

Revision: 22.6
July 1, 2014

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 and X-Ray 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 where 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 as well as X-rays. *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, Bragg crystals etc. In addition, elements can be defined to control the attributes of other elements. This can be used to simulate the “girder” which physically support components in the accelerator or to easily simulate the action of control room “knobs” that gang together, say, the current going through a set of quadrupoles.

One current area of development for *Bmad* is X-ray simulation. To that end, new element classes have been defined including a `mirror` element and a `crystal` element for simulations of crystal diffraction. The ultimate aim is to develop a environment where *Bmad* can be used for simulations starting from electron generation from a cathode, to X-ray generation in Wigglers and other elements, to X-ray tracking through to the experimental end stations.

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[[Tao](#)] 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[[Grote96](#), [Iselin94](#)]. *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[[Bmad](#)]. Errors and omissions are a fact of life for any reference work and comments from you, dear reader, are therefore most welcome. Please send any missives (or chocolates, or any other kind of sustenance) to:

David Sagan <dcs16@cornell.edu>

It is my pleasure to express appreciation to people who have contributed to this effort: To David Rubin for his support, to Etienne Forest (aka Patrice Nishikawa) for use of his remarkable PTC/FPP library not to mention his patience in explaining everything to me, to Mark Palmer and Matt Rendina for all their work maintaining the build system and porting *Bmad* to different platforms, to Frank Schmidt for permission to use the *MAD* tracking code. to Hans Grote for granting permission to adapt two figures from the *MAD* manual for use in this one, to Martin Berz for his DA package, and to Ivan Bazarov, Moritz Beckmann, Joel Brock, Sarah Buchan, Avishek Chatterjee, Joseph Choi, Robert Cope, Jim Crittenden, Gerry Dugan, Christie Chiu, Michael Ehrlichman, Ken Finkelstein, Mike Forster, Richard Helms, Georg Hoffstaetter, Chris Mayes, Karthik Narayan, Tia Plautz, Matt Randazzo, Michael Saelim, Jim Shanks, Jeff Smith, Jeremy Urban, Mark Woodley, and Jing Yee Chee for their help.

Contents

I Language Reference	19
1 Bmad Concepts and Organization	21
1.1 Lattice Elements	21
1.2 Lattice Branches	21
1.3 Lattice	22
1.4 Lord and Slave Elements	22
1.5 PTC: Polynomial Tracking Code	24
2 Lattice File Overview	25
2.1 Bmad Lattice File Format	25
2.2 MAD, SAD, and XSIF Lattice Files	25
2.3 Units and Constants	25
2.4 File Example and Syntax	26
2.5 Digested Files	28
2.6 Element Sequence Definition	28
2.7 Lattice Elements	28
2.8 Lattice Element Names	29
2.9 Lattice Element Attributes	30
2.10 Custom Element Attribute Names	31
2.11 Variable Types	31
2.12 Arithmetic Expressions	32
2.13 Intrinsic functions	33
2.14 Print Statement	33
2.15 Title Statement	34
2.16 Call Statement	34
2.17 Inline Call	34
2.18 Return and End_File statements	35
2.19 Expand_Lattice Statement	35
2.20 Debugging Statements	35
3 Elements	37
3.1 AB_Multipole	38
3.2 BeamBeam	39
3.3 Beginning_Ele	40
3.4 Bend_Sol_Quad	40
3.5 Bends: Rbend and Sbend	41
3.6 Capillary	45
3.7 Collimators: Ecollimator and Rcollimator	45
3.8 Crystal	46

3.9	Custom	48
3.10	Detector	49
3.11	Diffraction_Plate	49
3.12	Drift	50
3.13	E_Gun	51
3.14	Elseparator	52
3.15	EM_Field	52
3.16	Fiducial	53
3.17	Floor_Shift	54
3.18	Fork and Photon_Fork	55
3.19	Girder	57
3.20	Group	60
3.21	Hybrid	61
3.22	Instrument, Monitor, and Pipe	61
3.23	Kickers: Hkicker and Vkicker	62
3.24	Kicker	62
3.25	Lcavity	63
3.26	Marker	65
3.27	Match	65
3.28	Mirror	67
3.29	Multipole	67
3.30	Multilayer_mirror	68
3.31	Null_Ele	68
3.32	Octupole	68
3.33	Overlay	69
3.34	Patch	70
3.35	Quadrupole	72
3.36	RFcavity	72
3.37	Sad_Mult	74
3.38	Sample	75
3.39	Sextupole	76
3.40	Solenoid	76
3.41	Sol_Quad	77
3.42	Taylor	77
3.43	Wiggler	78
3.43.1	Map_Type Wigglers	78
3.43.2	Periodic_Type Wiggler Element Tracking	79
3.43.3	Common Wiggler Parameters	81
3.44	X_Ray_Init	81
4	Element Attributes	83
4.1	Dependent and Independent Attributes	83
4.2	Type, Alias and Descrip Attributes	84
4.3	Energy and Wavelength Attributes	85
4.4	Orientation: Offset, Pitch, Tilt, and Roll Attributes	86
4.4.1	Straight Line Element Orientation	86
4.4.2	Bend Element Orientation	88
4.4.3	Photon Reflecting Element Orientation	88
4.4.4	Reference Orbit Manipulator Element Orientation	89
4.4.5	Fiducial Element Orientation	89
4.4.6	Girder Orientation	90

CONTENTS	7
4.5 Hkick, Vkick, and Kick Attributes	90
4.6 Aperture and Limit Attributes	91
4.6.1 Apertures and Element Offsets	92
4.6.2 Aperture Placement	92
4.6.3 Apertures and X-Ray Generation	93
4.7 X-Rays Crystal & Compound Materials	93
4.8 Surface Properties for X-Ray elements	96
4.8.1 Surface Grid	97
4.9 Walls: Vacuum Chamber, Capillary and Diffraction Plate	98
4.9.1 Wall Syntax	98
4.9.2 Wall Sections	99
4.9.3 Interpolation Between Sections	101
4.9.4 Capillary Wall	102
4.9.5 Vacuum Chamber Wall	102
4.9.6 Diffraction Plate Wall	104
4.10 Length Attributes	105
4.11 Is_on Attribute	106
4.12 Multipole Attributes: An, Bn, KnL, Tn	106
4.13 EM Fields – Tables and Maps	106
4.13.1 Map	108
4.13.2 Grid	109
4.14 RF Couplers	110
4.15 Wakefields	110
4.15.1 Short-Range Wakes	110
4.15.2 Long-Range Wakes	111
4.16 Fringe Fields	112
4.17 Instrumental Measurement Attributes	113
5 Tracking, Spin, and Transfer Matrix Calculation Methods	115
5.1 Particle Tracking Methods	115
5.2 Linear Transfer Map Methods	119
5.3 Spin Tracking Methods	122
5.4 Integration Methods	124
5.5 Symplectify Attribute	125
5.6 Map_with_offsets Attribute	126
6 Beam Lines and Replacement Lists	127
6.1 Branch Construction Overview	127
6.2 Beam Lines and Lattice Expansion	127
6.3 Element Reversal	129
6.4 Beam Lines with Replaceable Arguments	129
6.5 Replacement Lists	129
6.6 Use Statement	130
6.7 Line and List Tags	130
7 Superposition, and Multipass	133
7.1 Superposition	133
7.1.1 Jumbo super_slaves	136
7.1.2 Changing Element Lengths when there is Superposition	137
7.2 Multipass	137
7.2.1 The Reference Energy in a Multipass Line	138

8 Lattice Parameter Statements	139
8.1 Parameter Statement	139
8.2 Beam_start Statement	141
8.3 Beam Statement	141
8.4 Beginning and Line Parameter Statements	142
9 Automatic Scaling of Accelerating Fields	145
10 Bmad Parameter Structures	147
10.1 Bmad Global Parameters	147
10.2 Beam Initialization Parameters	150
10.3 CSR Parameters	153
10.4 DE Optimizer Parameters	154
10.5 Programming Common Parameters	154
11 Lattice Examples	157
11.1 Example: Injection Line	157
11.2 Example: Energy Recovery Linac	158
11.3 Example: Patch Between reversed and non-reversed elements	159
11.4 Example: Colliding Beam Storage Rings	160
12 MAD/XSIF/SAD Lattice Conversion	163
12.1 MAD Conversion	163
12.2 XSIF Conversion	164
12.3 SAD Conversion	164
12.4 Translation Using the Universal Accelerator Parser	164
II Conventions and Physics	165
13 Coordinates	167
13.1 Local Reference Orbit	167
13.2 Reference Orbit Construction	168
13.3 Global Coordinates	170
13.3.1 Lattice Element Positioning	171
13.3.2 Position Transformation When Transforming Coordinates	173
13.3.3 Crystal and Mirror Element Coordinate Transformation	173
13.3.4 Element Misalignment and Origin Shift Transformation	174
13.3.5 Reflection Patch	174
13.4 Phase Space Coordinate System	175
13.5 Photon Phase Space Coordinate System	177
13.6 Time-based Phase Space Coordinate System	177
13.7 Relative Verses Absolute Time Tracking	177
14 Electromagnetic Fields	179
14.1 Magnetic Fields	179
14.2 RF fields	181
14.3 Wake fields	183
14.3.1 Short-Range Wakes	183
14.3.2 Long-Range Wakes	184

CONTENTS	9
15 Multiparticle Simulation	187
15.1 Bunch Initialization	187
15.1.1 Elliptical Phase Space Distribution	187
15.1.2 Kapchinsky-Vladimirsky Phase Space Distribution	188
15.2 Macroparticles	189
15.3 Touschek Scattering	191
16 Synchrotron Radiation	193
16.1 Synchrotron Radiation Damping and Excitation	193
16.2 Synchrotron Radiation Integrals	194
16.3 Coherent Synchrotron Radiation	197
17 Linear Optics	199
17.1 Coupling and Normal Modes	199
17.2 Dispersion Calculation	200
18 Taylor Maps	203
18.1 Taylor Maps	203
18.2 Symplectification	204
18.3 Map Concatenation and Feed-Down	205
18.4 Symplectic Integration	205
19 Tracking of Charged Particles	207
19.1 Element Coordinate System	207
19.1.1 Transform from Laboratory Entrance to Element Coordinates for Straight Elements	208
19.1.2 Transform from Laboratory Entrance to Element Coordinates for Bend Elements	208
19.1.3 Transform from Element Exit to Laboratory Coordinate for Straight Elements	209
19.1.4 Transform from Element Exit to Laboratory Coordinate for Bend Elements	210
19.2 Hamiltonian	210
19.3 Symplectic Integration	212
19.4 Spin Dynamics	213
19.5 BeamBeam Tracking	214
19.6 Bend Element: Fringe Tracking	215
19.7 Bend Element: Body Tracking	215
19.8 Drift Tracking	216
19.9 ElSeparator Tracking	216
19.10 Kicker, Hkicker, and Vkicker, Tracking	218
19.11 Lcavity Tracking	218
19.12 Mirror Tracking	219
19.13 Octupole Tracking	219
19.14 Patch Tracking	219
19.15 Quadrupole Tracking	220
19.16 RFcavity Tracking	221
19.17 Sextupole Tracking	221
19.18 Sol_Quad Tracking	222
19.19 Solenoid Tracking	223

20 Tracking of X-Rays	225
20.1 Coherent and Incoherent Photon Simulations	225
20.1.1 Incoherent Photon Tracking	225
20.1.2 Coherent Photon Tracking	226
20.1.3 Parially Coherent Photon Simulations	228
20.2 Element Coordinate System	228
20.2.1 Transform from Laboratory Entrance to Element Coordinates	229
20.2.2 Transform from Element Exit to Laboratory Coordinate	229
20.3 Mirror and Crystal Element Transformation	229
20.3.1 Transformation from Laboratory to Element Coordinates	229
20.3.2 Transformation from Element to Laboratory Coordinates	230
20.4 Crystal Element Tracking	232
20.4.1 Calculation of Entrance and Exit Bragg Angles	232
20.4.2 Crystal Coordinate Transformations	234
20.4.3 Laue Reference Orbit	236
20.4.4 Crystal Surface Reflections and Refractions	237
20.4.5 Bragg Crystal Tracking	238
20.4.6 Coherent Laue Crystal Tracking	240
20.4.7 Incoherent Laue Crystal Tracking	240
20.5 X-ray Targeting	241
21 Simulation Modules	243
21.1 Tune Tracker Simulator	243
21.1.1 Tune Tracker Components.	246
21.1.2 Tuning	247
21.1.3 Programmer Instructions	248
21.1.4 Tune Tracker Module	248
21.1.5 Tune Tracker Example Program	250
21.1.6 Save States	250
21.2 Instrumental Measurements	251
21.2.1 Orbit Measurement	251
21.2.2 Dispersion Measurement	251
21.2.3 Coupling Measurement	252
21.2.4 Phase Measurement	253
III Programmer's Guide	255
22 Bmad Programming Overview	257
22.1 The Bmad Libraries	257
22.2 Using getf and listf for Viewing Routine and Structure Documentation	258
22.3 Precision of Real Variables	259
22.4 Programming Conventions	260
22.5 Manual Notation	261
23 Introduction to Bmad programming	263
23.1 A First Program	263
23.2 Explanation of the Simple_Bmad_Program	265

24 The ele_struct	269
24.1 Initialization and Pointers	271
24.2 Element Attribute Bookkeeping	271
24.3 String Components	271
24.4 Element Key	272
24.5 The %value(.) array	272
24.6 Connection with the Lat_Struct	273
24.7 Limits	273
24.8 Twiss Parameters, etc.	273
24.9 Element Lords and Element Slaves	274
24.10 Coordinates, Offsets, etc.	274
24.11 Transfer Maps: Linear and Non-linear (Taylor)	275
24.12 Reference Energy and Time	276
24.13 EM Fields	276
24.14 Wakes	276
24.15 Wiggler Types	278
24.16 Multipoles	278
24.17 Tracking Methods	278
24.18 Custom and General Use Attributes	279
24.19 Bmad Reserved Variables	279
24.20 Creating Element Slices	280
25 The lat_struct	281
25.1 Initializing	282
25.2 Pointers	282
25.3 Branches in the lat_struct	282
25.4 Param_struct Component	284
25.5 Elements Controlling Other Elements	285
25.6 Lattice Bookkeeping	290
25.7 Finding Elements and Changing Attribute Values	292
25.8 Beam_start Component	293
25.9 Adding and Deleting Elements From a Lattice	293
26 Reading and Writing Lattices	295
26.1 Reading in Lattices	295
26.2 Digested Files	296
26.3 Writing Lattice files	296
27 Normal Modes: Twiss Parameters, Coupling, Emittances, Etc.	297
27.1 Components in the Ele_struct	297
27.2 Tune and Twiss Parameter Calculations	298
27.3 Tune Setting	299
27.4 Emittances & Radiation Integrals	299
27.5 Chromaticity Calculation	300
28 Tracking and Transfer Maps	301
28.1 The coord_struct	301
28.2 Tracking Through a Single Element	303
28.3 Tracking Through a Lattice Branch	303
28.4 Forking from Branch to Branch	305
28.5 Multi-turn Tracking	306
28.6 Closed Orbit Calculation	307

28.7	Partial Tracking through elements	307
28.8	Apertures	307
28.9	Tracking Methods	308
28.10	Time Tracking	308
28.11	Taylor Maps	308
28.12	Reverse Tracking	309
28.13	Beam (Particle Distribution) Tracking	310
28.14	Spin Tracking	310
28.15	X-ray Targeting	311
29	Miscellaneous Programming	313
29.1	Custom Elements and Custom Calculations	313
29.2	Physical and Mathematical Constants	316
29.3	Global Coordinates and S-positions	317
29.4	Reference Energy and Time	317
29.5	Common Structures	318
30	Etienne Forest's PTC/FPP	319
30.1	Accessing PTC	320
30.2	Phase Space	320
30.3	Initialization	321
30.4	Correspondence between Bmad elements and PTC fibres	321
30.5	Taylor Maps	321
30.6	Patches	322
30.7	Number of Integration Steps	322
31	OPAL	323
31.1	Phase Space	323
32	C++ Interface	325
32.1	C++ Classes and Enums	325
32.2	Conversion Between Fortran and C++	326
33	Quick_Plot Plotting	329
33.1	An Example	331
33.2	Plotting Coordinates	332
33.3	Length and Position Units	333
33.4	Y2 and X2 axes	334
33.5	Text	334
33.6	Styles	335
33.7	Structures	338
34	Helper Routines	341
34.1	Nonlinear Optimization	341
34.2	Matrix Manipulation	341
35	Bmad Library Routine List	343
35.1	Beam: Low Level Routines	344
35.2	Beam: Tracking and Manipulation	345
35.3	Branch Handling Routines	346
35.4	Coherent Synchrotron Radiation (CSR)	346
35.5	Collective Effects	346

CONTENTS	13
----------	----

35.6	Electro-Magnetic Fields	346
35.7	Helper Routines: File, System, and IO	347
35.8	Helper Routines: Math (Except Matrix)	348
35.9	Helper Routines: Matrix	349
35.10	Helper Routines: Miscellaneous	349
35.11	Helper Routines: String Manipulation	350
35.12	Inter-Beam Scattering (IBS)	351
35.13	Lattice: Element Manipulation	351
35.14	Lattice: Geometry	352
35.15	Lattice: Informational	353
35.16	Lattice: Low Level Stuff	355
35.17	Lattice: Manipulation	355
35.18	Lattice: Miscellaneous	357
35.19	Lattice: Reading and Writing Files	357
35.20	Matrices	358
35.21	Matrix: Low Level Routines	359
35.22	Measurement Simulation Routines	360
35.23	Multipass	360
35.24	Multipoles	360
35.25	Nonlinear Optimizers	361
35.26	Overloading the equal sign	361
35.27	Particle Coordinate Stuff	362
35.28	Photon Routines	362
35.29	Interface to PTC	363
35.30	Quick Plot Routines	364
35.30.1	Quick Plot Page Routines	364
35.30.2	Quick Plot Calculational Routines	364
35.30.3	Quick Plot Drawing Routines	365
35.30.4	Quick Plot Set Routines	366
35.30.5	Informational Routines	368
35.30.6	Conversion Routines	368
35.30.7	Miscellaneous Routines	369
35.30.8	Low Level Routines	369
35.31	Spin Tracking	370
35.32	Transfer Maps: Routines Called by make_mat6	371
35.33	Transfer Maps: Complex Taylor Maps	371
35.34	Transfer Maps: Taylor Maps	372
35.35	Tracking and Closed Orbit	374
35.36	Tracking: Low Level Routines	376
35.37	Tracking: Mad Routines	376
35.38	Tracking: Routines called by TRACK1	377
35.39	Twiss and Other Calculations	378
35.40	Twiss: 6 Dimensional	379
35.41	Wake Fields	380
35.42	Deprecated	380

IV Bibliography and Index	381
---------------------------	-----

Bibliography	383
--------------	-----

Routine Index	387
Index	393

List of Figures

1.1	Superposition example.	23
3.1	Coordinate systems for (a) <code>rbend</code> and (b) <code>sbend</code> elements.	42
3.2	True Rbend coordinates	44
3.3	Crystal element geometry.	47
3.4	Example with photon_fork elements.	56
3.5	Girder example.	58
3.6	Patch Element.	70
4.1	Geometry of Pitch and Offset attributes	87
4.2	Geometry of a Tilt	87
4.3	Geometry of a Bend	88
4.4	Geometry of a photon reflecting element orientation	89
4.5	Apertures for ecollimator and rcollimator elements.	91
4.6	Surface curvature geometry.	96
4.7	Capillary or vacuum chamber wall.	99
4.8	Convex cross-sections do not guarantee a convex volume.	101
4.9	vacuum chamber crotch geometry.	103
4.10	Example diffraction_plate element	104
7.1	Superposition example.	133
7.2	Superposition Offset.	134
11.1	Injection line into a dipole magnet.	157
11.2	Example Energy Recovery Linac.	159
11.3	Patching between reversed and non-reversed elements.	159
11.4	Dual ring colliding beam machine	161
13.1	The local Reference System.	167
13.2	Element LEGO blocks.	168
13.3	Element LEGO block concatenation.	169
13.4	The Global Coordinate System	170
13.5	Orientation of a Bend.	172
13.6	Mirror and crystal geometry	172
13.7	Interpreting phase space z at constant velocity.	175
16.1	CSR Calculation	197
19.1	Element Coordinate System.	208
19.2	ElSeparator electric field.	217

19.3	Standard patch transformation.	219
19.4	Solenoid with a hard edge.	223
20.1	Crystal, Mirror, and Multilayer_Mirror Element Coordinates.	228
20.2	Reference trajectory reciprocal space diagram for crystal diffraction.	232
20.3	Reference energy flow for Laue diffraction	236
20.4	Reflection from a crystal surface.	237
21.1	General diagram of a phase lock loop.	243
21.2	Flow chart of tune tracker module functions.	245
21.3	Plot of VCO response of typical tune tracker setup.	249
23.1	Example Bmad program	264
23.2	Output from the example program	267
24.1	The <code>ele_struct</code> (part 1).	269
24.2	The <code>ele_struct</code> (part 2).	270
25.1	Definition of the <code>lat_struct</code> .	281
25.2	Definition of the <code>param_struct</code> .	285
25.3	Example of multipass combined with superposition	287
28.1	Condensed track_all code.	305
30.1	PTC structure relationships	320
32.1	Example Fortran routine calling a C++ routine.	326
32.2	Example C++ routine callable from a Fortran routine.	326
33.1	<i>Quick Plot</i> example program.	330
33.2	Output of plot_example.f90.	331
33.3	A Graph within a Box within a Page.	333
33.4	Continuous colors using the function <code>pg_continuous_color</code> in PGPlot and PLPlot. Typical usage: <code>call qp_routine(..., color = pg_continuous_color(0.25_rp), ...)</code>	335

List of Tables

2.1	Physical units used by <i>Bmad</i>	26
2.2	Physical and mathematical constants recognized by <i>Bmad</i>	26
3.1	Table of element types suitable for use with relativistic particles.	37
3.2	Table of element types suitable for use with photons.	38
3.3	Table of controller elements.	38
4.1	Table of dependent variables.	83
4.2	Dependent variables that can be set in a primary lattice file.	84
4.3	Field and Strength Attributes.	84
5.1	Table of available tracking_method switches for a given element class.	118
5.2	Table of available mat6_calc_method switches for a given element class.	121
5.3	Table of available spin_tracking_method switches for a given element class.	123
14.1	F and n_{ref} for various elements.	181
21.1	Effect on VCO response of increasing K_P , K_I , or K_D	248
25.1	Bounds of the root branch array.	283
25.2	Possible element %lord_status/%slave_status combinations.	286
33.1	Plotting Symbols at Height = 40.0	336
33.2	PGPLOT Escape Sequences.	337
33.3	Roman to Greek Character Conversion	338

Part I

Language Reference

Chapter 1

Bmad Concepts and Organization

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

1.1 Lattice Elements

The basic building block *Bmad* uses to describe a machine is the `lattice element`. An element can be a physical thing that particles travel “through” like a bending magnet, a quadrupole or a Bragg crystal, or something like a `marker` element (§3.26) that is used to mark a particular point in the machine. Besides physical elements, there are `controller` elements (Table 3.3) that can be used for parameter control of other elements.

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

In a lattice `branch` (§1.2), The ordered array of elements are assigned a number (the element index) starting from zero. The zeroth `beginning_ele` (§3.3) element, which is always named BEGINNING, is automatically included in every branch and is used as a marker for the beginning of the branch. Additionally, every branch will, by default, have a final marker element (§3.26) named END.

1.2 Lattice Branches

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

Chapter §6 shows how a `branch` is defined in a lattice file with `line`, `list`, and `use` statements.

A `lattice` (§1.3), has an array of `branches`. Each `branch` in this array is assigned an index starting from 0. Additionally, each `branch` is assigned a name which is the `line` that defines the branch (§6.6).

1.3 Lattice

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

Branches can be interconnected using **fork** and **photon_fork** elements (§3.18). This is used to simulate forking beam lines such as a connections to a transfer line, dump line, or an X-ray beam line. The **branch** from which other **branches** fork is called a **root branch**.

A lattice may contain multiple **root branches**. For example, a pair of intersecting storage rings will generally have two **root branches**, one for each ring. The **use** statement (§6.6) in a lattice file will list the **root branches** of a lattice. To connect together lattice elements that are physically shared between branches, for example, the interaction region in colliding beam machines, **multipass** lines (§7.2) can be used.

The root branches of a lattice are defined by the **use** (§6.6) statement. To further define such things as dump lines, x-ray beam lines, transfer lines, etc., that branch off from a root branch, a forking element is used. **Fork** elements can define where the particle beam can branch off, say to a beam dump. **photon_fork** elements can define the source point for X-ray beams. Example:

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

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

Branches are named after the line that defines the **branch**. In the above example, the branch line would be named **d_line**. The root branch, by default, is called after the name in the **use** statement (§6.6).

The “branch qualified” name of an element is of the form

```
branch_name>>element_name
```

where **branch_name** is the name of the branch and **element_name** is the “regular” name of the element. Example:

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

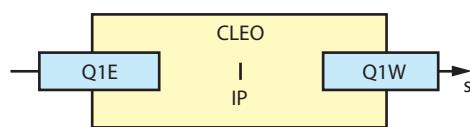
When parsing a lattice file, branches are not formed until the lattice is expanded (§2.19). Therefore, an **expand_lattice** statement is required before branch qualified names can be used in statements. See §2.8 for more details.

1.4 Lord and Slave Elements

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

Bmad tries to capture these interdependencies using what are referred to as **lord** and **slave** elements. The **lord** elements may be divided into two classes. In one class are the **controller** elements. These

A) Physical Layout:



B) Bmad Representation:

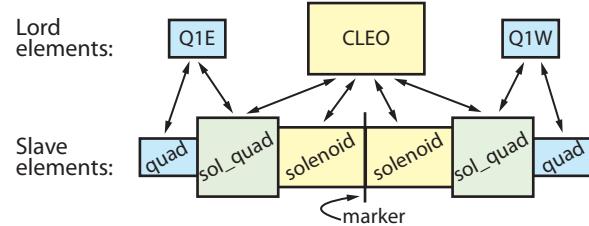


Figure 1.1: Superposition Example. A) Interaction region layout with quadrupoles overlapping a solenoid. B) The Bmad lattice representation has a list of split elements to track through and the undivided “lord” elements. Pointers (double headed arrows), keep track of the correspondence between the lords and their slaves.

are **overlay** (§3.33), **group** (§3.20), and **girder** (§3.19) elements that control the attributes of other elements which are their slaves.

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

An example will make this clear. **Superposition** (§7.1) is the ability to overlap lattice elements spatially. Fig. 1.1 shows an example which is a greatly simplified version of the IR region of Cornell’s CESR storage ring when CESR was an e+/e- collider. As shown in Fig. 1.1A, two quadrupoles named q1w and q1e are partially inside and partially outside the interaction region solenoid named cleo. In the lattice file, the IR region layout is defined to be

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

The line named **cesr** ignores the solenoid and just contains the interaction point marker element named **ip** which is surrounded by two drifts named **dft1** which are, in turn, surrounded by the **q1w** and **q1e** quadrupoles. The solenoid is added to the layout on the second line by using superposition. The “ref = ip” indicates that the solenoid is placed relative to **ip**. The default, which is used here, is to place the center of the superimposed **cleo** element at the center of the **ip** reference element. The representation of the lattice in **Bmad** will contain two branch **sections** (“sections” is explained more fully later): One section, called the **tracking section**, contains the elements that are needed for tracking particles. In the current example, as shown in Fig. 1.1B, the first IR element in the tracking section is a quadrupole that represents the part of **q1e** outside of the solenoid. The next element is a combination solenoid/quadrupole, called a **sol_quad**, that represents the part of **q1e** inside **cleo**, etc. The other branch section that **Bmad** creates is called the **lord section**. This section contains the undivided “physical” **super_lord** elements (§7.1) which, in this case are **q1e**, **q1w**, and **cleo**. Pointers are created between the lords and their **super_slave** elements in the tracking section so that changes in parameters of the lord elements can be transferred to their corresponding slaves.

super_lords are used when there are overlapping fields between elements, the other case where there is a separation between the physical element and the particle track comes when a particle passes through the same physical element multiple times such as in an Energy Recovery Linac or where different beams pass through the same element such as in an interaction region. In this case, **multipass_lords** representing the physical element and **multipass_slaves** representing the track can be constructed (§7.2). Superposition and multipass can be combined in situations where there are overlapping fields in elements where the particle passes through

For historical reasons, each **branch** in a lattice has a **tracking section** and a **lord section** and the **tracking section** is always the first (lower) part of the element array and the **lord section** inhabits

the second (upper) part of the array. All the `lord` elements are put in the `lord` section of branch 0 and all the other `lord` sections of all the other branches are empty.

As a side note, Etienne Forest’s PTC code (§1.5) uses separate structures to separate the physical element, which PTC calls an `element` from the particle track which PTC call a `fibre`. [Actually, PTC has two structures for the physical element, `element` and `elementp`. The latter being the “polymorph” version.] This `element` and `fibre` combination corresponds to *Bmad multipass_lord* and *multipass_slave* elements. PTC does not handle overlapping fields as *Bmad* does with `superposition` (§7.1).

1.5 PTC: Polynomial Tracking Code

Etienne Forest[Forest98] has written what is actually two software libraries: FPP and PTC. 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. For the purposes of this manual, PTC and FPP are generally considered one package and the combined PTC/FPP will be referred to as simply “PTC”. For programmers, interface documentation can be found in chapter §30.

Bmad interfaces to PTC in two ways: One way, called “single element” mode, uses PTC on a per element basis. In this case, the method used for tracking a given element can be selected on an element-by-element basis so non-PTC tracking methods can be mixed with PTC tracking methods to optimize speed and accuracy. [PTC tends to be accurate but slow.] The advantage of single element mode is the flexibility it affords. The disadvantage is that it precludes using PTC’s analysis tools which rely on the entire lattice being tracked via PTC. Such tools include normal form analysis beam envelope tracking, etc.

The alternative to single element mode is “whole lattice” mode where a series of PTC `layouts` (equivalent to a *Bmad* branch) are created from a *Bmad* lattice. Whether single element or whole lattice mode (or both) is used is determined by the program being run.

Chapter 2

Lattice File Overview

A lattice (§1) defines the sequence of elements that a particle will travel through along with the attributes (length, strength, orientation, etc.) of the elements. A lattice file (or files) is a file that is used to describe an accelerator or storage ring.

2.1 Bmad Lattice File Format

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.). Since *Bmad* is a software library, 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 routines but, if it does, the following chapters describe how to construct a valid lattice file.

2.2 MAD, SAD, and XSIF Lattice Files

Besides being able to parse *Bmad* lattice files, *Bmad* has software to parse XSIF[[Tenen01](#)] lattice files. See §12.2 for more details.

While *Bmad* cannot directly read in *MAD* [[Grote96](#)] or *SAD*[[SAD](#)] files, translation between *MAD* and *Bmad* lattice files is possible using the *Universal Accelerator Parser* as discussed in Chapter §12.

2.3 Units and Constants

Bmad uses SI (Système International) units as shown in Table 2.1.

Bmad uses SI (Système International) units as shown in Table 2.1. Note that *MAD* uses different units. For example, *MAD*’s unit of Particle Energy is GeV not eV.

Bmad defines commonly used physical and mathematical constants shown in Table 2.2. All symbols use straight SI units except for `e_mass` and `p_mass` which are provided for compatibility with *MAD*.

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

Table 2.1: Physical units used by *Bmad*.

<i>Symbol</i>	<i>Value</i>	<i>Units</i>	<i>Name</i>
pi	3.14159265359		
twopi	$2 * \text{pi}$		
fourpi	$4 * \text{pi}$		
e_log	2.718281828		
sqrt_2	1.4142135623731		
degrad	$180 / \text{pi}$		From rad to deg
degrees	$\text{pi} / 180$		From deg to rad
raddeg	$\text{pi} / 180$		From deg to rad
m_electron	$0.51099906 \cdot 10^6$	eV	Electron mass
m_muon	$0.1056583893 \cdot 10^6$	eV	Muon mass
m_pion_0	$134.9766 \cdot 10^6$	eV	π^0 mass
m_pion_charged	$139.57018 \cdot 10^6$	eV	π^+, π^- 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 / 2π
e_mass	$0.51099906 \cdot 10^{-3}$	GeV	Electron mass
p_mass	0.938271998	GeV	Proton mass

Table 2.2: Physical and mathematical constants recognized by *Bmad*.

2.4 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. An exclamation mark (!) denotes a comment and the exclamation mark and everything after the exclamation mark on a line are ignored. *Bmad* is generally case insensitive. Most names are converted to uppercase. Exceptions where a name is not converted include file names and atomic formulas for materials used in crystal diffraction.

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 the statement except for the last line. Additionally, lines that end with an “implicit continuation character” are automatically continued to the next line. The implicit continuation characters are

```
,   (   {   [   =
```

Notice that this is *not* like C/C++. Thus the following is bad syntax

```
wall = {
    section = {s = 0.45}      ! BAD SYNTAX. NO CONTINUATION CHARACTER HERE.
}                                ! BAD SYNTAX. NO CONTINUATION CHARACTER HERE.
```

Correct is:

```
wall = {
    section = {s = 0.45} }
```

or even:

```
wall = {
    section = {s = 0.45} &
}
```

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 (§7.1) and dots (.) are used by *Bmad* when creating names of tagged elements (§6.7). 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[geometry] = 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[geometry]` (§8.1) 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 (§8.4). A quadrupole named `q` and a drift element named `d` are specified and the entire lattice consists of element `q` followed by element `d`.

2.5 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 lattice files will be reread and a new digested file will be generated.

Note: If any of the random number functions (§2.13) are used in the process of creating the lattice, the digested file will be ignored. In this case, each time the lattice is read into a program, different random numbers will be generated for expressions that use such random numbers.

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.

2.6 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 §7.1 for more details.

2.7 Lattice Elements

The syntax for defining a lattice element roughly follows the MAD [Grote96] program:

```
ele_name: keyword [, attributes]
```

where `ele_name` is the element name, `keyword` is the type of element, and `attributes` is a list of the elements attributes. Chapter 3 gives a list of elements types with their attributes. `Overlay` and `group` type elements have a slightly different syntax:

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

and Girder elements have the syntax

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

For example:

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

2.8 Lattice Element Names

A valid element name may be up to 40 characters in length. The first character of the name must be a letter [A-Z]. After that, the rest of the name can contain only letters, digits [0-9], underscore “_”, period “.”, backslash “\”, or a hash mark “#”. It is best to avoid these last three symbols since *Bmad* uses them to denote “relationships”. Periods are used for tagging (§6.7), and backslash and hash marks are used for to compose names for superposition (§7.1) and multipass (§7.2) slave elements.

There is a short list of names that cannot be used as an element name. These reserved names are:

```
beam
beam_start
beginning
end
parameter
root
```

Where appropriate, for example when setting element attributes (§2.9), the wildcards “*” and “%” can be used to select multiple elements. The “*” character will match any number of characters (including zero) while “%” matches to any single character. Additionally, matching can be restricted to a certain element class using the syntax:

```
class::element_name
```

where **class** is a class name. For example:

```
m*           ! Match to all elements whose name begins with "m".
a%c          ! Match to "abc" but not to "ac" or "azzc".
quadrupole:::w ! Match to all quadrupoles whose name ends in "w"
```

After lattice expansion (§2.19), the general syntax to specify a set of elements is:

```
{class::}{branch_id>>}element_id{##N}
```

where {...} marks an optional component, **class** is a class name, **branch_id** is a branch name or index (§1.2), **element_id** is an element name or element index (§6.2), and ##N indicates that the Nth matching element is to be used. Examples:

```
quad::x_br>>q*           ! All quadrupoles of branch "x_br" whose name begins with "q".
2>>45                      ! element #45 of branch #2.
q01##3                      ! The 3rd element in each branch named q01.
```

Multiple elements in a lattice may share the same name. When multiple branches are present, to differentiate elements that appear in different branches, the “branch qualified” element name may be used. The branch qualified element name is of the form

```
branch_name>>element_name
```

where **branch_name** is the name of the branch and **element_name** is the “regular” name of the element. Example:

```
root>>q10w
x_branch>>crystal3
```

For **branch** lines (§1.2), the full “branch qualified” name of an element is of the form

```
branch_name>>element_name
```

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

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

Using the full name is only needed to distinguish elements that have the same regular name in separate branches. When parsing a lattice file, branches are not formed until the lattice is expanded (§2.19). Therefore an `expand_lattice` statement is required before full names can be used in statements.

2.9 Lattice 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

```
class::element_name[attribute_name]
```

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

```
bo1w[roll] = 6.5          ! Set an attribute value.
bo1w[1] = 6.5            ! Change an attribute value.
bo1w[1] = bo1w[rho] / 12   ! OK to reset an attribute value.
my_const = bo1w[rho] / bo1w[1] ! Use of attribute values in an expression.
```

Notice that there can be no space between the element name and the [opening bracket.

Chapter Chapter 3 lists the attributes appropriate for each element class.

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. For example

```
q_arc[k1] = 0.234          ! Set all elements named Q_ARC.
rfcavity::*[voltage] = 3.7    ! Set all RFcavity elements.
```

A prepended plus sign “+” indicates that if no elements are matched to, this is not an error. For example

```
+sextupole::*[tracking_method] = taylor
```

In this case, it is not an error if there are no sextupoles in the lattice.

The wild cards “*”, and “%” can be used to can be used (§2.8). Examples:

```
*[tracking_method] = bmad_standard ! Matches all elements.
quadrupole::Q*[k1] = 0.234      ! Matches all quadrupoles with names beginning with Q.
Q%1[k1] = 0.234                ! Matches to "Q01" but not "Q001".
```

Note: A name with wild cards will never match to the BEGINNING element (§6.6).

After lattice expansion (§2.19), the attributes of specific elements may be set using the syntax as discussed in Section §2.8. Example:

```
expand_lattice           ! Expand the lattice.
97[x_offset] = 0.0023     ! Set x_offset attribute of 97th element
b2>>si_cryst##2[tilt] = 0.1 ! Tilt the 2nd instance of "si_cryst" in branch "b2"
```

2.10 Custom Element Attribute Names

Real scalar custom element attributes may be defined for any class of element. Custom element attributes are useful with programs that need to associate “extra” information with particular lattice elements and it is desired that this extra information be settable from within a lattice file. For example, a program might need error tollerances for the strength of quadrupoles.

Adding custom attributes will not disrupt programs that are not designed to use the custom attributes. Currently, up to five custom attributes may be defined for any given element type. The syntax for defining custom attributes is:

```
parameter[custom_attributeN] = "{class::}attribute_name"
custom_attributeN may be one of:
  custom_attribute1
  ...
  custom_attribute5
```

and "attribute_name" is the name of the attribute. To restrict the custom attribute to a particular element class, the element class can be prefixed to the attribute name. Examples:

```
parameter[custom_attribute1] = "mag_id"
parameter[custom_attribute1] = "quadrupole::error_k1"
parameter[custom_attribute2] = "color"
```

The first line in the example assigns a custom attribute name of `mag_id` to all elements. The second line in the example overrides the setting of `custom_attribute1` for quadrupole elements only.

Once a custom attribute has been defined it may be set for an element of the correct type. Example:

```
parameter[custom_attribute2] = "lcavity::rms_phase_err"
...
12a: lcavity, rms_phase_err = 0.0034, ...
```

For someone creating a program, section §24.18 describes how to make the appropriate associations.

Note: If custom string information needs to be associated with an element, the `type`, `alias` and `descrip` element components (§4.2) are available.

2.11 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.

2.12 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
a / b	Division
$a \wedge b$	Exponentiation

Bmad also has a set of intrinsic functions. A list of these is given in §2.13.

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.
```

Another potential pitfall with immediate substitution is when using dependent element attributes (§4.1). For example:

```
b01w: sbend, l = 0.5, angle = 0.02
a_const = b01w[g]      ! No: bend g has not yet been computed!
```

Here the bend strength `g` (§3.5) will eventually be computed to be 0.04 (= `angle / l`) but that computation does not happen until lattice expansion (§2.19). In this case, the value of `a_const` will be the default value of `g` which is zero. As a rule of thumb, never rely on dependent attributes having their correct value.

2.13 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>atan2(y, x)</code>	Arc Tangent of y/x
<code>abs(x)</code>	Absolute Value
<code>factorial(n)</code>	Factorial
<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 (§2.19). 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`.

2.14 Print Statement

The `print` statement prints a message at the terminal when the lattice file is parsed by a program. Syntax:

```
print <String>
```

For example

```
print Remember! Q01 quad strength not yet optimized!
```

The `print` statement is useful to remind someone using the lattice of important details.

2.15 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"
```

2.16 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:

- 1) `../layout/my_layout.bmad` (relative to the calling file directory)
- 2) `../layout/my_layout.bmad` (relative to the current directory)

An XSIF (§2.1) lattice file may be called from within a *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 comments or debugging statements (§2.20). 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 (§2.19) and any additional statements in the *Bmad* lattice file operate on the expanded lattice.

2.17 Inline Call

Any lattice elements will have a set of attributes that need to be defined. As a convenience, it is possible to segregate an element attribute or attributes into a separate file and then “call” this file using an “inline call”. The inline call has three forms. In an element definition, the inline call has the form

```
<ele_name>: <ele_type>, ..., call::<file_name>, ...
```

or

```
<ele_name>: <ele_type>, ..., <attribute_name> = call::<file_name>, ...
```

where `<attribute_name>` is the name of the attribute and `<file_name>` is the name of the where the attribute structure is given. The third form of the inline call occurs when an element attribute is redefined and has the form

```
<ele_name>[<attribute_name>] = call::<file_name>
```

Example:

```
c: crystal, call::my_curvature.bmad, surface = call::my_surface.bmad, ...
```

2.18 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.

2.19 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 §2.13.
- All `lines` (§6.2), `lists` (§6.5), and `use` (§6.6) statements must come before lattice expansion.
- Some dependent variables may be set as if they are independent variables but only if done before lattice expansion. See §4.1.
- Setting the `phi0_multipass` attribute for an `Lcavity` or `RFcavity` multipass slave may only be done after lattice expansion (§7.2).
- Setting individual element attributes for tagged elements can only be done after lattice expansion (§6.7).

Lattice expansion is only done once so it is an error if multiple `expand_lattice` statements are present.

A lattice file where all the statements are post lattice expansion valid is called a “`secondary lattice file`”. To promote flexibility, *Bmad* has methods for parsing lattices in a two step process: First, a “primary” lattice file that defines the basic lattice is read. After the primary lattice has been parsed and lattice expansion has been done, the second step is to read in one or more secondary lattice files. Such secondary lattice files can be used, for example, to set such things as element misalignments. The point here is that there are no calls (§2.16) of the secondary files in the primary file so the primary lattice file does not have to get modified when different secondary files are to be used.

2.20 Debugging Statements

There are a few statements which can help in debugging the *Bmad* lattice parser itself. That is, these statements are generally only used by programmers. These statements are:

```
debug_marker
no_digested
no_superimpose
parser_debug
```

The `debug_marker` statement is used for marking a place in the lattice file where program execution is to be halted. This only works when running a program in conjunction with a program debugging tool.

The `no_digested` statement if present, will prevent *Bmad* from creating a digested file (§2.5). That is, the lattice file will always be parsed when a program is run.

The `no_superimpose` statement is used to suppress superpositions (§7.1). This is useful for debugging purposes.

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

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

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

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

Chapter 3

Elements

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

<i>Element</i>	<i>Section</i>	<i>Element</i>	<i>Section</i>
AB_Multipole	3.1	Monitor	3.22
BeamBeam	3.2	Multipole	3.29
Beginning_Ele	3.3	Null_Ele	3.31
Bend_Sol_Quad	3.4	Octupole	3.32
Custom	3.9	Patch	3.34
Drift	3.12	Photon_Fork	3.18
E_Gun	3.13	Pipe	3.22
Ecollimator	3.7	Quadrupole	3.35
ElSeparator	3.14	Rbend	3.5
EM_Field	3.15	Rcollimator	3.7
Fiducial	3.16	RFcavity	3.36
Floor_Shift	3.17	Sad_Mult	3.37
Fork	3.18	Sbend	3.5
HKicker	3.23	Sextupole	3.39
Hybrid	3.21	Sol_Quad	3.41
Instrument	3.22	Solenoid	3.40
Kicker	3.24	Taylor	3.42
Lcavity	3.25	VKicker	3.23
Marker	3.26	Wiggler	3.43
Match	3.27		

Table 3.1: Table of element types suitable for use with relativistic particles.

Most element types available in *MAD* are provided in *Bmad*. Additionally, *Bmad* provides a number of element types that are not available in *MAD*. A word of caution: In some cases where both *MAD* and *Bmad* provide the same element type, there will be an overlap of the attributes available but the two sets of attributes will not be the same. The list of element types known to *Bmad* is shown in Table 3.1, 3.2, and 3.3. Table 3.1 lists the elements suitable for use with relativistic particles, Table 3.2 which lists the elements suitable for use with photons, and finally Table 3.3 lists the controller element types that can be used for parameter control of other elements. Note that some element types are suitable for both

particle and photon use.

<i>Element</i>	<i>Section</i>	<i>Element</i>	<i>Section</i>
Beginning_Ele	3.3	Marker	3.26
Capillary	3.6	Match	3.27
Crystal	3.8	Monitor	3.22
Custom	3.9	Mirror	3.28
Detector	3.10	Multilayer_Mirror	3.30
Diffraction_Plate	3.11	Patch	3.34
Drift	3.12	Photon_Fork	3.18
Fiducial	3.16	Pipe	3.22
Floor_Shift	3.17	Sample	3.38
Fork	3.18	X_Ray_Init	3.44
Instrument	3.22		

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

<i>Element</i>	<i>Section</i>	<i>Element</i>	<i>Section</i>
Group	3.20	Overlay	3.33
Girder	3.19		

Table 3.3: Table of controller elements.

3.1 AB_Multipole

An `ab_multipole` is a thin multipole lens up to 20th order. The basic difference between this and a `multipole` (§3.29 is the input format. See section §14.1 for how the multipole coefficients are defined.

General `ab_multipole` Attributes are:

<i>Attribute Class</i>	§	<i>Attribute Class</i>	§
<code>an, bn</code> multipoles	4.12	Offsets & tilt	4.4
Aperture Limits	4.6	Length	4.10
Chamber wall	4.9	Is_on	4.11
Description strings	4.2	Tracking & transfer map	5
Reference energy	4.3		

For `an` and `bn`, n is in the range 0 through 20.

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. The `x_pitch` and `y_pitch` attributes are not used in tracking.

Unlike a `multipole`, an `ab_multipole` will *not* 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
```

3.2 BeamBeam

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[[Talman87](#)]

General `beambeam` attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	Offsets, pitches, & tilt	4.4
Chamber wall	4.9	Is_on	4.11
Description strings	4.2	Tracking & transfer map	5
Reference energy	4.3		

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       ! Dependent attribute (§4.1).

```

`n_part` is the nominal number of particles of the strong beam. `n_part` is set using the `parameter` command ([§8.1](#)) 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. See [§19.5](#) for details on how a `beambeam` element is tracked.

The `bbi_constant` is a measure of the beam-beam interaction strength. It is a dependent variable and is calculated from the equation

$$C_{bbi} = N m_e r_e / (2 \pi \gamma (\sigma_x + \sigma_y)) \quad (3.1)$$

In the linear region, near $x = y = 0$, the beam-beam kick is approximately

$$\begin{aligned} k_x &= -4 \pi x C_{bbi} / \sigma_x \\ k_y &= -4 \pi y C_{bbi} / \sigma_y \end{aligned} \quad (3.2)$$

The beam-beam tune shift is

$$\begin{aligned} dQ_x &= C_{bbi} \beta_x / \sigma_x \\ dQ_y &= C_{bbi} \beta_y / \sigma_y \end{aligned} \quad (3.3)$$

Example:

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

3.3 Beginning_Ele

An `beginning_ele` element, named `BEGINNING`, is placed at the beginning of every branch (§1.2) of a lattice to mark the start of the branch. The `beginning_ele` always has element index 0 (§1).

`beginning_ele` attributes are generally set using either `parameter` (§8.1) or `beginning` (§8.4) statements. In particular the initial energy may be set (§4.3).

3.4 Bend_Sol_Quad

A `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.

General `bend_sol_quad` attributes are:

Attribute Class	Section	Attribute Class	Section
<code>an, bn</code> multipoles	4.12	Integration settings	5.4
Aperture Limits	4.6	<code>Is_on</code>	4.11
Chamber wall	4.9	Length	4.10
Description strings	4.2	Offsets, pitches, & tilt	4.4
Field table or map	4.13	Reference energy	4.3
Fringe Fields	4.16	Symplectify	5.5
Hkick & Vkick	4.5	Tracking & transfer map	5

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

```

The magnetic field is:

$$\begin{aligned}
 \frac{qB_x}{P_0} &= -g_y + k_{1n}(y - y_q) - k_{1s}(x - x_q) - \frac{dks/ds}{2} x \\
 \frac{qB_y}{P_0} &= g_x + k_{1n}(x - x_q) + k_{1s}(y - y_q) - \frac{dks/ds}{2} y \\
 \frac{qB_s}{P_0} &= k_s + dks/ds
 \end{aligned} \tag{3.4}$$

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
```

3.5 Bends: Rbend and Sbend

Rbends and sbends are dipole bends.

General `rbend` and `sbend` attributes are:

Attribute Class	Section	Attribute Class	Section
<code>an, bn</code> multipoles	4.12	Integration settings	5.4
Aperture Limits	4.6	<code>Is_on</code>	4.11
Chamber wall	4.9	Length	4.10
Description strings	4.2	Offsets, pitches, & tilt	4.4
Field table or map	4.13	Reference energy	4.3
Fringe Fields	4.16	Symplectify	5.5
Hkick & Vkick	4.5	Tracking & transfer map	5

Attributes specific to `rbend` and `sbend` elements are:

```
angle      = <Real>      ! Design bend angle. A settable dependent variable (§4.1).
b_field   = <Real>      ! Design field strength (= P_0 g / q) (§4.1).
b_field_err = <Real>    ! Field strength error (§4.1).
b1_gradient = <Real>    ! Quadrupole field strength (§4.1).
b2_gradient = <Real>    ! Sextupole field strength (§4.1).
e1, e2     = <Real>      ! Face angles.
f1         = <Real>      ! SAD ‘‘linear’’ fringe field.
fint, fintx = <Real>    ! Face field integrals.
g          = <Real>      ! Design bend strength (= 1/rho).
g_err      = <Real>      ! Bend strength error (§4.1).
h1, h2     = <Real>      ! Face curvature.
hgap, hgappx = <Real>  ! Pole half gap.
k1         = <Real>      ! Quadrupole strength.
k2         = <Real>      ! Sextupole strength (§4.1).
l          = <Real>      ! ‘‘Length’’ of bend. See below.
l_arc     = <Real>      ! Arc length. For rbends only.
l_chord   = <Real>      ! Chord length. Dependent attribute. See §4.10.
n_ref_pass = <Int>      ! Multipass reference turn (§7.2).
ptc_field_geometry
```

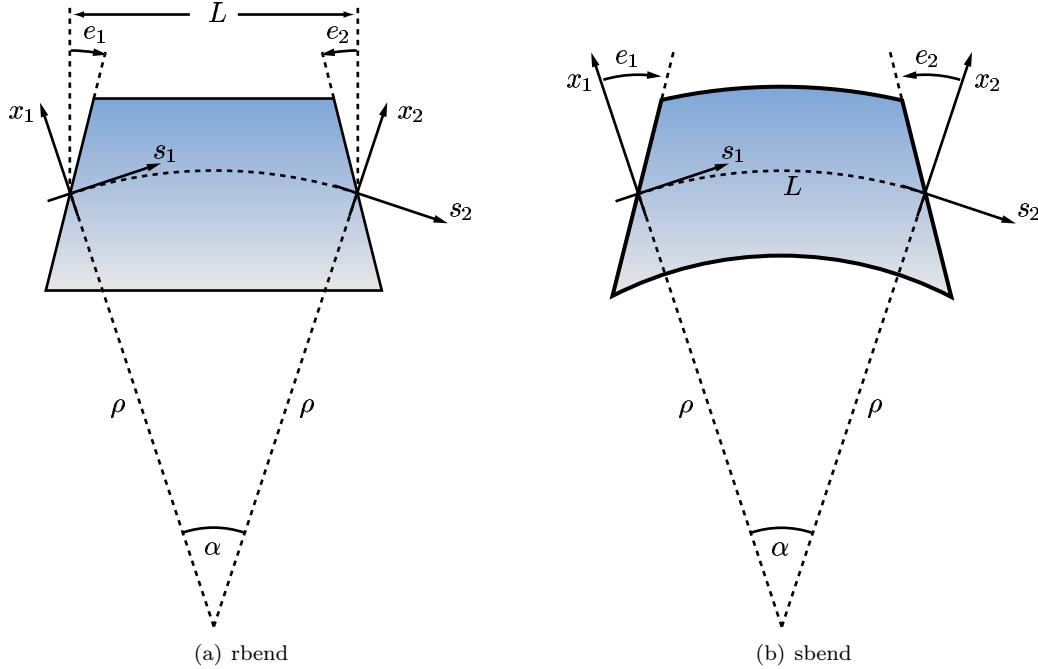


Figure 3.1: Coordinate systems for (a) `rbend` and (b) `sbend` elements. For the bends drawn as viewed from “above” (viewed from positive y), g , angle , ρ , $e1$ and $e2$ are all positive.

```

        = <Switch> ! See below.
rho      = <Real>    ! Design bend radius. A settable dependent variable (§4.1).
roll     = <Real>    ! See 4.4.

```

The difference between `rbend` and `sbend` elements 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 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

```

`angle`

The total design bend angle. A positive `angle` represents a bend towards negative x values (see Fig. 13.1).

`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 (Fig. 3.1a). Zero `e1` and `e2` for an `sbend` gives a wedge shaped magnet (Fig. 3.1b). An `sbend` with an `e1 = e2 = angle/2` is equivalent to an `rbend` with `e1 = e2 = 0` (see above). This formula holds for both positive and negative angles.

`f1`

`f1` is the strength of the SAD “linear” fringe field. This fringe is applied at both the entrance and exit faces.

fint, fintx, hgap, hgapx

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

$$F_{int} = \int_{pole} ds \frac{B_y(s)(B_{y0} - B_y(s))}{2H_{gap}B_{y0}^2} \quad (3.5)$$

with a similar equation for **fintx**. In the equation B_{y0} is the field in the interior of the dipole and H_{gap} is the pole half gap. The parameters **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, 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.

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

$$B_y(s) = \frac{B_{y0}}{1 + \exp[P(s)]} \quad (3.6)$$

where

$$P(s) = C_0 + C_1 s + C_2 s^2 + C_3 s^3 + \dots \quad (3.7)$$

The C_0 term simply shifts where the edge of the bend is. If all the C_n are zero except for C_0 and C_1 then

$$C_1 = \frac{1}{2 H_{gap} F_{int}} \quad (3.8)$$

g, g_err, rho

The design bending radius which determines the reference coordinate system is **rho** (see §13.1). $g = 1/\rho$ is the curvature function and is proportional to the design dipole magnetic field. The true field strength is given by $g + g_{err}$ so changing **g_err** leaves the design orbit unchanged but varies a particle's orbit.

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.

k1, b1_gradient

The normalized and unnormalized quadrupole strength.

k2, b2_gradient

The normalized and unnormalized sextupole strength.

l, l_chord

For compatibility with *MAD*, for an **rbend**, **l** is the chord length and not the arc length as it is for an **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). Alternatively for an **rbend**, instead of setting **l**, the **l_arc** attribute can be set to the true arc length.

ref_tilt

The **ref_tilt** attribute rotates a bend about the longitudinal axis at the entrance face of the bend. **ref_tilt** = 0 bends the bend in the $-y$ direction. See Fig. 13.5. It is important to

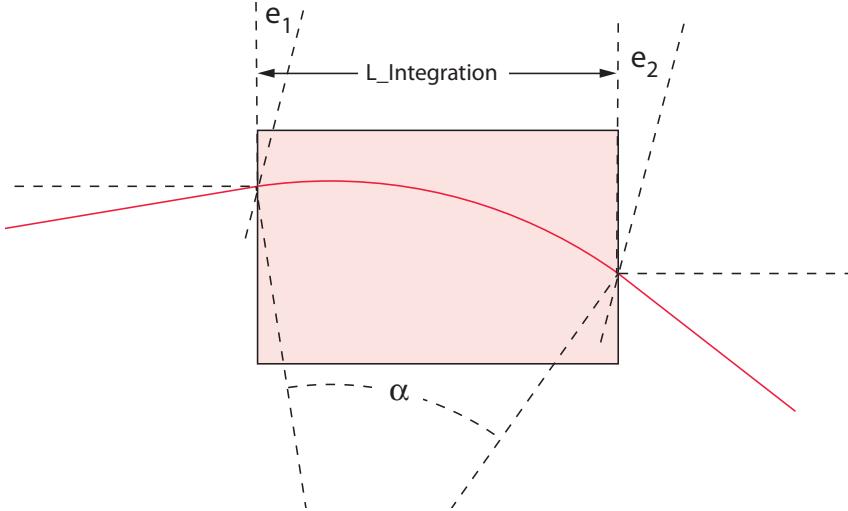


Figure 3.2: Coordinate system when `ptc_field_geometry` is set to `true_rbend`.

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

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 (§3.33) 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`.

The `n_ref_pass` attribute 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 §7.2 for more details.

In the local coordinate system (§13.1), looking from “above” (bend viewed from positive y), and with `ref_tilt` = 0, a positive `angle` represents a particle rotating clockwise. In this case, `g` will also be positive. For counterclockwise rotation, both `angle` and `g` will be negative but the length `l` is always positive. Also, looking from above, a positive e_1 represents a clockwise rotation of the entrance face and a positive e_2 represents a counterclockwise rotation of the exit face. This is true regardless of the sign of `angle` and `g`. Also it is always the case that the pole faces will be parallel when

$$e_1 + e_2 = \text{angle}$$

Example bend specification:

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

`ptc_field_geometry` determines how PTC integrates through a bend if PTC is being used for tracking. Possible values for `ptc_field_geometry` are:

```

sector      ! Default
straight
true_rbend ! Only valid for rbend elements

```

For `sector` tracking, the tracking coordinate reference frame is with respect to the arc of the reference trajectory. For `straight` tracking the tracking coordinate reference frame is with respect to the chord line. For a bend where the number of integration steps is large enough, and where there are no other fields besides the basic dipole field, the results are the same. When there are quadrupole or higher order fields, the fields are expanded about the tracking reference frame. Since Maxwell's equations must be satisfied, the higher order fields will differ when tracking with `sector` vs `straight` the difference in the fields will scale with the inverse of the bending radius $1/\rho$. The above discussion is true for `ptc_exact_model` set to True, for `ptc_exact_model` set to False, a simplified sector tracking model is used in all cases.

The `true_rbend` tracking of `ptc_field_geometry` is used only with `rbend` elements and the entrance and exit faces must be parallel as shown in Fig. 3.2. That is

$$e_1 + e_2 = 0$$

In this case, the tracking geometry is parallel, to the bend face as shown in the figure. This can be an advantage in some situations but Etienne discourages use of `true_rbend` due to complications of how to handle the reference frames in particular when you have more than one of them in a row.

3.6 Capillary

A `capillary` element is a glass tube that is used to focus x-ray beams.

General `capillary` attributes are:

Attribute Class	Section	Attribute Class	Section
Description strings	4.2	Offsets, Pitches & Tilt	4.4
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Is_on	4.11
Capillary Wall	4.9		

Attributes specific to a `capillary` element are:

```
critical_angle_factor = <Real> ! Critical angle * Energy (rad * eV)
```

The critical angle above which photons striking the capillary surface are refracted into the capillary material scales as $1/\text{Energy}$. The constant of critical angle * energy is given by the `critical_angle_factor`.

The inside wall of a capillary is defined using the same syntax used to define the chamber wall for other elements ([§4.9](#)).

The length of the capillary is a dependent variable and is given by the value of `s` of the last wall cross-section ([§4.9.4](#)).

3.7 Collimators: Ecollimator and Rcollimator

An `ecollimator` is a drift with elliptic collimation. An `rcollimator` is a drift with rectangular collimation.

General `ecollimator` and `rcollimator` attributes are:

Attribute Class	Section	Attribute Class	Section
Description strings	4.2	Offsets, Pitches & Tilt	4.4
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.10
Symplectify	5.5	Integration settings	5.4
Hkick & Vkick	4.5	Chamber wall	4.9

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
```

3.8 Crystal

A `crystal` element represents a crystal used for photon diffraction.

General `crystal` attributes are:

Attribute Class	Section	Attribute Class	Section
Description strings	4.2	Offsets, Pitches & Tilt	4.4
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Symplectify	5.5
Surface Properties	4.8		

Attributes specific to a `crystal` element are:

<code>b_param</code>	= <Real>	! b parameter
<code>crystal_type</code>	= <String>	! Crystal material and reflection plane.
<code>psi_angle</code>	= <Real>	! Rotation of H-vector about the surface normal.
<code>thickness</code>	= <Real>	! Thickness of crystal for Laue diffraction.
<code>ref_orbit_follows</code>	= <which_beam>	! Reference orbit aligned with what outgoing beam?

Dependent variables (§4.1) specific to a `crystal` element are:

<code>alpha_angle</code>	!	Angle of H-vector with respect to the surface normal.
<code>bragg_angle</code>	!	Nominal Bragg angle at the reference wave length.
<code>bragg_angle_in</code>	!	Angle between incoming beam and mirror surface.
<code>bragg_angle_out</code>	!	Angle between outgoing beam and mirror surface.
<code>d_spacing</code>	!	Lattice plane spacing.
<code>darwin_width_pi</code>	!	Darwin width for pi polarized light (radians).
<code>darwin_width_sigma</code>	!	Darwin width for sigma polarized light (radians).
<code>dbragg_angle_de</code>	!	Variation of the Bragg angle with energy (radians/eV).
<code>l</code>	!	Length of reference orbit.
<code>pendellosung_period_pi</code>	!	Pendellosung period for pi polarized light.
<code>pendellosung_period_sigma</code>	!	Pendellosung period for sigma polarized light.
<code>ref_wavelength</code>	!	Reference wavelength.
<code>ref_cap_gamma</code>	!	Γ at the reference wavelength.
<code>tilt_corr</code>	!	Tilt correction due to a finite <code>psi_angle</code> .
<code>v_unitcell</code>	!	Unit cell volume.

The `crystal_type` attribute defines the crystal material and diffraction lattice plane. The syntax is "ZZZ(ijk)" where ZZZ is the atomic formula for the material and ijk are the Miller indices for the diffraction plane. For example,

```
b_cryst1: crystal, crystal_type = "Si(111)", b_param = -1, ...
```

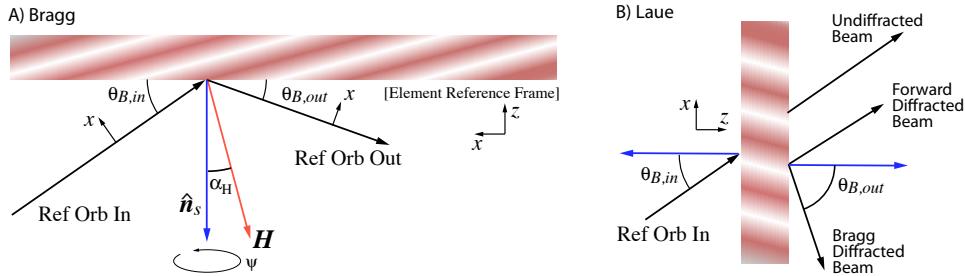


Figure 3.3: Crystal element geometry. A) Geometry for Bragg diffraction. The geometry shown is for `ref_tilt` = 0 (reference trajectory in the x - z plane). The angle α_H (`alpha_angle`) is the angle of the \mathbf{H} vector with respect to the surface normal $\hat{\mathbf{n}}$. For ψ (`psi_angle`) zero, the incoming reference orbit, the outgoing reference orbit, $\hat{\mathbf{n}}$, and \mathbf{H} are all coplanar. B) Geometry for Laue diffraction. In this case there are three outgoing beams: The Bragg diffracted beam, the forward diffracted beam, and the undiffracted beam.

The atomic formula is case sensitive so, for example, "SI(111)" is not acceptable. Given the `crystal_type`, the spacing between lattice planes (`d_spacing`), the unit cell volume (`v_unitcell`), and the structure factor [Bater64] values can be computed. The list of known crystals is given in §4.7.

The `b_param` is the standard asymmetry factor

$$b = \frac{\sin(\alpha_H + \theta_B)}{\sin(\alpha_H - \theta_B)} \quad (3.9)$$

where θ_B is the Bragg angle (`bragg_angle`)

$$\theta_B = \sin^{-1} \left(\frac{\lambda}{2d} \right) \quad (3.10)$$

and α_H (`alpha_angle`) is the angle of the reciprocal lattice \mathbf{H} vector with respect to the surface normal as shown in Fig. 3.3A. If `b_param` is set to -1 then there is Bragg reflection and `alpha_H` is zero. If `b_param` is set to 1 then there is Laue diffraction again with `alpha_H` zero. With the orientation shown in Fig. 3.3A, `alpha_H` is positive.

The `thickness` parameter is used with Laue diffraction only.

The `ref_orbit_follows` parameter sets how the outgoing reference orbit is constructed. This is only relevant with Laue diffraction. The possible settings of this parameter are:

```
bragg_diffracted
forward_diffracted
undiffracted
```

The geometry of this situation is shown in Fig. 3.3B. The reference orbit for the `undiffracted` beam is just a straight line extension of the incoming reference trajectory. This trajectory is that trajectory that photons whose energy is far from the Bragg condition (that is, far from the reference energy) will follow. The `forward_diffracted` reference orbit is parallel to the `undiffracted` trajectory and is the trajectory of the forward diffracted photons whose energy is the reference energy and whose incoming orbit is on the incoming reference trajectory. Finally, the `bragg_diffracted` reference orbit is the backward diffracted orbit.

Note: Changing the setting of `ref_orbit_follows` will change the reference orbit downstream of the crystal which, in turn, will change the placement all downstream elements.

The value of the element reference orbit length \mathbf{l} is calculated by *Bmad*. \mathbf{L} will be zero for Bragg diffraction. For Laue diffraction, \mathbf{l} will depend upon the crystal **thickness** and the setting of **ref_orbit_follows**.

If **psi_angle** is zero, the incoming reference orbit, the outgoing reference orbit, $\hat{\mathbf{n}}$ and \mathbf{H} are all coplanar. A non-zero **psi_angle** Rotates the \mathbf{H} vector around the $+\hat{x}$ axis of the Element Reference Frame (See Fig. 3.3A).

To keep the outgoing reference trajectory independent of the value of **psi_angle**, the crystal will be automatically tilted by the appropriate “tilt correction” **tilt_corr**. The calculation of **tilt_corr** is outlined in §20.4.2. **tilt_corr** will be zero if **psi_angle** is zero.

The reference trajectory for a Bragg **crystal** is that of a zero length bend (§13.3.3) and hence the length (\mathbf{l}) parameter of a crystal is fixed at zero. The orientation of the reference trajectory with respect to the crystal surface is specified by the incoming Bragg angle **bragg_angle_in** ($\theta_{g,in}$) and outgoing Bragg angle **bragg_angle_out** ($\theta_{g,out}$) as shown in Fig. 3.3A. These angles are computed from the photon reference energy and the other crystal parameters such that a photon with the reference energy traveling along the reference trajectory will be in the center of the Darwin curve (§20.4).

The reference trajectory in the global coordinate system (§13.3) is determined by the value of the **ref_tilt** parameter along with the value of **bragg_angle_in** + **bragg_angle_out**. These bragg angles take into account refraction so that the reference trajectory downstream of the crystal will be properly centered with respect to the reference photon. A positive **bragg_angle_in** + **bragg_angle_out** bends the reference trajectory in the same direction as a positive \mathbf{g} for a bend element. The

A **crystal** may be offset and pitched (4.4). The incoming local reference coordinates are used for these misalignments.

When a crystal is bent (§4.8), the \mathbf{H} vector is assumed follow the surface curvature. That is, it is assumed that the lattice planes are curved by the bending.

Example:

```
crystal_ele: crystal, crystal_type = 'Si(111)', b_param = -1
```

The **darwin_width_sigma** and **darwin_width_pi** parameters are the computed Darwin width, in radians, for sigma and pi polarized light respectively. Here the Darwin width $d\theta_D$ is defined as the width at the $\eta = \pm 1$ points (cf. Batterman[Bater64] Eq (32))

$$d\theta_D = \frac{2\Gamma|P|\operatorname{Re}([F_H F_{\bar{H}}]^{1/2})}{|b|^{1/2} \sin \theta_{tot}} \quad (3.11)$$

where

```
 $\theta_{tot} = \text{bragg\_angle\_in} + \text{bragg\_angle\_out}$ 
```

The **pendellosung_period_sigma** and **pendellosung_period_pi** are the pendellosung periods for Laue diffraction. If the crystal is set up for Bragg diffraction then the values for these parameters will be set to zero.

The **dbragg_angle_de** parameter is the variation in Bragg angle with respect to the photon energy and is given by the formula

$$\frac{d\theta_B}{dE} = -\frac{\lambda}{2dE \cos(\theta_B)} \quad (3.12)$$

3.9 Custom

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 §29.1 on how to write custom routines.

General `custom` attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	Is_on	4.11
Chamber wall	4.9	Length	4.10
Description strings	4.2	Offsets, pitches, & tilt	4.4
Field table or map	4.13	Reference energy	4.3
Fringe fields	4.16	Symplectify	5.5
Integration settings	5.4	Tracking & transfer map	5

As an alternative to defining a custom element, standard elements can be “customized” by setting one or more of the following attributes to `custom`:

tracking_method	§5.1
mat6_calc_method	§5.2
field_calc	§5.4
aperture_type	§4.6

As with a custom element, setting one of these attributes to `custom` necessitates the use of custom code to implement the corresponding calculation.

Attributes specific to a `custom` element are

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

`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, descrip = 'params.dat'
```

In this example the `descrip` string is being used to specify a file that contains parameters for the element.

3.10 Detector

A `detector` element is used to detect particles and X-rays. A `detector` is modeled as a plate upon which particles and x-rays can impinge.

General `detector` element attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	Reference energy	4.3
Chamber wall	4.9	Surface Properties	4.8
Description strings	4.2	Tracking & transfer map	5
Offsets, pitches, & tilt	4.4		

3.11 Diffraction _ Plate

A `diffraction_plate` element is a flat surface oriented, more or less, transversely to a x-ray beam through which photon can travel. A `diffraction_plate` can be used, for example, to model a Fresnel

zone plate or Young's double slits. A `diffraction_plate` element is used in places where diffraction effects must be taken into account. This is in contrast to setting an aperture attribute (§4.6 for other elements where diffraction effects are ignored).

General `diffraction_plate` element attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	Offsets, pitches, & tilt	4.4
Chamber wall	4.9	Reference energy	4.3
Description strings	4.2	Tracking & transfer map	5

Attributes specific to a `diffraction_plate` element are:

```
mode          = <type>      ! reflection or transmission
```

The `mode` switch sets whether X-rays are transmitted through the `diffraction_plate` or reflected. Possible values for the `mode` switch are:

```
reflection
transmission      ! Default
```

The geometry of the plate, that is, where the openings (in transmission mode) or reflection regions are, is defined using the "wall" attribute. See (§4.9) for more details.

In transmission mode, a `diffraction_plate` is nominally orientated transversely to the beam. Like all other elements, the `diffraction_plate` can be reoriented using the element's offsets, pitches and tilt attributes (§4.4).

The `aperture_type` (§4.6) parameter of a `diffraction_plate` will default to `auto` which will set the aperture limits to define a rectangular aperture that just cover the clear area of the plate.

Example:

```
fresnel: diffraction_plate, wall = {...}
```

3.12 Drift

A `drift` element is a space free and clear of any fields.

General `drift` attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	Offsets, pitches, & tilt	4.4
Description strings	4.2	Reference energy	4.3
Integration settings	5.4	Symplectify	5.5
Length	4.10	Tracking & transfer map	5

Example:

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

Note: If a chamber wall (§4.9) is needed for a field free space, use a `pipe` element instead of a `drift` [a wall for a drift is not allowed due to the way drifts are treated with superposition. That is, drifts "disappear" when superimposed upon. (§7.1)].

3.13 E_Gun

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

Attribute Class	Section	Attribute Class	Section
Symplectify	5.5	Offsets, pitches, & tilt	4.4
Description strings	4.2	Is_on	4.11
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.10
Hkick & Vkick	4.5	an, bn multipoles	4.12
Integration settings	5.4	Chamber wall	4.9

The attributes specific to an `e_gun` are

```
gradient      = <Real>      ! Gradient.
gradient_err = <Real>      ! Gradient error.
phi0         = <Real>      ! Phase (rad/2π) of the reference particle with
                           ! respect to the RF. phi0 = 0 is on crest.
phi0_err     = <Real>      ! Phase error (rad/2π)
rf_frequency = <Real>      ! Frequency of the RF field.
voltage       = <Real>      ! Voltage. Dependent attribute (§4.1).
voltage_err   = <Real>      ! Voltage error. Dependent attribute (§4.1).
```

The `voltage` is simply related to the `gradient` via the element length `l`:

```
voltage = gradient * l
```

If the `voltage` is set to a non-zero value, the length `l` must also be non-zero to keep the gradient finite.

An `e_gun` may either be DC if the `rf_frequency` component is zero or AC if not.

Electrons generated at the cathode can have zero initial momentum and this presents a special problem (§4.3). As a result, the use of `e_gun` elements are restricted and they can only be used in a “linear” (non-recirculating) lattice branch. Only one `e_gun` can be present in a lattice branch and, if it is present, it must be, except for possibly `marker` or `null_ele` elements, the first element in any branch.

Note: In order to be able to avoid problems with a zero reference momentum at the beginning of the `e_gun`, the reference momentum and energy associated with an `e_gun` element is calculated as outlined in Section §4.3. Additionally, the reference momentum at the exit end of the `e_gun`, that is `p0c`, must be non-zero. Thus, for example, if `p0c` is zero at the start of the lattice, the `e_gun` voltage must be non-zero.

Note: The default `tracking_method` (§5.1) setting for an `e_gun` is `time_runge_kutta` and the default `mat6_calc_method` is `tracking`.

In this example the field of an `e_gun` is given by a grid of field values (§4.13.2):

```
apex: e_gun, l = 0.23, field_calc = grid, rf_frequency = 187e6,
      field = { mode = {
        m = 0, harmonic = 1,
        master_scale = voltage,
        grid = call::apex_gun_grid.bmad }}
```

with the file `apex_gun_grid.bmad` being:

```
{
  type = rotationally_symmetric_rz,
```

```
r0 = (0, 0),
dr = (0.001, 0.001),
pt(0,0) = ( (0, 0), (0, 0), (1, 0), (0, 0), (0, 0), (0, 0)),
pt(0,1) = ( (0, 0), (0, 0), (0.99, 0), (0, 0), (0, 0), (0, 0)),
... }
```

3.14 Elseparator

An `elseparator` is an electrostatic separator.

General `elseparator` attributes are:

Attribute Class	Section	Attribute Class	Section
<code>an</code> , <code>bn</code> multipoles	4.12	Integration settings	5.4
Aperture Limits	4.6	<code>Is_on</code>	4.11
Chamber wall	4.9	Length	4.10
Description strings	4.2	Offsets, pitches, & tilt	4.4
Field table or map	4.13	Reference energy	4.3
Fringe Fields	4.16	Symplectify	5.5
Hkick & Vkick	4.5	Tracking & transfer map	5

Attributes specific to an `elseparator` 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).
```

For an `elseparator`, the kick for a positively charged particle is determined by `hkick` and `vkick`. The kick for a negatively charged particle is opposite this. 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
```

3.15 EM_Field

An `em_field` element can contain general electro-magnetic (EM) fields. Both AC and DC fields are accommodated. General `em_field` attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	<code>Is_on</code>	4.11
Chamber wall	4.9	Length	4.10
Description strings	4.2	Offsets, pitches, & tilt	4.4
Field table or map	4.13	Reference energy	4.3
Hkick & Vkick	4.5	Symplectify	5.5
Integration settings	5.4	Tracking & transfer map	5

`em_field` elements will be created when elements are superimposed (§7.1) and there is no other suitable element class.

3.16 Fiducial

A **fiducial** element is used to fix the position and orientation of the reference orbit within the global coordinate system at the location of the **fiducial** element. A **fiducial** element will affect the reference orbit both upstream and downstream of the element.

General **fiducial** element attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	Reference energy	4.3
Description strings	4.2	Tracking & transfer map	5

Attributes specific to a **fiducial** elements are:

```
origin_ele      = <Name>      ! Reference element.
origin_ele_ref_pt = <location> ! Reference pt on reference ele.
dx_origin      = <Real>       ! x-position offset
dy_origin      = <Real>       ! y-position offset
dz_origin      = <Real>       ! z-position offset
dtheta_origin   = <Real>       ! orientation angle offset.
dphi_origin    = <Real>       ! orientation angle offset.
dpsi_origin    = <Real>       ! orientation angle offset.
```

For tracking purposes, the **fiducial** element is considered to be a zero length marker. That is, the transfer map through a **fiducial** element is the unit map.

A **fiducial** element sets the reference orbit of itself and of the elements, both upstream and downstream, around it. This can be thought of as a two step process. The first step is to determine the global coordinates of the **fiducial** element itself, and the second step is to shift the coordinates of the elements around it.

The floor coordinates of the **fiducial** element are determined starting with an **origin_ele** element. If **origin_ele** is not specified, the origin of the global coordinates (§13.3) is used. If the **origin_ele** has a finite length, the reference point may be chosen using the **origin_ele_ref_pt** attribute which may be set to one of

```
entrance_end
center          ! Default
exit_end
```

Once the origin reference position is determined, the reference position of the **fiducial** element is calculated using the offset attributes

```
[dx_origin, dy_origin, dz_origin]
[dtheta_origin, dphi_origin, dpsi_origin]
```

The transformation between origin and fiducial positions is given in §13.3.4.

Once the position of the **fiducial** element is calculated, all elements of the lattice branch the **fiducial** element is contained in, *both* the upstream and downstream elements, are shifted so that everything is consistent. That is, the **fiducial** element orients the entire lattice branch. The exception here is that if there are **flexible patch** elements (§3.34) in the lattice branch, the **fiducial** element will only determine the positions up to the **flexible patch** element.

Example: A lattice branch with elements 0 through 103 has a **fiducial** element at position 34 and a **flexible patch** at position 67. In this case the **fiducial** element will determine the reference orbit for elements 0 through 66.

Rules:

- If an `origin_ele` is specified, the position of this element must be calculated before the position of the `fiducial` element is calculated (§13.1). This means, the `origin_ele` must be in a prior lattice branch from the branch the `fiducial` element is in or the `origin_ele` in the same branch as the `fiducial` element but is positioned upstream from the `fiducial` element and there is a `flexible patch` in between the two elements.
- If a `fiducial` element affects the position of element 0 in the lattice branch (that is, there are no `flexible patch` elements in between), any positioning of element 0 via `beginning` or `line parameter` statements (§8.4) are ignored.
- `Fiducial` elements must not over constrain the lattice geometry. For example, two `fiducial` elements may not appear in the same lattice branch unless separated by a `flexible patch`.

Another example is that if there are no `flexible patch` elements in the lattice, and if branch A has a `branch` element connecting to branch B, the geometry of branch A will be calculated first and the geometry of branch B can then be calculated from the known coordinates of the `fork` element. If branch B contains a `fiducial` element then this is an error since the coordinate calculation never backtracks to recalculate the coordinates of the elements of a branch once the calculation has finished with that branch.

Example:

```
f1: fiducial, origin_ele = mark1, x_offset = 0.04
```

3.17 Floor_Shift

A `floor_shift` element shifts the reference orbit in the global coordinate system without affecting particle tracking. That is, in terms of tracking a `floor_shift` element is equivalent to a `marker` (§3.26) element. Also see `patch` (§3.34) and `fiducial` (§3.16) elements.

General `floor_shift` element attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	Length	4.10
Description strings	4.2	Reference energy	4.3
is_on	4.11	Tracking & transfer map	5

Attributes specific to a `floor_shift` elements are:

```
l          = <Real>      ! Length
x_offset   = <Real>      ! x offset from previous element
y_offset   = <Real>      ! y offset from previous element
z_offset   = <Real>      ! z offset from previous element
x_pitch    = <Real>      ! rotation in the reference coords.
y_pitch    = <Real>      ! rotation in the reference coords.
tilt       = <Real>      ! rotation in the reference coords.
origin_ele = <Name>      ! Reference element.
origin_ele_ref_pt = <location> ! Reference pt on reference ele.
```

The `floor_shift` element shifts the reference orbit relative to the `origin_ele`. Unlike the `patch` element §3.34, the transfer map through a `floor_shift` element will be the unit map. That is, the phase space coordinates of a particle will not change when tracking through a `floor_shift` element. The reference position transformation through a `floor_shift` element is given in Section §13.3.4.

The `l` attribute can be used to adjust the longitudinal s position.

The `floor_shift` element can be used, for example, to restore the correct geometry when a section of the lattice is represented by, say, a `taylor` type element.

If `origin_ele` is not specified, previous element is used. If the `origin_ele` has a finite length, the reference point may be chosen using the `origin_ele_ref_pt` attribute which may be set to one of

```
entrance_end
center
exit_end      ! Default
```

PTC does not have an analogous element for the `Floor_shift` element. When converting to PTC, a `floor_shift` element will be treated as a `marker` element.

Example:

```
floor: floor_shift, z_offset = 3.2
```

This is equivalent to a drift.

3.18 Fork and Photon_Fork

A `fork` or `photon_fork` element marks the start of an alternative `branch` for the beam (or X-rays or other particles generated by the beam) to follow.

Collectively `fork` and `photon_fork` elements are called forking elements. An example geometry is shown in Fig. 3.4. The `branch` containing a forking element is called the “base branch”. The `branch` that the forking element points to is called the “target branch”.

The only difference between `fork` and `photon_fork` is that the default particle type for the target branch forked from a `fork` element is the same particle type as the base branch. The default particle type for the target branch from a `photon_fork` element is a photon. The actual particle associated with a branch can be set by setting the `particle` attribute of the forking element.

General `fork` and `photon_fork` attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	Length	4.10
Chamber wall	4.9	Offsets, pitches, & tilt	4.4
Description strings	4.2	Tracking & transfer map	5
Reference energy	4.3	Is_on	4.11

Attributes specific to `fork` and `photon_fork` elements are:

```
direction   = <+/- 1>      ! Fork for forward or backwards propagating particles?
to_line    = <LineName>    ! What line to fork to.
to_element  = <ElementID>  ! What element to attach to in the line being forked to.
new_branch  = <T/F>        ! Make a new branch from the to_line? Default = True.
```

Branch lines can themselves have forking elements. A branch line always starts out tangential to the line it is branching from. A patch element (§3.34) can be used to reorient the reference orbit as needed.

Example:

```
from_line: line = (... A, PB, B, ...) ! Defines base branch
pb: photon_fork, to_line = x_line
x_line: line = (X_PATCH, X1, X2, ...)           ! Defines target branch
x_patch: patch, x_offset = 0.01
use, from_line
```

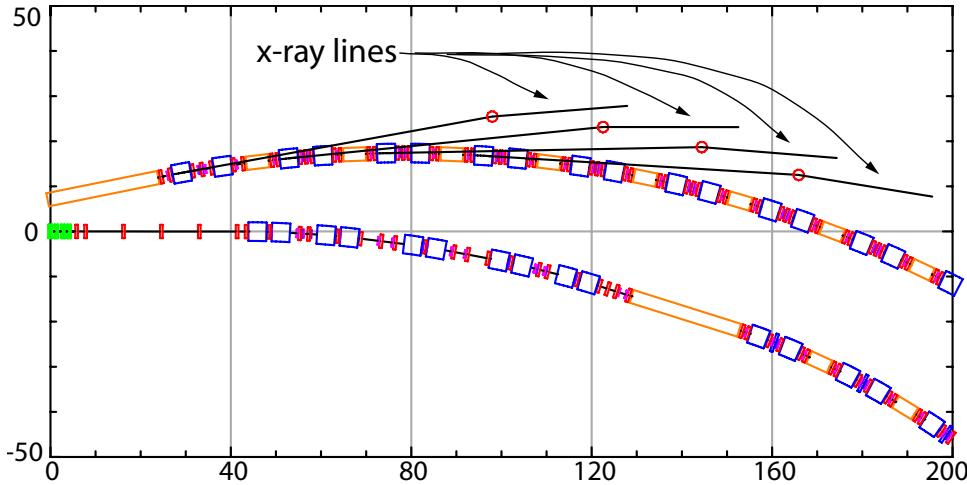


Figure 3.4: Example use of `photon_fork` elements showing four X-ray lines (branches) attached to a machine.

In this example, a photon generated at the fork element PB with $x = 0$ with respect to the `from_line` reference orbit through PB will, when transferred to the `x_line`, and propagated through `X_PATCH`, have an initial value for x of -0.01 .

Forking elements have zero length and, like `marker` elements, the position of a particle tracked through a forking element is constant. Additionally, like `marker` elements, The `x_offset`, `y_offset` and `tilt` attributes are not used by any *Bmad* routines. The `is_on` attribute is also not used by *Bmad* proper.

If the reference orbit needs to be shifted when forking from one ring to another ring, a patch can be placed in a separate “transfer” line to isolate it from the branches defining the rings. Example:

```

ring1: line = (... A, F1, B, ...)      ! First ring
x_line: line = (X_F1, X_PATCH, X_F2)  ! ‘Transfer’ line
ring2: line = (... C, F2, D, ...)      ! Second ring
use, ring1

f1: fork, to_line = x_line
f2: fork, to_line = x_line, direction = -1
x_patch: patch, x_offset = ...
x_f1: fork, to_line = ring1, to_element = f1, direction = -1
x_f2: fork, to_line = ring2, to_element = f2

```

Here the `fork F1` in `ring1` forks to `x_line` which in turn forks to `ring2`.

The above example also illustrates how to connect machines for particles going in the reverse direction. In this case, `ring2` has a `fork` element `f2` back through `x_line` and then to `ring1` via the `x_f2` fork. Notice that both `f2` and `x_f2` have their `direction` attribute set to `-1` to indicate that the fork is appropriate for particles propagating in the $-s$ direction. Additionally, since `f2` has `direction` set to `-1`, it will connect to the downstream end of the `x_line`. The default setting of `direction` is `1`.

The `to_element` attribute for a forking element is used to designate the element of the target branch that the forking element connects to. To keep things conceptually simple, the `to_element` must be a “marker-like” element which has zero length and unit transfer matrix. Possible `to_element` types are:

```

beginning_ele
fiducial

```

```
fork and photon_fork
marker
```

When the `to_element` is not specified, the default is to connect to the beginning of the target branch if `direction` is 1 and to connect to the end of the target branch if `direction` is -1. In this case, there is never a problem connecting to the beginning of the target branch since all branches have a `beginning_ele` element at the beginning. When connecting to the end of the target branch (with `direction` set to -1), the last element in the target branch must be a marker-like element. Note that, by default, a marker element is placed at the end of all branches (§6.1)

If the branch line is to transport particles different from the originating line the particle type and the beginning reference energy must be set for that line using line parameter statements (§8.4). This is useful, for example, for a photon branch line that is branching from a storage ring where the photon energy is not simply related to the particle energy.

Example showing an injection line branching to a ring which, in turn, branches to two x-ray lines:

```
inj: line = (... , br_ele, ...)           ! Define the injection line
use, inj                                ! Injection line is the root
br_ele: fork, to_line = ring              ! Fork element to ring
ring: line = (... , x_br, ... , x_br, ...) ! Define the ring
x_br: photon_fork, to_line = x_line      ! Fork element to x-ray line
x_line: line = (...)                     ! Define the x-ray line
x_line[E_tot] = 1e3
```

The `new_branch` attribute is, by default, `True` which means that the lattice branch created out of the `to_line` line is distinct from other lattice branches of the same name. Thus, in the above example, the two lattice branches made from the `x_line` will be distinct. If `new_branch` is set to `False`, a new lattice branch will not be created if a lattice branch created from the same line already exists. This is useful, for example, when a chicane line branches off from the main line and then branches back to it.

3.19 Girder

A **girder** is a support structure that orients the elements that are attached to it in space. A girder can be used to simulate any rigid support structure and there are no restrictions on how the lattice elements that are supported are oriented with respect to one another. Thus, for example, optical tables can be simulated.

General girder attributes are:

Attribute Class	Section	Attribute Class	Section
Description strings	4.2	Offsets, pitches, & tilt	4.4
Length	4.10		

Attributes specific to a girder are:

```
girder = {<List>} ! List of elements on the Girder
origin_ele      = <Name>      ! Reference element.
origin_ele_ref_pt = <location> ! Reference pt on reference ele.
dx_origin       = <Real>       ! x-position offset
dy_origin       = <Real>       ! y-position offset
dz_origin       = <Real>       ! z-position offset
dtheta_origin   = <Real>       ! orientation angle offset.
dphi_origin     = <Real>       ! orientation angle offset.
dpsi_origin     = <Real>       ! orientation angle offset.
```

Attributes specific to a `floor_shift` elements are:

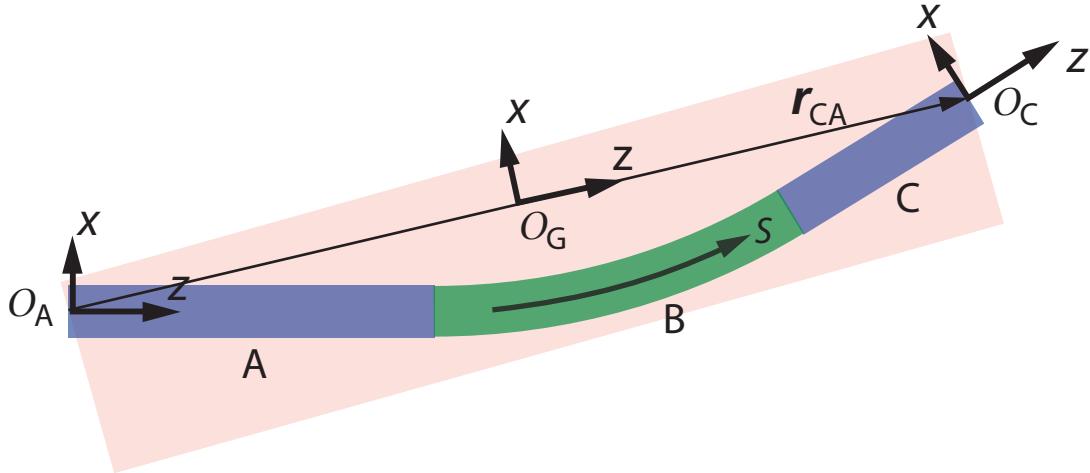


Figure 3.5: Girder supporting three elements labeled A, B, and C. O_A is the reference frame at the upstream end of element A, O_C is the reference frame at the downstream end of element C, and O_G is the default `origin` reference frame of the girder. r_{CA} is the vector from O_A to O_C . The length l of the girder is the difference in s between points O_C and O_A .

A simple example of a girder is shown in Fig. 3.5. Here a girder supports three elements labeled A, B, and C where B is a bend so the geometry is nonlinear. Such a girder may be specified in the lattice file like:

```
g1: girder = {A, B, C}
```

A lattice element may have at most one `girder` supporting it. However, a `girder` can be supported by another `girder` which in turn can be supported by a third `girder`, etc.

The `girder` statement syntax is:

```
<element_name>: GIRDER = {<ele1>, <ele2>, ..., <eleN> }, ...
```

A `girder` element will be created for each section of the lattice where there is a “consecutive” sequence of “slave” elements `<ele1>` through `<eleN>`. This section of the lattice from `<ele1>` through `<eleN>` is called the “girder support region”. “Consecutive” here means there are no other elements in the girder support region except for possibly `drift` and/or `marker` elements. `Drift` elements cannot be controlled by a girder but may appear in the girder slave list. If a drift-like element is desired, use a `pipe` element instead. `Marker` elements present in a girder support region, but not mentioned in the girder slave list, are simply ignored.

Wild card characters (§2.9) can be used in any element name in the girder slave list. Additionally, beam line names (§6.2) can be used. In this case, any `drift` elements within a beam line will be ignored.

The reference frame from which the girder’s offset, pitch, and tilt attributes (§4.4) are measured is constructed as follows: A reference frame, called the “`origin`” reference frame may be defined using the attributes `origin_ele` and `origin_ele_ref_pt` which constructs the girder’s `origin` frame to be coincident with the reference frame of another element. Example:

```
g2: girder = {...}, origin_ele = Q, origin_ele_ref_pt = entrance_end
```

In this example, girder g2 has an `origin` reference frame coincident with the entrance end frame of an element named Q. Valid values are

```
entrance_end
center      ! Default
exit_end
```

For `crystal`, `mirror`, and `multilayer_mirror` elements, setting `origin_ele_ref_pt` to `center` results in the reference frame being the frame of the surface (cf. Fig. 4.6).

If `origin_ele` is not given, the default `origin` frame is used. The default `origin` frame is constructed as follows: Let \mathcal{O}_A be the reference frame of the upstream end of the first element in the list of supported elements. In this example it is the upstream end of element A as shown in the figure. Let \mathcal{O}_C be the downstream end of the last element in the list of supported elements. In this example this is the downstream end of element C. The origin of the `girder`'s reference frame, marked \mathcal{O}_G in the figure, will be half way along the vector r_{CA} from the origin of \mathcal{O}_A to the origin of \mathcal{O}_B . The orientation of \mathcal{O}_G is constructed by rotating the \mathcal{O}_A coordinate system along an axis in \mathcal{O}_A 's x - y plane such that \mathcal{O}_A 's z axis ends up parallel with r_{CA} . In the example above, the rotation axis will be along \mathcal{O}_A 's y -axis.

Once the `origin` reference frame is established, the reference frame of the girder can be offset from the `origin` frame using the parameters

```
dx_origin    dtheta_origin
dy_origin    dphi_origin
dz_origin    dpsi_origin
```

The orientation of the `girder`'s reference frame from the `origin` frame is given in §13.3.4. Example:

```
g3: girder = { ... }, dx_origin = 0.03
```

This offsets girder g3's reference frame 3 cm horizontally from the default `origin` frame. If no offsets are given, the `origin` frame is the same as the girder's reference frame.

The length `l` of a girder, which is not used in any calculations, is a dependent attribute computed by `Bmad` and set equal to the `s` path length between points \mathcal{O}_C and \mathcal{O}_A .

The physical orientation of the girder with respect to its reference frame is, like other elements, determined by the offset, pitch and tilt orientation attributes as outlined in §4.4 and §13.3.4. When a girder is shifted in space, the elements it supports are also shifted. In this case, the orientation attributes (`x_offset`, `y_pitch`, etc.) give the orientation of the element with respect to the `girder`. The orientation with respect to the local reference coordinates is given by `x_offset_tot`, which are computed from the orientation attributes of the element and the `girder`. An example will make this clear:

```
q1: quad, l = 2
q2: quad, l = 4, x_offset = 0.02, x_pitch = 0.01
d: drift, l = 8
g4: girder = {q1, q2}, x_pitch = 0.002, x_offset = 0.03
this_line: line = (q1, d, q2)
use, this_line
```

In this example, g4 supports quadrupoles q1 and q2. Since the supported elements are colinear, the computation is greatly simplified. The reference frame of g4, which is the default `origin` frame, is at $s = 7$ meters which is half way between the start of q1 at $s = 0$ meters and the end of q2 which is at $s = 14$. The reference frames of q1 and q2 are at their centers so the s positions of the reference frames is

Element	S_ref	dS_from_g4
q1	1.0	-6.0
g4	7.0	0.0
q2	12.0	5.0

Using a small angle approximation to simplify the calculation, the `x_pitch` of g4 produces an offset at the center of q2 of $0.01 = 0.002 * 5$. This, added to the offsets of g4 and q2, give the total offset of q2 to be $0.06 = 0.01 + 0.03 + 0.02$. The total `x_pitch` of q2 is $0.022 = 0.02 + 0.001$.

3.20 Group

Group elements are a type of control element (§1.4) used to make variations in the attributes of other elements during execution of a program. 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. Also see **overlay** (§3.33) The difference between **group** and **overlay** elements is that **overlay** elements set the values of the attributes directly while **group** elements make changes to attribute values.

General **girder** attributes are:

Attribute Class	Section	Attribute Class	Section
Description strings	4.2		

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

```
command      = <Real>      ! Command value.
old_command = <Real>      ! Old command value.
coef        = <Real>      ! Conversion factor. Not used by Bmad.
```

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

The general syntax for a **group** element is

```
name: GROUP = {ele1[attrib1]:coef1, ele2[attrib2]:coef2, ...},
                           default_attrib = init_value
```

name is the name of the **group**, the list in brackets {...} is a list of element attributes to vary along with the coefficients used for the variation. In any item **ele**[**attrib**]:**coef** in the list, both **attrib** and **coef** are optional. If not present, **attrib** will default to **default_attrib**. If not present, **coef** will default to 1. Finally, **init_value** is the initial value to use for the **group**.

A **group** element is like an **overlay** element 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** will control all elements of a given name. Thus, in the above example, if there are multiple elements named **q1** then **gr** will control the **k1** attribute of all of them.

A **group** can be used to control an elements position and length using the following as the **default_attrib**

```
accordion_edge ! Element grows or shrinks symmetrically
```

```

start_edge      ! Varies element's upstream edge s-position
end_edge       ! Varies element's downstream edge s-position
symmetric_edge ! Varies element's overall s-position. Constant length.
z_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 `z_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`.

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 $-1.0 = -2.0 * 0.5$.

Different `group` elements may control the same element attribute. And a `group` element may control other `group` `overlay` or `girder` elements. However, It does not make sense for a `group` element to control the same attribute as an `overlay` element or for a `group` element to control a `dependent` attribute (§4.1).

3.21 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.

3.22 Instrument, Monitor, and Pipe

Essentially *Bmad* treats `instrument`, `monitor`, and `pipe` elements like a `drift`. There is a difference, however, when superimposing elements (§7.1). 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.

General `instrument`, `monitor`, and `pipe` attributes are:

Attribute Class	Section	Attribute Class	Section
Symplectify	5.5	Offsets, pitches, & tilt	4.4
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Length	4.10
Description strings	4.2	Is_on	4.11
Integration settings	5.4	Hkick & Vkick	4.5
Instrumental variables	4.17	Chamber wall	4.9

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
```

3.23 Kickers: Hkicker and Vkicker

An `hkicker` gives a beam a horizontal kick and a `vkicker` gives a beam a vertical kick. Also see the `kicker` (§3.24) element.

General `hkicker` `vkicker` attributes are:

Attribute Class	Section	Attribute Class	Section
<code>an</code> , <code>bn</code> multipoles	4.12	Integration settings	5.4
Aperture Limits	4.6	Is_on	4.11
Chamber wall	4.9	Length	4.10
Description strings	4.2	Offsets, pitches, & tilt	4.4
Field table or map	4.13	Reference energy	4.3
Fringe Fields	4.16	Symplectify	5.5
Hkick & Vkick	4.5	Tracking & transfer map	5

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
```

3.24 Kicker

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.

General `kicker` attributes are:

Attribute Class	Section	Attribute Class	Section
<code>an, bn</code> multipoles	4.12	Integration settings	5.4
Aperture Limits	4.6	Is_on	4.11
Chamber wall	4.9	Length	4.10
Description strings	4.2	Offsets, pitches, & tilt	4.4
Field table or map	4.13	Reference energy	4.3
Fringe Fields	4.16	Symplectify	5.5
Hkick & Vkick	4.5	Tracking & transfer map	5

Example:

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

3.25 Lcavity

An lcavity is a LINAC accelerating cavity. The main difference between an rfcavity and an lcavity is that, unlike an rfcavity, the reference energy (§13.4) through an lcavity is not constant.

General lcavity attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	Length	4.10
Chamber wall	4.9	Offsets, pitches, & tilt	4.4
Description strings	4.2	Reference energy	4.3
Field table or map	4.13	RF Couplers	4.14
Fringe Fields	4.16	Wakes	4.15
Hkick & Vkick	4.5	Symplectify	5.5
Integration settings	5.4	Tracking & transfer map	5
Is_on	4.11		

The attributes specific to an lcavity are

```
gradient      = <Real>      ! Accelerating gradient (V/m).
gradient_err  = <Real>      ! Accelerating gradient error (V/m).
field_factor  = <Real>      ! Used for auto scaling.
phi0          = <Real>      ! Phase (rad/2π) of the reference particle with
                           ! respect to the RF. phi0 = 0 is on crest.
phi0_multipass = <Real>    ! Phase with respect to a multipass lord (rad/2π).
phi0_err       = <Real>      ! Phase error (rad/2π)
phi0_ref       = <Real>      ! Phase offset used to deal with absolute time tracking
                           ! and other issues (see below).
e_loss         = <Real>      ! Loss parameter for short range wake fields (V/Coul).
rf_frequency   = <Real>      ! Rf frequency (Hz).
voltage        ! Cavity voltage. Dependent attribute (§4.1).
n_cell         = <Integer>   ! Number of cavity cells. Default is 1.
```

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

```
voltage = gradient * L
```

The energy kick felt by a particle, assuming no phase slippage, is

```
dE = gradient_tot * L * cos(twopi * (phase_particle + phase_ref))
```

where the total gradient is

```
gradient_tot = (gradient + gradient_err) * field_factor
```

the phase `phase_ref` is

```
phase_ref = phi0 + phi0_multipass + phi0_err + phi0_ref
```

and `phase_particle` is

```
phase_particle = -z * rf_frequency / velocity ! With relative time tracking (§13.7)
= t_particle * rf_frequency ! With absolute time tracking
```

With **relative time tracking**, the phase, `phase_particle`, of the particle with respect to the cavity's internal clock is proportional to z – the particles' phase space coordinate (§13.4). With **absolute time tracking**, `phase_particle` is proportional to the absolute time the particle reaches the cavity. See section §13.7 for a discussion on relative verses absolute time tracking. The switch to set the type of tracking for a lattice is `parameter[absolute_time_tracking]` (§8.1).

`phi0_multipass` is only to be used to shift the phase with respect to a `multipass` lord. See §7.2.

`phi0_ref` is a phase calculated by *Bmad*'s RF auto-phase module (§4.13).

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 * phase_ref)
```

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 * (phase_ref - phi(z))) ! WRONG
```

This is incorrect.

The `field_factor` parameter is used with auto scaling (§9). The default value for `field_factor` is 1.

When short-range wake fields are being simulated, with `bmad_com%sr_wakes_on = True` (§10.1), the `e_loss` attribute can be used to modify the gradient in order to maintain a constant average energy gain. That is, `e_loss` can be used to simulate the effect of a feedback circuit that attempts to maintain the average energy of the bunch after the element constant. The energy kick is then

```
dE(with wake) = dE + e_loss * n_part * e_charge
```

`n_part` is set using the `parameter` statement (§8.1) and represents the number of particles in a bunch. `e_charge` is the charge on an electron (Table 2.2). Notice that the `e_loss` term is independent of the sign of the charge of the particle.

The transverse trajectory through an `lcavity` is modeled using equations developed by Rosenzweig and Serafini[Rosen94] modified to give the correct phase-space area at non ultra-relativistic energies. See Section §19.11 for more details. Note: The transfer matrix for an `lcavity` with finite `gradient` is never symplectic. See §13.4. In addition, couplers (§4.14) and HOM wakes (§4.15) can be modeled.

When using `boris` or `runge_kutta`, tracking (§5.1) through a field map (§4.13), the `bmad_standard` field is `n_cell` half wave resonators.

Example:

```
lwf: lcavity, l = 2.3, rf_frequency = 500e6, voltage = 20e6
```

Note: The default `bmad_standard` tracking for `lcavity` elements when the velocity β is significantly different from 1 can only be considered as a rough approximation. Indeed, the only accurate way to simulate a cavity in this situation is by integrating through the actual field [Cf. Runge Kutta tracking (§5.1)]

3.26 Marker

A `marker` is a zero length element meant to mark a position.

General `marker` attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	Instrumental variables	4.17
Chamber wall	4.9	Is_on	4.11
Description strings	4.2	Offsets & tilt	4.4
Reference energy	4.3	Tracking & transfer map	5

Attributes specific to a `marker` element are:

```
x_ray_line_len = <Real>
```

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

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"
```

3.27 Match

A `match` element is used to match the Twiss parameters between two points.

General `match` attributes are:

Attribute Class	Section	Attribute Class	Section
Description strings	4.2	Is_on	4.11
Aperture Limits	4.6	Length	4.10
Reference energy	4.3	Integration settings	5.4

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 = <Logic>                  ! 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 = <Logic>           ! See below. Default is False.

```

The transfer map for a `match` element is a linear transformation with a “kick”:

$$r_1 = \mathbf{M} r_0 + \mathbf{V} \quad (3.13)$$

where r_1 is the output coordinates, and r_0 are the input coordinates. The matrix \mathbf{M} is the linear part of the map and the vector \mathbf{V} is the zeroth order part of the map.

Nomenclature: The parameters `beta_a0`, `alpha_a0`, etc. of the `match` element are called the “entrance Twiss parameters”. The parameters `beta_a1`, `alpha_a1`, etc. of the `match` element are called the “exit Twiss parameters”.

The matrix \mathbf{M} is calculated such that if the actual (computed) Twiss parameters at the exit end of the element preceding the `match` element are equal to the entrance Twiss parameters of the `match` element, then the computed Twiss parameters at the exit end of the `match` element will be the exit Twiss parameters of the `match` element and the phase advances (in radians) will be `dphi_a` and `dphi_b`. Notice that there is no guarantee that the actual Twiss parameters after a `match` element be equal to the exit Twiss parameters of the `match` element.

The kick term \mathbf{V} is constructed so that if a particle has coordinates (x_0 , px_0 , y_0 , py_0 , z_0 , pz_0) before the `match` element, the coordinates just after the element will be (x_1 , px_1 , y_1 , py_1 , z_1 , pz_1). With this, \mathbf{V} will be:

$$\mathbf{V} = \begin{pmatrix} x_1 \\ px_1 \\ y_1 \\ py_1 \\ z_1 \\ pz_1 \end{pmatrix} - \mathbf{M} \begin{pmatrix} x_0 \\ px_0 \\ y_0 \\ py_0 \\ z_0 \\ pz_0 \end{pmatrix} \quad (3.14)$$

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, ...
```

`Match_end`

The `match_end` attribute is used for appropriately setting the entranch Twiss parameters of the `match` element from within a program. If the `match_end` attribute is set to True, the entrance Twiss parameters are set to be equal to 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 exit Twiss parameters of the `match` element. The `match_end` attribute may only be used with `open` lattices (§8.1) since, for a `closed` lattice, it is not possible to calculate the Twiss parameters at the previous element independently of the exit Twiss parameters at the `match` element.

When running a program, if a `match` element initially has its `match_end` attribute is set to True, the *Bmad* bookkeeping routines will ensure that the `match` element’s beginning Twiss parameters are appropriately set as explained above. If `match_end` is now toggled to False by the program, the beginning Twiss attribute values, and hence the transfer matrix for the `match` element, will be frozen. Variation now of any parameter in the lattice that affects the calculated Twiss parameters through the `match` element will not affect the `match` element’s transfer matrix.

`match_end_orbit` The `match_end_orbit` attribute is similar to the `match_end` attribute. When running a program, if `match_end_orbit` is set to True, when any particle is tracked through the `match` element, the `match` element’s starting coordinate parameters, (x_0 , px_0 , y_0 , py_0 , z_0 , pz_0), will be set to the particle’s coordinates at the exit end of the previous element. That is, the

particle will always have coordinates equal to $(x_1, px_1, y_1, py_1, z_1, pz_1)$ at the end of the `match` element. If `match_end_orbit` is now toggled to False by the program, the ending coordinate parameters, and hence the \mathbf{V} vector, will become fixed. As with the `match_end` attribute, the `match_end_orbit` attribute may only be used with open lattices (§8.1).

3.28 Mirror

A `mirror` reflects photons.

General `mirror` attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	Reference energy	4.3
Description strings	4.2	Surface Properties	4.8
Offsets, pitches, & tilt	4.4	Tracking & transfer map	5

Attributes specific to a `mirror` element are:

```
graze_angle      = <Real>      ! Angle between incoming beam and mirror surface.
critical_angle   = <Real>      ! Critical angle.
```

The reference trajectory for a `mirror` is that of a zero length bend (§13.3.3) 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 `ref_tilt` parameters. A positive `graze_angle` bends the reference trajectory in the same direction as a positive `g` for a bend element.

A `mirror` may be offset and pitched (4.4). The incoming local reference coordinates are used for these misalignments.

3.29 Multipole

A `multipole` is a thin multipole lens up to 20th order. The basic difference between this and an `ab_multipole` is the input format. See section §14.1 for how the multipole coefficients are defined.

General `multipole` attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	Reference energy	4.3
Chamber wall	4.9	Is_on	4.11
Description strings	4.2	Offsets, pitches, & tilt	4.4
K_{nL} , T_n multipoles	4.12	Tracking & transfer map	5

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
```

3.30 Multilayer_mirror

A `multilayer_mirror` is a substrate upon which multiple layers of alternating substances have been deposited. The idea is similar to crystal diffraction: light reflected at each interface constructively interferes with light reflected from other interfaces. The amplified reflection offsets losses due to absorption.

General `crystal` attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	Symplectify	5.5
Description strings	4.2	Offsets, pitches & tilt	4.4
Reference energy	4.3	Tracking & transfer map	5
Surface Properties	4.8		

The attributes specific to a `multilayer_mirror` are

```
material_type      = <String> ! Materials in each layer.
d1_thickness       = <Real>    ! Thickness of layer 1
d2_thickness       = <Real>    ! Thickness of layer 2
n_cell             = <Integer> ! Number of cells (= Number of layers / 2)
```

Dependent attributes (§4.1) are

```
graze_angle        ! Angle between incoming beam and mirror surface.
v1_unitcell        ! Unit cell volume for layer 1
v2_unitcell        ! Unit cell volume for layer 2
```

A `multilayer_mirror` is constructed of a number of “cells”. The number of cells is set by `n_cell`. Each cell consists of two layers of dielectric material. The materials used is given by the `material_type` attribute. The format for this is

```
material_type = "<material_1>:<material_2>"
```

where `<material_1>` and `<material_2>` are the material names for the first and second layers of the cell respectively. The first layer is the bottom layer and the second layer is the top layer of the cell. Material names are case sensitive. So “FE” cannot be used in place of “Fe” A list of materials is given in §4.7 and can include crystal materials or elemental materials.

Example:

```
mm: multilayer_mirror, material_type = 'W:BORON_CARBIDE', n_cell = 100, &
     d1_thickness = 1e-9, d2_thickness = 1.5e-9
```

3.31 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 (§7.1). It is not generally useful otherwise.

3.32 Octupole

An `octupole` is a magnetic element with a cubic field dependence with transverse offset (§14.1). The `bmad_standard` calculation treats an octupole using a kick-drift-kick model.

General `octupole` attributes are:

Attribute Class	Section	Attribute Class	Section
<code>an, bn</code> multipoles	4.12	Integration settings	5.4
Aperture Limits	4.6	Is_on	4.11
Chamber wall	4.9	Length	4.10
Description strings	4.2	Offsets, pitches, & tilt	4.4
Field table or map	4.13	Reference energy	4.3
Fringe Fields	4.16	Symplectify	5.5
Hkick & Vkick	4.5	Tracking & transfer map	5

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
```

3.33 Overlay

Overlay elements are a type of `control` element (§1.4) used to make variations in the attributes of other elements while a program is running. For example, to simulate the action of a magnet power supply that controls a string of magnets. Also see `group` (§3.20) The difference between `group` and `overlay` elements is that `overlay` elements set the values of the attributes directly while `group` elements make changes to attribute values.

General `girder` attributes are:

Attribute Class	Section	Attribute Class	Section
Description strings	4.2		

The general syntax for a `overlay` element is

```
name: OVERLAY = {ele1[attrib1]:coef1, ele2[attrib2]:coef2, ...},
                           default_attrib = init_value
```

`name` is the name of the `overlay`, the list in brackets `{...}` is a list of element attributes to vary along with the coefficients used for the variation. In any item `ele[attrib]:coef` in the list, both `attrib` and `coef` are optional. If not present, `attrib` will default to `default_attrib`. If not present, `coef` will default to 1. Finally, `init_value` is the initial value to use for the `overlay`.

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 vertical colon ":" 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

```

Note: An older notation allowed a slash "/" to be used in place of the colon ":" for the separator between the slave name and the coefficient. While the slash is still accepted, its use is discouraged.

An **overlay** will control all elements of a given name. Thus, in the above example, if there are multiple elements in the lattice 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]:5.4, this_sextupole[k2], ...}, hkick
```

As illustrated above, different **overlay** elements may control the same element attribute. And an **overlay** element may control other **overlay**, **group** or **girder** elements. However, It does not make sense for an **overlay** element to control the same attribute as a **group** element or for an **overlay** element to control a dependent attribute (§4.1).

3.34 Patch

A **patch** element shifts the reference orbit and time. Also see **floor_shift** (§3.17) and **fiducial** (§3.16) elements.

General **patch** element attributes are:

Attribute Class	Section	Attribute Class	Section
is_on	4.11	Offsets, pitches, & tilt	4.4
Reference energy	4.3	Tracking & transfer map	5
Aperture Limits	4.6	Description strings	4.2
Length	4.10		

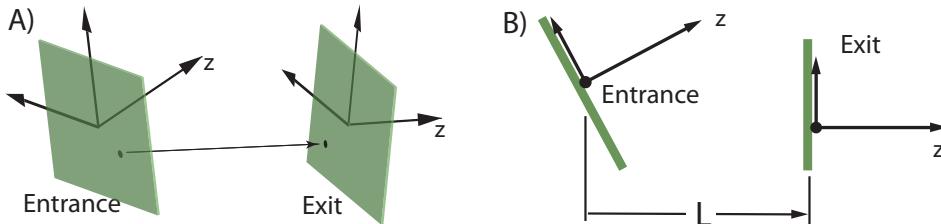


Figure 3.6: A) A **patch** element can align its exit face arbitrarily with respect to its entrance face. B) The reference length of a **patch** element is the longitudinal distance from the entrance origin to the exit origin using the reference coordinates at the exit end.

Attributes specific to a patch elements are:

```
x_offset      = <Real> ! Exit face offset from Entrance.
y_offset      = <Real> ! Exit face offset from Entrance.
z_offset      = <Real> ! Exit face offset from Entrance.
t_offset      = <Real> ! Reference time offset.
x_pitch       = <Real> ! Exit face orientation from Entrance.
y_pitch       = <Real> ! Exit face orientation from Entrance.
tilt          = <Real> ! Exit face orientation from Entrance.
e_tot_offset  = <Real> ! Reference energy offset (eV).
flexible      = <Logic> ! Default: False.
l             = <Real> ! Reference length. Dependent attribute (§4.1).
```

A straight line element like a drift or a quadrupole has the the exit face parallel to the entrance face. With a patch element, the entrance and exit faces can be arbitrarily oriented with respect to one another as shown in Fig. 3.6A. The length l of a patch a dependent (§4.1) parameter and is the longitudinal (z component) distance from the entrance origin to the exit origin using the exit end reference coordinates as shown in Fig. 3.6B.

There are two different ways the orientation of the exit face is determined. Which way is determined by the setting of the flexible attribute. With the flexible attribute set to False, the default, The exit face of the patch will be determined from the offset, tilt and pitch attributes as described in §13.3.4. This type of patch is called “rigid” or “inflexible” since the geometry of the patch is solely determined by the patch’s attributes and is independent of everything else. Example:

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

With flexible set to True, the exit face is taken to be the reference frame of the entrance face of the next element in the lattice. In this case, it must be possible to compute the reference coordinates of the next element before the reference coordinates of the patch are computed. A flexible patch will have the its offsets, pitches, and tilt as dependent parameters (§4.1) and these parameters will be computed. Here the patch is called “flexible” since the geometry of the patch will depend upon the geometry of the rest of the lattice and, therefore, if the geometry of the rest of the lattice is modified (is “flexed”), the geometry of the patch will vary as well. See Section §11.2 for an example.

With bmad_standard tracking (§5.1) A particle, starting at the upstream face of the patch, is propagated in a straight line to the downstream face and the suitable coordinate transformation is made to translate the particle’s coordinates from the upstream coordinate frame to the downstream coordinate frame (§19.14). In this case the patch element can be thought of as a generalized drift element.

If there are magnetic or electric fields within the patch, the tracking method through the patch must be set to either runge_kutta or custom. Example:

```
pa2: patch, tracking_method = runge_kutta, field_calc = custom,
      mat6_calc_method = tracking, ...
```

In order to supply a custom field when runge_kutta tracking is used, field_calc (§5.4) needs to be set to custom. In this case, custom code must be supplied for calculating the fields as a function of position (§29.1).

The e_tot_offset attribute offsets the reference energy:

```
E_tot_ref(exit) = E_tot_ref(entrance) + E_tot_offset (eV)
```

Setting the e_tot_offset attribute will affect a particle’s p_x , p_y and p_z coordinates via Eqs. (13.25) and (13.29). Notice that e_tot_offset does not affect a particle’s actual energy, it just affects the difference between the particle energy and the reference energy.

Important: The E_tot_offset, transformation may be applied either before or after the coordinate transformation. This matters when the particle speed is less than c . If necessary, use two patches in a row to ensure the E_tot_offset is applied in the correct order.

The t_offset attribute offsets the reference time:

```
t_ref(exit) = t_ref(entrance) + t_offset + dt_travel_ref
```

where `dt_transit_ref` is the time for the reference particle to travel through the patch and `dt_travel_ref` is the time the reference particle takes to travel the patch. Setting the `t_offset` attribute will affect a particle's z coordinate via Eqs. (13.26).

`dt_travel_ref` in the above equation is computed by:

```
dt_travel_ref = L / beta_ref
```

Where `L` is the length of the patch as shown in Fig. 3.6 and `beta_ref` is the reference velocity/ c at the exit end of the element. That is, the reference energy offset is applied *before* the reference particle is tracked through the patch. Since this point can be confusing, it is recommended that a `patch` element be split into two consecutive patches if the `patch` has finite `l` and `E_tot_offset` values.

When a lattice branch contains both normally oriented and reversed elements (§13.2), a `patch`, or series of `patches`, which reflects the z direction must be placed in between. See §11.3 for an example. Such a `patch`, (or patches) is called a **reflection patch**. See Section §13.3.5 for more details on how a reflection patch is defined.

3.35 Quadrupole

A **quadrupole** is a magnetic element with a linear field dependence with transverse offset (§14.1).

General quadrupole attributes are:

Attribute Class	Section	Attribute Class	Section
<code>an</code> , <code>bn</code> multipoles	4.12	Integration settings	5.4
Aperture Limits	4.6	<code>Is_on</code>	4.11
Chamber wall	4.9	Length	4.10
Description strings	4.2	Offsets, pitches, & tilt	4.4
Field table or map	4.13	Reference energy	4.3
Fringe Fields	4.16	Symplectify	5.5
Hkick & Vkick	4.5	Tracking & transfer map	5

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

```
b1_gradient      = <Real>      ! Field strength. (§4.1).
k1               = <Real>      ! Quadrupole strength.
f1               = <Real>      ! SAD ‘‘linear’’ fringe at entrance end.
f2               = <Real>      ! SAD ‘‘linear’’ fringe at exit end.
```

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

For a quadrupole with zero `tilt` and a positive `k1`, the quadrupole is horizontally focusing and vertically defocusing (§14.1).

Example:

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

3.36 RFcavity

An `rfcavity` is an RF cavity without acceleration generally used in a storage ring. The main difference between an `rfcavity` and an `lcavity` is that, unlike an `lcavity`, the reference energy (§13.4) through an `rfcavity` is constant.

General rfcavity attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	Length	4.10
Chamber wall	4.9	Offsets, pitches, & tilt	4.4
Description strings	4.2	Reference energy	4.3
Field table or map	4.13	RF Couplers	4.14
Fringe Fields	4.16	Wakes	4.15
Hkick & Vkick	4.5	Symplectify	5.5
Integration settings	5.4	Tracking & transfer map	5
Is_on	4.11		

Attributes specific to an rfcavity are:

```

rf_frequency = <Real>      ! Frequency
harmon       = <Real>      ! Harmonic number
voltage       = <Real>      ! Cavity voltage
field_factor = <Real>      ! Used for auto scaling.
phi0          = <Real>      ! Cavity phase
phi0_multipass = <Real>     ! Phase variation with multipass
gradient      = <Real>      ! Accelerating gradient (V/m). Dependent attribute (§4.1).
The phi0 attribute here is identical to the lag attribute of MAD. The integrated energy kick felt by a
particle, assuming no phase slippage, is
dE = -e_charge * voltage * sin(twopi * (phase_particle - phase_ref))
where
phase_ref = phi0 + phi0_multipass + phi0_ref
and and phase_particle is
phase_particle = -z * rf_frequency / velocity ! With relative time tracking (§13.7)
            = t_particle * rf_frequency ! With absolute time tracking

```

With relative time tracking, the phase, `phase_particle`, of the particle with respect to the cavity's internal clock is proportional to z – the particles' phase space coordinate (§13.4). With absolute time tracking, `phase_particle` is proportional to the absolute time the particle reaches the cavity. See section §13.7 for a discussion on relative verses absolute time tracking. The switch to set the type of tracking for a lattice is `parameter[absolute_time_tracking]` (§8.1).

`phi0_multipass` is only to be used to shift the phase with respect to a `multipass` lord. See §7.2. `e_charge` is the charge on an electron (Table 2.2). Notice that the energy kick is independent of the sign of the charge of the particle

`phi0_ref` is a phase calculated by Bmad's RF auto-phase module (§4.13).

If `harmon` is non-zero the `rf_frequency` is calculated by

```
rf_frequency = harmon * c_light * beta0 / L_lattice
```

where `L_lattice` is the total lattice length and `beta0` is the velocity of the reference particle at the start of the lattice. After the lattice has been read in, `rf_frequency` will be the independent variable (§4.1).

Couplers (§4.14) and HOM wakes (§4.15 can be modeled. In addition, if a field map is specified (§4.13), tracking using an integrator is possible.

If a field map is specified (§4.13), tracking using an integrator is possible. A field map is only used for `runge_kutta`, `adaptive_runge_kutta`, `boris` and `symp_lie_bmad` tracking (§5.1). Only the fundamental mode has an analytical formula for the symplectic tracking. In the future, the other modes could be used with `symp_lie_bmad` tracking using a field expansion about the centerline.

Example:

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

3.37 Sad_Mult

A `sad_mult` element is equivalent to a SAD[SAD] `mult` element. This element is a combination solenoid, multipole, bend, and RF cavity.

General sample attributes are:

Attribute Class	Section	Attribute Class	Section
<code>an, bn</code> multipoles	4.12	Length	4.10
Aperture Limits	4.6	Offsets, pitches, & tilt	4.4
Chamber wall	4.9	Reference energy	4.3
Description strings	4.2	Tracking & transfer map	5
Integration settings	5.4		

This element is in development.

Attributes specific to an `sextupole` element are:

```

angle          = <Real>      ! Bend angle. A settable dependent variable (§4.1)
bs_field       = <Real>      ! Solenoid field. See SAD bz attribute.
e1, e2         = <Real>      ! Bend face angles.
eps_step_scale = <Real>      ! Step size scale. Default = 1. See SAD eps attribute.
f1             = <Real>      ! Quadrupole fringe integral. See SAD f1 attribute.
f2             = <Real>      ! Quadrupole fringe integral. See SAD f2 attribute.
g              = <Real>      ! Bend strength 1/rho
rf_frequency   = <Real>      ! Rf frequency (Hz). See SAD volt attribute.
harmon         = <Real>      ! Harmonic number. See SAD harm attribute.
ks             = <Real>      ! Solenoid strength.
phi0           = <Real>      ! Cavity phase. See SAD phi attribute.
rho             = <Real>      ! Bend radius. A settable dependent variable (§4.1)
voltage         = <Real>      ! Cavity voltage. See SAD volt attribute.
x_offset_mult  = <Real>      ! Mult component offset. See SAD dx attribute.
y_offset_mult  = <Real>      ! Mult component offset. See SAD dy attribute.
x_pitch_mult   = <Real>      ! Mult component pitch. See SAD dpx or chi1 attribute.
y_pitch_mult   = <Real>      ! Mult component pitch. See SAD dpy or chi2 attribute.
fringe_type    = <Switch>    ! Type of fringe. See SAD disfrin attribute.
fringe_at      = <Switch>    ! Where fringe is applied. See SAD fringe attribute.

```

One difference between SAD and *Bmad* is that SAD defines the solenoid field by what are essentially a set of marker elements so that the solenoid field at a SAD `mult` element is not explicitly declared in the `mult` element definition. *Bmad*, on the other hand, requires a `sad_mult` element to explicitly declare the solenoid parameters.

Another difference between SAD and *Bmad* is that, within a solenoid, the reference trajectory is aligned with the solenoid axis (and not aligned with the axis of the elements within the solenoid region).

The SAD `mult` element uses normal `Kn` and skew `KSn` multipole components. The *Bmad* `sad_mult` element used normal `an` and skew `bn` multipole components. As can be seen from the equations in §14.1, there is a factor of $n!$ between the two representations.

The `fringe_type` and `fringe_at` switches are discussed in Sec. §4.16. The translation between these two switches and the `fringe` and `disfrin` switches of SAD is:

		fringe_at:			
		no_end	entrance_end	exit_end	both_ends*
fringe_type:	none	[0, ≠ 0]	[0, ≠ 0]	[0, ≠ 0]	[0, ≠ 0]
	sad_linear	[0, ≠ 0]	[1, ≠ 0]	[2, ≠ 0]	[3, ≠ 0]
	sad_nonlin_only*	[0, ≠ 0]	-	-	[0, = 0]
	sad_full	[0, ≠ 0]	[1, = 0]	[2, = 0]	[3, = 0]

*Default value.

Each entry is the table is of the form [fringe, disfrin]. The “-” indicates that there is no equivalent setting in SAD. The `sad_linear` fringe kick is a kick that is linear in the transverse (x, p_x, y, p_y) coordinates and comes from the finite width of the quadrupolar fringe field. The width of the quadrupolar fringe field is characterized by the `f1` and `f2` attributes. The `sad_nonlinear_only` fringe kick comes from the nonlinear part of the quadrupolar field plus the fringes of the other multipoles.

The `eps_step_scale` is equivalent to the SAD `eps` attribute. The default `eps_step_scale` value is 1 which is essentially the same default as SAD. From `eps_step_scale`, the values of `num_steps` and `ds_step` (§5.4) are computed. Thus, for a `sad_mult` element, `num_steps` and `ds_step` are dependent attributes (§4.1).

Sad conventions:

- A SAD `rotate` or `chi3` rotation is opposite to a *Bmad* `tilt`
- SAD element offsets (`dx`, `dy`, `dz`) are with respect to the entrance end of the element as opposed to *Bmad*'s convention of referencing to the element center.

3.38 Sample

A `sample` element is used to simulate a material sample which is illuminated by x-rays.

General `sample` attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	Offsets, pitches, & tilt	4.4
Chamber wall	4.9	Reference energy	4.3
Description strings	4.2	Surface Properties	4.8
Integration settings	5.4	Tracking & transfer map	5
Length	4.10		

This element is in development.

Attributes specific to an `solenoid` element are:

```
mode      = <Switch> ! Reflection or transmission.
material  = <type>   ! Type of material. §4.7
```

The `mode` parameter can be set to:

```
reflection
transmission
```

With `mode` set to `reflection`, photons will be back scattered from the sample surface isotropically. In this case the material properties will not matter. Additionally, a `patch` (§3.34) element will be needed after the `sample` element to properly reorient the reference orbit.

With `mode` set to `transmission`, photons will be transmitted through the sample. In this case `material` will be used to determine the attenuation and phase shift of the photons.

Example:

```
plate: sample, x_limit = 10e-3, y_limit = 20e-3, mode = reflection
```

3.39 Sextupole

A **sextupole** is a magnetic element with a quadratic field dependence with transverse offset (§14.1).

General `sextupole` attributes are:

Attribute Class	Section	Attribute Class	Section
<code>an, bn</code> multipoles	4.12	Integration settings	5.4
Aperture Limits	4.6	<code>Is_on</code>	4.11
Chamber wall	4.9	Length	4.10
Description strings	4.2	Offsets, pitches, & tilt	4.4
Field table or map	4.13	Reference energy	4.3
Fringe Fields	4.16	Symplectify	5.5
Hkick & Vkick	4.5	Tracking & transfer map	5

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
```

3.40 Solenoid

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

General `solenoid` attributes are:

Attribute Class	Section	Attribute Class	Section
<code>an, bn</code> multipoles	4.12	Integration settings	5.4
Aperture Limits	4.6	<code>Is_on</code>	4.11
Chamber wall	4.9	Length	4.10
Description strings	4.2	Offsets, pitches, & tilt	4.4
Field table or map	4.13	Reference energy	4.3
Fringe Fields	4.16	Symplectify	5.5
Hkick & Vkick	4.5	Tracking & transfer map	5

Attributes specific to an `solenoid` element are:

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

The `bmad_standard` tracking model (§5.1) uses a “hard edge” model where an impulse kick is applied at the entrance and exit ends of the element due to the fringe fields there.

Example:

```
cleo_sol: solenoid, l = 2.6, ks = 1.5e-9 * parameter[e_tot]
```

3.41 Sol_Quad

A `sol_quad` is a combination solenoid/quadrupole.

General `sol_quad` attributes are:

Attribute Class	Section	Attribute Class	Section
<code>an, bn</code> multipoles	4.12	Integration settings	5.4
Aperture Limits	4.6	Is_on	4.11
Chamber wall	4.9	Length	4.10
Description strings	4.2	Offsets, pitches, & tilt	4.4
Field table or map	4.13	Reference energy	4.3
Fringe Fields	4.16	Symplectify	5.5
Hkick & Vkick	4.5	Tracking & transfer map	5

Attributes specific to a `sol_quad` element are:

```
k1      = <Real>    ! Quadrupole strength.
ks      = <Real>    ! Solenoid strength.
bs_field = <Real>    ! Solenoid Field strength.
b1_gradient = <Real>  ! Quadrupole Field strength.
```

Example:

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

3.42 Taylor

A `taylor` is a Taylor map (§18.1). This can be used in place of the `MAD` matrix element.

General `taylor` attributes are:

Attribute Class	Section	Attribute Class	Section
Description strings	4.2	Is_on	4.11
Aperture Limits	4.6	Symplectify	5.5
Length	4.10	Tracking & transfer map	5
Reference energy	4.3	Offsets & tilt	4.4

Attributes specific to a `taylor` element are:

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

A term in a Taylor map is of the form

$$x_j(\text{out}) = C \cdot \prod_{i=1}^6 x_i^{e_i}(\text{in}) \quad (3.15)$$

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

$$p_y(\text{out}) = 2.73 \cdot y^2(\text{in}) p_z(\text{in}) \quad (3.16)$$

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 `l` length attribute is not in any map calculation. `l` can be 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.

Example `taylor` element definition:

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

3.43 Wiggler

A `wiggler` is basically a periodic array of alternating bends.

General `wiggler` attributes are:

Attribute Class	Section	Attribute Class	Section
<code>an</code> , <code>bn</code> multipoles	4.12	Integration settings	5.4
Aperture Limits	4.6	<code>Is_on</code>	4.11
Chamber wall	4.9	Length	4.10
Description strings	4.2	Offsets, pitches, & tilt	4.4
Field table or map	4.13	Reference energy	4.3
Fringe Fields	4.16	Symplectify	5.5
Hkick & Vkick	4.5	Tracking & transfer map	5

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”).

3.43.1 Map_Type Wigglers

The `map` type wigglers are modeled using the method of Sagan, Crittenden, and Rubin [[Sagan03](#)]. In this model the magnetic field is written as a sum of terms B_i

$$\mathbf{B}(x, y, z) = \sum_i \mathbf{B}_i(x, y, z; C, k_x, k_y, k_z, \phi_z) \quad (3.17)$$

Each term B_i is specified using five numbers: $(C, k_x, k_y, k_z, \phi_z)$. A term can take one of three forms: The first form is

$$\begin{aligned} B_x &= -C \frac{k_x}{k_y} \sin(k_x x) \sinh(k_y y) \cos(k_z z + \phi_z) \\ B_y &= C \frac{k_x}{k_y} \cos(k_x x) \cosh(k_y y) \cos(k_z z + \phi_z) \\ B_s &= -C \frac{k_z}{k_y} \cos(k_x x) \sinh(k_y y) \sin(k_z z + \phi_z) \end{aligned} \quad (3.18)$$

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

The second form is

$$\begin{aligned} B_x &= C \frac{k_x}{k_y} \sinh(k_x x) \sinh(k_y y) \cos(k_z z + \phi_z) \\ B_y &= C \frac{k_x}{k_y} \cosh(k_x x) \cosh(k_y y) \cos(k_z z + \phi_z) \\ B_s &= -C \frac{k_z}{k_y} \cosh(k_x x) \sinh(k_y y) \sin(k_z z + \phi_z) \end{aligned} \quad (3.19)$$

with $k_y^2 = k_z^2 - k_x^2$,

The third form is

$$\begin{aligned} B_x &= C \frac{k_x}{k_y} \sinh(k_x x) \sin(k_y y) \cos(k_z z + \phi_z) \\ B_y &= C \frac{k_x}{k_y} \cosh(k_x x) \cos(k_y y) \cos(k_z z + \phi_z) \\ B_s &= -C \frac{k_z}{k_y} \cosh(k_x x) \sin(k_y y) \sin(k_z z + \phi_z) \end{aligned} \quad (3.20)$$

with $k_y^2 = k_x^2 - k_z^2$.

The relationship between k_x , k_y , and k_z ensures that Maxwell's equations are satisfied.

Attributes specific to a `map` type `wiggler` element are:

```
term(i) = {<Wig_Term>}
b_max = <Real> ! Maximum magnetic field (in T) on the wiggler centerline.
! Dependent attribute (§4.1).
```

A `<Wig_Term>` is of the form C, k_x, k_y, k_z, ϕ_z as explained in above. `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
```

The `b_max` attribute for a `map` type `wiggler` is the maximum field computed for `polarity` = 1. The actual maximum field seen in tracking is thus scaled by the `polarity`.

There is no `bmad_standard` tracking for a `map_type` `wiggler`. `symp_lie_bmad` type tracking is discussed in §19.3

3.43.2 Periodic_Type Wiggler Element Tracking

For the `periodic` type wigglers the attributes are:

```
b_max = <Real> ! Maximum magnetic field (in T) on the wiggler centerline.
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).
```

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.

The horizontal motion looks like a drift with a superimposed sinusoidal oscillation. It is assumed that there is an integer number of periods in the oscillation so that the exit horizontal coordinates can be calculated from the initial coordinates using the equations for a drift. The vertical motion is a quadratic superimposed with a octupole. Vertical motion is calculated using a kick-drift-kick model.

`Periodic` type wigglers use a simplified model where the magnetic field components are

$$\begin{aligned} B_y &= B_{\max} \cosh(k_z y) \cos(k_z z + \phi_z) \\ B_z &= -B_{\max} \sinh(k_z y) \sin(k_z z + \phi_z) \end{aligned} \quad (3.21)$$

where B_{\max} is the maximum field on the centerline and k is given in terms of the pole length (`l_pole`) by

$$k_z = \frac{\pi}{l_{\text{pole}}} \quad (3.22)$$

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[Corbett99]

$$\frac{dp_y}{dz} = k_1 \left(y + \frac{2}{3} k_z^2 y^3 \right) \quad (3.23)$$

where

$$k_1 = \frac{-1}{2} \left(\frac{e B_{\max}}{P_0 (1 + p_z)} \right)^2 \quad (3.24)$$

with k_1 being the linear focusing constant. For radiation calculations the true horizontal trajectory with $y = 0$ is needed

$$x = \frac{\sqrt{2 |k_1|}}{k_z^2} \cos(k_z z) \quad (3.25)$$

With `periodic` type wigglers and `bmad_standard` tracking, the phase ϕ_z in Eqs. (3.21) 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

$$\phi_z = \frac{-k_y L}{2} \quad (3.26)$$

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.

When using a tracking through a periodic wiggler with a tracking method that integrates through the magnetic field (§5.4), The magnetic field is approximated using a single wiggler `term` as if the wiggler were a `map` type wiggler. This wiggler model has unphysical end effects and will give results that are different from the results obtained when using the `bmad_standard` tracking method.

Example:

```
wig_w: wiggler, l = 2.3, b_max = 2.3, l_pole = 6
```

3.43.3 Common Wiggler Parameters

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.

both types of wigglers have the following attributes:

```
x_ray_line_len = <Real>
polarity      = <Real> ! Used to scale the field strength.
k1           ! Vertical focusing strength. Dependent attribute (§4.1).
rho          ! Bending radius. Dependent attribute (§4.1).
```

`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.

3.44 X_Ray_Init

A `x_ray_init` element is used to simulate a general purpose x-ray source.

General `x_ray_init` attributes are:

Attribute Class	Section	Attribute Class	Section
Aperture Limits	4.6	Length	4.10
Chamber wall	4.9	Offsets, pitches, & tilt	4.4
Description strings	4.2	Reference energy	4.3
Integration settings	5.4	Tracking & transfer map	5

Attributes specific to an `x_ray_init` element are:

```
x_half_length = <Real>      ! Horizontal strong beam sigma
x_half_length = <Real>      ! Horizontal strong beam sigma
```


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. Some of 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 §8.1). `l_lattice` is calculated when the lattice is read in.

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

Table 4.1: Partial listing 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 (§2.19), *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.

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 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

<i>Element</i>	<i>Dependent Variable Set</i>	<i>Independent Variables Not Set</i>
Lcavity	voltage	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.

<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.

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 40 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 Energy and Wavelength Attributes: E_tot, P0C, and ref_wavelength

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, and for photons, these two values are the same (§13.4). Except for multipass elements (§7.2), e_tot and p0c are dependent attributes and, except for multipass elements, any setting of e_tot and p0c in the lattice input file is an error. The value of e_tot and p0c for an element is calculated by *Bmad* to be the same as the previous element except for e_gun, lcavity and patch elements. To set the e_tot or p0c at the start of the lattice use the `beginning` or `parameter` statements. See §8.1. Since the energy changes from the start to the end of an lcavity or. em_field, an lcavity or em_field has the dependent attributes

```
e_tot_start and  
p0c_start
```

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

The `beginning_ele` element (§3.3) also has associated `e_tot_start` and `p0c_start` attributes as well as `e_tot` and `p0c`. Generally, for an `beginning_ele`, `p0c_start` and `p0c` are the same and `e_tot_start` and `e_tot` are the same and the values for these attributes are set in the lattice file with the appropriate `parameter` (§8.1) or `beginning` (§8.4) statement. The exception occurs when there is an `e_gun` element in the lattice (§3.13). In this case, the `p0c_start` and `e_tot_start` attributes of the `beginning_ele` are set to the values as set in the lattice file and `e_tot` is set to

```
e_tot = e_tot_start + voltage
```

and `p0c` is calculated from `e_tot` and the mass of the particle being tracked. For example, if the lattice file contained:

```
beginning[p0c] = 0  
gun: e_gun, voltage = 0.5e6  
injector: line = (gun, ...)
```

Then the following energy values will be set for the beginning `beginning_ele` element:

```
p0c_start = 0  
e_tot_start = mc2  
e_tot      = mc2 + 0.5e6  
p0c       = Sqrt(e_tot - mc2^2)
```

where `mc2` is the particle rest mass. The reason for using this convoluted convention is to allow the setting, in the lattice file, of a zero reference momentum at the start of the lattice, while avoiding the calculational problems that would occur if the `e_gun` element truly had a starting reference momentum of zero. Specifically, the problem with zero reference momentum is that the phase space momentum would be infinity as can be seen from Eqs. (13.25).

For `multipass` elements, the reference energy is set by specifying one of `e_tot`, `p0c`, or `n_ref_pass` as described in §7.2.

For photons, the reference wavelength, `ref_wavelength` is also a dependent attribute calculated from the reference energy.

4.4 Orientation: Offset, Pitch, Tilt, and Roll Attributes

By default, an element, like a quadrupole, is aligned in space coincident with the reference orbit running through it (§13.2). A quadrupole can be displaced in space using the quadrupole’s “orientational” attributes. For a quadrupole, the orientational attributes only affect the physical element and not the reference orbit. However, the orientational attributes of some other elements, like the `fiducial` element, do affect the reference orbit. To sort all this out, lattice elements can be divided into seven classes:

1. Straight line elements (§4.4.1)

Straight line elements are elements where the reference orbit is a straight line. Examples include `quadrupoles`, and `sextupoles` as well as zero length elements like `markers`.

2. Dipole bends (§4.4.2)

Dipole bends are:

`sbend` & `rbend`

3. Photon reflecting elements (§4.4.3)

The reflecting elements are

`crystal`
`mirror`
`multilayer_mirror`

These elements have a kink in the reference orbit at the nominal element surface.

4. Reference orbit manipulator elements (§4.4.4)

Elements that are used to manipulate the reference orbit are

`fork` & `photon_fork`
`floor_shift`
`patch`

5. Fiducial Element (§4.4.5)

6. Girder Elements (§4.4.6)

7. Control Elements

Control elements are elements that control attributes of other elements. The control elements are:

`group`
`overlay`

These elements do not have orientational attributes.

4.4.1 Straight Line Element Orientation

The straight line elements have the following orientational attributes:

```
x_offset = <Real>
y_offset = <Real>
z_offset = <Real>
x_pitch = <Real>
y_pitch = <Real>
tilt      = <Real>
```

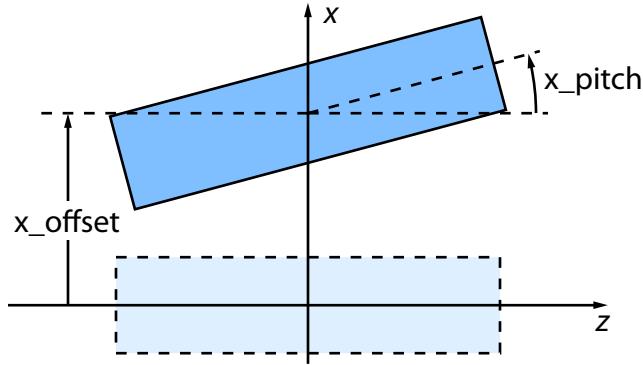


Figure 4.1: Geometry of Pitch and Offset attributes

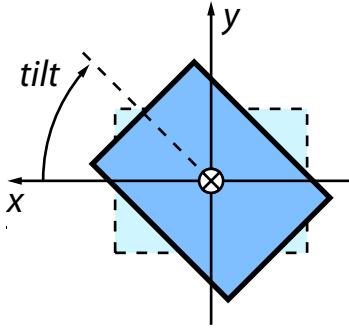


Figure 4.2: Geometry of a Tilt

For straight line elements the orientational attributes only shift the physical element and do not affect the reference orbit.

`x_offset` translates an element in the local x -direction as shown in Fig. 4.1. Similarly, `y_offset` and `z_offset` translate an element along the local y and z -directions respectively.

The `x_pitch` attribute rotates an element about the element's center such that the exit face of the element is displaced in the $+x$ -direction as shown in figure 4.1. An `x_pitch` represents a rotation around the positive y -axis.

Similarly, the `y_pitch` attribute rotates an element about the element's center using the negative x -axis as the rotation axis so that the exit face of the element is displaced in the $+y$ -direction.

The `x_pitch` and `y_pitch` 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. The sense of the rotation between `Bmad` and MAD is:

$$\begin{aligned} \text{x_pitch (Bmad)} &= \text{dtheta (MAD)} \\ \text{y_pitch (Bmad)} &= -\text{dphi (MAD)} \end{aligned}$$

The tilt attribute rotates the element in the (x, y) plane as shown in figure 4.2. The rotation axis is the positive z -axis. For example

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

Like MAD, `Bmad` allows the use of the `tilt` attribute without a value to designate a skew element. The default tilt is $\pi/(2(n + 1))$ where n is the order of the element:

```
sol_quad      n = 1
```

```

quadrupole    n = 1
sextupole     n = 2
octupole      n = 3

```

Note that `hkick` and `vkick` attributes are not affected by `tilt` except for `kicker` and `elseparator` elements.

4.4.2 Bend Element Orientation

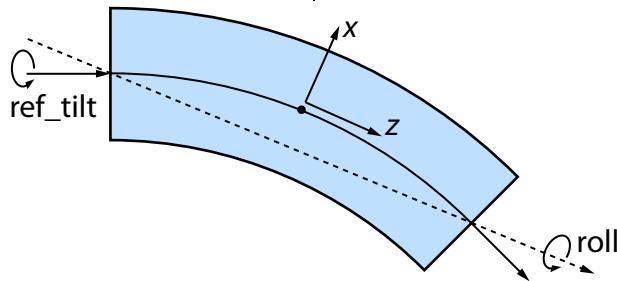


Figure 4.3: Geometry of a Bend. Like straight line elements, offsets and pitches are calculated with respect to the coordinates at the center of the bend. The exception is the `roll` attribute which is a rotation around the axis passing through the entrance and exit points. Shown here is the geometry for a bend with `ref_tilt` = 0. That is, the bend is in the $x - z$ plane.

The orientation attributes for `sbend` and `rbend` elements is

```

x_offset = <Real>
y_offset = <Real>
z_offset = <Real>
x_pitch = <Real>
y_pitch = <Real>
ref_tilt = <Real> ! Shifts and reference orbit rotation axis.
roll     = <Real>

```

The geometry for orienting a bend is shown in Fig. 4.3. Like straight line elements, the offset and pitch attributes are evaluated with respect to the center of the element.

Unlike the straight line elements, bends do not have a `tilt` attribute. Rather they have a `ref_tilt` and a `roll` attribute. The `roll` attribute rotates the bend along an axis that runs through the entrance point and exit point as shown in figure 4.3. A `roll` attribute, like the offset and pitch attributes does not affect the reference orbit. The major effect of a `roll` is to give a vertical kick to the beam. For 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.

The `ref_tilt` attribute of a bend rotates the bend about the z axis at the upstream end of the bend as shown in Fig. 4.3. Unlike `rolls` and `tilts`, `ref_tilt` also shifts the rotation axis of the reference orbit along with the physical element. A bend with a `ref_tilt` of $\pi/2$ will bend a beam vertically downward (§13.3). Note that the `ref_tilt` attribute of `Bmad` is the same as the MAD `tilt` attribute.

4.4.3 Photon Reflecting Element Orientation

Photon reflecting elements have the following orientational attributes:

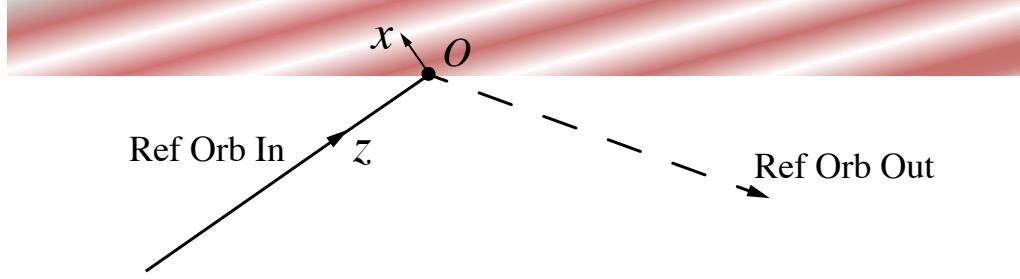


Figure 4.4: Geometry of a photon reflecting element orientation. The reference coordinates used for defining the orientational attribute is the entrance reference coordinates.

```

x_offset = <Real>
y_offset = <Real>
z_offset = <Real>
x_pitch = <Real>
y_pitch = <Real>
ref_tilt = <Real> ! Shifts both element and reference orbit.
tilt     = <Real>

```

Roughly, these elements can be viewed as zero length bends except, since there is no center position, the orientational attributes are defined with respect to the entrance coordinates as shown in Fig. 4.4. Like bend elements, the `ref_tilt` attribute rotates both the physical element and the reference coordinates. The `tilt` attribute rotates just the physical element. Thus the total rotation of the physical element about the entrance `z` axis is the sum `tilt + ref_tilt`.

4.4.4 Reference Orbit Manipulator Element Orientation

The `fork`, `photon_fork`, `floor_shift`, and `patch` elements use the following attributes to orient their exit edge with respect to their entrance edge:

```

x_offset = <Real>
y_offset = <Real>
z_offset = <Real>
x_pitch = <Real>
y_pitch = <Real>
tilt    = <Real>

```

Here "exit" edge for `fork` and `photon_fork` elements is defined to be the start of the line being branched to. [Within the line containing the fork, the `fork` element is considered to have zero length so the exit face in the line containing the fork is coincident with the entrance face.] The placement of the exit edge for these elements defines the reference orbit. Thus, unlike the corresponding attributes for other elements, the orientational attributes here directly control the reference orbit.

4.4.5 Fiducial Element Orientation

The `fiducial` element (§3.19) uses the following attributes to define its position:

```

origin_ele      = <Name> ! Reference element.
origin_ele_ref_pt = <location> ! Reference pt on reference ele.
dx_origin      = <Real>   ! x-position offset

```

```

dy_origin      = <Real>      ! y-position offset
dz_origin      = <Real>      ! z-position offset
dtheta_origin   = <Real>      ! orientation angle offset.
dphi_origin     = <Real>      ! orientation angle offset.
dpsi_origin     = <Real>      ! orientation angle offset.

```

See Section §3.16 for more details.

4.4.6 Girder Orientation

A **girder** (§3.19) element uses the same attributes as a **fiducial** element (§3.16) to orient the reference girder position. In addition, the following attributes are used to move the girder physically from the reference position:

```

x_offset = <Real>
y_offset = <Real>
z_offset = <Real>
x_pitch  = <Real>
y_pitch  = <Real>
tilt     = <Real>

```

Shifting the girder from its reference position shifts all the elements that are supported by the girder. See Section §3.19 for more details.

If an element is supported by a **girder** element (§3.19), the orientational attributes of the element are with respect to the orientation of the **girder**. The computed offsets, pitches and tilt with respect to the local reference coordinates are stored in the dependent attributes

```

x_offset_tot
y_offset_tot
z_offset_tot
x_pitch_tot
y_pitch_tot
tilt_tot
roll_tot

```

A ***_tot** attribute will only be present if the corresponding non ***_tot** attribute is present. For example, only **sbend** and **rbend** elements have a **roll_tot** attribute since only these elements have a **roll** attribute.

If an element is not supported by a **girder**, the values of the ***_tot** attributes will be the same value as the values of the corresponding non ***_tot** attributes.

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
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).

For an `elseparator` element, the `hkick` and `vkick` are appropriate for a positively charged particle. The kick for a negatively charged particle is opposite this.

4.6 Aperture and Limit Attributes

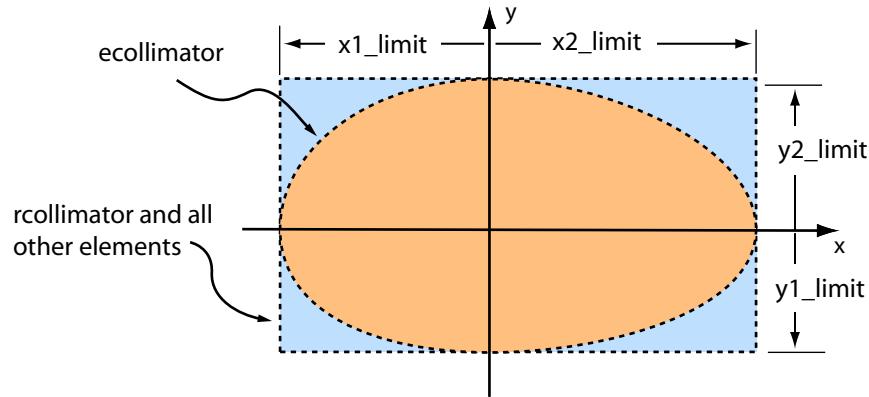


Figure 4.5: Apertures for ecollimator and rcollimator elements. Positive s points up out of the page.

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 4.5. 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:

```

auto          ! Default for diffraction_plate elements
custom
elliptical    ! Default for ecollimator elements
rectangular   ! Default for most elements.

```

The `custom` setting is used in the case where programs have been compiled with custom, non-Bmad, code to handle the aperture calculation. The `auto` setting is used for automatic calculation of a rectangular aperture. For a `diffraction_plate` element, the `auto` setting causes the four aperture limits to be set to just cover the clear area of a `diffraction_plate` (§4.9.6). For all other elements, the `auto` setting is

only to be used when there is an associated surface grid (§4.8.1) for the element and, in this case, *Bmad* to set the four limits to just cover the surface grid

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
```

Examples:

```
q1, quadrupole, 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
```

4.6.1 Apertures and Element Offsets

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 placement of the physical element the beam pipe passes through. That is, the beam pipe does not “touch” the physical element. 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 the following elements is for `offset_moves_aperture` to be `True`:

```
rcollimator,
ecollimator,
multilayer_mirror,
mirror, and
crystal
```

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. Additionally, when the aperture is at the `surface` (see below), any tilt will be used in the calculation.

Examples:

```
q1: quad, l = 0.6, x1_limit = 0.045, offset_moves_aperture = T
```

4.6.2 Aperture Placement

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

```
entrance_end
exit_end      ! Default for most elements
both_ends
```

```
continuous
surface
no_aperture
```

The `exit_end` setting is the default for most elements except for the following elements who have a default of `surface`:

```
crystal
diffraction_plate
mirror
multilayer_mirror
sample
```

In fact, for the following elements:

```
mirror,
multilayer_mirror
crystal
```

The `surface` setting for `aperture_at` must be used. Additionally, due to the complicated geometry of these elements, to keep things conceptionally simple, the rule is imposed that, for an aperture at the surface, the `offset_moves_aperture` setting must be left in its default state of True. Additionally, For `entrance_end` or `exit_end` apertures, `offset_moves_aperture` must be set to False.

Note: 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”. The `continuous` setting indicates that the aperture is continuous along the length of the element. This only matters when particle tracking involves stepping through an element a little bit at a time. For example, as in Runge-Kutta tracking (§5.1). For tracking where a formula is used to transform the particle coordinates at the entrance of an element to the coordinates at the exit end, the aperture is only checked at the end points so, in this situation, a `continuous` aperture is equivalent to the `both_ends` setting.

Examples:

```
q2: quad, aperture_type = elliptical, aperture_at = continuous
q1: quad, l = 0.6, x1_limit = 0.045, offset_moves_aperture = T
```

4.6.3 Apertures and X-Ray Generation

With X-ray simulation apertures can be used by *Bmad* to limit the directions in which photons are generated. This can greatly decrease simulation times. For example, a photon passing through a `diffraction_plate` element will diffract in an arbitrary direction. If a `downstream` element has an aperture set, *Bmad* can restrict the velocity directions so that the photons will fill the downstream aperture and the amount of time wasted tracking photons that ultimately would be collimated is minimal.

4.7 X-Rays Crystal & Compound Materials

For basic crystallographic and X-ray matter interaction cross-sections, *Bmad* uses the XRAYLIB[Schoon11] library. Crystal structure parameters in XRAYLIB are mainly from R. W. G. Wyckoff[Wyckoff65] with some structure parameters coming from NIST. The list of available structures is:

AlphaAlumina	GaP	KCl	Platinum
AlphaQuartz	GaSb	KTP	RbAP
Aluminum	Ge	LaB6	Sapphire
Be	Gold	LaB6_NIST	Si
Beryl	Graphite	LiF	Si_NIST
Copper	InAs	LiNbO3	Si2
CsCl	InP	Muscovite	SiC
CsF	InSb	NaCl	Titanium
Diamond	Iron	PET	TlAP
GaAs	KAP		

These names are case sensitive

Besides the above crystal list, *Bmad* can calculate structure factors for all the elements and the following list of materials. Material properties are from NIST. These names are case sensitive. That is, the NIST materials all use upper case. As noted in the table, several of the materials may be specified using the appropriate chemical formula. For example, liquid water may be referenced using the name H2O.

A_150_TISSUE_EQUIVALENT_PLASTIC	LITHIUM_TETRABORATE
ACETONE	LUNG_ICRP
ACETYLENE	M3_WAX
ADENINE	MAGNESIUM_CARBONATE
ADIPOSE_TISSUE_ICRP	MAGNESIUM_FLUORIDE
AIR_DRY_NEAR_SEA_LEVEL	MAGNESIUM_OXIDE
ALANINE	MAGNESIUM_TETRABORATE
ALUMINUM_OXIDE, Al2O3	MERCURIC_IODIDE
AMBER	METHANE
AMMONIA, NH3	METHANOL
ANILINE	MIX_D_WAX
ANTHRACENE	MS20_TISSUE_SUBSTITUTE
B_100_BONE_EQUIVALENT_PLASTIC	MUSCLE_SKELETAL
BAKELITE	MUSCLE_STRIATED
BARIUM_FLUORIDE	MUSCLE_EQUIVALENT_LIQUID_WITH_SUCROSE
BARIUM_SULFATE	MUSCLE_EQUIVALENT_LIQUID_WITHOUT_SUCROSE
BENZENE, C6H6	NAPHTHALENE
BERYLLIUM_OXIDE	NITROBENZENE
BISMUTH_GERMANIUM_OXIDE	NITROUS_OXIDE
BLOOD_ICRP	NYLON_DU_PONT_ELVAMIDE_8062
BONE_COMPACT_ICRU	NYLON_TYPE_6_AND_TYPE_6_6
BONE_CORTICAL_ICRP	NYLON_TYPE_6_10
BORON_CARBIDE, B4C	NYLON_TYPE_11_RILSAN
BORON_OXIDE, B2O3	OCTANE_LIQUID
BRAIN_ICRP	PARAFFIN_WAX
BUTANE	N_PENTANE
N_BUTYL_ALCOHOL	PHOTOGRAPHIC_EMULSION
C_552_AIR_EQUIVALENT_PLASTIC	PLASTIC_SCINTILLATOR_VINYLTOLUENE_BASED
CADMIDIUM_TELLURIDE	PLUTONIUM_DIOXIDE
CADMIDIUM_TUNGSTATE	POLYACRYLONITRILE
CALCIUM_CARBONATE	POLYCARBONATE_MAKROLON_LEXAN
CALCIUM_FLUORIDE	POLYCHLOROSTYRENE
CALCIUM_OXIDE	POLYETHYLENE
CALCIUM_SULFATE	POLYETHYLENE_TEREPHTHALATE_MYLAR
CALCIUM_TUNGSTATE	POLYMETHYL METHACRALATE_LUCITE_PERSPEX
CARBON_DIOXIDE	POLYOXYMETHYLENE
CARBON_TETRACHLORIDE	POLYPROPYLENE
CELLULOSE_ACETATE_CELLOPHANE	POLYSTYRENE
CELLULOSE_ACETATE_BUTYRATE	POLYTETRAFLUOROETHYLENE_TEFLON
CELLULOSE_NITRATE	POLYTRIFLUOROCHLOROETHYLENE
CERIC_SULFATE_DOSIMETER SOLUTION	POLYVINYL_ACETATE

Continued on next page

CESIUM_FLUORIDE	POLYVINYL_ALCOHOL
CESIUM_IODIDE	POLYVINYL_BUTYRAL
CHLOROBENZENE	POLYVINYL_CHLORIDE
CHLOROFORM	POLYVINYLDENE_CHLORIDE_SARAN
CONCRETE_PORTLAND	POLYVINYLDENE_FLUORIDE
CYCLOHEXANE	POLYVINYL_PYRROLIDONE
12_DDIHLOROBENZENE	POTASSIUM_IODIDE
DICHLORODIETHYL_ETHER	POTASSIUM_OXIDE
12_DICHLOROETHANE	PROPANE
DIETHYL_ETHER	PROPANE_LIQUID
NN_DIMETHYL_FORMAMIDE	N_PROPYL_ALCOHOL
DIMETHYL_SULFOXIDE	PYRIDINE
ETHANE	RUBBER_BUTYL
ETHYL_ALCOHOL	RUBBER_NATURAL
ETHYL_CELLULOSE	RUBBER_NEOPRENE
ETHYLENE	SILICON_DIOXIDE
EYE_LENS_ICRP	SILVER_BROMIDE
FERRIC_OXIDE	SILVER_CHLORIDE
FERROBORIDE	SILVER_HALIDES_IN_PHOTOGRAPHIC_EMULSION
FERROUS_OXIDE	SILVER_IODIDE
FERROUS_SULFATE_DOSIMETER_SOLUTION	SKIN_ICRP
FREON_12	SODIUM_CARBONATE
FREON_12B2	SODIUM_IODIDE
FREON_13	SODIUM_MONOXIDE
FREON_13B1	SODIUM_NITRATE
FREON_13I1	STILBENE
GADOLINIUM_OXYSULFIDE	SUCROSE
GALLIUM_ARSENIDE	TERPHENYL
GEL_IN_PHOTOGRAPHIC_EMULSION	TESTES_ICRP
GLASS_PYREX	TETRACHLOROETHYLENE
GLASS LEAD	THALLIUM_CHLORIDE
GLASS_PLATE	TISSUE_SOFT_ICRP
GLUCOSE	TISSUE_SOFT_ICRU_FOUR_COMPONENT
GLUTAMINE	TISSUE_EQUIVALENT_GAS_METHANE_BASED
GLYCEROL	TISSUE_EQUIVALENT_GAS_PROPANE_BASED
GUANINE	TITANIUM_DIOXIDE
GYPSUM_PLASTER_OF_PARIS	TOLUENE
N_HEPTANE	TRICHLOROETHYLENE
N_HEXANE	TRIETHYL_PHOSPHATE
KAPTON_POLYIMIDE_FILM	TUNGSTEN_HEXAFLUORIDE
LANTHANUM_OXYBROMIDE	URANIUMDICARBIDE
LANTHANUM_OXYSULFIDE	URANIUM_MONOCARBIDE
LEAD_OXIDE	URANIUM_OXIDE
LITHIUM_AMIDE	UREA
LITHIUM_CARBONATE	VALINE
LITHIUM_FLUORIDE	VITON_FLUOROELASTOMER
LITHIUM_HYDRIDE	WATER_LIQUID, H ₂ O
LITHIUM_IODIDE	WATER_VAPOR
LITHIUM_OXIDE	XYLENE

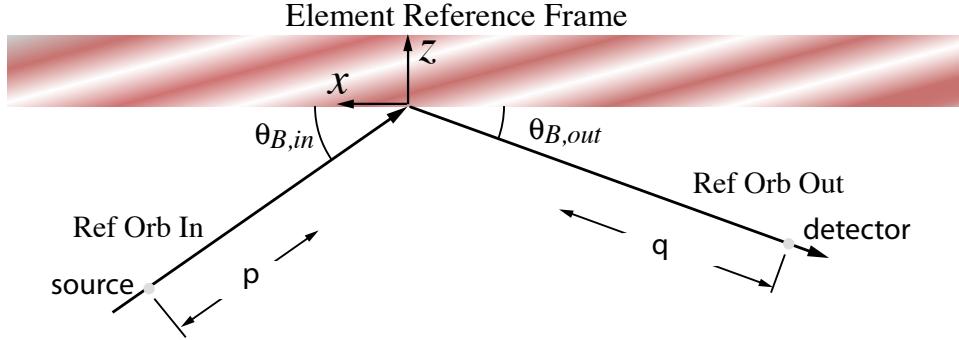


Figure 4.6: Surface curvature geometry. The element reference frame used to describe surface curvature has the x axis pointing towards the interior of the element, and the z axis along the plane defined by the entrance and exit reference orbit.

4.8 Surface Properties for X-Ray elements

The following X-ray elements have a surface which X-rays impinge upon:

crystal	§3.8
detector	§3.10
diffraction_plate	§3.11
mirror, and	§3.28
multilayer_mirror	§3.30
sample	§3.38

[There is also the `capillary` element but this element specifies its surface differently.]

The coordinate system used for characterizing the curvature of a surface is the element reference frame as shown in Fig. 4.6). This coordinate system has the x axis pointing towards the interior of the element, and the z axis along the plane defined by the entrance and exit reference orbit. In this coordinate system, the surface curvature is parameterized by a fourth order polynomial in z and y

$$-z = \sum_{2 \leq i+j \leq 4} c_{ij} x^i y^j \quad (4.1)$$

The coefficients are set in the lattice file by setting the following element attributes

`curvature_xM_yN` = <Real>

where M and N are integers in the range 0 through 4 with the restriction

$$2 \leq M + N \leq 4$$

Example:

c2: crystal, curvature_x2_y0 = 37, ...

in this example, `curvature_x2_y0` corresponds to the c_{20} term in Eq. (4.1). To get the effect of a nonzero $x^0 y^0$, $x^1 y^0$, or $x^0 y^1$ terms (since corresponding `curvature_xN_yM` are not permitted), element offsets and pitches can be used (§4.4).

Some useful formulas: Series expansion for a circle of radius R :

$$-z = \frac{x^2}{2R} + \frac{x^4}{8R^3} + \frac{x^6}{16R^5} + \frac{y^2}{2R} + \frac{y^4}{8R^3} + \frac{y^6}{16R^5} + \frac{x^2 y^2}{4R^3} + \frac{3x^4 y^2}{16R^5} + \frac{3x^2 y^4}{16R^5} \quad (4.2)$$

If p is the distance from the source to the crystal, and q is the distance from the crystal to the detector, the radius of the Rowland circle R_s in the sagittal plane is given by [Rio98]

$$\frac{1}{p} + \frac{1}{q} = \frac{\sin \theta_{g,in} + \sin \theta_{g,out}}{R_s} \quad (4.3)$$

where $\theta_{g,in}$ and $\theta_{g,out}$ are the entrance and exit graze angles. In the transverse plane (also called meridional plane), the radius R_t needed for focusing is

$$\frac{\sin^2 \theta_{g,in}}{p} + \frac{\sin^2 \theta_{g,out}}{q} = \frac{\sin \theta_{g,in} + \sin \theta_{g,out}}{R_t} \quad (4.4)$$

Example:

```
t_bragg = 1.3950647
rt = 1 ! Crystal transverse radius
rs = rt*(sin(t_bragg))^2
c: crystal, crystal_type = 'Si(553)', b_param = -1,
    curvature_x0_y2 = 1 / (2 * rs), curvature_x0_y4 = 1 / (8 * rs^3),
    curvature_x2_y0 = 1 / (2 * rt), curvature_x4_y0 = 1 / (8 * rt^3),
```

4.8.1 Surface Grid

A surface can be broken up into a grid of rectangles. This is useful, for example, in breaking up a **detector** element into pixel photo receptors or in simulating a rough surface for **crystals** and other elements. The general syntax is:

```
surface = {
    grid = {
        type = <type_name>,           ! Crystals: Off, Segmented, or H_Misalign
        ix_bounds = (<ix_min>, <ix_max>), ! Min/max index bounds in x-direction
        iy_bounds = (<iy_min>, <iy_max>), ! Min/max index bounds in y-direction
        r0 = (<x0>, <y0>),           ! (x,y) coordinates at grid origin
        dr = (<dx>, <dy>),           ! width and height of pixels.
        pt(<i>,<j>) = (<x_pitch>, <y_pitch>, <x_pitch_rms>, <y_pitch_rms>),
    } }
```

Example:

```
ccd: crystal, surface = {
    grid = {
        type = h_misalign,
        r0 = (0.0, 0.01), dr = (0.005, 0.005),
        ix_bounds = (1, 57), iy_bounds = (-30, 10),
        pt(1,-30) = (0.001, -0.002, 0, 0),
        pt(1,-29) = ...,
    } }
```

The grid is a two dimensional with bounds given by the **ix_bounds** and **iy_bounds** components. These two components must be present. In the above example the grid is 57 pixels in *x* and 41 pixels in *y*.

The physical placement of the grid on the element is determined by the **r0** and **dr** components. **r0** is optional and gives the (*x*, *y*) coordinates of the center of the pixel with index (0, 0). The **dr** component, which must be present, gives the pixel width and height. Thus the center of the (*i*, *j*) pixel is:

$$(x, y) = (r0(1), r0(2)) + (i*dr(1), j*dr(2))$$

The **type** component of the grid is used for **crystal** elements only. Possible **type** values are:

H_Misalign	<small>! Misalignment of crystal H vector</small>
Off	<small>! Ignore grid</small>
Segmented	<small>! Surface is a matrix of flat rectangles</small>

H_Misalign misaligns the *H* vector which is the normal to the diffracting planes of the the crystal (§20.4). When using **H_Misalign**, each **pt(*i*,*j*)** component gives the misalignment of *H* for the corresponding pixel. For an individual photon, the misalign of *H* will be

```
x_pitch_tot = <x_pitch> + r1 * <x_pitch_rms>
y_pitch_tot = <y_pitch> + r2 * <y_pitch_rms>
```

where `x_pitch_tot` and `y_pitch_tot` are the rotational misalignment (§4.4) used in the calculation, the quantities in brackets `<...>` are components of `pt`, and `r1` and `r2` are Gaussian distributed random numbers with unit rms. These random numbers are regenerated for each photon. Note: `pt` is only used with `H_Misalign`.

When the `type` component is set to `Segmented`, the crystal surface is modeled as a grid of flat “rectangles” (the actual shape is very close but not quite rectangular). Using a segmented crystal only makes sense when the crystal is curved. There is one rectangle for each pixel. Each rectangle has an extent in the (x, y) transverse dimensions equal to the extent of the corresponding pixel. The z coordinate of the vertices of the rectangular are adjusted so that

- 1) The rectangle is flat
- 2) The rectangle contacts the unsegmented surface two diagonally opposite vertices.
- 3) The other two diagonally opposite vertices will be as close as possible in the least squares sense from the unsegmented surface.

When the `type` component is set to `Off`, the grid will not be used.

4.9 Walls: Vacuum Chamber, Capillary and Diffraction Plate

The `wall` attribute for an element is used to define:

```
vacuum_chamber wall
capillary element (§3.6) inside wall
diffraction_plate (§3.11) geometry
```

The topics of the following subsections are:

- §4.9.1 General wall syntax.
- §4.9.2 Cross-section construction.
- §4.9.3 Capillary and vacuum chamber wall interpolation.
- §4.9.4 Capillary specific.
- §4.9.5 Vacuum chamber specific.
- §4.9.6 Diffraction_plate specific.

4.9.1 Wall Syntax

The syntax of the `wall` attribute is:

```
wall = {
    superimpose = <T/F>,                                ! Chamber wall only
    thickness = <real>                                   ! Default thickness.
    opaque_material = <material_type>                  ! Default opaque material.
    clear_material = <material_type>                   ! Default clear material.
    section = {
        type = <section_type>,                           ! Chamber and Diffraction_plate only
        s = <longitudinal_position>,                     ! Relative to beginning of element.
        x0 = <value>, y0 = <value>,                      ! Section origin
        material = <material_type>                      ! Diffraction_plate only.
        thickness = <real>                             ! Diffraction_plate only.
        dr_ds = <value>,                               ! Capillary and Chamber only
        v(1) = {<x>, <y>, <radius_x>, <radius_y>, <tilt>},
```

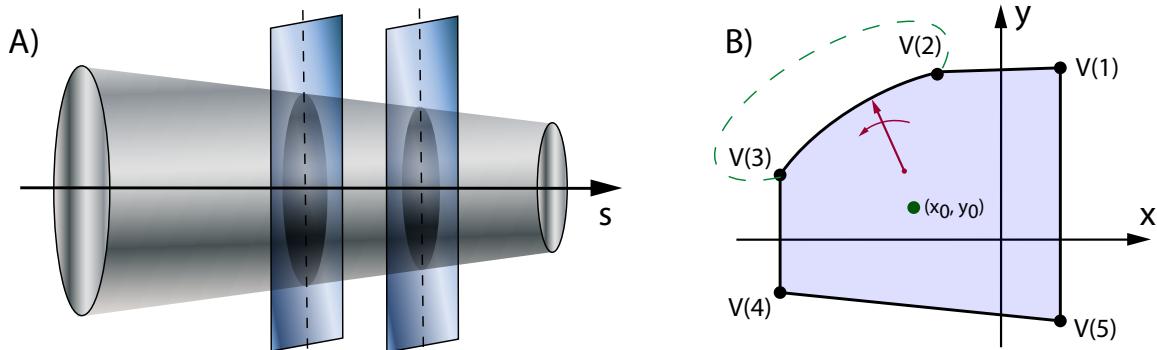


Figure 4.7: A) The inside wall of a capillary or the vacuum chamber wall of a non-capillary element is defined by a number of cross-sectional slices. B) Each cross-section is made up of a number of vertices. The segments between the vertices can be either a line segment, the arc of a circle, or a section of an ellipse.

```

v(2) = { ... },
...},
section = {
  s = <longitudinal_position>,
  v(1) = {... },
  ... },
  ...
}

```

A wall begins with “wall = {” and ends with a “}”. In between are a number of individual cross-section structures. Each individual cross-section begins with “section = {” and ends with a “}”. The *s* parameter of a cross-section gives the longitudinal position of the cross-section. Example:

```

this_cap: capillary,
wall = {
  section = { ! cross-section with top/bottom symmetry
    s = 0, v(1) = {0.02, 0.00},
    v(2) = {0.00, 0.02, 0.02}, v(3) = {-0.01, 0.01} },
  section = { ! Cross-section that is a tilted ellipse.
    s = 0.34,
    v(1) = {0.003, -0.001, 0.015, 0.008, 0.2*pi} } }

```

In this example an element called *this_cap* is a capillary whose wall is defined by two cross-sections.

4.9.2 Wall Sections

The wall is defined by a number of cross-sectional slices. For Fig. 4.7A shows the geometry for capillary or vacuum chamber walls. Each cross-section is defined by a longitudinal position *s* relative to the beginning of the element and a number of vertices. The vertices are defined with respect to the local sector origin (x_0, y_0) . The arc between each vertex may be either a straight line, an arc of a circle, or a section of an ellipse. For a capillary it is mandatory that a cross-section be convex. That is, given any two points within the cross-section, all points on the line segment connecting them must be within the cross-section.

The $v(<j>)$ within a cross-section define the vertices for each cross-section. The vertices are defined with respect to the section origin given by x_0 and y_0 . Each $v(<j>)$ has five parameters. It is mandatory to

specify the first two parameters $\langle x \rangle$ and $\langle y \rangle$. Specifying the rest, $\langle radius_x \rangle$, $\langle radius_y \rangle$, and $\langle tilt \rangle$, is optional. The default values, if not specified, is zero. The point $(\langle x \rangle, \langle y \rangle)$ defines the position of the vertex. The parameters $\langle radius_x \rangle$, $\langle radius_y \rangle$, and $\langle tilt \rangle$ define the shape of the segment of the cross-section between the given vertex and the preceding one.

```
<radius_x> = 0, <radius_y> = 0    --> Straight line segment.
<radius_x> != 0, <radius_y> = 0    --> Circular arc with radius = radius_x
<radius_x> = 0, <radius_y> != 0    --> Illegal!
<radius_x> != 0, <radius_y> != 0   --> Ellipse section.
```

When an ellipse is specified, $\langle radius_x \rangle$, and $\langle radius_y \rangle$ are the half width and half height of the semi-major axes and the $\langle tilt \rangle$ parameter gives the tilt of the ellipse. $\langle radius_x \rangle$ and $\langle radius_y \rangle$ must not be negative.

In the example above, for the first cross-section, $v(2)$ specifies a non-zero $\langle radius_x \rangle$ and, by default, $\langle radius_y \rangle$ is zero. Thus the segment of the cross-section between $v(1)$ and $v(2)$ is circular in nature with a radius of 0.02. Since $v(3)$ does not specify $\langle radius_x \rangle$ nor $\langle radius_y \rangle$, the cross-section between $v(2)$ and $v(3)$ is a straight line segment.

The vertex points must be arranged in a “counter clockwise manner”. For vertices $\langle v(i) \rangle$ and $\langle v(i+1) \rangle$ connected by a line segment this translates to

$$0 < \theta_{i+1} - \theta_i \pmod{2\pi} < \pi \quad (4.5)$$

where (r_n, θ_n) are the polar coordinates of the n^{th} vertex. For vertices connected by an arc, “counter clockwise manner” means that the line segment with one end at the center of the arc and the other end traversing the arc from $\langle v(i) \rangle$ to $\langle v(i+1) \rangle$ rotates in counter clockwise as shown in Fig. 4.7B.

The red line segment with one end at the center of the arc and the other end traversing the arc from, in this case, $V(2)$ to $V(3)$, rotates in counter clockwise manner. In general, there are two solutions for constructing such an arc. For positive radii, the solution chosen is the one whose center is closest to the section origin (x_0, y_0) . If the radii are negative, the center point will be the point farthest from the origin (the dashed line between $V(2)$ and $V(3)$ in the figure).

A restriction on cross-sections is that the section origin (x_0, y_0) must be in the interior of any cross-section and that for any cross-section a line drawn from the origin at any given angle θ will intersect the cross-section at exactly one point as shown in Fig. 4.7B. This is an important point in the construction of the wall between cross-sections as explained below.

The last vertex specified, call it $\langle v(n) \rangle$, should not have the same $\langle x \rangle$, $\langle y \rangle$ values as the first vertex $\langle v(1) \rangle$. That is, there will be a segment of the cross-section connecting $\langle v(n) \rangle$ to $\langle v(1) \rangle$. The geometry of this segment is determined by the parameters of $\langle v(1) \rangle$.

If there is mirror symmetry about the x or y axis for a cross-section, the “mirrored” vertices, on the “negative” side of the mirror plane, do not have to be specified. Thus if all the vertex points of a cross-section are in the first quadrant, that is, all $\langle x \rangle$ and $\langle y \rangle$ are zero or positive, mirror symmetry about both the x and y axes is assumed. If all the $\langle y \rangle$ values are zero or positive and some $\langle x \rangle$ values are positive and some are negative, mirror symmetry about the x axis is assumed. Finally, if all the $\langle x \rangle$ values are zero or positive but some $\langle y \rangle$ values are positive and some are negative, symmetry about the y axis is assumed. For example, for the first in the above example, since all the $\langle y \rangle$ values are non-negative and there are positive and negative $\langle x \rangle$ values, symmetry about the x axis is assumed.

The one exception to the above rule that $(\langle x \rangle, \langle y \rangle)$ is the vertex center is when a single vertex $v(1)$ is specified for a cross-section with a non-zero $\langle radius_x \rangle$. In this case, $(\langle x \rangle, \langle y \rangle)$ are taken to be the center of the circle or ellipse. In the example above, the second cross-section is a tilted ellipse with center at $(0.003, -0.001)$. If a cross-section has a single vertex and $\langle radius_x \rangle$ is not specified, the cross-section is a rectangle. For example

```
section = { s = 0.34, v(1) = {0.03, 0.01} }
```

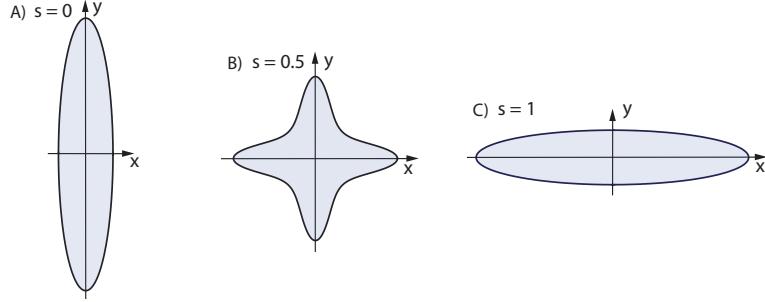


Figure 4.8: Example where convex cross-sections do not produce a convex volume. Cross-sections (A) and (C) are ellipses with a 5 to 1 aspect ratio. Half way in between, linear interpolation produces a convex cross-section as shown in (B).

4.9.3 Interpolation Between Sections

For capillary and vacuum chamber walls, the wall between cross-sections, is defined by interpolation. Let $r_{c1}(\theta)$ be the radius of the wall as a function of θ for a given cross-section defined at $s = s_1$. Let $r_{c2}(\theta)$ be the radius function at the next defined cross-section at $s = s_2$. The wall $r_c(\theta, s)$ at any point s between s_1 and s_2 is defined by the equation

$$r_c(\theta, s) = p_1(\tilde{s}) r_{c1}(\theta) + p_2(\tilde{s}) r_{c2}(\theta) \quad (4.6)$$

where

$$\tilde{s} \equiv \frac{s - s_1}{s_2 - s_1} \quad (4.7)$$

and p_1 and p_2 are cubic polynomials parameterized by

$$\begin{aligned} p_1 &= 1 - \tilde{s} + a_1 \tilde{s} + a_2 \tilde{s}^2 + a_3 \tilde{s}^3 \\ p_2 &= \tilde{s} + b_1 \tilde{s} + b_2 \tilde{s}^2 + b_3 \tilde{s}^3 \end{aligned} \quad (4.8)$$

If $a_i = b_i = 0$ for all $i = 1, 2, 3$, the interpolation is linear and this is the default if either of the parameters `dr_ds1` and `dr_ds2` are not given in the wall definition. These parameters are the slopes of the wall with respect to s at the end points

$$\text{dr_ds1} \equiv \left. \frac{d\bar{r}}{ds} \right|_{s=s_1}, \quad \text{dr_ds2} \equiv \left. \frac{d\bar{r}}{ds} \right|_{s=s_2} \quad (4.9)$$

where \bar{r} is the average r averaged over all θ . When both `dr_ds1` and `dr_ds2` are specified, the a_i and b_i are calculated so that the slopes of the wall match the values of `dr_ds1` and `dr_ds2` along with the constraints.

$$\begin{aligned} p_1(0) &= 1, & p_1(1) &= 0 \\ p_2(0) &= 0, & p_2(1) &= 1 \\ M &\equiv a_1^2 + a_2^2 + a_3^2 + b_1^2 + b_2^2 + b_3^2 \text{ is a minimum} \end{aligned} \quad (4.10)$$

The last constraint ensures a “smooth” transition between the two cross-sections.

To refer to a cross-section parameters after an element has been defined, the following syntax is used:

```
ele_name[wall.section(n).v(j).x] ! x value of jth vertex of nth cross-section
```

4.9.4 Capillary Wall

For a **capillary**, s must be zero for the first cross-section and the length of the capillary is given by the value of s of the last cross-section.

For a **capillary**, in order for *Bmad* to quickly track photons, *Bmad* assumes that the volume between the cross-sections is convex. The volume will be convex if each cross-section $r_c(\theta, s)$ at any given s is convex. Note that it is *not* sufficient for $r_c(\theta, s)$ to be convex at the specified cross-sections as shown in Fig. 4.8. Also note that it is perfectly fine for the total capillary volume to not be convex.

4.9.5 Vacuum Chamber Wall

The vacuum chamber wall is independent of the element apertures (§4.6). Unless a program is specifically constructed, the presence of a vacuum chamber wall will not affect particle tracking.

The vacuum chamber wall defined for an element may be shorter or longer than the element. The vacuum chamber wall for a particular lattice branch is the sum of all the chamber walls of the individual elements. That is, the chamber wall at any given point is determined by interpolation of the nearest sections upstream and downstream to the point. Thus a given lattice element need not contain a **wall** component for the chamber wall to be well defined at the element.

The exception to the above rule is when a **section** has its **type** component set to either:

```
wall_start
wall_end
```

If a section has a **type** of **wall_start**, the region between that section and the previous section will be considered to have no wall. If the **wall_start** section is the first section of the containing lattice branch, and if the lattice branch has an open geometry, then the region of no wall will start at the end of the branch. Similarly, if a section has a **type** of **wall_end**, the region between that section and the next section (or the end of the lattice branch if there is no next section and the branch has an open geometry) will not have a wall.

The chamber walls of any two elements may not overlap. The exception is when the **superimpose** attribute for a wall of an element is set to True. In this case, any other wall cross-sections from any other elements that overlap the superimposed wall are discarded. Superposition of a wall is useful, for example, in introducing mask regions into the wall.

If a branch has a closed geometry (§8.1), wall sections that extend beyond the ends of the branch are “wrapped” around.

If a particle is past the last wall cross-section or before the first wall cross-section, The following rules are used: If the branch has a **closed geometry**, the wall will be interpolated between the last and first cross-sections. If the branch has an **open** geometry, the wall is taken to have a constant cross-section in these regions.

The chamber wall is defined with respect to the local coordinate system (§13.1). That is, in a bend a wall that has a constant cross section is a section of a torus.

Patch elements complicate the wall geometry since the coordinate system at the end of the **patch** may be arbitrarily located relative to the beginning of the patch. To avoid confusion as to what coordinate system a wall section belongs to, **patch** elements are not allowed to define a wall. The wall through a patch is determined by the closest wall sections of neighboring elements. These wall sections need to be placed at the edge of the **patch**.

Each section has a **type** attribute. This attribute is not used for **capillary** elements. For a vacuum

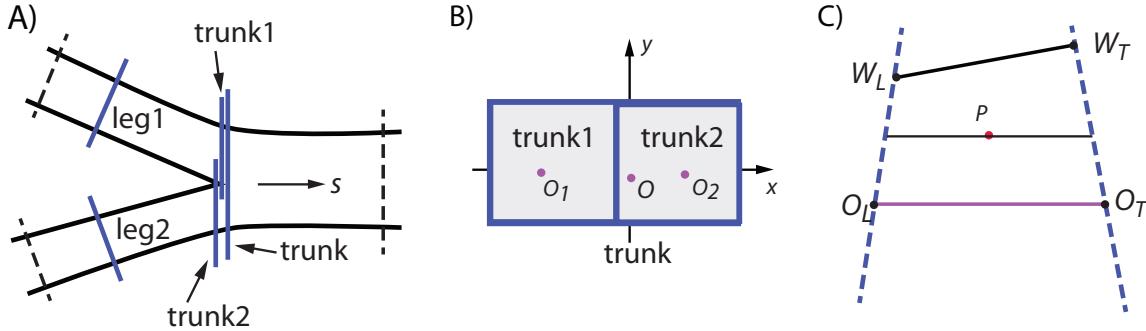


Figure 4.9: A) Crotch geometry: Two pipes labeled “leg1” and “leg2” merge into a single pipe called the “trunk” pipe. Five wall sections are used to define the crotch geometry (solid lines). Dashed lines represent sections not involved in defining the crotch. For purposes of illustration the three trunk sections are displaced longitudinally but in reality must have the same longitudinal coordinate. B) Example layout of the crotch1, crotch2 and crotch wall sections. O_1 , O_2 and O are the x_0, y_0 origins of the sections.

chamber wall, the `type` attribute is used to describe a “crotch” geometry where two pipes merge into one pipe. The possible values for the `type` attribute are:

```
normal      ! default
leg1
leg2
trunk1
trunk2
trunk
```

The geometry of a crotch is shown in Fig. 4.9A. Two pipes, called “leg1” and “leg2”, merge into one pipe called the “trunk” pipe. The trunk pipe can be either upstream or downstream of the leg pipes. To describe this situation, five sections are needed: One section in each leg pipe which need to have their `type` attribute set to `leg1` and `leg2`, and three sections in the trunk with one having a `type` attribute of `trunk1`, another having a `type` attribute of `trunk2` and the third having a `type` attribute of `trunk`. There can be no sections between the leg sections and the trunk sections.

All three trunk sections must be associated with the same element and have the same `s` value. In the list of sections of the element containing the trunk elements, the `trunk1` and `trunk2` sections must be listed first if the leg pipes are upstream of the trunk pipe (the situation shown in the figure) and must be listed last if the leg pipes are downstream. That is, the `trunk1` and `trunk2` sections are “between” the leg sections and the `trunk` section. It does not matter if `trunk1` is before or after `trunk2`.

The `trunk1` and `trunk2` sections must not overlap and the `trunk` section must be constructed so that its area is the union of the areas of `trunk1` and `trunk2`. An example is illustrated in Fig. 4.9B. Here the `trunk1` and `trunk2` sections are squares with origins labeled O_1 and O_2 in the figure. By necessity, these origins must be different since each must lie within the boundaries of their respective areas. The `trunk` section is a rectangle encompassing the two squares and has an origin labeled O .

Between `leg1` and `trunk1` sections the wall is interpolated using these two sections. Similarly for the region between `leg2` and `trunk2` sections. Away from these regions interpolation is done as outlined above but these regions need a different interpolation scheme since, `leg1` and `trunk1`, as well as `leg2` and `trunk2` sections do not have to be parallel to each other.

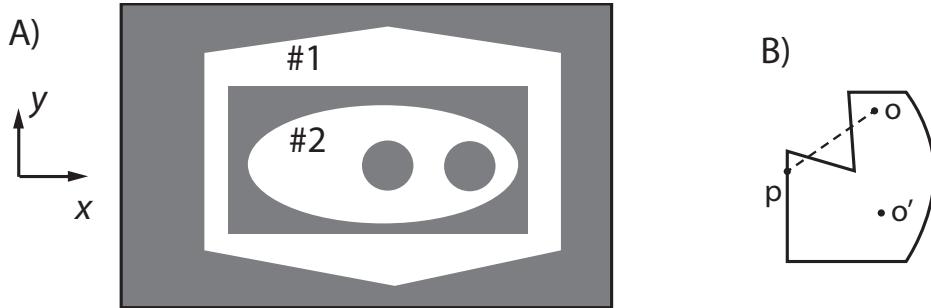


Figure 4.10: A) The diffraction plate surface is divided into “clear” (white) and “opaque” (black) areas. In this example there are two clear sections labeled #1 and #2. B) All wall sections must be star shaped with respect to the section’s origin. In this example, The section is *not* star shaped since a line drawn from the origin point o to the point p on the boundary intersects the boundary twice in between. In this case the section can be made star shaped by moving the origin to o' .

4.9.6 Diffraction Plate Wall

The `wall` of a `diffraction_plate` (§3.11) specifies the topology of the plate in terms of what areas of the plate will transmit or reflect X-rays and what areas will not. The areas where there is transmission or reflection are called “clear” areas and everything else is called “opaque”.

A `wall` is comprised a a ordered list of `sections` as discussed in §4.9.2. Each section of a `diffraction_plate` element must have its `type` attribute set to one of:

```
clear
mask
```

A section is called “clear” or “mask” depending upon the setting of its `type` attribute. Do not confuse “clear section” with “clear area”.

A clear area is defined by one or more consecutive wall sections. The first section that defines a clear area must be a clear section. All the other sections associated with a clear area must be mask sections. That is, a clear area starts with a clear section and any preceeding mask sections up to the next clear section or the end of the section list. As a consequence of the above rules, the fist section of the wall must be a clear section and the number of clear areas is equal to the number of clear sections.

The default behavior is that a photon will be transmitted if it is within any clear area. A photon is considered to be within a given clear area if its (x, y) coordinates put in within the corresponding clear section but not within any `mask` section of the clear area. `Mask` sections only affect the clear area they are associated with. See the example below.

Any clear section can be given a `material` and `thickness`. Available materials are listed in §4.7. A photon transversing a clear area with a defined material will be attenuated and have a phase shift. Note that `material` and `thickness` properties are not to be assigned to `mask` sections.

To enable `Bmad` to quickly calculate whether a photon has landed on a clear or mask section, All sections, both clear and mask, must be “star shaped” with respect to the (x_0, y_0) origin used by the section. That is, a line drawn from the section origin to any point on the section boundary must not pass through any boundary points of the section in between. This is illustrated in Fig. 4.10B where the section is not star shaped since a line drawn from the origin o to the point p on the boundary passes through two boundary points in between. In this case the section can trivially be made star shaped by moving the origin to point o' . If it is not possible to make a section star shaped by moving the origin, the section must be divided into multiple sections.

An example diffraction plate geometry is shown in Fig. 4.10A. In the figure, there are two openings labeled #1, and #2. A wall that constructs this geometry is:

```
zone1: diffraction_plate, wall = {
    thickness = <real>
    opaque_material = <material_type>
    clear_material = <material_type>
    section = {                      ! Clear area # 1
        type = clear,
        v(1) = {0.04, 0}, v(2) = {0.04, 0.022},
        v(3) = {0, 0.03},
    section = {
        type = mask,
        v(1) = {0.032, 0.016},
    section = {                      ! Clear area # 2
        type = clear,
        v(1) = {0, 0, 0.03, 0.013},
    section = {
        type = mask,
        v(1) = {0, 0, 0.005},
    section = {
        type = mask,
        x0 = 0.02,
        v(1) = {0, 0, 0.005} }
```

Clear area #1 has a clear section and one mask section. These sections rely on the four fold symmetry of the sections so that only points in the first quadrant need be specified. Clear area #2 has one clear section in the shape of an ellipse with two mask circles. Notice that the `mask` section of clear area #1 does not affect the clear area #2 even though it (completely) overlaps clear area #2.

Sections may overlap and a mask section does not have to be wholly within the corresponding clear section. If a photon is within multiple clear areas then, for the purposes of calculation, it is considered to be within the first possible clear area in the list.

4.10 Length Attributes

The length attributes are

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

The length `l` is the path length of the reference particle. The one exception is for an `rbend`, the length `l` set in the lattice file is the chord length (§3.5). internally, `Bmad` converts all `rbends` to `sbends` and stores the chord length under the `l_chord` attribute. Example:

```
b: rbend, l = 0.6 ! For rbends, l will be converted to l_chord
```

For a `girder` element the length `l` is a dependent attribute and is set by `Bmad` to be the difference in longitudinal position `s` of the downstream end of the last element supported relative to the upstream end of the first element.

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. See §13.1.

For `patch` elements the `l` length is, by definition, equal to `z_offset`. For `patch` elements, `l` is a dependent attribute and will be automatically set to `z_offset` by `Bmad`.

The length of a **capillary** element is a dependent variable and is given by the value of **s** of the last wall cross-section (§4.9.4).

The length of a crystal is zero for Bragg diffraction and is a dependent attribute dependent upon the crystal thickness for Laue diffraction. See §3.8 for more details.

4.11 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.

4.12 Multipole Attributes: An, Bn, KnL, Tn

Multipole formulas for are given in §14.1.

A **multipole** (§3.29) element specifies its multipole components using an Amplitude (**KnL**) and a tilt (**Tn**)

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

Here **n** ranges from 0 (dipole component) through 21. If **Tn** is given without a value, a default of $\pi/(2n + 2)$ will be used producing a skew field. Example:

```
m: multipole, k11 = 0.328, t1 ! Skew quadrupole
```

Following MAD, a non-zero dipole (**KOL** component will affect the reference orbit (just like a normal dipole will). This is not true for any other element.

An **ab_multipole** (§3.1) specifies multipoles using normal (**Bn**) and skew (**An**) components:

```
An = <Real>
Bn = <Real>
```

Here **n** ranges from 0 (dipole component) through 21. Example:

```
q1: ab_multipole, b0 = 0.12, a20 = 1e7
```

Elements like **quadrupoles** and **sextupoles** can have assigned to them multipole fields. In this case, the fields are specified using the same convention as the **ab_multipole**. For such non-multipole elements, the multipole strength is scaled by a factor $F r_0^{n_{\text{ref}}}/r_0^n$ (cf. Eq. (14.16)) 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. This behavior may be turned off by setting the **scale_multipoles** attribute. Example:

```
q1: quadrupole, b0 = 0.12, a20 = 1e7, scale_multipoles = F
```

4.13 EM Fields – Tables and Maps

Three dimensional electromagnetic fields can be specified in a number of ways. Setting a lattice element’s **field_calc** switch to **custom** (§5.4) allows for a custom calculation. The drawback here is that the appropriate custom code must be linked into any program that needs to do tracking (§29.1).

The other two possibilities are setting `field_calc` to either `grid` or `map`. `grid` defines the field using a three dimensional grid of points with interpolation used to evaluate the field between points. `map` defines the field as a sum of a number of modes (functions). As discussed below, both `grid` and `map` use the `field` attribute of an element to specify the needed parameters. Examples:

```
q: quadrupole, field_type = grid, field = {grid = {...}}, ...
r: rfcavity, field_type = map, field = {map = {...}}, ...
```

Fields so specified can be used with, for example, `runge_kutta` tracking (§5.1). Both RF and DC fields can be specified. However RF fields can only be associated with the following elements:

```
e_gun
em_field
lcavity
rfcavity
```

Currently there is the restriction that DC fields must use `grid`.

The syntax for specifying the electromagnetic fields is

```
field = {
    mode_to_autoscale = <Integer>, ! Default = 1.
    mode = {
        m           = <Integer>,      ! Mode number
        harmonic     = <Integer>,      ! Harmonic number
        phi0_ref     = <Real>,        ! Phase of oscillations.
        f_damp       = <Real>,        ! Oscillation damping factor. Default = 0.
        field_scale  = <Real>,        ! Scale factor for the E & B fields.
        master_scale = <Name>,        ! Master scaling parameter for E & B fields.
        phi0_azimuth = <Real>,        ! Azimuthal orientation.
        map          = {<EM_field_map>}, ! EM field map data
        grid         = {<EM_field_grid>} }, ! EM field grid data
    mode = {...}}
}
```

The electromagnetic field is specified as a series of `modes`. Each `mode` has a `harmonic` number which, if non-zero, identifies it as an RF field. The field associated with a mode can be specified using a `grid` of data points or by a `map` which specify the coefficients of an analytical form for the field. The `mode_to_autoscale` parameter is explained in Chapter §9

Example:

```
apex: e_gun, l = 0.23, field_calc = grid, rf_frequency = 187e6,
      field = { mode = {
          m = 0, harmonic = 1,
          master_scale = voltage,
          grid = call::apex_gun_grid.bmad }}
```

with the file `apex_gun_grid.bmad` being:

```
{
  type = rotationally_symmetric_rz,
  r0 = (0, 0),
  dr = (0.001, 0.001),
  pt(0,0) = ( (0, 0), (0, 0), (1, 0), (0, 0), (0, 0), (0, 0)),
  pt(0,1) = ( (0, 0), (0, 0), (0.99, 0), (0, 0), (0, 0), (0, 0)),
  ...
}
```

The field is scaled by two values specified by `field_scale` and `master_scale`. That is

$$[E, B](actual) = [E, B](\text{from map or grid}) * \text{field_scale} * \text{master_scale_value} \quad (4.11)$$

That is, the actual field is the value as determined from the `map` or `grid` data (see below) scaled by the value of `field_scale` times the “`master_scale_value`”. This `master_scale_value` is the value of the element parameter given by `master_scale`. For example, for a quadrupole element, if `master_scale` is set to “K1” then the fields are scaled by the quadrupole strength parameter. The purpose of this `master_scale` is to provide a way to scale the 3D fields with the element strength parameter (K1 for a quadrupole) which is separate from auto scaling. The `master_scale` also provides a way to scale separate modes jointly.

The field of a given mode oscillates as given by Eq. 14.17

$$e^{-i2\pi(f t + \theta_0)} \quad (4.12)$$

The phase of the oscillation, θ_0 comes from two sources: the phase `phi0_ref` set for the mode and an overall phase `phase_ref` given by

`phase_ref = phi0 + phi0_multipass`

Unfortunately, to be consistent with *MAD*, the definition of `phase_ref` for an `lcavity` (§3.25) and `e_gun` differ in sign to that for an `rfcavity` (§3.36). This being the case, θ_0 is

$$\theta_0 = \begin{cases} \text{phi0_ref} + \frac{f}{f_0} (\text{phase_ref} + \text{phi0_err}) & \text{lcavity \& e_gun element} \\ \text{phi0_ref} - \frac{f}{f_0} \text{phase_ref} & \text{rfcavity element} \end{cases} \quad (4.13)$$

where f is the mode frequency and f_0 is the frequency of the fundamental.

The phase `phi0_ref` of the fundamental mode can be auto-scaled by *Bmad* (§9) so that a particle with $z = 0$ will go through the cavity on crest for an `e_gun` or `lcavity` and on the zero-crossing for an `rfcavity`.

4.13.1 Map

The `map` specification has the form

```
map = {
    dz      = <Real>, ! Distance between sampled field points.
    e_coef_re = (<Real>, <Real>, ...), ! Real part of e.
    e_coef_im = (<Real>, <Real>, ...), ! Imaginary part of e.
    b_coef_re = (<Real>, <Real>, ...), ! Real part of b.
    b_coef_im = (<Real>, <Real>, ...), ! Imaginary part of b.
}
```

For rf fields the basic equations used for the mode decomposition of the rf fields are given in Section §14.2. `e_re` and `e_im` give the real and imaginary part of e and `b_re` and `b_im` give the real and imaginary part of b . All of these vectors must be present and have the same length. The exception is with an $m = 0$ mode either the e or b arrays can be omitted and will default to zero. The number of terms N for the e or b vectors must be a power of 2 and all modes must have the same number of terms. The n^{th} element in the e or b arrays, with n running from 0 to $N - 1$, is associated with a wavelength k_n

$$k_n = \begin{cases} \frac{2\pi n}{N dz} & n < \frac{N}{2} \\ \frac{2\pi(n-N)}{N dz} & \text{otherwise} \end{cases} \quad (4.14)$$

This convention follows the convention used by Numerical Recipes[Press92].

The longitudinal length of the field is

$$L_{\text{field}} = \frac{N - 1}{dz} \quad (4.15)$$

this may be different from the length l specified for the element. If there is a difference, the field is assumed to be centered on the element and drifts will be used at the entrance and exit ends of the element to make up the difference.

4.13.2 Grid

A grid of field points can be specified using the general format:

```
field = {
    grid = {
        type = <String>,
        r0   = (<Real>, <Real>, <Real>), ! Grid origin
        dr   = (<Real>, <Real>, <Real>), ! Grid spacing
        ele_anchor_pt = <Position> ! BEGINNING, CENTER, or END
        pt(<Integer>, <Integer>, <Integer>) = ( ... ), ! Grid field points
        ...
    }
}
```

The `type` switch sets the type of the grid and must come before all the other components. The possible settings of `type` are:

```
rotationally_symmetric_rz
xyz
```

The `rotationally_symmetric_rz` setting for `type` is for fields that are rotationally symmetric around the z axis. The format for this type of grid is

```
field = {grid = {
    type = rotationally_symmetric_rz,
    r0   = (<r0>, <z0>), ! Grid origin
    dr   = (<dr>, <dz>), ! Grid spacing
    pt(ir, iz) = (<E_r>, <E_phi>, <E_z>, <B_r>, <B_phi>, <B_z>)
    ...
}}
```

The `xyz` setting for `type` can be used for all fields. The format for this type of grid is

```
field = {grid = {
    type = xyz,
    r0   = (<x0>, <y0>, <z0>), ! Grid origin
    dr   = (<dx>, <dy>, <dz>), ! Grid spacing
    pt(ix, iy, iz) = (<E_x>, <E_y>, <E_z>, <B_x>, <B_y>, <B_z>)
    ...
}}
```

field components like `<E_z>` can be complex for RF fields or simply a real number for DC fields. When a field component is complex, the syntax for specifying it uses the form:

`(Re, Im)`

where `Re` and `Im` are the real and imaginary parts of the component. Example:

```
pt(0, 0, 7) = ((0.34, -4.3), (2.37, 9.34), ...) ! Complex field
pt(0, 0, 7) = (0.12, -0.33, ...) ! Real field
```

[For clarity sake, the following discusses the `xyz` case. Extension to other cases is straight forward.] There is no restriction on the bounds of the indexes (`ix`, `iy`, `iz`) of the `pt(ix, iy, iz)` array. Treating `r0` and `dr` as vectors, a point (ix, iy, iz) corresponds in space to the point

$$(x, y, z) = r0 + dr * (ix, iy, iz) + r_anchor$$

where `z` is measured from the beginning of the element and `r_anchor` is determined by the setting of `ele_anchor_pt`:

ele_anchor_pt	r_anchor
beginning	(0, 0, 0) ! Default
center	(0, 0, L/2)
end	(0, 0, L)

with L being the length of the element.

4.14 RF Couplers

For `lcavity` and `rfcavity` elements, 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/2π)
coupler_phase    = <Real>   ! Phase angle with respect to the RF (rad/2π)
```

The possible `coupler_at` settings are:

```
entrance_end
exit_end ! default
both_ends
```

The kick due to the coupler is

```
dP_x = amp * cos(phase) * cos(angle)
dP_y = amp * cos(phase) * sin(angle)
dE   = amp * (cos(angle) * x + sin(angle) * y) * sin(phase) * twopi * rf_frequency / c_light
```

where `dP_x` and `dP_y` are the transverse momentum kicks, `dE` is an energy kick, and

```
amp   = gradient * coupler_strength
phase = twopi * (phase_particle + phase_ref + coupler_phase)           ! For lcavity §3.25
       = pi/2 + twopi * (phase_particle - phase_ref + coupler_phase) ! For rfcavity §3.25
angle = twopi * coupler_angle
```

The energy kick is needed to keep things symplectic.

Example:

```
rf1: lcav, l = 4.5, gradient = 1.2e6, coupler_at = both_ends,
                           coupler_strength = 0.037
```

4.15 Wakefields

Wake fields can be specified for many elements. The attributes that characterize the wakes are:

```
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 formulas used to compute the wake field are given in §14.3. The wake field is only used when tracking bunches and not when tracking single particles.

4.15.1 Short-Range Wakes

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. An example input file is:

```
! 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
/

```

Wakes are specified via a set of “pseudo” modes (§14.3). The magnitude of `z_max` should be set to the maximum z value at which the pseudo mode fit is valid. *Bmad* uses `z_max` to check that the wake simulation is valid. That is, to be valid, the maximum distance between any two simulated particles should be less than the magnitude of `z_max`.

4.15.2 Long-Range Wakes

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 Ω/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 slash /. Everything outside this 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
          [Ohm/          Angle
          [Hz]       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
  lr(3) =     0     0.57   1.1e6   0   unpol
/

```

A frequency of zero is used to designate wakes that are part of the fundamental accelerating mode. *Bmad* needs to know if a wake is part of the fundamental mode due to timing issues as discussed in §13.7.

If the polarization angle is set to “unpolarized” the mode is taken to be unpolarized. [Note: Technically the unpolarized mode is actually two polarized normal modes. The axes of these two normal modes can be chosen arbitrary as long as they are at right angles to each other.]

`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"
```

4.16 Fringe Fields

The tracking through the fringe fields of such elements as bends, quadrupoles, etc is determined by the following element attributes

```
fringe_type      ! Used for everything with fringes
fringe_at        ! Used for all elements with fringes.
```

The **fringe_at** may be set to one of

```
no_end
both_ends          ! Default
entrance_end
exit_end
```

The **fringe_at** switch is used for vetoing fringe effects at either of the faces of the element. This is useful in vetoing the fringe effect in the interior of split elements.

The **fringe_type** switch is used to select how a fringe field is simulated. For **rbend** and **sbend** elements, **fringe_type** may be set to one of the following:

```
none
edge_focus_only
basic_bend          ! Default
full_bend
full_straight
sad_linear
sad_nonlin_only
sad_full
```

For non-bend elements, **fringe_type** may be set to one of:

```
none                ! Default
full_straight
sad_linear
sad_nonlin_only
sad_full
```

The **none** setting ignores any fringe fields and is the default for **quadrupoles**, **sextupoles**, etc. The **basic_bend** setting, which is the default for **rbend** and **sbend** elements, is essentially the basic vertical focusing effect that is present when there is a finite **e1** or **e2** face angle. With **bmad_standard** tracking, **basic_bend** also includes second order terms. The **edge_focus_only** setting ignores these second order terms. In some cases, for instance in a chicane, **basic_bend** is not good enough. With **full_bend** or **full_straight**, higher order effects are taken into account.

The difference between **full_bend** and **full_straight** is that with **full_straight** the fringe field is assumed to have the perfect symmetry of the multipole. At the bend edge, Maxwell's equations leads to next order terms with the same symmetry of the multipole. With **full_bend**, the fringe field is calculated assuming that there is translational invariance along the horizontal **x** axis. This differs from **full_straight** by adding an infinite number of multipoles consistent with the dipole symmetry. See Etienne Forest's book[Forest98] for more details.

Example:

```
b1: rbend, angle = pi/4, g = 0.3, fringe_type = full_bend
```

The **sad_linear**, **sad_nonlin_only** and **sad_full** settings of **fringe_type** emulate the fringe field tracking used in the SAD program[SAD]. The **sad_linear** setting only uses the linear part of the fringe, **sad_nonlin_only** ignores the linear part of the fringe, and **sad_full** uses the full fringe. For an **sbend** or **rbend** element, these SAD fringe fields are in addition to the fringe fields that occurs with a finite **e1** or **e2** face angle.

When using PTC tracking (§1.5), the parameter [ptc_max_fringe_order] (§8.1) determines the maximum order of the calculated fringe fields.

For programmers who deal with PTC directly: The translation between `fringe_type` on the *Bmad* side and `permfringe` and `bendfringe` on the PTC side is:

<i>fringe_type</i>	<i>permfringe</i>	<i>bendfringe</i>
none	False	False
basic_bend	False	True
full_bend	True	True
full_straight	True	False

4.17 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</i> (§21.2)	
<code>tilt</code>	θ_t	See §4.4
<code>x_offset</code>	x_{err}	See §4.4
<code>y_offset</code>	y_{err}	See §4.4
<code>x_gain_err</code>	$dg_{x,\text{err}}$	Horizontal gain error
<code>y_gain_err</code>	$dg_{y,\text{err}}$	Vertical gain error
<code>crunch</code>	ψ_{err}	Crunch angle
<code>tilt_calib</code>	θ_{err}	tilt angle calibration
<code>x_offset_calib</code>	x_{cal}	Horizontal offset calibration
<code>y_offset_calib</code>	y_{cal}	Vertical offset calibration
<code>x_gain_calib</code>	$dg_{x,\text{cal}}$	Horizontal gain calibration
<code>y_gain_calib</code>	$dg_{y,\text{cal}}$	Vertical gain calibration
<code>crunch_calib</code>	ψ_{cal}	Crunch angle calibration
<code>noise</code>	n_f	Noise factor
<code>de_eta_meas</code>	dE/E	Percent change in energy
<code>n_sample</code>	N_s	Number of sampling points
<code>osc_amplitude</code>	A_{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 §21.2 for the conversion formulas.

Chapter 5

Tracking, Spin, and Transfer Matrix Calculation Methods

Bmad allows for a number of methods that can be used to “track” a particle through a lattice element. Here “track” can mean one of three things:

- 1) Calculate a particle’s phase space coordinates at the exit end of the element given the coordinates at the entrance end.
- 2) Calculate the linear transfer map (Jacobian) through an element about a given reference orbit.
- 3) Calculate the a particle’s spin orientation at the exit end of the element given the coordinates at the beginning.

The different tracking methods that are available have different advantages and disadvantages in terms of speed, symplecticity, etc. What tracking method is used, is selected on an element-by-element basis using the attributes:

```
tracking_method      = <Switch>    ! phase space tracking method.  
mat6_calc_method    = <Switch>    ! 6x6 transfer matrix calculation.  
spin_tracking_method = <Switch>    ! Spin tracking method.
```

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.

These switches are discussed in more detail in the following sections.

5.1 Particle Tracking Methods

The `tracking_method` attribute of an element sets the algorithm that is used for single particle tracking through that element. Table 5.1 gives which methods are available for each type of element.

A note on terminology: Adaptive step size control used with the `Runge_Kutta` integrator 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.

Boris Second order Boris Integration[[Stoltz02](#)]. 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 Uses formulas for tracking. The formulas generally use the paraxial approximation. 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 as distributed evenly throughout an element.

Custom 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. See `s:custom.ele` for more details.

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. This method is not able to handle element misalignments or kicks, and becomes inaccurate as the particle energy deviates from the reference energy. MAD tracking is generally only used for testing purposes. Note: Thanks to CERN and Frank Schmidt for permission to use the MAD tracking code within *Bmad*.

runge_kutta This uses a 4th order Runge Kutta integration algorithm with adaptive step size control. This is essentially the `ODEINT` subroutine from Numerical Recipes[[Press92](#)]. This may be slow but it should be accurate. This method is non-symplectic. Warning: When using `custom` fields, if the fields do not obey Maxwell's equation, there is the possibility of the `runge_kutta` tracking halting mid way through an element. See section §[5.4](#) for more details.

Symp_Lie_Bmad 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 (§[1.5](#)), **Symp_Lie_Bmad** achieves about a factor of 10 improvement in speed over **Symp_Lie_PTC**.

Symp_Lie_PTC Symplectic tracking using a Hamiltonian with Lie operator techniques. This uses Etienne Forest's PTC (§[1.5](#)) 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 Forest's PTC software (§[1.5](#)). 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 the PTC (§[1.5](#)) 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 `taylor_map_includes_offsets` attribute (§5.6). `bmad_standard` and `taylor` tracking methods are identical. Note: Taylor maps for `match`, and `patch` elements are limited to first order.

`Time_Runge_Kutta` This method uses time as the independent variable instead of the longitudinal z position. The advantage of this method is that it can handle particles which reverse direction longitudinally. One use for this method is “dark current” tracking where low energy particles generated at the vacuum chamber walls can be found traveling in all directions. Notice that `time_runge_kutta` is different from using `absolute time tracking` as explained in §13.7.

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

^aSee §3.43.2 for more detailsTable 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 Linear Transfer Map Methods

The `mat6_calc_method` attribute sets how the 6x6 Jacobian transfer matrix for a given element is computed. Table 5.2 gives which methods are available for each type of element.

In addition to the `mat6_calc_method` switch, two element attributes that can affect the way the transfer matrix is calculated are `symplectify` and `taylor_map_includes_offsets`. These are discussed in sections §5.5 and §5.6 respectively.

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 §18.2.

Symplectic integration is like ordinary integration of a function $f(x)$ but what is integrated here is a Taylor map. Truncating the map to 0th order gives the particle trajectory and truncating to 1st order gives the transfer matrix (Jacobian). The order at which a Taylor series is truncated at is set by `taylor_order` (see §8.1). Like ordinary integration there are various formulas that one can use to do symplectic integration. In *Bmad* (or more precisely in PTC (§1.5)) 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 formulas generally use the paraxial approximation. 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. See `s:custom.ele` for more details.

MAD This uses the MAD 2nd transfer map. This method is not able to handle element misalignments or kicks, and becomes inaccurate as the particle energy deviates from the reference energy. MAD tracking is generally only used for testing purposes. Thanks must be given to CERN and Frank Schmidt for permission to use the MAD tracking code within *Bmad*.

Static This prevents the transfer matrix from being recomputed. Using **Static** in the input file is generally not a good idea since it prevents the matrix from being computed in the first place. Typically **Static** 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, **Symp_Lie_Bmad** achieves about a factor of 10 improvement in speed over **Symp_Lie_PTC**. However, **Symp_Lie_Bmad** cannot generate maps above first order.

Symp_Lie_PTC Symplectic integration using a Hamiltonian and Lie operators. This uses the PTC (§1.5) 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 Forest's PTC software (§1.5). 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 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 `taylor_map_includes_offsets` attribute (§5.6). `bmad_standard` and `taylor` tracking methods are identical. Note: Taylor maps for `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	Static	Symp_Lie_Bmad	Symp_Lie_PTC	Taylor	Tracking
ab_multipole	D	X		X		X	X	X
beambeam	D	X		X				X
bend_sol_quad		X		X	D			X
capillary	D	X		X				X
crystal	D	X		X				X
custom		D		X				X
drift	D	X	X	X		X	X	X
e_gun		X		X				D
ecollimator	D	X		X		X	X	X
elseparator	D	X	X	X		X	X	X
em_field		X		X				D
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	X	X
marker	D	X		X		X	X	X
match	D	X		X				X
mirror	D	X		X				X
monitor	D	X		X		X	X	X
multipole	D	X		X		X	X	X
multilayer_mirror	D	X		X				X
octupole	D	X		X		X	X	X
patch	D	X		X		X	X	X
quadrupole	D	X	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
sad_mult	D	X		X				X
sample	D	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	X
sol_quad	D	X	X	X	X	X	X	X
taylor	X	X		X				D
vkicker	D	X		X		X	X	X
wiggler	D	X		X	X ^a	X ^a	X ^a	X

^aSee §3.43.2 for more detailsTable 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 Spin Tracking Methods

The `spin_tracking_method` attribute of an elements sets the algorithm that is used for tracking a particle's spin through that element. Table 5.3 gives which methods are available for each type of element.

Since speed may be an issue, *Bmad* has an internal global parameter called `spin_tracking_on` (§10.1) that determines whether spin is tracked or not. Access to this parameter is implemented on a program-by-program basis.

`Bmad_Standard` Uses formulas for the calculation. The formulas generally use the paraxial approximation. The emphasis here is on speed.

`Custom` This method will call a routine `track1_spin_custom` which must be supplied by the programmer implementing the custom spin tracking calculation. See `s:custom.ele` for more details.

`Tracking` If `tracking_method` is set to `boris`, or `runge_kutta` then the spin will be tracked along with the phase space particle coordinates using the local fields. For all other `tracking_methods`, the spin will be tracked using the `bmad_standard` spin tracking method.

`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.

Example:

```
q: quadrupole, spin_tracking_method = symp_lie_ptc
```

	Bmad_Standard	Custom	Symp_Lie_PTC	Tracking
ab_multipole	D X			
beambeam		X		
bend_sol_quad	D X X	X	X	
capillary				
crystal				
custom		D X	X	
drift		D X X	X	
e_gun		X		D
ecollimator	D X X	X	X	
elseparator	D X X	X	X	
em_field		X		D
hkicker	D X X	X	X	
instrument	D X X	X	X	
kicker	D X X	X	X	
lcavity	D X X	X	X	
marker	D X X	X	X	
match		X		
mirror				
monitor	D X X	X	X	
multipole	D X X			
multilayer				
octupole	D X X	X	X	
patch		X X		
quadrupole	D X X	X	X	
rbend	D X X	X	X	
rcollimator	D X X	X	X	
rfcavity	D X X	X	X	
sad_mult		X		
sample				
sbend	D X X	X	X	
sextupole	D X X	X	X	
solenoid	D X X	X	X	
sol_quad	D X X	X	X	
taylor				
vkicker	D X X	X	X	
wiggler	D X X	X	X	

Table 5.3: Table of available `spin_tracking_method` switches for a given element class. “D” denotes the default method. “X” denotes an available method.

5.4 Integration Methods

“Integration methods” are tracking methods that involve integrating through an element’s magnetic and electric fields. 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      ! Uses PTC
taylor            ! Uses PTC
```

See section §18.1 for more information on Taylor maps and symplectic integration. The latter two methods involve using the PTC library (§1.5).

The methods that do not involve Taylor maps are

```
boris
runge_kutta
time_runge_kutta
```

there are a number of element 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
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^{th} 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`.

The `runge_kutta` and `time_runge_kutta` tracking uses adaptive step control independent of `ds_step`. These methods use two parameters from the `Bmad` global parameters structure (§10.1) namely:

```
bmad_com%rel_tol_adaptive_tracking
bmad_com%abs_to_adaptive_tracking
```

The estimated error of the integration is then bounded by

```
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`, 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
grid
map
```

`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 (§29.1). Elements that set `field_calc` to `grid` or `map` need the appropriate `field` attribute (§4.13).

Warning: When tracking a particle through a custom field using `runge_kutta`, it is important that the field obey Maxwell's equations. Fields that do not obey Maxwell's Equations may cause the `runge_kutta` adaptive step size control algorithm to take smaller and smaller steps until the step size becomes so small the tracking will stop. What happens is that the step size control algorithm takes a step and then takes two half steps over the same region and from this estimates the error in the calculation. If the error is larger than the allowed tolerance the control algorithm shortens the step and tries again. A field that does not obey Maxwell's equations can fool the control algorithm into thinking that the error is always larger than the allowed tolerance for any finite step size. A typical situation is where the field has an unphysical step across some boundary.

The phase space coordinates used with `boris` tracking are not the standard *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 $cP_0 = vE_0/C = E_0$ so that $dE/cP_0 = dE/E$ giving the standard *Bmad* coordinates.

When tracking uses the PTC library (§1.5), there are two global parameters that can be set in the lattice file that affect the calculation. These are:

```
parameter[ptc_exact_model]      = <Logical> ! "exact" tracking? Default: False
parameter[ptc_exact_misalignment] = <Logical> ! "exactly" misalign elements?
                                                ! Default: True
```

The default for `ptc_exact_model` is `False` and the default for `ptc_exact_misalignment` is `True`.

The `ptc_exact_model` parameter sets whether PTC uses an “exact” model for tracking. This can be important, for example, for bend tracking when the bend radius is small.

The `ptc_exact_misalignment` parameter determines whether misalignments are handled exactly or whether approximations are made that will speed up the calculation.

In addition to the above parameters, how the Hamiltonian is split when tracking with PTC can be set for individual elements using the `ptc_integration_type` parameter. Possible settings of this parameter are

```
drift_kick    ! See Eq.(125) of [Forest06]
matrix_kick   ! See Eq.(132) of [Forest06]. Default
ripken_kick   ! See Eq.(130) of [Forest06]
```

Example:

```
q2: quad, l = 0.6, k1 = 0.34, ptc_integration_type = drift_kick
```

A discussion of the different types of integration schemes is given by Forest[Forest06]. The equation that shows the appropriate splitting of the Hamiltonian for each integration type is referenced in the above list. The `ripken_kick` type is for benchmarking with the `SixTrack` program and is not otherwise generally useful.

5.5 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 §18.2 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
```

Note that for elements like an `lcavity` where the reference momentum at the downstream end of the element is different from the upstream end, the transfer matrix is never symplectic. In this case, “symplectification” involves first transforming the transfer matrix so that the reference momentum is the same upstream and downstream, then performing symplectification, and finally back transforming the reference momentum to their original values.

5.6 Map_with_offsets Attribute

The `taylor_map_includes_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 `taylor_map_includes_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 `taylor_map_includes_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 `taylor_map_includes_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 `taylor_map_includes_offsets` to `True` and if the orientation is varying it is faster to set `taylor_map_includes_offsets` to `False`.

If the global parameter `bmad_com%conserve_taylor_maps` (§10.1) 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 and Replacement Lists

This chapter describes how to define the ordered list of elements that make up a lattice branch (§1.2). In a lattice, branches may be connected together using `fork` or `photon fork` elements (`s:fork`), or by using `multipass` (§7.2).

6.1 Branch Construction Overview

A lattice branch is defined in a lattice file using what are called `beam lines` (§6.2) and `replacement lists` (§6.5). The `beam lines` are divided into two types - lines with (§6.4) and lines without (§6.2) `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.

Since lines can be nested within other lines, The same element name may be repeated multiple times in a branch. To distinguish between multiple elements of the same name, lines and lists may be `tagged` (§6.7) to produce unique element names.

There will also be a marker element named `END` automatically placed at the end of the lattice. This end marker will not be automatically placed in the lattice if a marker named `end` is defined in the lattice file at the end of the lattice. Additionally, a `parameter[no_end_marker]` statement (§8.1) can be used to suppress the insertion of the end marker.

6.2 Beam Lines and Lattice Expansion

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
```

The `use` statement is explained in Section §6.6. This example shows how a `beam line` member can refer to another `beam line`. This is helpful if the same sequence of elements appears repeatedly in the lattice.

The process of constructing the ordered sequences of elements that comprise the branches of the lattice is called **lattice expansion**. In the example above, when `line2` is expanded to form the lattice (in this case there is only one branch so `lattice` and `branch` can be considered synonymous), the definition of `line1` will be inserted in to produce the following lattice:

```
beginning, d, a, b, c, e, end
```

The `beginning` and `end` marker elements are automatically inserted at the beginning and end of the lattice. The `beginning` element will always exist but insertion of the `end` element can be suppressed by inserting into the lattice:

```
parameter [no_end_marker] = T      ! See: §8.1
```

Lattice expansion occurs at the end when a lattice file has been parsed or if an `expand_lattice` statement (§2.19) is present.

Each element is assigned an `element index` number starting from 0 for the `beginning` element, 1 for the next element, etc.

In the expanded lattice, any `null_Ele` type elements (§3.31) will be discarded. For example, if element `b` in the above example is a `null_Ele` then the actual expanded lattice will be:

```
beginning, d, a, c, e, end
```

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)
```

Lines can be defined in any order in the lattice file so a subline does not have to come before a line that references it. Additionally, element definitions can come before or after any lines that reference them.

A line can have the `multipass` attribute. This is covered in §7.2.

6.3 Element Reversal

An element is **reversed** if particles traveling through it enter at the “exit” end and leave at the “entrance” end. Being able to reverse elements is useful, for example, in describing the interaction region of a pair of rings where particles of one ring are going in the opposite direction relative to the particles in the other ring.

Element reversal is indicated by using a double negative sign “`--`” prefix. The double negative sign prefix can be applied to individual elements or to a line. If it is applied to a line, the line is both reflected (same as if a single negative sign is used) and each element is reflected. For example:

```
line1: line = (a, b, --c)
line2: line = (--line1)
line3: line = (c, --b, --a)
```

In this example, `line2` and `line3` are identical. Notice that the reversal of a reversed element makes the element unreversed.

Reversed elements, unlike other elements, have their local *z*-axis pointing in the opposite direction to the local *s*-axis (§13.2). This means that there must be a **reflection patch** (§13.3.5) between reversed and unreversed elements. See §11.3 for an example. Since this complicates matters, it is generally only useful to employ element reversal in cases where there are multiple intersecting lines with particle beams going in opposite directions through some elements. Where reversed elements are not needed, it is simple to define elements that are effectively reversed. For example:

```
b00: bend, angle = 0.023, e1 = ...
b00_rev: b00, angle = -b00[angle], e1 = -b00[e2], e2 = -b00[e1]
```

and `b00_rev` serves as a reversed version of `b00`.

6.4 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.
```

6.5 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.
```

6.6 Use Statement

The particular line or lines that defines the root branches (§1.3) to be used in the lattice is selected by the `use` statement. The general syntax is

```
use, line1, line2 ...
```

For example, `line1` may correspond to one ring and `line2` may correspond to the other ring of a dual ring colliding beam machine. In this case, `multipass` (§7.2) will be needed to describe the common elements of the two rings. Example

```
use, e_ring, p_ring
```

would pick the lines `e_ring` and `p_ring` for analysis. These will be the `root` branches.

`use` statements can come anywhere in the lattice, even before the definition of the lines they refer to. Additionally, there can be multiple `use` statements. The last `use` statement in the file defines which `line` to use.

6.7 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:

<code>line_name[tag_name]</code>	! Syntax for lines
<code>list_name[tag_name]</code>	! Syntax for lists
<code>replacement_line[tag_name](arg1, arg2, ...)</code>	! 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 ([§2.19](#)). 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.

Chapter 7

Superposition, and Multipass

This chapter covers two concepts: **superposition** (§7.1) and **multipass** (§7.2). Superposition is used when elements overlap spatially. Multipass is used when an element is “shared” between branches such as the interaction region shared by two storage rings, or when a beam goes through the same physical element in a branch multiple times as in an energy recovery linac.

In both cases, **lord** and **slave** elements (§1.4) are constructed by *Bmad* to hold the necessary information. In both cases, the **lord** elements will represent the “physical” element while the **slave** elements will embody the “beam path”.

7.1 Superposition

In practice the field at a particular point in the lattice may be due to more than one physical element. One example of this is a quadrupole magnet inside a larger solenoid magnet as shown in Fig. 7.1A. *Bmad* has a mechanism to handle this using what is called “superposition”. A simple example shows how this works (also see section §1.4):

Q: quad, l = 4

D: drift, l = 12

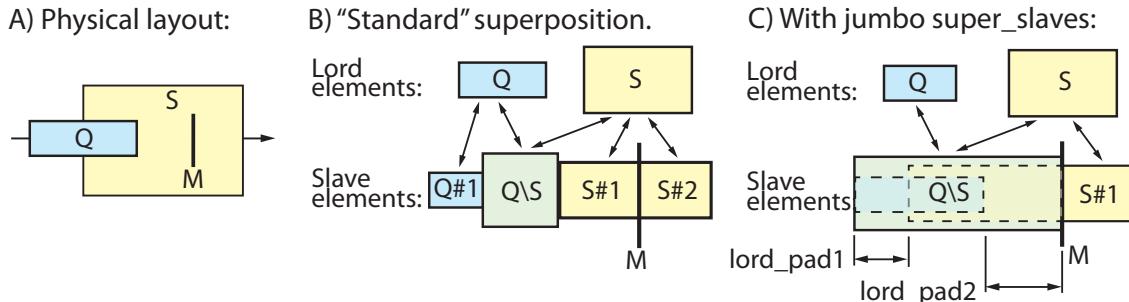


Figure 7.1: Superposition example. A) The physical layout involves a quadrupole partially inside a solenoid. B) The standard superposition procedure involves creating **super_slave** elements whose edges are at the boundaries where the physical elements overlap. C) When jumbo **super_slaves** are created, the **super_slaves** span the entire space where elements overlap.

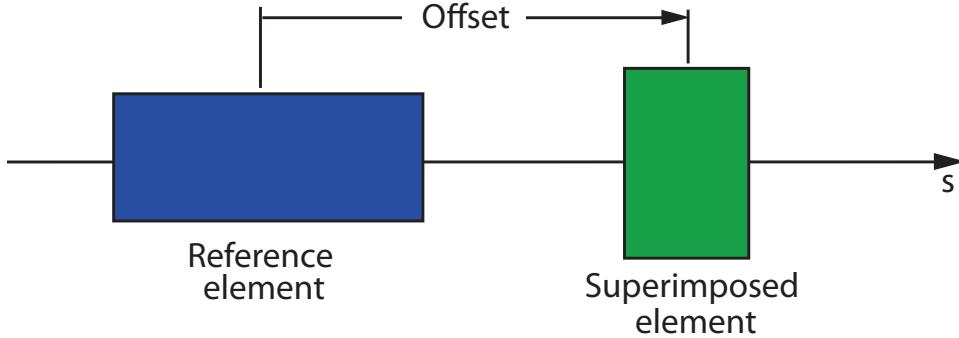


Figure 7.2: The superposition offset is the distance from the origin point of the reference element to the origin point of the element being superimposed.

```
S: solenoid, l = 8, superimpose, ref = Q, ele_origin = beginning
M: marker, superimpose, ref = S, offset = 1
lat: line = (Q, D)
use, lat
```

The `superimpose` attribute of element `S` superimposes `S` over the lattice `(Q, D)`. The placement of `S` is such that the beginning of `S` is coincident with the center of `Q` (this is explained in more detail below). Additionally, a marker `M` is superimposed at a distance of +1 meter from the center of `S`. The tracking part of the lattice (§1.4) looks like:

	Element	Key	Length	Total
1)	Q#1	Quadrupole	2	2
2)	Q\S	Sol_quad	2	4
3)	S#1	Solenoid	3	7
4)	M	Marker	0	
4)	S#2	Solenoid	3	10
5)	D#2	Drift	4	14

What `Bmad` has done is to split the original elements `(Q, D)` at the edges of `S` and then `S` was split where `M` is inserted. The first element in the lattice, `Q#1`, is the part of `Q` that is outside of `S`. Since this is only part of `Q`, `Bmad` has put a `#1` in the name so that there will be no confusion. (`#` 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 would expect. `S#1` is the part of `S` that is outside `Q` but before `M`. This element is just a solenoid. Next comes `M`, `S#1`, and finally `D#2` is the rest of the drift outside `S`.

In the above example, `Q` and `S` will be `super_lord` elements (`s:lord.slave`) and four elements in the tracking part of the lattice will be `super_slave` elements. This is illustrated in Fig. 7.1B.

Notice that the name chosen for the `sol_quad` element `Q\S` is dependent upon what is being superimposed upon what. If `Q` had been superimposed upon `S` then the name would have been `S\Q`.

When `Bmad` sets the element class for elements created from superpositions, `Bmad` will set the class of the element to something other than an `em_field` element (§3.15) if possible. If no other possibilities exist, `Bmad` will use `em_field`. For example, a `quadrupole` superimposed with a `solenoid` will produce a `sol_quad` element but a `solenoid` superimposed with a `rfcavity` element will produce an `em_field` element since there is no other class of element that can simultaneously handle solenoid and RF fields.

With the lattice broken up like this `Bmad` has constructed something that can be easily analyzed. However, the original elements `Q` and `S` still exist within the lord section of the lattice. `Bmad` has

bookkeeping routines so that if a change is made to the **Q** or **S** elements then these changes can get propagated to the corresponding slaves. It does not matter which element is superimposed. Thus, in the above example, **S** could have been put in the Beam Line (with a drift before it) and **Q** could then have been superimposed on top and the result would have been the same (except that the split elements could have different names).

If an element that naturally has zero length (for example, a **marker** element), is superimposed, or is superimposed upon, then the element will remain in the tracking part of the lattice and there will be no corresponding lord element. See Fig. 7.1.

The placement of a superimposed element is illustrated in Fig. 7.2. The placement of a superimposed element is determined by three factors: An origin point on the superimposed element, an origin point on the reference element, and an offset between the points. The attributes that determine these three quantities are:

```
create_jumbo_slave = <Logical>      ! See §7.1.1
ref            = <lattice_element>
offset          = <length>           ! default = 0
ele_origin     = <origin_location> ! Origin pt on element.
ref_origin     = <origin_location> ! Origin pt on ref element.
```

ref sets the reference element. If **ref** is not present then the start of the lattice is used. The location of the origin points are determined by the setting of **ele_origin** and **ref_origin**. The possible settings for these parameters are

```
beginning      ! Beginning (upstream) edge of element
center         ! Center of element. Default.
end            ! End (downstream) edge of element
```

center is the default setting. **offset** is the longitudinal offset between the origin points. The default offset is zero.

Note: There is an old syntax, deprecated but still supported for now, where the origin points were specified by the appearance of:

```
ele_beginning    ! Old syntax. Do not use.
ele_center       ! Old syntax. Do not use.
ele_end          ! Old syntax. Do not use.
ref_beginning    ! Old syntax. Do not use.
ref_center        ! Old syntax. Do not use.
ref_end          ! Old syntax. Do not use.
```

For example, “**ele_origin = beginning**” in the old syntax would be “**ele_beginning**”.

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 so part of the element will be at the beginning of the lattice and part of the element will be at the end. For consistency’s sake, this is done even if the **geometry** is set to **open** (for example, it is sometimes convenient to treat a circular lattice as linear). Example:

```
d: drift, l = 10
q: quad, l = 2, superimpose
machine: line = (d)
use, machine
```

The lattice will have three elements in the tracking section:

	Element	Key	Length
3)	Q#2	Quadrupole	1
2)	D#1	Drift	8
1)	Q#1	Quadrupole	1

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

When a superposition is made that overlaps a drift the drift, not being a "real" element, vanishes. That is, it does not get put in the lord section of the lattice. Note that if aperture limits (§4.6) have been assigned to a drift, the aperture limits can "disappear" when the superposition is done. Explicitly, if the exit end of a drift has been assigned aperture limits, the limits will disappear if the superimposed element overlays the exit end of the drift. A similar situation applies to the entrance end of a drift. If this is not desired, use a `pipe` element instead.

When the attributes of a `super_slave` are computed from the attributes of its `super_lords`, some types of attributes may be "missing". For example, it is, in general, not possible to set appropriate aperture attributes (§4.6) of a `super_slave` if the lords of the slave have differing aperture settings. When doing calculations, *Bmad* will use directly the corresponding attributes stored in the lord elements to correctly calculate things.

7.1.1 Jumbo `super_slave`s

The problem with the way `super_slave` elements are created as discussed above is that edge effects will not be dealt with properly when elements with non-zero fields are misaligned. When this is important, especially at low energy, a possible remedy is to instruct *Bmad* to construct "jumbo" `super_slave` elements. The general idea is to create one large `super_slave` for any set of overlapping elements. Returning to the superposition example at the start of Section §7.1, If the superposition of solenoid S is modified to be

```
S: solenoid, l = 8, superimpose, ref = Q, ele_origin = beginning,
    create_jumbo_slave = T
```

The result is shown in Fig. 7.1C. The tracking part of the lattice will be

	Element	Key	Length	Total
1)	Q\S	Sol_quad	2	4
2)	M	Marker	0	
3)	S#2	Solenoid	3	10
4)	D#2	Drift	4	14

Q and part of S have been combined into a jumbo `super_slave` named Q\S. Since the `super_lord` elements of a jumbo `super_slave` may not completely span the slave two attributes of each lord will be set to show the position of the lord within the slave. These two attributes are

```
lord_pad1 ! offset at upstream end
lord_pad2 ! offset at downstream end
```

`lord_pad1` is the distance between the upstream edge of the jumbo `super_slave` and a `super_lord`. `lord_pad2` is the distance between the downstream edge of a `super_lord` and the downstream edge of the jumbo `super_slave`. With the present example, the lords have the following padding:

	lord_pad1	lord_pad2
Q	0	3
S	2	0

One major drawback of jumbo `super_slave` elements is that the `tracking_method` (§5.1) will, by necessity, be set by *Bmad* to `runge_kutta` and the `mat6_calc_method` (§5.2) will be set to `tracking`.

Notice that the problem with edge effects for non-jumbo `super_slave` elements only occurs when elements with nonzero fields are superimposed on top of one another. Thus, for example, one does not need to use jumbo elements when superimposing a `marker` element.

7.1.2 Changing Element Lengths when there is Superposition

When a program is running, if `group` (§3.20) or `overlay` (§3.33) elements are used to vary the length of elements that are involved in superimposition, the results are different from what would have resulted if instead the lengths of the elements were changed in the lattice file. There are two reasons for this. First, once the lattice file has been parsed, lattices can be “mangled” by adding or removing elements in a myriad of ways. This means that it is not possible to devise a general algorithm for adjusting superimposed element lengths that mirrors what the effect of changing the lengths in the lattice file.

Second, even if a lattice has not been mangled, an algorithm for varying lengths that is based on the superimpose information in the lattice file could lead to unexpected results. To see this consider the first example in Section §7.1. If the length of `S` is varied in the lattice file, the upstream edge of `S` will remain fixed at the center of `Q` which means that the length of the `super_slave` element `Q#1` will be invariant. On the other hand, if element `S` is defined by

```
S: solenoid, l = 8, superimpose, offset = 6
```

This new definition of `S` produces produce exactly the same lattice as before. However, now varying the length of `S` will result in the center of `S` remaining fixed and the length of `Q#1` will not be invariant with changes of the length of `S`. This variation in behavior could be very confusing since, while running a program, one could not tell by inspection of the element positions what should happen if a length were changed.

To avoid confusion, *Bmad* uses a simple algorithm for varying the lengths of elements involved in superposition: The rule is that the length of the most downstream `super_slave` is varied. With the first example in Section §7.1, the `group G` varying the length of `Q` defined by:

```
G: group = Q, l
```

would vary the length of `Q\S` which would result in an equal variation of the length of `S`. To keep the length of `S` invariant while varying `Q` the individual `super_slave` lengths can be varied. Example:

```
G2: group = Q#1, S#1:-1, l
```

Another, in this case cleaner, way of doing this is just to vary the downstream edge of `Q`:

```
G3: group = Q, end_edge
```

7.2 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.

```
RF1: lcavity
linac_part: line[multipass] = (RF1, ...)
my_line: line = (linac_part, ..., linac_part)
use, my_line
expand_lattice
RF1\2[phi0_multipass] = 0.5
```

The tracking part of the lattice consists of two slave elements

```
RF1\1, ..., RF1\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 `RF1` in the lord part of the lattice (§1.4). Changes to attributes of the lord `RF1` element will be passed to the slave elements by *Bmad*’s bookkeeping routines. Assuming `RF1\1` is an accelerating cavity, to make `RF1\2` a decelerating

cavity the `phi0_multipass` attribute of `RF1\2` is set to 0.5. This is the one attribute that *Bmad*'s bookkeeping routines will not touch when transferring attribute values from `RF1` to its slaves. Notice that the `phi0_multipass` attribute had to be set after `expand_lattice` (§2.19) is used to expand the lattice since *Bmad* does immediate evaluation and `RF1\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, ...)
```

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 1st and 4th elements in the tracking part of the lattice and the second `A` lord controls the 2nd and 5th elements.

7.2.1 The Reference Energy 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. The default is for `n_ref_pass` to be set to 1.

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 lord element. The default if nothing is set, is for `n_ref_pass` to be set to 1. Note: If `ref_orbit` is set to `match_global_coords`, or for any element where the reference energy is not constant (like an `lcavity`), `n_ref_pass` must be used and must be set to 1.

Chapter 8

Lattice Parameter Statements

This chapter deals with statements that can be used to set “global” parameter values. That is, parameter values that are associated with the lattice as a whole and not simply associated with a single element.

8.1 Parameter Statement

The `parameter` statement is used to set the `lattice` name and other variables. If multiple branches are present (§1.2), these variables pertain to the `root` branch. The variables that can be set by `parameter` are

```
parameter[absolute_time_tracking] = <Logical> ! Absolute time used for RF clock?
parameter[aperture_limit_on]      = <Logical> ! Use aperture limits in tracking.
parameter[auto_scale_field_phase] = <Logical> ! Automatic phase scaling.
parameter[auto_scale_field_amp]   = <Logical> ! Automatic amplitude scaling.
parameter[custom_attributeN]     = <string> ! Defining custom attribues (§2.10).
parameter[e_tot]                 = <Real> ! Reference total Energy.
                                         ! Default: 1000 * rest_energy.
parameter[geometry]              = <Switch> ! Open or closed
parameter[lattice]                = <String> ! Lattice name
parameter[n_part]                = <Real> ! Number of particles in a bunch.
parameter[no_end_marker]         = <Logical> ! Default: False.
parameter[particle]               = <Switch> ! Reference species: positron, proton, etc.
parameter[p0c]                   = <Real> ! Reference momentum.
parameter[photon_type]           = <Switch> ! Incoherent or coherent photons?
parameter[ptc_exact_model]       = <Logical> ! PTC to do "exact" tracking?
parameter[ptc_exact_misalignment] = <Logical> ! PTC to "exactly" misalign elements?
parameter[ptc_max_fringe_order]   = <Integer> ! Max fringe order.
                                         ! Default: 2 => Quadrupole.
parameter[rel_tracking_charge]    = <Real> ! Charge of tracked particles relative
                                         ! to reference charge. Default is 1.
parameter[ran_seed]              = <Integer> ! Random number generator init.
parameter[taylor_order]          = <Integer> ! Default: 3
parameter[use_ptc_layout]         = <Logical> ! Construct layout for calculations?
parameter[use_hard_edge_drifts]   = <Logical>
```

Examples

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

The `parameter[no_end_marker]` is used to suppress the automatic inclusion of a marker named `END` at the end of the lattice (§6.1).

For more information on `parameter[ran_seed]` see §2.13.

For more information on defining custom attributes, see §2.10.

Valid `parameter[particle]` switches are:

```
electron,      positron,
muon,         antimuon,
proton,        antiproton,
photon,        pion_0,
pion_minus,   pion_plus
```

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 particles in a bunch. It is used with `BeamBeam` elements and is used to calculate the change in energy through an `LCavity`. See §3.25 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 `photon_type` switch is used to set the type of photons that are used in tracking. Possible settings are:

```
incoherent ! Default
coherent
```

The general rule is use incoherent tracking except when there is a `diffraction_plate` element in the lattice.

The `lattice` name is stored by `Bmad` for use by a program but it does not otherwise effect any `Bmad` routines.

Valid `geometry` settings are

```
closed ! Default w/o LCavity element present.
open   ! Default if LCavity elements present.
```

A machine with a `closed` geometry is something like a storage ring where the particle beam recirculates through the machine. A machine with an `open` geometry is something like a linac. In this case, the initial Twiss parameters need to be specified in the lattice file. If the `geometry` is not specified, `closed` is the default. The exception is that if there is an `LCavity` element present or the reference particle is a photon, `open` will be the default.

Note: `geometry` used to be called `lattice_type`, `closed` used to be called `circular_lattice` and `open` used to be called `linear_lattice`. This old syntax is still accepted in lattice files.

The `rel_tracking_charge` establishes the charge of particles being tracked relative to the charge of the reference particle. The default is 1.

The Taylor order (§18.1) 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.

The `auto_scale_field_phase` and `auto_scale_field_amp` switches toggle the automatic scaling of the RF phase and/or amplitude for `rfcavity`, `lcavity`, and `em_field` elements (§9).

The `absolute_time_tracking` switch sets whether the clock for the `lcavity` and `rfcavity` elements is tied to the reference particle or to uses the absolute time (§13.7). A value of `False` (the default) mandates relative time and a value of `True` mandates absolute time.

The `ptc_exact_model` and `ptc_exact_misalign` switches affect tracking using the PTC library. See §5.4 for more details.

When using PTC tracking (§1.5), the parameter [`ptc_max_fringe_order`] determines the maximum order of the calculated fringe fields. The default is 2 which means that fringe fields due to a quadrupolar field. These fields are 3rd order in the transverse coordinates.

8.2 Beam_start Statement

The `beam_start` statement is used to set the starting coordinates for particle tracking. If multiple branches are present (§1.2), these variables pertain to the root branch.

```
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>    ! Momentum deviation. Only used for non-photons
beam_start[E_photon]   = <Real>    ! Energy (eV). Only used for photons.
beam_start[emittance_a] = <Real>    ! A-mode emittance
beam_start[emittance_b] = <Real>    ! B-mode emittance
beam_start[emittance_z] = <Real>    ! Z-mode emittance
beam_start[field_x]    = <Real>    ! Photon beam field along x-axis
beam_start[field_y]    = <Real>    ! Photon beam field along y-axis
beam_start[phase_x]    = <Real>    ! Photon beam phase along x-axis
beam_start[phase_y]    = <Real>    ! Photon beam phase along y-axis
beam_start[t]           = <Real>    ! Absolute time
```

Normally the absolute time, set by `beam_start[t]`, is a dependent parameter set by solving Eq. (13.26) for t . The exception is when the initial velocity is zero. (This can happen if there is an `e_gun` (§3.13) element in the lattice). In this case, z must be zero and t is an independent parameter that can be set.

For photons, `px`, `py`, and `pz` are the normalized velocity components (Cf. Eq. (13.34)). For photons `pz` is a dependent parameter which will be set so that Eq. (13.35) is obeyed.

Example

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

8.3 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]`.

8.4 Beginning and Line Parameter Statements

For non-circular lattices, the `beginning` statement can be used to set the Twiss parameters and beam energy at the beginning of the first lattice branch.

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

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 of the first lattice branch (see §13.3). 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
```

If the floor position is not specified, the default is to place beginning element at the origin with all angles set to zero.

The `beginning` statement is useful in situations where only parameters for the first branch need be specified. If this is not the case, the parameters for any branch can be specified using a statement of the form

```
line_name[parameter] = <Value>
```

This construct is called a `line parameter` statement. Here `line_name` is the name of a line and `parameter` is the name of a parameter. The parameters that can be set here are identical to the parameters that can be set with the `beginning` statement. Example:

```
x_ray_fork: fork, to_line = x_ray
x_ray = (...)
```

```
x_ray[E_tot] = 100
```

Rules:

1. The floor position of a line can only be set if the line is used for a **root branch**.
2. Line parameters statements must come after the associated line. This rule is similar to the rule that element attribute redefinitions must come after the definition of the element.

Chapter 9

Automatic Scaling of Accelerating Fields

The elements that can have accelerating fields are:

```
e_gun      ! §3.13
em_field   ! §3.15
lcavity    ! §3.25
rfcavity   ! §3.36
```

The problem that arises with accelerating fields is how to set the overall amplitude (and phase if the fields are oscillating) of the field so that the reference particle has the desired acceleration. The problem becomes even more complicated at non-ultra relativistic energies where the velocity is not a constant. In this case, the proper amplitude and/or phase settings will depend upon what the incoming energy of the reference particle is.

The scaling problem is not present when `bmad_standard` tracking (§5.1) is used since `bmad_standard` tracking uses an integrated formula that is designed to give the proper acceleration.

To help with the scaling problem, *Bmad* has the capability to automatically scale an accelerating field's amplitude and/or phase. The two parameters that turn on/off auto scaling are (§8.1):

```
parameter[auto_scale_field_phase] = <Logical> ! Automatic phase scaling.
parameter[auto_scale_field_amp]   = <Logical> ! Automatic amplitude scaling.
```

The default value is True for both parameters.

Scaling takes place during program execution when a lattice is initially created (that is, when the lattice file is parsed) and when parameters in the lattice that would change the scaling are varied.

The parameters that are varied when the field is auto scaled depend upon how the field is calculated. When the fields are specified using modes (§4.13), each mode has a `field_scale` that varies the field amplitude and each mode has a phase offset `phi0_ref`. Not all modes are involved with auto scaling. For example, an `em_field` element may have an accelerating mode along with a mode for a solenoid field which is independently powered. In this instance, only the accelerating mode should be auto scaled. The “primary” mode that is auto scaled is set by the `mode_to_auto scale` component of the `field` (§4.13). Additionally, any other mode (called “secondary” modes) whose `master_scale` is the same as the `master_scale` of the primary auto scaled mode will be auto scaled. All modes involved in auto scaling have their `field_scale` adjusted by the same ratio. Additionally, when the phase is auto scaled, all secondary modes will have their `phi0_ref` phases adjusted by the same difference that the `phi0_ref` of the primary mode is changed. Example:

```
rf2334: lcavity, field_type = grid, field =
    mode_to_autoscale = 1,
    mode = master_scale = gradient, ..., ! Primary mode
    mode = master_scale = gradient, ..., ! Secondary mode
    mode = master_scale = none, ...      ! Not involved in auto scale
```

When the fields are *not* specified using modes, the element's **field_factor** parameter will be used for scaling the amplitude and the **phi0_ref** parameter (which is distinct from the **phi0_ref** associated with a mode) will be used for varying the phase.

Chapter 10

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** (§10.2) 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 §10.1 the syntax is

```
bmad_com%max_aperture_limit
```

10.1 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 along with the default values 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) default_ds_step     = 0.2          ! Integration step size.
  real(rp) significant_length = 1e-10         ! meter
  real(rp) rel_tol_tracking   = 1e-6          ! Runge-Kutta: Relative tolerance.
  real(rp) abs_tol_tracking   = 1e-8          ! Runge-Kutta: Absolute tolerance.
  real(rp) rel_tol_adaptive_tracking = 1e-8    ! Tracking relative tolerance.
  real(rp) abs_tol_adaptive_tracking = 1e-10   ! Tracking absolute tolerance.
  real(rp) init_ds_adaptive_tracking = 1e-3    ! Initial step size.
  real(rp) min_ds_adaptive_tracking = 0        ! Minimum step size to use.
  real(rp) fatal_ds_adaptive_tracking = 1e-8    ! Threshold for loosing particles.
  integer taylor_order = 3                  ! 3rd order is default
  integer default_integ_order = 2            ! PTC integration order
```

```

integer ptc_max_fringe_order = 2          ! PTC max fringe order (2 => Quadrupole !).
logical sr_wakes_on = T                 ! Short range wake fields?
logical lr_wakes_on = T                 ! Long range wake fields
logical mat6_track_symmetric = T        ! symmetric offsets
logical auto_bookkeeper = T             ! Automatic bookkeeping?
logical space_charge_on = F            ! Space charge switch
logical coherent_synch_rad_on = F       ! csr
logical spin_tracking_on = F            ! spin tracking?
logical radiation_damping_on = F        ! Damping toggle.
logical radiation_fluctuations_on = F   ! Fluctuations toggle.
logical conserve_taylor_maps = T        ! Enable bookkeeper to set
                                         ! ele%taylor_map_includes_offsets = F?
logical use_ptc_layout_default = F       ! Default for lat%use_ptc_layout
logical absolute_time_tracking_default = F ! Default for lat%absolute_time_tracking
logical auto_scale_field_phase_default = T ! Default for lat%auto_scale_field_phase
logical auto_scale_field_amp_default = T  ! Default for lat%auto_scale_field_amp
logical debug = F                      ! Used for code debugging.
end type

```

%max_aperture_limit

Sets 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

Sets the orbit displacement used in the routine that calculates the transfer matrix through an element via tracking. **%d_orb** needs to be large enough to avoid significant round-off errors but not so large that nonlinearities will affect the results. Also see **%mat6_track_symmetric**.

%default_ds_step

Step size for tracking code §5 that uses a fixed step size. For example, **symp_lie_ptc** and **boris** tracking.

%significant_length

Sets the scale to decide if two length values are significantly different. For example, The superposition code will not create any super_slave elements that have a length less than this.

%rel_tol_tracking

Relative tolerance to use in tracking. Specifically, Tolerance to use when finding the closed orbit.

%abs_tol_tracking

Absolute tolerance to use in tracking. Specifically, Tolerance to use when finding the closed orbit.

%rel_tol_adaptive_tracking

Relative tolerance to use in adaptive tracking. This is used in **runge_kutta** and **time_runge_kutta** tracking (§5.4).

%abs_tol_adaptive_tracking

Absolute tolerance to use in adaptive tracking. This is used in **runge-kutta** and **time_runge_kutta** tracking (§5.4).

%init_ds_adaptive_tracking

Initial step to use for adaptive tracking. This is used in **runge-kutta** and **time_runge_kutta** tracking (§5.4).

%min_ds_adaptive_tracking

This is used in `runge-kutta` and `time_runge_kutta` tracking (§5.4). Minimum step size to use for adaptive tracking. If To be useful, `%min_ds_adaptive_tracking` must be set larger than the value of `%fatal_ds_adaptive_tracking`. In this case, particles are never lost due to taking too small a step.

%fatal_ds_adaptive_tracking

This is used in `runge-kutta` and `time_runge_kutta` tracking (§5.4). If the step size falls below the value set for `%fatal_ds_adaptive_tracking`, a particle is considered lost. This prevents a program from “hanging” due to taking a large number of extremely small steps. The most common cause of small step size is an “unphysical” magnetic or electric field.

%taylor_order

Cutoff Taylor order of maps produced by `sym_lie_ptc`.

%default_integ_order

Order of the the integrator used by Etienne Forest’s PTC code (§22.1).

ptc_max_fringe_order

Maximum order for computing fringe field effects in PTC.

%sr_wakes_on

Toggle for turning on or off short-range higher order mode wake field effects.

%lr_wakes_on

Toggle for turning on or off long-range higher order mode wake field effects.

%mat6_track_symmetric

Toggle to turn off whether the transfer matrix from tracking routine (`twiss_from_tracking`) tracks 12 particles at both plus and minus `%d_orb` values or only tracks 7 particles to save time.

%auto_bookkeeper

Toggles automatic or intelligent bookkeeping. See section §25.6 for more details.

%space_charge_on

Toggle to turn on or off high energy space charge effect in particle tracking.

%coherent_synch_rad_on

Toggle to turn on or off the coherent space charge calculation.

%spin_tracking_on

Determines if spin tracking is performed or not.

%radiation_damping_on

Toggle to turn on or off effects due to radiation damping in particle tracking.

%radiation_fluctuations_on

Toggle to turn on or off effects due to radiation fluctuations in particle tracking.

%conserve_taylor_maps

Toggle to determine if the Taylor map for an element include any element “misalignments”. See Section §5.6 for more details.

%use_ptc_layout_default

Default setting to be applied to a lattice if `use_ptc_layout` is not specified in a lattice file. This feature has not yet been implementd.

%absolute_time_tracking_default

Default setting to be applied to a lattice if `absolute_time_tracking` (§8.1) is not specified in a lattice file. Additionally, if an element that is not associated with a lattice is tracked, `%absolute_time_tracking_default` will be used to determine whether absolute time tracking is used.

%auto_scale_field_phase_default

Default setting to be applied to a lattice if `auto_scale_field_phase` (§8.1) is not specified in a lattice file. Additionally, if an element that is not associated with a lattice is tracked, this parameter will be used to determine whether phase auto scaling is done.

%auto_scale_field_amp_default

Default setting to be applied to a lattice if `auto_scale_field_amp` (§8.1) is not specified in a lattice file. Additionally, if an element that is not associated with a lattice is tracked, this parameter will be used to determine whether amplitude auto scaling is done.

%debug

Used for communication between program units for debugging purposes.

10.2 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 "quick".
  real(rp) random_sigma_cutoff = -1     ! -1 => no cutoff used.
  real(rp) center_jitter(6) = 0.0       ! Bunch center rms jitter
  real(rp) emit_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 = T            ! Renormalize centroid?
  logical renorm_sigma = T             ! Renormalize sigma?
  !!! The following are used by all distribution types
  type(beam_spin_struct) spin           ! Spin
  real(rp) a_norm_emit                ! a-mode normalized emittance (=  $\gamma\epsilon$ )
  real(rp) b_norm_emit                ! b-mode normalized emittance (=  $\gamma\epsilon$ )
  real(rp) a_emit                    ! a-mode emittance (=  $\gamma\epsilon$ )
  real(rp) b_emit                    ! b-mode emittance (=  $\gamma\epsilon$ )
  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.
integer species = not_set$      ! Species. Default is branch reference species.
logical init_spin = F          ! initialize beam spinors
logical full_6D_coupling_calc = F ! Use 6x6 1-turn mat to match distribution?
logical use_t_coords = .false.   ! If true, the distributions will be
                                  ! calculated using time coordinates
logical use_z_as_t   = .false.   ! Only used if use_t_coords = .true.
                                  ! If true, particles will be distributed in t
                                  ! If false, particles will be distributed in s
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:

```

 "", or "RAN_GAUSS"  ! Random distribution (default).
 "ELLIPSE"           ! Ellipse distribution (§15.1.1)
 "KV"                ! Kapchinsky-Vladimirsky distribution (§15.1.2)
 "GRID"              ! Uniform distribution.

```

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.

Unlike all other distribution types, the GRID distribution is independent of the Twiss parameters at the point of generation. For the non-GRID distributions, the distributions are adjusted if there is local x - y coupling (§17.1). If `full_6D_coupling_calc` is set to True, the full 6-dimensional coupling matrix is used. If False, which is the default, The 4-dimensional \mathbf{V} matrix of Eq. (17.3) is used.

The parameters common to all the distribution types are marked in the `beam_init_struct` above. Either `a_norm_emit` or `a_emit` may be set but not both. similarly, either `b_norm_emit` or `b_emit` may be set but not both.

The parameters for the random distribution 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 "quick" method is somewhat faster than the "exact" method but not as accurate. With either conversion method, if `%random_sigma_cutoff` is set to a positive number, this limits the maximum sigma generated.

`%n_parcicle`

Number of random 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, %emit_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 (§15.1.1). 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 (§15.1.2)

```
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
```

The total number particles generated is the product of the individual distributions. For example:

```
type (beam_init_struct) bi
bi%distribution_type = ELLIPSE", "ELLIPSE", "GRID"
bi%ellipse(1)%n_ellipse = 4
bi%ellipse(1)%part_per_ellipse = 8
bi%ellipse(2)%n_ellipse = 3
bi%ellipse(2)%part_per_ellipse = 100
bi%grid(3)%n_x = 20
bi%grid(3)%n_px = 30
```

The total number of particles per bunch will be $32 \times 300 \times 600$. The exception is that when `RAN_GAUSS` is mixed with other distributions, the random distribution is overlayed with the other distributions instead of multiplying. For example:

```
type (beam_init_struct) bi
bi%distribution_type = RAN_GAUSS", "ELLIPSE", "GRID"
bi%ellipse(2)%n_ellipse = 3
bi%ellipse(2)%part_per_ellipse = 100
bi%grid(3)%n_x = 20
bi%grid(3)%n_px = 30
```

Here the number of particle is 300×600 . Notice that when `RAN_GAUSS` is mixed with other distributions, the value of `beam_init%n_particle` is ignored.

distributions are taken as describing particles in t -coordinates (§13.6). Furthermore, if `use_z_as_t` is true, then the z coordinates from the distribution will be taken as describing the time coordinates. For

example, particles may originate at a cathode at the same s , but different times. If false, then the z coordinate from the distribution describes particles at the same time but different s positions, and each particle gets `%location=inside$`. In this case, the bunch will need to be tracked with a tracking method that can handle inside particles, such as `time_runge_kutta`. All particles are finally converted to proper s -coordinate distributions for Bmad to use.

10.3 CSR Parameters

The Coherent Synchrotron Radiation (CSR) calculation is discussed in Section §16.3. Besides the parameters discussed below, the `coherent_synch_rad_on` parameter in Section §10.1 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 = T     ! Longitudinal csr component
    logical lsc_component_on = T      ! Longitudinal space charge component
    logical tsc_component_on = T      ! Transverse space charge component
    logical small_angle_approx = T   ! Use lcsr small angle approximation?
    logical print_taylor_warning = .true. ! Print Taylor element warning?
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 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. §16.3) 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.

Taylor map elements (§3.42) that have a finite length cannot be subdivided for the CSR calculaton. *Bmad* will ignore any `taylor` elements present in the lattice but will print a warning that it is doing so. So suppress the warning, `print_taylor_warning` should be set to False.

10.4 DE Optimizer Parameters

The Differential Evolution (DE) optimizer is used in nonlinear optimization problems. This optimizer is based upon the work of Storn and Price[[Storn96](#)]. There are a number of parameters that can be varied to vary how the optimizer works. These parameters are contained in a structure named `opti_de_param_struct`. the instance name is `opti_de_param`. This structure has components

Default

```
real(rp) CR          0.8    ! Crossover Probability.
real(rp) F           0.8    !
real(rp) l_best       0.0    ! Percentage of best solution used.
logical binomial_cross False   ! IE: Default = Exponential.
logical use_2nd_diff  False   ! use F * (x_4 - x_5) term
logical randomize_F   False   !
logical minimize_merit True    ! F => maximize the Merit func.

The "perturbed vector" is v = x_1 + l_best * (x_best - x_1) + F * (x_2 - x_3) + F * (x_4 - x_5)
The last term F * (x_4 - x_5) is only used if use_2nd_diff = T.
```

The crossover can be either "Exponential" or "Binary". Exponential crossover is what is described in the paper. With Exponential crossover the crossover parameters from a contiguous block and the average number of crossover parameters is approximately average crossovers $\sim \min(D, CR / (1 - CR))$ where D is the total number of parameters. With Binary crossover the probability of crossover of a parameter is uncorrelated with the probability of crossover of any other parameter and the average number of crossovers is average crossovers = $D * CR$

`randomize_F` = True means that the F that is used for a given generation is randomly chosen to be within the range $[0, 2^*F]$ with average F.

10.5 Programming Common Parameters

The `global_common_struct` is meant to hold common parameters that should not be modified by the user.

```
type global_common_struct
  logical be_thread_safe = .false.      ! Avoid thread unsafe practices?
end type
```

%be_thread_safe

Toggle to prevent non thread safe calculational optimizations from being done. Currently, converting *Bmad* to be thread safe is an active project. Please contact the *Bmad* maintainers for more details.

Chapter 11

Lattice Examples

This chapter gives some examples of how lattice files can be constructed to describe various machine geometries.

11.1 Example: Injection Line

An injection line is illustrated in Fig. 11.1. In this example, The path of an injected particle after it leaves the last element X of the injection line (dashed blue line) partially goes through the field of the dipole BND in the storage ring. One way to simulate this is:

```
INJ_L: line = (... , X, P, BND2, BR)
RING_L: line = (... , BND, M, ...)
P: patch, x_offset = -0.5, x_pitch = 0.15, z_offset = 0.3
BND: sbend, l = 6.2, g = 1/52
BND2: BND, l = 4.7, tracking_method = runge_kutta,
      field_calc = grid, field = {...}
BR: fork, to_line = RING_L, to_element = M
M: marker
use, INJ_L
```

In order to properly track particles through the fringe field of the dipole BND, a partial section of BND, called BND2, is placed in the injection line INJ_L. The `tracking_method` for BND2 is set to `runge_kutta` since the default `bmad_standard` tracking is not able to handle these fringe fields. Additionally, the

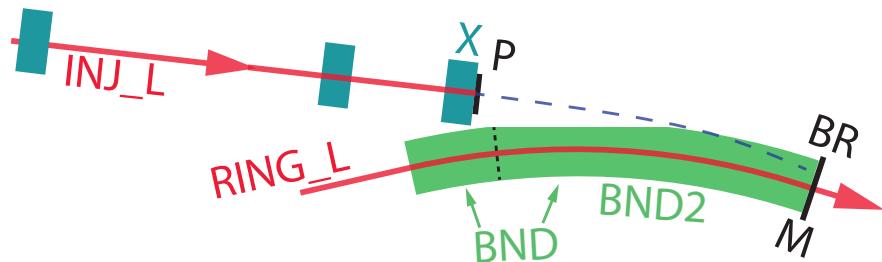


Figure 11.1: Injection line into a dipole magnet.

`field_calc` parameter of `BND2` is set to `grid` so that the actual field profile of this particular magnet can be used in the tracking. The field is specified in the `field` parameter (§4.13).

After traversing element `X` in the injection line, the particle goes through the patch `P` which offsets the reference trajectory so that following element, `BND2`, is properly positioned. The beginning of `BND2` is marked by a black dashed line in the figure. At the end of `BND2` the fork element `BR` connects `INJ_L` with the marker `M` in `RING_L`.

11.2 Example: Energy Recovery Linac

An Energy Recovery Linac (ERL) is illustrated in Fig. 11.2A. The ERL starts with an injection line that feeds a linac which accelerates the beam to some energy. The beam then transverses a return arc which reinjects the bunches into the linac. The length of the return arc is such that, on the second pass, the beam is decellerated giving its energy back to the RF cavities. Finally, the decellerated beam is steered through a dump line where, at the end, an absorber stops the beam.

A lattice file for modeling this ERL:

```
parameter[geometry] = open
parameter[absolute_time_tracking] = T

BEND_L1: sbend, angle = -25*pi/180, l = 0.2, ...
BEND_L2: BEND_L1

A_PATCH: patch, flexible = T
D_PATCH: patch, x_offset = 0.034, x_offset = asin(0.32)
INJECT: line = (...)
LINAC: line[multipass] = (BEND_L1, ..., BEND_L2)
ARC: line = (..., BEND_A7)
DUMP: line = (...)

ERL: line = (INJECT, LINAC, ARC, A_PATCH, LINAC, D_PATCH, DUMP)
```

Fig. 11.2B shows the injector and arc merging into the beginning of the linac. The first element of the linac is a bend named `BEND_L1`. The bending angle for `BEND_L1` has been set at the appropriate value for injection from the injector. To get the correct geometry for injection from the arc, a `patch` element, named `A_PATCH`, is placed in the `ERL` line between the arc and the linac. `A_PATCH` is a flexible patch which means that the exit edge of `A_PATCH` will automatically be aligned with the entrance edge of the next element which is `BEND_L1`.

Note that this use of a flexible patch works since the orientation of `BEND_L1` has been determined before the orientaiton of `A_PATCH` is determined. The orientaiton of elements is determined in order starting from the first element in the line (the exception to this rule is if there is a `floor_position` element) and the orientation of `BEND_L1` is thus determined right after the injector section on the first pass through the linac.

Fig. 11.2C shows the end of the linac splitting off into the dump and arc sections. The 1

A patch element named `A_PATCH` is used

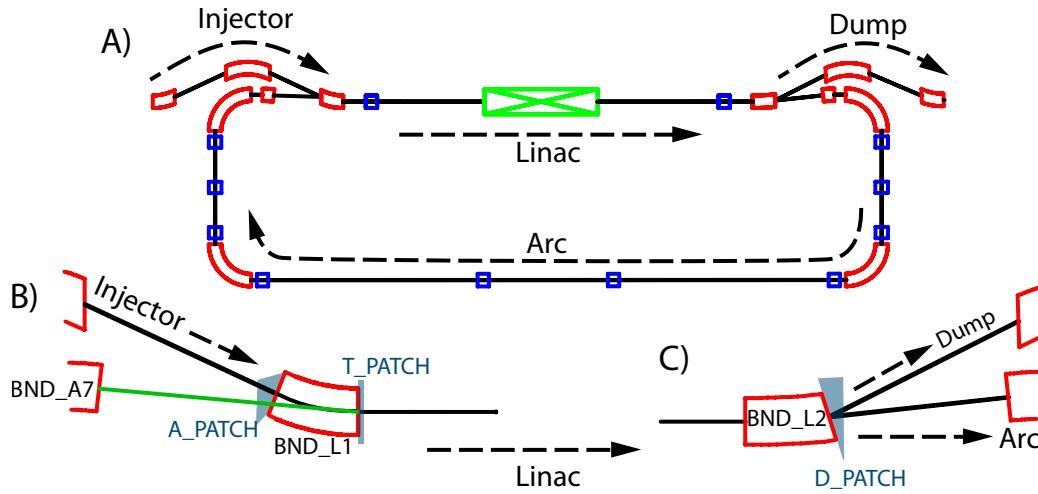


Figure 11.2: Example Energy Recovery Linac. A) The ERL consists of an injection line, accelerating linac, return arc, decellerating linac, and finally a beam dump. B) Close up of the section where the end of the injector and the end of the arc inject into the beginning of the linac. C) Close up of the end of the linac which injects into the dump and the beginning of the arc.

11.3 Example: Patch Between reversed and non-reversed elements

Between normal and reversed elements there must be a reflection patch element (§3.34). This is illustrated in Fig. 11.3. The basic lattice is

```
D: drift, l = 2
g_design = pi/12
B: sbend, l = 2, g = g_design, g_err = -2*g_design
P: patch, x_pitch = pi
fig_A: line = (D, --B)      ! Illegal. Do not use!
fig_B: line = (D, P, --B)   ! Correct
```

Line `fig_A` represents the situation shown in Fig. 11.3A. With no patch between the drift D and the reversed bend B, a particle leaving D at D's downstream end will find itself outside of both D and B. Clearly this is an unphysical situation. Sanity is restored in line `fig_B` shown in Fig. 11.3B. In this

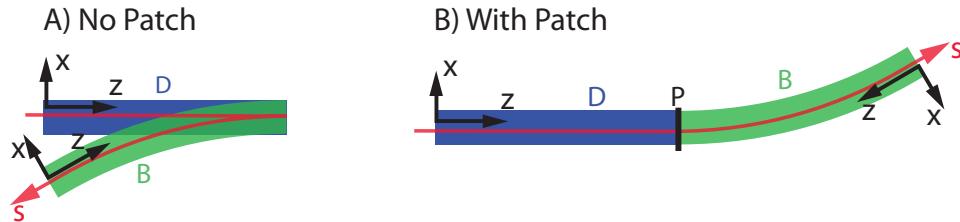


Figure 11.3: Drift element D is followed by a reversed drift element B. The view is from +y onto the x-z plane. A) If no patch is present then the geometry does not make physical sense. B) With a patch in between, a sane geometry can be obtained.

instance, the patch P rotates the reference coordinates around the y -axis leaving the y -axis invariant the bend of B is in the x - z plane. There are other patch parameter values that could be used to produce a reflection patch (§13.3.5). For example, Setting the patch's `y_pitch` to π would produce a reflection patch.

Since bend B is reversed, A particle moving downstream within B is going the opposite direction from the normal direction. If `g_err` were zero in this instance, a downstream moving particle would feel a force that will rotate the particle in a clockwise manner opposite from the counterclockwise direction of the bend. To counter this, `g_err` is set so the total bending field $g_{tot} = g + g_{err}$ is opposite the design field. That is, `g_err` is set so that $g_{tot} = -g$.

11.4 Example: Colliding Beam Storage Rings

The idealized layout of a pair of storage rings used for colliding counter rotating beams is shown in Fig. 11.4. Rings A and B intersect at two interaction regions labeled `ir1` and `ir2` where the beams collide. The basic lattice description is:

```
ir1: line[multipass] = (...)

ir2: line[multipass] = (...)

m: marker

fid: fiducial, origin_ele = m

...
a: line = (arc_a1, pa1_in, ir1, m, pa1_out, arc_a2, pa2_in, ir2, pa2_out)
b_rev: line = (arc_b1, pb1_in, ir1, fid, pb1_out, arc_b2, pb2_in, ir2, pb2_out)
b: line = (--b_rev)
use, a, b
```

Lines `ir1` and `ir2` are the two interaction regions which are declared `multipass` since they are shared by the two rings. Line `a` represents ring A where the beam which, by definition, travels in the same direction as increasing s , rotates clockwise. Line `b_rev` is a “reversed” line of ring B and, like `a`, represents a beam rotating clockwise. Line `b`, which represents ring B, is the reverse of `b_rev` and here the beam rotates counterclockwise. In this construction, all elements of `b` are reversed. While this is not mandatory (only the interaction regions must be reversed in `b`), having all of `b` reversed simplifies the geometry since this means that the local coordinate systems of both lines `a` and `b` will be “aligned” with the x -axis pointing to the outside of the ring and the y -axis pointing up, out of the page. Having non-aligned coordinate systems is possible but potentially very confusing.

The two rings are physically aligned using a marker `m` in `a` and a `fiducial` element `fid` in `b` that aligns with `m`. Each ring has four rigid `patch` elements, whose name begins with `p`, on either side of each interaction region.

The finished lattice will have two branches, The first branch (with index 0) will be derived from line `a` (and hence will be named “a”) and the second branch (with index 1) will be derived from line `b` (and hence will be named “b”). The multipass lords representing the physical IR elements will be in the “lord section” of branch 0.

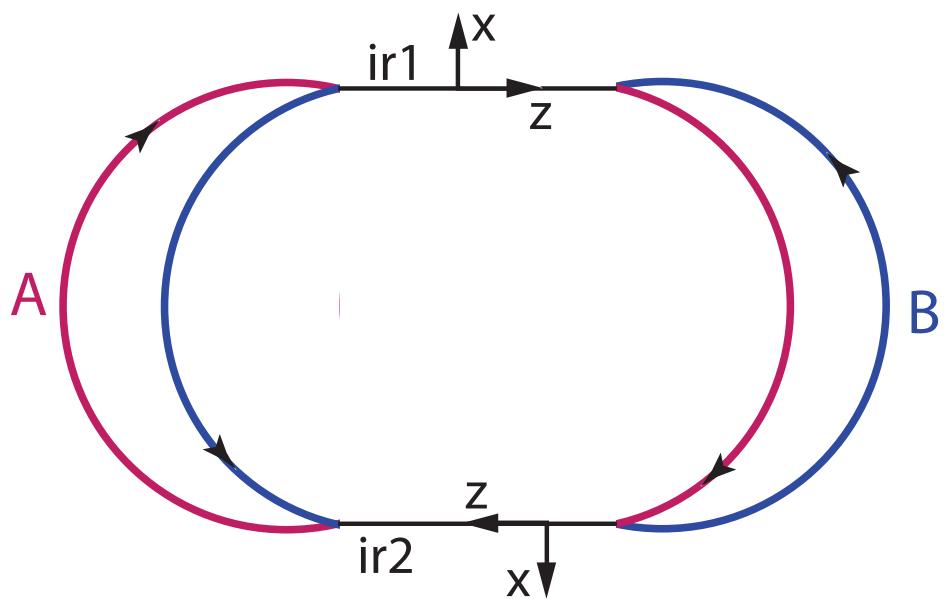


Figure 11.4: Dual ring colliding beam machine. The beam in the A ring rotates clockwise and in the B ring counterclockwise.

Chapter 12

MAD/XSIF/SAD Lattice Conversion

12.1 MAD Conversion

Conversion of lattice files from *MAD* to *Bmad* format can be done using the **Universal Accelerator Parser** (§12.4). Due to differences in language definitions, the conversions must be done with some care. The following differences should be noted:

- *Bmad*, unlike *MAD*, does not have any “action” commands. An action command is a command that makes a calculation. Examples include *MAD*’s SURVEY and TWISS commands.
- In *Bmad* all variables must be defined. In *MAD* undefined variables will default to 0.
- In *Bmad* all variables must be defined before being used (§2.12) while *MAD* does not have this constraint.
- *Bmad*, unlike *MAD*, does not allow variable values to be redefined.
- Elements like a `sad_mult` cannot be translated.

Besides using the **Universal Accelerator Parser** for conversion from *Bmad* to *MAD*, there is a conversion routine called `bmad_to_mad_or_xsif`. The advantage of this routine is that since *MAD* does not have a `wiggler` or a `sol_quad` element, this conversion routine can make an “equivalent” substitution. For a `sol_quad`, the equivalent substitution will be a drift-matrix-drift series of elements. For a `wiggler`, a series of bend and drift elements will be used (the program can also use a drift-matrix-drift model here but that is not as accurate). The bends and drifts for the `wiggler` model are constructed so that the global geometry of the lattice does not change. Additionally the bends and drifts are constructed to most nearly match the wiggler’s

```
Transfer matrix
$I_2$ and $I_3$ synchrotron radiation integrals (§16.2)
```

Note that the resulting model will not have the vertical cubic nonlinearity that the actual wiggler has.

The `bmad_to_mad_or_xsif` routine is embeded in the program `util_programs/bmad_to_mad_or_xsif` which can be used for *Bmad* to *MAD* or *XSIF* conversions.

12.2 XSIF Conversion

XSIF[Tenen01], developed at SLAC, stands for “Extended Standard Input Format.” XSIF is essentially a subset of the MAD [Grote96] input format.

Bmad has software to directly parse XSIF files so XSIF files may be used in place of *Bmad* lattice files. With some restrictions, an XSIF lattice file may be called from within a *Bmad* lattice file. See Section §2.16 for details.

Since XSIF does not have a `parameter[geometry]` statement (§8.1), 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.

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.

To convert between XSIF and *Bmad* the `Universal Accelerator Parser` can be used (§12.4). Additionally, the `bmad_to_mad_or_xsif` routine in *Bmad* can convert from *Bmad* to XSIF (cf. §12.1).

12.3 SAD Conversion

Conversion from SAD[SAD] to *Bmad* is accomplished using the Python script

```
util_programs/sad_to_bmad/sad_to_bmad.py
```

Currently, a converter from *Bmad* to SAD has not been implemented.

Currently, the following restrictions on SAD lattices apply:

- SAD must elements cannot have an associated RF field
- Misalignments in a `sol` element with `geo = 1` cannot be handled.

12.4 Translation Using the Universal Accelerator Parser

The Accelerator Markup Language (AML) / Universal Accelerator Parser (UAP) project[AML] 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.

The program `translate_driver` which comes with the UAP code can be used for conversions. To get help with how to run this program use the command

```
path-to-uap-dir/bin/translate_driver -help
```

Example:

```
path-to-uap-dir/bin/translate_driver -constants_first -bmad xxx.madx
```

This will convert a *MAD-X* file `xxx.madx` to *Bmad* format.

Part II

Conventions and Physics

Chapter 13

Coordinates

13.1 Local Reference Orbit

The local reference orbit is the curved path used to define a coordinate system for describing a particle's position as shown in Fig. 13.1. The reference orbit is also used for orientating lattice elements in space. At a given time t , a particle's position can be described by a point 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

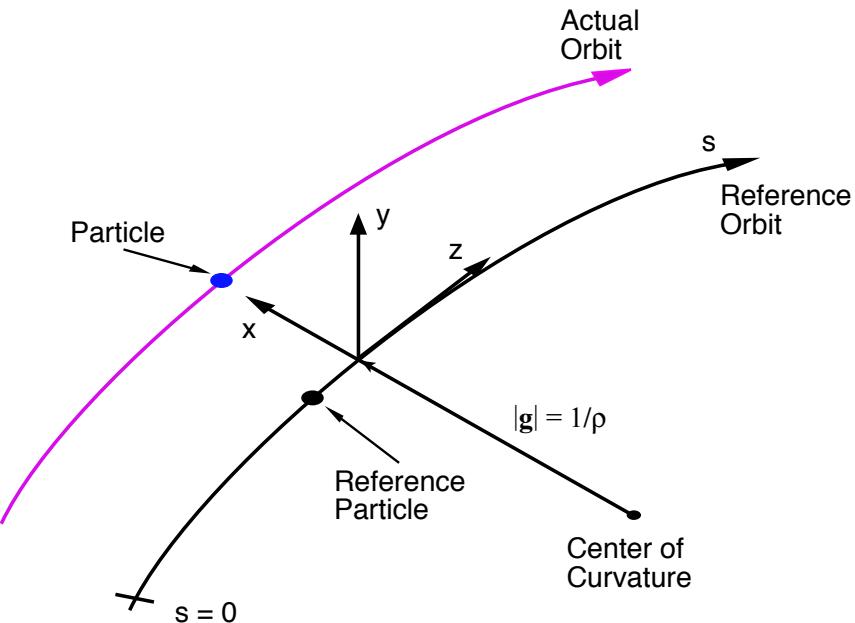


Figure 13.1: The local reference coordinate system. By construction, a particle's z coordinate is zero. This is not to be confused with the phase space z coordinate (§13.4). The curvature vector \mathbf{g} lies in the x - y plane and has a magnitude of $1/\rho$ where ρ is the bending radius. The z -axis will normally be parallel to the s -axis but for reversed elements it will be antiparallel.

to the reference orbit. The z -axis will generally be pointing in the direction of increasing s but, as discussed in detail below, will point counter to s for elements that are **reversed**. The x and y -axes are perpendicular to the reference orbit. As will be shown later, 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 $z = 0$. The coordinate system so constructed is called the “**local coordinate system**” or sometimes the “**laboratory coordinate system**” when there is need to distinguish it from the “**element coordinate system**” (§19.1) which is attached to the physical element.

There is a separate reference orbit for each branch (§1.2) of a lattice. When there are multiple branches, the reference orbit of a branch must not depend upon the configuration of branches later on in the array of branches in the lattice. As a consequence, the reference orbits of the branches can be calculated one at a time starting with the first branch.

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 element is centered with respect to the reference curve. However, by specifying offsets, pitches or a tilt (See §4.4), the physical element may be arbitrarily shifted 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 (§27.2). The former is fixed by the lattice while the latter can be any arbitrary orbit.

13.2 Reference Orbit Construction

Another way of thinking about the reference orbit is to imagine that each element has assigned to it a local coordinate reference frame assigned to it in the shape of a LEGO block¹ as shown in Fig. 13.2. Every block has an “**entrance**” and an “**exit**” reference frame. For most types of elements, the LEGO block for the element is “straight” as shown in Fig. 13.2A. That is, the reference curve through the block is a straight line segment and the length of the block is the length of the element. For a **bend** (§3.5), the reference curve through the block is a segment of a circular arc as shown in Fig. 13.2B. With a zero

¹Thanks to Dan Abell for this analogy.

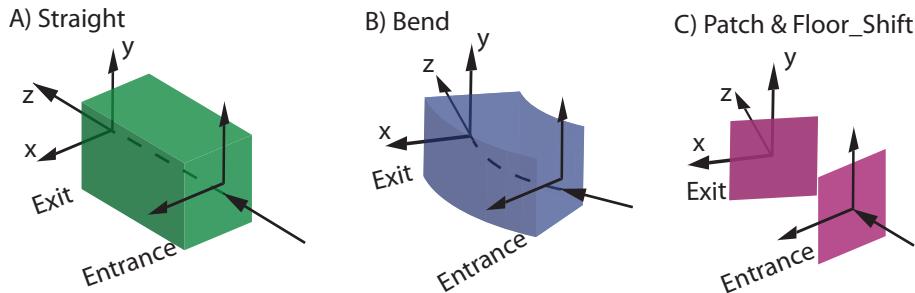


Figure 13.2: Element reference frame LEGO blocks. Shown are the blocks along with the entrance and exit reference frames. The physical element may be displaced in the local coordinate frame using offsets, tilt, and pitches. A) For straight line elements the two frames are colinear. B) For bend elements, the two frames are rotated with respect to each other. C) For **patch** and **floor_shift** elements the exit frame may be arbitrarily positioned with respect to the entrance.

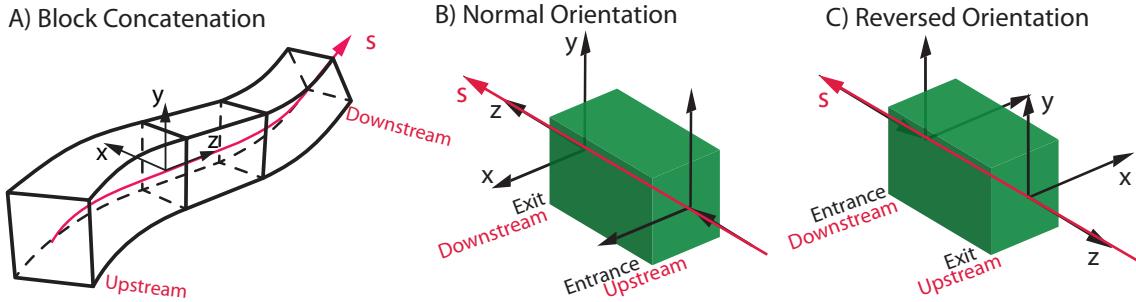


Figure 13.3: Element LEGO block concatenation. A) The reference orbit is constructed by concatenating the LEGO blocks together. B) For normal (non-reversed) elements the z -axis is parallel with the s -axis. C) For reversed elements the z -axis is antiparallel with the s -axis. By definition, the “entrance” and “exit” ends of an element are fixed relative to the element (relative to the z -axis) while the “upstream” and “downstream” ends are fixed relative to the s -axis.

`ref_tilt`, the rotation axis between the entrance and exit frames of a bend is the y -axis (§13.3). For `patch` (§3.34), and `floor_shift` (§3.17) elements, the exit face can be arbitrarily oriented with respect to the entrance end. In this case, the reference orbit between the entrance and exit faces is not defined.

Assuming for the moment that there are no `fiducial` elements present, the construction of the reference orbit starts at the beginning element of a branch. If the branch is a `root` branch (§1.3), The orientation of the beginning element can be set via the appropriate positioning statements (§8.4). If the branch is not a `root` branch, the position of the beginning element is determined by the position of the `fork` or `photon_fork` element from which the branch forks from.

Once the beginning element in a branch is positioned, succeeding element LEGO blocks are concatenated together as illustrated in Fig. 13.3A. When a block is joined, the end of the block that is mated to the previous block is called the “upstream” end and the other end which will be mated to the following block is called the “downstream” end. Normally, the `entrance` end of a block is used as the `upstream` end and the `exit` end is used as the `downstream` end as shown in Fig. 13.3B. However, for reversed elements (§6.3), the `upstream` end is the `exit` end of the block and vice versa. To put it another way, the `entrance` and `exit` end of the blocks reference the physical element. Thus, for example, the `e1` edge of a bend (§3.5) is always at the `entrance` face and the `e2` is always at the `exit` face. Also the field of a wiggler is (§3.43) is referenced to $z = 0$ which is at the `entrance` end. The `upstream` and `downstream` ends, on the other hand, are referenced to the reference orbit and the `downstream` end always is at a larger s position relative to the `upstream` end.

In all cases, when a LEGO block is joined to the previous block, the coordinate system at the mating end of the block (the `upstream` end) will be aligned with the mating end of the previous block (the `downstream` end). The situation where one of the blocks is reversed and the other one not, and neither is a `patch` nor `floor_shift` element, does not make physical sense since a particle which is moving downstream, when it comes to the face joining the blocks will have to magically reverse direction in order to travel through the next block. Thus, to have normal and reversed elements in a branch, `reflection patches` must be used in between. Whether it makes physical sense to put a `patch` element next to another element is more complicated. This depends upon whether the other element is reversed or not and whether the `patch` is reversed, whether the `patch` is `reflecting`, and the sign of `z_offset` for the `patch`. The basic criterion is that a particle leaving one block must enter the next. See Section §11.3 for an example.

If there are `fiducial` elements, the local reference frame is constructed beginning at these elements.

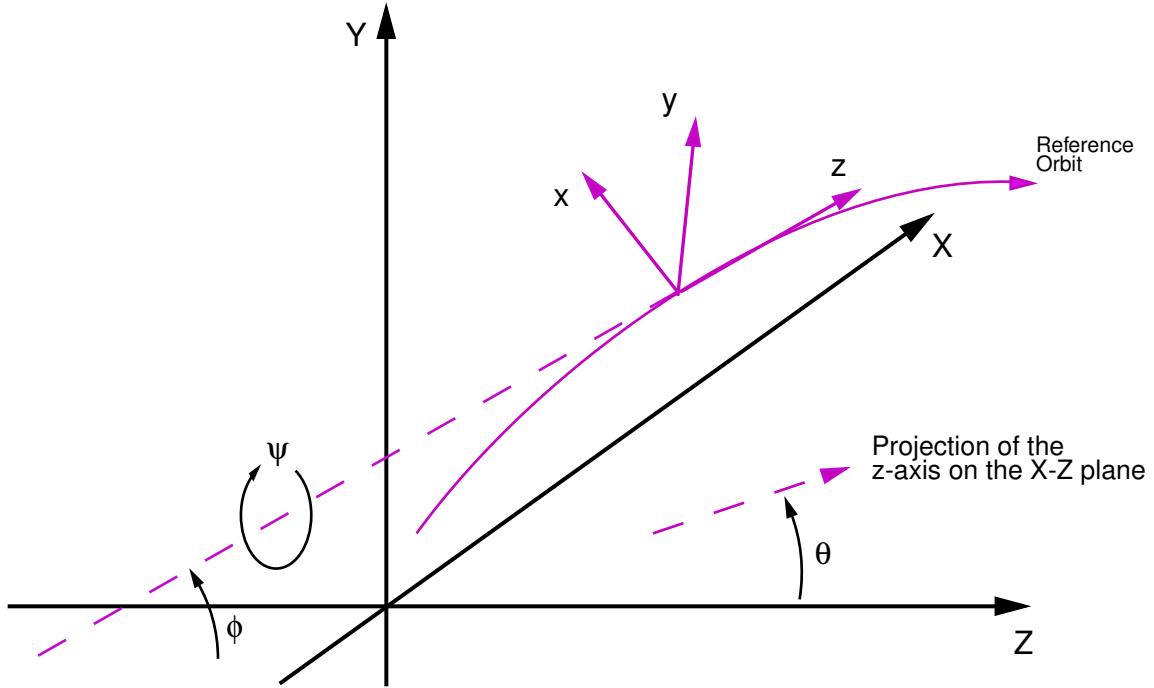


Figure 13.4: The Global Coordinate System.

13.3 Global Coordinates

The Cartesian **global** coordinate system, also called the ‘floor’ coordinate system, is the coordinate system “attached to the earth” that is used to describe the local coordinate system. Following the **MAD** convention, the **global** coordinate axis are labeled (X, Y, Z) . Conventionally, Y is the “vertical” coordinate and (X, Z) are the “horizontal” coordinates. To describe how the local coordinate system is oriented within the global coordinate system, each point on the s -axis of the local coordinate system is characterized by its (X, Y, Z) position and by three angles θ , ϕ , and ψ that describe the orientation of the local coordinate axes as shown in Fig. 13.4. These three angles are defined as follows:

θ 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.

ϕ 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.

ψ 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 ψ forms a right-handed screw with the z -axis.

By default, at $s = 0$, the reference orbit’s origin coincides with the (X, Y, Z) origin and the x , y , and z axes correspond to the X , Y , and Z axes respectively. θ decreases as one follows the reference orbit when going through a horizontal bend with a positive bending angle. This corresponds to x pointing radially outward. Without any vertical bends, the Y and y axes will coincide, and ϕ and ψ will both be zero. The **beginning** statement (§8.4) 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}

$$\mathbf{V} = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \quad (13.1)$$

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, z) . \mathbf{W} can be expressed in terms of the angles θ , ϕ , and ψ via the formula

$$\mathbf{W} = \mathbf{W}_\Theta \mathbf{W}_\Phi \mathbf{W}_\Psi \quad (13.2)$$

where

$$\mathbf{W}_\Theta(\theta) = \begin{pmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{pmatrix}, \quad \mathbf{W}_\Phi(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \phi & \sin \phi \\ 0 & -\sin \phi & \cos \phi \end{pmatrix}, \quad \mathbf{W}_\Psi(\psi) = \begin{pmatrix} \cos \psi & -\sin \psi & 0 \\ \sin \psi & \cos \psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (13.3)$$

13.3.1 Lattice Element Positioning

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

$$\mathbf{V}_i = \mathbf{W}_{i-1} \mathbf{L}_i + \mathbf{V}_{i-1}, \quad (13.4)$$

$$\mathbf{W}_i = \mathbf{W}_{i-1} \mathbf{S}_i \quad (13.5)$$

\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 dropped. For all elements whose reference orbit through them is a straight line, the corresponding \mathbf{L} and \mathbf{S} are

$$\mathbf{L} = \begin{pmatrix} 0 \\ 0 \\ L \end{pmatrix}, \quad \mathbf{S} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad (13.6)$$

Where L is the length of the element.

For a **bend**, the axis of rotation is dependent upon the bend's **ref_tilt** attribute as shown in Fig. 13.5A. The axis of rotation points in the negative y_0 direction for **ref_tilt** = 0 and is offset by the bend radius **rho**. Here (x_0, y_0, z_0) are the local coordinates at the entrance end of the bend with the z_0 axis being directed into the page in the figure. For a non-zero **ref_tilt**, the rotation axis is itself rotated about the z_0 axis by the value of **ref_tilt**. Fig. 13.5B shows the exit coordinates for four different values of **ref_tilt** and for a bend angle **angle** of $\pi/2$.

For a bend, \mathbf{L} and \mathbf{S} are given by

$$\mathbf{L} = \mathbf{T} \tilde{\mathbf{L}}, \quad \mathbf{S} = \mathbf{T} \tilde{\mathbf{S}} \mathbf{T}^{-1} \quad (13.7)$$

where

$$\tilde{\mathbf{L}} = \begin{pmatrix} \rho(\cos \alpha_b - 1) \\ 0 \\ \rho \sin \alpha_b \end{pmatrix}, \quad \tilde{\mathbf{S}} = \begin{pmatrix} \cos \alpha_b & 0 & -\sin \alpha_b \\ 0 & 1 & 0 \\ \sin \alpha_b & 0 & \cos \alpha_b \end{pmatrix}, \quad \mathbf{T} = \begin{pmatrix} \cos \theta_t & -\sin \theta_t & 0 \\ \sin \theta_t & \cos \theta_t & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (13.8)$$

with ρ being the bend radius (**rho**), α_b is the bend **angle** (§3.5), and θ_t is the **ref_tilt** angle (§4.4). Without a **ref_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 θ .

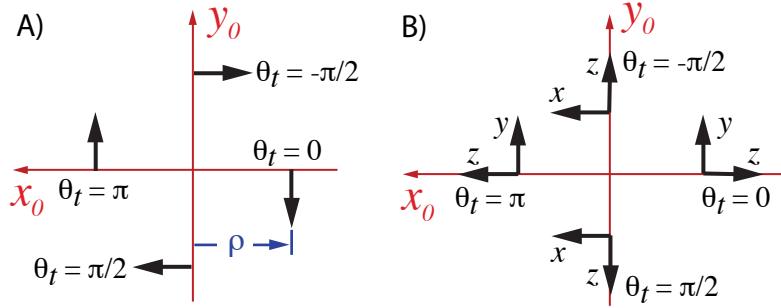


Figure 13.5: A) Rotation axes for four different `ref_tilt` angles of $\theta_t = 0, \pm\pi/2$, and π . (x_0, y_0, z_0) are the local coordinates at the entrance end of the bend with the z_0 axis being directed into the page. Any rotation axis will be displaced by a distance of the bend radius `rho` from the origin. B) The (x, y, z) coordinates at the exit end of the bend for the same four `ref_tilt` angles. In this case the bend angle is taken to be $\pi/2$.

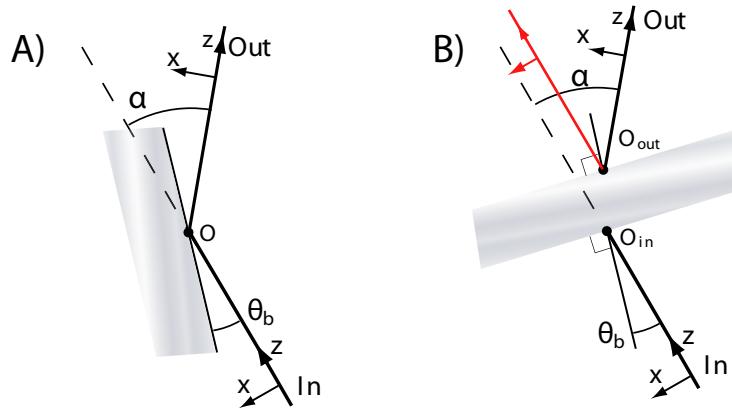


Figure 13.6: Mirror and crystal geometry. The geometry shown here is appropriate for a `ref_tilt` angle of $\theta_t = 0$. θ_g is the bend angle of the incoming (entrance) ray, and α_b is the total bend angle of the reference trajectory. A) Geometry for a mirror or a Bragg crystal. Point O is the origin of both the local coordinates just before and just after the reflection/diffraction. B) Geometry for a Laue crystal. Point O_{out} is the origin of the coordinates just after diffraction is displaced from the origin O_{in} just before diffraction due to the finite thickness of the crystal. here the bend angles are measured with respect to the line that is in the plane of the entrance and exit coordinates and perpendicular to the surface. For Laue diffraction, the user has the option of using the undiffracted beam (shown in red) as the reference trajectory.

The bend transformation (Eq. (13.7)) 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 π) 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 as shown in Fig. 13.5. That is, a lattice with only bends with $\theta_t = 0$ or π will lie in the horizontal plane (this assuming that the y -axis starts out pointing along the Y -axis as it does by default). For $\theta_t = \pm\pi/2$, the bend axis is the x -axis. A value of $\theta_t = +\pi/2$ represents a downward pointing bend.

13.3.2 Position Transformation When Transforming Coordinates

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

$$\mathbf{R}_{VW} = \mathbf{W}^{-1} (\mathbf{R} - \mathbf{V}) \quad (13.9)$$

This is essentially the inverse of Eq. (13.4). That is, vectors propagate inversely to the propagation of the coordinate system.

Using Eq. (13.9) with Eqs. (13.4), and (13.5), the transformation of a particle's position $\mathbf{r} = (x, y, z)$ and momentum $\mathbf{P} = (P_x, P_y, P_z)$ when the coordinate frame is transformed from frame $(\mathbf{V}_{i-1}, \mathbf{W}_{i-1})$ to frame $(\mathbf{V}_i, \mathbf{W}_i)$ is

$$\mathbf{r}_i = \mathbf{S}_i^{-1} (\mathbf{r}_{i-1} - \mathbf{L}_i), \quad (13.10)$$

$$\mathbf{P}_i = \mathbf{S}_i^{-1} \mathbf{P}_{i-1} \quad (13.11)$$

Notice that since \mathbf{S} (and \mathbf{W}) is the product of orthogonal rotation matrices, \mathbf{S} is itself orthogonal and its inverse is just the transpose

$$\mathbf{S}^{-1} = \mathbf{S}^T \quad (13.12)$$

13.3.3 Crystal and Mirror Element Coordinate Transformation

A **crystal** element (§3.28) diffracts photons and a **mirror** element (§3.28) reflects them. For a crystal setup for Bragg diffraction, and for a mirror, the reference orbit is modeled as a zero length bend with $\tilde{\mathbf{L}} = (0, 0, 0)$, as shown in Fig. 13.6A. Shown in the figure is the geometry appropriate for a **ref_tilt** angle of $\theta_t = 0$ (the rotation axis is here the y -axis). Since the mirror or crystal element is modeled to be of zero length, the origin points (marked \mathcal{O} in the figure) of the entrance and exit local coordinates are the same. For Laue diffraction, the only difference is that $\tilde{\mathbf{L}}$ is non-zero due to the finite thickness of the crystal as shown in Fig. 13.6B. This results in a separation between the entrance coordinate origin \mathcal{O}_{in} and the exit coordinate origin \mathcal{O}_{out} .

In all cases, the total bending angle is

$$\begin{aligned} \alpha_b &= \text{bragg_angle_in} + \text{bragg_angle_out} && ! \text{ Crystal} \\ \alpha_b &= 2 \text{graze_angle} && ! \text{ Mirror} \end{aligned} \quad (13.13)$$

With a mirror or Bragg diffraction, the bend angles are measured with respect to the surface plane. With Laue diffraction the bend angles are measured with respect to the line in the bend plane perpendicular to the surface.

For Laue diffraction, the user has the option of using the undiffracted beam (shown in red) as the reference trajectory.

The orientation of the exit coordinates (the local coordinates after the reflection) are only affected by the element's **ref_tilt** and bend angle parameters and is independent of all other parameters such as the radius of curvature of the surface, etc. The local z -axis of the entrance coordinates along with the z -axis of the exit coordinates define a plane which is called the element's **bend plane**. For a mirror, the graze angle is a parameter supplied by the user. For a crystal, the Bragg angles are calculated so that the reference trajectory is in the middle of the Darwin curve. Calculation of the Bragg angles for a crystal is given in Section §20.4.1.

13.3.4 Element Misalignment and Origin Shift Transformation

The **Element Body** coordinates are the coordinate system attached to an element. Without any misalignments (Here ‘misalignment’ is *defined* to be any offset, pitch or tilt (§4.4), the laboratory and element body coordinates are the same. With misalignments, the transformation is given by Eqs. (13.4) and (13.5) where

$$\mathbf{L} = (x_offset, y_offset, z_offset) \quad (13.14)$$

and the **S** matrix is defined by

$$\mathbf{S} = \mathbf{W}_\Theta \mathbf{W}_\Phi \mathbf{W}_\Psi \quad (13.15)$$

with

$$\Theta = x_pitch, \quad \Phi = y_pitch, \quad \Psi = tilt \quad (13.16)$$

The form of Eq. (13.15) was chosen to correspond to the form of Eq. (13.2).

For **patch** (§3.34) and **floor_shift** (§3.17) elements, the above equations are also used to calculate the shift in the reference coordinates from the entrance end to the exit end of the element.

For **rbend** and **sbend** elements the above equations are modified by using

$$\Psi = 0 \quad (13.17)$$

This is used since, unlike other elements, the **ref_tilt** attribute of a bend affects the reference orbit (cf. Eq. (13.7)). In the place of a **tilt**, the **roll** attribute can be used (§4.4). For a **roll**, the transformation at the entrance end to go from laboratory coordinates to element coordinates is

$$\mathbf{L} = 0, \quad \mathbf{S} = \mathbf{W}_\Theta(-angle/2) \mathbf{W}_\Psi(roll) \mathbf{W}_\Theta(angle/2) \quad (13.18)$$

where **angle** is the total bend angle. The transformation back to the laboratory frame at the exit end of the bend is

$$\mathbf{L} = 0, \quad \mathbf{S} = \mathbf{W}_\Theta(angle/2) \mathbf{W}_\Psi(-roll) \mathbf{W}_\Theta(-angle/2) \quad (13.19)$$

Notice that these transformations are not inverses of one another.

For **fiducial** and **girder** elements the above equations are used to calculate the alignment of reference coordinates with respect to “origin” coordinates. In this case, the vector **L** is constructed via

$$\mathbf{L} = (dx_origin, dy_origin, dz_origin) \quad (13.20)$$

And the attributes use to shift the reference coordinates orientation are:

$$\Theta = dtheta_origin, \quad \Phi = dphi_origin, \quad \Psi = dpsi_origin \quad (13.21)$$

13.3.5 Reflection Patch

A **Patch** (or a series of patches) that reflects the direction of the **z**-axis is called a **reflection patch**. By “reflected direction” it is meant that the dot product $\mathbf{z}_1 \cdot \mathbf{z}_2$ is negative where \mathbf{z}_1 is the **z**-axis vector at the **entrance** face and \mathbf{z}_2 is the **z**-axis vector at the **exit** face. This condition is equivalent to the condition on **S** in Eq. (13.15) of

$$S(3,3) < 0 \quad (13.22)$$

Using Eq. (13.15) gives after some simple algebra

$$\cos(x_pitch) \cos(y_pitch) < 0 \quad (13.23)$$

When there are a series of patches, The transformations of all the patches are concatenated together to form an effective **S** which can then be used with Eq. (13.22).

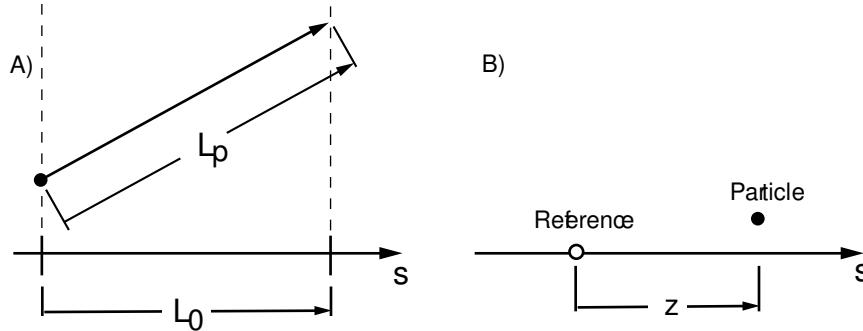


Figure 13.7: Interpreting phase space z at constant velocity: A) The change in z going through an element of length L_0 is $L_0 - L_p$. B) At constant time, z is the longitudinal distance between the reference particle and the particle.

13.4 Phase Space Coordinate System

Bmad uses the phase space coordinates

$$\mathbf{r}(s) = (x, p_x, y, p_y, z, p_z) \quad (13.24)$$

The longitudinal position s is the independent variable instead of the time. x and y , are the transverse coordinates of the particle as shown in Fig. 13.1. Note that x and y are independent of the position of the reference particle.

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

$$\begin{aligned} p_x &= \frac{P_x}{P_0} \\ p_y &= \frac{P_y}{P_0} \end{aligned} \quad (13.25)$$

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

The phase space z coordinate is

$$\begin{aligned} z(s) &= -\beta(s) c (t(s) - t_0(s)) \\ &\equiv -\beta(s) c \Delta t(s) \end{aligned} \quad (13.26)$$

$t(s)$ is the time at which the particle is at position s , $t_0(s)$ is the time at which the reference particle is at position s , and β is v/c with v being the particle velocity (and not the reference velocity). The reference particle is, by definition, “synchronized” with elements whose fields are oscillating and therefore the actual fields a particle will see when traveling through such an element will depend upon the particle’s phase space z . For example, the energy change of a particle traveling through an **1cavity** (§3.25) or **rfcavity** (§3.36) element is z dependent. Exception: With absolute time tracking (§13.7) fields are tied to the absolute time and not z .

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

$$\Delta z = L_0 - L_p \quad (13.27)$$

where, as shown in Fig. 13.7A, L_0 is the path length of the reference particle (which is just the length of the element) and L_p is the path length of the particle in traversing the element. Another way of interpreting phase space z is that, at constant β , and constant time, z is the longitudinal distance between the particle and the reference particle as shown in Fig. 13.7B. with positive z indicating that the particle is ahead of the reference particle.

Do not confuse the phase space z with the z that is the particle's longitudinal coordinate in the local reference frame as shown in Fig. 13.1. By construction, this latter z is always zero.

Notice that if a particle gets an instantaneous longitudinal kick so that β is discontinuous then, from Eq. (13.26), phase space z is discontinuous even though the particle itself does not move in space. In general, from Eq. (13.26), The value of z for a particle at s_2 is related to the value of z for the particle at s_1 by

$$z_2 = \frac{\beta_2}{\beta_1} z_1 - \beta_2 c (\Delta t_2 - \Delta t_1) \quad (13.28)$$

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

The longitudinal phase space momentum p_z is given by

$$p_z = \frac{\Delta P}{P_0} \equiv \frac{P - P_0}{P_0} \quad (13.29)$$

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

$$p_z = \frac{\Delta E}{E_0} \quad (13.30)$$

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 (§3.25) 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 phase space momenta and the slopes $x' \equiv dx/ds$ and $y' \equiv dy/ds$ is

$$x' \approx \frac{p_x}{1 + p_z} (1 + gx) \quad (13.31)$$

$$y' \approx \frac{p_y}{1 + p_z} (1 + gx) \quad (13.32)$$

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

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

$$\frac{dz}{ds} = -g x - \frac{1}{2} (x'^2 + y'^2) \quad (13.33)$$

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

A particle can also have a spin. The spin is characterized by the spinor $\Psi = (\psi_1, \psi_2)^T$ where $\psi_{1,2}$ are complex numbers (§19.4).

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. See Chapter §30 for more details.

13.5 Photon Phase Space Coordinate System

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

$$(x, \beta_x, y, \beta_y, z, \beta_z) \quad (13.34)$$

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

$$\beta_x^2 + \beta_y^2 + \beta_z^2 = 1 \quad (13.35)$$

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

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

$$\begin{aligned} E_x(\mathbf{r}, t) &\sim E_x e^{i(k(z-z_0)-\omega(t-t_{ref})+\phi_x)} \\ E_y(\mathbf{r}, t) &\sim E_y e^{i(k(z-z_0)-\omega(t-t_{ref})+\phi_y)} \end{aligned} \quad (13.36)$$

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

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

13.6 Time-based Phase Space Coordinate System

Some specialized routines use $(x, cp_x, y, cp_y, s, cp_s)_t$ phase space coordinates with time t as the independent variable. The positions x , y , and s are the same as in Fig. 13.1. The momenta are defined as

$$\begin{aligned} cp_x &\equiv mc^2\gamma\beta_x \\ cp_y &\equiv mc^2\gamma\beta_y \\ cp_s &\equiv mc^2\gamma\beta_s, \end{aligned} \quad (13.37)$$

and internally are stored in units of eV.

13.7 Relative Verses Absolute Time Tracking

Unlike other elements, the kick given a particle going through an `lcavity`, `rfcavity`, or possibly an `em_field` element depends upon the time that the particle enters the element relative to some “RF clock”. *Bmad* has two modes for calculating this time called “relative time tracking” and “absolute time tracking”.

With `relative time tracking`, which *Bmad* uses by default, the RF clock is determined by the reference particle. That is, the RF phase, `phi_RF`, seen by a particle is directly related to the particles’s z phase space coordinate (§13.4)

```

phi_RF = phi_particle + phi_ref
phi_particle = -z * rf_frequency / particle_velocity

```

where `phi_ref` is a fixed phase offset (generally set in the lattice file) and independent of the particle coordinates. With `absolute_time_tracking`, the RF phase is simply determined by the absolute time the particle enters the element

```
phi_particle = t_particle * rf_frequency
```

The switch to set the type of tracking for a lattice is `parameter[absolute_time_tracking]` (§8.1).

There are advantages and disadvantages to using either relative or absolute time tracking. Absolute time tracking is more correct since RF cavities are in reality synched to some clock. The problem with absolute time tracking is that the transfer map through the cavity is now a function of time and cannot be written as a function of z unless the total lattice length is a multiple of the RF wavelength. This complicates lattice analysis.

With relative time tracking the transfer map problem is swept under the rug. The penalty for using relative time tracking is that results can be unphysical. For example, the closed orbit is essentially independent of the RF frequency. From a different angle this can be viewed as a good feature since if one is only interested in, say, calculating the Twiss parameters, it can be an annoyance to have to worry that the ring one has constructed have the spacing between RF cavities be multiples of the RF wavelength and it is potentially confusing to see non-zero closed orbits when one is not expecting it.

The above discussion is limited to the cavity fundamental mode. Long-range wake fields, on the other hand, cannot be synchronized to the z coordinate since, in general, their frequencies are not commensurate with the fundamental mode frequency. For simulating the long-range wakes, the kick is thus, by necessity, tied to the absolute time. The exception is that a wake associated with the fundamental mode (that is, has the same frequency as the fundamental mode) will always use relative time if the fundamental is using relative time and vice versa.

Do not confuse absolute time tracking with the `time_runge_kutta` tracking method (§5.1). The `time_runge_kutta` method uses time as the independent variable instead of z . Absolute time tracking just means that the RF phase is dependent upon the time instead of z . It is perfectly possible to use absolute time tracking with code that uses z as the independent variable.

Chapter 14

Electromagnetic Fields

14.1 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

$$B_y(x, 0) = \sum_n B_n \frac{x^n}{n!} \quad (14.1)$$

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 3rd order is

$$B_x = B_1 y + B_2 xy + \frac{1}{6} B_3 (3x^2 y - y^3) + \dots \quad (14.2)$$

$$B_y = B_0 + B_1 x + \frac{1}{2} B_2 (x^2 - y^2) + \frac{1}{6} B_3 (x^3 - 3xy^2) + \dots \quad (14.3)$$

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

$$K_n L \equiv \frac{q L B_n}{P_0} \quad (14.4)$$

$L B_n$ 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 ρ is the bending radius of a particle with the reference energy in a field of strength B . [Note: If q is in Coulombs, P_0 is in kg*m/sec. If q is in units of the elementary charge, P_0 is in eV/c_light.]

Conventionally, *Bmad* always takes the sign of the charge q to be positive. This means that, for example, positive K_1 and positive B_1 are always associated with horizontally focusing quadrupoles for positrons as well as electrons. The only element where the sign of the particle's charge matters is an **elseparator** element (§3.14).

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

$$\Delta p_x = \frac{-qL B_y}{P_0} \quad (14.5)$$

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

$$\Delta p_y = \frac{qL B_x}{P_0} \quad (14.6)$$

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

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

$$\Delta p_x = \sum_{n=0}^{\infty} \frac{K_n L}{n!} \sum_{m=0}^{\lfloor \frac{n}{2} \rfloor} \binom{n}{2m} (-1)^{m+1} x^{n-2m} y^{2m} \quad (14.7)$$

$$\Delta p_y = \sum_{n=0}^{\infty} \frac{K_n L}{n!} \sum_{m=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n}{2m+1} (-1)^m x^{n-2m-1} y^{2m+1} \quad (14.8)$$

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 θ_n , the magnetic field at a point (x, y) can be expressed in complex notation as

$$B_y(x, y) + iB_x(x, y) = \frac{1}{n!} B_n e^{-i(n+1)\theta_n} e^{in\theta} r^n \quad (14.9)$$

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

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

$$\begin{aligned} \tilde{K}_n &= K_n \cos((n+1)\theta_n) \\ \tilde{K}S_n &= K_n \sin((n+1)\theta_n) \end{aligned} \quad (14.10)$$

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

$$\frac{qL}{P_0} (B_y + iB_x) = (b_n + ia_n) (x + iy)^n \quad (14.11)$$

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

$$b_n + ia_n = \frac{1}{n!} K_n L e^{-i(n+1)\theta_n} \quad (14.12)$$

or

$$K_n L = n! \sqrt{a_n^2 + b_n^2} \quad (14.13)$$

$$\tan[(n+1)\theta_n] = \frac{-a_n}{b_n} \quad (14.14)$$

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

$$(n+1)\theta_n = \frac{\pi}{2} \quad (14.15)$$

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

The a_n , b_n representation of multipole fields is in **AB_Multipole** elements (§3.1) as well as in other types of elements such as quadrupoles, sextupoles, etc. This allows error fields to be represented. When a_n and b_n multipole values are associated with an element that is not an **AB_Multipole** element, and if the **scale_multipoles** attribute (§4.12) is not set to **False**, a measurement radius r_0 and a scale factor F are used to scale the effect of the a_n and b_n so that the multipole strength scales as the element strength. The scaling formula is

$$[a_n(\text{actual}), b_n(\text{actual})] = [a_n(\text{input}), b_n(\text{input})] \cdot F \cdot \frac{r_0^{n_{\text{ref}}}}{r_0^n} \quad (14.16)$$

$a_n(\text{input})$ 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_{ref} are set automatically depending upon the type of element as shown in Table 14.1.

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.

Element	F	n_{ref}
Elseparator	$\sqrt{H_{\text{kick}}^2 + V_{\text{kick}}^2}$	0
Hkicker	Kick	0
Kicker	$\sqrt{H_{\text{kick}}^2 + V_{\text{kick}}^2}$	0
Rbend	$G * L$	0
Sbend	$G * L$	0
Vkicker	Kick	0
Wiggler	$\frac{2 c L_{\text{pole}} B_{\text{max}}}{\pi p_0 c}$	0
Quadrupole	$K_1 * L$	1
Sol_Qquad	$K_1 * L$	1
Solenoid	$K_S * L$	1
Sextupole	$K_2 * L$	2
Octupole	$K_3 * L$	3

Table 14.1: F and n_{ref} for various elements.

14.2 RF fields

The RF fields in an RF cavity can be parameterized as the sum over a number of modes. This parameterization used by *Bmad* essentially follows Abell[Abell06]. The electric field is written in the form

$$\mathbf{E}(\mathbf{r}) = \sum_{j=1}^M \mathbf{E}_j(\mathbf{r}) e^{-i(\omega_j t + \theta_{0j})} \quad (14.17)$$

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

$$\nabla^2 \mathbf{E}_j + k_{tj}^2 \mathbf{E}_j = 0 \quad (14.18)$$

where $k_{tj} = \omega_j/c$ with ω_j being the mode frequency.

For a given mode, the electric field can be characterized by the values on a cylindrical surface of some radius R . The field will be of the form

$$\begin{aligned} E_\rho(R, \phi, z) &= E_{\rho c}(R, z) \cos(m\phi - \phi_0) \\ E_\phi(R, \phi, z) &= E_{\phi s}(R, z) \sin(m\phi - \phi_0) \\ E_z(R, \phi, z) &= E_{zc}(R, z) \cos(m\phi - \phi_0) \end{aligned} \quad (14.19)$$

where in this, and subsequent equations, the mode index j has been dropped.

The azimuthal mode number m_j is a non-negative integer. If $m_j = 0$, there are two different modes, one with $\phi_{0j} = 0$ and the other with $\phi_{0j} = -\pi/2$. The $\phi_{0j} = 0$ mode is an accelerating mode with the electric field is in the form

$$\begin{aligned} E_\rho(\mathbf{r}) &= \sum_{n=-N/2+1}^{N/2} -e^{ik_{zn}z} i k_{zn} e_0(n) \tilde{I}_1(\kappa_n, \rho) \\ E_\phi(\mathbf{r}) &= 0 \\ E_z(\mathbf{r}) &= \sum_{n=-N/2+1}^{N/2} e^{ik_{zn}z} e_0(n) \tilde{I}_0(\kappa_n, \rho) \end{aligned} \quad (14.20)$$

where \tilde{I}_m is

$$\tilde{I}_m(\kappa_n, \rho) \equiv \frac{I_m(\kappa_n \rho)}{\kappa_n^m} \quad (14.21)$$

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

$$\kappa_n = \sqrt{k_{zn}^2 - k_t^2} = \begin{cases} \sqrt{k_{zn}^2 - k_t^2} & |k_{zn}| > k_t \\ -i \sqrt{k_t^2 - k_{zn}^2} & k_t > |k_{zn}| \end{cases} \quad (14.22)$$

with

$$k_{zn} = \frac{2\pi n}{N dz} \quad (14.23)$$

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

$$I_m(-i x) = i^{-m} J_m(x) \quad (14.24)$$

where J_m is a Bessel function of the first kind. The e_0 coefficients are obtained from the equation

$$\begin{aligned} e_0(n) &= \frac{1}{\tilde{I}_0(\kappa_n, R)} \frac{1}{N} \sum_{p=0}^{N-1} e^{-2\pi i np/N} E_{zc}(R, p dz) \\ &\equiv \frac{\tilde{E}_{zc}(R, n)}{\tilde{I}_0(\kappa_n, R)} \end{aligned} \quad (14.25)$$

The non-accelerating $m = 0$ mode with $\phi_0 = -\pi/2$ has an electric field in the form

$$\begin{aligned} E_\rho(\mathbf{r}) &= E_z(\mathbf{r}) = 0 \\ E_\phi(\mathbf{r}) &= \sum_{n=-N/2+1}^{N/2} e^{ik_{zn}z} b_0(n) \tilde{I}_1(\kappa_n, \rho) \end{aligned} \quad (14.26)$$

where

$$b_0(n) = \frac{\tilde{E}_{\phi s}(R, n)}{\tilde{I}_1(\kappa_n, R)} \quad (14.27)$$

For positive m , the electric field is in the form

$$\begin{aligned} E_\rho(\mathbf{r}) &= \sum_{n=-N/2+1}^{N/2} -i e^{i k_{zn} z} \left[k_{zn} e_m(n) \tilde{I}_{m+1}(\kappa_n, \rho) + b_m(n) \frac{\tilde{I}_m(\kappa_n, \rho)}{\rho} \right] \cos(m\phi - \phi_0) \\ E_\phi(\mathbf{r}) &= \sum_{n=-N/2+1}^{N/2} -i e^{i k_{zn} z} \left[k_{zn} e_m(n) \tilde{I}_{m+1}(\kappa_n, \rho) + \right. \\ &\quad \left. b_m(n) \left(\frac{\tilde{I}_m(\kappa_n, \rho)}{\rho} - \frac{1}{m} \tilde{I}_{m-1}(\kappa_n, \rho) \right) \right] \sin(m\phi - \phi_0) \\ E_z(\mathbf{r}) &= \sum_{n=-N/2+1}^{N/2} e^{i k_{zn} z} e_m(n) \tilde{I}_m(\kappa_n, \rho) \cos(m\phi - \phi_0) \end{aligned} \quad (14.28)$$

with

$$\begin{aligned} e_m(n) &= \frac{\tilde{E}_{zc}(R, n)}{\tilde{I}_m(\kappa_n, R)} \\ b_m(n) &= \frac{R}{\tilde{I}_m(\kappa_n, R)} \left[i \tilde{E}_{\rho c} - k_{zn} e_m(n) \tilde{I}_{m+1}(\kappa_n, R) \right] \end{aligned} \quad (14.29)$$

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

$$\mathbf{E} = -\partial_t \mathbf{A}, \quad \mathbf{B} = \nabla \times \mathbf{A} \quad (14.30)$$

Using Eq. (14.17), the vector potential can be obtained from the electric field via

$$\mathbf{A}_j = \frac{-i \mathbf{E}_j}{\omega_j} \quad (14.31)$$

Symplectic tracking through the RF field is discussed in Section §19.3. For the fundamental accelerating mode, The vector potential can be analytically integrated using the identity

$$\int dx \frac{x I_1(a \sqrt{x^2 + y^2})}{\sqrt{x^2 + y^2}} = \frac{1}{a} I_0(a \sqrt{x^2 + y^2}) \quad (14.32)$$

14.3 Wake fields

14.3.1 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

$$\Delta p_z(i) = \frac{-e L}{v P_0} \left(\frac{1}{2} W_{\parallel}^{SR}(0) |q_i| + \sum_{j \neq i} W_{\parallel}^{SR}(dz_{ij}) |q_j| \right) \quad (14.33)$$

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.

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

$$\Delta p_x(i) = \frac{-e L \sum_j |q_j| x_j W_{\perp}^{SR}(dz_{ij})}{v P_0} \quad (14.34)$$

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 “pseudo” modes where

$$W(z) = \sum_i A_i e^{d_i z} \sin(k_i z + \phi_i) \quad (14.35)$$

The set of mode parameters (A_i, d_i, k_i, ϕ_i) are chosen to fit the calculated wake potential. The reason why the mode approach is used in *Bmad* is due to the fact that, with modes, the calculation time scales as the number of particles N while a calculation based upon a table of wake vs z would scale as N^2 . [The disadvantage is that initially the user must perform a fit to the wake potential to generate the mode parameter values.]

14.3.2 Long-Range Wakes

Following Chao[Chao93] Eq. 2.88, the long-range wake fields are characterized by a set of cavity modes. The wake function W_m for a mode of order m is given by

$$W_m(t) = -c \left(\frac{R}{Q} \right)_m e^{-\omega t/2Q} \sin(\omega t) \quad (14.36)$$

The mode strength $(R/Q)_m$ has units of Volts/meter 2m .

Assuming that the macroparticle generating the wake is offset a distance r_w along the x -axis, a trailing macroparticle will see a kick

$$\Delta \mathbf{p}_{\perp} = -C I_m W_m(t) m r^{m-1} (\hat{\mathbf{r}} \cos m\theta - \hat{\theta} \sin m\theta) \quad (14.37)$$

$$= -C I_m W_m(t) m r^{m-1} (\hat{\mathbf{x}} \cos[(m-1)\theta] - \hat{\mathbf{y}} \sin[(m-1)\theta])$$

$$\Delta p_z = -C I_m W'_m(t) r^m \cos m\theta \quad (14.38)$$

where m is the order of the mode, C is given by

$$C = \frac{e}{c P_0} \quad (14.39)$$

and

$$I_m = q_w r_w^m \quad (14.40)$$

with q_w being the magnitude of the charge on the particle. Generalizing the above, a macroparticle at (r_w, θ_w) will generate a wake

$$-\Delta p_x + i\Delta p_y = C I_m W_m(t) m r^{m-1} e^{-im\theta_w} e^{i(m-1)\theta} \quad (14.41)$$

$$\Delta p_z = C I_m W'_m(t) r^m \cos[m(\theta - \theta_w)] \quad (14.42)$$

Comparing Eq. (14.41) to (14.9), and using the relationship between kick and field as given by (14.5) and (14.6), 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 time t_w with respect to the reference particle will produce wake components

$$\begin{aligned} \delta a_{\sin,m} &= c \left(\frac{R}{Q} \right)_m e^{\omega t_w/2Q} \cos(\omega t_w) I_m \sin(m\theta_w) \\ \delta a_{\cos,m} &= -c \left(\frac{R}{Q} \right)_m e^{\omega t_w/2Q} \sin(\omega t_w) I_m \sin(m\theta_w) \\ \delta b_{\sin,m} &= c \left(\frac{R}{Q} \right)_m e^{\omega t_w/2Q} \cos(\omega t_w) I_m \cos(m\theta_w) \\ \delta b_{\cos,m} &= -c \left(\frac{R}{Q} \right)_m e^{\omega t_w/2Q} \sin(\omega t_w) I_m \cos(m\theta_w) \end{aligned} \quad (14.43)$$

These are added to the existing wake components. The total is

$$a_{\sin,m} = \sum_{\text{particles}} \delta a_{\sin,m} \quad (14.44)$$

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

$$a_m = e^{-\omega t/2Q} (a_{\cos,m} \cos(\omega t) - a_{\sin,m} \sin(\omega t)) \quad (14.45)$$

$$b_m = e^{-\omega t/2Q} (b_{\cos,m} \cos(\omega t) - b_{\sin,m} \sin(\omega t))$$

Here t is the passage time of the particle with respect to the reference particle. In analogy to Eq. (14.41) and (14.42), the kick is

$$-\Delta p_x + i\Delta p_y = C m (b_m + ia_m) r^{m-1} e^{i(m-1)\theta} \quad (14.46)$$

$$\Delta p_z = -C r^m ((b'_m + ia'_m) e^{im\theta} + (b'_m - ia'_m) e^{-im\theta}) \quad (14.47)$$

where $a' \equiv da/dt$ and $b' \equiv db/dt$.

When simulating trains of bunches, the exponential factor $\omega t_w/2Q$ in Eq. (14.43) can become very large. To prevent numerical overflow, *Bmad* uses a reference time z_{ref} so that all times t in the above equations are replaced by

$$t \longrightarrow t - t_{\text{ref}} \quad (14.48)$$

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 ϕ . Equations (14.45) and (14.46) are unchanged. In place of Eq. (14.43), the contribution of a particle to a mode is

$$\begin{aligned}\delta a_{\sin,m} &= c \left(\frac{R}{Q} \right)_m e^{\omega t_w/2Q} \cos(\omega t_w) I_m [\sin(m\theta_w) \sin^2(m\phi) + \cos(m\theta_w) \sin(m\phi) \cos(m\phi)] \\ \delta a_{\cos,m} &= -c \left(\frac{R}{Q} \right)_m e^{\omega t_w/2Q} \sin(\omega t_w) I_m [\sin(m\theta_w) \sin^2(m\phi) + \cos(m\theta_w) \sin(m\phi) \cos(m\phi)] \\ \delta b_{\sin,m} &= c \left(\frac{R}{Q} \right)_m e^{\omega t_w/2Q} \cos(\omega t_w) I_m [\cos(m\theta_w) \cos^2(m\phi) + \sin(m\theta_w) \sin(m\phi) \cos(m\phi)] \\ \delta b_{\cos,m} &= -c \left(\frac{R}{Q} \right)_m e^{\omega t_w/2Q} \sin(\omega t_w) I_m [\cos(m\theta_w) \cos^2(m\phi) + \sin(m\theta_w) \sin(m\phi) \cos(m\phi)]\end{aligned}\tag{14.49}$$

Each mode is characterized by an R/Q , Q , ω , 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. (14.36) to compensate.

Chapter 15

Multiparticle Simulation

15.1 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.

15.1.1 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, ϕ) , is

$$J = \frac{1}{2}[\gamma x^2 + 2\alpha x x' + \beta x'^2] \quad (15.1)$$

$$\tan \phi = \frac{-\beta(x' + \alpha x)}{x} \quad (15.2)$$

The inverse is

$$\begin{pmatrix} x \\ x' \end{pmatrix} = \sqrt{2J} \begin{pmatrix} \sqrt{\beta} & 0 \\ -\frac{\alpha}{\sqrt{\beta}} & -\frac{1}{\sqrt{\beta}} \end{pmatrix} \begin{pmatrix} \cos \phi \\ \sin \phi \end{pmatrix}. \quad (15.3)$$

In action-angle coordinates, the normalized Gaussian phase space distribution, $\rho(J, \phi)$, is

$$\rho(J, \phi) = \frac{1}{2\pi\varepsilon} e^{-\frac{J}{\varepsilon}}. \quad (15.4)$$

where the emittance ε is just the average of J over the distribution

$$\varepsilon = \langle J \rangle \equiv \int dJ d\phi J \rho(J, \phi). \quad (15.5)$$

The beam sizes σ and σ' are

$$\sigma = \sqrt{\langle x^2 \rangle} = \sqrt{\varepsilon\beta} \quad (15.6)$$

$$\sigma' = \sqrt{\langle x'^2 \rangle} = \sqrt{\varepsilon\gamma}, \quad (15.7)$$

and the covariance is

$$\langle xx' \rangle = -\varepsilon\alpha. \quad (15.8)$$

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

$$B_n = \begin{cases} \frac{\varepsilon}{2} \left(\frac{n_\sigma n}{N} \right)^2 & \text{for } 0 \leq n < N_J \\ \infty & \text{for } n = N_J \end{cases} \quad (15.9)$$

where n_σ is called the “**boundary sigma cutoff**”. Within each region, an elliptical shell of constant J_n is constructed with N_ϕ particles equally spaced in ϕ . 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

$$N_\phi q_n = \int_{B_{n-1}}^{B_n} dJ \int_0^{2\pi} d\phi \rho(J, \phi) = \exp\left(-\frac{B_{n-1}}{\varepsilon}\right) - \exp\left(-\frac{B_n}{\varepsilon}\right) \quad (15.10)$$

The value of J_n is chosen to coincide with the average J within the region

$$N_\phi q_n J_n = \int_{B_{n-1}}^{B_n} dJ \int_0^{2\pi} d\phi J \rho(J, \phi) = \varepsilon(\xi + 1) e^{-\xi} \left| \frac{\frac{B_{n-1}}{\varepsilon}}{\frac{B_n}{\varepsilon}} \right|^{\frac{B_{n-1}}{\varepsilon}} \quad (15.11)$$

The **ellipse** phase space distribution is thus

$$\rho_{model}(J, \phi) = q_{tot} \sum_{n=1}^{N_J} q_n \delta(J - J_n) \sum_{m=1}^{N_\phi} \delta(\phi - 2\pi \frac{m}{N_\phi}) \quad (15.12)$$

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_σ , N_J , N_ϕ , and q_{tot} .

The **ellipse** distribution is two dimensional in nature but can easily be extended to six dimensions.

15.1.2 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

$$\rho(J_x, \phi_x, J_y, \phi_y) = \frac{1}{(2\pi)^2 \varepsilon_x \varepsilon_y} \exp\left(-\frac{J_x}{\varepsilon_x}\right) \exp\left(-\frac{J_y}{\varepsilon_y}\right) \quad (15.13)$$

$$= \frac{1}{(2\pi)^2 \varepsilon_x \varepsilon_y} \exp\left(-\frac{I_1}{\varepsilon}\right), \quad (15.14)$$

where the orthogonal action coordinates are:

$$I_1 = \left(\frac{J_x}{\varepsilon_x} + \frac{J_y}{\varepsilon_y} \right) \varepsilon \quad (15.15)$$

$$I_2 = \left(-\frac{J_x}{\varepsilon_y} + \frac{J_y}{\varepsilon_x} \right) \varepsilon \quad (15.16)$$

with $\varepsilon = (\frac{1}{\varepsilon_x^2} + \frac{1}{\varepsilon_y^2})^{-1/2}$. The reverse transformation is:

$$J_x = \left(\frac{I_1}{\varepsilon_x} - \frac{I_2}{\varepsilon_y} \right) \varepsilon \quad (15.17)$$

$$J_y = \left(\frac{I_1}{\varepsilon_y} + \frac{I_2}{\varepsilon_x} \right) \varepsilon. \quad (15.18)$$

The KV distribution is

$$\rho(I_1, I_2, \phi_x, \phi_y) = \frac{1}{A} \delta(I_1 - \xi), \quad (15.19)$$

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 ξ , 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 $[-\frac{\varepsilon_x}{\varepsilon_y} I_1, \frac{\varepsilon_y}{\varepsilon_x} I_1]$. 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 ρ is only dependent on I_1 ,

$$q = \int_0^\infty dI_1 \int_{I_2}^{I_2 + \Delta I_2} dI_2 \int_{\phi_x}^{\phi_x + \Delta \phi_x} d\phi_x \int_{\phi_y}^{\phi_y + \Delta \phi_y} d\phi_y \frac{1}{A} \delta(I_1 - \xi) \quad (15.20)$$

$$= \frac{1}{A} \Delta I_2 \Delta \phi_x \Delta \phi_y. \quad (15.21)$$

To represent the distribution with particles of equal weight, we must partition (I_2, ϕ_x, ϕ_y) -space into regions of equal volume.

The weight of each particle is

$$q = \frac{1}{NM_x M_y} = \frac{1}{N_{tot}} \quad (15.22)$$

where N_{tot} is the total number of particles

15.2 Macroparticles

Note: The macroparticle tracking code is not currently maintained in favor of tracking an ensemble of particles.

A macroparticle[Brown77] is represented by a centroid position $\bar{\mathbf{r}}$ and a 6×6 σ 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 $\sigma_z = \sqrt{\sigma(5, 5)}$.

σ is a real, non-negative symmetric matrix. The equation that defines the ellipsoid at a distance of n -sigma from the centroid is

$$(\mathbf{r} - \bar{\mathbf{r}})^t \sigma^{-1} (\mathbf{r} - \bar{\mathbf{r}}) = n \quad (15.23)$$

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

$$\sigma_2 = \mathbf{M}_{12} \sigma_1 \mathbf{M}_{12}^t \quad (15.24)$$

where \mathbf{M}_{12} is the Jacobian of the transport map from point s_1 to s_2 .

The Twiss parameters can be calculated from the sigma matrix. The dispersion is given by

$$\begin{aligned}\sigma(1, 6) &= \eta_x \sigma(6, 6) \\ \sigma(2, 6) &= \eta'_x \sigma(6, 6) \\ \sigma(3, 6) &= \eta_y \sigma(6, 6) \\ \sigma(4, 6) &= \eta'_y \sigma(6, 6)\end{aligned}\tag{15.25}$$

Ignoring coupling for now, the betatron part of the sigma matrix can be obtained from the linear equations of motion. For example, using

$$x = \sqrt{2\beta_x \epsilon_x} \cos \phi_x + \eta_x p_z\tag{15.26}$$

Solving for the first term on the RHS, squaring and averaging over all particles gives

$$\beta_x \epsilon_x = \sigma(1, 1) - \frac{\sigma^2(1, 6)}{\sigma(6, 6)}\tag{15.27}$$

It is thus convenient to define the betatron part of the sigma matrix

$$\sigma_\beta(i, j) \equiv \sigma(i, j) - \frac{\sigma(i, 6) \sigma(j, 6)}{\sigma(6, 6)}\tag{15.28}$$

and in terms of the betatron part the emittance is

$$\epsilon_x^2 = \sigma_\beta(1, 1) \sigma_\beta(2, 2) - \sigma_\beta^2(1, 2)\tag{15.29}$$

and the Twiss parameters are

$$\epsilon_x \begin{pmatrix} \beta_x & -\alpha_x \\ -\alpha_x & \gamma_x \end{pmatrix} = \begin{pmatrix} \sigma_\beta(1, 1) & \sigma_\beta(1, 2) \\ \sigma_\beta(1, 2) & \sigma_\beta(2, 2) \end{pmatrix}\tag{15.30}$$

If there is coupling, the transformation between the 4×4 transverse normal mode sigma matrix $\boldsymbol{\sigma}_a$ and the 4×4 laboratory matrix $\boldsymbol{\sigma}_x$ is

$$\boldsymbol{\sigma}_x = \mathbf{V} \boldsymbol{\sigma}_a \mathbf{V}^t\tag{15.31}$$

where \mathbf{V} is given by Eq. (17.3).

The sigma matrix is the same for all macroparticles and is determined by the local Twiss parameters:

$$\begin{aligned}\sigma(1, 1) &= \epsilon_x \beta_x \\ \sigma(1, 2) &= -\epsilon_x \alpha_x \\ \sigma(2, 2) &= \epsilon_x \gamma_x = \epsilon_x (1 + \alpha_x^2) / \beta_x \\ \sigma(3, 3) &= \epsilon_y \beta_y \\ \sigma(3, 4) &= -\epsilon_y \alpha_b \\ \sigma(3, 4) &= \epsilon_y \gamma_y = \epsilon_y (1 + \alpha_b^2) / \beta_y \\ \sigma(i, j) &= 0 \quad \text{otherwise}\end{aligned}\tag{15.32}$$

The centroid energy of the k^{th} macroparticle is

$$E_k = E_b + \frac{(n_{mp} - 2k + 1) \sigma_E N_{\sigma E}}{n_{mp}}\tag{15.33}$$

where E_b is the central energy of the bunch, n_{mp} is the number of macroparticles, σ_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

$$dE_{mp} = \frac{2\sigma_E N_{\sigma E}}{n_{mp}}\tag{15.34}$$

15.3 Touschek Scattering

[Developed by Michael Ehrlichman]

Touschek scattering occurs when a single scattering event between two particles in the same beam transfers transverse momentum to longitudinal momentum, and the resulting change in longitudinal momentum results in the loss of one or both particles. In the case of storage rings, these losses impose a beam lifetime. In low-emittance storage rings, Touschek scattering can be the dominant mechanism for particle loss. In the case of linear accelerators, these losses generate radiation in the accelerator tunnel. When the scattered particles collide with the beam chamber, x-rays are produced which can damage equipment and impose a biohazard. Studies of Touschek scattering typically look at beam lifetime and locations where scattering occurs and where particles are lost.

A commonly utilized theory for studying Touschek scattering is from Piwinski [Piwin98]. A basic outline of the derivation is,

1. Scatter two particles from a bunch in their COM frame using the relativistic Moller cross-section.
2. Boost from COM frame to lab frame. Changes to longitudinal momentum end up amplified by a factor of γ .
3. Integrate over 3D Gaussian distribution of particle positions and angles.

During the derivation many approximations are made which lead to a relatively simple formula. The integration is set up such that only those collisions which will result in particle loss are counted. The formula takes the momentum aperture as a parameter. The resulting formula is reproduced here to give the reader an idea of what influences the scattering rate, and how one might go about evaluating the formula,

$$R = \frac{r_e^2 c \beta_x \beta_y \sigma_h N_p^2}{8\sqrt{\pi} \beta^2 \gamma^4 \sigma_{x\beta}^2 \sigma_{y\beta}^2 \sigma_s \sigma_p} \int_{\tau_m}^{\infty} \left(\left(2 + \frac{1}{\tau} \right)^2 \left(\frac{\tau/\tau_m}{1+\tau} - 1 \right) + 1 - \frac{\sqrt{1+\tau}}{\sqrt{\tau/\tau_m}} \right. \\ \left. - \frac{1}{2\tau} \left(4 + \frac{1}{\tau} \right) \ln \frac{\tau/\tau_m}{1+\tau} \right) \frac{\sqrt{\tau}}{\sqrt{1+\tau}} e^{-B_1 \tau} I_0[B_2 \tau] d\tau, \quad (15.35)$$

where $\tau_m = \beta^2 \delta_m^2$ and δ_m is the momentum aperture. This formula gives the rate at which particles are scattered out of the bunch. It is assumed that two particles are lost per scattering event, one with too much energy and one with too little energy. If a machine with an unsymmetric momentum aperture is being studied, then the formula should be evaluated twice, once for each aperture, and the results averaged. Refer to [Piwin98] for definitions of the parameters involved. This formula is implemented in BMAD as part of the `touschek_mod` module.

Different formulas for calculating the Touschek scattering rate exist elsewhere in the literature. For example, Wiedemann [Wiede99], presents a formula with a simpler integrand. This formula, originally from a paper by LeDuff [Duff87], is derived in a fashion similar to Piwinski except that the formula does not take dispersion into account and uses a non-relativistic scattering cross-section. Since Piwinski's formula is the most robust, it is the one used in *Bmad*.

Particles are lost from Touschek scattering due to two effects. In storage rings, there is a momentum aperture defined by the RF system that is often referred to as the RF bucket. If the δp imparted by a Touschek scattering event exceeds this RF bucket, then the particle will no longer undergo synchrotron oscillations with the rest of the bunch and will coast through the accelerator. Second, if the Touschek scattering event occurs in a dispersive region, the scattered particles will take on a finite J and undergo

betatron oscillations. These oscillations can be large in amplitude and may cause the particles to collide with the beam pipe. To first order, the amplitude of J due to a scattering event that imparts a momentum deviation of Δp is,

$$J \approx \gamma_0 \mathcal{H}_0 \frac{\Delta p^2}{2}, \quad (15.36)$$

where γ_0 is relativistic γ and \mathcal{H}_0 is the dispersion invariant.

Chapter 16

Synchrotron Radiation

16.1 Synchrotron Radiation Damping and Excitation

Emission of synchrotron radiation by a particle can be decomposed into two parts. The deterministic average energy emitted produces damping while the stochastic fluctuating part of the energy spectrum produces excitation[Jowett87].

The treatment of radiation damping by *Bmad* essentially follows *MAD*. The average change in energy ΔE of a particle going through a section of magnet due to synchrotron radiation is

$$\frac{\Delta E}{E_0} = -k_d (1 + p_z) \quad (16.1)$$

where

$$k_d \equiv \frac{2 r_e}{3} \gamma_0^3 \langle g_0^2 \rangle L_p (1 + p_z) \quad (16.2)$$

r_e is the classical electron radius, L_p is the actual path length, γ_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

$$\frac{\Delta E}{E_0} = -k_f (1 + p_z) \quad (16.3)$$

where

$$k_f \equiv \left(\frac{55 r_e \hbar c}{24 \sqrt{3} m_e} L_p \gamma_0^5 \langle g_0^3 \rangle \right)^{1/2} (1 + p_z) \xi \quad (16.4)$$

ξ is a Gaussian distributed random number with unit sigma and zero mean.

Using Eqs. (16.2) and (16.4) the total change in p_z can be written as

$$\Delta p_z = \frac{\Delta E}{E_0} = -k_E (1 + p_z) \quad (16.5)$$

where

$$k_E = k_d + k_f \quad (16.6)$$

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

$$\begin{aligned}\Delta p_x &= -k_E p_x \\ \Delta p_y &= -k_E p_y\end{aligned}\tag{16.7}$$

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.

16.2 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[[Helm73](#), [Jowett87](#)]. With coupling, the equations need to be generalized and this is detailed below.

In the general case, the curvature vector $\mathbf{g} = (g_x, g_y)$, which points away from the center of curvature of the particle's orbit and has a magnitude of $|\mathbf{g}| = 1/\rho$, where ρ is the radius of curvature (see Fig. 13.1), does not lie in the horizontal plane. Similarly, the dispersion $\boldsymbol{\eta} = (\eta_x, \eta_y)$ will not lie in the horizontal plane. With this notation, the synchrotron integrals for coupled motion are:

$$I_0 = \oint ds \gamma_0 g \tag{16.8}$$

$$I_1 = \oint ds \mathbf{g} \cdot \boldsymbol{\eta} \equiv \oint ds (g_x \eta_x + g_y \eta_y) \tag{16.9}$$

$$I_2 = \oint ds g^2 \tag{16.10}$$

$$I_3 = \oint ds g^3 \tag{16.11}$$

$$I_{4a} = \oint ds [g^2 \mathbf{g} \cdot \boldsymbol{\eta}_a + \nabla g^2 \cdot \boldsymbol{\eta}_a] \tag{16.12}$$

$$I_{4b} = \oint ds [g^2 \mathbf{g} \cdot \boldsymbol{\eta}_b + \nabla g^2 \cdot \boldsymbol{\eta}_b] \tag{16.13}$$

$$I_{4z} = \oint ds [g^2 \mathbf{g} \cdot \boldsymbol{\eta} + \nabla g^2 \cdot \boldsymbol{\eta}] \tag{16.14}$$

$$I_{5a} = \oint ds g^3 \mathcal{H}_a \tag{16.15}$$

$$I_{5b} = \oint ds g^3 \mathcal{H}_b \tag{16.16}$$

$$I_{6b} = \oint ds g^3 \beta_b \tag{16.17}$$

where γ_0 is that usual relativistic factor and \mathcal{H}_a is

$$\mathcal{H}_a = \gamma_a \eta_a^2 + 2 \alpha_a \eta_a \eta'_a + \beta_a \eta'^2_a \tag{16.18}$$

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:

$$\begin{aligned} g_x(x, y) &= g_x + x k_1 + y s_1 \\ g_y(x, y) &= g_y + x s_1 - y k_1 \end{aligned} \quad (16.19)$$

where k_1 is the quadrupole moment and s_1 is the skew-quadrupole moment. Using this gives on-axis ($x = y = 0$)

$$\nabla g^2 = 2(g_x k_1 + g_y s_1, g_x s_1 - g_y k_1) \quad (16.20)$$

I_0 is not a standard radiation integral. It is useful, though, in calculating the average number of photons emitted. For electrons:

$$\mathcal{N} = \frac{5 r_f}{2\sqrt{3} \hbar c} I_0 \quad (16.21)$$

where \mathcal{N} is the average number of photons emitted by a particle over one turn, and the “classical radius factor” r_f is

$$r_f = \frac{e^2}{4\pi\epsilon_0} \quad (16.22)$$

r_f has a value of $1.4399644 \cdot 10^{-9}$ meters-eV for all particles of charge ± 1 .

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 δ 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. (16.20) and integrating over the edge field gives the contribution to I_4 from a non-zero e_1 as

$$I_{4z} = -\tan(e_1) g^2 (\cos(\theta) \eta_x + \sin(\theta) \eta_y) \quad (16.23)$$

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. (16.23) θ 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 α_p via

$$I_1 = \alpha_p L \quad (16.24)$$

where L is the storage ring circumference. The energy loss per turn is

$$U_0 = \frac{2 r_e E_0^4}{3 (mc^2)^3} I_2 \quad (16.25)$$

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

$$J_a = 1 - \frac{I_{4a}}{I_2}, \quad J_b = 1 - \frac{I_{4b}}{I_2}, \text{ and } J_z = 2 + \frac{I_{4z}}{I_2}. \quad (16.26)$$

Since

$$\boldsymbol{\eta}_a + \boldsymbol{\eta}_b = \boldsymbol{\eta}, \quad (16.27)$$

Robinson’s theorem, $J_a + J_b + J_z = 4$, is satisfied. Alternatively, the exponential damping coefficients per turn are

$$\alpha_a = \frac{U_0 J_a}{2E_0}, \quad \alpha_b = \frac{U_0 J_b}{2E_0}, \text{ and } \alpha_z = \frac{U_0 J_z}{2E_0}. \quad (16.28)$$

The energy spread is given by

$$\sigma_{pz}^2 = \left(\frac{\sigma_E}{E_0} \right)^2 = C_q \gamma_0^2 \frac{I_3}{2I_2 + I_{4z}} \quad (16.29)$$

where γ_0 is the usual energy factor and

$$C_q = \frac{55}{32\sqrt{3}} \frac{\hbar}{mc} = 3.84 \times 10^{-13} \text{ meter for electrons} \quad (16.30)$$

If the synchrotron frequency is not too large, the bunch length is given by

$$\sigma_z = \frac{I_1}{M(6,5)} \sigma_{pz} \quad (16.31)$$

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

$$\begin{aligned} \epsilon_a &= \frac{C_q}{I_2 - I_{4a}} \gamma_0^2 I_{5a} \\ \epsilon_b &= \frac{C_q}{I_2 - I_{4b}} \left(\gamma_0^2 I_{5b} + \frac{13}{55} I_{6b} \right) \end{aligned} \quad (16.32)$$

The I_{6b} term come from the finite vertical opening angle of the radiation[Rauben91]. Normally this term is very small compared to the emittance due to coupling or vertical kicks due to magnet misalignment.

For a non-circular machine, radiation integrals are still of interest if there are bends or steering elements. However, in this case, the appropriate energy factors must be included to take account any changes in energy due to any **lcavity** elements. For a non-circular machine, the I_1 integral is not altered and the I_4 integrals are not relevant. The other integrals become

$$L_2 = \int ds g^2 \gamma_0^4 \quad (16.33)$$

$$L_3 = \int ds g^3 \gamma_0^7 \quad (16.34)$$

$$L_{5a} = \int ds g^3 \mathcal{H}_a \gamma_0^6 \quad (16.35)$$

$$L_{5b} = \int ds g^3 \mathcal{H}_b \gamma_0^6 \quad (16.36)$$

In terms of these integrals, the energy loss through the lattice is

$$U_0 = \frac{2r_e mc^2}{3} L_2 \quad (16.37)$$

The energy spread assuming σ_E is zero at the start and neglecting any damping is

$$\sigma_E^2 = \frac{4}{3} C_q r_e (mc^2)^2 L_3 \quad (16.38)$$

The above equation is appropriate for a linac. In a storage ring, where there are energy oscillations, the growth of σ_E^2 due to quantum excitation is half that. One way to explain this is that in a storage ring, the longitudinal motion is “shared” between the z and pz coordinates and, to preserve phase space volume, this reduces σ_E^2 by a factor of 2.

Again neglecting any initial beam width, the transverse beam size at the end of the lattice is

$$\begin{aligned}\epsilon_a &= \frac{2}{3} C_q r_e \frac{L_{5a}}{\gamma_f} \\ \epsilon_b &= \frac{2}{3} C_q r_e \frac{L_{5b}}{\gamma_f}\end{aligned}\quad (16.39)$$

Where γ_f is the final gamma.

16.3 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 [[Sagan06](#)]. This formalism divides the total kick received by a particle due to another particle into two parts: One

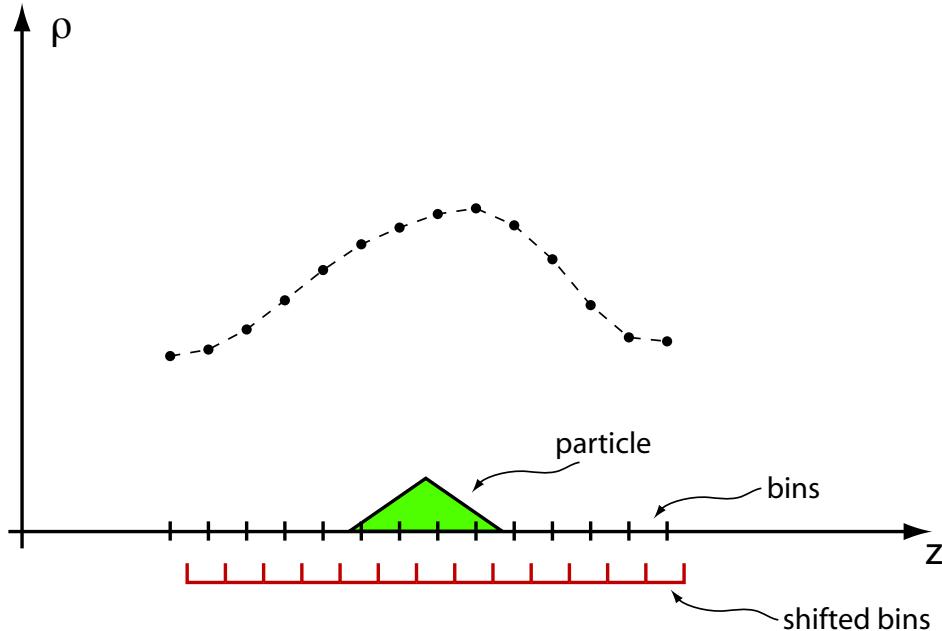


Figure 16.1: The Coherent Synchrotron Radiation kick is calculated by dividing longitudinally a bunch into a number of bins. To smooth the computed densities, each particle of the bunch is considered to have a triangular density distribution.

part 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 Fig. 16.1. 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 §10.3.

Chapter 17

Linear Optics

17.1 Coupling and Normal Modes

The coupling formalism used by *Bmad* is taken from the paper of Sagan and Rubin[Sagan99]. 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

$$\mathbf{T} = \mathbf{V} \mathbf{U} \mathbf{V}^{-1}, \quad (17.1)$$

where \mathbf{V} is symplectic, and \mathbf{U} is of the form

$$\mathbf{U} = \begin{pmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{B} \end{pmatrix}. \quad (17.2)$$

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

$$\mathbf{V} = \begin{pmatrix} \gamma \mathbf{I} & \mathbf{C} \\ -\mathbf{C}^+ & \gamma \mathbf{I} \end{pmatrix}, \quad (17.3)$$

where \mathbf{C} is a 2x2 matrix and $+$ superscript denotes the symplectic conjugate:

$$\mathbf{C}^+ = \begin{pmatrix} C_{22} & -C_{12} \\ -C_{21} & C_{11} \end{pmatrix}. \quad (17.4)$$

Since we demand that \mathbf{V} be symplectic we have the condition

$$\gamma^2 + \|\mathbf{C}\| = 1, \quad (17.5)$$

and \mathbf{V}^{-1} is given by

$$\mathbf{V}^{-1} = \begin{pmatrix} \gamma \mathbf{I} & -\mathbf{C} \\ \mathbf{C}^+ & \gamma \mathbf{I} \end{pmatrix}. \quad (17.6)$$

\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

$$\bar{\mathbf{U}} = \mathbf{G} \mathbf{U} \mathbf{G}^{-1}, \quad (17.7)$$

Where \mathbf{G} is given by

$$\mathbf{G} \equiv \begin{pmatrix} \mathbf{G}_a & \mathbf{0} \\ \mathbf{0} & \mathbf{G}_b \end{pmatrix}, \quad (17.8)$$

with

$$\mathbf{G}_a = \begin{pmatrix} \frac{1}{\sqrt{\beta_a}} & 0 \\ \frac{\alpha_a}{\sqrt{\beta_a}} & \sqrt{\beta_a} \end{pmatrix}, \quad (17.9)$$

with a similar equation for \mathbf{G}_b . With this definition, the corresponding $\bar{\mathbf{A}}$ and $\bar{\mathbf{B}}$ (cf. Eq. (17.2)) are just rotation matrices. The relationship between \mathbf{T} and $\bar{\mathbf{U}}$ is

$$\mathbf{T} = \mathbf{G}^{-1} \bar{\mathbf{V}} \bar{\mathbf{U}} \bar{\mathbf{V}}^{-1} \mathbf{G}, \quad (17.10)$$

where

$$\bar{\mathbf{V}} = \mathbf{G} \mathbf{V} \mathbf{G}^{-1}. \quad (17.11)$$

Using Eq. (17.8), $\bar{\mathbf{V}}$ can be written in the form

$$\bar{\mathbf{V}} = \begin{pmatrix} \gamma \mathbf{I} & \bar{\mathbf{C}} \\ -\bar{\mathbf{C}}^+ & \gamma \mathbf{I} \end{pmatrix}, \quad (17.12)$$

with the normalized matrix $\bar{\mathbf{C}}$ given by

$$\bar{\mathbf{C}} = \mathbf{G}_a \mathbf{C} \mathbf{G}_b^{-1}. \quad (17.13)$$

The normal mode coordinates $\mathbf{a} = (a, a', b, b')$ are related to the laboratory frame via

$$\mathbf{a} = \mathbf{V}^{-1} \mathbf{x}. \quad (17.14)$$

In particular the normal mode dispersion $\boldsymbol{\eta}_a = (\eta_a, \eta'_a, \eta_b, \eta'_b)$ is related to the laboratory frame dispersion $\boldsymbol{\eta}_x = (\eta_x, \eta'_x, \eta_y, \eta'_y)$ via

$$\boldsymbol{\eta}_a = \mathbf{V}^{-1} \boldsymbol{\eta}_x. \quad (17.15)$$

When there is no coupling ($\mathbf{C} = 0$), $\boldsymbol{\eta}_a$ and $\boldsymbol{\eta}_x$ are equal to each other.

17.2 Dispersion Calculation

The dispersion (η) and the dispersion derivative (η') are defined by the equations

$$\begin{aligned} \eta_x(s) &\equiv \left. \frac{dx}{dp_z} \right|_s, & \eta'_x(s) &\equiv \left. \frac{d\eta_x}{ds} \right|_s = \left. \frac{dx'}{dp_z} \right|_s \\ \eta_y(s) &\equiv \left. \frac{dy}{dp_z} \right|_s, & \eta'_y(s) &\equiv \left. \frac{d\eta_y}{ds} \right|_s = \left. \frac{dy'}{dp_z} \right|_s \\ \eta_z(s) &\equiv \left. \frac{dz}{dp_z} \right|_s \end{aligned} \quad (17.16)$$

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:

$$\mathbf{r}_2 = \mathbf{M} \mathbf{r}_1 + \mathbf{V} \quad (17.17)$$

Define the dispersion vector $\boldsymbol{\eta}$ by

$$\boldsymbol{\eta} = (\eta_x, \eta'_x(1+p_z), \eta_y, \eta'_y(1+p_z), \eta_z, 1) \quad (17.18)$$

Differentiating Eq. (17.17) with respect to energy, the dispersion at point 2 in terms of the dispersion at point 1 is

$$\boldsymbol{\eta}_2 = \left[\frac{dp_{z2}}{dp_{z1}} \right]^{-1} [\mathbf{M} \boldsymbol{\eta}_1] + \mathbf{V}_\eta \quad (17.19)$$

where

$$\mathbf{V}_\eta = \left[\frac{dp_{z2}}{dp_{z1}} \right]^{-1} \frac{1}{1+p_{z1}} \begin{pmatrix} M_{12} p_{x1} + M_{14} p_{y1} \\ M_{22} p_{x1} + M_{24} p_{y1} \\ M_{32} p_{x1} + M_{34} p_{y1} \\ M_{42} p_{x1} + M_{44} p_{y1} \\ M_{52} p_{x1} + M_{54} p_{y1} \\ M_{62} p_{x1} + M_{64} p_{y1} \end{pmatrix} - \begin{pmatrix} 0 \\ \frac{p_{x2}}{1+p_{z2}} \\ 0 \\ \frac{p_{y2}}{1+p_{z2}} \\ 0 \\ 0 \end{pmatrix} \quad (17.20)$$

The sixth row of the matrix equation gives dp_{z1}/dp_{z2} . Explicitly

$$\frac{dp_{z2}}{dp_{z1}} = \sum_{i=1}^6 M_{6i} \eta_{1i} + \frac{M_{62} p_{x1} + M_{64} p_{y1}}{1+p_{z1}} \quad (17.21)$$

For everything except RFcavity and Lcavity elements, dp_{z2}/dp_{z1} is 1.

For a non-circular machine, there are two ways one can imagine defining the dispersion: Either with respect to changes in energy at the beginning of the machine or with respect to the local change in energy at the point of measurement. The former definition will be called “non-local dispersion” and the latter definition will be called “local dispersion”. For a circular machine, local dispersion is always used. The dispersion defined in the above equations, which is what Bmad uses in calculations, is the local dispersion. The non-local dispersion $\tilde{\boldsymbol{\eta}}(s_1)$ at some point s_1 is related to the local dispersion $\boldsymbol{\eta}(s_1)$ via

$$\tilde{\boldsymbol{\eta}}(s_1) = \frac{dp_{z1}}{dp_{z0}} \boldsymbol{\eta}(s_1) \quad (17.22)$$

where s_0 is the beginning of the machine.

For a non-circular machine, there are advantages and disadvantages to using either local or non-local dispersion. Local dispersion has the problem that dp_{z2}/dp_{z1} in Eq. (17.19) may go through zero at a point producing infinite dispersions at that point. The non-local dispersion has the merit of reflecting what one would measure if the starting energy of the beam is varied. The local dispersion, on the other hand, reflects the correlations between the particle energy and particle position within a beam.

Chapter 18

Taylor Maps

18.1 Taylor Maps

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 $\mathbf{r}(\text{out})$

$$\mathbf{r}(\text{out}) = \mathcal{M} \mathbf{r}(\text{in}) \quad (18.1)$$

\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

$$r_i(\text{out}) = \sum_{j=1}^N C_{ij} \prod_{k=1}^6 r_k^{e_{ijk}}(\text{in}) \quad (18.2)$$

Where the C_{ij} are coefficients and the e_{ijk} are integer exponents. The order of the map is

$$\text{order} = \max_{i,j} \left(\sum_{k=1}^6 e_{ijk} \right) \quad (18.3)$$

The standard *Bmad* routine for printing a Taylor map might produce something like this:

Taylor Terms:									
Out	Coef	Exponents						Order	Reference
<hr/>									
1:	-0.60000000000000	0	0	0	0	0	0	0	0.2000000000
1:	1.00000000000000	1	0	0	0	0	0	1	
1:	0.14500000000000	2	0	0	0	0	0	2	
<hr/>									
2:	-0.18500000000000	0	0	0	0	0	0	0	0.0000000000
2:	1.30000000000000	0	1	0	0	0	0	1	
2:	3.80000000000000	2	0	0	0	0	1	3	
<hr/>									
3:	1.00000000000000	0	0	1	0	0	0	1	0.1000000000
3:	1.60000000000000	0	0	0	1	0	0	1	
3:	-11.138187077310	1	0	1	0	0	0	2	
<hr/>									

4:	1.00000000000000	0	0	0	1	0	0	1	0.0000000000
<hr/>									
5:	0.00000000000000	0	0	0	0	0	0	0	0.0000000000
5:	0.000001480008	0	1	0	0	0	0	1	
5:	1.00000000000000	0	0	0	0	1	0	1	
5:	0.00000000000003	0	0	0	0	0	1	1	
5:	0.00000000000003	2	0	0	0	0	0	2	
<hr/>									
6:	1.00000000000000	0	0	0	0	0	1	1	0.0000000000

Each line in the example represents a single **Taylor term**. The Taylor terms are grouped into 6 **Taylor series**. There is one series for each of the output phase space coordinate. The first column in the example, labeled “out”, (corresponding to the i index in Eq. (18.2)) indicates the Taylor series: 1 = $x(out)$, 2 = $p_x(out)$, etc. The 6 exponent columns give the e_{ijk} of Eq. (18.2). In this example, the second Taylor series ($\text{out} = 2$), when expressed as a formula, would read:

$$p_x(\text{out}) = -0.185 + 1.3 p_x(\text{in}) + 3.8 x^2(\text{in}) p_z(\text{in}) \quad (18.4)$$

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

$$(x, p_x, y, p_y, z, p_z)_{\text{ref}} = (0.2, 0, 0.1, 0, 0, 0, 0) \quad (18.5)$$

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

$$x(\text{out}) = A \sin(k x(\text{in})) \quad (18.6)$$

Then a Taylor map to 1st order is

$$x(\text{out}) = c_0 + c_1 x(\text{in}) \quad (18.7)$$

where

$$c_1 = A k \cos(k x_{\text{ref}}) \quad (18.8)$$

$$c_0 = A \sin(k x_{\text{ref}}) - c_1 x_{\text{ref}}$$

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.

18.2 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[Healy86] 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

$$\mathbf{V} = \mathbf{S}(\mathbf{I} - \mathbf{M})(\mathbf{I} + \mathbf{M})^{-1} \quad (18.9)$$

where \mathbf{S} is the matrix

$$\mathbf{S} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix} \quad (18.10)$$

\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

$$\mathbf{W} = \frac{\mathbf{V} + \mathbf{V}^t}{2} \quad (18.11)$$

A symplectic matrix \mathbf{F} is now obtained by inverting (18.9)

$$\mathbf{F} = (\mathbf{I} + \mathbf{SW})(\mathbf{I} - \mathbf{SW})^{-1} \quad (18.12)$$

18.3 Map Concatenation and Feed-Down

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

$$M_1 : x_1 = x_0 + 2 \quad (18.13)$$

and the map M_2 from point 1 to point 2 to be

$$M_2 : x_2 = x_1^2 + 3x_1 \quad (18.14)$$

Then concatenating the maps to form the map M_3 from point 0 to point 2 gives

$$M_3 : x_2 = (x_0 + 2)^2 + 3(x_0 + 2) = x_0^2 + 7x_0 + 10 \quad (18.15)$$

However if we are evaluating our maps to only 1st order the map M_2 becomes

$$M_2 : x_2 = 3x_1 \quad (18.16)$$

and concatenating the maps now gives

$$M_3 : x_2 = 3(x_0 + 2) = 3x_0 + 6 \quad (18.17)$$

Comparing this to Eq. (18.15) shows that by neglecting the 2nd order term in Eq. (18.14) leads to 0th and 1st order errors in Eq. (18.17). These errors can be traced to the finite 0th order term in Eq. (18.13). 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

$$M_3 = M_2(M_1) \quad (18.18)$$

Then if M_1 and M_2 are correct to nth order, M_3 will also be correct to nth order as long as M_1 has no constant (0th 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

$$\tilde{x}_0 = x_0 + 2 \quad (18.19)$$

18.4 Symplectic Integration

Symplectic integration, as opposed to concatenation, 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[Forest98]. 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

$$M_1 : x_1 = c_i \quad (18.20)$$

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

$$M_3 : x_2 = c_f \quad (18.21)$$

If the map from point 1 to point 2 is desired then the map M_1 is just set to the identity map

$$M_1 : x_1 = x_0 \quad (18.22)$$

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:

$$\int_{y_a}^{y_b} f(y) dy = h \left[\frac{1}{2} f(y_a) + \frac{1}{2} f(y_b) \right] + o(h^3 f^{(2)}) \quad (18.23)$$

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[Press92]. 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 (§5.4). 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 (§5.4). Timing tests performed for some wiggler elements (which have strong nonlinearities) showed that, in this case, the 2nd 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.

Chapter 19

Tracking of Charged Particles

Bmad can track both charged particles and X-rays. This chapter deals with charged particles and X-rays are handled in chapter §20.

For tracking and transfer map calculations (here generically called “tracking”), *Bmad* has various methods that can be applied to a given element (Cf. Chapter §5). This chapter discusses the `bmad_standard` calculation that is the default for almost all element types and the `symp_lie_bmad` calculation that does symplectic integration.

Generally, it will be assumed that tracking is in the forward direction.

19.1 Element Coordinate System

The general procedure for tracking through an element makes use of `element reference` coordinates (also called just `element` coordinates). Without any offsets, pitches or tilt (§4.4), henceforth called “misalignments”, the `element` coordinates are the same as the `laboratory reference` coordinates (or simply `laboratory` coordinates) (§13.1). The `element` coordinates stay fixed relative to the element. Therefore, if the element is misaligned, the `element coordinates` will follow as the element shifts in the laboratory frame as shown in Fig. 19.1.

Tracking a particle through an element is a three step process:

1. At the entrance end of the element, transform from the `laboratory` coordinates to the entrance `element` coordinates.
2. Track through the element ignoring any misalignments.
3. At the exit end of the element, transform from the exit `element reference` frame to the `laboratory` reference frame.

The transformation between `laboratory` and `element` reference frames will depend upon whether the element is straight or not. In any case, it is assumed that pitches are small so that second order terms can be ignored.

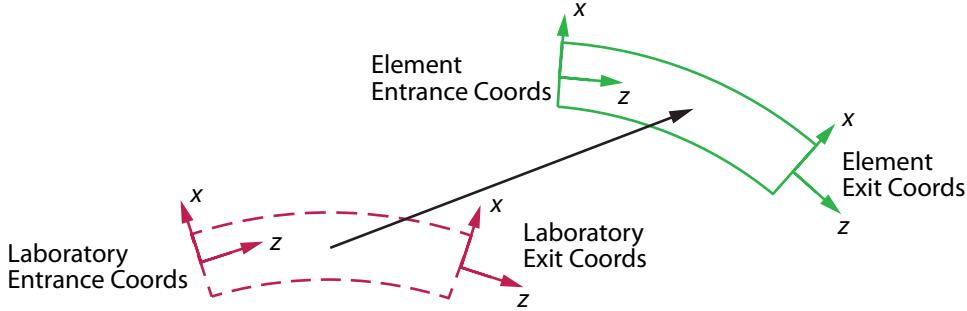


Figure 19.1: Element coordinates are coordinates attached to the physical element (solid green outline). The laboratory coordinates are fixed at the nominal position of the element (red dashed outline).

19.1.1 Transform from Laboratory Entrance to Element Coordinates for Straight Elements

The transformation from the laboratory coordinates to element coordinates for an element that has a straight reference trajectory through it is a two step process.

1. Track as in a drift a distance `z_offset_tot`.
2. Apply offsets and pitches

$$\begin{aligned}
 x_1 &= x_0 - x_{\text{off}} + \frac{L}{2} x'_{\text{pitch}} \\
 p_{x1} &= p_{x0} - (1 + p_{z0}) x'_{\text{pitch}} \\
 y_1 &= y_0 - y_{\text{off}} + \frac{L}{2} y'_{\text{pitch}} \\
 p_{y1} &= p_{y0} - (1 + p_{z0}) y'_{\text{pitch}} \\
 z_1 &= z_0 + x'_{\text{pitch}} x_1 + y'_{\text{pitch}} y_1 - \frac{L}{4} (x'^2_{\text{pitch}} + y'^2_{\text{pitch}})
 \end{aligned} \tag{19.1}$$

where x'_{pitch} and y'_{pitch} are the `x_pitch_tot` and `y_pitch_tot` attributes of the element (§4.4), and x_{off} , and y_{off} are the `x_offset_tot` and `y_offset_tot` attributes. Notice that z_1 is written in terms of x_1 and y_1

3. Apply the tilt θ_t (`tilt_tot`)

$$\begin{aligned}
 x_2 &= x_1 \cos \theta_t + y_1 \sin \theta_t \\
 p_{x2} &= p_{x1} \cos \theta_t + p_{y1} \sin \theta_t \\
 y_2 &= -x_1 \sin \theta_t + y_1 \cos \theta_t \\
 p_{y2} &= -p_{x1} \sin \theta_t + p_{y1} \cos \theta_t
 \end{aligned} \tag{19.2}$$

19.1.2 Transform from Laboratory Entrance to Element Coordinates for Bend Elements

Let \mathbf{r}_c be the displacement of the bend at the bend center (where offsets are specified (§4.4)).

The rotation matrix \mathbf{S}_c transforming from the center of the bend (where the misalignments are defined) to the entrance face of the bend is

$$\mathbf{S}_c = \mathbf{W}_{\Psi}^{-1}(\theta_t) \mathbf{W}_{\Theta}(\alpha_b/2) \mathbf{W}_{\Psi}(\theta_t) \quad (19.3)$$

$$(19.4)$$

where \mathbf{W}_{Ψ} and \mathbf{W}_{Θ} are given by Eqs. (13.3), θ_t is the `ref_tilt` attribute of the bend and α_b is the bend `angle`.

In the laboratory entrance coordinates, the pitches and offset are

$$\mathbf{W} = \mathbf{S} \begin{pmatrix} -y'_{pitch} \\ x'_{pitch} \\ 0 \end{pmatrix} \quad (19.5)$$

$$\mathbf{V} = \mathbf{S} \begin{pmatrix} x_{off} \\ y_{off} \\ z_{off} \end{pmatrix} + \mathbf{S} \mathbf{L}_c - \mathbf{L}_c \quad (19.6)$$

where the offset vector \mathbf{L}_c is

$$\mathbf{L}_c = \mathbf{W}_{\Psi}(\theta_t) \begin{pmatrix} 2\rho \sin^2(\alpha_b/2) \\ 0 \\ -\rho \sin(\alpha_b/2) \end{pmatrix} \quad (19.7)$$

The transformation to element coordinates is then

1.

$$\begin{aligned} x_1 &= x_0 - \mathbf{V}(1) \\ p_{x1} &= p_{x0} - \mathbf{W}(1)(1 + p_{z0}) \\ y_1 &= y_0 - \mathbf{V}(2) \\ p_{y1} &= p_{x0} + \mathbf{W}(2)(1 + p_{z0}) \\ z_1 &= z_0 + \mathbf{W}(2)x_0 - \mathbf{W}(1)y_0 \end{aligned} \quad (19.8)$$

2. Track as in a drift a distance $\mathbf{V}(3)$.

3. Apply the non-reference shifting tilt θ_t (`tilt_tot`)

$$\begin{aligned} x_2 &= x_1 \cos \theta_t + y_1 \sin \theta_t \\ p_{x2} &= p_{x1} \cos \theta_t + p_{y1} \sin \theta_t \\ y_2 &= -x_1 \sin \theta_t + y_1 \cos \theta_t \\ p_{y2} &= -p_{x1} \sin \theta_t + p_{y1} \cos \theta_t \end{aligned} \quad (19.9)$$

19.1.3 Transform from Element Exit to Laboratory Coordinate for Straight Elements

The back transformation from element to laboratory coordinates is accomplished by the transformation

1. Apply tilt θ_t

$$\begin{aligned} x_1 &= x_0 \cos \theta_t - y_0 \sin \theta_t \\ p_{x1} &= p_{x0} \cos \theta_t - p_{y0} \sin \theta_t \\ y_1 &= x_0 \sin \theta_t + y_0 \cos \theta_t \\ p_{y1} &= p_{x0} \sin \theta_t + p_{y0} \cos \theta_t \end{aligned} \quad (19.10)$$

2. Apply offsets and pitches

$$\begin{aligned}
 x_2 &= x_1 + x_{\text{off}} + \frac{L}{2} x'_{\text{pitch}} \\
 p_{x2} &= p_{x1} + (1 + p_{z1}) x'_{\text{pitch}} \\
 y_2 &= y_1 + y_{\text{off}} + \frac{L}{2} y'_{\text{pitch}} \\
 p_{y2} &= p_{y1} + (1 + p_{z1}) y'_{\text{pitch}} \\
 z_2 &= z_1 + x'_{\text{pitch}} x_1 + y'_{\text{pitch}} y_1 - \frac{L}{4} (x'^2_{\text{pitch}} + y'^2_{\text{pitch}})
 \end{aligned} \tag{19.11}$$

3. Track as in a drift a distance `-z_offset_tot`.

19.1.4 Transform from Element Exit to Laboratory Coordinate for Bend Elements

The back transformation from element to laboratory coordinates is accomplished by the transformation

1. Apply non-reference shifting tilt θ_t

$$\begin{aligned}
 x_1 &= x_0 \cos \theta_t - y_0 \sin \theta_t \\
 p_{x1} &= p_{x0} \cos \theta_t - p_{y0} \sin \theta_t \\
 y_1 &= x_0 \sin \theta_t + y_0 \cos \theta_t \\
 p_{y1} &= p_{x0} \sin \theta_t + p_{y0} \cos \theta_t
 \end{aligned} \tag{19.12}$$

2. Apply offsets and pitches

$$\begin{aligned}
 x_1 &= x_0 + \mathbf{V}(1) \\
 p_{x1} &= p_{x0} + \mathbf{W}(1) (1 + p_{z0}) \\
 y_1 &= y_0 + \mathbf{V}(2) \\
 p_{y1} &= p_{x0} - \mathbf{W}(2) (1 + p_{z0}) \\
 z_1 &= z_0 - \mathbf{W}(2) x_0 + \mathbf{W}(1) y_0
 \end{aligned} \tag{19.13}$$

3. Track as in a drift a distance `-z_offset_tot`.

19.2 Hamiltonian

The time dependent Hamiltonian H_t in the curvilinear coordinate system shown in Fig. 13.1 is

$$H_t = \tilde{\psi} + \left[\left(\frac{p_s - a_s}{1 + g x} \right)^2 + \tilde{m}^2 + (p_x - a_x)^2 + (p_y - a_y)^2 + \right]^{1/2} \tag{19.14}$$

where $(p_x, p_y, p_s / (1 + gx))$ are the momentum normalized by P_0 , ρ being the local radius of curvature of the reference particle, and \tilde{m} , \mathbf{a} and $\tilde{\psi}$ are the normalized mass, vector, and scalar potentials:

$$\tilde{m} = \frac{m c^2}{c P_0} \quad \left(a_x, a_y, \frac{a_s}{1 + g x} \right) = \frac{q \mathbf{A}}{P_0 c} \quad \tilde{\psi}(x, y, z) = \frac{q \psi}{P_0 c} \tag{19.15}$$

In terms of the normalized velocities β_x , β_y , the canonical momentum are

$$p_x = \frac{mc^2}{P_0 c} \beta_x + a_x, \quad p_y = \frac{mc^2}{P_0 c} \beta_y + a_y \quad (19.16)$$

The s -dependent Hamiltonian is obtained from H_t by solving for $-p_s$. For particles propagating in the positive s direction the s -dependent Hamiltonian is

$$H_{s,E} = -(1+g x) \sqrt{\tilde{E}^2 - \tilde{m}^2 - (p_x - a_x)^2 - (p_y - a_y)^2} - a_s \quad (19.17)$$

where $\tilde{E} = E/c P_0$ is the normalized energy and the equation has been simplified by assuming that $\tilde{\psi}$ is zero. Using a contact transformation to convert to *Bmad* coordinates (§13.4) gives

$$H \equiv H_s = -(1+g x) \sqrt{(1+p_z)^2 - (p_x - a_x)^2 - (p_y - a_y)^2} - a_s + \frac{1}{\beta_0} \sqrt{(1+p_z)^2 + \tilde{m}^2} \quad (19.18)$$

where β_0 is the reference velocity. The last term on the RHS of Eq. (19.18) accounts for the fact that the *Bmad* canonical z (Eq. (13.26)) has an “extra” term $\beta c t_0$ so that *Bmad* canonical z is with respect to the reference particle’s z .

The equations of motion are

$$\frac{dq_i}{ds} = \frac{\partial H}{\partial p_i} \quad \frac{dp_i}{ds} = -\frac{\partial H}{\partial q_i} \quad (19.19)$$

For backwards propagation, where p_s is negative, solving for p_s involves using a different part of the square root branch. the Hamiltonian H_{-s} is then

$$H_{-s} = (1+g x) \sqrt{(1+p_z)^2 - (p_x - a_x)^2 - (p_y - a_y)^2} - a_s - \frac{1}{\beta_0} \sqrt{(1+p_z)^2 + \tilde{m}^2} \quad (19.20)$$

Without an electric field, ψ is zero. Assuming a non-curved coordinate system ($g = 0$), and using the paraxial approximation (§13.4), Eq. (19.18) becomes

$$H = \frac{(p_x - a_x)^2}{2(1+p_z)} + \frac{(p_y - a_y)^2}{2(1+p_z)} - (1+g x) a_s + \frac{1}{\beta_0} \sqrt{(1+p_z)^2 + \tilde{m}^2} \quad (19.21)$$

Once the transverse trajectory has been calculated, the longitudinal position z_2 at the exit end of an element is obtained from symplectic integration of Eq. (19.21)

$$z_2 = z_1 - \frac{1}{2(1+p_{z1})^2} \int ds [(p_x - a_x)^2 + (p_y - a_y)^2] - \int ds g x \quad (19.22)$$

where z_1 is the longitudinal position at the entrance end of the element. Using the equations of motion Eqs. (19.19) this can also be rewritten as

$$z_2 = z_1 - \frac{1}{2} \int ds \left[\left(\frac{dx}{ds} \right)^2 + \left(\frac{dy}{ds} \right)^2 \right] - \int ds g x \quad (19.23)$$

For some elements, `bmad_standard` uses a truncated Taylor map for tracking. For elements without electric fields where the particle energy is a constant, the transfer map for a given coordinate r_i may be expanded in a Taylor series

$$r_{i,2} \rightarrow m_i + \sum_{j=1}^4 m_{ij} r_{j,1} + \sum_{j=1}^4 \sum_{k=j}^4 m_{ijk} r_{j,1} r_{k,1} + \dots \quad (19.24)$$

where the map coefficients $m_{ij\dots}$ are functions of p_z . For linear elements, the transfer map is linear for the transverse coordinates and quadratic for $r_i = z$.

Assuming mid-plane symmetry of the magnetic field, so that a_x and a_y can be set to zero[Iselin94], The vector potential up to second order is (cf. Eq. (14.1))

$$a_s = -k_0 \left(x - \frac{g x^2}{2(1+g)x} \right) - \frac{1}{2} k_1 (x^2 - y^2) \quad (19.25)$$

19.3 Symplectic Integration

Using Eq. (19.21) the Hamiltonian is written in the form

$$H = H_x + H_y + H_z \quad (19.26)$$

where

$$H_x = \frac{(p_x - a_x)^2}{2(1+\delta)}, \quad H_y = \frac{(p_y - a_y)^2}{2(1+\delta)}, \quad H_s = -a_s \quad (19.27)$$

For tracking, the element is broken up into a number of slices set by the element's `ds_step` attribute. For each slice, the tracking uses a quadratic symplectic integrator I :

$$I = T_{s/2} I_{x/2} I_{y/2} I_s I_{y/2} I_{x/2} T_{s/2} \quad (19.28)$$

$T_{s/2}$ is just a translation of the s variable:

$$s \rightarrow s + \frac{ds}{2} \quad (19.29)$$

And the other integrator components are

$$\begin{aligned} I_{x/2} &= \exp \left(: -\frac{ds}{2} H_x : \right) \\ I_{y/2} &= \exp \left(: -\frac{ds}{2} H_y : \right) \\ I_s &= \exp \left(: -ds H_s : \right) \end{aligned} \quad (19.30)$$

The evaluation of $I_{x/2}$ and $I_{y/2}$ is tricky since it involves both transverse position and momentum variables. The trick is to split the integration into three parts. For $I_{x/2}$ this is

$$\begin{aligned} I_{x/2} &= \exp \left(: -\frac{ds}{2} \frac{(p_x - A_x)^2}{2(1+\delta)} : \right) \\ &= \exp \left(: - \int A_x dx : \right) \exp \left(: -\frac{ds}{2} \frac{p_x^2}{2(1+\delta)} : \right) \exp \left(: \int A_x dx : \right) \end{aligned} \quad (19.31)$$

With an analogous expression for $I_{y/2}$.

For magnetic elements that do not have longitudinal fields (quadrupoles, sextupoles, etc.), a_x and a_y can be taken to be zero (cf. Eq. (19.25)). For wigglers (§3.43), Bmad computes \mathbf{A} using

$$\begin{aligned} A_x(x, y, s) &= 0 \\ A_y(x, y, s) &= \int_0^x d\tilde{x} B_z(\tilde{x}, y, s) \\ A_s(x, y, s) &= - \int_0^y d\tilde{x} B_y(\tilde{x}, y, s) \end{aligned} \quad (19.32)$$

given the form of the magnetic field of Eqs. (3.18), (3.19), and (3.20), the integrals in Eq. (19.32) and (19.31) are easily calculated.

For `lcaavity` and `rfcavity` elements, the vector potential is computed from Eq. (14.31).

19.4 Spin Dynamics

[Spin dynamics developed by Jeff Smith]

The classical spin vector \mathbf{S} is described in the local reference frame (§13.1) by a modified Thomas-Bargmann-Michel-Telegdi (T-BMT) equation[Hoff06]

$$\frac{d}{ds} \mathbf{S} = \left\{ \frac{(1 + \mathbf{r}_t \cdot \mathbf{g}) P}{v P_s} \boldsymbol{\Omega}_{BMT}(\mathbf{r}, \mathbf{P}, s) - \mathbf{g} \times \hat{\mathbf{z}} \right\} \times \mathbf{S} \quad (19.33)$$

where \mathbf{g} is the curvature function which points away from the center of curvature of the particle's orbit (see Fig. 13.1), $\mathbf{r}_t = (x, y)$ are the transverse coordinates, P is the momentum, P_s is the longitudinal momentum, v is the velocity, $\hat{\mathbf{z}}$ is the unit vector in the z -direction, and

$$\boldsymbol{\Omega}_{BMT}(\mathbf{r}, \mathbf{P}, t) = -\frac{q}{m\gamma} \left[(1 + G\gamma) \mathbf{B} - \frac{G\mathbf{P} \cdot \mathbf{B}}{(\gamma + 1)m^2 c^2} \mathbf{P} - \frac{1}{mc^2} \left(G + \frac{1}{1 + \gamma} \right) \mathbf{P} \times \mathbf{E} \right], \quad (19.34)$$

Here $\mathbf{E}(\mathbf{r}, t)$ and $\mathbf{B}(\mathbf{r}, t)$ are the electric and magnetic fields, γ is the particle's relativistic gamma factor, 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.

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. The conversion between SU(2) and SO(3) is

$$\mathbf{S} = \Psi^\dagger \boldsymbol{\sigma} \Psi \quad \longleftrightarrow \quad \Psi = \frac{1}{\sqrt{2(s_3 + 1)}} \begin{pmatrix} 1 + s_3 \\ s_1 + is_2 \end{pmatrix} \quad (19.35)$$

where $\boldsymbol{\sigma} = (\sigma_x, \sigma_y, \sigma_z)$ are the three Pauli matrices. In polar coordinates

$$\Psi = \begin{pmatrix} \psi_1 \\ \psi_2 \end{pmatrix} = e^{i\xi} \begin{pmatrix} \cos \frac{\theta}{2} \\ e^{i\phi} \sin \frac{\theta}{2} \end{pmatrix} \quad \longleftrightarrow \quad \mathbf{S} = \begin{pmatrix} \sin \theta \cos \phi \\ \sin \theta \sin \phi \\ \cos \theta \end{pmatrix} \quad (19.36)$$

The ξ being an extra phase factor. Due to the unitarity of the spin vector, $|\psi_1|^2 + |\psi_2|^2 = 1$. The spinor eigenvectors along the x , y and z axes are

$$\begin{aligned} \Psi_{x+} &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}, & \Psi_{x-} &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \\ \Psi_{y+} &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ i \end{pmatrix}, & \Psi_{y-} &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -i \end{pmatrix}, \\ \Psi_{z+} &= \begin{pmatrix} 1 \\ 0 \end{pmatrix}, & \Psi_{z-} &= \begin{pmatrix} 0 \\ -1 \end{pmatrix}. \end{aligned} \quad (19.37)$$

In spinor notation, the T-BMT equation can be written as

$$\frac{d}{dt} \Psi = -\frac{i}{2} (\boldsymbol{\sigma} \cdot \boldsymbol{\Omega}) \Psi. \quad (19.38)$$

The solution leads to a rotation of the spin vector by an angle α around a unit vector \mathbf{e} represented as

$$\begin{aligned}\Psi_f &= e^{-i\frac{\alpha}{2}\mathbf{e}\cdot\boldsymbol{\sigma}}\Psi_i \\ &= (a_0\mathbf{1}_2 - i\mathbf{a}\cdot\boldsymbol{\sigma})\Psi_i \\ &= \mathbf{A}\Psi_i.\end{aligned}\tag{19.39}$$

where Ψ_i is the initial spin state, Φ_f is the final spin state, and the SU(2) matrix \mathbf{A} , called a *quaternion*, describes the spin transport. \mathbf{A} has the normalization condition $a_0^2 + \mathbf{a}^2 = 1$. Thus the three components $\mathbf{a} = (a_1, a_2, a_3)$ completely describe \mathbf{A} .

With spinors, the matrix representation of the observable $S_{\mathbf{u}}$ corresponding to the measurement of the spin along the unit vector \mathbf{u} is

$$S_{\mathbf{u}} \equiv \frac{\hbar}{2} \boldsymbol{\sigma} \cdot \mathbf{u} \tag{19.40}$$

$$= \frac{\hbar}{2} \begin{pmatrix} u_z & u_x - i u_y \\ u_x + i u_y & u_z \end{pmatrix} \tag{19.41}$$

The expectation value of this operator, $\Psi^\dagger \mathbf{S}_{\mathbf{u}} \Psi$, representing the spin of a particle, satisfies the equation of motion of a classical spin vector in the particle's instantaneous rest frame.

For a distribution of spins, the polarization P_s 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 $\frac{2}{\hbar}$,

$$P_s = \frac{2}{\hbar} \frac{1}{N} \sum_{j=1}^N \Psi_j^\dagger S_{\mathbf{u}} \Psi_j \tag{19.42}$$

19.5 BeamBeam Tracking

A beam-beam element (§3.2) simulates the effect on a tracked particle of an opposing beam of particles moving in the opposite direction. The opposing beam, called the “strong” beam, is assumed to be Gaussian in shape.

The strong beam is divided up into `n_slice` equal charge (not equal thickness) slices. 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 complex error function formula [Talman87]. 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:

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

19.6 Bend Element: Fringe Tracking

The transformation for the entrance face of an `sbend` is

$$\begin{aligned} p_{x2} &= p_{x1} + k_x x_1 \\ p_{y2} &= p_{y1} + k_y y_1 \end{aligned} \quad (19.43)$$

where

$$\begin{aligned} k_x &= g_{tot} \tan(e_1) \\ k_y &= -g_{tot} \tan \left[e_1 - 2 |g_{tot}| f_{int} h_{gap} \frac{1 + \sin(e_1)^2}{\cos(e_1)} \right] \end{aligned} \quad (19.44)$$

where g_{tot} is the total bending strength (design + error). Similar equations are used for tracking the exit edge of the bend.

19.7 Bend Element: Body Tracking

The Hamiltonian for the body of an `sbend` is

$$H = (k_0 - g)x - g x p_z + \frac{1}{2} ((k_1 + g k_0)x^2 - k_1 y^2) + \frac{p_x^2 + p_y^2}{2(1 + p_z)} \quad (19.45)$$

This is simply solved

$$\begin{aligned} x_2 &= c_x (x - x_c) + s_x \frac{p_{x1}}{1 + p_{z1}} + x_c \\ p_{x2} &= \tau_x \omega_x^2 (1 + p_{z1}) s_x (x - x_c) + c_x p_{x1} \\ y_2 &= c_y y_1 + s_y \frac{p_{y1}}{1 + p_{z1}} \\ p_{y2} &= \tau_y \omega_y^2 (1 + p_{z1}) s_y y_1 + c_y p_{y1} \\ z_2 &= z_1 + m_5 + m_{51}(x - x_c) + m_{52}p_{x1} + m_{511}(x - x_c)^2 + \\ &\quad m_{512}(x - x_c)p_{x1} + m_{522}p_{x1}^2 + m_{533}y^2 + m_{534}y_1 p_{y1} + m_{544}p_{y1}^2 \\ p_{z2} &= p_{z1} \end{aligned} \quad (19.46)$$

where

$$\begin{aligned} k_x &= k_1 + g k_0 & \omega_x &\equiv \sqrt{\frac{|k_x|}{1 + p_{z1}}} \\ x_c &= \frac{g(1 + p_{z1}) - k_0}{k_x} & \omega_y &\equiv \sqrt{\frac{|k_1|}{1 + p_{z1}}} \end{aligned} \quad (19.47)$$

and

$$\begin{array}{llll} k_x > 0 & k_x < 0 & k_1 > 0 & k_1 < 0 \\ c_x = \cos(\omega_x L) & \cosh(\omega_x L) & c_y = \cosh(\omega_y L) & \cos(\omega_y L) \\ s_x = \frac{\sin(\omega_x L)}{\omega_x} & \frac{\sinh(\omega_x L)}{\omega_x} & s_y = \frac{\sinh(\omega_y L)}{\omega_y} & \frac{\sin(\omega_y L)}{\omega_y} \\ \tau_x = -1 & +1 & \tau_y = +1 & -1 \end{array} \quad (19.48)$$

and

$$\begin{aligned}
 m_5 &= -g x_c L \\
 m_{51} &= -g s_x & m_{52} &= \frac{\tau_x g}{1 + p_{z1}} \frac{1 - c_x}{\omega_x^2} \\
 m_{511} &= \frac{\tau_x \omega_x^2}{4} (L - c_x s_x) & m_{533} &= \frac{\tau_y \omega_y^2}{4} (L - c_y s_y) \\
 m_{512} &= \frac{-\tau_x \omega_x^2}{2(1 + p_{z1})} s_x^2 & m_{534} &= \frac{-\tau_y \omega_y^2}{2(1 + p_{z1})} s_y^2 \\
 m_{522} &= \frac{-1}{4(1 + p_{z1})^2} (L + c_x s_x) & m_{544} &= \frac{-1}{4(1 + p_{z1})^2} (L + c_y s_y)
 \end{aligned}$$

19.8 Drift Tracking

Eq. (19.18) for a drift has $\mathbf{a} = 0$ and $g = 0$. The Hamiltonian for a drift is then

$$H = \frac{p_x^2 + p_y^2}{2(1 + p_z)} \quad (19.49)$$

This gives the map

$$\begin{aligned}
 x_2 &= x_1 + \frac{L p_{x1}}{1 + p_{z1}} \\
 p_{x2} &= p_{x1} \\
 y_2 &= y_1 + \frac{L p_{y1}}{1 + p_{z1}} \\
 p_{y2} &= p_{y1} \\
 z_2 &= z_1 - \frac{L(p_{x1}^2 + p_{y1}^2)}{2(1 + p_{z1})^2} \\
 p_{z2} &= p_{z1}
 \end{aligned} \quad (19.50)$$

19.9 ElSeparator Tracking

[Thanks to Etienne Forest for the derivation of the elseparator equation of motion.]

The Hamiltonian for an electric separator is

$$H = -p_s = - \left\{ \left(\frac{1}{\beta_0} + \delta + k_E x \right)^2 - \tilde{m}^2 - p_x^2 - p_y^2 \right\}^{1/2} \quad (19.51)$$

Here the canonical coordinates $(-ct, \delta)$ are being used, \tilde{m} is defined in Eq. (19.15), and $p_s = -H$ is just the longitudinal momentum. In the above equation, k_E is the normalized field

$$k_E = \frac{q E}{P_0 c} \quad (19.52)$$

The field is taken to be pointing along the x -axis with positive k_E accelerating a particle in the positive x direction. To solve the equations of motion, a “hard edge” model is used where k_E is constant inside the separator and the field ends abruptly at the separator edges.

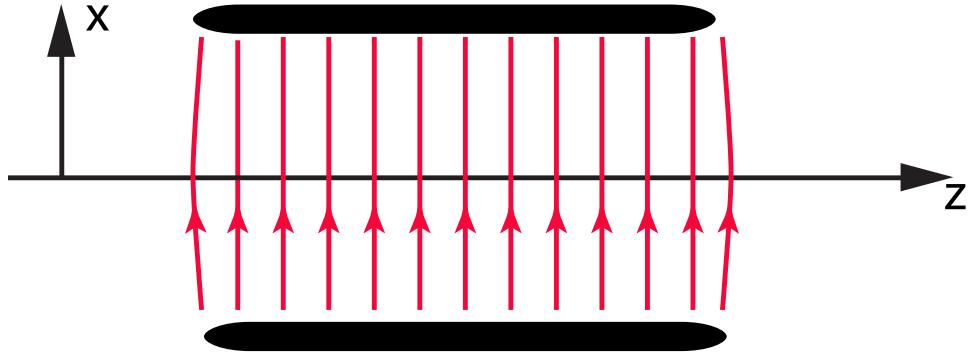


Figure 19.2: Elseparator Electric field. The fringe field lines break the translational invariance in x .

Since, as shown in Fig. 19.2, the fringe fields break the translational invariance in x , it is important here that the $x = 0$ plane be centered within the separator plates. With this, the canonical momentum δ just outside the separator assumes its free space form of $\delta = (E - E_0)/E_0$. This is analogous to the case of a solenoid where, to ensure that the canonical transverse momenta assume their free space form just outside the solenoid, the z -axis must be along the centerline of the solenoid.

The solution of the equations of motion is:

$$\begin{aligned} x &= (x_0 - x_c) \cosh\left(\frac{k_E L}{p_s}\right) + \frac{p_{x0}}{k_E} \sinh\left(\frac{k_E L}{p_s}\right) + x_c \\ p_x &= k_E (x_0 - x_c) \sinh\left(\frac{k_E L}{p_s}\right) + p_{x0} \cosh\left(\frac{k_E L}{p_s}\right) \\ y &= y_0 + L \frac{p_{y0}}{p_s} \\ p_y &= p_{y0} \\ c \delta t &= \int_0^L -\frac{\partial H}{\partial \delta} = (x_0 - x_c) \sinh\left(\frac{k_E L}{p_s}\right) + \frac{p_{x0}}{k_E} \left[\cosh\left(\frac{k_E L}{p_s}\right) - 1 \right] \end{aligned} \quad (19.53)$$

where the critical position x_c is

$$x_c = -\frac{\widetilde{E}}{k_E} \quad (19.54)$$

and

$$\widetilde{E} \equiv \frac{1}{\beta_0} + \delta = \frac{E}{P_0 c} \quad (19.55)$$

Eqs. (19.53) predict that for $x < x_c$ and $p_{x0} = 0$ a particle will, unphysically, accelerate in the negative x direction. In actuality, a particle in this instance will be reflected backwards by the longitudinal component of the edge field. Specifically, the argument of the square root in Eq. (19.51) must be non-negative and a particle will only make it through the separator if

$$x_0 > \frac{1}{k_E} \left(\sqrt{\widetilde{m}^2 + p_{x0}^2 + p_{y0}^2} - \widetilde{E} \right) \quad (19.56)$$

19.10 Kicker, Hkicker, and Vkicker, Tracking

The Hamiltonian for a horizontally deflecting kicker or separator is

$$H = \frac{p_x^2 + p_y^2}{2(1 + p_z)} - k_0 x \quad (19.57)$$

This gives the map

$$\begin{aligned} x_2 &= x_1 + \frac{1}{1 + p_{z1}} \left(L p_{x1} + \frac{1}{2} k_0 L^2 \right) \\ p_{x2} &= p_{x1} + k_0 L \\ y_2 &= y_1 + \frac{L p_{y1}}{1 + p_{z1}} \\ p_{y2} &= p_{y1} \\ z_2 &= z_1 - \frac{L}{2(1 + p_{z1})^2} \left(p_{x1}^2 + p_{y1}^2 + p_{x1} k_0 L + \frac{1}{3} k_0^2 L^2 \right) \\ p_{z2} &= p_{z1} \end{aligned} \quad (19.58)$$

The generalization when the kick is not in the horizontal plane is easily derived.

19.11 Lcavity Tracking

The transverse trajectory through an Lcavity is modeled using equations developed by Rosenzweig and Serafini[Rosen94] (R&S) with

$$\begin{aligned} b_0 &= 1 \\ b_{-1} &= 1 \end{aligned} \quad (19.59)$$

and all other b_n set to zero.

The transport equations in R&S were developed in the ultra-relativistic limit with $\beta = 1$. To extend these equations, the transport through the cavity body (R&S Eq. (9)) has been modified to give the correct phase-space area at non ultra-relativistic energies:

$$\begin{pmatrix} x \\ x' \end{pmatrix}_2 = \begin{pmatrix} \cos(\alpha) & \sqrt{\frac{8}{\eta(\Delta\phi)}} \frac{\beta_1 \gamma_1}{\gamma'} \cos(\Delta\phi) \sin(\alpha) \\ -\sqrt{\frac{\eta(\Delta\phi)}{8}} \frac{\gamma'}{\beta_2 \gamma_2 \cos(\Delta\phi)} \sin(\alpha) & \frac{\beta_1 \gamma_1}{\beta_2 \gamma_2} \cos(\alpha) \end{pmatrix} \begin{pmatrix} x \\ x' \end{pmatrix}_1 \quad (19.60)$$

The added factors of β give the matrix the correct determinate of $\beta_1 \gamma_1 / \beta_2 \gamma_2$. While the added factors of β do correct the phase space area, the above equation can only be considered as a rough approximation for simulating particles when β is significantly different from 1. Indeed, the only accurate way to simulate such particles is by integrating through the actual field [Cf. Runge Kutta tracking (§5.1)]

The change in z going through a cavity is calculated by first calculating the particle transit time Δt

$$\begin{aligned} c \Delta t &= \int_{s_1}^{s_2} ds \frac{1}{\beta(s)} \\ &= \int_{s_1}^{s_2} ds \frac{E}{\sqrt{E^2 - (mc^2)^2}} \\ &= \frac{c P_{z2} - c P_{z1}}{G} \end{aligned} \quad (19.61)$$

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_{z1} and P_{z2} are the entrance and exit momenta. Using Eq. (13.26), the change in z is thus

$$z_2 = \frac{\beta_2}{\beta_1} z_1 - \beta_2 \left(\frac{c P_{z2} - c P_{z1}}{G} - \frac{c \bar{P}_{z2} - c \bar{P}_{z1}}{\bar{G}} \right) \quad (19.62)$$

where \bar{P} and \bar{G} are the momentum and gradient of the reference particle.

Note that the above transport equations are only symplectic on-axis. There are second order terms in the transverse coordinates that are missing. To obtain a proper symplectic matrix, the `symplectify` attribute of an `lcavity` element (§5.5) can be set to True.

19.12 Mirror Tracking

19.13 Octupole Tracking

The Hamiltonian for an upright octupole is

$$H = \frac{p_x^2 + p_y^2}{2(1 + p_z)} + \frac{k_3}{24}(x^4 - 6x^2y^2 + y^4) \quad (19.63)$$

An octupole is modeled using a kick-drift-kick model.

19.14 Patch Tracking

The transformation of the reference coordinates through a “standard” patch (a patch where custom fields are not used) is given by Eqs. (13.4) and (13.5). At the entrance end of the patch, a particle’s position and momentum in the entrance coordinate system will be

$$\begin{aligned} \mathbf{r} &= (x, y, 0) \\ \mathbf{P} &= (P_x, P_y, P_z) = (p_x, p_y, \pm \sqrt{(1 + p_z)^2 - p_x^2 - p_y^2}) P_{0\text{ent}} \end{aligned} \quad (19.64)$$

where p_x , p_y and p_z are the phase space momenta, and z , which is coordinate z and not phase space z , is always zero by construction as shown in Fig. 19.3 [Also see Fig. 13.1 and the discussion in §13.4.] The

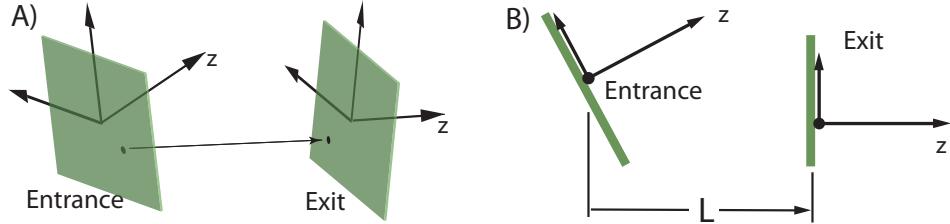


Figure 19.3: Standard tracking through a patch element. A particle’s starting coordinate at the entrance end of the patch has, by construction, coordinate $z = 0$. The particle is drifted, as in a field free region, between the entrance $z = 0$ plane and the exit $z = 0$ plane.

sign of the longitudinal momentum P_z is determined by whether the particle is traveling in the positive s or negative s direction (which will occur when an element is flipped longitudinally).

The transformation between entrance and exit coordinate systems is given by Eqs. (13.10) and (13.11)

$$\begin{aligned}\mathbf{r} &\rightarrow \mathbf{S}^{-1}(\mathbf{r} - \mathbf{L}) \\ \mathbf{P} &\rightarrow \mathbf{S}^{-1}\mathbf{P}\end{aligned}\tag{19.65}$$

where \mathbf{L} is given by Eq. (13.14)

$$\mathbf{L} = (\text{x_offset}, \text{y_offset}, \text{z_offset})$$

After this transformation, the particle must be propagated by a longitudinal length $-r_z$ to intersect the $r_z = 0$ plane of the exit face.

$$\begin{aligned}\mathbf{r} &\rightarrow (r_x - r_z \frac{P_x}{P_z}, r_y - r_z \frac{P_y}{P_z}, 0) \\ \mathbf{P} &\rightarrow \mathbf{P}\end{aligned}\tag{19.66}$$

The final \mathbf{r} and \mathbf{P} can now be used compute the particles phase space coordinates, along with the time t and the reference time t_{ref} at the exit end.

$$\begin{aligned}x &\rightarrow r_x & p_x &\rightarrow \frac{P_x}{P_{0\text{exi}}} \\ y &\rightarrow r_y & p_y &\rightarrow \frac{P_y}{P_{0\text{exi}}} \\ z &\rightarrow z + r_z \frac{|\mathbf{P}|}{P_z} + L_0 \frac{\beta}{\beta_0} + \beta t_{\text{offset}} & p_z &\rightarrow \frac{(1 + p_z) P_{0\text{ent}} - P_{0\text{exi}}}{P_{0\text{exi}}} \\ t &\rightarrow t - r_z \frac{|\mathbf{P}|}{P_z \beta} & t_{\text{ref}} &\rightarrow t_{\text{ref}} + t_{\text{offset}} + L_0 \frac{1}{\beta_0}\end{aligned}\tag{19.67}$$

where the exit reference momentum $P_{0\text{exi}}$ is related to the entrance reference momentum $P_{0\text{ent}}$ through `e_tot_offset`. In the above equation, β is the particle velocity, β_0 is the velocity of the reference particle, and L_0 is the drift length of the reference particle

$$L_0 = \frac{1}{S_{33}^{-1}} (S_{31}^{-1} \text{x_offset} + S_{32}^{-1} \text{y_offset} + S_{33}^{-1} \text{z_offset})\tag{19.68}$$

19.15 Quadrupole Tracking

The `bmad_standard` calculates the transfer map through an upright quadrupole and then transforms that map to the laboratory frame.

The Hamiltonian for an upright quadrupole is

$$H = \frac{p_x^2 + p_y^2}{2(1 + p_z)} + \frac{k_1}{2}(x^2 - y^2)\tag{19.69}$$

This is simply solved

$$\begin{aligned}
 x_2 &= c_x x_1 + s_x \frac{p_{x1}}{1 + p_{z1}} \\
 p_{x2} &= \tau_x \omega^2 (1 + p_{z1}) s_x x_1 + c_x p_{x1} \\
 y_2 &= c_y y_1 + s_y \frac{p_{y1}}{1 + p_{z1}} \\
 p_{y2} &= \tau_y \omega^2 (1 + p_{z1}) s_y y_1 + c_y p_{y1} \\
 z_2 &= z_1 + m_{511} x_1^2 + m_{512} x_1 p_{x1} + m_{522} p_{x1}^2 + m_{533} y_1^2 + m_{534} y_1 p_{y1} + m_{544} p_{y1}^2 \\
 p_{z2} &= p_{z1}
 \end{aligned} \tag{19.70}$$

where

$$\omega \equiv \sqrt{\frac{|k_1|}{1 + p_{z1}}} \tag{19.71}$$

and

$$\begin{array}{llll}
 k_1 > 0 & k_1 < 0 & k_1 > 0 & k_1 < 0 \\
 c_x = \cos(\omega L) & \cosh(\omega L) & c_y = \cosh(\omega L) & \cos(\omega L) \\
 s_x = \frac{\sin(\omega L)}{\omega} & \frac{\sinh(\omega L)}{\omega} & s_y = \frac{\sinh(\omega L)}{\omega} & \frac{\sin(\omega L)}{\omega} \\
 \tau_x = -1 & +1 & \tau_y = +1 & -1
 \end{array} \tag{19.72}$$

with this

$$\begin{array}{ll}
 m_{511} = \frac{\tau_x \omega^2}{4} (L - c_x s_x) & m_{533} = \frac{\tau_y \omega^2}{4} (L - c_y s_y) \\
 m_{512} = \frac{-\tau_x \omega^2}{2(1 + p_{z1})} s_x^2 & m_{534} = \frac{-\tau_y \omega^2}{2(1 + p_{z1})} s_y^2 \\
 m_{522} = \frac{-1}{4(1 + p_{z1})^2} (L + c_x s_x) & m_{544} = \frac{-1}{4(1 + p_{z1})^2} (L + c_y s_y)
 \end{array} \tag{19.73}$$

19.16 RFcavity Tracking

Tracking through an rfcavity uses a kick-drift-kick model. The kick is a pure energy kick (see equations in §3.36) and the phase of the RF is calculated under the assumption that the waveform moves at a phase velocity equal to the velocity of the reference particle.

The transverse forces due to the RF are ignored. This is a resonable approximation when the acceleration is small. Lcavity elements should be used in place of rf cavity elements when this is not so.

19.17 Sextupole Tracking

The Hamiltonian for an upright sextupole is

$$H = \frac{p_x^2 + p_y^2}{2(1 + p_z)} + \frac{k_2}{6}(x^3 - 3xy^2) \tag{19.74}$$

Tracking through a sextupole uses a kick-drift-kick model.

19.18 Sol_Quad Tracking

The Hamiltonian is

$$H = \frac{(p_x + \frac{k_s}{2} y)^2}{2(1 + p_z)} + \frac{(p_y - \frac{k_s}{2} x)^2}{2(1 + p_z)} + \frac{k_1}{2}(x^2 - y^2) \quad (19.75)$$

Solving the equations of motion gives

$$\begin{aligned} x_2 &= m_{11} x_1 + m_{12} p_{x1} + m_{13} y_1 + m_{14} p_{y1} \\ p_{x2} &= m_{21} x_1 + m_{22} p_{x1} + m_{23} y_1 + m_{24} p_{y1} \\ y_2 &= m_{31} x_1 + m_{32} p_{x1} + m_{33} y_1 + m_{34} p_{y1} \\ p_{y2} &= m_{41} x_1 + m_{42} p_{x1} + m_{43} y_1 + m_{44} p_{y1} \\ z_2 &= z_1 + \sum_{j=1}^4 \sum_{k=j}^4 m_{5jk} r_j r_k \\ p_{z2} &= p_{z1} \end{aligned} \quad (19.76)$$

where

$$\begin{aligned} m_{11} &= \frac{1}{2f} (f_{0+} c + f_{0-} c_h) & m_{31} &= -m_{24} \\ m_{12} &= \frac{1}{2f(1 + p_{z1})} \left(\frac{f_{++}}{\omega_+} s + \frac{f_{--}}{\omega_-} s_h \right) & m_{32} &= -m_{14} \\ m_{13} &= \frac{\tilde{k}_s}{4f} \left(\frac{f_{+-}}{\omega_+} s + \frac{f_{-+}}{\omega_-} s_h \right) & m_{33} &= \frac{1}{2f} (f_{0-} c + f_{0+} c_h) \\ m_{14} &= \frac{\tilde{k}_s}{f(1 + p_{z1})} (-c + c_h) & m_{34} &= \frac{1}{2f(1 + p_{z1})} \left(\frac{f_{+-}}{\omega_+} s + \frac{f_{-+}}{\omega_-} s_h \right) \\ m_{21} &= \frac{-(1 + p_{z1})}{8f} \left(\frac{\xi_{1+}}{\omega_+} s + \frac{\xi_{2+}}{\omega_-} s_h \right) & m_{41} &= -m_{23} \\ m_{22} &= m_{11} & m_{42} &= -m_{13} \\ m_{23} &= \frac{\tilde{k}_s^3 (1 + p_{z1})}{4f} (c - c_h) & m_{43} &= \frac{-(1 + p_{z1})}{8f} \left(\frac{\xi_{1-}}{\omega_+} s + \frac{\xi_{2-}}{\omega_-} s_h \right) \\ m_{24} &= \frac{\tilde{k}_s}{4f} \left(\frac{f_{++}}{\omega_+} s + \frac{f_{--}}{\omega_-} s_h \right) & m_{44} &= m_{33} \end{aligned} \quad (19.77)$$

and

$$\begin{aligned} \tilde{k}_1 &= \frac{k_1}{1 + p_{z1}} & \tilde{k}_s &= \frac{k_s}{1 + p_{z1}} \\ f &= \sqrt{\tilde{k}_s^4 + 4\tilde{k}_1^2} & f_{\pm 0} &= f \pm \tilde{k}_s^2 \\ f_{0\pm} &= f \pm 2\tilde{k}_1 & f_{\pm\pm} &= f \pm \tilde{k}_s^2 \pm 2\tilde{k}_1 \\ \omega_+ &= \sqrt{\frac{f_{+0}}{2}} & \omega_- &= \sqrt{\frac{f_{-0}}{2}} \\ s &= \sin(\omega_+ L) & s_h &= \sinh(\omega_- L) \\ c &= \cos(\omega_+ L) & c_h &= \cosh(\omega_- L) \\ \xi_{1\pm} &= \tilde{k}_s^2 f_{+\mp} \pm 4\tilde{k}_1 f_{\pm\mp} & \xi_{2\pm} &= \tilde{k}_s^2 f_{-\pm} \pm 4\tilde{k}_1 f_{-\mp} \end{aligned} \quad (19.78)$$

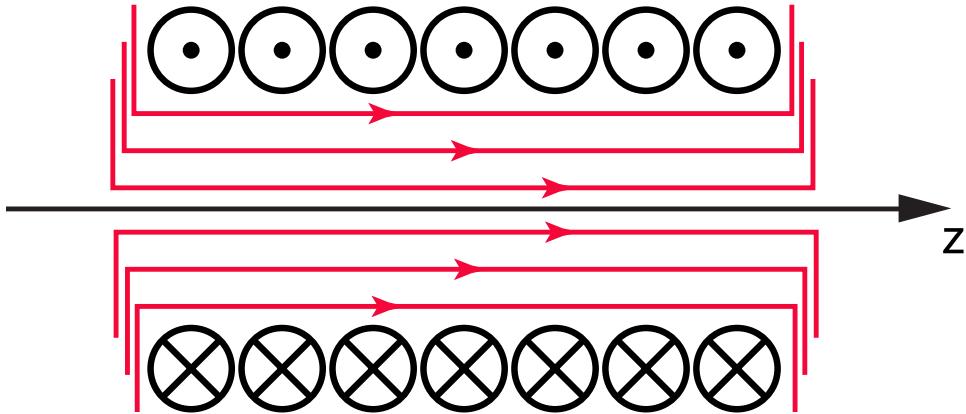


Figure 19.4: Solenoid with a hard edge. The field is assumed to end abruptly at the edges of the solenoid. Here, for purposes of illustration, the field lines at the ends are displaced from one another.

The m_{5jk} terms are obtained via Eq. (19.22)

$$m_{5jk} = -\frac{\tau_{jk}}{2(1+p_{z1})^2} \int ds \left[\left(m_{2j} + \frac{k_s}{2} m_{3j} \right) \left(m_{2k} + \frac{k_s}{2} m_{3k} \right) + \left(m_{4j} - \frac{k_s}{2} m_{1j} \right) \left(m_{4k} - \frac{k_s}{2} m_{1k} \right) \right] \quad (19.79)$$

where

$$\tau_{jk} = \begin{cases} 1 & j = k \\ 2 & j \neq k \end{cases} \quad (19.80)$$

The needed integrals involve the product of two trigonometric or hyperbolic functions. These integrals are trivial to do but the explicit equations for m_{5jk} are quite long and in the interests of brevity are not reproduced here.

19.19 Solenoid Tracking

The Hamiltonian for a solenoid is

$$H = \frac{(p_x + \frac{k_s}{2} y)^2}{2(1+p_z)} + \frac{(p_y - \frac{k_s}{2} x)^2}{2(1+p_z)} \quad (19.81)$$

where the normalized field k_s is

$$k_s = \frac{B}{P_0} \quad (19.82)$$

The solution to the equations of motion for a constant k_s are

$$\begin{aligned} x_2 &= \frac{1+c}{2} x_1 + \frac{s}{k_s} p_{x1} + \frac{s}{2} y_1 + \frac{1-c}{k_s} p_{y1} \\ p_{x2} &= \frac{-k_s s}{4} x_1 + \frac{1+c}{2} p_{x1} - \frac{k_s (1-c)}{4} y_1 + \frac{s}{2} p_{y1} \\ y_2 &= \frac{-s}{2} x_1 - \frac{1-c}{k_s} p_{x1} + \frac{1+c}{2} y_1 + \frac{s}{k_s} p_{y1} \\ p_{y2} &= \frac{k_s (1-c)}{4} x_1 + \frac{-s}{2} p_{x1} - \frac{k_s s}{4} y_1 + \frac{1+c}{2} p_{y1} \\ z_2 &= z_1 + \frac{L}{2(1+p_{z1})^2} \left[\left(p_{x1} + \frac{k_s}{2} y_1 \right)^2 + \left(p_{y1} - \frac{k_s}{2} x_1 \right)^2 \right] \\ p_{z2} &= p_{z1} \end{aligned} \quad (19.83)$$

where

$$\begin{aligned} c &= \cos \left(\frac{k_s}{2} L \right) \\ s &= \sin \left(\frac{k_s}{2} L \right) \end{aligned} \quad (19.84)$$

To be useful, the canonical momenta p_x and p_y in the above equations must be connected to the cononical momenta used for other elements (drifts, quadrupoles, etc.) that may be placed to either side of the solenoid. These side elements use zero a_x and a_y (cf. Eq. (19.16)). The vector potential used in the solenoid canonical momenta may be made zero at the edges of the solenoid if the solenoid fringe field is assumed to end abruptly at the edges of the solenoid (as shown in Fig. 19.4), and the reference axis **z**-axis (at $x = y = 0$) is placed along the centerline of the solenoid so that there is cylindrical symmetry around the **z**-axis.

Chapter 20

Tracking of X-Rays

Bmad can track both charged particles and X-rays. This chapter deals with X-rays. Charged particles are handled in chapter §19.

20.1 Coherent and Incoherent Photon Simulations

Bmad can track photons either **coherently** or **incoherently**. In both cases, the photon has a transverse electric field

$$(E_x, E_y) \quad (20.1)$$

E_x and E_y are complex and therefore have both amplitude and phase information. When photons are tracked incoherently, the phase information is not used for calculating X-ray intensities.

In addition to coherent and incoherent tracking, partially coherent simulations can be done by using sets of photons with the photons in any one set treated as coherent and the photons between sets being treated as incoherent.

20.1.1 Incoherent Photon Tracking

In a simulation with incoherent photons, some number of photons, N_0 , will be generated and the i^{th} photon ($i = 1, \dots, N_0$) will have a initial “electric field” components $E_{x0}(i), E_{y0}(i)$ assigned to it. The field amplitude E_0 will be $\sqrt{E_{x0}^2 + E_{y0}^2}$.

At some an observation point, the power S per unit area falling on some small area dA due to either x or y component of the electric field is

$$S_{x,y} = \frac{\alpha_p}{N_0 dA} \sum_{j \in \text{hits}} E_{x,y}^2(j) \quad (20.2)$$

where α_p is a constant that can be chosen to fit the simulation against experimental results, and the sum is over photons who intersect the area. The factors of N_0 and dA in the above equation make, within statistical flucuations, S independent of N_0 and, for dA small enough, S will be independent of dA as it should be. The total power is just $S_x + S_y$.

When traveling through vacuum, the electric field of a photon is a constant. As an example, consider a point source raidiating uniformly in 4π solid angle with each photon having the same initial field E_0 . An

observation area dA situated a distance R from the source will intercept $N_0 dA/4\pi R^2$ photons which gives a power of

$$S_w = \frac{\alpha_p E_0^2}{4\pi R^2} \quad (20.3)$$

which falls off as $1/R^2$ as expected.

At some places the light may be split into various “channels”. An example is Laue diffraction where X-rays can excite the α and β branches of the dispersion surface. Or a partially silvered mirror where some of the light is reflected and some is transmitted. In such a case, the probability P_i of a photon traveling down the i^{th} channel is

$$P_i \widehat{E}_i^2 = \frac{S_i}{S_0} \quad (20.4)$$

where S_i is the power flowing into channel i , S_0 is the power flowing into the junction, and $\widehat{E}_i = E_i/E_0$ is the ratio of the electric field amplitudes of any photon just before and just after being shunted into the i^{th} channel. The probabilities must be properly normalized

$$\sum P_i = 1 \quad (20.5)$$

If the ratio of the electric field of any photon just before and just after being shunted into the i^{th} channel is not a constant, than \widehat{E}_i must be adjusted so that \widehat{E}_i^2 is equal to the average of $\widehat{E}_i^2(j)$ for all photons j channeled into channel i .

As long as Eqs. (20.4) and (20.5) are satisfied, the choice of the P_i , and \widehat{E}_i are arbitrary. This freedom allows simulation to be optimized for efficiency. For example, In an actual experiment much of the light can be lost never to reach a detector and be counted. To decrease the simulation time, simulated photons may be limited to be generated with a direction to be within some solid angle Ω_1 if photons with a direction outside this solid angle will not contribute to the simulation results. In this case, there are two channels. Channel 1 consists of all photons whose direction is withing Ω_1 and channel 2 is all the other photons. To limit the photons to channel 1, P_1 is taken to be 1 and P_2 is taken to be 0. Additionally, if the light, say, is being generated isotropically from a surface into a $\Omega_0 = 2\pi$ solid angle then

$$\widehat{E}_1 = \sqrt{\frac{\Omega_1}{\Omega_0}} \quad (20.6)$$

\widehat{E}_2 is infinite here but since no photons are generated in channel 2 this is not a problem.

20.1.2 Coherent Photon Tracking

In a simulation with coherent photons, some number of photons, N_0 , will be generated and the i^{th} photon ($i = 1, \dots, N_0$) will have an initial electric field $E_{x0}(i), E_{y0}(i)$ assigned to it. These quantities will be complex.

At some an observation point, the field E at some small area dA due to either x or y component of the electric field is

$$E = \frac{\alpha_p}{N_0 dA} \sum_{j \in \text{hits}} E(j) \quad (20.7)$$

where α_p is a constant that can be chosen to fit the simulation against experimental results, and the sum is over photons who intersect the area. In the above equation $E(j)$ is either the x or y component of the electric field as is appropriate. The factors of N_0 and dA in the above equation make, within statistical flucuations, E independent of N_0 and, for dA small enough, E will be independent of dA as it should be.

When traveling through a vacuum, the photons travel ballistically in straight lines. This is justified by using the stationary phase approximation with Kirchhoff's integral. The electric field of a photon varies with the propagation length. There is nothing physical in this and is just a way to make the bookkeeping come out correctly. As an example, consider a point source radiating uniformly in 4π solid angle with each photon having the same initial field component (either x or y) E_1 . An observation area dA situated a distance R from the source will intercept $N_0 dA/4\pi R^2$ photons and each photon will have a field of $E_1 R \exp(i k R)$ where k is the photon wave number (all photons must have the same k to be coherent). This gives an electric field at the observation point of

$$E = \frac{\alpha_p E_1 \exp(i k R)}{4\pi R} \quad (20.8)$$

which falls off as $1/R$ as expected.

At an aperture where diffraction effects are to be simulated, the following procedure is used:

1. The electric field components are multiplied by the propagation length L :

$$E \rightarrow E L \quad (20.9)$$

The propagation length is reset to zero so that at the next point where the propagation length is factored into the electric field the propagation length will be the length starting at the aperture.

2. The photon is given a random direction over the 2π solid angle of the opening.
3. The electric field components are scaled by

$$E \rightarrow E \frac{k}{4\pi i} (\cos \theta_1 + \cos \theta_2) \quad (20.10)$$

where θ_1 and θ_2 are the direction cosines of the incoming and outgoing directions of the photon with respect to the longitudinal reference axis.

This algorithm is designed so that the resulting fields at points downstream from the aperture as computed from a simulation will, to within statistical errors, be the same as one would get using Kirchhoff's integral. That is, the simulation is constructed to be a Monte Carlo integration of Kirchhoff's integral.

What is, and what is not considered a place where there are diffraction effects is dependent upon the problem. For example, there are diffraction effects associated with light reflecting from a mirror (or any other object) of finite size. If these effects are important to the experiment, then a procedure similar to the one above must be followed.

At places where there are no diffraction effects a simulation can treat the photons ballistically or can use the aperture procedure outlined above. While in theory it is possible to choose what to do, in practice the aperture procedure increases the number of photons that must be tracked for a given resolution. Thus, from a practical standpoint the ballistic alternative should always be used.

As explained in §20.1.1, at some places the light may be split into various "channels". With coherent photons, the analog to Eq. (20.4) is

$$P_i \widehat{E}_i = \frac{E_i}{E_0} \quad (20.11)$$

where here \widehat{E}_i can be complex to take into account phase shifts. The same considerations about choosing the P_i and \widehat{E}_i apply to coherent photons as incoherent photons. In particular, \widehat{E}_1 for the case of isotropic emission from a surface as in the example in §20.1.1 (cf. Eq. (20.6)) is

$$\widehat{E}_1 = \frac{\Omega_1}{\Omega_0} \quad (20.12)$$

20.1.3 Partially Coherent Photon Simulations

When there is partial coherence the photons must be divided into sets. All of the photons of a given set are considered coherent while the photons of different sets are treated incoherently.

The procedure is to track all the photons of one set coherently and calculate the field using equation Eq. (20.7). The fields of different sets are then combined to calculate a power using Eq. (20.2).

20.2 Element Coordinate System

The general procedure for tracking through an element makes use of **element reference** coordinates (also called just **element** coordinates). Without any offsets, pitches or tilt (§4.4), henceforth called “misalignments”, the **element** coordinates are the same as the **laboratory reference** coordinates (or simply **laboratory** coordinates) (§13.1). The **element** coordinates stay fixed relative to the element. Therefore, if the element is misaligned, the **element coordinates** will follow as the element shifts in the laboratory frame as shown in Fig. 19.1.

For **crystal** (§3.8), **mirror** (§3.28), and **multilayer_mirror** (§3.30) elements, the “kinked” reference trajectory through the element complicates the calculation. For these elements, there are three coordinate systems attached to the element as shown in Fig. 20.1. Besides the **element entrance** and **element exit** coordinates, there are **element surface** coordinates with z perpendicular to the surface pointing inward.

Tracking a particle through an element is therefore a three step transformation:

1. At the entrance end of the element, transform from the laboratory reference coordinates to the element’s **entrance** or **surface** coordinates.
2. Track through the element ignoring any misalignments.
3. At the exit end of the element, transform from the element coordinates to the **laboratory exit** coordinates.

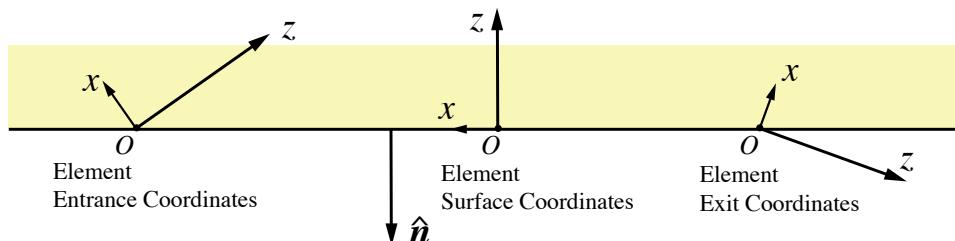


Figure 20.1: The three element coordinate systems for **crystal** (Bragg configuration), **mirror**, and **multilayer_mirror** elements. The origin **O** of all three are the same but are shown spread out for clarity. $\hat{\mathbf{n}}$ is the normal to the element surface.

20.2.1 Transform from Laboratory Entrance to Element Coordinates

For elements that have a reference orbit kink (§20.2), the element coordinates here are the `surface` coordinates. Otherwise the element coordinates are the entrance coordinates.

1. Apply offsets, pitches and tilt using the formulas in §13.3.2 along with Eqs. (13.14), (13.15), and (13.16).
2. Apply the `tilt` to the electric field (Eq. (13.36)).
3. For `crystal`, `mirror`, and `multilayer_mirror` elements rotate to element surface coordinates.
4. Transform the photon's position as if in a drift by a distance $-z$ where z is the photon's longitudinal coordinate. That is, z will be zero at the end of the transform to element coordinates (remember that z is the distance from the start of the element (§13.5)).

20.2.2 Transform from Element Exit to Laboratory Coordinate

The back transformation from element to laboratory coordinates is accomplished by the transformation

1. For `crystal`, `mirror`, and `multilayer_mirror` elements rotate to element from element surface coordinates to element exit coordinates
2. Apply the reverse `tilt` to the electric field (Eq. (13.36)).
3. Apply reverse offsets, pitches and tilt using the formulas in §13.3.2 along with Eqs. (13.14), (13.15), and (13.16).

20.3 Transformation for Mirror and Crystal Elements Between Laboratory and Element Coordinates

20.3.1 Transformation from Laboratory to Element Coordinates

With photons, the intensities must also be transformed. The transformation from the the entrance laboratory coordinates to the entrance element coordinates is:

1. Track as in a drift a distance `z_offset_tot`.
2. Apply offsets and pitches: The effective “length” of the element is zero (§13.3.3) so the origin of the element coordinates is the same point around which the element is pitched so

$$\begin{aligned} x_1 &= x_0 - x_{\text{off}} \\ p_{x1} &= p_{x0} - (1 + p_{z0}) x'_{\text{pitch}} \\ y_1 &= y_0 - y_{\text{off}} \\ p_{y1} &= p_{x0} - (1 + p_{z0}) y'_{\text{pitch}} \\ z_1 &= z_0 + x'_{\text{pitch}} x_1 + y'_{\text{pitch}} y_1 \end{aligned} \tag{20.13}$$

where $x_{\text{off}} \equiv \text{x_offset}$, $x'_{\text{pitch}} \equiv \text{x_pitch}$, etc.

3. Apply `ref_tilt` and `tilt`:

$$\begin{aligned} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} &= \mathbf{R}(\theta_{tot}) \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \\ \begin{pmatrix} p_{x2} \\ p_{y2} \end{pmatrix} &= \mathbf{R}(\theta_{tot}) \begin{pmatrix} p_{x1} \\ p_{y1} \end{pmatrix} \\ \begin{pmatrix} \mathbf{E}_{x2} \\ \mathbf{E}_{y2} \end{pmatrix} &= \mathbf{R}(\theta_{tot}) \begin{pmatrix} \mathbf{E}_{x1} \\ \mathbf{E}_{y1} \end{pmatrix} \end{aligned} \quad (20.14)$$

where \mathbf{E} is shorthand notation for

$$\mathbf{E} \equiv E e^{i\phi} \quad (20.15)$$

with E being the field intensity and ϕ being the field phase angle. In the above equations \mathbf{R} is the rotation matrix

$$\mathbf{R}(\theta) = \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} \quad (20.16)$$

with θ_{tot} being

$$\theta_{tot} = \begin{cases} \text{ref_tilt} + \text{tilt} + \text{tilt_corr} & \text{for crystal elements} \\ \text{ref_tilt} + \text{tilt} & \text{for mirror elements} \end{cases} \quad (20.17)$$

The `tilt_corr` correction is explained in §20.4.2.

20.3.2 Transformation from Element to Laboratory Coordinates

The back transformation from exit element coordinates to exit laboratory coordinates is accomplished by the transformation

1. Apply `ref_tilt` and `tilt`: `ref_tilt` rotates the exit laboratory coordinates with respect to the exit element coordinates in the same way `ref_tilt` rotates the entrance laboratory coordinates with respect to the entrance element coordinates. The forward and back transformations are thus just inverses of each other. With `tilt`, this is not true. `tilt`, unlike `ref_tilt`, does not rotate the output laboratory coordinates. There is the further complication in that `tilt` is a rotation about the *entrance* laboratory coordinates. The first step is to express `tilt` with respect to the exit coordinates. This is done with the help of the \mathbf{S} matrix of Eq. (13.7) with α given by Eq. (13.13). The effect of the `tilt` can be modeled as a rotation vector \mathbf{e}_{in} in the entrance laboratory coordinates pointing along the z -axis

$$\mathbf{e}_{in} = (0, 0, \text{tilt}) \quad (20.18)$$

In the exit laboratory coordinates, the vector \mathbf{e}_{out} is

$$\mathbf{e}_{out} = \mathbf{S} \mathbf{e}_{in} \quad (20.19)$$

The z component of \mathbf{e}_{out} combines with `ref_tilt` to give the transformation

$$\begin{aligned} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} &= \mathbf{R}(-\theta_t) \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \\ \begin{pmatrix} p_{x2} \\ p_{y2} \end{pmatrix} &= \mathbf{R}(-\theta_t) \begin{pmatrix} p_{x1} \\ p_{y1} \end{pmatrix} \\ \begin{pmatrix} \mathbf{E}_{x2} \\ \mathbf{E}_{y2} \end{pmatrix} &= \mathbf{R}(-\theta_t) \begin{pmatrix} \mathbf{E}_{x1} \\ \mathbf{E}_{y1} \end{pmatrix} \end{aligned} \quad (20.20)$$

where θ_t is $\text{ref_tilt} + \mathbf{e}_{out,z}$. The x and y components of \mathbf{e}_{out} give rotations around the x and y axes

$$p_{x3} = p_{x2} - \mathbf{e}_{out,y} \quad (20.21)$$

$$p_{y3} = p_{y2} + \mathbf{e}_{out,x} \quad (20.22)$$

2. Apply pitches: Since pitches are defined with respect to the entrance laboratory coordinates, they have to be translated to the exit laboratory coordinates

$$\mathbf{P}_{out} = \mathbf{S} \mathbf{P}_{in} \quad (20.23)$$

where $\mathbf{P}_{in} = (x'_{pitch}, y'_{pitch}, 0)$ is the pitch vector in the entrance laboratory frame and \mathbf{P}_{out} is the vector in the exit laboratory frame. The transformation is then

$$p_{x4} = p_{x3} - \mathbf{P}_{out,y} \quad (20.24)$$

$$p_{y4} = p_{y3} + \mathbf{P}_{out,x} \quad (20.25)$$

3. Apply offsets: Again, offsets are defined with respect to the entrance laboratory coordinates. Like pitches, the translation is

$$\mathbf{O}_{out} = \mathbf{S} \mathbf{O}_{in} \quad (20.26)$$

where $\mathbf{O}_{in} = (x_{off}, y_{off}, s_{off})$ is the offset in the entrance laboratory frame. The transformation is

$$x_5 = x_4 + \mathbf{O}_{out,x} - p_{x4} \mathbf{O}_{out,z} \quad (20.27)$$

$$y_5 = y_4 + \mathbf{O}_{out,y} - p_{y4} \mathbf{O}_{out,z} \quad (20.28)$$

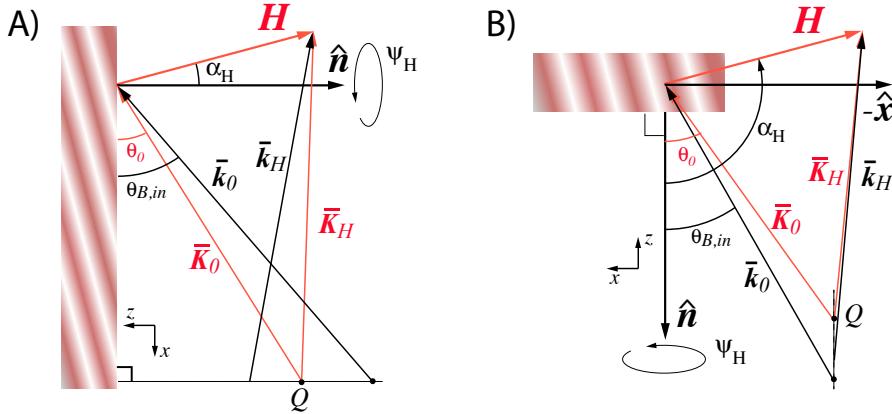


Figure 20.2: Reference trajectory reciprocal space diagram for A) Bragg diffraction and B) Laue diffraction. The bar over the vectors indicates that they refer to the reference trajectory. The x - z coordinates shown are the element surface coordinates. All points in the diagram are in the plane of the paper except for the tip of \mathbf{H} . $\bar{\mathbf{K}}_0$, and $\bar{\mathbf{K}}_H$ are the wave vectors inside the crystal and $\bar{\mathbf{k}}_0$ and $\bar{\mathbf{k}}_H$ are the wave vectors outside the crystal. The reference photon traveling along the reference trajectory has $\bar{\mathbf{K}}_0$ and $\bar{\mathbf{K}}_H$ originating at the Q point. For Laue diffraction, the crystal faces are assumed parallel. For Bragg diffraction the crystal normal is in the $-\hat{x}$ direction while for Laue diffraction the crystal normal is in the $-\hat{z}$ direction

20.4 Crystal Element Tracking

[Crystal tracking developed by Jing Yee Chee, Ken Finkelstein, and David Sagan]

Crystal diffraction is modeled using dynamical diffraction theory. The notation here follows Batterman and Cole [Bater64]. The problem can be divided up into two parts. First the reference trajectory must be calculated. This means calculating the incoming grazing angle $\theta_{B,in}$ and outgoing grazing angle $\theta_{B,out}$ as well as calculating the transformations between the various coordinate systems. This is done in §20.4.1, §20.4.2, and §20.4.3. The second part is the actual tracking of the photon and this is covered in §20.4.5, §20.4.6 and §20.4.7.

20.4.1 Calculation of Entrance and Exit Bragg Angles

Fig. 20.2 shows the geometry of the problem. The bar over the vectors indicates that they refer to the reference trajectory. The reference trajectory is calculated such that the reference photon will be in the center of the Darwin curve. That is, the internal wave vectors $\bar{\mathbf{K}}_0$ and $\bar{\mathbf{K}}_H$ originate from the Q point (See [Bater64] Figs. 8 and 29).

The external wave vectors \mathbf{k}_0 , and \mathbf{k}_H and the internal wave vectors have magnitude

$$|\mathbf{k}_0| = |\mathbf{k}_H| = \frac{1}{\lambda} \quad (20.29)$$

$$|\bar{\mathbf{K}}_0| = |\bar{\mathbf{K}}_H| = \frac{1 - \delta}{\lambda} \quad (20.30)$$

where λ is the wavelength, and δ is

$$\delta = \frac{\lambda^2 r_e}{2 \pi V} F'_0 = \frac{\Gamma}{2} F'_0 = \frac{1}{2} \Gamma F'_0 \quad (20.31)$$

with r_e being the classical electron radius, V the unit cell volume, and F'_0 is the real part of the F_0 structure factor.

In element surface coordinates (which will be the coordinate system used henceforth), $\bar{\mathbf{k}}_0$ lies in the x - z plane. $\bar{\mathbf{K}}_0$ is related to $\bar{\mathbf{k}}_0$ via Batterman Eq. (25)

$$\mathbf{K}_0 = \mathbf{k}_0 + q_0 \hat{\mathbf{n}} \quad (20.32)$$

where the value of q_0 is to be determined. Here, and in equations below, if the equation is true in general, and not just for the reference trajectory, the bar superscript is dropped.

Since $\hat{\mathbf{n}}$ is in the $-\hat{x}$ direction, $\bar{\mathbf{K}}_0$ is also in the x - z plane. Thus $\bar{\mathbf{k}}_0$ and $\bar{\mathbf{K}}_0$ can be written in the form

$$\begin{aligned} \bar{\mathbf{k}}_0 &= \frac{1}{\lambda} \begin{pmatrix} -\cos \theta_{B,in} \\ 0 \\ \sin \theta_{B,in} \end{pmatrix}, & \bar{\mathbf{K}}_0 &= \frac{1-\delta}{\lambda} \begin{pmatrix} -\cos \theta_0 \\ 0 \\ \sin \theta_0 \end{pmatrix} & [\text{Bragg}] \\ \bar{\mathbf{k}}_0 &= \frac{1}{\lambda} \begin{pmatrix} \sin \theta_{B,in} \\ 0 \\ \cos \theta_{B,in} \end{pmatrix}, & \bar{\mathbf{K}}_0 &= \frac{1-\delta}{\lambda} \begin{pmatrix} \sin \theta_0 \\ 0 \\ \cos \theta_0 \end{pmatrix} & [\text{Laue}] \end{aligned} \quad (20.33)$$

Where, as shown in Fig. 20.2, $\theta_{B,in}$, and θ_0 are the angles of $\bar{\mathbf{k}}_0$ and $\bar{\mathbf{K}}_0$ with respect to the x -axis for Bragg reflections and with respect to the z -axis for Laue reflection.

α_H (`alpha_angle`) is the angle that \mathbf{H} makes with respect to the $-\hat{z}$ axis and ψ_H (`psi_angle`) is the rotation of \mathbf{H} around the $-\hat{z}$ axis such that for $\psi_H = 0$, \mathbf{H} is in the x - z plane and oriented as shown in Fig. 20.2. Thus

$$\mathbf{H} \equiv \frac{1}{d} \hat{\mathbf{H}} = \frac{1}{d} \begin{pmatrix} -\sin \alpha_H \cos \psi_H \\ \sin \alpha_H \sin \psi_H \\ -\cos \alpha_H \end{pmatrix} \quad (20.34)$$

where $\hat{\mathbf{H}}$ is \mathbf{H} normalized to 1. α_H is determined via the setting of `b_param` and via Eq. (3.9).

The vectors \mathbf{K}_0 and \mathbf{H} must add up to the reciprocal lattice vector \mathbf{K}_H

$$\mathbf{K}_H = \mathbf{K}_0 + \mathbf{H} \quad (20.35)$$

Taking the length of both sides of this equation and using Eqs. (20.30), (20.33), and (20.34) gives for θ_0

$$\sin \theta_0 = \begin{cases} \frac{-\beta \hat{H}_z - \hat{H}_x \sqrt{\hat{H}_x^2 + \hat{H}_z^2 - \beta^2}}{\hat{H}_x^2 + \hat{H}_z^2} & \text{Bragg} \\ \frac{-\beta \hat{H}_x + \hat{H}_z \sqrt{\hat{H}_x^2 + \hat{H}_z^2 - \beta^2}}{\hat{H}_x^2 + \hat{H}_z^2} & \text{Laue} \end{cases} \quad (20.36)$$

where

$$\beta \equiv \frac{\lambda}{2 d (1 - \delta)} \quad (20.37)$$

Once θ_0 has been calculated, $\theta_{B,in}$ can be calculated from Eq. (20.32)

$$\cos \theta_{B,in} = (1 - \delta) \cos \theta_0 \quad [\text{Bragg}] \quad (20.38)$$

$$\sin \theta_{B,in} = (1 - \delta) \sin \theta_0 \quad [\text{Laue}] \quad (20.39)$$

The outgoing reference wave vector k_H is computed using the equation

$$\mathbf{K}_H = \mathbf{k}_H + q_H \hat{\mathbf{n}} \quad (20.40)$$

Using this with Eqs. (20.34) and (20.35) gives

$$\begin{aligned}\bar{k}_{H,x} &= \bar{K}_{H,z} = \frac{1}{d} \hat{H}_x + k_{0,x} \\ \bar{k}_{H,y} &= \bar{K}_{H,y} = \frac{1}{d} \hat{H}_y \\ \bar{k}_{H,z} &= \sqrt{\frac{1}{\lambda^2} - \bar{k}_{H,x}^2 - \bar{k}_{H,y}^2}\end{aligned}\tag{20.41}$$

The total bending angle of the reference trajectory is then

$$\theta_{bend} = \tan^{-1} \left(\frac{|\bar{\mathbf{k}}_0 \times \bar{\mathbf{k}}_H|}{\bar{\mathbf{k}}_0 \cdot \bar{\mathbf{k}}_H} \right)\tag{20.42}$$

The outgoing Bragg angle $\theta_{B,out}$ is then *defined* to be the difference between the total bend angle and the entrance Bragg angle.

$$\theta_{B,out} \equiv \theta_{bend} - \theta_{B,in}\tag{20.43}$$

20.4.2 Crystal Coordinate Transformations

There are four transformations needed between coordinates denoted by Σ_1 , Σ_2 , Σ_3 , and Σ_4

Σ_1 Transform from laboratory entrance to element entrance coordinates.

Σ_2 Transform from element entrance to surface coordinates.

Σ_3 Transform from surface to element exit coordinates.

Σ_4 Transform from element exit to laboratory exit coordinates.

The total transformation is just the map represented by \mathbf{S} and \mathbf{V} of Eqs. (13.4) and (13.5)

$$[\mathbf{S}, \mathbf{V}] = \Sigma_4 \Sigma_3 \Sigma_2 \Sigma_1\tag{20.44}$$

The transformation Σ_1 is given in §20.3.1 and the transformation Σ_4 is given in §20.3.2. In general, the transformation Σ_1 needs a “tilt correction” (Eq. (20.17)), as explained below, when ψ_H is nonzero. [The exception is when the undiffracted or forward_diffracted beam is tracked with Laue geometry. In these cases, no tilt correction is needed.] Since this tilt correction is independent of any misalignments, the tilt correction calculation proceeds assuming here that there are no misalignments. The finite \mathbf{V} due to the finite crystal thickness in Laue diffraction will also be ignored for the moment.

Without misalignments, and with ψ_H zero, the transformation Σ_1 is, as it is for every other type of element, just the unit matrix.

$$\Sigma_1 = \mathbf{I}\tag{20.45}$$

That is, the two coordinate systems are identical. Furthermore, the transformation Σ_2 from element entrance coordinates to surface coordinates is a rotation around the y axis

$$\begin{aligned}\Sigma_2 &= \mathbf{R}_y(\theta_{B,in}) \equiv \begin{pmatrix} \cos \theta_{B,in} & 0 & \sin \theta_{B,in} \\ 0 & 1 & 0 \\ -\sin \theta_{B,in} & 0 & \cos \theta_{B,in} \end{pmatrix} && [\text{Laue}] \\ &= \mathbf{R}_y(\theta_{B,in} - \frac{\pi}{2}) && [\text{Bragg}]\end{aligned}\tag{20.46}$$

The transformation from element surface coordinates to element exit coordinates, Σ_3 , is another rotation around the y axis

$$\begin{aligned}\Sigma_3 &= \mathbf{R}_y(\theta_{B,out}) && [\text{Laue}] \\ &= \mathbf{R}_y(\theta_{B,out} + \frac{\pi}{2}) && [\text{Bragg}]\end{aligned}\tag{20.47}$$

and the transformation from element exit coordinates to laboratory exit coordinates, Σ_{out} is the unity matrix

$$\Sigma_4 = \mathbf{I} \quad (20.48)$$

Thus, the combined transformation \mathbf{S} from laboratory entrance to laboratory exit coordinates is a rotation around the y axis of $\theta_{B,in} + \theta_{B,out}$ as explained in section §13.3

$$\mathbf{S} = \Sigma_4 \Sigma_3 \Sigma_2 \Sigma_1 = \mathbf{R}_y(\theta_{B,in} + \theta_{B,out}) \quad (20.49)$$

When ψ_H is non-zero, the situation is complicated since, if \mathbf{S} as calculated above is used, the vector $\bar{\mathbf{k}}_H$ would be bent out of the x - z plane even though it has been assumed that the `ref_tilt` θ_t is zero. But $\bar{\mathbf{k}}_H$ points in the same direction as the z axis of the outgoing reference trajectory. Furthermore, by *definition*, the reference trajectory has the form given by Eq. (13.8) with the \mathbf{T} matrix depending only upon the `ref_tilt` parameter (which is here taken to be zero). To satisfy Eq. (13.8), the crystal must be reorientated to keep the \mathbf{k}_H vector in the x - z plane of the laboratory entrance coordinates. The reorientation is done by rotating the crystal about the laboratory entrance \mathbf{z} axis by an amount θ_{corr} (`tilt_corr`).

With this tilt correction the transformation Σ_1 is a rotation about the z axis

$$\Sigma_1 = \begin{pmatrix} \cos \theta_{corr} & -\sin \theta_{corr} & 0 \\ \sin \theta_{corr} & \cos \theta_{corr} & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (20.50)$$

To calculate a value for θ_{corr} , note that the transformation Σ_2 from element entrance coordinates to element surface coordinates is not affected by a finite ψ_H and so Eq. (20.46) is unmodified. The \mathbf{k}_H vector, expressed in laboratory entrance coordinates, is $\Sigma_1^{-1} \Sigma_2^{-1} \mathbf{k}_H$ where the components of \mathbf{k}_H are given by Eq. (20.41). To satisfy Eq. (13.8), this vector must have zero y component

$$(\Sigma_1^{-1} \Sigma_2^{-1} \mathbf{k}_H) \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = 0 \quad (20.51)$$

Solving gives

$$\theta_{corr} = \tan^{-1} \frac{k_{H,y}}{k_{H,z} \sin \theta_{B,in} - k_{H,x} \cos \theta_{B,in}} \quad (20.52)$$

The transformation Σ_3 from element surface coordinates to element exit coordinates is now obtained by requiring that the total transformation from laboratory entrance to laboratory exit coordinates be the $\tilde{\mathbf{S}}$ matrix given in Eq. (13.8)

$$\Sigma_3 \Sigma_2 \Sigma_1 = \begin{pmatrix} \cos \theta_{bend} & 0 & -\sin \theta_{bend} \\ 0 & 1 & 0 \\ \sin \theta_{bend} & 0 & \cos \theta_{bend} \end{pmatrix} \quad (20.53)$$

In the above equation, the transformation Σ_4 has been dropped since it is the unit matrix independent of ψ_H .

For Laue diffraction when the non-diffracted beam is tracked, the exit coordinate system corresponds to the entrance coordinate system. That is, \mathbf{V} is the unit matrix. In this case, there is no tilt correction and $\Sigma_3 = \mathbf{R}_y(-\theta_{B,in})$ is just the inverse of Σ_2 .

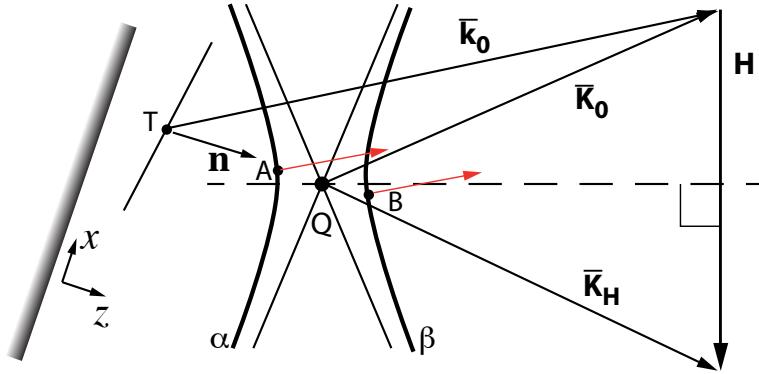


Figure 20.3: Energy flow used to determine the reference orbit for Laue diffraction.

20.4.3 Laue Reference Orbit

For Laue diffraction, with the reference orbit following the **undiffracted** beam, the reference orbit at the exit surface is just the extension of the reference orbit at the entrance surface. Since the reference orbit's direction is $\bar{\mathbf{k}}_0$, the reference orbit displacement vector \mathbf{L} (cf. Eq. (13.4)) is given by

$$\mathbf{L} = \frac{t^2}{\bar{\mathbf{k}}_0 \cdot \mathbf{t}} d\bar{\mathbf{k}}_0 \quad [\text{undiffracted}] \quad (20.54)$$

where

$$\mathbf{t} = \begin{pmatrix} 0 \\ 0 \\ t \end{pmatrix} \quad (20.55)$$

with t being the crystal thickness and the z -axis pointing into the crystal as illustrated in Fig. 20.3. The **S** roatation matrix (Eq. (13.5)) for the undiffracted beam referece will be the unit matrix.

With the reference orbit following the **forward_diffracted** or **Bragg_diffracted** beam, the displacement vector \mathbf{L} follows the energy flow associated with the tie points labeled A or B in Fig. 20.3. These tie points are defined by the intersection of the dispersion surfaces and the vector \mathbf{n} originating from the point T as shown in the figure. The energy flow is perpendicular to the dispersion surface and it can be shown that since, by construction, \mathbf{n} goes through the Q point and since the dispersion surfaces are hyperbolies, the energy flow from A and B tie points are colinear. The direction of the energy flow is given by:

$$\bar{\mathbf{K}}_f = \xi_H \bar{\mathbf{K}}_H + \xi_0 \bar{\mathbf{k}}_0 \quad (20.56)$$

\mathbf{L} is thus

$$\mathbf{L} = \frac{t^2}{\bar{\mathbf{K}}_f \cdot \mathbf{t}} \bar{\mathbf{K}}_f \quad (20.57)$$

where \mathbf{t} is a vector whose length is the thickness of the crystal oriented perpendicular to the crystal surfaces (that is, in the z direction of the local coordinate system). At the exit surface, if the reference orbit is following the **forward_diffracted** beam, the orientation of the **element_exit** coordinates will be the same as the orientation of the **element_entrance** coordinates. That is, **S** (Eq. (13.5)) is the unit matrix. If the reference orbit is following the Bragg diffracted beam, **S** is the same as for Bragg diffraction

$$\mathbf{S} = \begin{pmatrix} \cos \theta_{bend} & 0 & -\sin \theta_{bend} \\ 0 & 1 & 0 \\ \sin \theta_{bend} & 0 & \cos \theta_{bend} \end{pmatrix} \quad (20.58)$$

Figure 20.4: Reflection from a crystal surface.

20.4.4 Crystal Surface Reflections and Refractions

There are corrections to the field amplitude and phase when a photon reflects or refracts from the surface of a crystal. The situation is illustrated in Fig. 20.4. A plane wave incident on a crystal surface with

$$E = \widehat{E}_0 \exp(i \mathbf{k}_0 \cdot \mathbf{r}) \quad (20.59)$$

An outgoing plane wave has a field

$$E = \widehat{E}_1 \exp(i \mathbf{k}_1 \cdot \mathbf{r}) \quad (20.60)$$

A simulation of this condition will start with a number of photons with wave vector \mathbf{k}_0 and electric field E . After reflecting from the surface, the photons will have wave vector \mathbf{k}_1 . Now imagine a set of N photons that flow through an area dA_0 before being reflected from the surface.

Since the electric field is \widehat{E}_0 , for incoherent photon tracking

$$\widehat{E}_0^2 = \frac{\alpha_p E_0^2 N}{dA_0} \quad (20.61)$$

where α_p is the simulation constant (cf. Eq. (20.2)). After the photons are reflected they will have some field E_1 and thus

$$\widehat{E}_1^2 = \frac{\alpha_p E_1^2 N}{dA_1} \quad (20.62)$$

Where dA_1 is the area that the photons flow through which is related to dA_0 via

$$\frac{dA_1}{dA_0} = \frac{\mathbf{k}_1 \cdot \mathbf{z}}{\mathbf{k}_0 \cdot \mathbf{z}} \equiv |b| \quad (20.63)$$

Combining the above three equations, the change in field for a photon as it reflects from the surface is

$$\frac{E_1}{E_0} = \frac{\widehat{E}_1}{\widehat{E}_0} \sqrt{|b|} \quad \text{Incoherent} \quad (20.64)$$

For coherent photon tracking the electric field at dA_0 is

$$\widehat{E}_0 = \frac{\alpha_p E_0 N}{dA_0} \quad (20.65)$$

After the photons are reflected they will have some field E_1 and thus

$$\widehat{E}_1 = \frac{\alpha_p E_1 N}{dA_1} \quad (20.66)$$

Combining these equations the change in field for a photon as it reflects from the surface is

$$\frac{E_1}{E_0} = \frac{\widehat{E}_1}{\widehat{E}_0} |b| \quad \text{Coherent} \quad (20.67)$$

Additionally, for coherent tracking, all photons in a plane wave must have the same phase when passing through an area transverse to the wave. Thus the two photons labeled a and b in Fig. 20.4 must have the same phase advance in going from dA_0 to dA_1 . The difference in the phase advance for photon b

relative to a from dA_0 to the surface is $\mathbf{k}_0 \cdot \mathbf{r}$ where \mathbf{r} is the vector between where photon b hits the surface relative to photon a . Similarly, the difference in the phase advance for photon b relative to a from the surface to dA_0 is $-\mathbf{k}_1 \cdot \mathbf{r}$. Since the total phase advance for both photons is the same from dA_0 to dA_1 the phase shift $d\phi_b$ of photon b as it is reflected from the surface relative to the phase shift $d\phi_a$ is

$$d\phi_b = d\phi_a - (\mathbf{k}_1 - \mathbf{k}_0) \cdot \mathbf{r} \quad (20.68)$$

This shift in the reflection phase can be related to the lattice diffraction planes. The wave vector difference can be written

$$\mathbf{k}_1 - \mathbf{k}_0 = \mathbf{H} + q \hat{\mathbf{n}} \quad (20.69)$$

where $\hat{\mathbf{n}}$ is perpendicular to the surface. Combining Eqs. (20.68) and (20.69) and since \mathbf{r} is in the plane of the surface

$$d\phi_b = d\phi_a - \mathbf{H} \cdot \mathbf{r} \quad (20.70)$$

This shows that the reflection shift has the same periodicity as the pattern of the lattice planes at the surface of the crystal. Notice that for a mirror, where one point on the surface is the same as any other, $d\phi_b$ must be equal to $d\phi_a$. Using this in Eq. (20.68) gives

$$\mathbf{k}_1 \cdot \mathbf{r} = \mathbf{k}_0 \cdot \mathbf{r} \quad (20.71)$$

and since $|\mathbf{k}_1| = |\mathbf{k}_0|$ this proves that the angle of incidence is equal to the angle of reflection for a mirror.

In practice, the registration of the surface planes with respect to the surface is not specified in a simulation. Thus the reflection phase shift can only be calculated up to a constant offset.

20.4.5 Bragg Crystal Tracking

The starting photon coordinates are specified in the laboratory entrance coordinates. The transformation from laboratory entrance coordinates to element entrance coordinates $\tilde{\mathbf{k}}_0$ is given in §20.3. The transformation to element surface coordinates \mathbf{k}_0 is

$$\mathbf{k}_0 = \Sigma_2 \tilde{\mathbf{k}}_0 \quad (20.72)$$

with Σ_2 given by Eq. (20.46). The outgoing wave vector \mathbf{k}_H is related to \mathbf{k}_0 via

$$\mathbf{k}_H = \mathbf{k}_0 + \mathbf{H} + q_t \hat{\mathbf{n}} \quad (20.73)$$

where q_t is determined by using Eqs. (20.33) and (20.34) in Eq. (20.29)

$$\begin{aligned} k_{H,x} &= k_{0,x} + H_x \\ k_{H,y} &= k_{0,y} + H_y \\ k_{H,z} &= \sqrt{\lambda^2 - k_{H,x}^2 - k_{H,y}^2} \end{aligned} \quad (20.74)$$

To compute the field amplitude of the outgoing photon, the equation to be solved is ([Bater64] Eq. (21))

$$\xi_0 \xi_H = \frac{1}{4} k^2 P^2 \Gamma^2 F_H F_{\bar{H}} \quad (20.75)$$

where ξ_0 and ξ_H are given by [Bater64] Eq. (18) and P is the polarization factor

$$P = \begin{cases} 1 & \sigma \text{ polarization state} \\ \cos 2\theta_g & \pi \text{ polarization state} \end{cases} \quad (20.76)$$

$2\theta_g$ is the angle between \mathbf{K}_0 and \mathbf{K}_H which is well approximated by $\theta_{B,in} + \theta_{B,out}$.

The solution to Eq. (20.75) is ([Bater64] Eq. (31))

$$\begin{aligned}\xi_0 &= \frac{1}{2} k |P| \Gamma [F_H F_{\bar{H}}]^{1/2} |b|^{1/2} [\eta \pm (\eta^2 + \text{sgn}(b))^{1/2}] \\ \xi_H &= \frac{1}{2} k |P| \Gamma [F_H F_{\bar{H}}]^{1/2} \frac{1}{|b|^{1/2} [\eta \pm (\eta^2 + \text{sgn}(b))^{1/2}]}\end{aligned}\quad (20.77)$$

where the + part of \pm is for the α branch and the - part of \pm is for the β branch and sgn is the sign function

$$\text{sgn}(b) \equiv \begin{cases} 1 & b > 0 \\ -1 & b < 0 \end{cases}\quad (20.78)$$

and η is given by [Blas94] Eq. (5)

$$\eta = \frac{-ba + \Gamma F_0 (1-b)}{2 \Gamma |P| \sqrt{|b| F_H F_{\bar{H}}}}\quad (20.79)$$

with the asymmetry factor b for the photon being tracked being given by [Blas94] Eq. (3)

$$b \equiv \frac{\hat{\mathbf{n}} \cdot \hat{\mathbf{k}}_0}{\hat{\mathbf{n}} \cdot (\hat{\mathbf{k}}_0 + \hat{\mathbf{H}})}\quad (20.80)$$

and the angular deviation variable a is given by [Blas94] Eq. (4)

$$a \equiv \frac{H^2 + 2 \mathbf{k}_0 \cdot \mathbf{H}}{k_0^2} = -2 \Delta\theta \sin(2\theta_B)\quad (20.81)$$

Once ξ_0 and ξ_H are determined, the ratio of the incoming and outgoing fields for the α or β branches can be computed via ([Bater64] Eq. (24))

$$r_E \equiv \frac{\mathbf{E}_H}{\mathbf{E}_0} = \frac{-2\xi_0}{k P \Gamma F_{\bar{H}}} = \frac{-k P \Gamma F_H}{2\xi_H}\quad (20.82)$$

where the α or β subscript has been suppressed. The total field which is the sum of the fields on the branches is computed using the boundary conditions

$$\mathbf{E}_0 = \mathbf{E}_{0\alpha} + \mathbf{E}_{0\beta}, \quad 0 = \mathbf{E}_{H\alpha} + \mathbf{E}_{H\beta}\quad (20.83)$$

Using the above two equations gives

$$\begin{aligned}\mathbf{E}_{0\alpha} &= \mathbf{E}_0 \frac{r_{E\beta}}{r_{E\beta} - r_{E\alpha}} & \mathbf{E}_{H\alpha} &= \mathbf{E}_0 \frac{r_{E\alpha} r_{E\beta}}{r_{E\beta} - r_{E\alpha}} \\ \mathbf{E}_{0\beta} &= -\mathbf{E}_0 \frac{r_{E\alpha}}{r_{E\beta} - r_{E\alpha}} & \mathbf{E}_{H\beta} &= -\mathbf{E}_0 \frac{r_{E\alpha} r_{E\beta}}{r_{E\beta} - r_{E\alpha}}\end{aligned}\quad (20.84)$$

As can be seen from Battermann and Cole Figs. (8) and (29), the α tie point is excited and the β tie point is not if $\xi_{0\alpha} < \xi_{0\beta}$ and vice versa. Since only one tie point is excited, The external field ratio is equal to the internal field ratio

$$\frac{E_H^e}{E_0^i} = \frac{E_{Hj}}{E_{0j}}\quad (20.85)$$

where j is α or β as appropriate.

20.4.6 Coherent Laue Crystal Tracking

Laue diffraction has two interior wave fields (branches), labeled α and β , corresponding to the two tie points that are excited on the two dispersion surfaces. For coherent tracking, a photon has some probability to be channeled to follow the α or β branch. The electric field ratios \widehat{E}_α and \widehat{E}_β (cf. Eq. (20.11)) are taken to be equal to each other. With this choice, the probabilities P_α and P_β for being channeled to the α or β branches are such that a branch with a greater intensity will have a greater number of photons channelled down it.

When a crystal's `ref_orbit_follows` parameter is set to `bragg_diffracted`, The branching probabilities are

$$P_\alpha = \frac{|E_{H\alpha}|}{|E_{H\alpha}| + |E_{H\beta}|}, \quad P_\beta = \frac{|E_{H\beta}|}{|E_{H\alpha}| + |E_{H\beta}|}, \quad \widehat{E}_{H\alpha} = \widehat{E}_{H\beta} = \frac{|E_{H\alpha}| + |E_{H\beta}|}{|E_0^i|} \quad (20.86)$$

where (see Batermann and Cole[Bater64] Eqs (42)),

$$\begin{aligned} E_{H\alpha} &= -E_0^i \frac{|b|^{1/2}}{2 \cosh v} \frac{|P|}{P} \frac{[F_H F_{\bar{H}}]^{1/2}}{F_H} \exp(-2\pi i \mathbf{K}'_{H\alpha} \cdot \mathbf{r}_\alpha) \exp(-2\pi \mathbf{K}''_{H\alpha} \cdot \mathbf{r}_\alpha) \\ E_{H\beta} &= E_0^i \frac{|b|^{1/2}}{2 \cosh v} \frac{|P|}{P} \frac{[F_H F_{\bar{H}}]^{1/2}}{F_H} \exp(-2\pi i \mathbf{K}'_{H\beta} \cdot \mathbf{r}_\beta) \exp(-2\pi \mathbf{K}''_{H\beta} \cdot \mathbf{r}_\beta) \end{aligned} \quad (20.87)$$

where \mathbf{r}_α and \mathbf{r}_β are the vectors from the entrance surface to the exit surface for the α and β wave fields

$$\mathbf{r}_\alpha = \frac{t^2}{\mathbf{S}_\alpha \cdot \mathbf{t}} \mathbf{S}_\alpha, \quad \mathbf{r}_\beta = \frac{t^2}{\mathbf{S}_\beta \cdot \mathbf{t}} \mathbf{S}_\beta \quad (20.88)$$

with

$$\begin{aligned} \mathbf{S}_\alpha &= e^{-2v} \mathbf{s}_0 + \left| b \frac{F_H F_{\bar{H}}}{F_H^2} \right| \mathbf{s}_H \\ \mathbf{S}_\beta &= e^{2v} \mathbf{s}_0 + \left| b \frac{F_H F_{\bar{H}}}{F_H^2} \right| \mathbf{s}_H \end{aligned} \quad (20.89)$$

The phase shift of the electric field is obtained from the phase of $E_{H\alpha}$ if the photon is channeled into the α branch and $E_{H\beta}$ if the photon is channeled into the β branch.

When a crystal's `ref_orbit_follows` parameter is set to `forward_diffracted` or `undefracted`, the algorithm is similar to the `bragg_diffracted` case except $E_{0\alpha}$ and $E_{0\beta}$ are used in place of $E_{H\alpha}$ and $E_{H\beta}$ with

$$\begin{aligned} E_{0\alpha} &= E_0^i \frac{e^{-v}}{2 \cosh v} \exp(-2\pi i \mathbf{K}'_{0\alpha} \cdot \mathbf{r}_\alpha) \exp(-2\pi \mathbf{K}''_{0\alpha} \cdot \mathbf{r}_\alpha) \\ E_{0\beta} &= E_0^i \frac{e^{-v}}{2 \cosh v} \exp(-2\pi i \mathbf{K}'_{0\beta} \cdot \mathbf{r}_\alpha) \exp(-2\pi \mathbf{K}''_{0\beta} \cdot \mathbf{r}_\beta) \end{aligned} \quad (20.90)$$

Since a simulation photon has two polarization components, the above equations are used for one polarization component and for the second polarization component the same branch is used as for the first with an appropriately scaled \widehat{E} .

20.4.7 Incoherent Laue Crystal Tracking

Laue diffraction has two interior wave fields (branches), labeled α and β , corresponding to the two tie points that are excited on the two dispersion surfaces. For incoherent tracking it is assumed that

these wave fields overlap at the exit surface ([Bater64] Eq. (87)) and so add coherently. This is a good approximation if the crystal is very thin where the wave fields do not travel an appreciable spatial distance and is also a good approximation when the crystal is thick since the β branch will be heavily attenuated. At intermediate thicknesses this approximation is good when a photon is near the Bragg angle since, in this case, the fields will be traveling in similar directions.

Another approximation is that the path

$$\begin{aligned} E_{H\alpha} &= -E_0^i \frac{|b|^{1/2}}{2 \cosh v} \frac{|P|}{P} \frac{[F_H F_{\bar{H}}]^{1/2}}{F_H} \exp(-2\pi i \mathbf{K}'_{H\alpha} \cdot \mathbf{r}_\alpha) \exp(-2\pi \mathbf{K}''_{H\alpha} \cdot \mathbf{r}_\alpha) \\ E_{H\beta} &= E_0^i \frac{|b|^{1/2}}{2 \cosh v} \frac{|P|}{P} \frac{[F_H F_{\bar{H}}]^{1/2}}{F_H} \exp(-2\pi i \mathbf{K}'_{H\beta} \cdot \mathbf{r}_\beta) \exp(-2\pi \mathbf{K}''_{H\beta} \cdot \mathbf{r}_\beta) \end{aligned} \quad (20.91)$$

20.5 X-ray Targeting

X-rays can have a wide spread of trajectories resulting in many “doomed” photons that hit apertures or miss the detector with only a small fraction of “successful” photons actually contributing to the simulation results. The tracking of doomed photons can therefore result in an appreciable lengthening of the simulation time. To get around this, *Bmad* can be setup to use what is called “targeting” to greatly reduce the number of doomed photons generated.

Photons can be generated either at a source like a `wiggler` element or at a place where diffraction is simulated like at a `diffraction_plate` element. To be able to do targeting, an element with apertures defined must be present downstream from the generating element. The idea is to only generate photons that are going in the general direction of the “target” which is the space within the aperture.

A necessary restriction for targeting to work is that the photon must travel in a straight line through all elements between the generating element and the element with the apertures. So, for example, a `crystal` element would not be allowed between the two two elements. A `crystal` element could be the aperture element as long as the aperture was defined before photons were diffracted. That is, if the aperture was at the upstream end of the crystal or was defined with respect to the `crystal` surface.

The target is defined by the four corners of the aperture. In `element` coordinates, the (x, y, z) values of the corners are:

```
(-x1_limit, -y1_limit, z_lim)
(-x1_limit, y2_limit, z_lim)
( x2_limit, -y1_limit, z_lim)
( x2_limit, y2_limit, z_lim)
```

where `x1_limit`, etc. are the the aperture limits (§4.6) and `z_lim` will be zero except if the element’s `aperture_at` parameter is set to `entrance_end` in which case `z_lim` will be set to `-L` where `L` is the length of the element.

If the aperture is associated with a curved surface (for example with a `crystal` element), four extra corner points are also used to take into account that the aperture is not planer. These extra points have (x, y, z) values in `element` coordinates of

```
(-x1_limit, -y1_limit, z_surface(-x1_limit, -y1_limit))
(-x1_limit, y2_limit, z_surface(-x1_limit, y2_limit))
( x2_limit, -y1_limit, z_surface( x2_limit, -y1_limit))
( x2_limit, y2_limit, z_surface( x2_limit, y2_limit))
```

where `z_surface(x,y)` is the `z` value of the surface at the particular (x, y) point being used. Notice that in this case `z_lim` is zero.

The coordinates of the four or eight corner points are converted from **element** coordinates of the aperture element to **element** coordinates of the photon generating element. Additionally, the approximate center of the aperture, which in **element** coordinates of the aperture element is $(0, 0, z_{lim})$, is converted to **element** coordinates of the photon generating element.

The above calculation only has to be done once at the beginning of a simulation.

When a photon is to be emitted from a given point $(x_{emit}, y_{emit}, z_{emit})$, the problem is how to restrict the velocity vector $(\beta_x, \beta_y, \beta_z)$ (which is normalized to 1) to minimize the number of doomed photons generated. The problem is solved by constructing a vector \mathbf{r} for each corner point:

$$\mathbf{r} = (x_{lim}, y_{lim}, z_{lim}) - (x_{emit}, y_{emit}, z_{emit}) \quad (20.92)$$

The direction of each \mathbf{r} is characterized in polar coordinates (ϕ, y) defined by

$$\begin{aligned} y &= \frac{r_y}{|\mathbf{r}|} \\ \tan \phi &= \frac{r_x}{r_z} \end{aligned} \quad (20.93)$$

For now make the assumption that r_z is positive and larger than r_x and r_y for all \mathbf{r} . Let ϕ_{max} and ϕ_{min} be the maximum and minimum ϕ values over all the \mathbf{r} . Similarly, let y_{min} and y_{max} be the minimum and maximum y values over all the \mathbf{r} . The rectangle in (ϕ, y) space defined by these four min and max values almost covers the projection of the aperture onto the unit sphere. There is a correction that must be made due to the fact that a straight line of constant y in (x, y, z) space projects to a curved line when projected onto (ϕ, y) space. Therefore a correction must be made to y_{min} when $y_{min} < 0$:

$$y_{min} \rightarrow \frac{y_{min}}{\sqrt{(1 - y_{min}^2) \cos^2(\phi_{max} - \phi_{min})/2 + y_{min}^2}} \quad (20.94)$$

with a similar correction for y_{max} that must be made when $y_{max} > 0$.

The above prescription works as long as the projection of the aperture onto (ϕ, y) space does not touch the branch cut at $\phi = \pi$ or cover the singular points $y = \pm 1$. Generally these restrictions are fulfilled since **z** is the direction of the reference orbit. If this is not the case, a transformation can be made where rotation matrices are constructed to transform between the **element** coordinates of the emitting element and what are called **target** coordinates defined so that \mathbf{r} for the **center** point has the form $(0, 0, |\mathbf{r}|)$. The procedure for calculating the photon velocity vector is now

1. Rotate all the corner \mathbf{r} from **element** to **target** coordinates.
2. Calculate min and max values for ϕ and y .
3. Calculate the velocity vector such that the (ϕ, y) of this vector falls within the min and max values in the last step.
4. Rotate the velocity vector back to **element** coordinates.

Chapter 21

Simulation Modules

In the *Bmad* “ecosystem”, various modules have been developed to simulate machine hardware. This chapter provides documentation.

21.1 Tune Tracker Simulator

[Tune tracker simulation developed by Michael Ehrlichman]

The digital tune tracker (dT) is device for determining the fractional tune of a storage ring and exciting resonant beam oscillations. In short, the dTT starts with a beam position monitor (BPM) to detect the beam oscillations. The signal from the BPM is feed into a phase-locked loop (PLL) circuit which oscillates at the oscillation frequency the beam. A signal from the PLL is used to modulate a kicker to keep the beam oscillating at the beam’s oscillation frequency. The tune tracker is capable of exciting both betatron and synchrotron oscillations.

In general, a PLL is a control system for matching the frequency of a VCO to some incoming periodic reference signal. It does this by adjusting the frequency of the VCO according to the phase difference between the VCO output and the reference signal. A general diagram of a PLL is shown in Fig. 21.1.

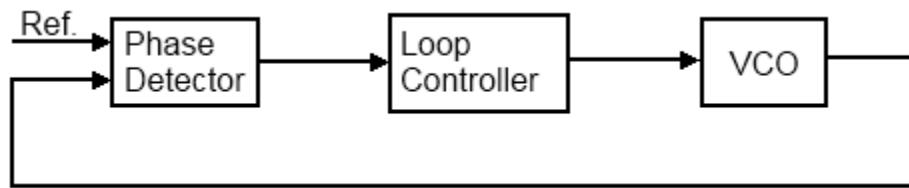


Figure 21.1: General diagram of a phase lock loop. The loop adjusts the speed of a VCO according to the phase difference between the VCO and an incoming signal.

In Fig. 21.1, the phase detector compares the two incoming signals, and outputs a signal that is proportional to the difference in phase. There are various ways to implement a phase detector. The tune tracker phase detector is a mixer followed by low-pass filter, which will be discussed in more detail below. The loop controller, also known as a loop filter, is usually some combination of (P)roportional, (I)ntegration, and (D)ifferentiation paths. The loop controller sums these paths, and the resulting signal is sent to the VCO. The output of the loop controller rises when the VCO is slower than the reference, and drops when

the VCO is faster. The gains of the three PID paths are adjusted to produce a control loop that is stable and can efficiently lock onto the reference signal from some incorrect initial frequency. A real-world PLL also needs to track perturbations to the reference frequency.

A diagram of the digital tune tracker is shown in Fig. 21.2. This function takes in one new BPM measurement per call, and returns a sinusoidal signal that has the same frequency as the BPM signal. The phase of the returned signal differs from the phase of the BPM signal by ϕ_0 , the phase advance from position at the BPM to position at the kicker on the next turn.

Comparing this diagram to the PLL in Fig. 21.1, the boxes from *Subtract closed orbit offset* to the end of the *Low Pass Filter* comprise the phase detector. After the *Low Pass Filter* to just before the *Gain Kvco* box comprises the loop controller. The *Gain Kvco* box comprises the VCO. The feedback loop consists of the update to ω that follows *Gain Kvco*.

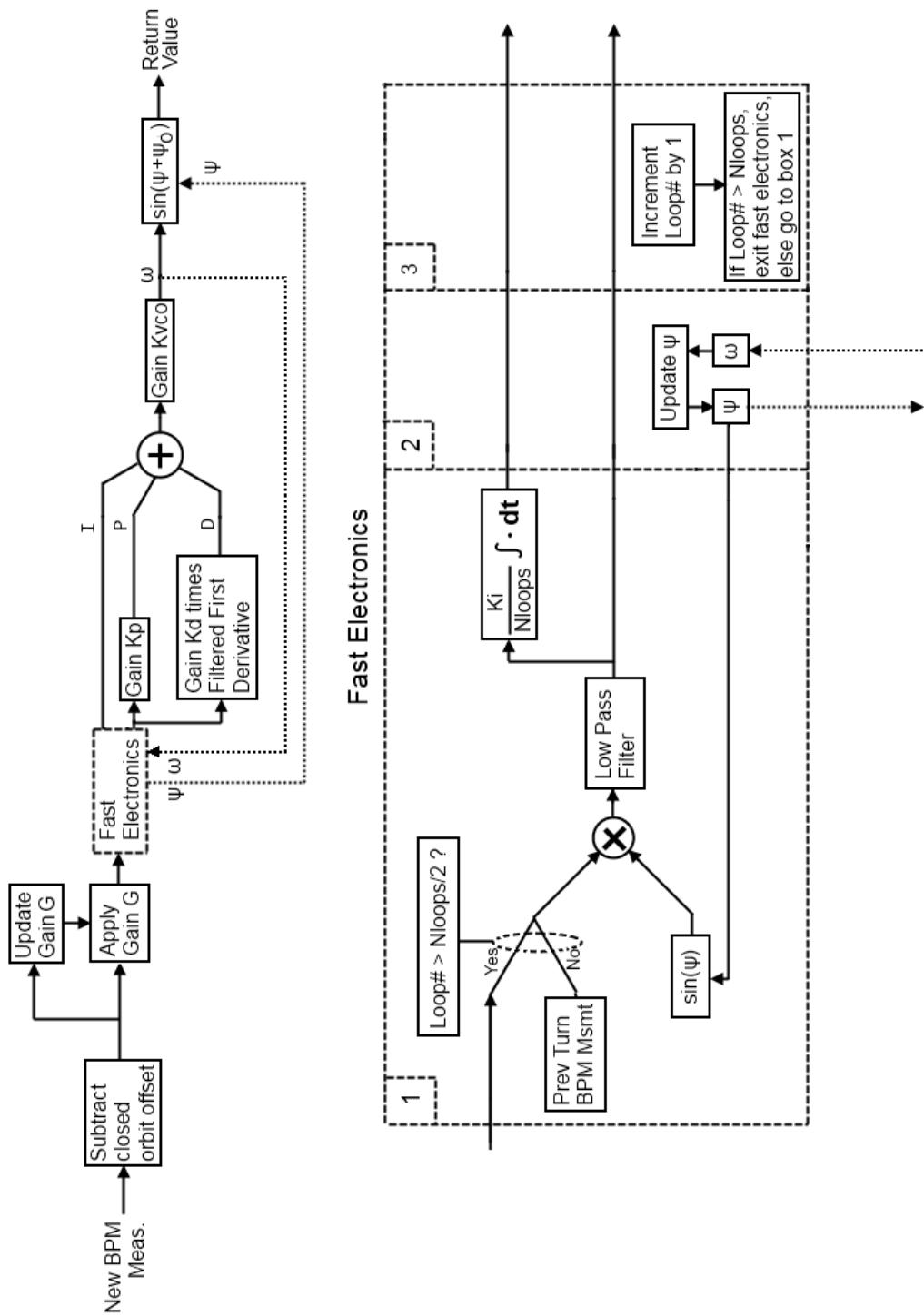


Figure 21.2: Flow chart of tune tracker module functions.

21.1.1 Tune Tracker Components.

New BPM Meas. This consists of one BPM measurement. It is either horizontal or vertical position at the BPM location. It is assumed that each measurement is separated by the same Δt , and that Δt is an integer multiple of the ring period.

Subtract closed orbit offset Due to misalignments or the presence of wigglers, the closed orbit at the BPM may be non-zero. The closed orbit is subtracted from the incoming data. The closed orbit offset is a constant parameter set by the user during initialization.

Apply Gain G This block adjusts the incoming measurement such that,

$$\text{Meas. Out} = \frac{\text{Meas. In}}{G}$$

The purpose of this block is to normalize the BPM measurements. When the tune tracker first starts, the amplitude of the beam oscillations is small. When the tune tracker is locked on, the amplitude of the beam oscillations becomes large. The set of loop controller gains that allows the tune tracker to quickly lock onto the initially small signal are not optimal when the beam oscillations are large. This normalization overcomes that problem, allowing for a set of loop controller gains that are optimal both at start up and when oscillations are large.

Update Gain G This is a digital filter of the form,

$$G = g_a \text{abs}(x) + (1 - g_a) G,$$

which is a exponentially weighted moving average, or equivalently, a low-pass filter. x is the incoming data point. g_a is a hardcoded time constant. It is set so that G equilibrates to the rms beam size after about 10 periods.

Fast Electronics Fast electronics contains the components of the tune tracer that cycle every few RF periods. Whereas the rest of the tune tracker cycles once per storage ring period, the fast electronics cycle every few RF buckets.

Prev Turn BPM Msmt Stores the previous turn's BPM measurement.

Loop# > $N_{loops}/2$? N_{loops} is the number of times the fast electronics cycle every storage ring period. For the first half of these loops, the BPM measurement from the previous turn is used. For the last half, the current BPM measurement is used.

sin(ψ) Sinusoid of the current modulator angle ψ .

ψ Current modulator angle. Updated at the end of every fast electronics loop.

\times (times) The BPM measurement and sin(ψ) are multiplied. Let the BPM signal be parameterized as $\cos(\omega_{beam}t)$, and write ψ as $\omega_{vco}t$ so we have,

$$\cos(\omega_{beam}t) \sin(\omega_{vco}t).$$

A trig identity allows this to be written as,

$$\frac{1}{2} (\sin((\omega_{beam} + \omega_{vco})t) - \sin((\omega_{beam} - \omega_{vco})t)).$$

If ω_{vco} is roughly ω_{beam} , then the first sin term has roughly twice the beam frequency and the second term has a very small frequency. Therefore, passing the multiplied signal through a low-pass filter removes the first term, leaving only the second term.

In stable operation, ω_{vco} makes small amplitude oscillations about ω_{beam} . In this case, the time integral of $(\omega_{beam} - \omega_{vco})t$ is small for all t , and a small angle approximation for sine can be applied, and the output of the low pass filter can be written as,

$$(\omega_{beam} - \omega_{vco})t, \quad (21.1)$$

which is proportional to the phase difference between the beam and the VCO. Thus completes the phase detector.

$\frac{K_I}{N_{loops}} \int \cdot dt$ This is the (I)ntegrated channel of the PID controller. It integrates the output of the phase detector, providing negative feedback. If the integrated phase is positive, the I channel slows the VCO down. If negative, it speeds the VCO up. This channel is expected to equilibrate to some non-zero value proportional to the difference between the beam frequency and the VCO base frequency.

Update ψ This increments the modulator phase according to,

$$\psi_{n+1} = \psi_n + \frac{T_{ring}}{N_{loops}} \omega_{vco}. \quad (21.2)$$

In order to more accurately model things, ω_{vco} is not updated within the fast electronics loop.

Increment Loop# by 1 Loop# counts the number of times the fast electronics has been looped through.

If Loop# > N_{loops} , exit, else, loop N_{loops} is the number of times the fast electronics cycle per ring period.

Gain K_P This is the (P)roportional channel of the PID controller. It provides negative feedback proportional to the output of the phase detector. This channel responds faster than the integrated channel and so helps the tune tracker track sudden changes in the frequency of the beam. In practice, it also damps tune tracker oscillations. If K_P is too small, the VCO may oscillate indefinitely around the fractional tune of the beam. See section §21.1.2 below for more details.

Gain $K_D \times$ filtered derivative This is the (D)ifferential channel of the PID controller. Provides a filtered first derivative of the output of the phase detector. The derivative is obtained a polynomial fitted to a history of the phase detector output. This channel can be used to smooth out the VCO response and reduce overshoot. See section §21.1.2 below for more details.

+ (sum) This block sums the three output of the PID controller.

Gain K_{vco} This block represents the response of a voltage controlled oscillator. The frequency of the output of a VCO is proportional to the voltage of its input. This block simply multiplies the output of the PID controller by a constant. This block updates the VCO frequency.

sin ($\psi + \psi_0$) Calculates kicker amplitude to be returned to the calling program. This is the kick amplitude that should be used during the next turn. ψ_0 is the phase advance from position at the BPM to position at next turn's kicker.

21.1.2 Tuning

Tuning the tune tracker consists of adjusting the four gain parameters K_P , K_I , K_D , and K_{vco} . The goal is to set the parameters such that the tune tracker quickly locks onto the fractional tune of the beam, and settles quickly. If multiple tunes are being explored, then the size of the convergence region is also

Parameter	Rise Time	Overshoot	Settling Rate	Precision	Stability Region
K_P	slight decrease	decrease	faster	degrade	shrink
K_I	decrease	increase	slower	no change	shrink
K_D ^{see note}	slight increase	slight decrease	slightly faster	degrade	grow

Table 21.1: Effect on VCO response of increasing K_P , K_I , or K_D . Note: The effects of the derivative channel are theoretical and have not been verified extensively.

important. i.e. The set of parameters should work for a wide range of fractional tunes. If jitter is being explored, then the ability of the tune tracker to track a varying fractional tune is also important.

A plot of the VCO speed during typical tune tracker operation is shown in Fig. 21.3. A tune tracker with a base period of 0.571 oscillations per ring period tracks a beam with a fractional tune of 0.5691. Marked on this plot are 4 important features. They are the rise time, overshoot, settling rate, and precision. The rise time is how long it takes the tune tracker to approach the fractional tune of the beam. This is the location of the first peak after operation begins. The overshoot is the difference between the amplitude of the first peak and the fractional tune. The settling rate measures the decay rate of the oscillations of the VCO about the fractional tune. The precision is the noise in the VCO frequency. The discrete nature of the BPM measurements introduces noise at the phase detector stage. The proportional channel adds this noise to the VCO frequency. Shown in Tab. [21.1] is the effect increasing the PID controller gains.

Stability region refers to the range of fractional tunes that the tune tracker will converge to when starting from an initial tune in that range. In practice, if K_P and K_I are increased to optimize response to a particular fractional tune, the the stability region will shrink. e.g. A large K_P and K_I could be found that very quickly lock on to a fractional tune of 0.569, but are unstable for 0.622. But note that a smaller K_P and K_I can be found that give acceptable performance at 0.569, and also acceptable performance at 0.622 (and even 0.074).

21.1.3 Programmer Instructions

This section contains instructions on how to implement the tune tracker module and how to use the tune tracker driver program. The tune tracker is written as an object or class, and so supports multiple instances.

After setting up the tune tracker in a program, it is best to first verify that the tune tracker is efficiently locking onto the beam and that the beam oscillations are indeed growing with time. The tune tracker should lock on after about 10,000 turns, and the beam oscillations should reach equilibrium amplitude after about 10 damping periods.

See the tune tracker section in the Physics chapter of this manual for an example of how the VCO frequency should respond with time in a successful implementation. Guidelines for setting the gain parameters are also located in the Physics chapter.

21.1.4 Tune Tracker Module

The tune tracker module has four public functions. They are,

```
Function id = init_dTT(tt_param_struct) !Constructor, creates new tune tracker  
!instance, handles initialization.  
!Passed tt_param_struct, Returns id of  
!created instance.
```

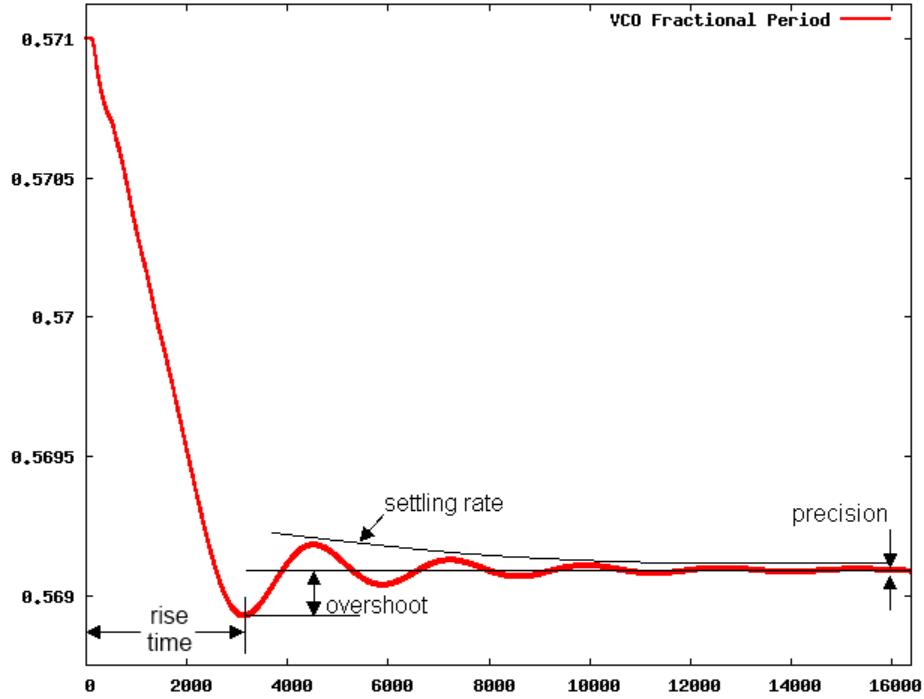


Figure 21.3: Plot of VCO response of typical tune tracker setup. VCO base frequency is 0.571. Beam fractional tune is 0.5691.

```

Function z = TT_update(bpm_msmt,id)           !Main function, passed one new BPM
                                               !measurement and returns VCO modulator
                                               !amplitude.
Subroutine dest_dTT(id)                      !Destructor, closes log files and
                                               !deallocates memory.
                                               !Takes as argument id of instance
                                               !to be destroyed.
Function z = get_dTT(name,id)                 !Get function, name is either 'wf',
                                               !in which case the function returns
                                               !the VCO frequency  $\omega_0 + \Delta\omega$ ,
                                               !or 'dw', in which case just  $\Delta\omega$ 
                                               !is returned.

```

Pseudocode for a tune tracker implementation is below.

```

Populate tt_param_struct with tune tracker parameters.
id = init_dTT(tt_param_struct)
Loop over turns:
  Track once through the ring, obtain orbit.
  bpm_msmt = x or y at BPM location.
  z = TT_update(bpm_msmt,id)
  Adjust kicker amplitude (or cavity phase) according to z
dest_dTT(id)

```

See `tune_tracker.f90` for the contents of `tt_param_struct`.

If a horizontal or vertical tune tracker is to be implemented, the BPM measurements should be the x or y coordinate at the BPM location, and the kicks should be changes in x' or y' at the kicker location. If a longitudinal tune tracker is to be implemented the BPM measurements should be the x coordinate in a dispersive region, and the kicks should be changes in phase at an RF cavity. Similarly, if a longitudinal tune tracker is being used in conjunction with a horizontal tune tracker, the BPM of the horizontal tune tracker should be placed in a region of low dispersion. This will prevent the horizontal tune tracker from inadvertently locking onto the synchrotron tune.

21.1.5 Tune Tracker Example Program

An example program is located in `src/examples/tune_tracker`. This program contains code for implementing simultaneous horizontal, vertical, and longitudinal tune trackers.

For most implementations of the tune tracker module, a good approach would be to either use a modified version of the tune tracker example program, or copy and paste code from the example program. Keep in mind that the example program implements multiple simultaneous tune trackers. An implementation of just one tune tracker would be much simpler.

The example program performs the following steps.

1. Read in file which contains tune tracker parameters such as gains.
2. Parse lattice with `twiss3` subroutines. The `twiss3` subroutines are needed to obtain the longitudinal phase advance when implementing longitudinal tune trackers.
3. Check that the kick attribute of the kicker element is actually free.
4. Calculate phase advance from position at BPM to position at kicker. Note that this is the phase of the kicker *on the next turn*.
5. Creates log files which record position at BPM and kicker, the VCO frequency, and slope at the kicker.
6. Loops over turns, passing BPM measurements to `TT_update` and adjusting kicker amplitudes based on the returned value. Note that it will take the beam oscillations several damping times to equilibrate. This often means tracking for several 10,000 turns. Use of the save state parameter may shorten simulation times in certain applications.
7. Calculates and FFT of the BPM data and records in log file.

Also located in `src/examples/tune_tracker` is an example .in file that implements horizontal, vertical, and longitudinal tune trackers. The gains used in the .in file were selected for their large convergence region. More aggressive tuning may lead to faster lock on. See the Physics section for details.

21.1.6 Save States

If `tt_param_struct%useSaveState` is set to true, then the destructor `dest_dTT` will save the tune tracker state variables to a file called `tt_state.#`. Similarly, the constructor `init_dTT` will set the state variables according to the save state file. The constructor will also return the beam coordinates to be used at element zero of the ring.

Save states allow you to pick up where you left off, and not have to wait for the beam oscillations to equilibrate. For example, you could have one program that tracks for 100,000 turns, allowing the tune tracker to lock on and the beam excitation to equilibrate with radiation damping. The simulation writes a save state file which can be used as a starting point for other simulations that perform studies on the excited beam.

21.2 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 §4.17 and the conversion formulas are outlined below.

21.2.1 Orbit Measurement

For orbits, the relationship between measured position $(x, y)_{\text{meas}}$ and true position $(x, y)_{\text{true}}$ is

$$\begin{pmatrix} x \\ y \end{pmatrix}_{\text{meas}} = n_f \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} + \mathbf{M}_m \left[\begin{pmatrix} x \\ y \end{pmatrix}_{\text{true}} - \begin{pmatrix} x \\ y \end{pmatrix}_0 \right] \quad (21.3)$$

with

$$\begin{pmatrix} x \\ y \end{pmatrix}_0 = \begin{pmatrix} x_{\text{err}} - x_{\text{cal}} \\ y_{\text{err}} - y_{\text{cal}} \end{pmatrix} \quad (21.4)$$

and

$$\mathbf{M}_m = \begin{pmatrix} g_x \cos(d\theta + d\psi) & g_x \sin(d\theta + d\psi) \\ -g_y \sin(d\theta - d\psi) & g_y \cos(d\theta - d\psi) \end{pmatrix} \quad (21.5)$$

where

$$\begin{aligned} d\psi &= \psi_{\text{err}} - \psi_{\text{cal}} \\ d\theta &= \theta_{\text{err}} - \theta_{\text{cal}} \\ g_x &= 1 + dg_{x,\text{err}} - dg_{x,\text{cal}} \\ g_y &= 1 + dg_{y,\text{err}} - dg_{y,\text{cal}} \end{aligned} \quad (21.6)$$

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. θ_{err} and ϕ_{err} are error tilt and “crunch” angles, and θ_{cal} and ϕ_{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.

21.2.2 Dispersion Measurement

A dispersion measurement is considered to be the result of measuring the orbit at two different energies. The measured values are then

$$\begin{pmatrix} \eta_x \\ \eta_y \end{pmatrix}_{\text{meas}} = \frac{\sqrt{2} n_f}{dE/E} \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} + \mathbf{M}_m \begin{pmatrix} \eta_x \\ \eta_y \end{pmatrix}_{\text{true}} \quad (21.7)$$

The factor of $\sqrt{2}$ comes from the fact that there are two measurements.

21.2.3 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_{osc} . The measured coupling is computed as follows. First, consider excitation of the a -mode which can be written in the form:

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix}_{\text{true}} = A_{\text{osc}} \begin{pmatrix} \cos \phi_i \\ K_{22a} \cos \phi_i + K_{12a} \sin \phi_i \end{pmatrix}_{\text{true}} \quad (21.8)$$

i is the turn number and ϕ_i is the oscillation phase on the i^{th} turn. The coefficients K_{22a} and K_{12a} are related to the coupling $\bar{\mathbf{C}}$ via David and Rubin [Sagan99] Eq. 54:

$$\begin{aligned} K_{22a} &= \frac{-\sqrt{\beta_b}}{\gamma \sqrt{\beta_a}} \bar{\mathbf{C}}_{22} \\ K_{12a} &= \frac{-\sqrt{\beta_b}}{\gamma \sqrt{\beta_a}} \bar{\mathbf{C}}_{12} \end{aligned} \quad (21.9)$$

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:

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix}_{\text{true}} = \begin{pmatrix} q_{a1x} \\ q_{a1y} \end{pmatrix}_{\text{true}} A_{\text{osc}} \cos(\phi_i + d\phi) + \begin{pmatrix} q_{a2x} \\ q_{a2y} \end{pmatrix}_{\text{true}} A_{\text{osc}} \sin(\phi_i + d\phi) \quad (21.10)$$

where $d\phi$ is the phase of the reference oscillator with respect to the beam. Comparing Eq. (21.8) with Eq. (21.10) gives the relation

$$\begin{aligned} K_{22a} &= \frac{q_{a1x} q_{a1y} + q_{a2x} q_{a2y}}{q_{a1x}^2 + q_{a2x}^2} \\ K_{12a} &= \frac{q_{a1x} q_{a2y} - q_{a2x} q_{a1y}}{q_{a1x}^2 + q_{a2x}^2} \end{aligned} \quad (21.11)$$

This equation is general and can be applied in either the true or measurement frame of reference. Eq. (21.3) can be used to transform $(x_i, y_i)_{\text{true}}$ in Eq. (21.8) to the measurement frame of reference. Only the oscillating part is of interest. Averaging over many turns gives

$$\begin{pmatrix} q_{a1x} \\ q_{a1y} \end{pmatrix}_{\text{meas}} = \mathbf{M}_m \begin{pmatrix} q_{a1x} \\ q_{a1y} \end{pmatrix}_{\text{true}}, \quad \begin{pmatrix} q_{a2x} \\ q_{a2y} \end{pmatrix}_{\text{meas}} = \mathbf{M}_m \begin{pmatrix} q_{a2x} \\ q_{a2y} \end{pmatrix}_{\text{true}} \quad (21.12)$$

This neglects the measurement noise. A calculation shows that the noise gives a contribution to the measured K_{22a} and K_{12a} of

$$K_{22a} \rightarrow K_{22a} + r_1 \frac{n_f}{N_s A_{\text{osc}}}, \quad K_{12a} \rightarrow K_{12a} + r_2 \frac{n_f}{N_s A_{\text{osc}}} \quad (21.13)$$

Using the above equations, the transformation from the true coupling to measured coupling is as follows: From a knowledge of the true $\bar{\mathbf{C}}$ and Twiss values, the true K_{22a} and K_{12a} can be calculated via Eq. (21.9). Since the value of $d\phi$ does not affect the final answer, $d\phi$ in Eq. (21.10) is chosen to be zero. Comparing this to Eq. (21.8) gives

$$\begin{pmatrix} q_{a1x} \\ q_{a1y} \end{pmatrix}_{\text{true}} = \begin{pmatrix} 1 \\ K_{22a} \end{pmatrix}_{\text{true}}, \quad \begin{pmatrix} q_{a2x} \\ q_{a2y} \end{pmatrix}_{\text{true}} = \begin{pmatrix} 0 \\ K_{12a} \end{pmatrix}_{\text{true}} \quad (21.14)$$

Now Eq. (21.12) is used to convert to the measured q 's and Eq. (21.11) then gives the measured K_{22a} and K_{12a} . Finally, Applying Eq. (21.13) and then Eq. (21.9) gives the measured $\bar{\mathbf{C}}_{22}$ and $\bar{\mathbf{C}}_{12}$.

A similar procedure can be applied to b -mode oscillations to calculate values for the measured $\bar{\mathbf{C}}_{11}$ and $\bar{\mathbf{C}}_{12}$. K_{11b} and K_{12b} are defined by

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix}_{\text{true}} = A_{\text{osc}} \begin{pmatrix} K_{11b} \cos \phi_i + K_{12b} \sin \phi_i \\ \cos \phi_i \end{pmatrix}_{\text{true}} \quad (21.15)$$

Comparing this to David and Rubin [Sagan99] Eq. 55 gives

$$\begin{aligned} K_{11b} &= \frac{\sqrt{\beta_a}}{\gamma \sqrt{\beta_b}} \bar{\mathbf{C}}_{11} \\ K_{12b} &= \frac{-\sqrt{\beta_a}}{\gamma \sqrt{\beta_b}} \bar{\mathbf{C}}_{12} \end{aligned} \quad (21.16)$$

The q_{x1b} , q_{y1b} , q_{x2b} and q_{y2b} are defined by using Eq. (21.10) with the “a” subscript replaced by “b”. The relationship between K and q is then

$$\begin{aligned} K_{11b} &= \frac{q_{b1y} q_{b1x} + q_{b2y} q_{b2x}}{q_{b1y}^2 + q_{b2y}^2} \\ K_{12b} &= \frac{q_{b1y} q_{b2x} - q_{b2y} q_{b1x}}{q_{b1y}^2 + q_{b2y}^2} \end{aligned} \quad (21.17)$$

21.2.4 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_{osc} . Following the analysis of the previous subsection, the phase ϕ is

$$\begin{pmatrix} \phi_a \\ \phi_b \end{pmatrix}_{\text{meas}} = \begin{pmatrix} \phi_a \\ \phi_b \end{pmatrix}_{\text{true}} + \frac{n_f}{N_s A_{\text{osc}}} \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} - \begin{pmatrix} \tan^{-1} \left(\frac{q_{a2x}}{q_{a1x}} \right) \\ \tan^{-1} \left(\frac{q_{b2y}}{q_{b1y}} \right) \end{pmatrix}_{\text{meas}} \quad (21.18)$$

Part III

Programmer's Guide

Chapter 22

Bmad Programming Overview

22.1 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 other “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 developed non-CESR specific code libraries are: subsidiary libraries are:

bmad The **bmad** library contains the routines for relativistic charged particle simulation including particle tracking, Twiss calculations, symplectic integration, etc., etc.

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.

The **package** libraries are:

FFTW FFTW is a C subroutine library for computing the discrete Fourier transform in one or more dimensions. FFTW has a Fortran 2003 API.

GSL / FGSL The Gnu Scientific Library (GSL), written in C, provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total. The FGSL library provides a Fortran interface to the GSL library.

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 33.

LAPACK / LAPACK95 LAPACK is a widely used package of linear algebra routines written in Fortran77. The LAPACK95 library provides a Fortran95 interface to LAPACK.

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 30 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 manual[Forest02]. The core Differential Algebra (DA) package used by PTC was developed by Martin Berz[Berz89].

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”[Press92]. 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 double precision reals which is the standard for the other libraries (See §22.3).

XRAYLIB The XRAYLIB library provides routines for obtaining parameters pertinent to the X-ray interaction with matter.

xsif `xsif` is a library from SLAC to read in `xsif` format files. See §2.1 for more details. The only *Bmad* routine to use this library is `xsif_parser`.

22.2 Using `getf` and `listf` for Viewing Routine and Structure Documentation

As can be seen from the program example in Chapter 23 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/cesrulib/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/cesrulib/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(40): Name of the element to find.
!   lat      -- Lat_struct: Lattice 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/cesrulib/cesr_libs-devel/cvssrc/bmad/code/element_locator.f90
/home/cesrulib/cesr_libs-devel/cvssrc/bmad/code/elements_locator.f90
/home/cesrulib/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.

22.3 Precision of Real Variables

Historically, *Bmad* come 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. 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. Currently, `Bmad` is always compiled double precision and it is a near certainty that there would have to be some fixes if there was ever a need for compiling single precision.

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.

22.4 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.

22.5 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 23

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.

23.1 A First Program

Consider the example program shown in Fig. 23.1. This program is provided with the *Bmad* distribution in a file called `simple_bmad_program.f90`. `simple_bmad_program.f90` is in the library area in a directory called `examples/simple_bmad_program`. Directions on how to compile and run the program are given on the *Bmad* web page[[Bmad](#)].

```

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, cleo
7 type (ele_pointer_struct), allocatable :: eles(:)
8 integer i, ix, n_loc
9 logical err
10
11 ! Programs should always implement "intelligent bookkeeping".
12 bmad_com%auto_bookkeeper = .false.
13
14 ! Read in a lattice, and modify the ks solenoid strength of the
15 ! element named "cleo_sol".
16
17 call bmad_parser ("lat.bmad", lat) ! Read in a lattice.
18
19 call lat_ele_locator ('CLEO_SOL', lat, eles, n_loc, err) ! Find element
20 cleo => eles(1)%ele           ! Point to cleo_sol element.
21 cleo%value(ks$) = cleo%value(ks$) + 0.001 ! Modify ks component.
22 call set_flags_for_changed_attribute (cleo, cleo%value(ks$))
23 call lat_make_mat6 (lat, cleo%ix_ele)      ! Remake transfer matrix
24
25 ! Calculate starting Twiss params if the lattice is closed,
26 ! and then propagate the Twiss parameters through the lattice.
27
28 if (lat%param%geometry == closed$) call twiss_at_start (lat)
29 call twiss_propagate_all (lat)       ! Propagate Twiss parameters
30
31 ! Print info on the first 11 elements
32
33 print *, ' Ix  Name          Ele_type          S      Beta_a'
34 do i = 0, 10
35   ele => lat%ele(i)
36   print '(i4,2x,a16,2x,a,2f12.4)', i, ele%name, key_name(ele%key), ele%s, ele%a%beta
37 enddo
38
39 ! print information on the CLEO_SOL element.
40
41 print *
42 print *, '!-----',
43 print *, '! Information on element: CLEO_SOL'
44 print *
45 call type_ele (cleo, .false., 0, .false., 0, .true., lat)
46
47 deallocate (eles)
48
49 end program

```

Figure 23.1: Example Bmad program

23.2 Explanation of the Simple_Bmad_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` variable (line 5), which is of type `lat_struct` (§25.3), holds all of the lattice information: The list of elements, their attributes, etc. The setting of `bmad_com%auto_bookkeeper` to `False` in line 12 enables the “intelligent” bookkeeping of lattice attributes as discussed in §25.6). The call to `bmad_parser` (line 17) causes the lattice file `lat.bmad` to be parsed and the lattice information is stored the `lat` variable. Note: To get a listing of the `lat_struct` components or to find out more about `bmad_parser` use the `getf` command as discussed in §22.2.

The routine `lat_ele_locator` (§25.7) is used in line 19 to find the element in the lattice with the name `CLEO_SOL`. Line 20 defines a pointer variable named `cleo` which is used here as shortcut notation rather than having to write `ele(1)%ele` when referring to this element. Line 21 changes the `ks` solenoid strength of this element. Since an element attribute has been changed, the call to `set_flags_for_changed_attribute` in line 22 is needed for *Bmad* to correctly do the necessary lattice bookkeeping (§25.6). The call to `lat_make_mat6` in line 23 recalculates the linear transfer matrix for the `CLEO_SOL` element.

In line 28, the program checks if the lattice is circular (§8.1) and, if so, uses the routine `twiss_at_start` 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)` (§27.2). 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 output some information. Lines 22 through 26 of the program print information on the first 11 elements in the lattice. 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 25. 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 35) makes for a cleaner syntax and reduces typing. The table that is produced is shown in lines 1 through 12 of Fig. 23.2. The first column is the element index i . The second column, `ele%name`, is the name of the element. The third column, `key_name(eleclass. ele%key)` is an integer 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 `type_ele` routine on line 45 of the program is used to type out the `CLEO_SOL`’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_bmad_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 Fig. 23.2 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 §7.1 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 25, the lattice list that is used for tracking extends from `lat%ele(0)` through `lat%ele(n)` where $n = lat%n_ele_track$. The CLEO_SOL element is put in the section of `lat%ele(n)` with $n > lat%n_ele_track$ 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 `examples` the working directory, the `mk` command will compiling and link the program. The executable, `simple_bmad_program` is to be found in `../production/bin/` directory. Go to the directory

```
simple_bmad_program
```

and run the program with the command

```
./../production/bin/simple_bmad_program
```

This gives the output as shown in Fig. 23.2.

```

1   Ix  Name           Ele_type          S      Beta_a
2   0   BEGINNING     BEGINNING_ELE    0.0000  0.9381
3   1   IP_L0          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   QOOE\CLEO_SOL SOL_QUAD         1.7550  7.8619
8   6   QOOE#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   QOOE\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   QOOE\CLEO_SOL          5         -----  3.182E-01

```

Figure 23.2: Output from the example program

Chapter 24

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 Figs. 24.1 and 24.2. 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 downstream end of the element.

For printing information about an element, the `type_ele` or `type_ele` routines can be used (§23.1). The difference between the two is that `type_ele` will print to the terminal window while `type_ele` will return an array of strings containing the element information.

```
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 ! Description string.
  type (twiss_struct) a, b, z ! Twiss parameters at end of element \sref{c:normal.modes}.
  type (xy_disp_struct) x, y ! Projected dispersions \sref{c:normal.modes}.
  type (bookkeeping_state_struct) bookkeeping ! Element attribute bookkeeping
  type (branch_struct), pointer :: branch ! Pointer to branch containing element.
  type (em_fields_struct), pointer :: em_field ! DC and RF E/M fields
  type (floor_position_struct) floor ! Global floor position.
  type (ele_struct), pointer :: lord ! Pointer to a slice lord.
  type (mode3_struct), pointer :: mode3 ! Full 6-dimensional normal mode decomposition.
  type (fibre), pointer :: ptc_fiber ! PTC tracking.
  type (genfield), pointer :: ptc_genfield ! For symp_map
  type (rad_int_ele_cache_struct), pointer :: rad_int_cache ! Radiation integral calc cached values
  type (wake_struct), pointer :: wake ! Wakes
  type (space_charge_struct), pointer :: space_charge
  type (taylor_struct) :: taylor(6) ! Taylor terms
  ele_struct definition continued on next figure...
```

Figure 24.1: The `ele_struct`. structure definition. The complete structure is shown in this and the following figure.

```

... ele_struct definition continued from previous figure.

type (wall3d_struct) :: wall3d           ! Chamber or capillary wall
type (wig_struct), pointer :: wig         ! Wiggle field
type(coord_struct) map_ref_orb_in(6)      ! Transfer map ref orbit at upstream end of element.
type(coord_struct) map_ref_orb_out(6)     ! Transfer map ref orbit at downstream end of element.
type(coord_struct) time_ref_orb_in(6)     ! Reference orbit at upstream end for ref_time calc.
type(coord_struct) time_ref_orb_out(6)    ! Reference orbit at downstream end for ref_time calc.
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 downstream end.
real(rp) ref_time                       ! Time ref particle passes downstream end.
real(rp), pointer :: r(:,:,:,:)          ! For general use. Not used by Bmad.
real(rp), pointer :: a_pole(:)           ! multipole
real(rp), pointer :: b_pole(:)           ! multipoles
real(rp), pointer :: const(:)           ! Working constants.

integer key                            ! key value
integer sub_key                         ! EG for wiggler: 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 spin_tracking_method             ! bmad_standard$, symp_lie_ptc$, etc.
integer field_calc                        ! Used with Boris, Runge-Kutta integrators.
integer aperture_at                      ! Aperture location: exit_end$, ...
integer aperture_type                    ! Type of aperture: rectangular$, elliptical$, or custom$.
integer orientation                       ! -1 -> Element is longitudinally reversed. +1 -> Normal.
logical symplectify                     ! Symplectify mat6 matrices.
logical mode_flip                         ! Have the normal modes traded places?
logical multipoles_on                   ! For turning multipoles on/off
logical scale_multipoles                ! multipole components scaled by the strength of element?
logical taylor_map_includes_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 csr_calc_on                      ! Coherent synchrotron radiation calculation
logical offset_moves_aperture           ! element offsets affects aperture?
end type

```

Figure 24.2: The `ele_struct`. The complete structure is shown in this and the preceding figure.

24.1 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 (§25.2) 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.
```

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`. The exception here are the “Electro-magnetic field component” pointers `ele%wig_term`, `ele%em_field%mode(:)%map`, and `ele%em_field%mode(:)%grid`. Since these components potentially contain large arrays, and since the individual sub-components of these components are not likely to be individually modified, The field component pointers of `ele1` and `ele2` after the set `ele1 = ele2` will point at the same memory locations.

Note: The assignment `ele1 = ele2` will not modify `ele1%ix_ele` or `ele1%ix_branch`. If `ele1` is associated with a lattice then `ele1%lat` will also be unaffected.

24.2 Element Attribute Bookkeeping

When a value of an attribute in an element changes, the values of other attributes may need to be changed (§4.1). Furthermore, in a lattice, changes to one element may necessitate changes to attribute values in other elements. For example, changing the accelerating gradient in an `lcavity` will change the reference energy throughout the lattice.

The attribute bookkeeping for a lattice can be complicated and, if not done intelligently, can cause programs to be slow if attributes are continually being changed. In order to keep track what bookkeeping has been done, the `ele%status` component is used by the appropriate bookkeeping routines for making sure the bookkeeping overhead is kept to a minimum. However, “intelligent” bookkeeping is only done if explicitly enabled in a program. See §25.6 for more details.

24.3 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 (§26.1), 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.

24.4 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 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 Sbend 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 (§26.1), all `rbend` elements are converted into `sbend` elements (§3.5). To keep track of what the original definition of the element was, the `%sub_key` component will be set to `sbend$` or `rbend$` 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` (§26.3).

24.5 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 3.

To obtain a list of attribute names and associated `%value(:)` indexes, the program `element_attributes` can be used. This program is included in the standard *Bmad* distribution.

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. When the element is an `rfcavity`, `%field_master` is used to store the setting of the `harmon_master` flag.

The `%old_value(:)` component of the `ele_struct` is used by the `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`.

24.6 Connection with the Lat_Struct

If an element is part of a `lat_struct` (§25), the `%ix_ele` and `%ix_branch` components of the `ele_struct` identify where the element is. Additionally, the `%lat` component will point to the encompassing lattice. That is

```
type (lat_struct), pointer :: lat
type (ele_struct), pointer :: ele2
if (ele%ix_ele > -1) then
  ie = ele%ix_ele
  ib = ele%ix_branch
  lat => ele%lat
  ele2 => lat%branch(ib)%ele(ie)
  print *, associated(ele2, ele) ! Will print True.
endif
```

In this example the `ele2` pointer is constructed to point to the `ele` element. The test (`ele%ix_ele > -1`) is needed since `ele_struct` elements may exist outside of any `lat_struct` instance. Such “external” elements always have `%ix_ele < 0`. A value for `%ix_ele` of -2 is special in that it prevents the `deallocate_ele_pointers` routine from deallocating the pointers of an element which has its `%ix_ele` set to -2.

An element “slice” is an example of an element that exists external to any `lat_struct` instance. A slice is an `ele_struct` instance that represents some sub-section of a given element. Element slices are useful when tracking particles only part way through an element (§28.7).

24.7 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 §28.8 for more details.

24.8 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 downstream end of the element. See Chapter 27 for more details.

24.9 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` (§25).

24.10 Coordinates, Offsets, etc.

The “upstream” and “downstream” ends of an element are, by definition, where the physical ends of the element would be if there were no offsets. In particular, if an element has a finite `z_offset`, the physical ends will be displaced from upstream and downstream ends. See §28.2 for more details.

The `%floor` component gives the global “floor” coordinates (§13.3) at the downstream 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(z_offset$)
%value(x_pitch$)
%value(y_pitch$)
%value(tilt$)
```

If the element is supported by a `girder` element (§3.19) 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(z_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 downstream end of an element is stored in `%s` and the reference time is stored in `%ref_time`. See §13.3 for more details.

24.11 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 upstream end of the element about which the transfer matrix is computed is stored in `%map_ref_orb_in` and the the reference orbit at the downstream end is stored in `%map_ref_orb_out`. In the calculation of the transfer map, the vector `%vec0` is set so that

```
map_ref_orb_out = %mat6 * map_ref_orbit_in + %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 upstream 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 (§18.1).

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 (§22.1). The PTC partially inverted map does not have any zeroth order terms so the zeroth order terms are stored in the `%gen0(6)` vector.

24.12 Reference Energy and Time

The reference energy and reference time are computed around a reference orbit which is different from the reference orbit used for computing transfer maps (§24.11). The energy and time reference orbit for an element is stored in

```
ele%time_ref_orb_in      ! Reference orbit at upstream end
ele%time_ref_orb_out     ! Reference orbit at downstream end
```

Generally `ele%time_ref_orb_in` is the zero orbit. The exception comes when an element is a `super_slave`. In this case, the reference orbit through the `super_slaves` of a given `super_lord` is constructed to be continuous. This is done for consistancy sake. For example, to ensure that when a marker is superimposed on top of a wiggler the reference orbit, and hence the reference time, is not altered.

24.13 EM Fields

`%em_field` component holds information on the electric and magnetic fields of an element (§3.15) Since `ele%em_field` is a pointer its association status must be tested before any of its sub-components are accessed.

```
type (ele_struct) ele
...
if (associated(ele%em_field)) then
  ...
end if
```

The `ele%em_field` component is of type `em_fields_struct` which holds an array of modes

```
type em_fields_struct
  type (em_field_mode_struct), allocatable :: mode(:)
end type
```

Each mode has components

```
type em_field_mode_struct
  integer m                      ! Mode varies as cos(m*phi - phi_0)
  real(rp) freq                  ! Oscillation frequency (Hz)
  real(rp) :: f_damp = 0          ! 1/Q damping factor
  real(rp) :: phi0_ref = 0        ! Mode oscillates as: twopi * (f * t + phi0_ref)
  real(rp) :: phi0_azimuth = 0    ! Azimuthal orientation of mode.
  real(rp) :: field_scale = 1    ! Factor to scale the fields by
  type (em_field_mode_map_struct), pointer :: map => null()
  type (em_field_grid_struct), pointer :: grid => null()
end type
```

24.14 Wakes

The `ele%wake` component holds information on the wakes associated with an element. Since `ele%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, 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.

See §14.3 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 (wake_sr_mode_struct) :: sr_long
  type (wake_sr_mode_struct) :: sr_trans
  type (wake_lr_struct), allocatable :: lr(:)
  real(rp) :: z_sr_max = 0
end type

```

The short-range wake parameterization uses pseudo-modes (Eq. (14.35)). 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 wake_sr_mode_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. (14.35). `%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 wake_lr_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 §3.25. `%polarized` is a logical that indicates whether the mode has a polarization angle. If so, then `%angle` is the polarization angle.

24.15 Wiggler Types

The `%sub_key` component of the `ele_struct` is used to distinguish between `map` type and `periodic` type wigglers (§24.4):

```
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 (§3.43.1) 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$
```

24.16 Multipoles

The multipole components of an element (See §14.1) are stored in the pointers `%a_pole(:)` and `%b_pole(:)`. If `%a_pole` and `%b_pole` are allocated they always have a range `%a_pole(0:n_pole_maxx)` and `%b_pole(0:n_pole_maxx)`. Currently `n_pole_maxx = 20`. For a `Multipole` element, the `%a_pole(n)` array stores the integrated multipole strength K_{nL} , and the `%b_pole(n)` array stores the tilt T_n .

A list of *Bmad* routines for manipulating multipoles can be found in §35.24.

24.17 Tracking Methods

A number of `ele_struct` components control tracking and transfer map calculations. These are:

```
%mat6_calc_method
%tracking_method
%taylor_order
%symplectify
%multipoles_on
%taylor_map_includes_offsets
```

```
%is_on
%csr_calc_on
%offset_moves_aperture
```

See Chapter §28 for more details.

24.18 Custom and General Use Attributes

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 `ele%r(:,:,:)`, `ele%ix_pointer` and `ele%logic`

```
type ele_struct
  ...
  real(rp), pointer :: r(:,:,:,:) => null()
  integer ix_pointer
  logical logic
  ...
end type
```

Additionally, there are 5 slots in the `%value(:)` array for general use. they have indexes labeled `custom_attribute1$` through `custom_attribute5$`. 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$ = custom_attribute1$
...
lat%ele(i)%value(time_stamp$) = ...
```

In the lattice file, a string can be associated with `custom_attributeN$` (§2.10). Thus, in the lattice file, to associate "time_stamp" with `custom_attribute1`:

```
parameter[custom_attribute1] = "time_stamp"
```

Defining the custom attribute in the lattice file allows the `type_ele` routine to print the custom information. To setup the association between a string and a custom attribute from within a program

```
type (lat_struct), target :: lat
type (ele_struct), pointer :: ele
integer, parameter :: time_stamp$ = custom_attribute1$
...
ele => lat%branch(1)%ele(2) ! Point to some element
print *, attribute_name(ele, time_stamp$), ' Has value:' ele%value(time_stamp$)
```

If not defined through a lattice file, custom attributes can also be defined directly from within a program using the `set_attribute_alias` routine

```
type (lat_struct) lat
logical err_flag
...
call set_attribute_alias ('CUSTOM_ATTRIBUTE2', 'QUADRUPOLE::ERROR', err_flag, lat)
```

The `lat` argument is only needed if the lattice is to be written to a file using `write_bmad_lattice_file`.

24.19 Bmad Reserved Variables

`indexele_struct!%ixx` `indexele_struct!%old_is_on` `indexele_struct!%bmad_logic` `indexele_struct!%const`
A number of `ele_struct` components are reserved for use by *Bmad* routines only. These are:

```
%ixx
%old_is_on
%bmad_logic
```

The `%ixx` and `%bmad_logic` components are used for internal *Bmad* bookkeeping purposes.

24.20 Creating Element Slices

It is sometimes convenient to split an element longitudinally into “slices” that represent a part of the element. This is complicated by the fact that elements are not necessarily uniform. For example, map type wigglers are nonuniform and bend elements have end effects. Furthermore, attributes like `hkick` need to be scaled with the element length.

To create an element slice, the routine `create_element_slice` can be used. Example:

```
type (ele_struct) ele, sliced_ele
...
sliced_ele = ele
sliced_ele%value(l$) = l_slice ! Set the sliced element's length
call create_element_slice (sliced_ele, ele, l_start, param, ...)
```

See the documentation on `create_element_slice` for more details (§[22.2](#)).

Chapter 25

The lat_struct

The `lat_struct` is the structure that holds of all the information about a lattice (§1.3). The components of a `lat_struct` are listed in Fig. 25.1.

```
type lat_struct
    character(40) 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 25.1: Definition of the `lat_struct`.

The `%ele_init` component within the `lat_struct` is not used by *Bmad* and is available for general program use.

25.1 Initializing

Normally initialization of a `lat_struct` lattice is done by `bmad_parser` when a lattice file is parsed and does not have to be done by the programmer. When a programmer needs to initialize a lattice, however, `init_lat` is used to initialize the lattice with a single branch. After this initial setup, the routines `allocate_branch_array` and `allocate_lat_ele_array` can be used to set up additional branches. Example:

```
type (lat_struct) lat
...
call init_lat (lat, 1000)           ! Branch(0) has 1001 elements.
call allocate_branch_array (lat, 2) ! Allocate Branch(1) and Branch(2).
call allocate_lat_ele_array (lat, 20, 1) ! Branch(1) has 21 elements
call allocate_lat_ele_array (lat, 30, 2) ! Branch(2) has 31 elements.
```

25.2 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.

`deallocate_lat_pointers` 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.

25.3 Branches in the lat_struct

The lattice is divided up into the “root branch” (§6.6) and, if there are `fork` or `photon_fork` elements, a number “forked” branches.

The branches of a lattice is contained in the `lat%branch(0:)` array. The `%branch(0:)` array is always indexed from 0 with the 0 branch being a root branch. The definition of the `branch_struct` structure is

```
type branch_struct
character(40) name
integer ix_branch                      ! Index in lat%branch(:) array.
integer ix_from_branch                  ! -1 => No forking element to this branch.
integer ix_from_ele                     ! Index of forking element
```

Element index		
section	min	max
tracking	0	%n_ele_track
control	%n_ele_track+1	%n_ele_max

Table 25.1: Bounds of the tracking and control parts of the root branch (`lat%branch(0)%ele(:)`) array.

```

integer, pointer :: n_ele_track      ! Number of tracking elements
integer, pointer :: n_ele_max
type (mode_info_struct), pointer :: a, b, z
type (ele_struct), pointer :: ele(:)
type (lat_param_struct), pointer :: param
type (wall3d_struct), pointer :: wall3d
type (ptc_branch1_info_struct) ptc
type (normal_form_struct) normal_form_with_rf, normal_form_no_rf
end type

```

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 fork or photon_fork element that marks the beginning of the i^{th} branch. Example:

```

type (lat_struct), target :: lat
type (ele_struct), pointer :: ele
...
ib = lat%branch(3)%ix_from_branch
ie = lat%branch(3)%ix_from_ele
! ele is the fork or photon_fork element for lat%branch(3)
ele => lat%branch(ib)%ele(ie)
! This is the same as the above.
ele => pointer_to_ele(lat%branch(3)%ix_from_branch, lat%branch(3)%ix_from_ele)

```

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 maintain compatibility with older code, the following components point to the same memory blocks

<code>lat%ele(:)</code>	\longleftrightarrow	<code>lat%branch(0)%ele(:)</code>
<code>lat%n_ele_track</code>	\longleftrightarrow	<code>lat%branch(0)%n_ele_track</code>
<code>lat%n_ele_max</code>	\longleftrightarrow	<code>lat%branch(0)%n_ele_max</code>
<code>lat%param</code>	\longleftrightarrow	<code>lat%branch(0)%param</code>

All `%branch%ele(:)` arrays are allocated with zero as the lower bound. The `%ele(0)` element of all branches is an `beginning_ele` element with its `%name` component set to “BEGINNING”. `%ele(0)%mat6` is always the unit matrix. For the root 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 (§25.5). The bounds of these two parts is given in Table 25.3. Only the root branch has a lord section so `%branch%n_ele_track` and `%branch%n_ele_max` are the same for all other branches. Since the root 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(:)`.

for a given `fork` or `photon_fork` element, the index of the branch that is being forked to and the index of the element that is being forked to is stored in:

```

ix_branch = nint(branch_ele%value(ix_branch_to$)) ! branch index
ix_element = nint(branch_ele%value(ix_element_to$)) ! element index
direction = nint(branch_ele%value(direction$))

```

The direction will be +1 for forward forking and -1 for backward forking.

25.4 Param_struct Component

The `%param` component within each `lat%branch(:)` is a `lat_param_struct` structure whose definition is shown in Fig. 25.2 This structure would be more aptly named `branch_param_struct` but is named otherwise for historical reasons.

`%param%total_length` is the length of the branch 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.

For closed branches, `%param%t1_with_RF` and `%param%t1_no_RF` are the 1-turn transfer matrices from the start of the branch 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%unstable_factor` will be set to the growth rate per turn of the unstable mode.

For open branches, if a particle is lost in tracking, `%param%unstable_factor` will be set to

```
orbit_amplitude / limit - 1
```

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:

```

electron$,      positron$,
muon$,         antimuon$,
proton$,        antiproton$,
photon$,        pion_0$,
pion_minus$,   pion_plus$

```

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 e . 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:       ', charge_of(lat%param%particle)

```

```

type lat_param_struct
    real(rp) n_part          ! Particles/bunch (for beambeam elements).
    real(rp) total_length     ! total_length of lattice
    real(rp) unstable_factor  ! closed branch: growth rate/turn.
                               ! open branch: |orbit/limit|
    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, etc.
    integer geometry            ! open$, etc...
    integer ixx                 ! Integer for general use
    logical stable              ! For closed branch. Is lat stable?
    logical aperture_limit_on  ! use apertures in tracking?
    type (bookkeeper_status_struct) bookkeeping_state
                               ! Overall status for the branch.
end type

```

Figure 25.2: Definition of the param_struct.

25.5 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 (§3.33) 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 (§7.1).

`girder_lord$`

A `girder_lord` element is a `girder` element (§3.19). That is, the element will have `ele%key = girder$`.

`multipass_lord$`

`multipass_lord` elements are created when multipass lines are present (§7.2).

`overlay_lord$`

An `overlay_lord` is an `overlay` element (§3.33). That is, such an element will have `ele%key = overlay$`.

`group_lord$`

A `group_lord` is a `group` element (§3.20). That is, such an element will have `ele%key = group$`.

		ele%lord_status								lord%lord_status					
		not_a_lord\$	group_lord\$	girder_lord\$	overlay_lord\$	multipass_lord\$	super_lord\$			not_a_lord\$	group_lord\$	girder_lord\$	overlay_lord\$	multipass_lord\$	super_lord\$
ele%slave_status								slave%slave_status							
free\$	X	X	X	X	X	X									
control_slave\$	X	X	X	X	X	X									
multipass_slave\$	X					X									
slice_slave\$	X														
super_slave\$	X														

(a) Possible ele%lord_status and ele%slave_status combinations within an individual element.

(b) Possible %lord_status and %slave_status combinations for any lord/slave pair.

Table 25.2: 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.

not_a_lord\$

This element does not control anything.

Any element whose %lord_status is something other than not_a_lord\$ is called a lord element. In the tracking part of the branch (§25.3), %lord_status will always be not_a_lord\$. In the lord section of the branch, under normal circumstances, there will never be any not_a_lord elements. However, it is permissible, and sometimes convenient, for programs to set the %lord_status of a lord element to not_a_lord\$.

The possible values for the ele%slave_status component are:

free\$

Free elements do not have any lord elements controlling them.

multipass_slave\$

A multipass_slave element is the slave of a multipass_lord (§7.2).

control_slave\$

A control_slave is an element that has one or more overlay_lord or girder_lord lords. A control_slave can also have associated group_lord elements.

slice_slave\$

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

super_slave\$

A super_slave element is an element in the tracking part of the branch that has one or more super_lord lords (§7.1).

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 branch. The other types can be in either the tracking or control parts of the branch.

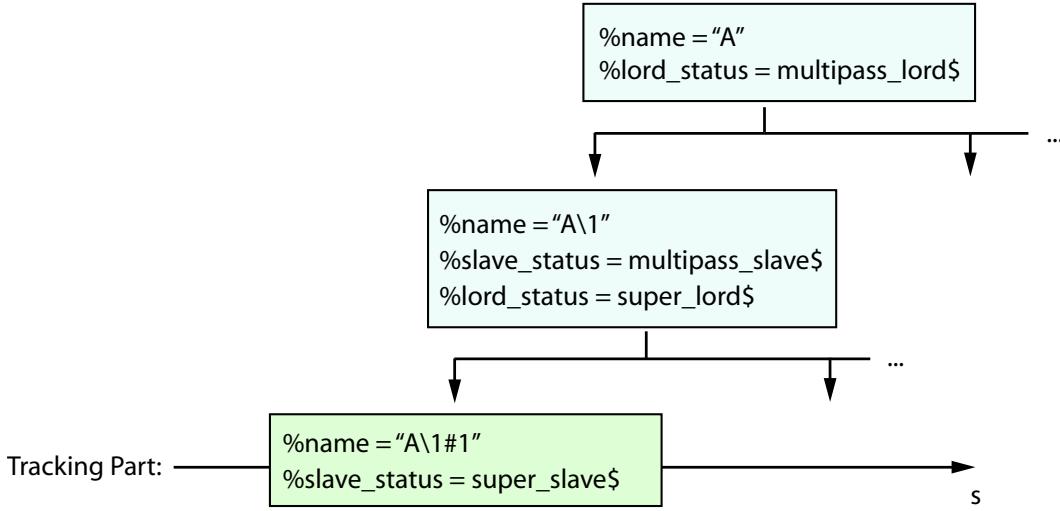


Figure 25.3: Example of multipass combined with superposition. A `multipass_lord` element named A controls a set of `multipass_slaves` (only one shown). The `multipass_slave` elements are also `super_lord` elements and they will control `super_slave` elements in the tracking part of the branch.

Only some combinations of `%lord_status` values and `%slave_status` values are permissible for a given element. Table 25.2(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 25.2(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 §7.2, 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 branch 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 Fig. 25.3.

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
  ! lord_ele points to the i^th lord element of this_ele
  lord_ele => pointer_to_lord (this_ele, i)
  ...
enddo

do i = 1, this_ele%n_slave ! Loop over all slaves of this_ele
  ! slave_ele points to the i^th slave element of this_ele
  slave_ele => pointer_to_slave (this_ele, i)
  
```

```
...
enddo
```

The lord/slave bookkeeping is bidirectional. That is, for any given element, call it `this_ele`, consider the i^{th} lord:

```
lord_ele_i => pointer_to_lord (this_ele, i)
```

then there will always be some index j such that the element pointed to by

```
pointer_to_slave(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^{th} slave

```
slave_ele_i => pointer_to_slave (this_ele, i)
```

there will always be some index j such that the element pointed to by

```
pointer_to_lord(slave_ele_i, j)
```

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 (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 (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 (this_ele, 1)`.

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 root 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 (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 (lord, 1, ix_con)
  print *, lord%ix1_slave == ix_con ! Will print "T" for True
  slave => pointer_to_slave (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(slave, 1, ix_con)
  print *, 'Multipass_slave: ', slave%name
  print *, 'Is in pass number:', ix_con - lord%ix1_slave + 1
endif

```

Except for a `slice_slave`, 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. Somewhat simplified, the code for the `pointer_to_lord` function is:

```

function pointer_to_lord (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

```

This method for finding the lords of an element is considered “private”. That is, no code outside of the official *Bmad* library should rely on this.

`slice_slave` element bookkeeping has is different depending upon whether the element being sliced is a `super_slave` or not. If the element being sliced is a `super_slave`, a `slice_slave` element that is created is, for bookkeeping purposes, considered to be a slave of the `super_slave`’s lords. In this case, the bookkeeping is exactly the same as that of any `super_slave`, and `pointer_to_lord` will return a pointer to one of the `super_slave`’s lords.

On the other hand, if a non `super_slave` element is being sliced, the `%lord` pointer component of the `slice_slave` will be set to point to the element being sliced.

25.6 Lattice Bookkeeping

The term “lattice bookkeeping” refers to the updating of the appropriate parameter values when a given parameter in the lattice is changed. For example, if the accelerating gradient of an `lcavity` element is modified, the reference energy parameter of all elements downstream of the `lcavity` will need to be changed and this can also alter the transfer maps of the `lcavity` and downstream elements. *Bmad* divides the lattice bookkeeping into “core” part and everything else. The core part itself is divided into five parts:

Attribute bookkeeping

This refers to intra-element dependent attribute bookkeeping (§4.1).

Control bookkeeping

This refers to Lord/Slave bookkeeping for `overlay` (§3.33) and `group` (§3.20) elements, and for `superposition` (§7.1) and `multipass` (§7.2) lords.

Floor Position bookkeeping This refers to bookkeeping to keep track of an elements global “floor” position stored in the `ele%floor` structure.

Length bookkeeping This refers to bookkeeping to keep track of the longitudinal s-position of an element stored in the `ele%s` component.

Reference Energy bookkeeping This refers to the reference energy assigned to each element (§29.4).
`ele%value(E_tot$)` and `ele%value(p0c$)`

Historically, as the concept of lattice bookkeeping was being developed, to be back compatible with existing programs, calls to bookkeeping routines were added to calculational routines such as the tracking routine `track1` and the routine for calculating the linear transfer map `make_mat6`. This “automatic” bookkeeping system is inefficient since there is no good way to keep track of what element attributes have been modified which leads to redundant bookkeeping calculations. Eventually, as *Bmad* developed and became more complicated, it was found that the unnecessary bookkeeping load was generally causing a significant slowdown in program execution time — even in programs where no element attributes were changed. To avoid this, an “intelligent” bookkeeping system was developed. In order to be back compatible with existing programs, the automatic bookkeeping system is the default. However, given the fact that the automatic bookkeeping system has known deficiencies, and given the overhead with

maintaining two bookkeeping systems, the current plan is to start phasing out the automatic bookkeeping system sometime in the not-so-far future. Thus old programs should be converted to the new system and all new programs should use the new bookkeeping system.

To use intelligent bookkeeping, a program must set the global `bmad_com%auto_bookkeeper` to false. This is done once at the start of the program. When a set of attributes needs to be modified, the `set_flags_for_changed_attribute` routine must be called for each element attribute that is set. After all the attributes have been set, `lattice_bookkeeper` is called to do the core bookkeeping. Example

```
type (lat_struct) lat
...
bmad_com%auto_bookkeeper = .false.      ! Only needs to be done once.
...
lat%ele(i)%value(gradient$) = 1.05e6  ! Change, say, the gradient of an RFCavity
call set_flags_for_changed_attribute (lat%ele(i), lat%ele(i)%value(gradient$))

... Set attributes of other elements ...

call lattice_bookkeeper (lat)          ! Do once after all attribute sets done.
```

The argument list for `set_flags_for_changed_attribute` is

```
set_flags_for_changed_attribute (ele, attribute)
```

The `attribute` argument may be either real, integer, or logical.

The `set_flags_for_changed_attribute` routine sets flags in the `ele%status` structure. This structure is of type `bookkeeper_status_struct` and looks like

```
type bookkeeper_status_struct
    integer attributes      ! Intra element dependent attribute status
    integer control         ! Lord/slave bookkeeping status
    integer floor_position  ! Global (floor) geometry status
    integer length          ! Longitudinal position status
    integer ref_energy       ! Reference energy status
    integer mat6             ! Linear transfer map status
    integer rad_int          ! Radiation integrals cache status
end type
```

All components of this structure give the status of some lattice bookkeeping aspect. The first five components of this structure correspond to the five core bookkeeping parts discussed above. The other two components are discussed below.

Possible values for the status components are

```
super_ok$
ok$
stale$
```

The `set_flags_for_changed_attribute` routine sets the appropriate status components of an element to `stale$` which marks that element for the appropriate bookkeeping. When the bookkeeping is done by `lattice_bookkeeper`, the `stale$` status components are set to `ok$`. The `super_ok$` value is reserved for use by any program that needs to do its own custom bookkeeping. How this works is as follows: The *Bmad* bookkeeping routines will never convert a status component with value `super_ok$` to `ok$` without first doing some needed bookkeeping. Thus if a program sets a status component to `super_ok$` and then later on finds that the status component is set to `ok$`, the program knows that bookkeeping has been done. An example will make this clear. Suppose a program needs to keep track of a collection of high order transfer maps between various points in a lattice. Suppose that the constant calculation of these maps would slow the program down so it is desired to recalculate a given map only when necessary. To

implement this, the program could set the `ele%status%mat6` attribute of all the element to `super_ok$` when the maps are calculated. If the program subsequently finds a `ele%status%mat6` attribute of an element set to `ok$` it knows that it should recalculate any transfer maps that span that element.

It is guaranteed that when `lattice_bookkeeper` is run, all five core status components will not be `stale$`. The routines used by `lattice_bookkeeper` are:

```
attribute_bookkeeper      ! Intra-element attributes
control_bookkeeper       ! Lord/slave control
s_calc                  ! Longitudinal element s-position
lat_geometry             ! Global (floor) positions.
lat_compute_ref_energy_and_time ! Reference energy
```

In general, these routines should not be called directly since the correct way to do things is not always straight forward. See the code for `lattice_bookkeeper` for more details.

After the core bookkeeping is done, a program can call `lat_make_mat6` to remake the transfer matrices. `lat_make_mat6` will remake the transfer matrices if either the `ele%status%mat6` flag is `stale$` or the reference orbit around which the existing transfer matrix was computed has shifted. `lat_make_mat6` will set the `ele%status%mat6` flag to `ok$` for all elements whose transfer matrices are recomputed.

25.7 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
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, its 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 an 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`.

25.8 Beam_start Component

The `lat%beam_start` component is a `coord_struct` structure for holding the information obtained from `beam_start` statements (§8.2) 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.

25.9 Adding and Deleting Elements From a Lattice

Modifying the number of elements in a lattice involves a bit of bookkeeping. To help with this there are a number of routines.

For adding elements there are three basic routines: To add a lord element, the `new_control` routine is used. To add a new element to the tracking part of the lattice, use the `insert_element` routine. Finally to split an element into two pieces, the routine `split_lat` is used. These basic routines are then used in such routines as `create_overlay` that creates overlay elements, `create_group` which creates group elements, `add_superimpose` which superimposes elements, etc.

The routine `remove_eles_from_lat` is used to delete elements from a lattice.

Chapter 26

Reading and Writing Lattices

26.1 Reading in Lattices

`xsif_parser` `bmad_parser` `bmad_parser2`

Bmad has routines for reading XSIF (§2.1) 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` (§25) 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 (§2.5) 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
type (lat_struct) lat
call bmad_parser ('lattice_file', lat)      ! read in a lattice.
call bmad_parser2 ('aperture_file', lat)     ! read in the aperture limits.
```

An alternative to using `bmad_parser` and `xsif_parser` is to use the combined *Bmad* and XSIF parser `bmad_and_xsif_parser`. This parser will assume that the input file is using *Bmad* syntax unless the file name is prefixed by the string “`xsif::`”.

26.2 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
```

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.

`read_digested_bmad_file` `write_digested_bmad_file` 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`

26.3 Writing Lattice files

`write_bmad_lattice_file` 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.

Chapter 27

Normal Modes: Twiss Parameters, Coupling, Emittances, Etc.

27.1 Components in the Ele_struct

The `ele_struct` (§24) 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 (§17.1) 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      ! Energy normalized emittance (=  $\gamma \epsilon$ )
end type
```

The projected horizontal and vertical dispersions in an `ele_struct` are contained in the `ele%x` and `ele%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 components `ele%emit`, `ele%norm_emit`, `ele%sigma`, `ele%sigma_p` are not set by the standard *Bmad* routines and are present for use by any program.

The relationship between the projected and normal mode dispersions are given by Eq. (17.14). The 2x2 coupling matrix \mathbf{C} (Eq. (17.3)) is stored in the `ele%c_mat(2,2)` component of the `ele_struct` and the γ factor of Eq. (17.3) 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 coupling matrices.
```

See §35.20 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 `set_flags_for_changed_attribute` routine can be used. Example:

```
type (lat_struct), target :: lat
real(rp), pointer :: attrib_ptr
...
attrib_ptr => lat%ele(ix_ele)%value(k1$) ! Point to some attribute.
attrib_ptr = value                         ! Change the value.
call set_flags_for_changed_attribute (lat%ele(ix_ele), attrib_ptr)
```

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[Sagan99] 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. (17.2) 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[Wolski06]. 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. (17.1).

27.2 Tune and 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 (§8.4). For circular machines, the routine `twiss_at_start` may be used (§8.4)

```
type (lat_struct) lat
...
if (lat%param%geometry == closed$) call twiss_at_start(lat)
```

In either case, the initial Twiss parameters are placed in `lat%ele(0)`. The tune is placed in the variables `lat%a%tune` and `lat%b%tune`.

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 (§28). `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, non-zero kickers or quadrupole offsets, and then remakes the transfer matrices, which are stored in `lat%ele(i)%mat6`, around the closed orbit.

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)
```

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_intra_ele`.

27.3 Tune Setting

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 make 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.

27.4 Emittances & Radiation Integrals

See Section §16.2 for details on the radiation integral formulas.

The routine `radiation_integrals` is used to calculate the normal mode emittances along with the radiation integrals:

```
type (lat_struct) lat
type (normal_modes_struct) modes
type (rad_int_all_ele_struct) ele_rad_int
...
call radiation_integrals (lat, orbit, modes, rad_int_by_ele = ele_rad_int)
```

The `modes` argument, which is of type `normal_modes_struct`, holds the radiation integrals integrated over the entire lattice.

```
type normal_modes_struct
  real(rp) synch_int(0:3) ! Synchrotron integrals I0, I1, I2, and I3
  real(rp) sigE_E          ! SigmaE/E
  real(rp) sig_z            ! Sigma_Z
  real(rp) e_loss           ! Energy loss / turn (eV)
  real(rp) rf_voltage       ! Total rf cavity voltage (eV)
```

```

real(rp) pz_aperture      ! pz aperture limit
type (anormal_mode_struct) a, b, z
type (linac_normal_mode_struct) lin
end type

```

In particular, the %a, %b, and %z components, which are of type `anormal_mode_struct` hold the emittance values:

```

type anormal_mode_struct
  real(rp) emittance          ! Beam emittance
  real(rp) synch_int(4:6)     ! Synchrotron integrals
  real(rp) j_damp              ! damping partition number
  real(rp) alpha_damp          ! damping per turn
  real(rp) chrom                ! Chromaticity
  real(rp) tune                 ! "Fractional" tune in radians
end type

```

The `ele_rad_int` argument, which is of type `rad_int_all_ele_struct`, holds the radiation integrals on an element-by-element basis.

```

type rad_int_all_ele_struct
  type (rad_int1_struct), allocatable :: ele(:) ! Array is indexed from 0
end type

```

27.5 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 28

Tracking and Transfer Maps

Bmad has routines for tracking two types of objects called “particles” and “macroparticles”. Particles are characterized by a six-vector representing the particle’s phase space coordinates and a pair of complex numbers characterizing the particle’s spin. A macroparticle is like a particle with the addition of a 6×6 “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. However macroparticle tracking could be revived in the future if there is a demonstrated need for it.

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 a “beam”. Both types particle tracking are covered in this chapter.

28.1 The coord_struct

The `coord_struct` holds the coordinates of a particle at a given longitudinal position in the lattice. The definition of the `coord_struct` is

```
type coord_struct
  real(rp) vec(6)      ! (x, px, y, py, z, pz)
  real(rp) s            ! Longitudinal position
  real(rp) t            ! Absolute time (not relative to reference).
  complex(rp) spin(2)   ! Spin in spinor notation
  real(rp) field(2)     ! Photon (x, y) field intensity.
  real(rp) phase(2)     ! Photon (x, y) phase.
  real(rp) charge        ! charge in a particle (Coul).
  real(rp) path_len      ! path length (used by coherent photons).
  real(rp) p0c           ! For non-photons: Reference momentum. Negative -> going backwards.
                         ! For photons: Photon momentum (not reference).
  real(rp) beta          ! Velocity / c_light.
  integer ix_ele         ! Index of element particle was tracked through.
```

```

        ! May be -1 or -2 if element is not associated with a lattice.
integer state      ! alive$, lost$, lost_neg_x_aperture$, etc.
integer direction   ! Longitudinal direction of motion = +/- 1
integer species     ! Positron$, proton$, etc.
integer location    ! upstream_end$, inside$, or downstream_end$
end type

```

The `%vec(:)` array defines the phase space coordinates (§13.4). Note that for photons, the definition of the phase space coordinates (§13.5) is different from that used for charged particles.

For charged-particles, the reference momentum in eV is stored in the `%p0c` component. For photons, `%p0c` is the actual (not reference) momentum. For charged-particles, `%p0c` may be negative if the particle is traveling backwards longitudinally. For photons, `%vec(6)` (β_z) will be negative if the photon is going backward.

The normalized velocity v/c is stored in `%beta`.

The `%spin` component gives a particles spin (§19.4) in spinor notation.

The `%s` component gives the absolute s-position of the particle and `%t` gives the absolute time.

The `%field_x` and `%field_y` components are for photon tracking and are in units of field/sqrt(cross-section-area). That is, the square of these units is an intensity. It is up to individual programs to define an overall scaling factor for the intensity if desired.

The `%ix_ele` component gives the index of the element in the `lat%branch(ib)%ele(:)` array that was tracked through. If the element tracked through is not associated with a lattice, `%ix_ele` is set to -1. When initializing a `coord_struct` (see below), `%ix_ele` will be initialized to `not_set$`.

The `%state` component will be one of:

```

alive$
lost$
lost_neg_x_aperture$
lost_pos_x_aperture$
lost_neg_y_aperture$
lost_pos_y_aperture$
lost_z_aperture$

```

The `alive$` setting indicates that the particle is alive. If a particle is “dead”, `%state` component will be set to one of the other settings. The `lost_neg_x_aperture$` setting indicates that the particle was lost at an aperture on the $-x$ side of the element. The `lost_z_aperture$` setting is used to indicate an energy aperture where a particle has been decelerated to the point where it is turning around. The `lost$` setting is used when neither of the `lost_*_aperture$` settings are not appropriate. For example, `lost$` is used in Runge-Kutta tracking when the adaptive step size becomes too small.

Finally the `%location` component Indicates where a particle is longitudinally with respect to the element being tracked. `%location` will be one of:

```

entrance_end$
inside$
exit_end$

```

`entrance_end$` indicates that the particle is at the element’s entrance ($-s$) end and `exit_end$` indicates that the particle is at the element’s exit ($+s$) end. `inside$` indicates that the particle is in between. If the element has edge fields (for example, the `e1` and `e2` edge fields of a bend), a particle at the `entrance_end$` or `exit_end$` is considered to be just outside the element.

To initialize a `coord_struct` so it can be used as the start of tracking, the `init_coord` routine can be used:

```

type (coord_struct) start_orb
real(rp) phase_space_start(6)
...
phase_space_start = [...]
call init_coord (start_orb, phase_space_start, lat%ele(i), lat%param%particle)

```

Here `init_coord` will initialize `start_orb` appropriately for tracking through element `lat%ele(i)` with the particle species set to the species of the reference particle given by `lat%param%particle`.

28.2 Tracking Through a Single Element

`track1` is the routine used for tracking through a single element

```

type (coord_struct), start_orb, end_orb
type (ele_struct) ele
real(rp) start_phase_space(6)
logical err
...
start_phase_space = [...]
call init_coord (start_orb, start_phase_space, ele, photon$)
call track1 (start_orb, ele, end_orb, err_flag = err)
if (.not. particle_is_moving_forward(end_orb)) then
    print *, 'Particle is lost and gone forever...'
endif

```

To check if a particle is still traveling in the forward direction, the `particle_is_moving_forward` routine can be used as shown in the above example.

The “virtual” entrance and exit ends of a lattice element are, by definition, where the physical ends of the element would be if there were no offsets. In particular, if an element has a finite `z_offset` (§24.10), the physical ends will be displaced from the virtual ends. The position `ds` of a particle with respect to the physical entrance end of the element is

```
ds = coord%s - (ele%s + ele%value(z_offset_tot$) - ele%value(1$))
```

When tracking through an element, the starting and ending positions always correspond to the virtual ends. If there is a finite `z_offset`, the tracking of the element will involve tracking through drifts just before and just after the tracking of the body of the element so that the particle ends at the proper virtual exit end.

28.3 Tracking Through a Lattice Branch

When tracking through a lattice branch, one often defines an array of `coord_structs` – one for each element of the lattice branch. In this case, the i^{th} `coord_struct` corresponds to the particle coordinates at the end of the i^{th} element. Since the number of elements in the lattice is not known in advance, the array must be declared to be allocatable. The lower bound of the array must be set to zero to match a `lat%branch(i)%ele(:)` array. The upper bound should be the upper bound of the `%branch(i)%ele(:)` array. The routine `reallocates_coord` will allocate an array of `coord_structs`:

```

type (coord_struct), allocatable :: orbit(:)
type (lat_struct) lat
...
call reallocates_coord (orbit, lat, ix_branch)

```

Alternatively, the `save` attribute can be used 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. Note that in the main program, the `save` attribute is not permitted. If a `coord_struct` array is passed to a routine, the routine must explicitly set the lower bound to zero if the array is not declared as allocatable:

```
subroutine my_routine (orbit1, orbit2)
  use bmad
  implicit none
  type (coord_struct), allocatable :: orbit1(:) ! OK
  type (coord_struct) orbit2(0:) ! Also OK
  ...

```

Declaring the array allocatable is mandatory if the array is to be resized or the array is passed to a routine that declares it allocatable.

For an entire lattice, the `coord_array_struct` can be used to define an array of `coord_array` arrays:

```
type coord_array_struct
  type (coord_struct), allocatable :: orb(:)
end type
```

The routine `reallocate_coord_array` will allocate an `coord_array_struct` instance

```
type (coord_array_struct), allocatable :: all_orbit(:)
type (lat_struct) lat
...
call reallocate_coord_array (all_orbit, lat)
...
```

Once an array of `coord_struct` elements is defined, the `track_all` routine can be used to track through a given lattice branch

```
type (coord_struct), allocatable :: orbit(:)
integer ib, track_state
...
ib = 1 ! Branch to track through
call init_coord(orbit(0), init_phase_space, lat%branch(ib)%ele(0), proton$)
call track_all (lat, orbit, ib, track_state, err_flag)
if (track_state /= moving_forward$) print *, 'Particle lost at element:', track_state
```

After tracking, `orbit(i)` will correspond to the particles orbit at the end of `lat%branch(ib)%ele(i)`.

For routines like `track_all` where an array of `coord_structs` is used, an integer `track_state` argument is provided that is set to `moving_forward$` if the particle survives to the end, or is set to the index of the element at which the particle either hit an aperture or the particle's longitudinal velocity is reversed.

The reason why the reversal of the particle's longitudinal velocity stops tracking is due to the fact that the standard tracking routines, which are *s*-based (that is, use longitudinal position *s* as the independent coordinate), are not designed to handle particles that reverse direction. To properly handle this situation, time-based tracking needs to be used (§28.10). Notice that this is different from tracking a particle in the reversed ($-s$) direction which can be handled by constructing a “reversed” lattice (§28.12).

Alternatively to `track_all`, the routine `track_many` can be used to track through a selected number of elements or to track backwards (See §28.12).

The `track_all` routine serves as a good example of how tracking works. A condensed version of the code is shown in Fig. 28.3. 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} element.

```

1 subroutine track_all (lat, orbit, ix_branch, track_state, err_flag)
2   use bmad
3   implicit none
4   type (lat_struct), target :: lat
5   type (branch_struct), pointer :: branch
6   type (coord_struct), allocatable :: orbit(:)
7   integer, optional :: ix_branch, track_state
8   logical, optional :: err_flag
9   logical err
10
11 !
12
13 branch => lat%param(integer_option(0, ix_branch))
14 branch%param%ix_track = moving_forward
15 if (present(track_state)) track_state = moving_forward\$
16
17 do n = 1, branch%n_ele_track
18   call track1 (orbit(n-1), branch%ele(n), branch%param, orbit(n), err_flag = err)
19   if (.not. particle_is_moving_forward(orbit(n))) then
20     if (present(track_state)) track_state = n
21     orbit(n+1:)%status = not_set$
22   return
23   endif
24 enddo
25 end subroutine

```

Figure 28.1: Condensed track_all code.

28.4 Forking from Branch to Branch

Tracking from a `fork` or `photon_fork` (§3.18) element to the target `branch` is not “automatic”. That is, since the requirements of how to handle forking can vary greatly from one situation to the next, *Bmad* does not try to track from one `branch` to the next in any one of its tracking routines.

The discussion here is restricted to the case where the particle being tracked is simply transferred from the forking element to the target branch. [Thus the subject of photon generation is not covered here.]

There are two cases discussed here. The first case is when a given branch (called `to_branch`) has an associated forking element in the `from_branch` that forks to the beginning of the `to_branch`. Appropriate code is:

```

type (lat_struct), target :: lat ! Lattice
type (branch_struct) :: to_branch ! Given target branch
type (branch_struct), pointer :: from_branch ! Base branch
type (ele_struct), pointer :: fork_ele
type (coord_struct), allocatable :: from_orbit(:), to_orbit(:)
integer ib_from, ie_from

ib_from = to_branch%ix_from_branch

if (ib_from < 0) then
  ! Not forked to ...
else

```

```

from_branch => lat%branch(ib_from)
ie_from = to_branch%ix_from_ele
fork_ele => from_branch%ele(ie_from)
to_orbit(0) = from_orbit(ie_from)
call transfer_twiss (fork_ele, to_branch%ele(0))
endif

```

`from_orbit(0:)` and `to_orbit(0:)` are arrays holding the orbits at the exit end of the elements for the `from_branch` and `to_branch` respectively. The call to `transfer_twiss` transfers the Twiss values to the `to_branch` which can then be propagated through the `to_branch` using `twiss_propagate_all`.

The second case starts with the `fork_ele` forking element. This is similar to the first case but is a bit more general since here the element, called `to_ele` in the `to_branch` that is connected to `fork_ele` need not be the starting element of `to_branch`.

```

type (lat_struct), target :: lat      ! Lattice
type (branch_struct), pointer :: to_branch ! Target branch
type (ele_struct), pointer :: to_ele
type (coord_struct), allocatable :: from_orbit(:), to_orbit(:)
integer ib_to, ie_to

ib_to = nint(fork_ele%value(ix_to_branch$))
ie_to = nint(fork_ele%value(ix_to_element$))

to_branch => lat%branch(ib_to)
to_ele => to_branch%ele(ie_to)
to_orbit(to_ele%ix_ele) = from_orbit(fork_ele%ix_ele)

```

Notice that, by convention, the transferred orbit is located at the exit end of the `to_ele`.

28.5 Multi-turn Tracking

Multi-turn tracking over a branch is simply a matter of setting the coordinates at the beginning zeroth element equal to the last tracked element within a loop:

```

type (lat_struct) lat          ! lattice to track through
type (coord_struct), allocatable :: orbit(:)
...
call reallocate_coord (orbit, lat, ix_branch = 1)
orbit(0)%vec = [0.01, 0.2, 0.3, 0.4, 0.0, 0.0] ! init
do i = 1, n_turns
    call track_all (lat, orbit, 1)
    orbit(0) = orbit(lat%branch(1)%n_ele_track)
end do

```

Often times it is only the root branch, `branch(0)`, that is to be tracked. In this case, the above reduces to

```

type (lat_struct) lat          ! lattice to track through
type (coord_struct), allocatable :: orbit(:)
...
call reallocate_coord (orbit, lat%n_ele_max)
orbit(0)%vec = [0.01, 0.2, 0.3, 0.4, 0.0, 0.0] ! init
do i = 1, n_turns
    call track_all (lat, orbit)

```

```

orbit(0) = orbit(lat%n_ele_track)
end do

```

28.6 Closed Orbit Calculation

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.

28.7 Partial Tracking through elements

There are two routines for tracking partially through an element:

```

twiss_and_track_at_s (lat, s, ele, orb, orb_at_s, ix_branch, err, use_saved_data)
twiss_and_track_intra_ele (ele, param, l_start, l_end, track_entrance,
                           track_exit, orbit_start, orbit_end, ele_start, ele_end, err)

```

Both routines make use of element “slices” (§24.6) which are elements that represent some sub-section of an element. The routine `create_element_slice` can be used to create such slices.

28.8 Apertures

The routine `check_aperture_limit` checks the aperture at a given element. The `ele%aperture_type` component determines the type of aperture. Possible values for `ele%aperture_type` are

```

rectangular$
elliptical$
custom$

```

With `custom$`, a program needs to be linked with a custom version of `check_aperture_limit_custom`.

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 decelerating the particle and the particle did not have enough energy going into the cavity to make it to the exit.

28.9 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  
print *, 'Tracking_method: ', calc_method_name(ele%tracking_method)
```

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. To convert the integer `%tracking_method` value to a string suitable for printing, use the `tracking_method_name` array.

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.

For determining what tracking methods are valid for a given element, use `valid_tracking_method` and `valid_mat6_calc_method` functions

```
print *, 'Method is valid: ', valid_tracking_method(ele, boris$)  
Bmad simulates radiation damping and excitation by applying a kick just before and after each element.
```

28.10 Time Tracking

Time tracking uses time as the independent variable as opposed to the standard s based tracking. Time tracking is useful when a particle’s trajectory can reverse itself longitudinally. For example, low energy particles generated when a relativistic particle hits the vacuum chamber wall are good candidates for time tracking.

Currently, the only `ele%tracking_method` available for time tracking is `time_runge_kutta$`. Time tracking needs extra bookkeeping due to the fact that the particle may reverse directions. See the `dark_current_tracker` program as an example.

28.11 Taylor Maps

A list of routines for manipulating Taylor maps is given in §35.34. The order of the Taylor maps is set in the lattice file using the `parameter` statement (§8.1). 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 regardless 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 (§18.1). 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 (§18.1) is given by `%ele%integrator_order`. PTC (§30) currently implements integrators of order 2, 4, or 6.

28.12 Reverse Tracking

Reverse tracking in when a particle goes in the direction of decreasing `s`. The `track_many` routine can be used for this. See the `track_many` routine for more details.

For static fields 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 or RF cavities break this symmetry.

28.13 Beam (Particle Distribution) Tracking

Tracking with multiple particles is done with a `beam_struct` instance:

```
type beam_struct
    type (bunch_struct), allocatable :: bunch(:)
end type
```

A `beam_struct` is composed of an array of bunches of type `bunch_struct`:

```
type bunch_struct
    type (coord_struct), allocatable :: particle(:)
    integer, allocatable :: ix_z(:) ! bunch%ix_z(1) is index of head particle, etc.
    real(rp) charge_tot ! Total charge in bunch (Coul).
    real(rp) charge_live ! Total charge of live particles in bunch (Coul).
    real(rp) z_center ! Longitudinal center of bunch (m). Note: Generally, z_center of
                      ! bunch #1 is 0 and z_center of the other bunches is negative.
    real(rp) t_center ! Center of bunch creation time relative to head bunch.
    integer species ! electron$, proton$, etc.
    integer ix_ele ! Element this bunch is at.
    integer ix_bunch ! Bunch index. Head bunch = 1, etc.
end type
```

The `bunch_struct` has an array of particles of type `coord_struct` (§28.1).

Initializing a `beam_struct` to conform to some initial set of Twiss parameters and emittances is done using the routine `init_beam_distribution`:

```
type (lat_struct) lat
type (beam_init_struct) beam_init
type (beam_struct) beam
...
call init_beam_distribution (lat%ele(0), lat%param, beam_init, beam)
```

The `lat%ele(0)` argument, which is of type `ele_struct`, gives the twiss parameters to initialize the beam to. In this case, we are starting tracking from the beginning of the lattice. The `beam_init` argument which is of type `beam_init` gives additional information, like emittances, which is needed to initialize the beam. See Section §10.2 for more details.

Tracking a beam is done using the `track_beam` routine

```
type (lat_struct) lat
type (beam_struct) beam
...
call track_beam (lat, beam)
```

or, for tracking element by element, `track1_beam` can be used.

For analyzing a bunch of particles, that is, for computing such things as the sigma matrix from the particle distribution, the `calc_bunch_params` routine can be used.

28.14 Spin Tracking

See Section §5.3 for a list of spin tracking methods available. To turn spin tracking on, use the `bmad_com%spin_tracking_on` flag. `ele%spin_tracking_method` sets the method used for spin tracking. After properly initializing the spin in the `coord_struct`, calls to `track1` will track both the particle orbit and the spin.

28.15 X-ray Targeting

X-rays can have a wide spread of trajectories resulting in many “doomed” photons that hit apertures or miss the detector with only a small fraction of “successful” photons actually contributing to the simulation results. The tracking of doomed photons can therefore result in an appreciable lengthening of the simulation time. To get around this, *Bmad* can be setup to use what is called “targeting” to minimize the number of doomed photons generated.

This is explained in detail in §20.5. The coordinates of the four or eight corner points and the center target point are stored in:

```
gen_ele%photon%target%corner(:)%r(1:3)  
gen_ele%photon%target%center%r(1:3)
```

where `gen_ele` is the generating element (not the element with the aperture).

Chapter 29

Miscellaneous Programming

29.1 Custom Elements and Custom Calculations

Normally a “custom” calculation is a calculation, done by *Bmad*, that is instead handled by having the appropriate *Bmad* code call the appropriate custom code. Implementing custom calculations involve writing custom code and linking this code into a program. There are essentially two ways to do custom calculations. One method involves using a `custom` element (§3.9). The other method involves setting the appropriate component of an element to `custom`. An appropriate component is one of

<code>tracking_method</code>	§5.1
<code>mat6_calc_method</code>	§5.2
<code>field_calc</code>	§5.4
<code>aperture_type</code>	§4.6

There are seven routines that implement custom calculations:

<code>check_aperture_limit_custom</code>
<code>em_field_custom</code>
<code>init_custom</code>
<code>make_mat6_custom</code>
<code>make_mat6_custom2</code>
<code>radiation_integrals_custom</code>
<code>track1_custom</code>
<code>track1_custom2</code>
<code>track1_bunch_custom</code>
<code>track1_spin_custom</code>
<code>wall_hit_handler_custom</code>

[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 exception is the dummy `init_custom` routine which will simply do nothing when called.

If a particular custom routine is not called in a program, then, obviously, a non-dummy version does not have to be implemented. To link in a non-dummy version of a custom routine, put the code file in the same directory as the program code and relink.

While coding a custom routine, it is important to remember that it is *not* permissible to modify any routine argument that does not appear in the list of output arguments shown in the comment section at the top of the file.

The `init_custom` routine is called by `bmad_parser` at the end of parsing for any lattice element that is a `custom` element or has set any one of the element components as listed above to `custom`. The `init_custom` routine can be used to initialize the internals of the element. For example, consider a `custom` element defined in a lattice file by

```
my_element: custom, val1 = 1.37, descrip = "field.dat", mat6_calc_method = tracking
```

In this example, the `descrip` (§4.2) component is used to specify the name of a file that contains parameters for this element. When `init_custom` is called for this element (see below), the file can be read and the parameters stored in the element structure. Besides the `ele%value` array, parameters may be stored in the general use components given in §24.18.

The `make_mat6_custom` and `make_mat6_custom2` routines are called by the `make_mat6` routine when calculating the 6x6 transfer matrix (Jacobian) through an element. Two routines are needed here due to the specifics of the coding for the `make_mat6` routine. `make_mat6` is divided into three sections: The code in the first section is at the start of `make_mat6` and looks like:

```
if (ele%mat6_calc_method == custom$) then
    call make_mat6_custom (ele, param, a_start_orb, a_end_orb)
    return
endif
```

Call this the “custom” section. The next section in `make_mat6` uses `ele%mat6_calc_method` to call the appropriate routine for the indicated calculational method. This section looks like:

```
select case (ele%mat6_calc_method)
case (custom2$)
    call make_mat6_custom2 (ele, param, a_start_orb, a_end_orb)
case (taylor$)
    call make_mat6_taylor (ele, param, a_start_orb)
    if (.not. end_input) call track1_taylor (a_start_orb, ele, param, a_end_orb)
case (bmad_standard$)
    call make_mat6_bmad (ele, param, a_start_orb, a_end_orb, end_in, err)
    ... etc ...
end select
```

Let us call this the “method specific” section. Notice that when `ele%mat6_calc_method` is set to `custom2$` – not `custom$` – then `make_mat6_custom2` is called. The last section, call it the “common” section, is everything else. The common section involves modifying the transfer matrix to include the effect of space charge, symplectifying the matrix, etc.

The two custom routines are meant to handle two different cases: `make_mat6_custom` is used when the entire matrix calculation needs to be customized. `make_mat6_custom2` is used if it is desired to not to “reinvent the wheel” and to have `make_mat6` handle the “common” calculation in the common section. Notice that in the lattice file, there is no `custom2` setting for the `mat6_calc_method` element parameter. This is done to hide implementation details from the user. Thus, to use `make_mat6_custom2`, `make_mat6_custom` must look like:

```
ele%mat6_calc_method = custom2$ ! Was, of course, custom$
call make_mat6 (ele, param, start_orb, end_orb) ! Recursive call back to make_mat6
ele%mat6_calc_method = custom$ ! Restore
```

The `track1_custom` and `track1_custom2` routines are called by the `track1` routine when calculating the transfer matrix through an element. Two routines are needed here due to the specifics of the coding for the `track1` routine. See the discussion on the `make_mat6_custom` and `make_mat6_custom2` routines above.

The `check_aperture_limit_custom` routine is used to check if a particle has hit an aperture in tracking. It is called by the standard *Bmad* routine `check_aperture_limit` when `ele%aperture_type` is set to

`custom$`. A `custom` element has the standard limit attributes (§4.6) so a `custom` element does not have to implement custom aperture checking code.

The `em_field_custom` routine is called by the electro-magnetic field calculating routine `em_field_calc` when `ele%field_calc` is set to `custom$`. As an alternative to `em_field_custom`, a `custom` element can use the `field` attribute (§4.13) to characterize the element's electromagnetic fields.

Note: When tracking through a `patch` element, the first step is to transform the particle's coordinates from the entrance frame to the exit frame. This is done since it simplifies the tracking. [The criterion for stopping the propagation of a particle through a `patch` is that the particle has reached the exit face and the calculation to determine if a particle has reached the exit face is simplified if the particle's coordinates are expressed in the coordinate frame of the exit face.] Thus for `patch` elements, unlike all other elements, the particle coordinates passed to `em_field_custom` are the coordinates with respect to the exit coordinate frame and not the entrance coordinate frame. If field must be calculated in the entrance coordinate frame, a transformation between entance and exit frames must be done:

```

subroutine em_field_custom (ele, param, s_rel, time, orb, &
                           local_ref_frame, field, calc_dfield, err_flag)
use lat_geometry_mod
...
real(rp) w_mat(3,3), w_mat_inv(3,3), r_vec(3), r0_vec(3)
real(rp), pointer :: v(:)
...
! Convert particle coordinates from exit to entrance frame.
v => ele%value ! v helps makes code compact
call floor_angles_to_w_mat (v(x_pitch$), v(y_pitch$), v(tilt$), w_mat, w_mat_inv)
r0_vec = [v(x_offset$), v(y_offset$), v(z_offset$)]
r_vec = [orb%vec(1), orb%vec(3), s_rel] ! coords in exit frame
r_vec = matmul(w_mat, r_vec) + r0_vec ! coords in entrance frame

! Calculate field and possibly field derivative
...
!

! Convert field from entrance to exit frame
field%E = matmul(w_mat_inv, field%E)
field%B = matmul(w_mat_inv, field%B)
if (logic_option(.false., calc_dfield)) then
    field%dE = matmul(w_mat_inv, matmul(field%dE, w_mat))
    field%dB = matmul(w_mat_inv, matmul(field%dB, w_mat))
endif

```

The `wall_hit_handler_custom` routine is called when the Runge-Kutta tracking code `odeint_bmad` detects that a particle has hit a wall (§4.9). [This is separate from hitting an aperture that is only defined at the beginning or end of an lattice element.] The dummy `wall_hit_handler_custom` routine does nothing. To keep tracking, the particle must be marked as alive

```

subroutine wall_hit_handler_custom (orb, ele, s, t)
...
orb%state = alive$ ! To keep on truckin'
...

```

Note: `odeint_bmad` normally does not check for wall collisions. To change the default behavior, the `runge_kutta_com` common block must modified. This strucutre is defined in `runge_kutta_mod.f90`:

```

type runge_kutta_common_struct
logical :: check_wall_aperture = .false.

```

```

integer :: hit_when = outside_wall$    ! or wall_transition$
end type

type (runge_kutta_common_struct), save :: runge_kutta_com

```

To check for wall collisions, the `%check_wall_aperture` component must be set to true. The `%hit_when` components determines what constitutes a collision. If this is set to `outside_wall$` (the default), then any particle that is outside the wall is considered to have hit the wall. If `%hit_when` is set to `wall_transition$`, a collision occurs when the particle crosses the wall boundary. The distinction between `outside_wall$` and `wall_transition$` is important if particles are to be allowed to travel outside the wall.

29.2 Physical and Mathematical Constants

Common physical and mathematical constants that can be used in any expression are defined in the file:

```
sim_utils/interfaces/physical_constants.f90
```

The following constants are defined

```

pi = 3.14159265358979d0
twopi = 2 * pi
fourpi = 4 * pi
sqrt_2 = 1.41421356237310d0
sqrt_3 = 1.73205080757d0
complex: i_imaginary = (0.0d0, 1.0d0)

e_mass = 0.51099906d-3    ! DO NOT USE! In GeV
p_mass   = 0.938271998d0    ! DO NOT USE! In GeV

m_electron = 0.51099906d6  ! Mass in eV
m_proton   = 0.938271998d9 ! Mass in eV

c_light = 2.99792458d8          ! speed of light
r_e = 2.8179380d-15           ! classical electron radius
r_p = r_e * m_electron / m_proton ! proton radius
e_charge = 1.6021892d-19       ! electron charge

h_planck = 4.13566733d-15      ! eV*sec Planck's constant
h_bar_planck = 6.58211899d-16 ! eV*sec h_planck/twopi

mu_0_vac = fourpi * 1e-7        ! Permeability of free space
eps_0_vac = 1 / (c_light**2 * mu_0_vac) ! Permittivity of free space

classical_radius_factor = r_e * m_electron ! Radiation constant

g_factor_electron = 0.001159652193 ! Anomalous gyro-magnetic moment
g_factor_proton   = 1.79285        ! Anomalous gyro-magnetic moment

```

29.3 Global Coordinates and S-positions

The routine `lat_geometry` will compute the global floor coordinates at the end of every element in a lattice. `lat_geometry` works by repeated calls to `ele_geometry` which takes the floor coordinates at the end of one element and calculates the coordinates at the end of the next. For conversion between orientation matrix \mathbf{W} (§13.3) and the orientation angles θ, ϕ, ψ , the routines `floor_angles_to_w_mat` and `floor_w_mat_to_angles` can be used.

The routine `s_calc` calculates the longitudinal s positions for the elements in a lattice.

29.4 Reference Energy and Time

The reference energy and time for the elements in a lattice is calculated by `lat_compute_ref_energy_and_time`. The reference energy associated with a lattice element is stored in

```
ele%value(E_tot_start$)      ! Total energy at upstream end of element (eV)
ele%value(p0c_start$)        ! Momentum * c_light at upstream end of element (eV)
ele%value(E_tot$)            ! Total energy at downstream end (eV)
ele%value(p0c$)              ! Momentum * c_light at downstream end(eV)
```

Generally, the reference energy is constant throughout an element so that `%value(E_tot_start$) = %value(E_tot$)` and `%value(p0c_start$) = %value(p0c$)`. Exceptions are elements of type:

```
custom,
em_field,
hybrid, or
lcavity
```

In any case, the starting `%value(E_tot_start$)` and `%value(p0c_start$)` values of a given element will be the same as the ending `%value(E_tot$)` and `%value(p0c$)` energies of the previous element in the lattice.

The reference time and reference transit time is stored in

```
ele%ref_time                  ! Ref time at downstream end
ele%value(delta_ref_time$)
```

The reference orbit for computing the reference energy and time is stored in

```
ele%time_ref_orb_in           ! Reference orbit at upstream end
ele%time_ref_orb_out          ! Reference orbit at downstream end
```

Generally `ele%time_ref_orb_in` is the zero orbit. The exception comes when an element is a `super_slave`. In this case, the reference orbit through the `super_slaves` of a given `super_lord` is constructed to be continuous. This is done for consistancy sake. For example, to ensure that when a marker is superimposed on top of a wiggler the reference orbit, and hence the reference time, is not altered.

`group` (§3.20), `overlay` (§3.33), and `super_lord` elements inherit the reference from the last slave in their slave list (§25.5). For `super_lord` elements this corresponds to inheriting the reference energy of the slave at the downstream end of the `super_lord`. For `group` and `overlay` elements a reference energy only makes sense if all the elements under control have the same reference energy.

Additionally, photonic elements like `crystal`, `capillary`, `mirror` and `multilayer_mirror` elements have an associated photon reference wavelength

```
ele%value(ref_wavelength$)     ! Meters.
```

29.5 Common Structures

There are two common variables 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_com` structure is documented in Section §10.1. The `bmad_status_struct` The `bmad_status_struct` structure is, along with the default values:

```
type bmad_status_struct
    logical :: ok          = .true.    ! Error flag
    logical :: type_out    = .true.    ! Print error messages?
    logical :: exit_on_error = .true.    ! Exit program on error?
end type
```

The `%ok` component can be set to indicate that an error has been encountered in a routine. Using `%ok` is deprecated for a number of reasons. For one, it is not thread safe. The long term goal is to replace this by adding error flag arguments to the appropriate routines.

The `%type_out` component tells a routine if warning or error messages should be printed. With Severe errors, the code may ignore `%type_out` and print an error message irregardless.

The `%exit_on_error` component tell a routine if it is ok to stop a program on a severe error.

Chapter 30

Etienne Forest's PTC/FPP

The PTC/FPP (Polymorphic Tracking Code / Fully Polymorphic Package) library of Etienne Forest handles Taylor maps to any arbitrary order. this is also known as Truncated Power Series Algebra (TPSA). The core Differential Algebra (DA) package used by PTC was developed by Martin Berz[Berz89]. For more information see the FPP/PTC manual[Forest02].

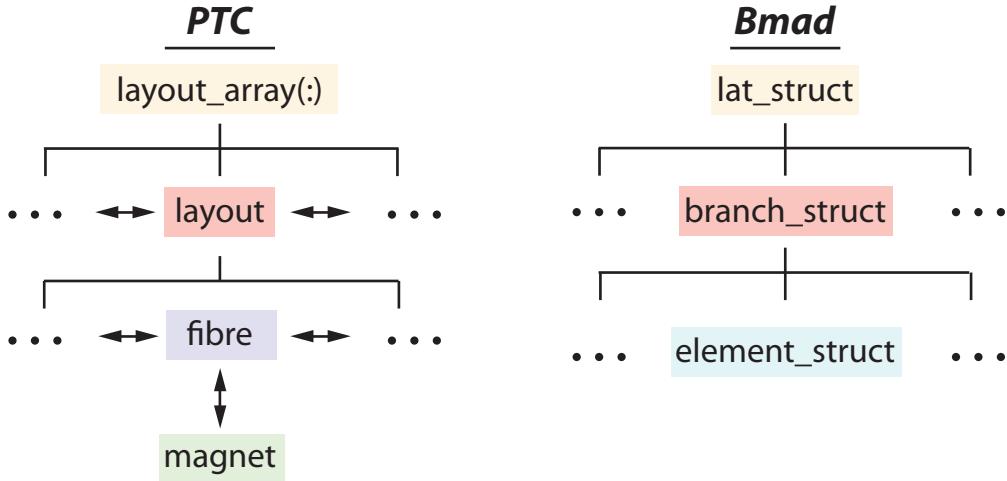


Figure 30.1: Simplified diagram showing the organization of the major PTC structures involved in defining a lattice contrasted with *Bmad*. With PTC, the top level structure is an array of `layout_array` structures. Each `layout_array` holds a `layout` structure. A `layout` structure has pointers to the previous and next `layouts` indicated by the horizontal arrows. Each `layout` has pointers to a linked list of `fibre` structures. The `fibre` structures represent the reference trajectory through an element. Each `fibre` structure has a pointer to a `element` and an `elementp` structures which represent the physical element. With *Bmad*, the `lat_struct` roughly corresponds to the PTC `layout_array(:)`, the `branch_struct` roughly corresponds to the PTC `layout` and the `element_struct` roughly corresponds to the PTC `fibre`, `element` and `elementp` structures.

30.1 Accessing PTC

Bmad uses a `lat_struct` structure to hold the information on a machine and a `lat_struct` has an array of `branch_structs` with each `branch_struct` holding an array of `ele_structs`. The `ele_struct` holds the information on the individual elements. An `ele_struct` holds information about both the physical element and the reference orbit through it.

PTC has a somewhat different philosophy as illustrated in Fig. 30.1. A PTC `layout` structure is roughly equivalent to a *Bmad* `branch_struct`. A `layout` has a pointer to a linked list of `fibre` structures. Each `fibre` has a pointer to a `magnet` structure which holds the information about the physical element and each `fibre` holds information about the reference orbit through the element.

separates the actual uses a `layout` structure

30.2 Phase Space

PTC uses different longitudinal phase space coordinates compared to *Bmad*. *Bmad*'s phase space coordinates are (§13.4)

$$(x, p_x, y, p_y, z, p_z) \quad (30.1)$$

In PTC one can choose between several different coordinate systems. The one that *Bmad* uses is

$$(x, p_x, y, p_y, p_t, c\Delta t) \quad (30.2)$$

where

$$p_t = \frac{\Delta E}{P_0} \quad (30.3)$$

This choice of phase space is set in `set_ptc`. Specifically, the PTC global variable `DEFAULT`, which is of type `internal_states`, has the `%time` switch set to `True`.

`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 §35.29 for more details.

30.3 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.

30.4 Correspondence between Bmad elements and PTC fibres

When a PTC `layout` is created from a *Bmad* `lat_struct` instance using the routine `lat_to_ptc_layout`, the correspondence between the *Bmad* elements and the PTC fibres is maintained through the `ele%ptc_fiber` pointer. The following rules apply:

1. There will be marker `fibres` at the beginning and end of the `layout`. The beginning `fibre` will correspond to `branch%ele(0)`. The end `fibre` will not have a corresponding *Bmad* element.
2. Generally there will be a one-to-one correspondence between `fibres` and `branch%ele` elements. The exception is where a “hard edge” model is used for tracking. In this case, there will be three `fibres` for the *Bmad* element: Two drift `fibres` with a `fibre` of the appropriate type inbetween. In this case, `ele%ptc_fiber` will point to the last (drift) `fibre`.

Remember: The attributes like reference energy, etc. for a *Bmad* `ele_struct` instance are referenced to the exit end of the element. For PTC the reference edge for a `fibre` is the entrance end.

30.5 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_taylorstructure`:

```
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
```

30.6 Patches

There is a significant difference between how patches are treated in PTC and *Bmad*. In PTC, a patch is just thought of as a coordinate transformation for propagating a particle from one **fibre** to the next. As such, the **patch** is part of a **fibre**. That is, any **fibre** representing tracking through quadrupoles, bends, etc. will have patches for the entrance and exit ends of the **fibre**.

With *Bmad*, on the other hand, a **patch** is a “first class” element on par with all other elements be they quadrupoles, bends, etc. When translating a **patch** from *Bmad* to PTC, the **patch** is represented in PTC as a **marker** element with a patch at the exit end.

30.7 Number of Integration Steps

“Drift like” elements in PTC will use, by default, only one integration step. *Bmad* uses the default when translating from *Bmad* lattice elements to PTC fibres. The *Bmad* lattice elements that are drift like are:

```
drift
ecollimator
instrument
monitor
pipe
rcollimator
```

Chapter 31

OPAL

OPAL (Object Oriented Parallel Accelerator Library) is a tool for charged-particle optic calculations in large accelerator structures and beam lines including 3D space charge. OPAL is built from first principles as a parallel application, OPAL admits simulations of any scale: on the laptop and up to the largest High Performance Computing (HPC) clusters available today. Simulations, in particular HPC simulations, form the third pillar of science, complementing theory and experiment.

OPAL includes various beam line element descriptions and methods for single particle optics, namely maps up to arbitrary order, symplectic integration schemes and lastly time integration. OPAL is based on IPPL (Independent Parallel Particle Layer) which adds parallel capabilities. Main functions inherited from IPPL are: structured rectangular grids, fields and parallel FFT and particles with the respective interpolation operators. Other features are, expression templates and massive parallelism (up to 8000 processors) which makes it possible to tackle the largest problems in the field.

The manual can be obtained at

<http://amas.web.psi.ch/docs/opal/>

31.1 Phase Space

OPAL uses different longitudinal phase space coordinates compared to *Bmad*. *Bmad*'s phase space coordinates are

$$(x, p_x/p_0, y, p_y/p_0, -\beta c(t - t_0), (p - p_0)/p_0) \quad (31.1)$$

OPAL uses

$$(x, \gamma\beta_x, y, \gamma\beta_y, z, \gamma\beta_z) \quad (31.2)$$

`convert_particle_coordinates_s_to_t` and `convert_particle_coordinates_s_to_t` are conversion routines ...

Chapter 32

C++ Interface

To ease the task of using *C++* routines with *Bmad*, there is a library called `cpp_bmad_interface` which implements a set of *C++* classes in one-to-one correspondence with the major *Bmad* structures. 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.

The list of all classes is given in the file

```
cpp_bmad_interface/include/cpp_bmad_classes.h
```

The general rule is that the equivalent class to a *Bmad* structure named `xxx_struct` will be named `CPP_xxx`. Additionally, for each *Bmad* structure, there is a opaque class named `Bmad_xxx_class` for use in the translation code discussed below. The names of these opaque classes have the form `Bmad_xxx_class` and are used to define pointer instances in routine argument lists.

32.1 C++ Classes and Enums

Generally, The *C++* classes have been set up to simply mirror the corresponding *Bmad* structures. For example, the `CPP_lat` class has a string component named `.version` that mirrors the `%version` component of the `lat_struct` structure. There are some exceptions. For example, structure components that are part of PTC (§1.5) are not present in the classes.

While generally the same component name is used for both the *Bmad* structures and the *C++* classes, in the case where there is a *C++* reserved word conflict, the *C++* component name will be different.

A header file `bmad_enums.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++*.

All of the *C++* class components that are arrays or matrices are zero based so that, for example, the index of the `.vec[i]` array in a `CPP_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. The exception to this rule is the `%value(:)` array of the `ele_struct` which has a span from 1 to `num_ele_attrib$`. In this case, To keep the conversion of the constructs like `ele%value(k1$)` consistant, the corresponding `ele.value[]` array has goes from 0 to `Bmad::NUM_ELE_ATTRIB` with the 0th element being unused.

```

1 subroutine f_test
2   use bmad_cpp_convert_mod
3   implicit none
4
5   interface
6     subroutine cpp_routine (f_lat, c_coord) bind(c)
7       import f_lat, c_ptr
8       type (lat_struct) :: f_lat
9       type (c_ptr), value :: c_coord
10      end subroutine
11    end interface
12
13   type (lat_struct), target :: lattice // lattice on Fortran side
14   type (coord_struct), target :: orbit
15   type (c_ptr), value :: c_lat
16   !
17   call lat_to_c (c_loc(lattice), c_lat) ! Fortran side convert
18   call cpp_routine (c_lat, c_loc(orbit)) ! Call C++ routine
19   call lat_to_f (c_lat, c_loc(lattice)) ! And convert back
20 end subroutine

```

Figure 32.1: Example Fortran routine calling a $C\+$ routine.

```

1 #include "cpp_bmad_classes.h"
2
3 using namespace Bmad;
4
5 extern "C" cpp_routine (CPP_lat& c_lat, Bmad_coord_class* f_coord, f_lat) {
6   CPP_coord c_coord;
7   coord_to_c (f_coord, c_coord);           // C++ side convert
8   // ... do calculations ...
9   cout << c_lat.name << " " << c_lat.ele[1].value[K1] << endl;
10  coord_to_f (c_coord, f_coord);           // And convert back
11 }

```

Figure 32.2: Example $C\+$ routine callable from a Fortran routine.

32.2 Conversion Between Fortran and $C\+$

A simple example of a Fortran routine calling a $C\+$ routine is shown in Figs. 32.1 and 32.2. Conversion between structure and classes can happen on either the Fortran side or the $C\+$ side. In this example, the `lat_struct` / `CPP_lat` conversion is on the Fortran side and the `coord_struct` / `CPP_coord` is on the $C\+$ side.

On the Fortran side, the interface block defines the argument list of the $C\+$ routine being called.

On the $C\+$ side, `f_coord` is an instance of the `Bmad_coord_class` opaque class.

A $C\+$ routine calling a Fortran routine has a similar structure to the above example. The interface block in Fig. 32.1 can be used as a prototype. For additional examples of conversion between Fortran and $C\+$, look at the test code in the directory

`cpp_bmad_interface/interface_test`

Chapter 33

Quick Plot Plotting

The plotting package included in the *Bmad* distribution is PGPLOT (see §22.1). 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 `dcs16@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.

```

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", title = "That Darn Graph")
58   call qp_draw_data (x, t, draw_line = .false., symbol_every = 4)
59   call qp_restore_state
60 end subroutine
61 end program

```

Figure 33.1: Quick Plot example program.

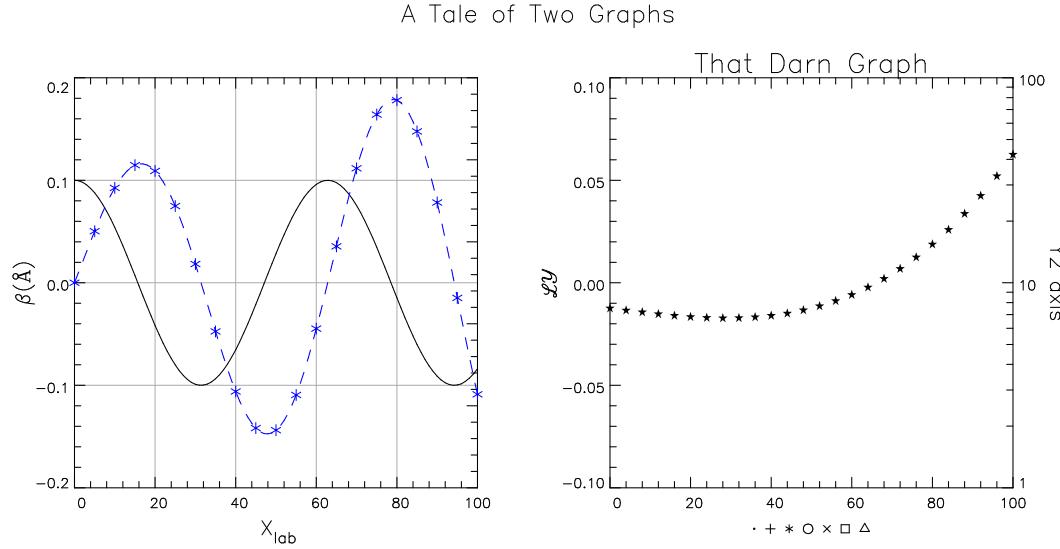


Figure 33.2: Output of plot_example.f90.

33.1 An Example

An example of how *Quick Plot* can be used in a program is shown in Fig. 33.1. 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 Fig. 33.2 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
...

```

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 forth 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 Fig. 33.2, 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 an 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.

33.2 Plotting Coordinates

Quick Plot uses the following concepts as shown in Fig. 33.3

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
<hr/>	
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 Fig. 33.3, 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).

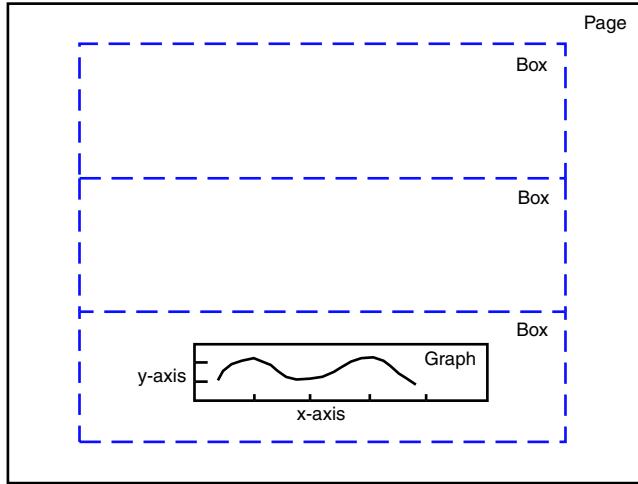


Figure 33.3: A Graph within a Box within a Page.

33.3 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/GRAFH/LB and the set default is INCH/PAGE/LB. Use `qp_set_parameters` to change this.

Examples:

```
"DATA"           -- This is the draw default.
"DATA/GRAFH/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.
```

33.4 Y2 and X2 axes

The top and right axes of a graph are known as X2 and Y2 respectively as shown in Fig. 33.3. 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
```

33.5 Text

PGPLOT defines certain escape sequences that can be used in text strings to draw Greek letters, etc. These escape sequences are given in Table 33.2.

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.
```

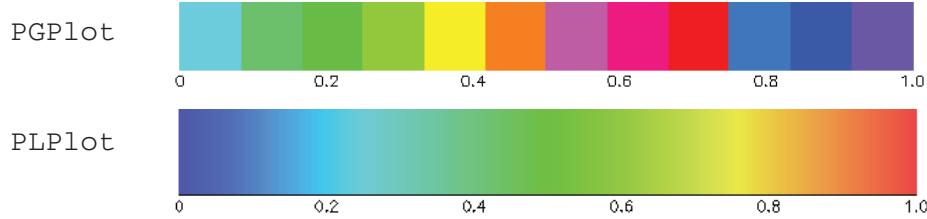


Figure 33.4: Continuous colors using the function `pg_continuous_color` in PGPlot and PLPlot. Typical usage: `call qp_routine(..., color = pg_continuous_color(0.25_rp), ...)`

33.6 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 Fig. 33.1. 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\$	

Integers from [17, (largest integer)] represent continuous colors. The function `pq_continuous_color` maps [0.0, 1.0] to these integers. See Fig. 33.4.

The fill styles are:

1 -- solid_fill\$
2 -- no_fill\$
3 -- hatched\$
4 -- cross_hatched\$

The symbol types are:

0 -- square_sym\$

0	1	2	3	4
□	•	+	*	○
×	□	△	⊕	○
◇	◇	☆	▲	+ x
✡	■	●	★	□
◦	◦	◦	◦	○
25	26	27	28	29
↑	↓	-1	-2	-3
-4	-5	-6	-7	-8

Table 33.1: Plotting Symbols at Height = 40.0

\u	Start a superscript, or end a subscript
\d	Start a subscript, or end a superscript (note that \u and \d must always be used in pairs)
\b	Backspace (i.e., do not advance text pointer after plotting the previous character)
\fn	Switch to Normal font (1)
\fr	Switch to Roman font (2)
\fi	Switch to Italic font (3)
\fs	Switch to Script font (4)
\\"	Backslash character (\)
\x	Multiplication sign (x)
\.	Centered dot (·)
\A	Angstrom symbol (Å)
\gx	Greek letter corresponding to roman letter x
\mn \mnn	Graph marker number n or nn (1-31)
\(nnnn)	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 “”. This makes a number of special characters (e.g., mathematical, musical, astronomical, and cartographical symbols) available.

Table 33.2: PGPLT Escape Sequences.

```

1 -- dot_sym$
2 -- plus_sym$
3 -- times_sym$
4 -- circle_sym$
5 -- x_sym$
7 -- triangle_sym$
8 -- circle_plus_sym$
9 -- circle_dot_sym$
10 -- square_concave_sym$
11 -- diamond_sym$
12 -- star5_sym$
13 -- triangle_filled_sym$
14 -- red_cross_sym$
15 -- star_of_david_sym$
16 -- square_filled_sym$
17 -- circle_filled_sym$
18 -- star5_filled_sym$

```

Beside this list, PGPLT maps other numbers onto symbol types. The PGPLT 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.

```

Table 33.1 shows some of the symbols and there associated numbers. Note: At constant height PGPLT gives symbols of different size. To partially overcome this, *Quick Plot* scales some of the symbols to give a more uniform appearance. Table 33.1 was generated using a height of 40 via the call

Roman	a	b	g	d	e	z	y	h	i	k	l	m
Greek	α	β	γ	δ	ϵ	ζ	η	θ	ι	κ	λ	μ
Roman	n	c	o	p	r	s	t	u	f	x	q	w
Greek	ν	ξ	o	π	ρ	σ	τ	υ	ϕ	χ	ψ	ω
Roman	A	B	G	D	E	Z	Y	H	I	K	L	M
Greek	A	B	Γ	Δ	E	Z	H	Θ	I	K	Λ	M
Roman	N	C	O	P	R	S	T	U	F	X	Q	W
Greek	N	Ξ	O	Π	P	Σ	T	Υ	ϕ	X	Ψ	Ω

Table 33.3: Conversion for the string " $\backslash g<r>$ " where " $<r>$ " is a Roman character to the corresponding Greek character.

```
call qp_draw_symbol (0.5_rp, 0.5_rp, "%BOX", k, height = 40.0_rp)
```

Table 33.3 shows how the character string " $\backslash g<r>$ ", where " $<r>$ " is a Roman letter, map onto the Greek character set.

33.7 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
  integer label_color      ! black$ (default), red$, etc.
```

```
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 major_div_nominal ! Nominal value. Def = 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 ! 'LINEAR' (default), 'LOG', or 'CUSTOM'.
character(16) bounds ! 'GENERAL' (default), 'ZERO_AT_END', ZERO_SYMMETRIC.
integer tick_side ! +1 = draw to the inside (def), 0 = both, -1 = outside.
integer number_side ! +1 = draw to the inside, -1 = outside (default).
logical draw_label ! Draw the label? Default = True.
logical draw_numbers ! Draw the numbering? Default = True.
end type
type qp_plot_struct
    character(80) :: title = ''
    type (qp_axis_struct) x, y, x2, y2
    type (qp_axis_struct), pointer :: xx, yy ! Pointer to axes used for plotting.
    logical :: draw_box = .true.
    logical :: draw_title = .true.
    logical :: draw_grid = .true.
    logical :: x2_mirrors_x = .true.
    logical :: y2_mirrors_y = .true.
    logical :: xx_points_to_x
    logical :: yy_points_to_y
end type
```


Chapter 34

Helper Routines

This chapter gives an overview of various computational helper routines.

34.1 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 (§22.1 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[Storn96]. 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.

34.2 Matrix Manipulation

There are a number of *Bmad* routines for matrix manipulation as listed in §35.20. 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 `mat_inverse` `mat_make_unit`

```
real(rp) mat(6,6), mat_inv(6,6)
call mat_make_unit (mat)      ! make a unit matrix
call mat_inverse (mat, mat_inv) ! Compute the inverse matrix.
```


Chapter 35

Bmad Library Routine List

Below are a list of *Bmad* and *sim_utils* routines sorted by their functionality. Use the `getf` and `listf` ([§22.2](#)) 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	35.1
Beam: Tracking and Manipulation	35.2
Branch Handling	35.3
Coherent Synchrotron Radiation (CSR)	35.4
Collective Effects	35.5
Electro-Magnetic Fields	35.6
Helper Routines: File, System, and IO	35.7
Helper Routines: Math (Except Matrix)	35.8
Helper Routines: Matrix	35.9
Helper Routines: Miscellaneous	35.10
Helper Routines: String Manipulation	35.11
Inter-Beam Scattering (IBS)	35.12
Lattice: Informational	35.15
Lattice: Element Manipulation	35.13
Lattice: Geometry	35.14
Lattice: Low Level Stuff	35.16
Lattice: Manipulation	35.17
Lattice: Miscellaneous	35.18
Lattice: Reading and Writing Files	35.19
Matrices	35.20
Matrix: Low Level Routines	35.21
Measurement Simulation Routines	35.22
Multipass	35.23
Multipoles	35.24
Optimizers (Nonlinear)	35.25
Overload Equal Sign	35.26
Particle Coordinate Stuff	35.27
Photon Routines	35.28
PTC Interface	35.29
Quick Plot	35.30
Spin	35.31
Transfer Maps: Routines Called by make_mat6	35.32
Transfer Maps: Complex Taylor Maps	35.33
Transfer Maps: Taylor Maps	35.34
Tracking: Tracking and Closed Orbit	35.35
Tracking: Low Level Routines	35.36
Tracking: Mad Routines	35.37
Tracking: Routines Called by track1	35.35
Twiss and Other Calculations	35.39
Twiss: 6-Dimensional	35.40
Wake Fields	35.41
Deprecated	35.42

35.1 Beam: Low Level Routines

The following helper routines are generally not useful for general use.

bend_edge_kick (orb, ele, param, particle_at, mat6)

Subroutine to track through the edge field of an sbend. Reverse tracking starts with the particle just outside the bend and

find_bunch_sigma_matrix (particle, charge, avg, sigma, sigma_s)

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)

Routine to order the particles longitudinally in terms of decreasing %vec(5). That is from large z (head of bunch) to small z.

track1_beam (beam_start, lat, ele, beam_end, err)

Routine to track a beam of particles through a single element. Overloaded by `track1_beam`.

track1_bunch (bunch_start, lat, ele, bunch_end, err)

Routine to track a bunch of particles through an element.

track1_bunch_hom (bunch_start, ele, param, bunch_end)

Routine to track a bunch of particles through an element.

35.2 Beam: Tracking and Manipulation

See §28.13 for a discussion of using a collection of particles to simulate a bunch.

bbi_kick (x_norm, y_norm, r, kx, ky)

Routine to compute the normalized kick due to the beam-beam interaction using the normalized position for input.

calc_bunch_params (bunch, bunch_params, err, print_err)

Finds all bunch parameters defined in `bunch_params_struct`, both normal-mode and projected

calc_bunch_params (bunch, bunch_params, plane, slice_center, slice_spread, err, print_err)

Finds all bunch parameters for a slice through the beam distribution.

init_beam_distribution (ele, param, beam_init, beam)

Routine 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)

Routine to initialize either a random or tail-weighted distribution of particles.

reallocate_beam (beam, n_bunch, n_particle)

Routine to reallocate memory within a `beam_struct`.

reallocate_bunch (bunch, n_particle)

Subroutine to reallocate particles within a `bunch_struct`.

track1_bunch_custom (bunch_start, lat, ele, bunch_end, err_flag)

Dummy routine for custom bunch tracking.

track_beam (lat, beam, ele1, ele2, err)

Routine to track a beam of particles from the end of `lat%ele(ix1)` Through to the end of `lat%ele(ix2)`.

35.3 Branch Handling Routines

allocate_branch_array (lat, upper_bound)

Routine to allocate or re-allocate an branch array. The old information is saved.

transfer_branch (branch1, branch2)

Routine to set branch2 = branch1. This is a plain transfer of information not using the overloaded equal.

transfer_branches (branch1, branch2)

Routine to set branch2 = branch1. This is a plain transfer of information not using the overloaded equal.

35.4 Coherent Synchrotron Radiation (CSR)

csr_bin_particles (particle, bin)

Routine to bin the particles longitudinally in s.

csr_bin_kicks (lat, ele, s_travel, bin, small_angle_approx)

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 (kick1, i_bin, k_factor, bin) result (i_this)

Routine to calculate the CSR kick integral.

z_calc_csr (d, k_factor, bin, small_angle_approx, dz_dd) result (z_this)

Routine to calculate the distance between the source particle and the kicked particle.

d_calc_csr (dz_particles, k_factor, bin, small_angle_approx) result (d_this)

Routine to calculate the distance between source and kick points.

35.5 Collective Effects

setup_ultra_rel_space_charge_calc (calc_on, lattice, n_part, mode, closed_orb)

Routine to initialize constants needed by the transverse space charge tracking routine track1_space_charge.

touschek_lifetime (mode, Tl, lat)

Routine to calculate the Touschek lifetime for a lat.

35.6 Electro-Magnetic Fields

em_field_calc (ele, param, s_rel, time, orbit, local_ref_frame, field, calc_dfield, err_flag)

Routine to calculate the E and B fields for an element.

em_field_custom(ele, param, s_rel, time, orb, local_ref_frame, field, calc_dfield, err_flag)

Custom routine for calculating fields.

35.7 Helper Routines: File, System, and IO

append_subdirectory (dir, sub_dir, dir_out, err)

Routine 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)

Routine for getting a single character from the terminal. Also see: get_a_char

get_a_char (this_char, wait, ignore_this)

Routine 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, to_routine, min_level, max_level)

Routine 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 (...)

Routine 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)

Routine 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 (ix_unit, error_flag)

Routine 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)

Routine to type out a file to the screen.

35.8 Helper Routines: Math (Except Matrix)

abs_sort (array, index, n)

Routine 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.

cross_product (a, b)

Returns the cross product of a x b

linear_fit (x, y, n_data, a, b, sig_a, sig_b)

Routine to fit to $y = A + B x$

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, ran_state)

Routine to set what random number generator algorithm is used. If this routine is never called then pseudo_random\$ is used.

ran_gauss (harvest)

Routine to return a Gaussian distributed random number with unit sigma.

ran_gauss_converter (set, set_sigma_cut, get, get_sigma_cut, ran_state)

Routine to set what conversion routine is used for converting uniformly distributed random numbers to Gaussian distributed random numbers.

ran_seed_put (seed, ran_state)

Routine to seed the random number generator.

ran_seed_get (seed, ran_state)

Routine to return the seed used for the random number generator.

ran_uniform (harvest)

Routine to return a random number uniformly distributed in the interval $[0, 1]$. This routine uses the same algorithm as ran from

spline_akima (spline, ok)

Given a set of (x,y) points we want to interpolate between the points. This routine computes the semi-hermite cubic spline developed by akima

spline_evaluate (spline, x, ok, y, dy)

Routine 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.

35.9 Helper Routines: Matrix

mat_eigen (mat, eval_r, eval_i, evec_r, evec_i, error, print_err)

Routine for determining the eigen vectors and eigen values of a matrix.

mat_inverse (mat, mat_inv, ok, print_err)

Routine 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)

Routine to construct a 2x2 rotation matrix for translation from point 1 to point 2.

mat_symplectify (mat_in, mat_symp, p0_ratio, r_root)

Routine to form a symplectic matrix that is approximately equal to the input matrix.

mat_symp_error (mat, p0_ratio, err_mat) result (error)

Routine to check the symplecticity of a square matrix

mat_symp_conj (mat1, mat2)

Routine to take the symplectic conjugate of a square matrix.

mat_symp_decouple (t0, tol, stat, u, v,

ubar, vbar, g, twiss1, twiss2, gamma, type_out)

Routine to find the symplectic eigen-modes of the one turn 4x4 coupled transfer matrix T0.

mat_type (mat, nunit, header, num_form)

Routine to output matrices to the terminal or to a file

35.10 Helper Routines: Miscellaneous

date_and_time_stamp (string, numeric_month)

Routine to return the current date and time in a character string.

err_exit()

Routine 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, exact)

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_out, string_default, opt_string)

Routine to return True or False depending upon the state of an optional string.

35.11 Helper Routines: String Manipulation

downcase_string (string)

Routine to convert a string to lowercase:

indexx_char (arr,index)

Routine to sort a character array. This routine 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, can_abbreviate, matched_name)

Routine 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.

string_to_int (line, default, value, err_flag)

Routine to convert a string to an integer.

string_to_real (line, default, value, err_flag)

Routine to convert a string to an real.

string_trim(in_string, out_string, word_len)

Routine 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)

Routine to trim a string of leading delimiters and also to return the length of the first word.

str_downcase (dst, src)

Routine 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:

35.12 Inter-Beam Scattering (IBS)

ibs_lifetime(lat,ibs_sim_params,maxratio,lifetime,granularity)

This module computes the beam lifetime due to the diffusion process according to equation 12

35.13 Lattice: Element Manipulation

These routine are for adding elements, moving elements, etc.

add_lattice_control_structs (lat, ele, add_at_end)

Routine to adjust the control structure of a lat so that extra control elements can be added.

add_superimpose (lat, super_ele_in, ix_branch, err_flag, super_ele_out, save_null_drift, create_jumbo_slave)

Routine to make a superimposed element.

attribute_bookkeeper (ele, param, force_bookkeeping)

Routine to make sure the attributes of an element are self-consistent.

create_group (lat, ix_lord, contrl, err, err_print_flag)

Routine to create a group control element.

create_girder (lat, ix_girder, contrl, ele_init)

Routine to add the controller information to slave elements of an girder_lord.

create_overlay (lat, ix_overlay, attrib_name, contrl, err, err_print_flag)

Routine to add the controller information to slave elements of an overlay_lord.

create_wiggler_model (wiggler_in, 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, ix_branch)

Routine to Insert a new element into the tracking part of the lat structure.

**create_element_slice (sliced_ele, ele_in, l_slice,
offset, param, include_upstream_end,
include_downstream_end, err_flag, old_slice)**
 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 (r_in, keep_ele, remove_markers, r_out, ix_out, use_taylor, orb0)
 Routine to concatenate together elements to make a hybrid lat

new_control (lat, ix_ele)
 Routine to create a new control element.

**pointer_to_attribute (ele, attrib_name, do_allocation,
ptr_attrib, err_flag, err_print_flag, ix_attrib)**
 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, eles, ix_attrib)**
 Returns an array of pointers to an attribute with name attrib_name within elements with name ele_name.

pointer_to_branch
 Routine to return a pointer to a lattice branch.

pointer_to_ele (lat, ix_ele, ix_branch) result (ele_ptr)
pointer_to_ele (lat, ele_loc_id) result (ele_ptr)
 Routine to point to a given element.

remove_eles_from_lat (lat, check_sanity)
 Routine to remove an elements from the lattice.

set_ele_status_stale (ele, status_group, set_slaves)
 Routine to set a status flags to stale in an element and the corresponding ones for any slaves the element has.

set_status_flags (bookkeeping_state, stat)
 Routine to set the bookkeeping status block.

**split_lat (lat, s_split, ix_branch, ix_split, split_done,
add_suffix, check_sanity, save_null_drift, err_flag)**
 Routine to split a lat at a point.

update_hybrid_list (lat, n_in, keep_ele, keep_overlays_and_groups)
 Routine used to specify a list of element that should not be hybridized by make_hybrid_lat.

35.14 Lattice: Geometry

ele_geometry (floor0, ele, floor, len_scale, treat_as_patch)
 Routine 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, w_mat_inv)
 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_to_local (floor0, global_position,
calculate_angles, is_delta_position) result (local_position)**
 Returns local floor position relative to floor0 given a global floor position. This is an essentially an inverse of routine local_to_floor.

floor_w_mat_to_angles (w_mat, theta0, theta, phi, psi, floor0)
 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)
 Routine to calculate the physical placement of all the elements in a lattice. That is, the physical machine layout on the floor.

local_to_floor (floor0, dr, theta, phi, psi) result (floor1)
 Starting from a given reference frame and given a shift in position, return the resulting reference frame.

patch_flips_propagation_direction (x_pitch, y_pitch) result (is_flip)
 Routine to determine if the propagation direction is flipped in a patch. This is true if the transformation matrix element S(3,3) = cos(x_pitch) * cos(y_pitch)

position_in_global_frame (local_position, ele, w_mat) result (global_position)
 Given a position local to ele, return global floor coordinates

position_in_local_frame (global_position, ele, status, w_mat) result(local_position)
 Given a position in global coordinates, return local curvilinear coordinates in ele relative to floor0

s_calc (lat)
 Routine to calculate the longitudinal distance S for the elements in a lat.

switch_local_positions (position0, ele0, ele_try, position1, ele1, ww_mat)
 Subroutine to take a local position0 in ele0 and find a local position near ele_try. If this position is beyond the bounds of ele_try, neighboring elements will be

w_mat_for_x_pitch (w_mat, x_pitch)
 Routine to return the transformation matrix for an x_pitch.

w_mat_for_y_pitch (w_mat, y_pitch)
 Routine to return the transformation matrix for an y_pitch.

w_mat_for_tilt (w_mat, tilt)
 Routine to return the transformation matrix for an tilt.

35.15 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 (ele, name, full_name)

Function to return the index of an attribute for a given element type and the name of the attribute

attribute_name (ele, ix_att)

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.

switch_attrib_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_if_s_in_bounds (branch, s, err_flag, translated_s)

Routine to check if a given longitudinal position s is within the bounds of a given branch of a lattice.

lat_sanity_check (lat, err_flag)

Routine to check the control links in a lat structure, etc.

element_at_s (lat, s, choose_max, ix_branch, err_flag, s_eff, position) result (ix_ele)

Routine to return the index of the element at position s.

ele_has_offset (ele) result (has_offset)

Function to tell if an element has a non-zero offset, pitch or tilt.

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 (ele_taylor, ele2) result (equiv)

Routine 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 (ele, ele1, ele2, ix_multipass)

Routine to find the end points of an element.

get_slave_list (lord, slaves, n_slave)

Routine 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, above_ubound_is_err)

Routine to locate all the elements in a lattice that corresponds to loc_str.

n_attrib_string_max_len () result (max_len)

Routine to return the the maximum number of characters in any attribute name known to bmad.

name_to_list (lat, ele_names, use_ele)

Routine to make a list of the elements in a lat whose name matches the names in the ele_names list.

pointer_to_indexed_attribute (ele, ix_attrib, do_allocation,
ptr_attrib, err_flag, err_print_flag)

Returns a pointer to an attribute of an element ele with attribute index ix_attrib.

pointer_to_lord (slave, ix_lord, ix_control, ix_slave) result (lord_ptr)

Function to point to a lord of a slave.

pointer_to_multipass_lord (ele, 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 (lord, ix_slave, ix_control) result (slave_ptr)

Function to point to a slave of a lord.

type_ele (ele, type_zero_attrib, type_mat6, type_taylor,
twiss_out, type_control, type_wake, type_floor_coords,
type_field, type_wall, lines, n_lines)

Like type_ele but the output is stored in a string array.

type_twiss (ele, frequency_units, compact_format, lines, n_lines)

Subroutine to print or put in a string array Twiss information from an element.

valid_tracking_method (ele, species, tracking_method, num_valid) result (is_valid)

Routine to return whether a given tracking method is valid for a given element.

valid_mat6_calc_method (ele, species, mat6_calc_method,
num_valid) result (is_valid)

Routine to return whether a given mat6_calc method is valid for a given element.

35.16 Lattice: Low Level Stuff

bracket_index (s_arr, i_min, i_max, s, ix)

Routine to find the index ix so that s(ix) \leq s < s(ix+1). If s < s(1) then ix = 0

check_controller_controls (contrl, name, err)

Routine to check for problems when setting up group or overlay controllers.

deallocate_ele_pointers (ele, nullify_only, nullify_branch, dealloc_poles)

Routine to deallocate the pointers in an element.

re_allocate_eles (eles, n, save_old, exact)

Routine to allocate an array of ele_pointer_structs.

twiss1_propagate (twiss1, mat2, length, twiss2, err)

Routine to propagate the twiss parameters of a single mode.

35.17 Lattice: Manipulation

allocate_element_array (ele, upper_bound, init_ele0)

Routine to allocate or re-allocate an element array.

allocate_lat_ele_array (lat, upper_bound, ix_branch)

Routine to allocate or re-allocate an element array.

control_bookkeeper (lat, ele, mark_eles_as_stale)

Routine 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)

Routine to deallocate the pointers in a lat.

init_ele (ele, key, sub_key, ix_ele, ix_branch, branch)

Routine to initialize an element.

init_lat (lat, n)

Routine to initialize a Bmad lat.

lattice_bookkeeper (lat, err_flag)

Routine to do bookkeeping for the entire lattice.

reallocating_coord (coord, n_coord)

Routine to reallocate an allocatable coord_struct array to at least: coord(0:n_coord).

reallocating_coord_array (coord_array, lat)

Routine to allocate an allocatable coord_array_struct array to the proper size for a lattice.

set_attribute_alias (attrib_name, alias_name, err_flag, lat)

Routine to setup an alias for element attributes like custom_attribute1\$, etc. in the attribute name table.

set_ele_defaults (ele)

Subroutine to set the defaults for an element of a given type.

set_on_off (key, lat, switch, orb, use_ref_orb, ix_branch)

Routine to turn on or off a set of elements (quadrupoles, RF cavities, etc.) in a lat.

transfer_ele (ele1, ele2, nullify_pointers)

Routine to set ele2 = ele1. This is a plain transfer of information not using the overloaded equal.

transfer_eles (ele1, ele2)

Routine to set ele2(:) = ele1(:). This is a plain transfer of information not using the overloaded equal.

transfer_ele_taylor (ele_in, ele_out, taylor_order)

Routine to transfer a Taylor map from one element to another.

transfer_lat (lat1, lat2)

Routine to set lat2 = lat1. This is a plain transfer of information not using the overloaded equal.

transfer_lat_parameters (lat_in, lat_out)

Routine to transfer the lat parameters (such as lat%name, lat%param, etc.) from one lat to another.

zero_ele_kicks (ele)

Subroutine to zero any kick attributes like hkick, bl_vkick, etc. See also: ele_has_kick, ele_has_offset, zero_ele_offsets.

zero_ele_offsets (ele)

Routine to zero the offsets, pitches and tilt of an element.

35.18 Lattice: Miscellaneous

c_multi (n, m, no_n_fact, c_full)

Routine to compute multipole factors: $c_multi(n, m) = +/- ("n choose m")/n!$

ele_compute_ref_energy_and_time (ele, param,

e_tot_start, p0c_start, ref_time_start, err_flag)

Routine to compute the reference energy and reference time at the end of an element given the reference enegy and reference time at the start of the element.

lat_compute_ref_energy_and_time (lat, err_flag)

Routine to compute the reference energy for each element in a lattice.

field_interpolate_3d (position, field_mesh, deltas, position0)

Function to interpolate a 3d field.

order_super_lord_slaves (lat, ix_lord)

Routine to make the slave elements of a super_lord in order.

release_rad_int_cache (ix_cache)

Routine to release the memory associated with caching wiggler values.

set_flags_for_changed_attribute (ele, attrib)

Routine to mark an element as modified for use with "intelligent" bookkeeping.

35.19 Lattice: Reading and Writing Files

aml_parser (lat_file, lat, make_mats6, digested_read_ok, use_line, err_flag)

Routine to parse an AML input file and put the information in a lat_struct.

bmad_and_xsif_parser (lat_file, lat, make_mats6, digested_read_ok,

use_line, err_flag)

Subroutine to parse either a Bmad or XSIF (extended standard input format) lattice files.

bmad_parser (lat_file, lat, make_mats6, digested_read_ok, use_line, err_flag)

Routine to parse (read in) a Bmad input file.

bmad_parser2 (lat_file, lat, orbit, make_mats6, err_flag)

Routine to parse (read in) a Bmad input file to modify an existing lattice.

bmad_to_mad_or_xsif (out_type, out_file_name, lat, use_matrix_model,

ix_start, ix_end, converted_lat, err)

Routine 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.

init_custom (ele, err_flag)

Routine for initializing custom elements or elements that do custom calculations.

read_digested_bmad_file (digested_file, lat, inc_version, err_flag)

Routine to read in a digested file.

write_bmad_lattice_file (bmad_file, lat, err)

Routine to write a Bmad lattice file using the information in a lat_struct.

write_digested_bmad_file (digested_name, lat, n_files, file_names, extra, err_flag)

Routine to write a digested file.

xsif_parser (xsif_file, lat, make_mats6, digested_read_ok, use_line, err_flag)

Routine to parse an XSIF (extended standard input format) lattice file.

35.20 Matrices

c_to_cbar (ele, cbar_mat)

Routine to compute Cbar from the C matrix and the Twiss parameters.

cbar_to_c (cbar_mat, a, b, c_mat)

Routine 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.

concat_transfer_mat (mat_1, vec_1, mat_0, vec_0, mat_out, vec_out)

Routine to concatenate two linear maps

determinant (mat) result (det)

Routine to take the determinant of a square matrix This routine is adapted from Numerical Recipes.

do_mode_flip (ele, err_flag)

Routine to mode flip the Twiss parameters of an element

make_g2_mats (twiss, g2_mat, g2_inv_mat)

Routine 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)

Routine to make the matrices needed to go from normal mode coords to coordinates with the beta function removed.

make_mat6 (ele, param, start_orb, end_orb, end_in, err_flag)

Routine to make the 6x6 transfer matrix for an element.

make_v_mats (ele, v_mat, v_inv_mat)

Routine to make the matrices needed to go from normal mode coords to X-Y coords and vice versa.

mat6_from_s_to_s (lat, mat6, vec0, s1, s2, orbit, ix_branch, one_turn, unit_start, err_flag, ele_save)

Subroutine to calculate the transfer map between longitudinal positions s1 to s2.

mat6_to_taylor (vec0, mat6, bmad_taylor)

Routine to form a first order Taylor map from the 6x6 transfer matrix and the 0th order transfer vector.

match_ele_to_mat6 (ele, vec0, mat6, err_flag)

Routine to make the 6 x 6 transfer matrix from the twiss parameters.

multi_turn_tracking_to_mat (track, i_dim, map1, map0, track0, chi)

Routine to analyze 1-turn tracking data to find the 1-turn transfer matrix and the closed orbit offset.

transfer_matrix_calc (lat, rf_on, xfer_mat, xfer_vec, ix1, ix2, ix_branch, one_turn)

Routine 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)

Routine to form the 4x4 1-turn coupled matrix with the reference point at the end of an element.

lat_make_mat6 (lat, ix_ele, ref_orb, ix_branch, err_flag)

Routine to make the 6x6 linear transfer matrix for an element

taylor_to_mat6 (a_taylor, r_in, vec0, mat6, r_out)

Routine to calculate the linear (Jacobian) matrix about some trajectory from a Taylor map.

transfer_mat2_from_twiss (twiss1, twiss2, mat)

Routine to make a 2 x 2 transfer matrix from the Twiss parameters at the end points.

transfer_mat_from_twiss (ele1, ele2, m)

Routine 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)

Routine to extract the Twiss parameters from the one-turn 2x2 matrix

twiss_from_mat6 (mat6, map0, ele, stable, growth_rate, status, type_out)

Routine to extract the Twiss parameters from the one-turn 6x6 matrix

twiss_to_1_turn_mat (twiss, phi, mat2)

Routine to form the 2x2 1-turn transfer matrix from the Twiss parameters.

35.21 Matrix: Low Level Routines

Listed below are helper routines that are not meant for general use.

drift_mat6_calc (mat6, length, ele, param, start, end)

Routine to calculate a drift transfer matrix with a possible kick.

sol_quad_mat6_calc (ks_in, k1_in, s_len, orb, m, dz_coef)

Routine to calculate the transfer matrix for a combination solenoid/quadrupole element.

tilt_mat6 (mat6, tilt)

Routine 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.

35.22 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.

to_eta_reading (eta_actual, 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, reading, err)

Find the measured coupling values given the actual ones

35.23 Multipass

multipass_all_info (lat, info)

Routine to put multipass to a multipass_all_info_struct structure.

multipass_chain (ele, 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:

35.24 Multipoles

ab_multipole_kick (a, b, n, coord, kx, ky, dk)

Routine to put in the kick due to an ab_multipole.

multipole_kicks (knl, tilt, coord, ref_orb_offset)

Routine 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)

Routine to convert ab type multipoles to kt (MAD standard) multipoles.

multipole_ele_to_ab (ele, param, use_ele_tilt, has_nonzero_pole, a, b)

Routine to put the scaled element multipole components (normal and skew) into 2 vectors.

multipole_ele_to_kt (ele, param, use_ele_tilt, has_nonzero_pole, knl, tilt)

Routine to put the scaled element multipole components (strength and tilt) into 2 vectors.

multipole_init(ele, zero)

Routine to initialize the multipole arrays within an element.

multipole_kick (knl, tilt, n, coord, ref_orb_offset)

Routine to put in the kick due to a multipole.

multipole_kt_to_ab (knl, tn, an, bn)

Routine to convert kt (MAD standard) multipoles to ab type multipoles.

35.25 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()

Routine that clears out previous saved values of the optimizer.

suggest_lmdif (XV, FV, EPS, ITERMX, at_end, reset_flag)

Reverse communication routine.

super_mrqmin (y, weight, a,

covar, alpha, chisq, funcs, alamda, status, maska)

Routine to do non-linear optimizations. This routine is essentially mrqmin from Numerical Recipes with some added features.

opti_de (v_best, generations, population, merit_func, v_del, status)

Differential Evolution for Optimal Control Problems. This optimizer is based upon the work of Storn and Price.

35.26 Overloading the equal sign

These routines are overloaded by the equal sign so should not be called explicitly.

branch_equal_branch (branch1, branch2)

Routine that is used to set one branch equal to another.

bunch_equal_bunch (bunch1, bunch2)

Routine that is used to set one macroparticle bunch to another. This routine takes care of the pointers in bunch1.

coord_equal_coord (coord1, coord2)

Routine that is used to set one coord_struct equal to another.

ele_equal_ele (ele1, ele2)

Routine 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)

Routine that is used to set one element vector equal to another. This routine takes care of the pointers in ele1.

real_8_equal_taylor (y8, bmad_taylor)

Routine to overload " $=$ " in expressions real_8 (PTC) = bmad_taylor.

lat_equal_lat (lat_out, lat_in)

Routine 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)

Routine 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)

Routine to overload " $=$ " in expressions bmad_taylor = real_8 (PTC)

universal_equal_universal (ut1, ut2)

Routine that is used to set one PTC universal_taylor structure equal to another.

35.27 Particle Coordinate Stuff

convert_coords (in_type_str, coord_in, ele, out_type_str, coord_out, err_flag)

Routine to convert between lab frame, normal mode, normalized normal mode, and action-angle coordinates.

convert_pc_to (pc, particle, E_tot, gamma, kinetic, beta, brho, dbeta, err_flag)

Routine to calculate the energy, etc. from a particle's momentum.

convert_total_energy_to (E_tot, particle, gamma, kinetic, beta, pc, brho, dbeta, err_flag)

Routine to calculate the momentum, etc. from a particle's total energy.

init_coord (orb, vec, ele, particle)

Routine to initialize a coord_struct.

type_coord (coord)

Routine to type out a coordinate.

35.28 Photon Routines

bend_photon_init (g_bend_x, g_bend_y, gamma, orbit, E_min, E_max, E_integ_prob)

Routine to initialize a photon

bend_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.

bend_photon_energy_init (e_rel, r_in)

Routine to convert a random number in the interval [0,1] to a photon energy.

35.29 Interface to PTC

concat_real_8 (y1, y2, y3)

Routine to concatenate two real_8 taylor series.

ele_to_fibre (ele, ptc_fibre, param, use_offsets, integ_order, steps, for_layout)

Routine to convert a Bmad element to a PTC fibre element.

map_coef (y, i, j, k, l)

Function to return the coefficient of the map $y(:)$ up to 3rd order.

kill_ptc_genfield (ptc_genfield)

Subroutine to kill a ptc_genfield.

kind_name (this_kind)

Function to return the name of a PTC kind.

real_8_equal_taylor (y8, bmad_taylor)

Routine to overload " $=$ " in expressions $\text{real_8} = \text{bmad_taylor}$

real_8_to_taylor (y8, v0, beta0, beta1, bmad_taylor)

Routine to convert from a real_8 taylor map in Etienne's PTC to a taylor map in Bmad.

real_8_init (y, set_taylor)

Routine to allocate a PTC real_8 variable.

remove_constant_taylor (taylor_in, taylor_out, c0, remove_higher_order_terms)

Routine to remove the constant part of a taylor series.

lat_to_ptc_layout (lat)

Routine to create a PTC layout from a Bmad lat.

**set_ptc (e_tot, particle, taylor_order, integ_order, n_step, no_cavity,
exact_modeling, exact_misalign)**

Routine to initialize PTC.

set_taylor_order (order, override_flag)

Routine to set the taylor order.

sort_universal_terms (ut_in, ut_sorted)

Routine to sort the taylor terms from "lowest" to "highest".

taylor_equal_real_8 (bmad_taylor, y8)

Routine to overload " $=$ " in expressions $\text{bmad_taylor} = \text{y8}$

taylor_to_real_8 (bmad_taylor, v0, beta0, beta1, y8)

Routine to convert from a taylor map in Bmad to a real_8 taylor map in Etienne's PTC.

type_layout (lay)

Routine to print the global information in a PTC layout.

type_map1 (y, type0, n_dim)

Routine to type the transfer map up to first order.

type_fibre (ptc_fibre, print_coords, lines, n_lines)

Routine to print the global information in a fibre.

type_map (y)

Routine to type the transfer maps of a real_8 array.

type_real_8_taylors (y)

Routine to type out the taylor series from a real_8 array.

taylor_to_genfield (bmad_taylor, ptc_genfield, c0)

Routine to construct a genfield (partially inverted map) from a taylor map.

universal_to_bmad_taylor (u_taylor, bmad_taylor)

Routine to convert from a universal_taylor map in Etienne's PTC to a taylor map in Bmad.

vec_bmad_to_ptc (vec_bmad, beta0, vec_ptc, conversion_mat)

Routine to convert from Bmad to PTC coordinates.

vec_ptc_to_bmad (vec_ptc, beta0, vec_bmad, conversion_mat)

Routine to convert from PTC to Bmad coordinates.

35.30 Quick Plot Routines

35.30.1 Quick Plot Page Routines

qp_open_page (page_type, i_chan, x_len, y_len, units, plot_file, scale)

Routine to Initialize a page (window) for plotting.

qp_select_page (iw)

Routine to switch to a particular page for drawing graphics.

qp_close_page()

Routine to finish plotting on a page.

35.30.2 Quick Plot Calculational 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_str, data_min, data_max, div_min, div_max, bounds, axis_type, slop_factor)

Routine 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, axis, slop_factor)

Routine 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)

Routine to calculate the number of decimal places needed to display the axis numbers.

qp_calc_axis_scale (data_min, data_max, axis, niceness_score, slop_factor)
 Routine to calculate a "nice" plot scale given the minimum and maximum of the data.

qp_calc_minor_div (delta, div_max, divisions)
 Routine to calculate the number of minor divisions an axis should have.

qp_convert_rectangle_rel (rect1, rect2)
 Routine to convert a "rectangle" (structure of 4 points) from one set of relative units to another

35.30.3 Quick Plot Drawing Routines

qp_clear_box()
 Routine to clear the current box on the page.

qp_clear_page()
 Routine to clear all drawing from the page.

**qp_draw_circle (x0, y0, r, angle0, del_angle,
 units, width, color, line_pattern, clip)**
 Routine to plot a section of an ellipse.

**qp_draw_ellipse (x0, y0, r_x, r_y, theta_xy,
 angle0, del_angle, units, width, color, line_pattern, clip)**
 Routine to plot a section of an ellipse.

qp_draw_axes(x_lab, y_lab, title, draw_grid)
 Routine to plot the axes, title, etc. of a plot.

qp_draw_data (x_dat, y_dat, draw_line, symbol_every, clip)
 Routine to plot data, axes with labels, a grid, and a title.

**qp_draw_graph (x_dat, y_dat, x_lab, y_lab, title,
 draw_line, symbol_every, clip)**
 Routine to plot data, axes with labels, a grid, and a title.

qp_draw_graph_title (title)
 Routine to draw the title for a graph.

qp_draw_grid()
 Routine to draw a grid on the current graph.

qp_draw_histogram (x_dat, y_dat, fill_color, fill_pattern, line_color, clip)
 Routine to plot data, axes with labels, a grid, and a title.

**qp_draw_curve_legend (x_origin, y_origin, units, line, line_length,
 symbol, text, text_offset, draw_line, draw_symbol, draw_text)**
 Routine to draw a legend with each line in the legend having a line, a symbol, some text.

qp_draw_text_legend (text, x_origin, y_origin, units)
 Routine to draw a legend of lines of text.

qp_draw_main_title (lines, justify)
 Routine to plot the main title at the top of the page.

qp_draw_polyline (x, y, units, width, color, line_pattern, clip, style)
 Routine to draw a polyline.

qp_draw_polyline_no_set (x, y, units)

Routine 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)

Routine to draw a polyline. See also qp_draw_polyline

qp_draw_line (x1, x2, y1, y2, units, width, color, line_pattern, clip, style)

Routine to draw a line.

qp_draw_rectangle (x1, x2, y1, y2, units, color, width, line_pattern, clip, style)

Routine to draw a rectangular box.

qp_draw_symbol (x, y, units, type, height, color, fill_pattern, line_width, clip)

Draws a symbol at (x, y)

**qp_draw_symbols (x, y, units, type, height, color,
fill_pattern, line_width, clip, symbol_every)**

Draws a symbol at the (x, y) points.

**qp_draw_text (text, x, y, units, justify, height, color, angle,
background, uniform_spacing, spacing_factor)**

Routine to draw text.

qp_draw_text_no_set (text, x, y, units, justify, angle)

Routine to display on a plot a character string. See also: qp_draw_text.

qp_draw_text_basic (text, len_text, x0, y0, angle, justify)

Routine to display on a plot a character string. See also: qp_draw_text.

qp_draw_x_axis (who, y_pos)

Routine to draw a horizontal axis.

qp_draw_y_axis (who, x_pos)

Routine to draw a horizontal axis.

qp_paint_rectangle (x1, x2, y1, y2, units, color, fill_pattern)

Routine 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)

Routine to form the text string for an axis number.

35.30.4 Quick Plot Set Routines

**qp_calc_and_set_axis (axis, data_min, data_max,
div_min, div_max, bounds, axis_type, slop_factor)**

Routine to calculate a "nice" plot scale given the minimum and maximum of the data.

qp_eliminate_xy_distortion(axis_to_scale)

This routine 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_str, 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)**

Routine 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)

Routine to set the box on the physical page. This routine divides the page into a grid of boxes.

qp_set_graph (title)

Routine to set certain graph attributes.

qp_set_graph_limits()

Routine to calculate the offsets for the graph. This routine 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)

Routine 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,

x2_mirrors_x, y2_mirrors_y, box, margin, page_border)

Routine to set various attributes. This routine can be used in place of other qp_set_* routines.

qp_set_line (who, line)

Routine to set the default line attributes.

qp_set_margin (x1_marg, x2_marg, y1_marg, y2_marg, units)

Routine 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)

Routine to set the border around the physical page.

qp_set_page_border_to_box ()

Routine 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)

Routine to set the default clipping state.

qp_set_parameters (text_scale, default_draw_units, default_set_units,

default_axis_slop_factor)

Routine to set various quick plot parameters.

qp_subset_box (ix, iy, ix_tot, iy_tot, x_marg, y_marg)

Routine 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)

Routine 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_pattern, line_width, clip)

Routine to set the type and size of the symbols used in plotting data.

qp_set_line_attrib (style, width, color, pattern, clip)

Routine to set the default line attributes.

qp_set_graph_attrib (draw_grid, draw_title)

Routine to set attributes of the current graph.

**qp_set_text_attrib (who, height, color,
background, uniform_spacing, spacing_factor)**

Routine to set the default text attributes.

qp_use_axis (x, y)
Routine to set what axis to use: X or X2, Y or Y2.

35.30.5 Informational Routines

**qp_get_axis_attrib (axis_str, a_min, a_max, div, places, label,
draw_label, draw_numbers, minor_div, mirror, number_offset,
label_offset, major_tick_len, minor_tick_len, ax_type)**

Routine to get the min, max, divisions etc. for the X and Y axes.

qp_get_layout_attrib (who, x1, x2, y1, y2, units)
Routine to get the attributes of the layout.

qp_get_line (style, line)
Routine to get the default line attributes.

**qp_get_parameters (text_scale, default_draw_units, default_set_units,
default_axis_slop_factor)**

Routine to get various quick_plot parameters.

qp_get_symbol (symbol)
Routine 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.

35.30.6 Conversion Routines

qp_from_inch_rel (x_inch, y_inch, x, y, units)

Routine to convert from a relative position (an offset) in inches to other units.

qp_from_inch_abs (x_inch, y_inch, x, y, units)

Routine 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)

Routine to convert a relative (x, y) into inches.

qp_to_inch_abs (x, y, x_inch, y_inch, units)

Routine 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)

Routine to convert a relative (x, y) into inches.

qp_to_inches_abs (x, y, x_inch, y_inch, units)

Routine to convert an absolute position (x, y) into inches referenced to the left bottom corner of the page.

35.30.7 Miscellaneous Routines

qp_read_data (iu, err_flag, x, ix_col, y, iy_col, z, iz_col, t, it_col)

Routine to read columns of data.

35.30.8 Low Level Routines

qp_clear_box_basic (x1, x2, y1, y2)

Routine to clear all drawing from a box. That is, white out the box region.

qp_clear_page_basic()

Routine to clear all drawing from the page.

qp_close_page_basic()

Routine 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)

Routine to convert a (x, y) point from one set of relative units to another.

qp_convert_point_abs (x_in, y_in, units_in, x_out, y_out, units_out)

Routine to convert a (x, y) point from one set of absolute units to another.

qp_draw_symbol_basic (x, y, symbol)

Routine to draw a symbol.

qp_init_com_struct ()

Routine to initialize the common block qp_state_struct. This routine is not for general use.

qp_join_units_string (u_type, region, corner, units)

Routine 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, page_scale)

Routine to Initialize a page (window) for plotting.

qp_paint_rectangle_basic (x1, x2, y1, y2, color, fill_pattern)

Routine to fill a rectangle with a given color. A color of white essentially erases the rectangle.

qp_pointer_to_axis (axis_str, axis_ptr)

Routine to return a pointer to an common block axis.

qp_restore_state()

Routine to restore saved attributes. Use qp_save_state to restore the saved state.

qp_restore_state_basic (buffer_basic)

Routine to restore the print state.

qp_save_state (buffer_basic)

Routine to save the current attributes. Use qp_restore_state to restore the saved state.

qp_save_state_basic ()

Routine to save the print state.

qp_select_page_basic (iw)

Routine to switch to a particular page for drawing graphics.

qp_set_char_size_basic (height)

Routine to set the character size.

qp_set_clip_basic (clip)

Routine to set the clipping state. Note: This affects both lines and symbols.

qp_set_color_basic (ix_color, set_background)

Routine to set the color taking into account that GIF inverts the black for white.

qp_set_graph_position_basic (x1, x2, y1, y2)

Routine to set the position of a graph. Units are inches from lower left of page.

qp_set_line_width_basic (line_width)

Routine to set the line width.

qp_set_symbol_fill_basic (fill)

Routine to set the symbol fill style.

qp_set_symbol_size_basic (height, symbol_type, uniform_size)

Routine to set the symbol_size

qp_set_text_background_color_basic (color)

Routine to set the character text background color.

qp_split_units_string (u_type, region, corner, units)

Routine to split a units string into its components.

qp_text_len_basic (text)

Function to find the length of a text string.

qp_translate_to_color_index (name, index)

Routine to translate from a string to a color index.

35.31 Spin Tracking

spinor_to_polar (coord, polar)

Routine to convert a spinor into polar coordinates.

polar_to_vec (polar, vec)

Routine to convert a spin vector from polar coordinates to Cartesian coordinates.

polar_to_spinor (polar, coord)

Routine to convert a spin vector in polar coordinates to a spinor.

vec_to_polar (vec, polar, phase)

Routine to convert a spin vector from Cartesian coordinates to polar coordinates preserving the complex phase.

spinor_to_vec (coord, vec)

Routine to convert a spinor to a spin vector in Cartesian coordinates.

vec_to_spinor (vec, coord, phase)

Routine 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, spin)

Subroutine to track the spin with the Euler four-vector quaternion a.

track1_spin (start_orb, ele, param, end_orb)

Routine to track the particle spin through one element.

35.32 Transfer Maps: Routines Called by make_mat6

`Make_mat6` is the routine for calculating the transfer matrix (Jacobian) 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, end_in, err)

Routine to make the 6x6 transfer matrix for an element using closed formulas.

make_mat6_custom (ele, param, c0, c1, err_flag)

Routine for custom calculations of the 6x6 transfer matrices.

make_mat6_custom2 (ele, param, c0, c1, err_flag)

Routine for custom calculations of the 6x6 transfer matrices.

make_mat6_symp_lie_ptc (ele, param, c0)

Routine to make the 6x6 transfer matrix for an element using the PTC symplectic integrator.

make_mat6_taylor (ele, param, orb_in)

Routine to make the 6x6 transfer matrix for an element from a Taylor map.

make_mat6_tracking (ele, param, c0, c1)

Routine to make the 6x6 transfer matrix for an element by tracking 7 particle with different starting conditions.

35.33 Transfer Maps: Complex Taylor Maps

add_complex_taylor_term (bmad_complex_taylor, coef, exp)

Subroutine `add_complex_taylor_term` (`bmad_complex_taylor`, `coef`, `i1, i2, i3, i4, i5, i6, i7, i8, i9`) Routine to add a `complex_taylor` term to a `complex_taylor` series.

complex_taylor_coef (bmad_complex_taylor, exp)

Function `complex_taylor_coef` (`bmad_complex_taylor`, `i1, i2, i3, i4, i5, i6, i7, i8, i9`) Function to return the coefficient for a particular `complex_taylor` term from a `complex_taylor` Series.

complex_taylor_equal_complex_taylor (complex_taylor1, complex_taylor2)

Subroutine that is used to set one complex_taylor equal to another. This routine takes care of the pointers in complex_taylor1.

complex_taylor_make_unit (bmad_complex_taylor)

Subroutine to make the unit complex_taylor map: $r(\text{out}) = \text{Map} * r(\text{in}) = r(\text{in})$

complex_taylor_exponent_index(expn) result(index)

Function to associate a unique number with a complex_taylor exponent.

complex_taylor_to_mat6 (a_complex_taylor, r_in, vec0, mat6, r_out)

Subroutine to calculate, from a complex_taylor map and about some trajectory: The 1st order (Jacobian) transfer matrix.

complex_taylors_equal_complex_taylors (complex_taylor1, complex_taylor2)

Subroutine to transfer the values from one complex_taylor map to another: complex_taylor1 <= complex_taylor2

init_complex_taylor_series (bmad_complex_taylor, n_term, save)

Subroutine to initialize a Bmad complex_taylor series (6 of these series make a complex_taylor map). Note: This routine does not zero the structure. The calling

kill_complex_taylor (bmad_complex_taylor)

Subroutine to deallocate a Bmad complex_taylor map.

mat6_to_complex_taylor (vec0, mat6, bmad_complex_taylor)

Subroutine to form a first order complex_taylor map from the 6x6 transfer matrix and the 0th order transfer vector.

sort_complex_taylor_terms (complex_taylor_in, complex_taylor_sorted)

Subroutine to sort the complex_taylor terms from "lowest" to "highest" of a complex_taylor series.

track_complex_taylor (start_orb, bmad_complex_taylor, end_orb)

Subroutine to track using a complex_taylor map.

truncate_complex_taylor_to_order (complex_taylor_in, order, complex_taylor_out)

Subroutine to throw out all terms in a complex_taylor map that are above a certain order.

type_complex_taylors (bmad_complex_taylor, max_order, lines, n_lines)

Subroutine to output a Bmad complex_taylor map.

35.34 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)

Routine to concatenate two taylor series: $taylor3(x) = taylor2(taylor1(x))$

ele_to_taylor (ele, param, orb0, taylor_map_includes_offsets)

Routine 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)

Routine to see if two elements are equivalent in terms of attributes so that their Taylor Maps would be the same.

init_taylor_series (bmad_taylor, n_term, save)

Routine to initialize a Bmad Taylor series.

kill_taylor (bmad_taylor)

Routine to deallocate a Bmad Taylor map.

mat6_to_taylor (mat6, vec0, bmad_taylor)

Routine to form a first order Taylor map from the 6x6 transfer matrix and the 0th order transfer vector.

set_taylor_order (order, override_flag)

Routine to set the taylor order.

sort_taylor_terms (taylor_in, taylor_sorted)

Routine 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)

Routine to transfer the values from one taylor map to another: Taylor1 \leq 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)

Routine to transfer the values from one taylor map to another.

taylor_make_unit (bmad_taylor)

Routine to make the unit Taylor map

taylor_to_mat6 (a_taylor, c0, mat6, c1)

Routine to calculate the linear (Jacobian) matrix about some trajectory from a Taylor map.

taylor_inverse (taylor_in, taylor_inv, err, ref_pt)

Routine to invert a taylor map.

taylor_propagate1 (bmad_taylor, ele, param)

Routine 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_orb, bmad_taylor, end_orb)

Routine to track using a Taylor map.

transfer_ele_taylor (ele_in, ele_out, taylor_order)

Routine to transfer a Taylor map from one element to another.

Routine 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)

Routine to throw out all terms in a taylor map that are above a certain order.

type_taylors (bmad_taylor, max_order, lines, n_lines)

Routine to output a Bmad taylor map.

35.35 Tracking and Closed Orbit

The following routines perform tracking and closed orbit calculations.

check_aperture_limit (orb, ele, particle_at, param, check_momentum)

Routine to check if an orbit is outside an element's aperture.

check_aperture_limit_custom (orb, ele, particle_at, param, err_flag)

Routine to check if an orbit is outside an element's aperture. Used when `ele%aperture_type` is set to `custom$`

closed_orbit_calc (lat, closed_orb, i_dim, direction, ix_branch, err_flag)

Routine 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, err_flag)

Routine to find the closed orbit via tracking.

compute_even_steps (ds_in, length, ds_default, ds_out, n_step)

Routine to compute a step size `ds_out`, close to `ds_in`, so that an integer number of steps spans the length.

dynamic_aperture (lat, orb0, theta_xy, aperture_param, aperture)

Routine to determine the dynamic aperture of a lattice via tracking.

multi_turn_tracking_analysis (track, i_dim, track0, ele, stable, growth_rate, chi, err_flag)

Routine to analyze multi-turn tracking data to get the Twiss parameters etc.

multi_turn_tracking_to_mat (track, i_dim, mat1, track0, chi)

Routine to analyze 1-turn tracking data to find the 1-turn transfer matrix and the closed orbit offset.

offset_particle (ele, param, set, coord, set_tilt,

set_multipoles, set_hvkicks, set_z_offset, ds_pos)

Routine to effectively offset an element by instead offsetting the particle position to correspond to the local element coordinates.

offset_photon (ele, orbit, set, offset_position_only, rot_mat)

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.

particle_is_moving_backward (orbit) result (is_moving_backward)

Routine to determine if a particle is moving in the backward -s direction. If not moving backward it is dead or is moving backward.

particle_is_moving_forward (orbit) result (is_moving_forward)

Routine to determine if a particle is moving in the forward +s direction. If not moving forward it is dead or is moving backward.

tilt_coords (tilt_val, coord)

Routine to effectively tilt (rotate in the x-y plane) an element by instead rotating the particle position with negative the angle.

track1 (start_orb, ele, param, end_orb, track, err_flag, ignore_radiation)

Routine 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, ele, bunch_end, err, s_start, s_end)

Routine to track a bunch of particles through the element lat%ele(ix_ele) with csr radiation effects.

track1_spin_custom (start, ele, param, end, err_flag, track)

Dummy routine for custom spin tracking. This routine needs to be replaced for a custom calculation.

track_all (lat, orbit, ix_branch, track_state, err_flag)

Routine to track through the lat.

track_from_s_to_s (lat, s_start, s_end, orbit_start, orbit_end, all_orb, ix_branch, track_state)

Routine to track a particle between two s-positions.

track_many (lat, orbit, ix_start, ix_end, direction, ix_branch, track_state)

Routine to track from one element in the lat to another.

twiss_and_track (lat, orb, ok)

twiss_and_track (lat, orb_array, ok)

Routine to calculate the twiss parameters, transport matrices and orbit.

twiss_and_track_at_s (lat, s, ele_at_s, orb, orb_at_s, ix_branch, err, use_last, compute_floor_coords)

Routine to calculate the Twiss parameters and orbit at a particular longitudinal position.

twiss_and_track_intra_ele (ele, param, l_start, l_end, track_upstream_end, track_downstream_end, orbit_start, orbit_end, ele_start, ele_end, err, compute_floor_coords)

Routine to track a particle within an element.

twiss_from_tracking (lat, ref_orb0, symp_err, err_flag, d_orb)

Routine to compute from tracking the Twiss parameters and the transfer matrices for every element in the lat.

wall_hit_handler_custom (orb, ele, s, t)

This routine is called by the Runge-Kutta integrator odeint_bmad when a particle hits a wall.

35.36 Tracking: Low Level Routines

odeint_bmad (orb_start, ele, param, orb_end, s1, s2, local_ref_frame, err_flag, track)

Routine to do Runge Kutta tracking.

create_uniform_element_slice (ele, param, i_slice, n_slice_tot, sliced_ele, s_start, s_end)

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, t, ds, end)

Routine to track 1 step using boris tracking.

track_a_drift (orb, ele, length)

Routine to track through a drift.

track_a_bend (start_orb, ele, param, end_orb)

Particle tracking through a bend element.

35.37 Tracking: Mad Routines

make_mat6_mad (ele, param, c0, c1)

Routine 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, param, energy, map)

Routine to make a 2nd order transport map a la MAD.

mad_add_offsets_and_multipoles (ele, map)

Routine 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)

Routine to make a transport map for a drift space. The equivalent MAD-8 routine is: TMDRF

mad_elsep (ele, energy, map)

Routine to make a transport map for an electric separator. The equivalent MAD-8 routine is: TMSEP

mad_sextupole (ele, energy, map)

Routine to make a transport map for a sextupole. The equivalent MAD-8 routine is: TMSEXT

mad_sbend (ele, energy, map)

Routine to make a transport map for a sector bend element. The equivalent MAD-8 routine is: TMBEND

mad_sbend_fringe (ele, energy, into, map)

Routine 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)

Routine 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)

Routine to compute the linear focusing functions. The equivalent MAD-8 routine is: TMFOC

mad_quadrupole (ele, energy, map)

Routine to make a transport map for an quadrupole element. The equivalent MAD-8 routine is: TMSEXT

mad_rfcavity (ele, energy, map)

Routine to make a transport map for an rfcavity element. The equivalent MAD-8 routine is: TMRF

mad_solenoid (ele, energy, map)

Routine to make a transport map for an solenoid. The equivalent MAD-8 routine is: TMSEXT

mad_tmsymm (te)

routine to symmetrize the 2nd order map t. The equivalent MAD-8 routine is: tmsymm

mad_tmtilt (map, tilt)

Routine to apply a tilt to a transport map. The equivalent MAD-8 routine is: TMTILT

mad_concat_map2 (map1, map2, map3)

Routine to concatenate two 2nd order transport maps.

mad_track1 (c0, map, c1)

Routine to track through a 2nd order transfer map. The equivalent MAD-8 routine is: TMTRAK

track1_mad (start_orb, ele, param, end_orb)

Routine 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, energy, taylor)

Routine to convert a mad order 2 map to a taylor map.

taylor_to_mad_map (taylor, energy, map)

Routine 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)

Routine to initialize a 2nd order transport map to unity.

35.38 Tracking: Routines called by TRACK1

Note: Generally you don't call these routines directly.

symp_lie_bmad (ele, param, start_orb, end_orb, calc_mat6, track, offset_ele)

Symplectic integration through an element to 0th or 1st order.

track1_boris (orb_start, ele, param, orb_end, err_flag, track, s_start, s_end)

Routine to do Boris tracking.

track1_bmad (start_orb, ele, param, end_orb, err_flag)

Particle tracking through a single element BMAD_standard style.

track1_custom (start_orb, ele, param, end_orb, track, err_flag)

Dummy routine for custom tracking.

track1_custom2 (start_orb, ele, param, end_orb, track, err_flag)
 Dummy routine for custom tracking.

track1_linear (start_orb, ele, param, end_orb)
 Particle tracking through a single element using the transfer matrix..

track1_radiation (start, ele, param, end, edge)
 Routine to put in radiation damping and/or fluctuations.

track1_runge_kutta (start_orb, ele, param, end_orb, err_flag, track)
 Routine to do tracking using Runge-Kutta integration.

track1_symp_lie_ptc (start_orb, ele, param, end_orb)
 Particle tracking through a single element using a Hamiltonian and a symplectic integrator.

track1_symp_map (start_orb, ele, param, end_orb)
 Particle tracking through a single element using a partially inverted taylor map (In PTC/FPP this is called a genfield).

track1_taylor (start_orb, ele, param, end_orb)
 Routine to track through an element using the elements taylor series.

35.39 Twiss and Other Calculations

calc_z_tune (lat)
 Routine to calculate the synchrotron tune from the full 6X6 1 turn matrix.

chrom_calc (lat, delta_e, chrom_x, chrom_y, err_flag,
 low_E_lat , high_E_lat , low_E_orb , high_E_orb , ix_branch)
 Routine to calculate the chromaticities by computing the tune change when the energy is changed.

chrom_tune (lat, delta_e, target_x, target_y, err_tol, err_flag)
 Routine to set the sextupole strengths so that the lat has the desired chromaticities.

quad_beta_ave (lat, ix_ele, beta_a_ave, beta_b_ave)
 Routine to compute the average betas in a quad.

radiation_integrals (lat, orbit, mode, ix_cache, ix_branch, rad_int_by_ele)
 Routine to calculate the synchrotron radiation integrals, the emittance, and energy spread.

radiation_integrals_custom (lat, ir, orb, err_flag)
 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_a_set, phi_b_set, dk1, lat, orb, ok)
 Routine to Q_tune a lat. This routine will set the tunes to within 0.001 radian (0.06 deg).

set_z_tune (lat, z_tune, ok)
 Routine to set the longitudinal tune by setting the RF voltages in the RF cavities.

transfer_map_calc (lat, t_map, ix1, ix2, ix_branch, integrate, one_turn, unit_start)
 Routine to calculate the transfer map between two elements.

**transfer_map_from_s_to_s (lat, t_map, s1, s2, ix_branch, integrate,
one_turn, unit_start, err_flag)**

Subroutine to calculate the transfer map between longitudinal positions s1 to s2.

transfer_twiss (ele_in, ele_out)

Routine to transfer the twiss parameters from one element to another.

twiss_and_track (lat, orb)

Routine to calculate the Twiss and orbit parameters. This is not necessarily the fastest routine.

twiss_at_element (lat, ix_ele, start, end, average)

Routine 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)

Routine to calculate the Twiss parameters and orbit at a particular longitudinal position.

twiss_at_start (lat, status, ix_branch)

Routine to calculate the Twiss parameters at the start of the lat.

twiss_from_tracking (lat, closed_orb_, d_orb, error)

Routine to compute from tracking, for every element in the lat, the Twiss parameters and the transfer matrices.

twiss_propagate1 (ele1, ele2, err_flag)

Routine to propagate the Twiss parameters from the end of ELE1 to the end of ELE2.

twiss_propagate_all (lat, ix_branch, err_flag)

Routine to propagate the Twiss parameters from the start to the end.

twiss_propagate_many (lat, ix_start, ix_end, direction, ix_branch, err_flag)

Routine to propagate the Twiss parameters from one element in the lat to another.

twiss_to_1_turn_mat (twiss, phi, mat2)

Routine to form the 2x2 1-turn transfer matrix from the Twiss parameters.

35.40 Twiss: 6 Dimensional

normal_mode3_calc (mat, tune, B, HV, synchrotron_motion)

Decompose a $2n \times 2n$ symplectic matrix into normal modes. For more details see:

twiss3_propagate_all (lat)

Routine to propagate the twiss parameters using all three normal modes.

twiss3_propagate1 (ele1, ele2)

Routine to propagate the twiss parameters using all three normal modes.

twiss3_at_start (lat, error)

Routine to propagate the twiss parameters using all three normal modes.

35.41 Wake Fields

init_wake (wake, n_sr_long, n_sr_trans, n_lr)

Routine to initialize a wake struct.

lr_wake_apply_kick (ele, t0_bunch, orbit)

Routine 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_long_wake_particle (ele, orbit)

Subroutine to apply the short-range wake kick to a particle and then add to the existing short-range wake the contribution from the particle.

sr_trans_wake_particle (ele, orbit)

Subroutine to apply the short-range wake kick to a particle and then add to the existing short-range wake the contribution from the particle.

track1_sr_wake (bunch, ele)

Routine to apply the short range wake fields to a bunch.

track1_lr_wake (bunch, ele)

Routine 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.

35.42 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)

Routine to calculate the emittance, energy spread, and synchrotron integrals. This routine assumes that bends are in the horizontal plane.

Part IV

Bibliography and Index

Bibliography

- [Abell06] Dan Abell, “Numerical computation of high-order transfer maps for rf cavities”, Phys. Rev. ST Accel. Beams, vol. 9 (5) pp. 052001, (2006).
- [AML] The Accelerator Markup Language / Universal Accelerator Project web page:
<http://www.lepp.cornell.edu/~dcs/aml/>
- [Bater64] B. Batterman, and H. Cole, “Dynamical Diffraction of X Rays by Perfect Crystals”, Rev. Mod. Phys., **36**, 3, pp. 681–717, (1964).
- [Berz89] M. Berz, “Differential Algebraic Description of Beam Dynamics to Very High Orders,” Particle Accelerators, Vol. 24, pp. 109-124, (1989).
- [Blas94] R. C. Blasdell and A. T. Macrander, “Modifications to the 1989 SHADOW ray-tracing code for general asymmetric perfect-crystal optics,” Nuc. Instr. & Meth. A **347**, 320 (1994).
- [Bmad] The Bmad web site:
<http://www.lepp.cornell.edu/~dcs/bmad>
- [Rio98] Manuel Sanchez del Rio, “Ray tracing simulations for crystal optics,” Proc. SPIE 3448, Crystal and Multilayer Optics, **230** (1998).
- [Brown77] K. L. Brown, F. Rothacker, D. C. Carey, and Ch. Iselin, “TRANSPORT Appendix,” Fermilab, unpublished, (December 1977).
- [Chao93] Alexander Chao, *Physics of Collective Beam Instabilities in High Energy Accelerators*, Wiley, New York (1993).
- [Corbett99] J. Corbett and Y. Nosochkov, “Effect of Insertion Devices in SPEAR-3,” Proc. 1999 Part. Acc. Conf., p. 238, (1999).
- [Duff87] J. Le Duff, *Single and Multiple Touschek Effects*. Proc. CAS Berlin 1987, CERN 89-01, 1987.
- [Forest02] E. Forest, F. Schmidt, E. McIntosh, *Introduction to the Polymorphic Tracking Code*, CERNâ€¢SLâ€¢2002â€¢044 (AP), and KEK-Report 2002-3 (2002). Can be obtained at:
<http://frs.web.cern.ch/frs/report/sl-2002-044.pdf>
- [Forest06] Etienne Forest, ‘Geometric integration for particle accelerators,’ J. Phys. A: Math. Gen. **39** (2006) 5321â€¢5377.

- [Forest88] E. Forest, J. Milutinovic, “Leading Order Hard Edge Fringe Fields Effects Exact in $(1+\delta)$ and Consistent with Maxwell’s Equations for Rectilinear Magnets,” Nuc. Instrum. and Methods in Phys. Research A **269**, pp 474-482, (1988).
- [Forest98] E. Forest, *Beam Dynamics: A New Attitude and Framework*, Harwood Academic Publishers, Amsterdam (1998).
- [Grote96] 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>
- [Healy86] L. M. Healy, *Lie Algebraic Methods for Treating Lattice Parameter Errors in Particle Accelerators*. Doctoral thesis, University of Maryland, unpublished, (1986).
- [Helm73] 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).
- [Hoff06] G. Hoffstaetter, *Hight-Energy Polarized Proton Beams, A Modern View*, Springer. Springer Tracks in Modern Physics Vol 218, (2006).
- [Iselin94] F. C. Iselin, *The MAD program Physical Methods Manual*, unpublished, (1994). Can be obtained at:
<http://mad.home.cern.ch/mad>
- [Jowett87] 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).
- [Kohn95] V. G. Kohn, “On the Thcory of Reflectivitlby an X-Ray Multilaler Mirror” physica status solidi (b), **187**, 61, (1995).
- [Press92] 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).
- [Piwin98] Anton Piwinski, *The Touschek Effect in Strong Focusing Storage Rings*. DESY 98-179, 1998.
- [Rosen94] J. Rosenzweig and L. Serafini, “Transverse Particle Motion in Radio–Frequency Linear Accelerators,” Phys Rev E, Vol. 49, p. 1599, (1994).
- [SAD] Documentation on the KEK developed SAD program for accelerator simulation and design can be found at:
<http://acc-physics.kek.jp/SAD/>
- [Sagan03] D. Sagan, J. Crittenden, and D. Rubin. “A Symplectic Model for Wigglers,” Part. Acc. Conf. (2003).
- [Sagan99] D. Sagan and D. Rubin “Linear Analysis of Coupled Lattices,” Phys. Rev. ST Accel. Beams **2**, 074001 (1999).
<http://link.aps.org/doi/10.1103/PhysRevSTAB.2.074001>
- [Sagan06] D. Sagan, “An Efficient Formalism for Simulating the Longitudinal Kick from Coherent Synchrotron Radiation,” Proc. Europ. Part. Accel. Conf. p. 2829 — 31 (2006).

- [Storn96] 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).
- [Stoltz02] 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).
- [Talman87] 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).
- [Tao] D. Sagan, J. Smith, *The Tao Manual*. Can be obtained at:
http://www.lepp.cornell.edu/~dcs/bmad/tao_entry_point.html
- [Rauben91] T. Raubenheimer, “Tolerances to Limit the Vertical Emittance in Future Storage Rings”, Particle Accelerators, 1991, **36**, pp.75-119. SLAC-PUB-4937 Rev., (1991).
- [Wiede99] H. Wiedemann, *Particle Accelerator Physics*, Springer, New York, 3rd Edition (2007).
- [Wolski06] A. Wolski, “Alternative approach to general coupled linear optics,” Phys. Rev. ST Accel. Beams 9, 024001 (2006).
- [Wyckoff65] R. W. G. Wyckoff, *Crystal Structures*, Interscience Publ. (1965).
- [Schoon11] T. Schoonjans et al. “The xraylib library for X-ray-matter interactions. Recent developments,” Spectrochimica Acta Part B: Atomic Spectroscopy **66**, pp. 776-784 (2011).
- [Tenen01] P. Tenenbaum, “LIBXSIF, A Stand alone Library for Parsing the Standard Input Format,” Proc. 2001 Part. Acc. Conf. p. 3093 — 95 (2001). Documentation at
<http://www-project.slac.stanford.edu/lc/ilc/TechNotes/LCCNotes/PDF/LCC-0060%20rev.1.pdf>

Routine Index

ab_multipole_kick, 360
abs_sort, 348
add_complex_taylor_term, 371
add_lattice_control_structs, 351
add_superimpose, 294, 351
add_taylor_term, 372
allocate_branch_array, 282, 346
allocate_element_array, 355
allocate_lat_ele_array, 282, 355
aml_parser, 357
angle_between_polars, 371
append_subdirectory, 347
attribute_bookkeeper, 273, 292, 351
attribute_free, 353
attribute_index, 272, 353
attribute_name, 272, 354
attribute_type, 354

bbi_kick, 345
bend_edge_kick, 344
bend_photon_energy_init, 362
bend_photon_init, 362
bend_photon_vert_angle_init, 362
bmad_and_xsif_parser, 295, 357
bmad_parser, 264, 265, 278, 282, 295, 298, 314, 357
bmad_parser2, 295, 357
bmad_to_mad_or_xsif, 163, 296, 357
bracket_index, 355
branch_equal_branch, 361
bunch_equal_bunch, 361

c_multi, 357
c_to_cbar, 297, 358
calc_bunch_params, 310, 345
calc_bunch_params_slice, 345
calc_z_tune, 378
cbar_to_c, 358
cesr_getarg, 347
cesr_iargc, 347
check_aperture_limit, 307, 314, 374
check_aperture_limit_custom, 307, 313, 314, 374
check_controller_controls, 355
check_if_ele_is_monitor, 360
check_if_s_in_bounds, 354
chrom_calc, 300, 378
chrom_tune, 300, 378
clear_lat_1turn_mats, 358
closed_orbit_calc, 374
closed_orbit_from_tracking, 374
combine_consecutive_elements, 357
complex_error_function, 348
complex_taylor_coef, 371
complex_taylor_equal_complex_taylor, 371
complex_taylor_exponent_index, 372
complex_taylor_make_unit, 372
complex_taylor_to_mat6, 372
complex_taylors_equal_complex_taylors, 372
compute_even_steps, 374
concat_ele_taylor, 372
concat_real_8, 363
concat_taylor, 309, 372
concat_transfer_mat, 358
control_bookkeeper, 292, 355
convert_coords, 362
convert_pc_to, 362
convert_total_energy_to, 362
coord_equal_coord, 361
create_element_slice, 280, 307, 351
create_girder, 351
create_group, 294, 351
create_overlay, 294, 351
create_sol_quad_model, 357
create_uniform_element_slice, 376
create_unique_ele_names, 357
create_wiggler_model, 351
cross_product, 348
csr_bin_kicks, 346
csr_bin_particles, 346
csr_kick_calc, 346

d_calc_csr, 346
 date_and_time_stamp, 349
 deallocate_ele_array_pointers, 356
 deallocate_ele_pointers, 271, 273, 355
 deallocate_lat_pointers, 282, 356
 determinant, 358
 dir_close, 347
 dir_open, 347
 dir_read, 347
 do_mode_flip, 358
 downcase_string, 350
 drift_mat6_calc, 359
 dynamic_aperture, 374

 ele_compute_ref_energy_and_time, 357
 ele_equal_ele, 271, 361
 ele_geometry, 274, 317, 352
 ele_has_offset, 354
 ele_loc_to_string, 354
 ele_to_fibre, 363
 ele_to_lat_loc, 354
 ele_to_taylor, 309, 372
 ele_vec_equal_ele_vec, 361
 element_at_s, 354
 element_locator, 380
 elements_locator, 380
 elements_locator_by_key, 380
 em_field_calc, 315, 346
 em_field_custom, 313, 315, 346
 emit_calc, 380
 equivalent_taylor_attributes, 354, 373
 err_exit, 349

 field_interpolate_3d, 357
 file_suffixer, 347
 find_bunch_sigma_matrix, 345
 find_element_ends, 354
 floor_angles_to_w_mat, 317, 352
 floor_to_local, 352
 floor_w_mat_to_angles, 317, 353

 get_a_char, 347
 get_file_time_stamp, 347
 get_slave_list, 354
 get_tty_char, 347

 i_csr, 346
 ibs_lifetime, 351
 index_nocase, 350
 indexx_char, 350
 init_beam_distribution, 310, 345
 init_bunch_distribution, 345

 init_complex_taylor_series, 372
 init_coord, 302, 362
 init_custom, 313, 314, 357
 init_ele, 356
 init_floor, 353
 init_lat, 282, 356
 init_spin_distribution, 345
 init_taylor_series, 373
 init_wake, 380
 initial_lmdif, 361
 insert_element, 294, 351
 integer_option, 349
 is_integer, 350
 is_logical, 350
 is_real, 350

 key_name, 354
 key_name_to_key_index, 354
 kill_complex_taylor, 372
 kill_ptc_genfield, 363
 kill_taylor, 309, 373
 kind_name, 363

 lat_compute_ref_energy_and_time, 292, 317,
 357
 lat_ele_locator, 264–266, 292, 354
 lat_equal_lat, 282, 362
 lat_geometry, 274, 292, 317, 353
 lat_make_mat6, 265, 275, 292, 299, 359
 lat_sanity_check, 354
 lat_to_ptc_layout, 321, 363
 lat_vec_equal_lat_vec, 362
 lattice_bookkeeper, 266, 291, 356
 linear_fit, 348
 local_to_floor, 353
 logic_option, 349
 lr_wake_apply_kick, 380
 lunget, 347

 mad_add_offsets_and_multipoles, 376
 mad_concat_map2, 377
 mad_drift, 376
 mad_elsep, 376
 mad_map_to_taylor, 377
 mad_quadrupole, 377
 mad_rfcaavity, 377
 mad_sbend, 376
 mad_sbend_body, 376
 mad_sbend_fringe, 376
 mad_sextupole, 376
 mad_solenoid, 377
 mad_tmfoc, 376

mad_tmsymm, 377
 mad_tmtilt, 377
 mad_track1, 377
 make_g2_mats, 358
 make_g_mats, 358
 make_hybrid_lat, 352
 make_mad_map, 376
 make_mat6, 275, 290, 314, 358
 make_mat6_bmad, 371
 make_mat6_custom, 313, 314, 371
 make_mat6_custom2, 313, 314, 371
 make_mat6_mad, 376
 make_mat6_symp_lie_ptc, 371
 make_mat6_taylor, 371
 make_mat6_tracking, 371
 make_unit_mad_map, 377
 make_v_mats, 297, 358
 map_coef, 363
 mat6_from_s_to_s, 358
 mat6_to_complex_taylor, 372
 mat6_to_taylor, 358, 373
 mat_eigen, 349
 mat_inverse, 341, 349
 mat_make_unit, 341, 349
 mat_rotation, 349
 mat_symp_conj, 349
 mat_symp_decouple, 349
 mat_symp_error, 349
 mat_symplectify, 349
 mat_type, 349
 match_ele_to_mat6, 358
 match_reg, 350
 match_wild, 350
 match_word, 350
 mexp, 360
 milli_sleep, 347
 modulo2, 348
 multi_turn_tracking_analysis, 374
 multi_turn_tracking_to_mat, 359, 374
 multipass_all_info, 360
 multipass_chain, 360
 multipole_ab_to_kt, 360
 multipole_ele_to_ab, 360
 multipole_ele_to_kt, 360
 multipole_init, 360
 multipole_kick, 361
 multipole_kicks, 360
 multipole_kt_to_ab, 361

 n_attrib_string_max_len, 354
 name_to_list, 354

 new_control, 294, 352
 normal_mode3_calc, 298, 379

 odeint_bmad, 315, 376
 offset_particle, 374
 offset_photon, 374
 on_off_logic, 350
 one_turn_mat_at_ele, 359
 opti_de, 341, 361
 opti_lmdif, 341, 361
 orbit_amplitude_calc, 374
 order_particles_in_z, 345
 order_super_lord_slaves, 357
 out_io, 347
 output_direct, 347

 particle_is_moving_backward, 374
 particle_is_moving_forward, 303, 374
 patch_flips_propagation_direction, 353
 pointer_to_attribute, 352
 pointer_to_branch, 352
 pointer_to_ele, 352
 pointer_to_indexed_attribute, 354
 pointer_to_lord, 287–289, 355
 pointer_to_multipass_lord, 355, 360
 pointer_to_slave, 287, 288, 355
 pointers_to_attribute, 352
 polar_to_spinor, 370
 polar_to_vec, 370
 position_in_global_frame, 353
 position_in_local_frame, 353

 qp_axis_niceness, 364
 qp_calc_and_set_axis, 330, 332, 364, 366
 qp_calc_axis_divisions, 364
 qp_calc_axis_params, 364
 qp_calc_axis_places, 364
 qp_calc_axis_scale, 364
 qp_calc_minor_div, 365
 qp_clear_box, 365
 qp_clear_box_basic, 369
 qp_clear_page, 365
 qp_clear_page_basic, 369
 qp_close_page, 330, 364
 qp_close_page_basic, 369
 qp_convert_point_abs, 369
 qp_convert_point_rel, 369
 qp_convert_rectangle_rel, 365
 qp_draw_axes, 330, 332, 365
 qp_draw_circle, 365
 qp_draw_curve_legend, 365
 qp_draw_data, 330, 365

qp_draw_ellipse, 365
 qp_draw_graph, 365
 qp_draw_graph_title, 365
 qp_draw_grid, 365
 qp_draw_histogram, 365
 qp_draw_line, 366
 qp_draw_main_title, 365
 qp_draw_polyline, 365
 qp_draw_polyline_basic, 366
 qp_draw_polyline_no_set, 365
 qp_draw_rectangle, 333, 366
 qp_draw_symbol, 366
 qp_draw_symbol_basic, 369
 qp_draw_symbols, 366
 qp_draw_text, 330, 331, 366
 qp_draw_text_basic, 366
 qp_draw_text_legend, 365
 qp_draw_text_no_set, 366
 qp_draw_x_axis, 366
 qp_draw_y_axis, 366
 qp_eliminate_xy_distortion, 366
 qp_from_inch_abs, 368
 qp_from_inch_rel, 368
 qp_get_axis_attrib, 368
 qp_get_layout_attrib, 368
 qp_get_line_attrib, 368
 qp_get_parameters, 368
 qp_get_symbol_attrib, 368
 qp_init_com_struct, 369
 qp_join_units_string, 369
 qp_justify, 369
 qp_open_page, 330, 332, 364
 qp_open_page_basic, 369
 qp_paint_rectangle, 366
 qp_paint_rectangle_basic, 369
 qp_pointer_to_axis, 369
 qp_read_data, 330, 331, 369
 qp_restore_state, 330, 332, 369
 qp_restore_state_basic, 369
 qp_save_state, 330, 332, 369
 qp_save_state_basic, 370
 qp_select_page, 364
 qp_select_page_basic, 370
 qp_set_axis, 330, 334, 366
 qp_set_box, 330, 332, 366
 qp_set_char_size_basic, 370
 qp_set_clip, 367
 qp_set_clip_basic, 370
 qp_set_color_basic, 370
 qp_set_graph, 367
 qp_set_graph_attrib, 330, 367
 qp_set_graph_limits, 367
 qp_set_graph_placement, 367
 qp_set_graph_position_basic, 370
 qp_set_layout, 367
 qp_set_line, 367
 qp_set_line_attrib, 330, 332, 367
 qp_set_line_width_basic, 370
 qp_set_margin, 330, 332, 334, 367
 qp_set_page_border, 330–332, 367
 qp_set_page_border_to_box, 367
 qp_set_parameters, 334, 367
 qp_set_symbol, 367
 qp_set_symbol_attrib, 330, 332, 367
 qp_set_symbol_fill_basic, 370
 qp_set_symbol_size_basic, 370
 qp_set_text_attrib, 367
 qp_set_text_background_color_basic, 370
 qp_split_units_string, 370
 qp_subset_box, 367
 qp_text_height_to_inches, 368
 qp_text_len, 368
 qp_text_len_basic, 370
 qp_to_axis_number_text, 366
 qp_to_inch_abs, 368
 qp_to_inch_rel, 368
 qp_to_inches_abs, 368
 qp_to_inches_rel, 368
 qp_translate_to_color_index, 370
 qp_use_axis, 334, 368
 quad_beta_ave, 378
 quaternion_track, 371
 radiation_integrals, 299, 378
 radiation_integrals_custom, 313, 378
 ran_engine, 348
 ran_gauss, 348
 ran_gauss_converter, 348
 ran_seed_get, 348
 ran_seed_put, 348
 ran_uniform, 348
 randomize_lr_wake_frequencies, 380
 re_allocate, 349
 re_allocate_eles, 355
 re_associate, 349
 read_a_line, 347
 read_digested_bmad_file, 296, 358
 real_8_equal_taylor, 361, 363
 real_8_init, 363
 real_8_to_taylor, 363
 real_option, 350
 reallocatem_beam, 345

reallocate_bunch, 345
reallocate_coord, 303, 356
reallocate_coord_array, 304, 356
relative_mode_flip, 378
release_rad_int_cache, 357
remove_constant_taylor, 363
remove_eles_from_lat, 294, 352

s_calc, 292, 317, 353
set_attribute_alias, 279, 356
set_ele_defaults, 356
set_ele_status_stale, 352
set_flags_for_changed_attribute, 265, 291, 298,
 357
set_on_off, 356
set_ptc, 321, 363
set_status_flags, 352
set_taylor_order, 309, 363, 373
set_tune, 299, 378
set_z_tune, 299, 378
setup_ultra_rel_space_charge_calc, 346
skip_header, 347
sol_quad_mat6_calc, 359
sort_complex_taylor_terms, 372
sort_taylor_terms, 373
sort_universal_terms, 363
spinor_to_polar, 370
spinor_to_vec, 370
spline_akima, 348
spline_evaluate, 348
split_lat, 294, 352
splitfilename, 347
sr_long_wake_particle, 380
sr_trans_wake_particle, 380
str.downcase, 351
str.match_wild, 350
str.replace, 351
string_option, 350
string_to_int, 350
string_to_real, 350
string_trim, 350
string_trim2, 351
suggest_lmdif, 361
super_ludcmp, 348
super_mrqmin, 341, 361
switch_attrib_value_name, 354
switch_local_positions, 353
symp_lie_bmad, 377
system_command, 348

taylor_coef, 309, 373

taylor_equal_real_8, 362, 363
taylor_equal_taylor, 373
taylor_inverse, 373
taylor_make_unit, 373
taylor_minus_taylor, 373
taylor_plus_taylor, 373
taylor_propagate1, 373
taylor_to_genfield, 364
taylor_to_mad_map, 377
taylor_to_mat6, 359, 373
taylor_to_real_8, 363
taylors_equal_taylors, 373
tilt_coords, 375
tilt_mat6, 359
to_eta_reading, 360
to_orbit_reading, 360
to_phase_and_coupling_reading, 360
touschek_lifetime, 346
track1, 290, 303, 314, 375
track1_beam, 310, 345
track1_beam_simple, 375
track1_bmad, 377
track1_boris, 377
track1_boris_partial, 376
track1_bunch, 345
track1_bunch_csr, 375
track1_bunch_custom, 313, 345
track1_bunch_hom, 345
track1_custom, 313, 314, 377
track1_custom2, 313, 314, 377
track1_linear, 378
track1_lr_wake, 380
track1_mad, 377
track1_radiation, 378
track1_runge_kutta, 378
track1_spin, 371
track1_spin_custom, 313, 375
track1_sr_wake, 380
track1_symp_lie_ptc, 378
track1_symp_map, 378
track1_taylor, 378
track_a_bend, 376
track_a_drift, 376
track_all, 304, 375
track_beam, 310, 345
track_complex_taylor, 372
track_from_s_to_s, 375
track_many, 304, 309, 375
track_taylor, 373
transfer_branch, 346
transfer_branches, 346

transfer_ele, 356
 transfer_ele_taylor, 356, 373
 transfer_eles, 356
 transfer_lat, 356
 transfer_lat_parameters, 356
 transfer_map_calc, 378
 transfer_map_from_s_to_s, 378
 transfer_mat2_from_twiss, 359
 transfer_mat_from_twiss, 359
 transfer_matrix_calc, 359
 transfer_twiss, 306, 379
 truncate_complex_taylor_to_order, 372
 truncate_taylor_to_order, 373
 twiss1_propagate, 355
 twiss3_at_start, 379
 twiss3_propagate1, 379
 twiss3_propagate_all, 379
 twiss_and_track, 375, 379
 twiss_and_track_at_s, 299, 307, 375, 379
 twiss_and_track_intra_ele, 299, 307, 375
 twiss_at_element, 379
 twiss_at_start, 264, 265, 298, 379
 twiss_from_mat2, 359
 twiss_from_mat6, 359
 twiss_from_tracking, 149, 375, 379
 twiss_propagate1, 298, 379
 twiss_propagate_all, 264, 265, 298, 299, 379
 twiss_propagate_many, 379
 twiss_to_1_turn_mat, 359, 379
 type_complex_taylors, 372
 type_coord, 362
 type_ele, 264, 265, 269, 279, 355
 type_map, 363
 type_map1, 363
 type_ptc_fibre, 363
 type_ptc_layout, 363
 type_real_8_taylors, 364
 type_taylors, 374
 type_this_file, 348
 type_twiss, 355

 universal_equal_universal, 362
 universal_to_bmad_taylor, 364
 upcase_string, 351
 update_hybrid_list, 352

 valid_mat6_calc_method, 308, 355
 valid_tracking_method, 308, 355
 vec_bmad_to_ptc, 364
 vec_ptc_to_bmad, 364
 vec_to_polar, 370

Index

! comment symbol, 27
\$
 character to denote a parameter, 260
& continuation symbol, 27

ab_multipole, 38, 106, 117, 120, 122, 181
abs, 33
abs_tol_adaptive_tracking, 124, 147
abs_tol_tracking, 147
absolute time tracking, 177
absolute_time_tracking, 139, 141
absolute_time_tracking_default, 147
Accelerator Markup Language (AML), 164
accordion_edge, 60
acos, 33
adaptive_runge_kutta
 and field maps, 73
alias, 60, 84
alpha_a, 142
alpha_angle, 46
alpha_b, 142
an, *see* multipole, an
angle, 40–42, 74, 83, 171
antimuon, 140
antimuon\$, 284
antiproton, 140
antiproton\$, 284
aperture, 91, 106, 273
aperture_at, 91, 92
aperture_limit_on, 140
aperture_type, 91
arithmetic expressions, 32
 constants, 32
 intrinsic functions, *see* intrinsic functions
 variables, 31
asin, 33
atan, 33
auto_bookkeeper, 147
auto_scale_field_amp, 139, 141
auto_scale_field_amp_default, 147
auto_scale_field_phase, 139, 141

auto_scale_field_phase_default, 147
automatic field scaling, 145

b1_gradient, 43, 72, 77, 83
b2_gradient, 43, 76, 83
b3_gradient, 69, 83
b_field, 41, 83
b_field_err, 41, 83
b_max, 79, 83
b_param, 46
bbi_constant, 39, 83
be_thread_safe, 147
beam, 29, 301
beam initialization parameters, 150
beam line, *see* line
beam statement, 141
beam tracking
 list of routines, 345
beam_init_struct, 150
beam_start, 29, 293
beam_start statement, 141
beambeam, 39, 83, 117, 120, 122, 140, 214, 284
beginning, 29
beginning element, 21, 30
beginning statement, 27, 142, 170
beginning_ele, 40, 56, 85
bend_sol_quad, 40, 117, 120, 122
bend_tilt, 40
bendfringe, 113
beta_a, 142
beta_a0, 66
beta_a1, 66
beta_b, 142
beta_b0, 66
beta_b1, 66
bl_hkick, 83, 90
bl_kick, 83, 90
bl_vkick, 83, 90
Bmad, 2
 distribution, 257
 error reporting, 3

general parameters, **147**
 information, **3**
 lattice file format, **25**
 lattice format, *see* lattice file format
 statement syntax, **26**
 bmad version number, **296**
 bmad_com, **318**
 bmad_common_struct
 auto_bookkeeper, **291**
 max_aperture_limit, **307**
 bmad_parser, **296**
 bmad_standard
 mat6_calc_method, **119**
 spin_tracking_method, **122**
 tracking method, **176**
 tracking_method, **116**
 bmad_status, **318**
 bn, *see* multipole, bn
 bookkeeper_status_struct, **291**
 bookkeeping
 automatic, **290**
 intelligent, **290**
 boris, **124, 125**
 and field maps, **73**
 and Taylor maps, **124**
 tracking_method, **116**
 both_ends, **92**
 bragg_angle, **46**
 bragg_angle_in, **46**
 bragg_angle_out, **46**
 branch, **21, 22, 127, 142**
 root, **282**
 branch_struct, **282**
 bs_field, **74, 77, 83**
 bs_gradient, **76**
 bunch, **301**
 bunch initialization, **187**

 C++ interface, **325**
 classes, **325**
 Fortran calling C++, **326**
 call
 inline, **34**
 call statement, **34**
 canonical coordinates, *see* phase space coordinates
 capillary, **45, 105, 117, 120, 122**
 wall, **102**
 charge, **39, 83**
 chromaticity, **300**
 closed, **140**

 closed orbit, **307**
 cmat_ij, **142**
 coef, **60**
 coherent synchrotron radiation, *see* CSR
 coherent tracking, **225**
 coherent_synch_rad_on, **147**
 command, **60**
 comment symbol (!), **27**
 complex taylor map
 list of routines, **371**
 constants, **25, 316**
 continuation symbol (&), **27**
 continuous, **92**
 control_slave\$, **286**
 control_struct, **288**
 controller element, **21**
 conversion to other lattice formats, **163**
 coord_array_struct, **304**
 coord_struct, **301**
 coordinates, **167**
 global, *see* global coordinates
 list of routines, **362**
 phase space, *see* phase space coordinates
 reference, *see* reference orbit
 cos, **33**
 coupler_angle, **110**
 coupler_at, **110**
 coupler_phase, **110**
 coupler_strength, **110**
 coupling, *see* normal mode
 critical_angle_factor, **45**
 crotch chamber geometry, **102**
 crunch, **113**
 crunch_calib, **113**
 crystal, **46, 92, 93, 106, 120, 122, 168, 173, 228, 229**
 tilt correction, **234**
 crystal_type, **46**
 CSR, **197**
 csr parameters, **153**
 custom, **48, 117, 120, 122, 125, 313**
 mat6_calc_method, **119**
 reference energy, **317**
 spin_tracking_method, **122**
 tracking_method, **116**

 d1_thickness, **68**
 d2_thickness, **68**
 d_orb(6), **147**
 d_spacing, **46**
 dbragg_angle_de, **46**

de optimizer parameters, **154**
 de_eta_meas, **113**
 debug_marker statement, **35**
 default_ds_step, **147**
 default_integ_order, **147**
 delta_e, **49**, **83**
 dependent attribute, **83**
 deprecated routines, **380**
 descrip, **60**, **84**, **313**
 detector, **49**
 diffraction_plate, **49**, **93**
 digested files, **28**, **296**
 dispersion, **190**, **194**, **200**
 dks_ds, **40**
 dphi_a, **66**
 dphi_b, **66**
 drift, **50**, **117**, **120**, **122**, **135**, **216**
 superposition, **136**
 ds_step, **83**, **124**, **309**

 e1, **41**, **42**, **74**
 e2, **41**, **42**, **74**
 e_field, **52**, **83**
 e_gun, **51**, **85**, **141**, **145**
 e_loss, **63**, **83**
 e_tot, **83**, **85**, **138–140**, **142**
 e_tot_offset, **71**
 e_tot_start, **85**
 ecollimator, **45**, **91**, **92**, **117**, **120**, **122**
 ele
 %status, **291**
 ele_origin, **135**
 ele_pointer_struct, **264**
 ele_struct, **264**, **269**
 %a, **273**, **297**
 %a_pole(:,), **278**
 %alias, **271**
 %aperture_type, **307**
 %b, **273**, **297**
 %b_pole(:,), **278**
 %c, **297**
 %c_mat, **273**, **297**
 %component_name, **274**
 %descrip, **271**
 %em_field, **276**
 %emit, **297**
 %field_master, **272**
 %floor, **274**
 %gamma_c, **273**, **297**
 %gen0, **275**
 %gen_field, **275**

 %ic1_lord, **274**, **289**
 %ic2_lord, **274**, **289**
 %ix1_slave, **274**
 %ix2_slave, **274**
 %ix_branch, **273**
 %ix_ele, **273**, **292**
 %ix_pointer, **279**
 %key, **265**, **272**
 %lat, **273**
 %logic, **279**
 %lord_status, **274**, **285**, **286**
 %map_ref_orb_in, **275**
 %map_ref_orb_out, **275**
 %mat6, **275**, **308**
 %mode3, **273**, **298**
 %mode_flip, **273**
 %n_lord, **274**, **287**
 %n_slave, **274**, **287**
 %name, **271**
 %norm_emit, **297**
 %old_value(:,), **272**
 %r, **279**
 %ref_time, **275**
 %s, **265**, **275**
 %sigma, **297**
 %sigma_p, **297**
 %slave_status, **274**, **285**, **286**
 %sub_key, **272**
 %tracking_method, **308**
 %type, **271**
 %value(:,), **272**
 %vec0, **275**
 %wake, **276**, **277**
 %x, **265**
 %z, **273**, **297**
 attribute values, **272**
 components not used by Bmad, **279**
 in lat_struct, **283**
 initialization, **271**
 multipoles, **278**
 pointer components, **271**
 Taylor maps, **275**
 transfer maps, **275**
 electron, **140**
 electron\$, **284**
 element, **21**, **37**
 class, **115**
 name, **29**
 names, **29**
 table of class types, **117**
 element attribute, **83**

dependent and independent, 83
 Element attribute bookkeeping, 271
 element attributes, 30
 defining custom attributes, 31
 element coordinates, 207, 228
 element reversal, 129
 elseparator, 52, 83, 90, 117, 120, 122, 181, 216, 218
 em_field, 52, 85, 107, 141, 145, 177
 emittance_a, 141
 emittance_b, 141
 emittance_z, 141
 end, 29
 end element, 21, 140
 end_edge, 60
 end_file statement, 35
 energy, 141
 Enge function, 43
 entrance_end, 92, 168, 307
 eps_step_scale, 74
 eta_x, 142
 eta_x0, 66
 eta_y, 142
 eta_y0, 66
 etap_x, 142
 etap_x0, 66
 etap_y, 142
 etap_y0, 66
 exit_end, 92, 168, 307
 exit_on_error, 318
 exp, 33
 expand_lattice, 33, 35, 131, 137

 f1, 41, 42, 72, 74
 f2, 72, 74
 f_damp, 107
 factorial, 33
 fftw
 library, 257
 fgsl
 library, 257
 fibre, 321
 fiducial, 53, 56, 174
 field, 107
 field_calc, 49, 106, 124
 field_master, 84
 field_scale, 107
 field_x, 141
 field_y, 141
 fint, 41, 42
 fintx, 41, 42

 flexible, 71
 flexible patch, 53, 71
 floor coordinates, *see* global coordinates
 floor_position_struct, 274
 floor_shift, 54, 168, 174
 follow_diffracted_beam, 46
 Forest, Etienne, *see* PTC/FPP
 fork, 55, 56, 282, 305
 FPP, *see* PTC/FPP
 FPP/PTC
 phase space convention, 176
 free\$, 285, 286
 freq, 107
 fringe fields, 112
 fringe_at, 112
 fringe_type, 74, 112
 functions, *see* intrinsic functions

 g, 40, 41, 43, 74, 83
 g_err, 41, 43, 83
 gap, 52, 83
 geometry, 135, 139, 264
 getf, 258
 girder, 57, 61, 70, 105, 135, 174, 274, 283, 288
 girder_lord\$, 285
 global coordinates, 170, 317
 in ele_struct, 274
 list of routines, 352
 reference orbit origin, 170
 grad_loss_sr_wake, 147
 gradient, 51, 63, 83
 gradient_err, 51
 graze_angle, 68
 grid, 107
 group, 60, 70, 135, 283, 288
 reference energy, 317
 group_lord, 285
 gsl
 library, 257

 h1, 41, 43
 h2, 41, 43
 h_displace, 62
 harmon, 73, 74, 83, 272
 harmon_master, 73
 hgap, 41
 hgpx, 41, 42
 hkick, 52, 62, 83, 90
 kicker, 62, 90, 117, 120, 122, 181, 218
 hybrid, 61
 reference energy, 317

incoherent tracking, 225
infleible patch, 71
instrument, 61, 113, 117, 120, 122
integration methods, 124
integrator_order, 124, 309
intrinsic functions, 33
is_on, 56, 65, 106

k1, 40, 41, 43, 72, 77, 79, 83
k2, 43, 76, 83
k3, 69, 83
kick, 62, 83, 90
kicker, 62, 90, 117, 120, 122, 181, 218
kill_fringe, 74
knl, *see* multipole, knl
ks, 40, 74, 76, 77, 83

l, 42, 43, 46, 54, 66, 67, 83, 105
l_arc, 41, 43
l_chord, 41, 43, 83, 105
LAPACK
 library, 258
LAPACK95
 library, 258
lat_param_struct, 284
 %n_part, 284
 %stable, 284
 %t1_no_RF, 284
 %t1_with_RF, 284
 %total_length, 284
 aperture_limit_on, 307
 end_lost_at, 307
 ix_lost, 307
 ix_track, 304
 lost, 307
lat_struct, 264, 274, 281
 %beam_start, 293
 %branch(:,), 282
 %control, 288, 289
 %ele(:,), 265, 283
 %ele_init, 281
 %ic, 289
 %ix1_slave, 289
 %ix2_slave, 289
 %n_ele_max, 283
 %n_ele_track, 283
 %param, *see* lat_param_struct
 example use of, 265
 initializing, 282
 pointers, 282
lattice, 22, 139
 expansion, 35, 83
 lattice element, 21
 lattice files, 25
 MAD files, 296
 name syntax, 27
 parser debugging, 35
 reading, 295
 reading and writing routines, 357
 XSIF, *see* XSIF
 lattice statement, 140
lcavity, 63, 83, 85, 107, 110, 117, 120, 122, 138, 140, 141, 145, 175–177, 213, 218, 221, 276
 and geometry, 140
 and param%n_part, 284
 reference energy, 317
 reference phase, 108
length of elements, 105
limit, 91
line, 28, 35, 127, 265
 with arguments, 129
linear, 308
 tracking_method, 116
list, 35, 127, 129
listf, 258
log, 33
logicals, 31
lord, 285
lord_pad1, 136
lord_pad2, 136
lords
 ordering, 288
lr_freq_spread, 110, 278
lr_wake_file, 110
lr_wakes_on, 147

m, 107
macroparticles, 189
 tracking, 301
MAD, 3, 25, 37, 129, 130, 141, 164, 170, 171, 179, 296
 beam statement, 141
 conversion, 163
 delayed substitution, 32
 element rotation origin, 87
 MAD-8, 296
 mat6_calc_method, 119
 phase space convention, 176
 radiation, 193
 syntax compatibility with BMAD, 32
tracking_method, 116

units, 25
 magnetic fields, 179
 map, *see* transfer map, 107
 marker, 51, 56, 65, 113, 117, 120, 122
 master_scale, 107
 mat6_calc_method, 49, 115, 119
 mat6_track_symmetric, 147
 match, 65, 117, 120, 122
 match_end, 66
 match_end_orbit, 66
 material_type, 68
 matrix
 list of routines, 358
 matrix manipulation, 341
 max_aperture_limit, 147
 measurement, 251
 measurement simulations
 list of routines, 360
 mirror, 67, 92, 93, 168, 173, 219, 228, 229
 mode, 50, 107
 mode3_struct, 298
 monitor, 61, 113, 117, 120, 122
 multilayer_mirror, 68, 92, 93, 228
 multipass, 29, 35, 63, 73, 133, 137
 list of routines, 360
 multipass_lord, 288
 multipass_lord\$, 285
 multipass_slave, 288
 multipass_slave\$, 286
 multipole, 67, 106, 117, 120, 122, 181
 %scale_multipoles, 278
 an, bn, 106, 180
 in ele_struct, 278
 KnL, Tn, 179, 180
 in ele_struct, 278
 knL, tn, 106
 list of routines, 360
 muon, 140
 muon\$, 284
 n_cell, 63, 68
 n_part, 83, 139–141
 in BeamBeam element, 39
 n_pole, 79
 n_ref_pass, 85, 138
 n_sample, 113
 n_slice, 39
 no_aperture, 92
 no_digested statement, 35
 no_end_marker, 139
 no_superimpose statement, 35
 noise, 113
 normal mode
 a-mode, 199
 b-mode, 199
 Coupling, 199
 not_a_lord\$, 285
 null_ele, 51, 68
 num_steps, 83, 124
 numerical recipes
 library, 258, 341
 octupole, 68, 83, 117, 120, 122, 181, 219
 tilt default, 87
 offset, 135
 offset_moves_aperture, 91
 ok, 318
 old_command, 60
 OPAL, 323
 phase space, 323
 open, 66, 135, 140
 optimizers, 341
 orbit
 measurement, 251
 origin_ele, 53, 57
 origin_ele_ref_pt, 53, 57
 osc_amplitude, 113
 overlay, 59, 61, 135, 283, 288, 293
 reference energy, 317
 overlay, 69
 p0c, 85, 138–140, 142
 p0c_start, 85
 parameter, 29
 parameter statement, 26, 27, 83, 85, 139
 geometry, 164
 paraxial approximation, 176
 parser_debug statement, 35
 particle, 141
 patch, 53, 55, 70, 85, 105, 117, 120, 122, 140,
 159, 168, 174, 219, 288
 and chamber wall, 102
 example, 158
 reflection, 174
 pendellosung_period_pi, 46
 pendellosung_period_sigma, 46
 permfringe, 113
 pgplot
 and Quick_Plot, 329
 library, 257
 phase space coordinates, 175, 301
 MAD convention, 176

PTC convention, 176
 phase_x, 141
 phase_y, 141
 phi0, 63, 73, 74
 phi0_azimuth, 107
 phi0_multipass, 63, 73
 phi0_ref, 107
 phi_a, 142
 phi_b, 142
 phi_origin, 53, 57
 phi_position, 142
 photon, 140
 phase space coordinates, 177
 photon_fork, 55, 56, 282, 305
 photon_type, 139, 140
 photons
 list of routines, 362
 pion_0, 140
 pion_0\$, 284
 pion_minus, 140
 pion_minus\$, 284
 pion_plus, 140
 pion_plus\$, 284
 pipe, 61
 superposition, 136
 polarity, 79
 positron, 140
 positron\$, 284
 print statement, 33
 programming
 conventions, 260
 example program, 263
 precision (rp), 259
 programming common parameters, 154
 proton, 140
 proton\$, 284
 psi_angle, 46, 235
 psi_origin, 53, 57
 psi_position, 142
 PTC, 24
 single element mode, 24
 whole lattice mode, 24
 PTC/FPP, 176, 319
 initialization, 321
 library, 258
 list of routines, 363
 patch, 322
 phase space, 320
 real_8, 321
 Taylor Maps, 321
 universal_taylor, 321
 ptc_exact_misalign, 125, 139, 141
 ptc_exact_model, 125, 139, 141
 ptc_max_fringe_order, 112, 141
 px, 141
 px0, 66
 px1, 66
 py, 141
 py0, 66
 py1, 66
 pz, 141
 pz0, 66
 pz1, 66
 qp_axis_struct, 338
 qp_line_struct, 338
 qp_symbol_struct, 338
 quad_tilt, 40
 quadrupole, 72, 83, 117, 120, 122, 181, 212, 220
 tilt default, 87
 quick plot
 list of routines, 364
 quick_plot, 329
 axes, 334
 color styles, 335
 fill styles, 335
 line styles, 335
 position units, 333
 structures, 338
 symbol styles, 335
 symbol table, 335
 radiation damping and excitation, *see*
 synchrotron radiation
 radiation_damping_on, 147
 radiation_fluctuations_on, 147
 radius, 106, 181
 ran, 28, 33, 35
 ran_gauss, 28, 33, 35
 ran_seed, 33, 139
 rbend, 41, 83, 90, 105, 112, 117, 120, 122, 128,
 168, 171, 174, 181, 272, 280
 rcollimator, 45, 91, 92, 117, 120, 122
 ref, 135
 ref_origin, 135
 ref_tilt, 43, 173, 230, 235
 ref_time, 142
 ref_wave_length, 46, 85
 reference energy, 106, 290, 317
 reference orbit, 106, 167
 construction, 168
 origin in global coordinates, 170

reflection of elements, 128
 rel_tol_adaptive_tracking, 124, 147
 rel_tol_tracking, 147
 rel_tracking_charge, 140
 relative time tracking, 177
 replacement list, see list
 reserved names, 29
 return statement, 35
 RF field map, 73
 rf fields, 181
 rf_frequency, 63, 73, 74, 83
 rfcavity, 72, 83, 107, 110, 117, 120, 122, 141, 145, 175, 177, 213, 221, 276
 reference phase, 108
 rho, 40, 43, 74, 79, 83, 171
 rigid patch, 71
 roll, 41, 86, 174
 roll_tot, 90
 root, 29
 root branch, 22, 142, 169
 rp, 259
 runge_kutta, 124, 125
 and field maps, 73
 and Taylor maps, 124
 tracking_method, 116

 s-positions, 317
 SAD
 hyperbf, 164
 sad, 112
 sad_mult, 74
 sample, 75, 93
 sbend, 41, 83, 90, 105, 112, 117, 120, 122, 128, 168, 171, 174, 181, 215, 272, 280
 secondary lattice file, 35
 sextupole, 76, 83, 117, 120, 122, 181, 212, 221
 tilt default, 87
 sig_x, 39, 83
 sig_y, 39, 83
 sig_z, 39
 sim_utils library, 257
 sin, 33
 slave, 285
 ordering, 288
 slice_slave\$, 286
 sol_quad, 77, 83, 117, 120, 122, 181, 222
 conversion to MAD, 163
 tilt default, 87
 solenoid, 76, 83, 117, 120, 122, 181, 216, 223
 space_charge_on, 147
 spin, 213, 301

 spin tracking
 list of routines, 370
 methods, 122
 spin_tracking_method, 115, 122
 spin_tracking_on, 147
 sqrt, 33
 sr_wake_file, 110
 sr_wakes_on, 147
 ss:coher, 226
 start_edge, 60
 static
 mat6_calc_method, 119
 status, 318
 strings, 31
 structures, 260
 sub_type_out, 318
 super_lord, 288
 super_lord\$, 285
 super_slave, 288
 super_slave\$, 286
 superimpose, 28, 29, 133
 example, 266
 superposition, 133
 reference energy, 317
 surface, 92
 surface curvature, 96
 surface grid, 97
 switches, 31
 symmetric_edge, 60
 symp_lie_Bmad, 309
 tracking_method, 116
 symp_lie_bmad, 124
 and field maps, 73
 and Taylor maps, 124
 mat6_calc_method, 119
 symp_lie_PTC, 309
 symp_lie_ptc, 124
 and Taylor maps, 124
 mat6_calc_method, 119
 spin_tracking_method, 122
 tracking_method, 116
 symp_map, 309
 tracking_method, 116, 119
 symplectic
 conjugate, 199
 integration, 124
 symplectic integration, 203, 205, 212
 symplectification, 204
 symplectify, 125
 symplectify, 119
 synchrotron radiation

calculating, 308
damping and excitation, 193
integrals, 194

t_offset, 71
tags for Lines and Lists, 35
tags for lines and lists, 130
tan, 33
Tao, 2
tao, 341
taylor, 77, 117, 120, 122, 124, 309
and Taylor maps, 124
deallocating, 309
mat6_calc_method, 119
tracking_method, 116

taylor Map, 308
taylor map, 203
feed-down, 205
list of routines, 372
reference coordinates, 204
structure in ele_struct, 275
with digested files, 296

taylor_map_includes_offsets, 119, 126
taylor_order, 139, 140, 147
term (for a Wiggler), 79
theta_origin, 53, 57
theta_position, 142
thickness, 46
tilt, 40, 54, 56, 59, 62, 65, 69, 71, 72, 86, 92, 113, 171, 174, 230, 274
tilt_calib, 113
tilt_corr, 46, 230, 234
tilt_err_tot, 90
tilt_tot, 90, 274
time
phase space coordinates, 177
time_runge_kutta, 124
tracking_method, 117

title statement, 34

tn, *see* multipole, tn

to_element, 56

Touschek Scattering, 191

tracking, 301
apertures, 307
list of routines, 374
Macroparticles, 301
mat6_calc_method, 120, 122
partial, 307
particle distributions, 310
reverse, 309
spin, 310

tracking methods, 115
tracking_method, 49, 115
transfer map
in ele_struct, 275
mat6_calc_method, *see* mat6_calc_method
Taylor map, *see* Taylor map

tune
calculation, 298
setting, 299

tune tracker simulation, 243

twiss
list of routines, 378, 379
twiss parameters, 298
calculation, 298

twiss_struct, 297

type, 60, 84
type_out, 318

units
with MAD, 25

Universal Accelerator Parser (UAP), 164

use, 35
use statement, 26, 28, 130
use_ptc_layout_default, 147

v1_unitcell, 68
v2_unitcell, 68
v_displace, 62
v_unitcell, 46
val1, ..., Val12, 49
variables, *see* lattice file format, variables

vkick, 52, 62, 83, 90
vkicker, 62, 90, 117, 120, 122, 181, 218
voltage, 51, 52, 63, 73, 74, 83
voltage_err, 51

wake fields, 183
in ele_struct, 276
list of routines, 380
long-range, 184
short-range, 183
wake_lr_struct, 277
wake_sr_mode_struct, 277
wall, 98
wig_term_struct, 278
wiggler, 78, 83, 105, 117, 120, 122, 168, 212, 272, 280
conversion to MAD, 163
types, 278

x, 141
x0, 66

x1, 66
 x1_limit, 91, 273
 x2_limit, 91, 273
 x_gain_calib, 113
 x_gain_err, 113
 x_half_length, 81
 x_limit, 91, 273
 x_offset, 39, 54, 56, 59, 62, 65, 71, 86, 91, 92,
 113, 168, 229, 274
 x_offset_calib, 113
 x_offset_mult, 74
 x_offset_tot, 90, 274
 x_origin, 53, 57
 x_pitch, 38, 39, 54, 59, 62, 71, 86, 92, 168, 229,
 274
 x_pitch_mult, 74
 x_pitch_tot, 90, 274
 x_position, 142
 x_quad, 40
 x_ray_init, 81
 x_ray_line_len, 65, 81
 xraylib
 library, 258
 XSIF, 25, 295
 hyperbf, 164
 reference, 385
 xsif
 library, 258
 xy_disp_struct, 297

y, 141
 y0, 66
 y1, 66
 y1_limit, 91
 y2_limit, 91
 y_gain_calib, 113
 y_gain_err, 113
 y_half_length, 81
 y_limit, 91
 y_offset, 39, 54, 56, 59, 62, 65, 71, 86, 92, 113,
 168, 229, 274
 y_offset_calib, 113
 y_offset_mult, 74
 y_offset_tot, 90, 274
 y_origin, 53, 57
 y_pitch, 38, 39, 54, 59, 62, 71, 86, 92, 168, 229,
 274
 y_pitch_mult, 74
 y_pitch_tot, 90, 274
 y_position, 142
 y_quad, 40