calculate the synchrotron tune from the full 6X6 1 turn matrix.

**chrom\_calc (lat, delta\_e, chrom\_x, chrom\_y)**

Subroutine to calculate the chromaticities by computing the tune change when then energy is changed.

**chrom\_tune (lat, delta\_e, target\_x, target\_y, err\_flag)**

Subroutine to set the sextupole strengths so that the lat has the desired chromaticities.

**quad\_beta\_ave (lat, ix\_ele, beta\_x\_ave, beta\_y\_ave)**

Subroutine to compute the average betas in a quad.

**radiation\_integrals (lat, orb, mode)**

Subroutine to calculate the synchrotron radiation integrals, the emittance, and energy spread.

**radiation\_integrals\_custom (lat, ir, orb)**

User supplied routine to calculate the synchrotron radiation integrals for a custom element.

**relative\_mode\_flip (ele1, ele2)**

Function to see if the modes of ELE1 are flipped relative to ELE2.

**set\_tune (phi\_x\_set, phi\_y\_set, dk1, lat, orb, ok)**

Subroutine to Q\_tune a lat. This routine will set the tunes to within 0.001 radian (0.06 deg).

**set\_z\_tune (lat)**

Subroutine to set the longitudinal tune by setting the RF voltages in the RF cavities.

**transfer\_map\_calc (lat, t\_map, ix1, ix2, integrate, one\_turn, unit\_start)**

Subroutine to calculate the transfer map between two elements.

**transfer\_map\_calc\_at\_s (lat, t\_map, s1, s2, integrate, one\_turn, unit\_start)**

Subroutine to calculate the transfer map between longitudinal positions s1 to s2.

**twiss\_and\_track (lat, orb)**

Subroutine to calculate the Twiss and orbit parameters. This is not necessarily the fastest routine.

**twiss\_and\_track\_partial (ele1, ele2, param, del\_s, ele3, start, end)**

Subroutine to propagate partially through ELE2 the Twiss parameters and the orbit.

**twiss\_at\_element (lat, ix\_ele, start, end, average)**

Subroutine to return the Twiss parameters at the beginning, end, or the average of an element.

**twiss\_and\_track\_at\_s (lat, s, ele, orb\_, here)**

Subroutine to calculate the Twiss parameters and orbit at a particular longitudinal position.

**twiss\_at\_start (lat)**

Subroutine to calculate the Twiss parameters at the start of the lat.

**twiss\_from\_tracking (lat, closed\_orb\_, d\_orb, error)**

Subroutine to compute from tracking, for every element in the lat, the Twiss parameters and the transfer matrices.

**twiss\_propagate1 (ele1, ele2)**

Subroutine to propagate the Twiss parameters from the end of ELE1 to the end of ELE2.

**twiss\_propagate\_all (lat, set\_match)**

Subroutine to propagate the Twiss parameters from the start to the end.

**twiss\_propagate\_many (lat, ix\_start, ix\_end, direction)**

Subroutine to propagate the Twiss parameters from one element in the lat to another.

**twiss\_to\_1\_turn\_mat (twiss, phi, mat2)**

Subroutine to form the 2x2 1-turn transfer matrix from the Twiss parameters.

## 23.37 Twiss: 6 Dimensional

**normal\_mode3\_calc (mat, tune, g, v, synchrotron\_motion)**

Decompose a  $2n \times 2n$  symplectic matrix into normal modes. For more details see:

**twiss3\_propagate\_all (lat)**

Subroutine to propagate the twiss parameters using all three normal modes.

**twiss3\_propagate1 (ele1, ele2)**

Subroutine to propagate the twiss parameters using all three normal modes.

**twiss3\_at\_start (lat)**

Subroutine to propagate the twiss parameters using all three normal modes.

## 23.38 Wake Fields

**init\_wake (wake, n\_sr\_table, n\_sr\_mode\_long, n\_sr\_mode\_trans, n\_lr)**

Subroutine to initialize a wake struct.

**lr\_wake\_apply\_kick (ele, s\_ref, orbit)**

Subroutine to apply the long-range wake kick to a particle.

**randomize\_lr\_wake\_frequencies (ele, set\_done)**

Routine to randomize the frequencies of the lr wake HOMs.

**sr\_table\_apply\_trans\_kick** (ele, leader, charge, follower)

Subroutine to put in the kick for the short-range wakes.

**sr\_mode\_long\_wake\_add\_to** (ele, orbit, charge)

Subroutine to add to the existing short-range wake the contribution from a passing (macro)particle.

**sr\_mode\_long\_wake\_apply\_kick** (ele, orbit)

Subroutine to put in the kick for the short-range wakes.

**sr\_mode\_long\_self\_wake\_apply\_kick** (ele, charge, orbit)

Subroutine to put in the kick for the short-range wakes

**sr\_mode\_trans\_wake\_add\_to** (ele, orbit, charge)

Subroutine to add to the existing short-range wake the contribution from a passing (macro)particle.

**sr\_mode\_trans\_wake\_apply\_kick** (ele, orbit)

Subroutine to put in the kick for the short-range wakes

**track1\_sr\_wake** (bunch, ele)

Subroutine to apply the short range wake fields to a bunch.

**track1\_lr\_wake** (bunch, ele)

Subroutine to put in the long-range wakes for particle tracking.

**zero\_lr\_wakes\_in\_lat** (lat)

Routine to zero the long range wake amplitudes for the elements that have long range wakes in a lattice.

## 23.39 Deprecated

**elements\_locator** (ele\_name, lat, indx, err)

Replaced by lat\_ele\_locator.

**elements\_locator\_by\_key** (key, lat, indx)

Replaced by lat\_ele\_locator.

**element\_locator** (ele\_name, lat, ix\_ele)

Replaced by lat\_ele\_locator.

**emit\_calc** (lat, what, mode)

Subroutine to calculate the emittance, energy spread, and synchrotron integrals. This subroutine assumes that bends are in the horizontal plane.



# Bibliography

- [1] D. Sagan, J. Smith, *The Tao Manual*. Can be obtained at:  
[http://www.lepp.cornell.edu/~dcs/bmad/tao\\_entry\\_point.html](http://www.lepp.cornell.edu/~dcs/bmad/tao_entry_point.html)
- [2] H. Grote, F. C. Iselin, *The MAD Program User's Reference Manual*, Version 8.19, CERN/SL/90-13 (AP) (REV. 5) (1996). Can be obtained at:  
<http://mad.home.cern.ch/mad>
- [3] F. C. Iselin, *The MAD program Physical Methods Manual*, unpublished, (1994). Can be obtained at:  
<http://mad.home.cern.ch/mad>
- [4] The Bmad web site:  
<http://www.lepp.cornell.edu/~dcs/bmad>
- [5] L. M. Healy, *Lie Algebraic Methods for Treating Lattice Parameter Errors in Particle Accelerators*. Doctoral thesis, University of Maryland, unpublished, (1986).
- [6] D. Sagan, J. Crittenden, and D. Rubin. "A Symplectic Model for Wigglers," Part. Acc. Conf. (2003).
- [7] J. Corbett and Y. Nosochkov, "Effect of Insertion Devices in SPEAR-3," Proc. 1999 Part. Acc. Conf., p. 238, (1999).
- [8] J. M. Jowett, "Introductory Statistical Mechanics for Electron Storage Rings," AIP Conf. Proc. 153, Physics of Part. Acc., M. Month and M. Dienes Eds., pp. 864, (1987).
- [9] D. Sagan and D. Rubin "Linear Analysis of Coupled Lattices," Phys. Rev. ST Accel. Beams **2**, 074001 (1999).
- [10] K. L. Brown, F. Rothacker, D. C. Carey, and Ch. Iselin, "TRANSPORT Appendix," Fermilab, unpublished, (December 1977).
- [11] Alexander Chao, *Physics of Collective Beam Instabilities in High Energy Accelerators*, Wiley, New York (1993).
- [12] R. H. Helm, M. J. Lee, P. L. Morton, and M. Sands, "Evaluation of Synchrotron Radiation Integrals," IEEE Trans. Nucl. Sci. NS-20, 900 (1973).
- [13] D. Sagan, "An Efficient Formalism for Simulating the Longitudinal Kick from Coherent Synchrotron Radiation," Proc. Europ. Part. Accel. Conf. p. 2829 — 31 (2006).
- [14] P. Tenenbaum, "LIBXSIF, A Stand alone Library for Parsing the Standard Input Format," Proc. 2001 Part. Acc. Conf. p. 3093 — 95 (2001).

- [15] R. Talman, “Multiparticle Phenomena and Landau Damping,” in AIP Conf. Proc. **153** pp. 789–834, M. Month and M. Dienes editors, American Institute of Physics, New York (1987).
- [16] J. Rosenzweig and L. Serafini, “Transverse Particle Motion in Radio–Frequency Linear Accelerators,” Phys Rev E, Vol. 49, p. 1599, (1994).
- [17] P. H. Stoltz and J. R. Cary, “Efficiency of a Boris–like Integration Scheme with Spatial Stepping,” Phys. Rev. Special Topics — Accel. & Beams **5**, 094001 (2002).
- [18] W. Press, B. Flannery, S. Teukolsky, and W. Wetterling, *Numerical Recipes in Fortran, the Art of Scientific Computing*, Second Edition, Cambridge University Press, New York, (1992).  
W. Press, B. Flannery, S. Teukolsky, and W. Wetterling, *Numerical Recipes in Fortran90, the Art of Parallel Scientific Computing*, Cambridge University Press, New York, (1996).
- [19] H. Wiedemann, *Particle Accelerator Physics*, Springer, New York, (1999).
- [20] E. Forest, *Beam Dynamics: A New Attitude and Framework*, Harwood Academic Publishers, Amsterdam (1998).
- [21] E. D. Courant and H. S. Snyder “Theory of Alternating–Gradient Synchrotron,” Ann. Physics, **3**, p. 1-48. (1958).
- [22] The Accelerator Markup Language / Universal Accelerator Project web page:  
<http://www.lepp.cornell.edu/~dcs/aml/>
- [23] R. Storn, and K. V. Price, “Minimizing the real function of the ICEC’96 contest by differential evolution” IEEE conf. on Evolutionary Computation, 842-844 (1996).
- [24] A. Wolski, “Alternative approach to general coupled linear optics,” Phys. Rev. ST Accel. Beams **9**, 024001 (2006).



# Routine Index

ab\_multipole\_kick, 210  
abs\_sort, 198  
add\_lattice\_control\_structs, 203  
add\_sr\_long\_wake, 192  
add\_superimpose, 203  
add\_taylor\_term, 221  
adjust\_super\_lord\_s\_position, 204  
allocate\_branch\_array, 194  
allocate\_ele\_array, 205  
allocate\_lat\_ele\_array, 205  
aml\_parser, 207  
amode\_to\_c, 194  
angle\_between\_polars, 220  
angle\_to\_canonical\_coords, 193  
append\_subdirectory, 197  
arr2mat, 194  
attribute\_bookkeeper, 154, 203  
attribute\_free, 201  
attribute\_index, 138, 201  
attribute\_name, 138, 201  
attribute\_type, 201  
attribute\_value\_name, 201  
  
bane, 201  
bbi\_kick, 193  
beam\_equal\_beam, 211  
bjmt, 201  
bmad\_com\_to\_c, 194  
bmad\_parser, 131, 132, 142, 146, 157, 162, 165, 207  
bmad\_parser2, 157, 207  
bmad\_to\_mad\_or\_xsif, 158, 207  
bracket\_index, 204  
branch\_equal\_branch, 211  
bunch\_equal\_bunch, 211  
  
c\_logic, 194  
c\_multi, 206  
c\_str, 194  
c\_to\_cbar, 161, 207  
calc\_bunch\_params, 193  
calc\_bunch\_params\_slice, 193  
calc\_macro\_bunch\_params, 224  
calc\_z\_tune, 227  
canonical\_to\_angle\_coords, 193  
cbar\_to\_c, 207  
cesr\_getarg, 197  
cesr\_iarg, 197  
changed\_attribute\_bookkeeper, 162, 203  
check\_aperture\_limit, 167, 222  
check\_controller\_controls, 204  
check\_if\_ele\_is\_monitor, 209  
check\_lat\_controls, 201  
chrom\_calc, 163, 227  
chrom\_tune, 163, 227  
cimp, 201  
clear\_lat\_1turn\_mats, 207  
closed\_orbit\_calc, 222  
closed\_orbit\_from\_tracking, 222  
combine\_consecutive\_elements, 207  
complex\_error\_function, 198  
compute\_bpm\_transformation\_numbers, 209  
compute\_even\_steps, 222  
compute\_reference\_energy, 154, 206  
concat\_ele\_taylor, 221  
concat\_real\_8, 212  
concat\_taylor, 168, 221  
control\_bookkeeper, 154, 205  
control\_to\_c, 194  
convert\_coords, 211  
convert\_pc\_to, 206  
convert\_total\_energy\_to, 206  
coord\_equal\_coord, 211  
coord\_to\_c, 194  
create\_girder, 203  
create\_group, 203  
create\_overlay, 203  
create\_sol\_quad\_model, 207  
create\_unique\_ele\_names, 207  
create\_wiggler\_model, 203  
cross\_product, 206  
csr\_bin\_kicks, 196

- csr\_bin\_particles, 196
- csr\_kick\_calc, 196
- d\_calc\_csr, 196
- date\_and\_time\_stamp, 199
- deallocate\_branch, 194
- deallocate\_ele\_array\_pointers, 205
- deallocate\_ele\_pointers, 135, 204
- deallocate\_lat\_pointers, 146, 205
- determinant, 207
- dir\_close, 197
- dir\_open, 197
- dir\_read, 197
- dispersion\_to\_orbit, 204
- do\_mode\_flip, 207
- downcase\_string, 200
- drift\_mat6\_calc, 209
- dynamic\_aperture, 223
- ele\_at\_s, 201
- ele\_equal\_ele, 136, 211
- ele\_geometry, 139, 204
- ele\_to\_c, 194
- ele\_to\_fibre, 212
- ele\_to\_lat\_loc, 201
- ele\_to\_taylor, 168, 221
- ele\_vec\_equal\_ele\_vec, 211
- element\_locator, 229
- elements\_locator, 229
- elements\_locator\_by\_key, 229
- em\_field, 171
- em\_field\_calc, 196
- em\_field\_custom, 171, 196
- em\_field\_kick, 196
- em\_field\_to\_c, 194
- em\_filed\_custom, 170
- emit\_calc, 229
- equivalent\_taylor\_attributes, 202, 221
- err\_exit, 199
- f\_logic, 194
- field\_interpolate\_3d, 206
- file\_suffixer, 197
- find\_bunch\_sigma\_matrix, 192
- find\_element\_ends, 202
- floor\_angles\_to\_w\_mat, 204
- floor\_position\_to\_c, 195
- floor\_w\_mat\_to\_angles, 204
- g, 201
- get\_a\_char, 197
- get\_element\_slave\_list, 202
- get\_file\_time\_stamp, 197
- get\_tty\_char, 197
- i\_csr, 196
- ibs\_equilibrium, 196
- ibs\_lifetime, 201
- ibs\_rates, 196
- ibsequilibrium2, 196
- index\_nocase, 200
- indexx\_char, 200
- init\_beam\_distribution, 169, 193
- init\_bunch\_distribution, 193
- init\_coord, 211
- init\_ele, 205
- init\_floor, 204
- init\_lat, 205
- init\_macro\_distribution, 224
- init\_spin\_distribution, 193
- init\_taylor\_series, 221
- init\_wake, 228
- initial\_lmdif, 210
- insert\_element, 203
- integer\_option, 199
- ion\_kick, 193
- is\_integer, 200
- is\_logical, 200
- is\_real, 200
- key\_name, 146, 202
- key\_name\_to\_key\_index, 202
- kick\_csr\_lsc, 196
- kill\_gen\_field, 212
- kill\_taylor, 168, 221
- kind\_name, 212
- lat\_ele\_locator, 132, 133, 154, 202
- lat\_equal\_lat, 146, 211
- lat\_geometry, 139, 154, 204
- lat\_make\_mat6, 140, 154, 162, 208
- lat\_reverse, 169, 205
- lat\_to\_c, 195
- lat\_to\_layout, 212
- lat\_vec\_equal\_lat\_vec, 211
- lattice\_bookkeeper, 133, 153, 205
- linac\_mode\_to\_c, 195
- linear\_fit, 198
- logic\_option, 199
- lr\_wake\_apply\_kick, 228
- lr\_wake\_to\_c, 195
- lunget, 197
- mad\_add\_offsets\_and\_multipoles, 225

mad\_concat\_map2, 226  
mad\_drift, 225  
mad\_elsep, 225  
mad\_map\_to\_taylor, 226  
mad\_quadrapole, 225  
mad\_rfcavity, 225  
mad\_sbend, 225  
mad\_sbend\_body, 225  
mad\_sbend\_fringe, 225  
mad\_sextupole, 225  
mad\_sol\_quad, 226  
mad\_solenoid, 225  
mad\_tmfoc, 225  
mad\_tmsymm, 226  
mad\_tmtilt, 226  
mad\_track1, 226  
make.mat6, 171  
make\_g2\_mats, 207  
make\_g\_mats, 208  
make\_hybrid\_lat, 203  
make\_mad\_map, 225  
make\_mat6, 140, 208  
make\_mat6\_bmad, 220  
make\_mat6\_custom, 171, 221  
make\_mat6\_mad, 225  
make\_mat6\_symp\_lie\_ptc, 221  
make\_mat6\_taylor, 221  
make\_mat6\_tracking, 221  
make\_unit\_mad\_map, 226  
make\_v\_mats, 161, 208  
makeup\_super\_slave, 204  
makeup\_super\_slave1, 203  
map\_coef, 212  
mat2arr, 195  
mat6\_calc\_at\_s, 208  
mat6\_dispersion, 209  
mat6\_to\_taylor, 208, 221  
mat\_det, 189  
mat\_eigen, 199  
mat\_inverse, 199  
mat\_make\_unit, 189, 199  
mat\_rotation, 199  
mat\_symp\_conj, 199  
mat\_symp\_decouple, 199  
mat\_symp\_error, 199  
mat\_symplectify, 199  
mat\_to\_mp\_sigma, 224  
mat\_type, 199  
match\_ele\_to\_mat6, 208  
match\_reg, 200  
match\_wild, 200  
match\_word, 200  
mexp, 210  
milli\_sleep, 197  
mode\_info\_to\_c, 195  
modes\_to\_c, 195  
modulo2, 198  
mp\_beam\_equal\_mp\_beam, 211  
mp\_sigma\_to\_mat, 224  
mp\_to\_angle\_coords, 224  
mp\_to\_canonical\_coords, 224  
mtto, 201  
multi\_turn\_tracking\_analysis, 223  
multi\_turn\_tracking\_to\_mat, 208, 223  
multipass\_all\_info, 209  
multipass\_chain, 209  
multipole\_ab\_to\_kt, 210  
multipole\_ele\_to\_ab, 210  
multipole\_ele\_to\_kt, 210  
multipole\_init, 210  
multipole\_kick, 210  
multipole\_kicks, 210  
multipole\_kt\_to\_ab, 210  
n\_attrib\_string\_max\_len, 202  
name\_to\_list, 206  
new\_control, 203  
normal\_mode3\_calc, 162, 228  
odeint\_bmad, 224  
offset\_particle, 223  
offset\_photon, 223  
on\_off\_logic, 200  
one\_turn\_mat\_at\_ele, 208  
opti\_de, 189, 210  
opti\_lmdif, 189, 210  
orbit\_amplitude\_calc, 223  
orbit\_to\_dispersion, 204  
order\_particles\_in\_z, 193  
order\_super\_lord\_slaves, 206  
out\_io, 197  
output\_direct, 197  
param\_to\_c, 195  
photon\_energy\_init, 212  
photon\_init, 212  
photon\_vert\_angle\_init, 212  
pointer\_to\_attribute, 203  
pointer\_to\_ele, 203  
pointer\_to\_indexed\_attribute, 202  
pointer\_to\_lord, 150, 152, 202  
pointer\_to\_multipass\_lord, 209  
pointer\_to\_slave, 150, 152, 202

- pointers\_to\_attribute, 203
- polar\_to\_spinor, 220
- polar\_to\_vec, 220
  
- qp\_axis\_niceness, 214
- qp\_calc\_and\_set\_axis, 180, 181, 214, 216
- qp\_calc\_axis\_divisions, 214
- qp\_calc\_axis\_params, 214
- qp\_calc\_axis\_places, 214
- qp\_calc\_axis\_scale, 214
- qp\_calc\_minor\_div, 214
- qp\_clear\_box, 214
- qp\_clear\_box\_basic, 218
- qp\_clear\_page, 214
- qp\_clear\_page\_basic, 218
- qp\_close\_page, 180, 213
- qp\_close\_page\_basic, 218
- qp\_convert\_point\_abs, 218
- qp\_convert\_point\_rel, 218
- qp\_convert\_rectangle\_rel, 214
- qp\_draw\_axes, 180, 181, 214
- qp\_draw\_circle, 214
- qp\_draw\_curve\_legend, 215
- qp\_draw\_data, 180, 214
- qp\_draw\_ellipse, 214
- qp\_draw\_graph, 214
- qp\_draw\_graph\_title, 214
- qp\_draw\_grid, 215
- qp\_draw\_histogram, 215
- qp\_draw\_line, 215
- qp\_draw\_main\_title, 215
- qp\_draw\_polyline, 215
- qp\_draw\_polyline\_basic, 215
- qp\_draw\_polyline\_no\_set, 215
- qp\_draw\_rectangle, 182, 215
- qp\_draw\_symbol, 215
- qp\_draw\_symbol\_basic, 218
- qp\_draw\_symbols, 215
- qp\_draw\_text, 180, 181, 215
- qp\_draw\_text\_basic, 215
- qp\_draw\_text\_legend, 215
- qp\_draw\_text\_no\_set, 215
- qp\_draw\_x\_axis, 215
- qp\_draw\_y\_axis, 215
- qp\_eliminate\_xy\_distortion, 216
- qp\_from\_inch\_abs, 218
- qp\_from\_inch\_rel, 218
- qp\_get\_axis, 217
- qp\_get\_layout\_attrib, 217
- qp\_get\_line, 217
- qp\_get\_parameters, 217
- qp\_get\_symbol, 217
- qp\_init\_com\_struct, 218
- qp\_join\_units\_string, 219
- qp\_justify, 219
- qp\_open\_page, 180, 182, 213
- qp\_open\_page\_basic, 219
- qp\_paint\_rectangle, 215
- qp\_paint\_rectangle\_basic, 219
- qp\_pointer\_to\_axis, 219
- qp\_read\_data, 180, 181, 218
- qp\_restore\_state, 180, 181, 219
- qp\_restore\_state\_basic, 219
- qp\_save\_state, 180, 181, 219
- qp\_save\_state\_basic, 219
- qp\_select\_page, 213
- qp\_select\_page\_basic, 219
- qp\_set\_axis, 180, 183, 184, 216
- qp\_set\_box, 180–182, 216
- qp\_set\_char\_size\_basic, 219
- qp\_set\_clip, 216
- qp\_set\_clip\_basic, 219
- qp\_set\_color\_basic, 219
- qp\_set\_graph, 216
- qp\_set\_graph\_attrib, 180, 217
- qp\_set\_graph\_limits, 216
- qp\_set\_graph\_placement, 216
- qp\_set\_graph\_position\_basic, 219
- qp\_set\_layout, 216
- qp\_set\_line, 216
- qp\_set\_line\_attrib, 180, 181, 217
- qp\_set\_line\_style\_basic, 219
- qp\_set\_line\_width\_basic, 219
- qp\_set\_margin, 180–183, 216
- qp\_set\_page\_border, 180–182, 216
- qp\_set\_page\_border\_to\_box, 216
- qp\_set\_parameters, 183, 217
- qp\_set\_symbol, 217
- qp\_set\_symbol\_attrib, 180, 181, 217
- qp\_set\_symbol\_fill\_basic, 219
- qp\_set\_symbol\_size\_basic, 219
- qp\_set\_text\_attrib, 217
- qp\_set\_text\_background\_color\_basic, 220
- qp\_split\_units\_string, 220
- qp\_subset\_box, 217
- qp\_text\_height\_to\_inches, 218
- qp\_text\_len, 217
- qp\_text\_len\_basic, 220
- qp\_to\_axis\_number\_text, 216
- qp\_to\_inch\_abs, 218
- qp\_to\_inch\_rel, 218
- qp\_to\_inches\_abs, 218

- qp\_to\_inches\_rel, 218
- qp\_translate\_to\_color\_index, 220
- qp\_use\_axis, 184, 217
- quad\_beta\_ave, 227
- quaternion\_track, 220
- radiation\_integrals, 171, 227
- radiation\_integrals\_custom, 171, 227
- ran\_engine, 198
- ran\_gauss, 198
- ran\_gauss\_converter, 198
- ran\_seed\_get, 198
- ran\_seed\_put, 198
- ran\_uniform, 198
- randomize\_lr\_wake\_frequencies, 228
- re\_allocate, 199
- re\_allocate\_eles, 205
- re\_associate, 199
- read\_a\_line, 197
- read\_digested\_bmad\_file, 158, 207
- real\_8\_equal\_taylor, 211, 212
- real\_8\_init, 212
- real\_8\_to\_taylor, 212
- real\_option, 199
- reallocate\_beam, 193
- reallocate\_coord, 205
- reallocate\_macro\_beam, 224
- relative\_mode\_flip, 227
- release\_rad\_int\_cache, 206
- remove\_constant\_taylor, 212
- remove\_eles\_from\_lat, 203
- reverse\_ele, 205
- s\_calc, 154, 204
- set\_design\_linear, 205
- set\_on\_off, 205
- set\_ptc, 212
- set\_taylor\_order, 168, 212, 221
- set\_tune, 163, 227
- set\_z\_tune, 163, 227
- setup\_radiation\_tracking, 168, 223
- setup\_trans\_space\_charge\_calc, 196
- skip\_header, 197
- slice\_ele\_calc, 224
- sol\_quad\_mat6\_calc, 209
- sort\_taylor\_terms, 221
- sort\_universal\_terms, 213
- spinor\_to\_polar, 220
- spinor\_to\_vec, 220
- spline\_akima, 198
- spline\_evaluate, 198
- split\_lat, 203
- splitfilename, 197
- sr\_mode\_long\_self\_wake\_apply\_kick, 229
- sr\_mode\_long\_wake\_add\_to, 229
- sr\_mode\_long\_wake\_apply\_kick, 229
- sr\_mode\_trans\_wake\_add\_to, 229
- sr\_mode\_trans\_wake\_apply\_kick, 229
- sr\_mode\_wake\_to\_c, 195
- sr\_table\_apply\_trans\_kick, 228
- sr\_table\_wake\_to\_c, 195
- str\_downcase, 200
- str\_match\_wild, 200
- str\_substitute, 200
- string\_option, 199
- string\_to\_int, 200
- string\_to\_real, 200
- string\_trim, 200
- string\_trim2, 200
- suggest\_lmdif, 210
- super\_ludcmp, 198
- super\_mrqrmin, 189, 210
- symp\_lie\_bmad, 226
- system\_command, 198
- taylor\_coef, 168, 221
- taylor\_equal\_real\_8, 211, 213
- taylor\_equal\_taylor, 221
- taylor\_inverse, 222
- taylor\_make\_unit, 222
- taylor\_minus\_taylor, 222
- taylor\_plus\_taylor, 222
- taylor\_propagate1, 222
- taylor\_term\_to\_c, 195
- taylor\_to\_c, 195
- taylor\_to\_genfield, 213
- taylor\_to\_mad\_map, 226
- taylor\_to\_mat6, 208, 222
- taylor\_to\_real\_8, 213
- taylors\_equal\_taylors, 222
- theta\_floor, 204
- tilt\_coords, 223
- tilt\_mat6, 209
- to\_eta\_reading, 209
- to\_orbit\_reading, 209
- to\_phase\_and\_coupling\_reading, 209
- touschek\_lifetime, 196
- track1, 166, 171, 223
- track1\_adaptive\_boris, 226
- track1\_beam, 193
- track1\_beam\_simple, 223
- track1\_bmad, 226

- track1\_boris, 226
- track1\_boris\_partial, 224
- track1\_bunch, 193
- track1\_bunch\_csr, 223
- track1\_bunch\_custom, 171, 193
- track1\_bunch\_hom, 193
- track1\_custom, 171, 226
- track1\_linear, 226
- track1\_lr\_wake, 229
- track1\_macro\_beam, 224
- track1\_macroparticle, 225
- track1\_mad, 226
- track1\_particle, 193
- track1\_radiation, 226
- track1\_runge\_kutta, 227
- track1\_spin, 220
- track1\_sr\_wake, 229
- track1\_symp\_lie\_ptc, 227
- track1\_symp\_map, 227
- track1\_taylor, 227
- track\_a\_bend, 224
- track\_a\_drift, 224
- track\_all, 165, 166, 223
- track\_beam, 194
- track\_macro\_beam, 225
- track\_many, 166, 169, 223
- track\_taylor, 222
- transfer\_branch, 194
- transfer\_branches, 194
- transfer\_ele, 205
- transfer\_ele\_taylor, 206, 222
- transfer\_eles, 205
- transfer\_lat, 206
- transfer\_lat\_parameters, 206
- transfer\_lat\_taylor, 206, 222
- transfer\_map\_calc, 227
- transfer\_map\_calc\_at\_s, 227
- transfer\_mat2\_from\_twiss, 208
- transfer\_mat\_from\_twiss, 208
- transfer\_matrix\_calc, 208
- truncate\_taylor\_to\_order, 222
- twiss1\_propagate, 205
- twiss3\_at\_start, 228
- twiss3\_propagate1, 228
- twiss3\_propagate\_all, 228
- twiss\_and\_track, 223, 227
- twiss\_and\_track\_at\_s, 163, 223, 228
- twiss\_and\_track\_intra\_ele, 223
- twiss\_and\_track\_partial, 163, 223, 227
- twiss\_at\_element, 228
- twiss\_at\_start, 131, 132, 162, 163, 228
- twiss\_from\_mat2, 208
- twiss\_from\_mat6, 208
- twiss\_from\_tracking, 224, 228
- twiss\_propagate1, 162, 228
- twiss\_propagate\_all, 131, 132, 162, 163, 228
- twiss\_propagate\_many, 228
- twiss\_to\_1\_turn\_mat, 208, 228
- twiss\_to\_c, 195
- type2\_ele, 135, 202
- type2\_taylor, 222
- type2\_twiss, 202
- type\_coord, 212
- type\_ele, 132, 133, 135, 202
- type\_fibre, 213
- type\_layout, 213
- type\_map, 213
- type\_map1, 213
- type\_real\_8\_taylor, 213
- type\_taylor, 222
- type\_this\_file, 198
- type\_twiss, 202
- universal\_equal\_universal, 211
- universal\_to\_bmad\_taylor, 213
- upcase\_string, 200
- update\_hybrid\_list, 204
- vec\_bmad\_to\_ptc, 213
- vec\_ptc\_to\_bmad, 213
- vec\_to\_polar, 220
- vec\_to\_spinor, 220
- wake\_to\_c, 195
- wig\_term\_to\_c, 195
- wiggler\_vec\_potential, 206
- write\_bmad\_lattice\_file, 158, 207
- write\_digested\_bmad\_file, 158, 207
- xsif\_parser, 157, 207
- xy\_disp\_to\_c, 195
- z\_calc\_csr, 196
- zero\_ele\_offsets, 206
- zero\_lr\_wakes\_in\_lat, 229

# Index

- \$
  - character to denote a parameter, [128](#)
- ab\_multipole, [24](#), [61](#), [64](#), [66](#), [99](#)
- abs, [22](#)
- abs\_tol, [68](#)
- abs\_tolerance, [81](#)
- Accelerator Markup Language (AML), [158](#)
- accordion\_edge, [46](#)
- acos, [22](#)
- adaptive\_boris, [68](#), [69](#)
  - and Taylor maps, [68](#)
  - tracking method, [63](#)
- alias, [47](#), [56](#)
- alpha\_a, [77](#)
- alpha\_b, [77](#)
- an, *see* multipole, an
- angle, [25](#), [27](#), [28](#), [55](#), [91](#)
- antiproton, [75](#)
- antiproton\$, [148](#)
- aperture, [59](#), [138](#)
- aperture\_at, [59](#), [60](#)
- aperture\_limit\_on, [76](#)
- aperture\_type, [59](#)
- arithmetic expressions, [21](#)
  - constants, [21](#)
  - intrinsic functions, *see* intrinsic functions
  - variables, [20](#)
- asin, [22](#)
- atan, [22](#)
- auto\_bookkeeper, [81](#)
- b1\_gradient, [28](#), [40](#), [42](#), [56](#)
- b2\_gradient, [28](#), [41](#), [56](#)
- b3\_gradient, [39](#), [56](#)
- b\_field, [27](#), [56](#)
- b\_field\_err, [27](#), [56](#)
- b\_max, [44](#), [55](#)
- bbi\_constant, [24](#), [55](#)
- beam, [165](#)
- beam initialization parameters, [82](#)
- beam line, *see* line
- beam statement, [76](#)
- beam tracking
  - list of routines, [193](#)
- beam\_init\_struct, [82](#)
- beam\_start statement, [76](#)
- beambeam, [24](#), [55](#), [64](#), [66](#), [75](#), [148](#)
- beginning element, [71](#)
- beginning statement, [18](#), [77](#), [90](#)
- bend\_sol\_quad, [25](#), [64](#), [66](#)
- bend\_tilt, [25](#)
- beta\_a, [77](#)
- beta\_a0, [37](#)
- beta\_a1, [37](#)
- beta\_b, [77](#)
- beta\_b0, [37](#)
- beta\_b1, [37](#)
- bl\_hkick, [56](#), [58](#)
- bl\_kick, [56](#), [58](#)
- bl\_vkick, [56](#), [58](#)
- Bmad, [2](#)
  - distribution, [125](#)
  - error reporting, [3](#)
  - general parameters, [81](#)
  - information, [3](#)
  - lattice file format, [17](#)
  - lattice format, *see* lattice file format
  - statement syntax, [18](#)
- bmad version number, [157](#)
- bmad\_com, [171](#)
- bmad\_common\_struct
  - %auto\_bookkeeper, [154](#)
  - max\_aperture\_limit, [167](#)
- bmad\_parser, [157](#)
- bmad\_standard, [64](#)
  - tracking method, [94](#)
  - transfer map method, [66](#)
- bmad\_status, [171](#)
- bn, *see* multipole, bn
- boris, [63](#), [68](#), [69](#), [170](#)
  - and Taylor maps, [68](#)

- branch, **29**, **74**, **146**
  - main, **146**
- branch\_struct, **146**
- bs\_field, **42**, **56**
- bs\_gradient, **42**
- bunch, **165**
- bunch initialization, **111**
- C++ interface, **175**
  - calling Fortran, **177**
  - classes, **175**, **176**
  - Fortran calling C++, **177**
  - list of routines, **194**
- call statement, **77**
- canonical coordinates, *see* phase space coordinates
- charge, **24**, **55**
- chromaticity, **163**
- circular\_lattice, **76**
- closed orbit, **166**
- cmat\_ij, **77**
- coef, **47**
- coherent synchrotron radiation, *see* CSR
- coherent\_synch\_rad\_on, **81**
- command, **47**
- compute\_ref\_energy, **81**
- constants, **20**, **171**
- control\_struct, **152**
- coord\_struct, **165**
- coordinates, **89**
  - global, *see* global coordinates
  - list of routines, **211**
  - phase space, *see* phase space coordinates
  - reference, *see* reference orbit
- cos, **22**
- coupler\_angle, **34**
- coupler\_at, **34**
- coupler\_phase, **34**
- coupler\_strength, **34**
- coupling, *see* normal mode
- crunch, **62**
- crunch\_calib, **62**
- crystal, **30**
- CSR, **121**
- CSR parameters, **84**
- custom, **30**, **64**, **66**, **69**, **171**
- d\_orb(6), **81**
- de\_eta\_meas, **62**
- default\_ds\_step, **81**
- default\_integ\_order, **81**
- delta\_e, **30**, **33**, **55**
- dependent attribute, **55**
- deprecated routines, **229**
- descrip, **47**, **56**
- digested files, **18**, **157**
- dispersion, **107**, **113**, **117**
- dks\_ds, **25**
- dphi0, **33**, **41**
- dphi\_a, **37**
- dphi\_b, **37**
- drift, **30**, **49**, **64**, **66**
- ds\_step, **55**, **68**, **169**
- e1, **27**, **28**
- e2, **27**, **28**
- e\_field, **31**, **55**
- e\_loss, **33**, **55**
- e\_tot, **51**, **55**, **57**, **75**, **77**
- e\_tot\_start, **57**
- ecollimator, **29**, **59**, **64**, **66**
- ele\_beginning, **49**
- ele\_center, **49**
- ele\_end, **49**
- ele\_pointer\_struct, **132**
- ele\_struct, **132**, **135**
  - %a, **138**, **161**
  - %a(:), **142**
  - %alias, **136**
  - %b, **138**, **161**
  - %b(:), **142**
  - %c\_mat, **138**, **161**
  - %component\_name, **139**
  - %descrip, **136**
  - %field\_calc, **170**
  - %field\_master, **138**
  - %floor, **139**
  - %gamma\_c, **138**, **161**
  - %gen0, **140**
  - %gen\_field, **140**
  - %ic1\_lord, **139**, **153**
  - %ic2\_lord, **139**, **153**
  - %ix1\_slave, **139**
  - %ix2\_slave, **139**
  - %ix\_pointer, **143**
  - %key, **131**, **136**
  - %logic, **143**
  - %lord\_status, **139**, **149**, **150**
  - %map\_ref\_orb\_in, **140**
  - %map\_ref\_orb\_out, **140**
  - %mat6, **140**, **168**
  - %mode3, **138**, **162**



- `%mode_flip`, 138
- `%n_lord`, 139, 150
- `%n_slave`, 139, 150
- `%name`, 136
- `%old_value(:)`, 138
- `%r`, 143
- `%ref_time`, 139
- `%s`, 131, 139
- `%slave_status`, 139, 149, 150
- `%sub_key`, 137
- `%tracking_method`, 168
- `%type`, 136
- `%value(:)`, 138
- `%vec0`, 140
- `%wake`, 140
- `%x`, 131
- `%z`, 138, 161
- attribute values, 138
- components not used by Bmad, 143
- in `lat_struct`, 146
- initialization, 135
- multipoles, 142
- pointer components, 135
- Taylor maps, 140
- transfer maps, 140
- electron, 75
- `electron$`, 148
- element, 23
  - class, 63
  - control, 45
  - lord, 45
  - slave, 45
  - table of class types, 64
- element attribute, 20, 55
  - dependent and independent, 55
- elseparator, 31, 55, 58, 64, 66, 99
- `emittance_a`, 76
- `emittance_b`, 76
- `emittance_z`, 76
- `end_edge`, 46
- `end_file` statement, 78
- energy, 76
- `entrance_end`, 60, 89, 167
- `eta_x`, 77
- `eta_x0`, 37
- `eta_y`, 77
- `eta_y0`, 37
- `etap_x`, 77
- `etap_x0`, 37
- `etap_y`, 77
- `etap_y0`, 37
- `exit_end`, 60, 89, 167
- `exit_on_error`, 172
- `exp`, 22
- `expand_lattice`, 22, 49, 50, 74, 78
- `field_calc`, 68
- `field_master`, 56
- fields (electric and magnetic)
  - custom calculation, 170
- `fint`, 27, 28
- `fintx`, 27, 28
- floor coordinates, *see* global coordinates
- `floor_position_struct`, 139
- Forest, Etienne, *see* PTC/FPP
- FPP, *see* PTC/FPP
- FPP/PTC
  - phase space convention, 95
- `free$`, 149
- functions, *see* intrinsic functions
- `g`, 25, 27, 28, 55, 56
- `g_err`, 27, 28, 56
- `gap`, 31, 55
- `getf`, 126
- `girder`, 31, 45, 49, 139, 146, 152
- `girder_lord$`, 149
- global coordinates, 90
  - in `ele_struct`, 139
  - list of routines, 204
  - reference orbit origin, 90
- `grad_loss_sr_wake`, 81
- gradient, 33, 55
- `group`, 45, 46, 49, 146, 152
- `group_lord`, 149
- `group_slave$`, 149
- `h1`, 27, 28
- `h2`, 27, 28
- `h_displace`, 33
- harmon, 41, 55
- `hgap`, 27
- `hgapx`, 27, 28
- `hkick`, 31, 33, 55, 56, 58
- `hkicker`, 31, 58, 64, 66, 99
- hybrid, 31
- instrument, 32, 62, 64, 66
- integration methods, 68
- `integrator_order`, 68, 169
- intrinsic functions, 22
- `is_on`, 36, 61

- k1, [25](#), [27](#), [28](#), [40](#), [42](#), [44](#), [55](#), [56](#)
- k2, [28](#), [41](#), [56](#)
- k3, [39](#), [56](#)
- kick, [31](#), [56](#), [58](#)
- kicker, [33](#), [58](#), [64](#), [66](#), [99](#)
- kn1, *see* multipole, kn1
- ks, [25](#), [42](#), [56](#)
  
- l, [27](#), [37](#), [38](#), [55](#), [61](#)
- l\_chord, [27](#), [55](#), [61](#)
- lat\_param\_struct, [147](#)
  - %aperture\_limit\_on, [148](#)
  - %end\_lost\_at, [148](#)
  - %ix\_lost, [148](#)
  - %lost, [148](#)
  - %n\_part, [148](#)
  - %plane\_lost\_at, [148](#)
  - %stable, [148](#)
  - %t1\_no\_RF, [148](#)
  - %t1\_with\_RF, [148](#)
  - %total\_length, [147](#)
  - aperture\_limit\_on, [167](#)
  - end\_lost\_at, [166](#), [167](#)
  - ix\_lost, [166](#), [167](#)
  - lost, [166](#), [167](#)
  - plane\_lost\_at, [166](#)
- lat\_struct, [132](#), [139](#), [145](#), [163](#)
  - %a, [163](#)
  - %b, [163](#)
  - %branch(:), [146](#)
  - %control, [152](#), [153](#)
  - %ele(:), [131](#), [146](#)
  - %ele\_init, [156](#)
  - %ic, [153](#)
  - %ix1\_slave, [153](#)
  - %ix2\_slave, [153](#)
  - %n\_ele\_max, [146](#), [165](#)
  - %n\_ele\_track, [146](#), [165](#)
  - %param, *see* lat\_param\_struct
  - example use of, [131](#)
  - pointers, [145](#)
- lattice, [71](#), [75](#)
  - expansion, [55](#), [78](#)
- lattice files, [17](#)
  - MAD files, [158](#)
  - name syntax, [18](#)
  - parser debugging, [79](#)
  - reading, [157](#)
  - reading and writing routines, [207](#)
  - XSIF, *see* XSIF
- lattice statement, [76](#)
- lattice\_type, [49](#), [75](#), [132](#)
- lcavity, [33](#), [51](#), [55](#), [57](#), [64](#), [66](#), [75](#), [94](#)
  - and lattice\_type, [76](#)
  - and param%n\_part, [148](#)
- limit, [59](#)
- line, [19](#), [71](#), [71](#), [133](#)
  - with arguments, [72](#)
- linear, [168](#)
  - tracking method, [64](#)
- linear\_lattice, [37](#), [49](#), [76](#)
- list, [71](#), [73](#)
- listf, [126](#)
- log, [22](#)
- logicals, [20](#)
- lord, [148](#)
- lords
  - ordering, [151](#)
- lr\_freq\_spread, [142](#)
- lr\_wake\_file, [33](#)
- lr\_wake\_struct, [141](#)
- lr\_wakes\_on, [81](#)
  
- macroparticle
  - list of routines, [224](#)
- macroparticles, [113](#)
  - tracking, [165](#)
- MAD, [3](#), [17](#), [23](#), [71–73](#), [76](#), [90](#), [91](#), [158](#)
  - beam statement, [76](#)
  - delayed substitution, [21](#)
  - element rotation origin, [57](#)
  - lattice format, [17](#)
  - MAD-8, [158](#)
  - Mat6\_calc\_method
    - transfer map method, [66](#)
  - phase space convention, [94](#)
  - syntax compatibility with BMAD, [21](#)
  - tracking method, [64](#)
  - units, [97](#)
- magnetic fields, [97](#)
- map, *see* transfer map
- map\_with\_offsets, [63](#), [69](#)
- marker, [36](#), [62](#), [64](#), [66](#)
- mat6\_calc\_method, [63](#), [66](#)
- mat6\_track\_symmetric, [81](#)
- match, [36](#), [64](#), [66](#)
- match\_end, [37](#)
- match\_end\_orbit, [37](#)
- matrix
  - list of routines, [207](#)
- matrix manipulation, [189](#)
- max\_aperture\_limit, [81](#)

- measurement, 108
- measurement simulations
  - list of routines, 209
- mirror, 38, 92
- mode3\_struct, 162
- monitor, 32, 62, 64, 66
- multipass, 34, 41, 45, 50, 78
  - list of routines, 209
- multipass\_lord, 151
- multipass\_lord\$, 149
- multipass\_slave, 151
- multipass\_slave\$, 149
- multipole, 38, 61, 64, 66, 99
  - an, bn, 61, 98
  - in ele\_struct, 142
  - KnL, Tn, 98
  - in ele\_struct, 142
  - knL, tn, 61
  - list of routines, 210
- n\_part, 55, 75, 76
  - in BeamBeam element, 25
- n\_pole, 44
- n\_ref\_pass, 57
- n\_sample, 62
- n\_slice, 24
- no\_digested statement, 78, 79
- no\_superimpose statement, 78, 79
- noise, 62
- none, 64, 66
- normal mode
  - a-mode, 106
  - b-mode, 106
  - Coupling, 106
- not\_a\_lord\$, 149
- null\_ele, 39
- num\_steps, 55, 68
- numerical recipes
  - library, 125, 189
- octupole, 39, 56, 64, 66, 99
- offset, 49
- offset\_moves\_aperture, 59
- ok, 172
- old\_command, 47
- optimizers, 189
- orbit
  - measurement, 108
- osc\_amplitude, 62
- overlay, 32, 45, 49, 146, 152, 155
- overlay\_slave\$, 149
- p0c, 51, 57, 75, 77
- p0c\_start, 57
- parameter statement, 18, 55, 57, 75
  - lattice\_type, 17
- paraxial approximation, 94
- parser\_debug statement, 78, 79
- particle, 76
- patch, 39, 51, 57, 64, 66, 75, 89, 93, 151
- patch\_in\_slave, 151
- patch\_in\_slave\$, 149
- pgplot
  - and Quick\_Plot, 179
  - library, 126
- phase space coordinates, 93
  - MAD convention, 94
  - PTC convention, 95
- phi0, 33, 41
- phi\_a, 77
- phi\_b, 77
- phi\_position, 77
- photon\_branch, 29, 74, 146
- photons
  - list of routines, 212
- pipe, 32
- pitch, 57
- polarity, 43
- positron, 75
- positron\$, 148
- programming
  - conventions, 128
  - debug executable, 133
  - example program, 131
  - gmake, 133
  - precision (rp), 127
  - production executable, 133
- proton, 75
- proton\$, 148
- psi\_position, 77
- PTC/FPP, 95, 173
  - initialization, 174
  - library, 125
  - list of routines, 212
  - phase space, 173
  - real\_8, 174
  - Taylor Maps, 174
  - universal\_taylor, 174
- px, 76
- px0, 37
- px1, 37
- py, 76
- py0, 37

- pyl, 37
- pz, 76
- pz0, 37
- pz1, 37
- pz\_offset, 39
- qp\_axis\_struct, 188
- qp\_line\_struct, 187
- qp\_symbol\_struct, 187
- quad\_tilt, 25
- quadrupole, 40, 56, 64, 66, 99
- quick plot
  - list of routines, 213
- quick\_plot, 179
  - axes, 183
  - color styles, 184
  - fill styles, 184
  - line styles, 184
  - position units, 182
  - structures, 187
  - symbol styles, 186
  - symbol table, 186
- radiation damping and excitation, *see*
  - synchrotron radiation
- radiation\_damping\_on, 81
- radiation\_fluctuations\_on, 81
- radius, 61, 99
- ran, 19, 22, 78
- ran\_gauss, 19, 22, 78
- ran\_seed, 22, 75
- rbend, 26, 51, 55, 56, 61, 64, 66, 89, 91, 99, 137
- rcollimator, 29, 59, 64, 66
- ref, 49
- ref\_beginning, 49
- ref\_center, 49
- ref\_end, 49
- reference orbit, 89
  - origin in global coordinates, 90
- rel\_tol, 68
- rel\_tolerance, 81
- replacement list, *see* list
- return statement, 78
- rf\_frequency, 33, 41, 55
- rfcavity, 41, 55, 64, 66
- rho, 25, 28, 44, 55, 91
- roll, 27, 57
- routine
  - ele\_loc\_to\_string, 201
- rp, 127
- runge\_kutta, 64, 68, 170
  - and Taylor maps, 68
- s\_offset, 46, 57
- sbend, 26, 51, 55, 56, 61, 64, 66, 89, 91, 99, 137
- sextupole, 41, 56, 64, 66, 99
- sig\_x, 24, 55
- sig\_y, 24, 55
- sig\_z, 24
- sim\_utils library, 125
- sin, 22
- slave, 148
  - ordering, 151
- sol\_quad, 42, 56, 64, 66, 99
- solenoid, 42, 56, 64, 66, 99
- spin tracking, 120
  - list of routines, 220
- spin\_tracking\_on, 81
- sqrt, 22
- sr\_wake\_file, 33
- sr\_wakes\_on, 81
- start\_edge, 46
- status, 172
- strings, 20
- structures, 128
- sub\_type\_out, 172
- super\_lord, 151, 166
- super\_lord\$, 149
- super\_slave, 151
- super\_slave\$, 149
- superimpose, 19, 47
  - example, 133
- switches, 20
- symmetric\_edge, 46
- symp\_lie\_Bmad, 64, 168
- symp\_lie\_bmad, 68
  - and Taylor maps, 68
  - transfer map method, 66
- symp\_lie\_PTC, 168
- symp\_lie\_ptc, 68
  - and Taylor maps, 68
  - tracking method, 64
  - transfer map method, 66
- symp\_map, 168
  - tracking method, 64
- symplectic
  - conjugate, 106
  - integration, 68
- symplectic integration, 99, 101
- symplectification, 102
- symplectify, 69
- sympliectify, 63

- synchrotron radiation
  - calculating, 168
  - damping and excitation, 105
  - integrals, 117
- tags for Lines and Lists, 73, 78
- tan, 22
- Tao, 2
- tao, 189
- taylor, 42, 64, 66, 68, 168
  - and Taylor maps, 68
  - deallocating, 168
  - tracking method, 64
  - transfer map method, 66
- taylor Map, 168
  - list of routines, 221
- taylor map, 99
  - feed-down, 101
  - reference coordinates, 100
  - structure in ele\_struct, 140
  - with digested files, 158
- taylor\_order, 75, 76, 81
- taylor\_order statement, 76
- term (for a Wiggler), 43
- theta\_position, 77
- tilt, 25, 28, 32, 36, 39, 40, 57, 60, 62, 91–93, 139
- tilt\_calib, 62
- tilt\_tot, 139
- title statement, 77
- tn, *see* multipole, tn
- tracking, 165
  - apertures, 167
  - example, 166
  - list of routines, 222
  - Macroparticles, 165
  - particle distributions, 169
  - reverse, 169
  - spin, 170
  - transfer map method, 66
- tracking methods, 63
- tracking part, 45
- tracking\_method, 63
- trans\_space\_charge\_on, 81
- transfer map
  - in ele\_struct, 140
  - mat6\_calc\_method, *see* mat6\_calc\_method
  - Taylor map, *see* Taylor map
- tune calculation, 163
- twiss
  - list of routines, 227, 228
- twiss parameters, 162
  - calculation, 162
  - calculation with custom elements, 171
- twiss\_struct, 161
- type, 47, 56
- type\_out, 172
- units, 97
  - with MAD, 97
- Universal Accelerator Parser (UAP), 158
- use statement, 18, 19, 71
- v\_displace, 33
- val1, ..., Val12, 30
- variables, *see* lattice file format, variables
- vkick, 31, 33, 55, 56, 58
- vkicker, 31, 58, 64, 66, 99
- voltage, 31, 41, 55
- wake fields, 114
  - in ele\_struct, 140
  - in Lcavity, 35
  - list of routines, 228
  - long-range, 115
  - short-range, 114
- wig\_term\_struct, 142
- wiggler, 43, 55, 61, 64, 66, 90, 104, 137
  - types, 142
- x, 76
- x0, 37
- x1, 37
- x1\_limit, 59, 138
- x2\_limit, 59, 138
- x\_gain\_calib, 62
- x\_gain\_err, 62
- x\_limit, 59, 138
- x\_offset, 25, 32, 36, 39, 40, 57, 59, 60, 62, 90, 139
- x\_offset\_calib, 62
- x\_offset\_tot, 139
- x\_pitch, 24, 25, 32, 39, 57, 60, 90, 93, 139
- x\_pitch\_tot, 139
- x\_position, 77
- x\_quad, 25
- x\_ray\_line\_len, 43
- XSIF, 17, 157
  - lcavity and lattice\_type, 17
  - library, 126
  - reference, 231
- xy\_disp\_struct, 161
- y, 76

y0, 37  
y1, 37  
y1\_limit, 59  
y2\_limit, 59  
y\_gain\_calib, 62  
y\_gain\_err, 62  
y\_limit, 59  
y\_offset, 25, 32, 36, 39, 57, 60, 62, 90, 139  
y\_offset\_calib, 62  
y\_offset\_tot, 139  
y\_pitch, 24, 25, 32, 39, 57, 60, 90, 93, 139  
y\_pitch\_tot, 139  
y\_position, 77  
y\_quad, 25  
  
z, 76  
z0, 37  
z1, 37  
z\_offset, 39  
z\_position, 77