Revision: March 16, 2026

The Tao Manual

David Sagan

Contents

List of Figures

List of Tables

Part I

Reference Guide

Chapter 1

Overview

1.1 In the beginning...

Tao stands for "Tool for Accelerator Optics". *Tao* is a general purpose program for simulating high energy particle beams in accelerators and storage rings. This manual assumes you are already familiar with the basics of particle beam dynamics and its formalism. There are several books that introduce the topics very well. A good place to start is, for example, *The Physics of Particle Accelerators* by Klaus Wille[Wil00].

The simulation engine that *Tao* uses for doing such things as particle tracking is the *Bmad* software library[Bma06]. *Bmad* was developed as an object-oriented library so that common tasks, such as reading in a lattice file and particle tracking, did not have to be coded from scratch every time someone wanted to develop a program to calculate this, that or whatever. An understanding of the nitty-gritty details of the routines that comprise *Bmad* is not necessary, however, one should be familiar with the conventions that *Bmad* uses and this is covered in the *Bmad* manual.

Bmad was developed before *Tao*. As *Bmad* was being developed, it became apparent that many simulation programs had common needs: For example, plotting data, viewing machine parameters, etc. Because of this commonality, the *Tao* program was developed to reduce the time needed to develop a working programs without sacrificing flexibility. That is, while the "vanilla" version of the *Tao* program is quite a powerful simulation tool, *Tao* has been designed to be easily customizable so that extending *Tao* to solve new and different problems is relatively straight forward.

So, what is *Tao* good for? A large variety of applications: Single and multiparticle tracking, lattice simulation and analysis, lattice design, machine commissioning and correction, etc. Furthermore, it is designed to be extensible using interface "hooks" built into the program. This versatility has been used, for example, to enable *Tao* to directly read in measurement data from Cornell's Cesr storage ring and Jefferson Lab's FEL. Think of *Tao* as an accelerator design and analysis environment. But even without any customizations, *Tao* will do much analysis.

More information, including the most up–to–date version of this manual, can be found at the *Bmad* web site at:

```
classe.cornell.edu/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 Scott Berg, Michael Ehrlichman, Chris Mayes, and Jeff Smith for bug reports, suggestions, code improvements, Etc. To John Mastroberti and Kevin Kowalski for their work on a graphics user interface and associated plotting, And last but not least thanks also must go to Dave Rubin and Georg Hoffstaetter for their help, support, and patience.

1.2 Bmad and Tao Tutorial

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

1.3 Tao Examples

Example input files for running a number of different simulations can be found with the *Bmad* and *Tao* Tutorial (§1.2). Additionally, there are a number of examples in the directory

```
  $ACC_ROOT_DIR/bmad-doc/tao_examples/
```

where `$ACC_ROOT_DIR` is the base directory of your local *Bmad* Distribution or Release (a full description of *Bmad* Distributions and Releases is given in the *Bmad* and *Tao* Tutorial).

1.4 Manual Organization

This manual is divided into two parts. Part I is the reference section which defines the terms used by *Tao*, discusses *Tao* commands, etc. Part II is a programmer's guide which shows how to extend *Tao*'s capabilities including interfacing to Python and how to incorporate custom calculations into *Tao*.

1.5 Other Bmad based programs

Tao is not the only program based upon *Bmad*. There are other programs which are specialized for certain computations such as long-term tracking, beam break-up instability, multi-objective optimization, etc. The discussion of these programs is beyond the scope of this manual. More information along with instructions for obtaining the programs can be obtained from the *Bmad* web site (§1.1).

Chapter 2

Introduction

2.1 Obtaining Tao

A `Distribution` is a set of files, including *Bmad* and *Tao* source files, which are used to build the *Bmad*, the *Tao* program, and various other simulation programs. A `Release` is like a `Distribution` except that it is created on the Linux computer system at CLASSE (Cornell's Laboratory for Accelerator-based Sciences and Education). More information can be obtained from the *Bmad* web site.

If there is no local *Bmad* Guru to guide you, download and setup instructions for downloading a Distribution, environment variable setup, and building *Tao* is contained on the *Bmad* web site and will not be covered here.

2.2 Starting and Initializing Tao

The syntax for starting *Tao* is given in Sec. §10.1.

Initialization occurs when *Tao* is started. Initialization information is stored in one or more files as discussed in Chapter §10.

2.3 Running Tao with OpenMP

`OpenMP` is a standard that enables programs to run calculations with multiple threads which will reduce computation time. Certain calculations done by *Tao*, including beam tracking and dynamic aperture calculations, can be run multithreaded via OpenMP if the *Tao* executable file has been properly compiled. Interested users should consult their local *Bmad* Guru for guidance. Note: `OpenMP` multithreading involves using multiple cores of a single machine (unlike `Open MPI` which involves multiple machines). Therefore, it is not necessary to have a cluster of machines to use `OpenMP`.

To set the number of threads when running a program compiled with `OpenMP`, set the environment variable `OMP_NUM_THREADS`. Example:

```
  export OMP_NUM_THREADS=8
```

This may also be set during Tao runtime as the global parameter `n_threads`. For example:

```
  set global n_threads = 1  ! Use only a single thread
  set global n_threads = 4  ! Use four threads
```

See §11.29.13 for more information.

To the local *Bmad* Guru: Compiling and linking of *Tao* with `OpenMP` is documented on the *Bmad* web site. By default, `OpenMP` is not enabled. Essentially, OpenMP is enabled by modifying the `dist_prefs` file before compiling and linking.

2.4 Command Line Mode and Single Mode

After *Tao* is initialized, *Tao* interacts with the user though the command line. *Tao* has two modes for this. In `command line` mode, which is the default mode, *Tao* waits until the the `return` key is depressed to execute a command. Command line mode is described in Chapter §11.

In `single` mode, single keystrokes are interpreted as commands. *Tao* can be set up so that in `single mode` the pressing of certain keys increase or decrease variables. While the same effect can be achieved in the standard `line mode`, `single mode` allows for quick adjustments of variables. See Chapter §12 for more details.

2.5 Lattice Calculations

By default *Tao* recalculates lattice parameters and does tracking of particles after each command. The exception is for commands that do not change any parameter that would affect such calculations such as the `show` command. See §3.6 for more details. If the recalculation takes a significant amount of time, the recalculation may be suppressed using the `set global lattice_calc_on` command (§11.29.13) or the `set universe` command (§11.29.28).

2.6 Command Files and Aliases

Typing repetitive commands in command line mode can become tedious. *Tao* has two constructs to mitigate this: Aliases and Command Files.

Aliases are just like aliases in Unix. See Section §11.1 for more details.

Command files are like Unix shell scripts. A series of commands are put in a file and then that file can be called using the `call` command (§11.2).

Tao will call a command file at startup. The default name of this startup file is `tao.startup` but this name can be changed (§10.2).

Do loops (§11.9) are allowed with the following syntax:

```
  do <var> = <begin>, <end> {, <step>}
    ...
    tao command [[<var>]]
    ...
  enddo
```

The `<var>` can be used as a variable in the loop body but must be bracketed "[[<var>]]". The step size can be any integer positive or negative but not zero. Nested loops are allowed and command files can be called within do loops.

```
do i = 1, 100
  call set_quad_misalignment [[i]] ! command file to misalign quadrupoles
  zero_quad 1e-5*2^([[i]]-1) ! Some user supplied command to zero quad number [[i]]
enddo
```

To reduce unnecessary calculations, the logicals `global%lattice_calc_on` and `global%plot_on` can be toggled from within the command file. Also setting `global%quiet` can turn off verbose output to the terminal. Note: Setting `global%lattice_calc_on` to False will also disable any plotting calculations. Example

```
set global quiet = all          ! Turn off verbose output to the terminal.
set global lattice_calc_on = F  ! Turn off lattice  and plotting calculations
... do some stuff ...
set global lattice_calc_on = T  ! Turn back on
set global quiet = off
```

See §10.6 for more details.

A `end_file` command (§11.10) can be used to signal the end of the command file.

The `pause` command (§11.17) can be used to temporarily pause the command file.

Chapter 3

Organization and Structure

This chapter discusses how *Tao* is organized. After you are familiar with the basics of *Tao*, you might be interested to exploit its versatility by extending *Tao* to do custom calculations. For this, see Chapter 14.

3.1 The Organization of Tao: The Super_Universe

Many simulation problems fall into one of three categories:

- Design a lattice subject to various constraints.
- Simulate errors and changes in machine parameters. For example, you want to simulate what happens to the orbit, beta function, etc., when you change something in the machine.
- Simulate machine commissioning including simulating data measurement and correction. For example, you want to know what steering strength changes will make an orbit flat.

Programs that are written to solve these types of problems have common elements: You have variables you want to vary in your model of your machine, you have "data" that you want to view, and, in the first two categories above, you want to match the machine model to the data (in designing a lattice the constraints correspond to the data).

With this in mind, *Tao* was structured to implement the essential ingredients needed to solve these simulation problems. The information that *Tao* knows about can be divided into five (overlapping) categories:

Lattice
: Machine layout and component strengths, and the beam orbit (§3.4).

Data
: Anything that can be measured. For example: The orbit of a particle or the lattice beta functions, etc. (§6)

Variables
: Essentially, any lattice parameter or initial condition that can be varied. For example: quadrupole strengths, etc. (§5).

Plotting
Information used to draw graphs, display the lattice floor plan, etc. (§7).

Global Parameters
Tao has a set of parameters to control every aspect of how it behaves from the random number seed *Tao* uses to what optimizer is used for fitting data.

3.2 The Super_universe

The information in *Tao* deals is organized in a hierarchy of `‘‘structures’’`. At the top level, everything known to *Tao* is placed in a single structure called the `super_universe`.

Within the `super_universe`, lies one or more `universes` (§3.3), each `universe` containing a particular machine lattice and its associated data. This allows for the user to do analysis on multiple machines or multiple configurations of a single machine at the same time. The `super_universe` also contains the `variable`, `plotting`, and `global parameter` information.

3.3 The Universe

The *Tao* `super_universe` (§3.2) contains one or more `universes`. A `universe` contains a `lattice` (§3.4) plus whatever data (§6) one wishes to study within this lattice (i.e. twiss parameters, orbit, phase, etc.). Actually, there are three lattices within each universe: the **design** lattice, **model** lattice and **base** lattice. Initially, when *Tao* is started, all three lattices are identical and correspond to the lattice read in from the lattice description file (§10.4).

There are several situations in which multiple universes are useful. One case where multiple universes are useful is where data has been taken under different machine conditions. For example, suppose that a set of beam orbits have been measured in a storage ring with each orbit corresponding to a different steering element being set to some non-zero value. To determine what quadrupole settings will best reproduce the data, multiple universes can be setup, one universe for each of the orbit measurements. Variables can be defined to simultaneously vary the corresponding quadrupoles in each universe and *Tao*’s built in optimizer can vary the variables until the data as determined from the `model` lattice (§3.4) matches the measured data. This `orbit response matrix` (ORM) analysis is, in fact, a widely used procedure at many laboratories.

If multiple universes are present, it is important to be able to specify, when issuing commands to tao and when constructing *Tao* initialization files, what universe is being referred to when referencing parameters such as data, lattice elements or other stuff that is universe specific. [Note: *Tao* variables are *not* universe specific.] If no universe is specified with a command, the `default` universe will be used. This default universe is set set by the `set default universe` command (§11.29). When *Tao* starts up, the default universe is initially set to universe 1. Use the `show global` (§11.30) command to see the current default universe.

the syntax used to specify a particular universe or range of universes is attach a prefix of the form:

```
[<universe_range>]@<parameter>
```

Commas and colons can be used in the syntax for `<universe_range>`, similar to the `element list` format used to specify lattice elements (§4.3). When there is only a single Universe specified, the brackets `[...]` are optional. When the universe prefix is not present, the current default universe is used. The current default universe can also be specified using the number `-1`. Additionally, a “`*`” can be used as a wild card to denote all of the universes. Examples:

```
[2:4,7]@orbit.x ! The orbit.x data in universes 2, 3, 4 and 7.
[2]@orbit.x     ! The orbit.x data in universe 2.
2@orbit.x       ! Same as "2@orbit.x".
orbit.x         ! The orbit.x data in the current default universe.
-1@orbit.x      ! Same as "orbit.x".
*@orbit.x       ! orbit.x data in all the universes.
*@*             ! All the data in all the universes.
```

3.4 Lattices

A `lattice` consists of a machine description (the strength and placement of elements such as quadrupoles and bends, etc.), along with the beam orbit through them. There are actually three types of lattices:

Design Lattice
The `design` lattice corresponds to the lattice read in from the lattice description file(s) (§10.4). In many instances, this is the particular lattice that one wants the actual physical machine to conform to. The `design` lattice is fixed. Nothing is allowed to vary in this lattice.

Model Lattice
Initially the `model` lattice is the same as the `design` lattice. Except for some commands that explicitly set the `base` lattice, all *Tao* commands to vary lattice variables vary quantities in the `model` lattice. In particular, things like orbit correction involve varying `model` lattice variables until the `data`, as calculated from the `model`, matches the `data` as actually measured.

Base Lattice
It is sometimes convenient to designate a reference lattice so that changes in the `model` from the reference point can be examined. This reference lattice is called the `base` lattice. The `set` command (§11.29) is used to transfer information from the `design` or `model` lattices to the base lattice. Initially, the `base` lattice is set equal to the `design` lattice by *Tao*.

Lattices can have multiple `branches`. For example, two intersecting rings can be represented as a lattice with two branches, one for each ring. See the *Bmad* manual for more details. Many *Tao* commands operate on a particular lattice branch. For example, the `show lat` command prints the lattice elements of a particular branch. If no branch is specified with a command, the default branch is used. The default branch is set with the `set default branch` command (§11.29). Initially, when *Tao* is started, the default branch is set to branch 0. Use the `show global` (§11.30) command to see the current default branch.

3.5 Tracking Types

The are two types of tracking implemented in *Tao*: single particle tracking and many particle multi-bunch tracking. Single particle tracking is just that, the tracking of a single particle through the lattice. Many particle multi-bunch tracking creates a Gaussian distribution of particles at the beginning of the lattice and tracks each particle through the lattice, including any wakefields. Single particle tracking is used by default. The `global%track_type` parameter (§10.6), which is set in the initialization file, is used to set the tracking.

Particle spin tracking has also been set up for single particle and many particle tracking. See Sections §10.6 and §10.7 for details on setting up spin tracking.

3.6 Lattice Calculation

After each *Tao* command is processed, the lattice and "merit" function are recalculated and the plot window is regenerated. The merit function determines how well the `model` fits the measured data. See Chapter 8 for more information on the merit function and its use by the optimizer.

Below are the steps taken after each *Tao* command execution:

1. The data and variables used by the optimizer are re-determined. This is affected by commands such as `use, veto,` and `restore` and any changes in the status of elements in the ring (e.g. if any elements have been turned off).
2. If changes have been made to the lattice (e.g. variables changed) then the model lattice for all universes will be recalculated. The `model` orbit, linear transfer matrices and Twiss parameters are recalculated for every element. All data types will also be calculated at each element specified in the initialization file. For single particle tracking the linear transfer matrices and Twiss parameters are found about the tracked orbit. Tracking is performed using the tracking method defined for each element (i.e. Bmad Standard, Symplectic Lie, etc...). See the *Bmad* Reference manual for details on tracking and finding the linear transfer matrices and Twiss parameters.
3. The `model` data is recalculated from the `model` orbit, linear transfer matrices, Twiss parameters, particle beam information and global lattice parameters. Any custom data type calculations are performed *before* the standard *Tao* data types are calculated.
4. Any user specified data post-processing is performed in `tao_hook_post_process_data`.
5. The contributions to the merit function from the variables and data are computed.
6. Data and variable values are transferred to the plotting structures.
7. The plotting window is regenerated.

If a closed orbit is to be calculated, *Tao* uses an iterative method to converge on a solution where *Tao* starts with some initial orbit at the beginning of the lattice, tracks from this initial orbit through to the end of the lattice, and then adjusts the beginning orbit until the end orbit matches the beginning orbit. A problem arises if the tracked particle is lost before it reaches the end of the lattice since *Tao* has no good way to calculate how to adjust the beginning orbit to prevent the particle from getting lost. In this case, *Tao*, in desperation, will try the orbit specified by `particle_start` in the *Bmad* lattice file (see the *Bmad* manual for more details on setting `particle_start`). Note: `particle_start` can be varied while running *Tao* using the `set particle_start` (§11.29) or `change particle_start` (§11.3) commands.

If the recalculation takes a significant amount of time, the recalculation may be suppressed using the `set global lattice_calc_on` command (§11.29.13) or the `set universe` command (§11.29.28).

Chapter 4

Syntax

4.1 Command Line Syntax

In "`line mode`" (§11), commands are case sensitive. Multiple commands may be entered on one line using the semicolon ";" character as a separator. [However, a semicolon used as as part of an `alias` (§11.1) definition is part of that definition.] An exclamation mark "`!`" denotes the beginning of a comment and the exclamation mark and everything after it to the end of the line is ignored. Example:

```
set default uni = 2; show global  ! Two commands and a comment
```

The length of a command on a single line is currently limited to 1000 characters. Multiple lines may be used for a single command by putting a "&" character at the end of a line to be continued. Example:

```
set default &    ! Continue command to next line
uni = 2
```

Note that, for historical reasons, Bmad itself is case insensitive. Thus things like lattice element names within *Tao* commands will similarly be case insensitive.

4.2 Specifying a Single Lattice Element

A full description of how to specify a lattice element is given in section §3.6 "`Matching to Lattice Element Names`" in the *Bmad* manual. Generally, elements are specified using either their names or by their index number. Additionally, in *Tao*, the universe in which the element exists may be specified by prepending the element name by the universe number followed by the "`@`" sign. Examples:

```
Q3##2      ! 2nd instance of element named Q3 in branch 0 of the default universe.
134        ! Element with index 134 in branch 0 of the default universe.
1>>13      ! Element with index 13 in branch 1 of the default universe.
2@1>>TZ    ! Element named TZ in branch 1 of universe 2.
B37        ! Element named B37 of the default universe.
0@B37      ! Same as the previous example.
```

Note: element names are *not* case sensitive.

4.3 Lattice Element List Format

The syntax for specifying a set of lattice elements is called `element list` format. A element list is a list of items separated by a comma.[1] Each item of the list is one of:

Item Type	*Example*
A single element (§4.2)	"1»Q10W"
A name with wild card characters	"5@q*"
A range of elements in the form <ele1>:<ele2>	"b23w:67"
A class::name specification	"sbend::b*"

Example element list:

```
23, 45:74, quad::q*
```

The wild card characters "*" and/or "%" can be used. The "*" wildcard matches any number of characters, The "%" wildcard matches a single character. For example, "q%1*" matches any element whose name begins with "q" and whose third character is "1". If there are multiple elements in the lattice that match a given name, all such elements are included. Thus "d12" will match to all elements of that name. Examples

```
"134"        ! Element with index 134 in branch 0 of the default universe.
"1>>13"      ! Element with index 13 in branch 1 of the default universe.
"5@q*"       ! All elements whose name begins with "q" of universe 5.
"2@3>>q1##4" ! The fourth element named "q1" in branch 3 of universe 2.
"*@sex10w"   ! Element "sex10w" of all universes.
"b37"        ! Element "b37" of the default universe.
"0@b37"      ! Same as the previous example.
```

Note: element names are *not* case sensitive.

An element index item is simply the index of the number in the lattice list of elements. A prefix followed by the string "»" can be used to specify a branch. As with element names, a universe prefix can be given. Example

```
2@3>>183   ! Element number 183 of branch # 3 of universe 2.
```

A range of elements is specified using the format:

```
{<class>::}<ele1>:<ele2>
```

`<ele1>` is the element at the beginning of the range and `<ele2>` is the element at the end of the range. Either an element name or index can be used to specify `<ele1>` and `<ele2>`. Both `<ele1>` and `<ele2>` are part of the range. The optional `<class>` prefix can be used to select only those elements in the range that match the class. Example:

```
quad::sex10w:sex20w
```

This will select all quadrupoles between elements `sex10w` and `sex20w`.

A `class::name` item selects elements based upon their class (Eg: `quadrupole`, `marker`, etc.), and their name. The syntax is:

```
<element class>::<element name>
```

where `<element class>` is an element class and `<element name>` is the element name that can (and generally does) contain the wild card characters "%" and "*". Essentially this is an extension of the `element name` format. As with element names, a universe prefix can be given. Example:

```
"4@quad::q*"   ! All quadrupole whose name starts with "q" of universe 4.
```

[1]A blank space may be acceptable in some circumstances but a comma is always safe.

4.4 Arithmetic Expressions

Tao is able to handle arithmetic expressions within commands (§11) and in strings in a *Tao* initialization file. Arithmetic expressions can be used in a place where a real value or an array of real values are required. The standard binary operators are defined:

$a + b$	Addition
$a - b$	Subtraction
$a * b$	Multiplication
a / b	Division
$a \wedge b$	Exponentiation

The following intrinsic functions are also recognized (this is the same list as the *Bmad* parser):

`sqrt`(x)	Square Root
`log`(x)	Logarithm
`exp`(x)	Exponential
`sin`(x), `cos`(x)	Sine and Cosine
`tan`(x), `cot`(x)	Tangent and Cotangent
`asin`(x), `acos`(x)	Arc sine and Arc Cosine
`atan`(y)	Arc Tangent
`atan2`(y, x)	Arc Tangent
`abs`(x)	Absolute Value
`factorial(x)`	Factorial
`ran`()	Random number between 0 and 1
`ran_gauss`()	Gaussian distributed random number with unit RMS
`ran_gauss`(sig_cut)	Gaussian distributed random number truncated at sig_cut.
`int(x)`	Nearest integer with magnitude less then x
`nint(x)`	Nearest integer to x
`floor(x)`	Nearest integer less than x
`ceiling(x)`	Nearest integer greater than x
`modulo(a, p)`	a - floor(a/p) * p. Will be in range [0, p].
`average(arr)`	Average value of an array
`rms(arr)`	RMS value of an array
`sum(arr)`	Sum of array values.
`min(arr)`	Minimum of array values.
`max(arr)`	Maximum of array values.
`mass_of`(A)	Mass of particle A
`charge_of`(A)	Charge, in units of the elementary charge, of particle A
`anomalous_moment_of`(A)	Anomalous magnetic moment of particle A
`species`(A)	Integer ID associated with species A

Both `ran` and `ran_gauss` use a seeded random number generator. Setting the seed is described in Section §10.6.

Expressions may involve arrays of values. For example:

```
lat::orbit.x[5:8]      ! X-orbit at lattice elements 5 through 8.
[1, 2, 3]              ! A vector of size three.
```

When using vectors with binary operators or intrinsic functions, the standard rules apply. For example:

```
s * [a, b, c]          = [s*a, s*b, s*c]
[a, b, c] - [x, y, z] = [a-x, b-y, c-z]
tan([a, b, c])         = [tan(a), tan(b), tan(c)]
sum([a, b, c])         = a+b+c
min(a, b, c)           ! Error: Correct is min([a, b, c])
```

Note that *Tao* does not make a distinction between a scalar and a vector of length one.

See the following sections for the syntax for using data, variable, and lattice parameters in an expression. Use the `show value` command (§11.30.38) to show the results of expressions.

4.5 Specifying Data Parameters in Expressions

A data (§6.1) parameter "`token`" is a string that specifies a scalar or an array of data parameters. The general form for data tokens in expressions (§4.4) is:

```
  {[<universe(s)>]@}data::<d2.d1_name>[<index_list>]|<component>
```

where:

```
  <universe(s)>       Optional universe specification (§3.3)
  <d2.d1_name>        D2.D1 data name
  <index_list>        List of indexes.
  <component>         Component.
```

examples:

```
  [2:4,7]@data::orbit.x      ! The orbit.x data in universes 2, 3, 4 and 7.
  [2]@data::orbit.x          ! The orbit.x data in universe 2.
  2@data::orbit.x[4]         ! Fourth orbit.x datum in universe 2.
  data::orbit.x[4,7:9]|meas  ! Default uni measured values of datums 4, 7, 8, and 9.
  *@data::orbit.x            ! orbit.x data in all the universes.
  *@data::*                  ! All the data in all the universes.
```

It is important to keep in mind that data must be defined at startup in the appropriate initialization file as discussed in Sec. §10.10 before reference is made to data in an expression. The `<d2.d1_name>` data names that have been defined at initialization time may be viewed using the `show data` command. Note that these names are user defined and do not have to correspond to the data types given in Sec. §6.9. See Sec. §4.7 for how to use "lattice parameters" that correspond to the data types given in Sec. §6.9 and that do not need to be defined at initialization.

See Sec. §6.2 for a list of datum `<component>`s (when running *Tao*, view a particular datum with the `show data` command to see the list).

`<index_list>` is a list of indexes. `<index_list>` will determine how many elements are in the array. For example, `orbit.x[10:21,44]` represents an array of 13 elements.

Depending upon the context, some parts of a token may be omitted. For example, with the `set data` command the "`data::`" part of the token may be omitted. Example:

```
  set data 2@orbit.x|meas = var::quad_k1[5]|model - orbit.y[3]|ref
```

Here *Tao* will default to evaluating a token as data. In general, what may be omitted should be clear in context.

Data components that are computed by *Tao* may be used on the right hand side of an equal sign but may not be set. For example, the `model` value of a datum is computed by *Tao* but the `ref` value is not.

If multiple tokens are used in an expression, all tokens must evaluate to arrays of the same size.

4.6 Specifying Variable Parameters in Expressions

A variable (§5) parameter "`token`" is a string that specifies a scalar or an array of variable parameters. The general form for variable tokens in expressions (§4.4) is:

```
  var::<v1_name>[<index_list>]<component>
```

where:

```
  <universe(s)>       Optional universe specification (§3.3)
  <v1_name>           V1 variable name.
  <index_list>        List of indexes.
  <component>         component.
```

Examples:

```
  var::*                      ! All the variables
  var::quad_k1[*]|design      ! All design values of quad_k1.
  var::quad_k1[]|model        ! No values. That is, the empty set.
  var::quad_k1|model          ! Same as quad_k1[*]|model
```

It is important to keep in mind that variables must be defined at startup in the appropriate initialization file as discussed in Sec. §10.9 before reference is made to them in an expression. The defined `<v1_name>` variable names can be viewed using the `show variable` command. Since these names are user defined they will change if different initialization files are used.

See Sec. §5 for a list of `<components>` of a variable.

`<index_list>` is a list of indexes. `<index_list>` will determine how many elements are in the array. For example, `k_quad[10:21,44]` represents an array of 13 elements.

Depending upon the context, some parts of a token may be omitted. For example, with the `set variable` command the "`var::`" part of the token may be omitted. Example:

```
  set var quad_k1[5]|meas = data::2@orbit.x|meas
```

Here *Tao* will default to evaluating a token as a variable component. In general, what may be omitted should be clear in context.

Variable components that are computed by *Tao* may be used on the right hand side of an equal sign but may not be set. For example, the `design` value of a variable is computed by *Tao* but the `meas` value is not.

If multiple tokens are used in an expression, all tokens must evaluate to arrays of the same size.

4.7 Specifying Lattice Parameters in Expressions

"Lattice parameters" are like `data` parameters (§4.5) except lattice parameters are calculated from the lattice and do not have to be defined at initialization time. A lattice parameter "`token`" is a string that specifies a scalar or an array of lattice parameters. The general form for data tokens in expressions (§4.4) is:

```
  {[<universe(s)>]@}lat::<eval_param>[{<ref_ele>&}<element_list>{-><s_offset>}]{|<component>}
```

where:

```
  <universe(s)>       Optional universe specification (§3.3)
  <eval_param>        Name of the parameter to evaluate.
                        Possible data types listed in Sec. §6.9.
  <ref_ele>           Optional reference element.
  <element_list>      Evaluation point or points.
  <s_offset>          Longitudinal offset to evaluate at.
  <component>         Optional component.
```

The `<s_offset>` string can be an expression. Any parameter in this expression, if not qualified, will be interpreted as a parameter of the element containing the evaluation point. For example

```
  3@lat::orbit.x[q10w->-L/2]|model
```
Here "L" in the `<s_offset>` string "`-L/2`" is interpreted as the length of the element `q10w`. Other examples:
```
  3@lat::orbit.x[34:37]            ! Array of orbits at element 34 through 37 in universe 3.
  3@lat::orbit.x[q10w]|model       ! Orbit.x model value at exit end of element q10w
  3@lat::orbit.x[q10w->0.1]|model  ! Same as above except eval point is shifted 0.1 meters.
  lat::sigma.12[q10w]              ! Beam sigma matrix component at element q10w computed
                                   !  from lattice parameters.
```
The list of possible lattice `<eval_param>` names is given in Sec. §6.9. The table 6.2 shows which data names are associated with the lattice. Lattice parameters are independent of `data` parameters. For example, `lat::orbit.x` refers to the horizontal orbit while `data::orbit.x` refers to user defined data whose name corresponds to `orbit.x` and in fact there is nothing to prevent a user from assigning the name `orbit.x` to data that is derived from, say, the Twiss beta function.

Also notice the difference between, say, "`lat::orbit.x[10]`" and "`data::orbit.x[10]`". With the "`lat::`" source, the element index, in this case `10`, refers to the 10th lattice element. With the "`data::`" source, "`10`" refers to the 10^{th} element in the `orbit.x` data array which may or may not correspond to the 10^{th} lattice element.

The optional `<ref_ele>` specifies a reference element for the evaluation. For example
```
  lat::r.56[q0&qa:qb]
```
is an array of the $r(5, 6)$ matrix element of the transport map between element `q0` and each element in the range from element `qa` and `qb`.

The optional `<s_offset>` specifies a longitudinal offset for the evaluation point. This may be an expression.

4.8 Specifying Beam Parameters in Expressions

Beam parameters are like lattice parameters (§4.7) except beam parameters are derived from tracking a beam of particles and may only be used in an expression if beam tracking is turned on. A beam parameter "`token`" is a string that specifies a scalar or an array of beam parameters. The general form for data tokens in expressions (§4.4) is:
```
  {[<universe(s)>]@}beam::<eval_param>[{<ref_ele>&}<element_list>]{|<component>}
```
where:
```
  <universe(s)>      Optional universe specification (§3.3)
  <eval_param>       Name of the parameter
  <ref_ele>          Optional reference element.
  <element_list>     Evaluation point or points.
  <component>        Component.
```
Examples:
```
  2@beam::sigma.x[q10w]          Beam sigma at element q10w.
  beam::n_particle_loss[2&56]    Particle loss between elements 2 and 56.
```
The list of possible beam `<eval_param>` names is given in Sec. §6.9. The table 6.2 shows which data names are associated with beam tracking.

4.9 Specifying Element Parameters in Expressions

"Element parameters" are parameters associated with lattice elements like the quadrupole strength associated with an element. Element parameters also include derived quantities like the computed Twiss parameters and the beam orbit. An element parameter "`token`" is a string that specifies a scalar or an array of element parameters. The general form for element tokens in expressions is:

```
  {<universe(s)>@}ele::<element_list>[<parameter>]{|<component>}
  {<universe(s)>@}ele_mid::<element_list>[<parameter>]{|<component>}
```

where:

`<universe(s)>`	`Optional universe specification (§3.3)`
`<element_list>`	`List of element names or indexes.`
`<parameter>`	`Name of the element parameter`
`<component>`	`Component.`

Examples:

`3@ele_mid::34[orbit_x]`	`Orbit at middle of element with index 34 in universe 3.`
`ele::sex01w[k2]`	`Sextupole component of element sex01w`
`ele::Q01W[is_on]|model`	`The on/off status of element Q01W.`

There is some overlap between element parameters and lattice parameters (§4.7). For historical reasons, the `element` parameter syntax roughly follows a convention developed for *Bmad* lattice files which is somewhat different from the convention developed for *Tao* data. For example, the a-mode beta is named `beta.a` in *Tao* while *Bmad* uses the name `beta_a`. See the *Bmad* manual for more information on the *Bmad* lattice file syntax. The following table lists the parameters that have both *Tao* datum and *Bmad* element parameter names

Tao Datum	*Bmad* Element Parameter
`alpha.a, alpha.b`	`alpha_a, alpha_b`
`beta.a, beta.b`	`beta_a, beta_b`
`cmat.11, etc.`	`cmat_11, etc.`
`e_tot`	`e_tot`
`eta.a, eta.b`	`eta_a, eta_b`
`eta.x, eta.y`	`eta_x, eta_y`
`etap.a, etap.b`	`etap_a, etap_b`
`etap.x, etap.y`	`etap_x, etap_y`
`floor.x, floor.y, floor.z`	`x_position, y_position, z_position`
`floor.theta, floor.phi, floor.psi`	`theta_position, phi_position, psi_position`
`gamma.a, gamma.b`	`gamma_a, gamma_b`
`phase.a, phase.b`	`phi_a, phi_b`

Table 4.1: *Tao* datums that have equivalent *Bmad* element parameters.

The following table lists the parameters that have both *Tao* datum and *Bmad* particle orbit names

Tao Datum	*Bmad* Orbit Parameter
`orbit.x, orbit.y, orbit.z`	`x, y, z`
`orbit.px, orbit.py, orbit.pz`	`px, py, pz`
`spin.x, spin.y, spin.z`	`spin_x, spin_y, spin_z`
`spin.amp spin.theta, spin.phi`	`spinor_polarization, spinor_theta, spinor_phi`

Table 4.2: *Tao* datums that have equivalent *Bmad* orbital parameters.

For parameters that are varying throughout the element, like the Twiss parameters, `ele::` will evaluate the parameter at the exit end of the element and `ele_mid::` will evaluate the parameter at the middle of the element. For parameters that do not vary, like the quadrupole strength, use the `ele::` syntax.

Element list format (§4.3) is used for the `<element_list>` so an array of elements can be defined.

For element parameter that evaluate to a logical, if they are used on the right hand side of an expression where the result is a real number, a `True` value will be converted to a value of `1` and a `False` value is converted to a value of `0`.

4.10 Format Descriptors

Some *Tao* commands like `show lattice` (§11.30.21) have optional arguments where the format output of various quantities can be specified. *Tao* follows Fortran format descriptor notation. Since complete information is available on the Web (do a search for "fortran edit descriptor"), only a brief introduction will be given here.

Format descriptors are case insensitive. The commonly used descriptors with *Tao* are:

Form	Output
Aw	String
Fw.d	Real numbers. Fixed point (no exponent).
nPFw.d	Real numbers. Fixed point with the decimal point shifted n places.
ESw.d	Real numbers. Floating point (with exponent).
Lw	Logicals.
Iw	Integers.
Iw.r	Integers padded with zeros to width r.
wX	White space.
Tc	Tab to column c.

In the above, "`w`" is the width of the field (number of charactgers in the printed string) and "`d`" is the number of digits to the right of the decimal place,

Examples:

	Internal		
Format	Quantity	Output String	Comment
F7.2	76.1234	`"  76.12"`	Right justified.
1PF7.2	76.1234	`" 761.23"`	Shifted decimal place 1 digit.
F0.2	76.1234	`"76.12"`	0 Field width => Output width exactly fits.
F3.2	76.1234	`"***"`	Number overflows field width.
ES9.2	76.1234	`" 7.61E+01"`	Right justified.
L3	True	`"  T"`	Right justified.
I0	34	`"34"`	0 Field width => Actual width = number of digits.
I4	34	`"  34"`	Right justified.
I4.3	34	`" 034"`	Number padded with a zero to three digits.
A3	"abcdef"	`"abc"`	String truncated.
A3	`"ab    "`	`"ab "`	String truncated but looks left justified.
A	"abcdef"	`"abcdef"`	Output width exactly fits string.
A8	"abcdef"	`"  abcdef"`	Right justified.
4x		`"    "`	Four spaces.
T45			Next output string starts at column 45

Note: When a format descriptor is being used to construct a table (EG `show lattice` command), using a "`0`" for the field width is ill-advised since columns will not be properly aligned.

A comma delimited list is used for outputting multiple quantities. For example, the format "`I4, A`" is used to output an integer followed by a string.

If multiple quantities with the same format are to be outputted a `multiplier` prefix number can be used. For example, "`3A`" is equivalent to "`A, A, A`". If the format has a `P` prefix then parentheses can be used to separate the multiplier from the `P` prefix. Example: "`2(3PF7.2)`" is equivalent to "`3PF7.2, 3PF7.2`".

Note to programmers: In a code file, a format string must always be enclosed in parentheses.

Chapter 5

Variables

5.1 Overview

Tao defines objects called `variables` whose main purpose is to enable optimizations (§8). Essentially, a variable acts as as a controller for a lattice parameter. For example, a variable may control the `k1` quadrupole strength of a particular lattice element. Another use for variables is that a block of variables can be plotted for visual inspection. `Variables` can be defined in the *Tao* initialization files (§10.9).

Blocks of variables are associated with what is called a `v1_var` structure as illustrated in Figure 5.1. Each `v1_var` structure defined has a `name` with which to refer to in *Tao* commands. For example, if "`quad_k1`" is the name of a `v1_var`, then `quad_k1[5]` references the variable with index 5 in the array associated with the `quad_k1 v1_var` structure.

A set of variables within a `v1_var` block can be referred to by using using a comma `,` to separate their indexes. Additionally, a Colon "`:`" can be use to specify a range of variables. For example

```
  quad_k1[3:6,23]
```

refers to variables 3, 4, 5, 6, and 23. Instead of a number, the associated lattice element name can be used so if, in the above example, the lattice element named `q01` is associated with `quad_k1[1]`, etc., then the following is equivalent:

```
  quad_k1[q03:q06,q23]
```

Using lattice names instead of numbers is not valid if the same lattice element is associated with more than one variable in a `v1_var` array. This can happen, for example, if one variable controls an element's `x_offset` and another variable controls the same element's `y_offset`.

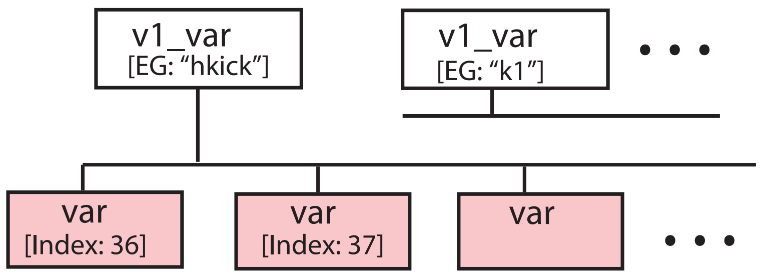


Figure 5.1: A `v1_var` structure holds an array of variables. Illustrated is a `var_var` structure holding an array of variables with each variable controlling the `hkick` attribute of a particular lattice element.

In referring to variables, a "*" can be used as a wild card to denote "all". Thus:

```
*                  ! All the variables
quad_k1[*]|design ! All design values of quad_k1.
quad_k1[]|model   ! No values. That is, the empty set.
quad_k1|model     ! Same as quad_k1[*]|model
```

5.2 Anatomy of a Variable

A given variable may control a single parameter of one element (or `particle_start`) in a `model` lattice of a single universe or it can be configured to simultaneously control an element attribute across multiple universes. Any one variable cannot control more than one attribute of one element. However, a variable may control an overlay or group element which, in turn, can control numerous elements.

Each individual variable has a number of values associated with it: The list of components that can be set or refereed to are:

```
ele_name     ! Associated lattice element name.
attrib_name  ! Name of the attribute to vary.
ix_attrib    ! Index in ele%value(:) array if appropriate.
s            ! longitudinal position of ele.

meas         ! Value of variable at time of a data measurement.
ref          ! Value at time of the reference data measurement.
model        ! Value in the model lattice.
base         ! Value in the base lattice.
design       ! Value in  the design lattice.
correction   ! Value determined by a fit to correct the lattice.
old          ! Scratch value.

weight       ! Weight used in the merit function.
delta_merit  ! Diff used to calculate the merit function term.
merit        ! merit_term = weight * delta^2.
merit_type   ! "target" or "limit"
dMerit_dVar  ! Merit derivative.

high_lim     ! High limit for the model_value.
low_lim      ! Low limit for the model_value.
step         ! For fitting/optimization: What is considered a small change.

key_bound    ! Variable bound to keyboard key?
ix_key_table ! Has a key binding?

ix_v1        ! Index of this var in the s%v1_var(i)%v(:) array.
ix_var       ! Index number of this var in the s%var(:) array.
ix_dvar      ! Column in the dData_dVar derivative matrix.

exists       ! Does the variable exist?
good_var     ! The variable can be varied (set by Tao).
good_user    ! The variable can be varied (set by the user).
good_opt     ! For use by extension code.
good_plot    ! Is variable within the horizontal extent of the plot?
```

```
  useit_opt    ! Variable is to be used for optimizing.
  useit_plot   ! Variable is to be used for plotting.
```

attrib_name
Name of the attribute to vary. Consult the *Bmad* manual for appropriate attribute names. If the attribute is associated with a lattice element, the `show element -all` command will list most attributes of interest. It is important to keep in mind that it is not possible to use attributes that are computed (that is, dependent attributes).

base
The value of the variable as derived from the `base` lattice (§3.3).

delta_merit
Difference value used to calculate the contribution of the variable to the merit function (Eq. (8.1)).

design
The value of the variable as given in the `design` lattice.

dMerit_dVar
Derivative of the merit function with respect to the variable.

ele_name
Associated lattice element name. For controlling the starting position in a lattice with open geometry the element name is `particle_start` (which is the name used if the starting position is set in the lattice file).

exists
The variable exists. Non-existent variables can serve as place holders in a variable array.

good_opt
Logical not modified by Tao proper and reserved for use by extension code. See below.

good_plot
Set by Tao. Is variable point within the horizontal extent of the plot?

good_var
Logical controlled by *Tao* and used to veto variables that should not be varied during optimization. For example, variables that do not affect the merit function. See below.

good_user
Logical set by the user using `veto`, `use`, and `restore` commands to indicate whether the variable should be used when optimizing. See below.

high_lim
High limit for the model value during optimization (§8.3) beyond which the contribution of the variable to the merit function is nonzero.

ix_attrib
Index assigned by *Bmad* to the attribute being controlled. Used for diagnosis and not of general interest.

ix_dvar
Column index of the variable in the dData_dVar derivative matrix constructed by *Tao*. Used for diagnostics and not of general interest.

ix_key_table
Index of the variable in the key table (§12.1).

ix_v1
Index of this variable in the variable array of the associated `v1_var` variable. For example, a variable named `q1_quad[10]` would have `ix_v1` equal to 10.

ix_var
For ease of computation, *Tao* establishes an array that holds all the variables. `ix_var` is the index number for this variable in this array. Used for diagnostics and not of general interest.

key_bound
Variable bound to keyboard key (§12.1)?

measured
The value of the variable as obtained at the time of a `data` measurement.

merit
The contribution to the merit function Eq. (8.1) from the variable. Use the `show merit` command to set the variables and data which contribute most to the merit function.

merit_type
"target" or "limit".

low_lim
Lower limit for the model value during optimization (§8.3) beyond which the contribution of the variable to the merit function is nonzero.

model
The value of the variable as given in the `model` lattice.

reference
The Value of the variable as obtained at the time of a `reference` data measurement (§8.4).

s
longitudinal position of element whose attribute the variable is controlling. Since a variable may control multipole attributes in multiple elements at different s-positions, The value of `s` may not be relevant.

step
What is considered a small change in the variable but large enough to be able to compute derivatives by changing the variable by `step`. Used for fitting/optimization.

useit_opt
Variable is to be used for optimization. See below.

useit_plot
If True, variable is used when plotting variable values. See below.

weight
Weight used in the merit function. w_j in Eq. (8.1)

These components and others can be refereed to in expressions using the notation documented in Sec. §4.6.

5.3 Use in Optimization and Graphing

Use the `show var` (§11.30) command to see variable information

When using optimization for lattice correction or lattice design (§8), Individual variables can be excluded from the process using the `veto` (§11.36), `restore` (§11.26), and `use` (§11.35) commands. These set the `good_user` component of a variable. This, combined with the setting `exists`, `good_var`, and `good_opt` determine the setting of `useit_opt` which is the component that determines if the datum is used in the computation of the merit function.

```
  useit_opt = exists & good_user & good_opt & good_var
```

The settings of everything but `good_user` and `good_opt` is determined by *Tao*

For a given `graph` that potentially will use a given variable for plotting, the `useit_plot` component is set to True if the variable is actually used for plotting. `useit_plot` is set by *Tao* using the prescription:

```
  useit_plot = exists & good_plot & good_var &
                            (good_user | graph:draw_only_good_user_data_or_vars)
```

Since `useit_plot` is set on a graph by graph basis, If multipole graphs that use a particular variable are to be plotted, The setting of `useit_plot` at the end of plotting will just be the setting for the last graph that was plotted.

5.4 Slave Value Mismatch

A "slave value mismatch" happens when a variable is controlling multiple parameters and something happens so that some subset of the controlled parameters have a change in value. For example, if a lattice has multiple elements named `‘‘Q’’`, and a variable controls the `k1` attribute of all of these elements, then, say, if only the value of `k1` of element `Q##2` (the 2nd instance of `Q` in the lattice) is changed, there will be a slave value mismatch. This can happen with the `change element` or `set element` command, and can also happen with the `read lattice` command. Additionally, slave value mismatches can happen if there are multiple universes with variables controlling parameters in more than one universe. In this case, modifying parameters in only one universe can cause slave value mismatches.

Tao can fix slave value mismatches when the `set element` or `change element` commands by using the changed parameter to set all the slave parameters. In other cases, *Tao* will not know what is changed and will simply set all the slave values to the value of the first slave.

Chapter 6

Data

A *Tao* ``datum'' is a parameter associated with a lattice that is used in lattice correction or design (§8). Example data includes the vertical orbit at a particular position or the horizontal emittance of a storage ring. This chapter explains how data is organized in *Tao* while Section §10.10 explains how, in an initialization file, to define the structures that hold the data. When running *Tao*, the `show data` (§11.30) command can be used to view information about the data.

6.1 Data Organization

The horizontal orbit at a particular BPM is an example of an individual `datum`. For ease of manipulation, arrays of datums are grouped into what is called a `d1_data` structure. Furthermore, sets of `d1_data` structures are grouped into what is called a `d2_data` structure. This is illustrated in Figure 6.1. For example, a `d2_data` structure for orbit data could contain two `d1_data` structures — one `d1_data` structure for the horizontal orbit data and another `d1_data` structure for the vertical orbit data. Each datum of, say, the horizontal orbit `d1_data` structure would then correspond to the horizontal orbit at some point in the machine.

When issuing *Tao* commands, all the data associated with a `d2_data` structure is specified using the `d2_data` structure's `name`. The data associated with a `d1_data` structure is specified using the format

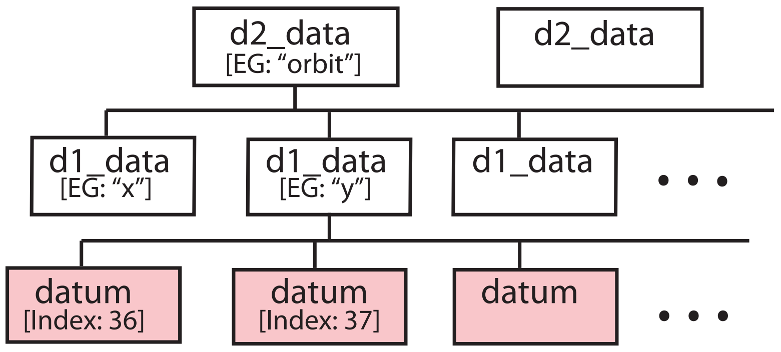


Figure 6.1: A `d2_data` structure holds a set of `d1_data` structures. A `d1_data` structure holds an array of datums.

```
  d2_name.d1_name
```

For example, if a `d2_data` structure has the name "`orbit`", and one of its `d1_data` structures has the name "`x`", then *Tao* commands that refer to the data in this `d1_data` structure use the name "`orbit.x`". Sometimes there is only one `d1_data` structure for a given `d2_data` structure. In this case the data can be referred to simply by using the `d2_data` structure's name. The individual datums can be referred to using the notation

```
  <d2_name>.<d1_name>[<list_of_datum_indexes>]
```

For example, `orbit.x[10]` refers to the horizontal orbit datum with index 10. Notice that the beginning (lowest) datum index is user selectable and is therefore not necessarily 1.

Period characters are not allowed in both the `d2_data` and `d1_data` names.

It is important to note that the name given to `d2_data` and `d1_data` structures is arbitrary and does not have to correspond to the type of data contained in the structures. In fact, a `d1_data` array can contain heterogeneous data types. Thus, for example, it is perfectly permissible (but definitely not recommended) to set up the data structures so that, say, `orbit.x[10]` is the a-mode emittance at a certain element and `orbit.x[11]` is the b-mode beta function at the same element.

Ranges of data can be referred to using using a comma `,` to separate the indexes combined with the notation `n1:n2` to specify all the datums between `n1` and `n2` inclusive. For example

```
  orbit.x[3:6,23]
```

refers to datums 3, 4, 5, 6, and 23.

If multiple universes are present, each universe will have its own set of `d2_data` structures. The name of a particular `d2_data` structure may be the same as a `d2_data` structure in a separate universe. So, for example, an `orbit d2_data` structure may be present in multiple universes.

As explained in §3.3, the prefix `"@"` may be used to specify which universe the data applies to. The general notation is

```
  [<universe_range>]@<d2_name>.<d1_name>[<datum_index>]
```

Examples:

```
  [2:4,7]@orbit.x ! The orbit.x data in universes 2, 3, 4 and 7.
  [2]@orbit.x     ! The orbit.x data in universe 2.
  2@orbit.x       ! Same as "2@orbit.x".
  orbit.x         ! The orbit.x data in the current default universe.
  -1@orbit.x      ! Same as "orbit.x".
```

As explained in Section §6.2, each individual datum has a number of components. The syntax to refer to a component is:

```
  d2_name.d1_name[datum_index]|component
```

For example:

```
  orbit.x[3:10]|meas     ! The measured data values
```

In referring to datums, a "`*`" can be used as a wild card to denote "all". Thus:

```
  *@orbit.x       ! The orbit.x data in all universes.
  *               ! All the data in the current default universe.
  *.*             ! Same as "*"
  *@*             ! All the data in all the universes.
  *@*.*           ! Same as "*@*"
  orbit.x[*]|meas ! All measured values of orbit.x
  orbit.x[]|meas  ! No values. That is, the empty set.
  orbit.x|meas    ! Same as orbit.x[*]|meas.
```

The last example shows that when referring to an entire block of data encompassed by a `d1_data` structure, the `[*]` can be omitted.

6.2 Anatomy of a Datum

Each datum has a number of components associated with it:

```
  data_type       ! Character: Type of data: "orbit.x", etc.
  ele_name        ! Character: Name of lattice element where datum is evaluated at.
  ele_start_name  ! Character: Name of starting lattice element in a range.
  ele_ref_name    ! Character: Name of reference lattice element.
  merit_type      ! Character: Type of constraint: "target", "max", etc.
  data_source     ! Character: How the datum is calculated. "lat", "beam", etc.
  ix_ele*         ! Integer: Index of "ele" in the lattice element list.
  ix_branch*      ! Integer: Lattice branch index.
  ix_ele_start*   ! Integer: Index of "ele_start" in the lattice element list.
  ix_ele_ref*     ! Integer: Index of "ele_ref" in the lattice element list.
  ix_ele_merit*   ! Integer: Lattice index where merit is evaluated.
  ix_d1*          ! Integer: Index number in d1_data structure
  ix_data*        ! Integer: Index in the global data array
  ix_dModel*      ! Integer: Row number in the dModel_dVar derivative matrix.
  ix_bunch        ! Integer: Bunch number to get the data from.
  eval_point      ! Character/integer: Evaluation point relative to the lattice element.
  meas            ! Real: User set measured datum value.
  ref             ! Real: User set measured datum value from the reference data set.
  model*          ! Real: Datum value as calculated from the model.
  design*         ! Real: What the datum value is in the design lattice.
  old*            ! Real: Used by Tao to save the model at some previous time.
  base*           ! Real: The value as calculated from the base model.
  fit†            ! Real: This value is not used by Tao.
  invalid         ! Real: The value used for delta_merit if good_model = False.
  error_rms       ! Real: Measurement error. Used for plotting.
  delta_merit*    ! Real: Diff used to calculate the merit function term.
  weight          ! Real: Weight for the merit function term
  merit*          ! Real: Merit function term value: weight * delta^2
  s*              ! Real: longitudinal position of ele.
  s_offset        ! Real: Offset of the evaluation point.
  spin_axis       ! Structure: Used for Spin G-matrix calculations.
  exists*         ! Logical: Does the datum exist?
  good_model*     ! Logical: Does the model  component contain a valid value?
  good_design*    ! Logical: Does the design component contain a valid value?
  good_base*      ! Logical: Does the base   component contain a valid value?
  good_meas       ! Logical: Does the meas   component contain a valid value?
  good_ref        ! Logical: Does the ref    component contain a valid value?
  good_user       ! Logical: Does the user want this datum used in optimization?
  good_opt†       ! Logical: Similar to good_user. Can be used in Tao extensions.
  good_plot†      ! Logical: Is datum within the horizontal extent of the plot?
  useit_plot*     ! Logical: Is this datum to be used in plotting?
  useit_opt*      ! Logical: Is this datum to be used for optimization?
```

*Set by *Tao*. †Used by *Tao* extensions. Not user settable.

When running *Tao*, the `show data` (§11.30) command can be used to view the components of a datum. The `set` command (§11.29) can be used to set some of these components. Note: Some of these components are set by the user and some of these components will be calculated by *Tao*. A description of what components can be set is given in Sec. §10.10.

base
The value of the datum as calculated from the base lattice (§6.3).

data_source
The `data_source` component specifies where the data is coming from (§6.6).

data_type
The type of data (§6.9). For example, `beta.a`. At startup, if the `data_type` is not specified, it is set to `<d2_name>.<d1_name>` where `<d2_name>` is the name of the associated `d2` data structure and `<d1_name>` is the name of the associated `d1` data structure (§10.10).

delta_merit
Difference used to calculate the contribution of the datum to the merit function (§8.4).

design
The value of the datum as calculated from the design lattice (§6.3).

ele_name
Name of the associated lattice element where the datum is evaluated at (§6.7) or, if `ele_start_name` is set, the last element in the evaluation range. Not used for datums that are `global`, like the emittance. Also see `eval_point` and `s_offset` components.

ele_start_name
Starting element of a range of lattice elements (§6.7).

ele_ref_name
Reference lattice element (§6.7). Not to be confused with the `ref` component which is a user settable value.

error_rms
The error associated with the measured or reference data values. Used for drawing error bars. See the documentation on `curve(N)%draw_error_bars` in Sec. §10.13.2. for more details.

eval_point
Set to `"beginning"`, `"center"`, or `"end"`. Used with `s_offset` to determine where the datum is evaluated at (§6.4). The evaluation point will be ignored if using an evaluation range (`ele_start_name` is set) is used.

exists
Set by *Tao* to True if the datum exists (§6.5).

fit
Not used by *Tao*. Can be used with custom code.

good_base
Set by *Tao*. Is the `base` value valid?

good_design
Set by *Tao*. Is the value as calculated from the `design` lattice valid? For example, if the datum is the particle orbit at some BPM in a ring and if it is not possible to compute the orbit at the BPM due to the lattice being unstable then `good_design` will be False.

good_meas
Set by *Tao*. Is the `meas` value valid?

good_model
Set by the user. Is the value as calculated from the `model` lattice valid? For example, if the datum is the particle orbit at some BPM in a lattice with open geometry and if it is not possible to compute the orbit at the BPM due to the particle being lost upstream of the element then `good_model` will be False.

good_opt
: Set by the user. This component is similar `good_user` except that it is unaffected by the `veto`, `restore` and `use` commands.

good_plot
: Set by Tao. Is datum within the horizontal extent of the plot?

good_ref
: Set by the user. Is the `ref` value valid?

good_user
: Set by the user. Is the datum valid for optimization or plotting? Use the commands `veto`, `restore`, `use` and `set data` to set while running *Tao*.

invalid
: The value used in the computation of `delta_merit` one of the following three conditions is True: 1) if `good_model` = False, or 2) if `good_base` is False when the global `opt_with_base` is True, or 3) if `good_design` is False when the global `opt_with_ref` is True.

ix_branch
: The index of the lattice branch that contains `ele`, `ref_ele`, and `start_ele`.

ix_ele
: Index of the lattice element where the datum is evaluated at.

ix_ele_start
: Index of the start element.

ix_ele_ref
: Index of the reference element.

ix_ele_merit
: Set by *Tao*. When the `merit_type` is set to `max` or `min` and there is a range of elements that over which the there is an evaluation, ix_ele_merit is set to the element where the value is the `max` or `min`.

ix_d1
: Index of the associated `d1_data` array.

ix_data
: For convenience, all the datums of a given universe are put into one large array. `ix_data` is the index of the datum in this array. This is useful for debugging purposes.

ix_dModel
: For optimization, *Tao* creates a derivative matrix dMerit_i/dVar_j. `ix_dmodel` is set to the ith column of this matrix. This is useful for debugging purposes

ix_bunch
: For datums that have `data_source` set to `beam`, `ix_bunch` selects which bunch of the beam the datum is evaluated at.

meas
: The value of the datum as obtained from some measurement (§6.3).

merit
: The contribution to the merit function due to this datum (§8.4).

merit_type
: The type of merit (§8.2). Possible values are:

```
  "target"
  "min",     "max"
  "abs_min", "abs_max"
  "max-min"                      ! Only used when datum specifies a range of elements.
  "average", "integral", "rms"   ! Only used when datum specifies a range of elements.
```

model
The value of the datum as calculated from the `model` lattice (§6.3).

old
A datum value that was saved at some point in *Tao*'s calculations. This value can be ignored (§6.3).

ref
The reference datum value as obtained from some reference measurement (§6.3). Set by the User. Not to be confused with the reference element.

s
Longitudinal s-position of the lattice element.

s_offset
Offset of the evaluation point when there is an associated lattice element (§6.4). The offset will be ignored if using an evaluation range (`ele_start_name` is set) is used.

spin_axis The `spin_axis` component is a structure used for Spin G-matrix calculations (`spin_g_matrix.`ij data type). `Spin_axis` gives the (l, n_0, m) coordinate axes at the reference element. See §10.10 for documentation on how to set this structure.

useit_opt
Datum is used for optimization. `useit_opt` is set by *Tao* using the other `logicals` components using the prescription:

```
  useit_opt = exists & useit_opt & good_user & good_meas &
              good_ref (if reference data is used in optimization)
Notice that if, for example, good_model is False then the datum will still be used for optimizatio
in this case the invalid value set by the user will be used in the computation for delta_merit in
place of a value computed from lattice.
```

useit_plot
Set `True` if the datum is valid for plotting for a particular graph. This component gets reevaluated for each `graph` that potentially uses the datum so the value observed after plotting is refreshed is simply the value as calculated for the last `graph` considered. The value for `useit_plot` is evaluated using other logical components using the prescription:

```
  useit_plot = exists & good_plot &
                  (good_user | graph:draw_only_good_user_data_or_vars) &
                  good_meas (if measured data is being plotted) &
                  good_ref (if reference data is being plotted) &
                  good_model (if model data is being plotted)
```

weight
Weight used in evaluating the contribution of the datum to the merit function (§8.4).

6.3 Datum values

A given datum has six values associated it:

meas
When fitting data, the `meas` value is the value of the datum as obtained from some measurement. When designing lattices, the `meas` value is the desired value of the datum. For example, when designing a lattice for a colliding ring machine, a datum may be constructed for the beta function at the interaction point with the `meas` value set to the desired value. See Chapter 8 for more details.

base
The datum value as calculated from the `base` lattice (§3.4).

design
The value of the datum as calculated from the `design` lattice (§3.4).

fit
The `fit` value is not used by *Tao* directly and is available for use by custom code.

model
The value of the datum as calculated from the `model` lattice (§3.4).

old
A datum value that was saved at some point in *Tao*'s calculations. This value can be ignored.

ref
When fitting data, `ref` is the datum value as obtained from some reference measurement. For example, a measurement before some variable is varied could be designated as the `reference`, and the datum taken after the variation could be designated the `measured` datum. When designing lattices, the `ref` value is the value of the datum associated with the `design` or `base` lattice (determined by the setting of the global `opt_with_base` parameter (§8.2). Note: The `meas` value of a datum is always associated with the `model` lattice.

6.4 Evaluation Point of a Datum

.

When the datum is to be evaluated at a specific point in the lattice, that is, when there is an associated lattice element, the default position for evaluating the datum is at the downstream end of the element. This evaluation point can be shifted using the `eval_point` and/or `s_offset` components.

The `eval_point` component can be set to one of:

```
  beginning   ! entrance end of lattice element.
  center      ! Center of lattice element
  end         ! Exit end of lattice element. Default.
```

The evaluation point is shifted by `s_offset` from the `eval_point`.

If there is a reference point, the setting of `eval_point` is used to determine where the reference point. The setting of `s_offset` is ignored for the reference point.

Due to internal logic considerations, Not all `data_types` are compatible with a finite `s_offset` or a setting of `eval_point` to `center`. The table of `data_types` (§6.2) shows which `data_types` are compatible and which are not.

Another restriction is that specifying a range of elements for evaluation (that is, specifying `ele_start_name` §10.10) is not compatible with a finite `s_offset` or a setting of `eval_point` to `center`.

6.5 Datums in Optimization

When using optimization for lattice correction or lattice design (§8), Individual datums can be excluded from the process using the `veto` (§11.36), `restore` (§11.26), and `use` (§11.35) commands. These set the `good_user` component of a datum. This, combined with the setting `exists`, `good_meas`, `good_ref`, and `good_opt` determine the setting of `useit_opt` which is the component that determines if the datum is used in the computation of the merit function. The settings of everything but `good_user` is determined by *Tao*. The value of `good_user` for a datum can be set in an initialization file (§10.10) or on the command line using the `use`, (§11.35) `veto` (§11.36), or `restore` (§11.26) commands.

The `exists` component is set by *Tao* to True if the datum exists and False otherwise. A datum may not exist if the type of datum requires the designation of an associated element but the `ele_name` component is blank. For example, a `d1_data` array set up to hold orbit data may use a numbering scheme that fits the lattice so that , say, datum number 34 in the array does not correspond to an existing BPM.

The `good_model` component is set according to whether a datum value can be computed from the `model` lattice. For example, If a circular lattice is unstable, the beta function and the closed orbit cannot be computed. Similarly, the `good_design` and `good_base` components mark whether the `design` and `base` values respectively are valid.

The `delta_merit` component of a datum is set to the `delta` value used in computing the contribution to the merit function (§8.2). If it is not possible to compute the datum value, then the `invalid` component is used for the computation of `delta_merit`. It is not possible to compute the datum value if one of the following three conditions is True: 1) `good_model` is False, 2) `good_design` is False and the global `opt_with_ref` is True, or 3) `good_base` is False and the global `opt_with_base` is True.

`good_meas` is set True if the `meas` component value is set in the data initialization file (§10.10) or is set using the `set` command (§11.29). Similarly, `good_ref` is set True if the `ref` component has been set. `good_ref` only affects the setting of `useit_opt` if the optimization is using reference data as set by the global variable `opt_with_ref` (§10.6).

Finally `good_opt` is meant for use in custom versions of *Tao* (§14) and is always left True by the standard *Tao* code.

Example of using a `show data` (§11.30) to check the logicals in a datum:

```
  Tao> show data 3@beta[1]

  Universe:   3
  %ele_name           = IP_L0
  %ele_ref_name       =
  %ele_start_name     =
  %data_type          = beta.a
      ... etc ...
  %exists             =  T
  %good_model         =  T
  %good_meas          =  F
  %good_ref           =  F
  %good_user          =  T
  %good_opt           =  T
  %good_plot          =  F
  %useit_plot         =  F
  %useit_opt          =  F
```

Here `useit_opt` is False since `good_meas` is False and `good_meas` is False since the `meas` value of the datum (not shown) was not set in the *Tao* initialization file or set using the `set` command.

6.6 Data_source

The `data_source` component specifies where the data is coming from. Possible values are:

```
  "beam"        ! Data from from multiparticle beam distribution
```

```
"data"        ! Data from from a Tao datum in a data array.
"lat"         ! Data from from the lattice.
```

If `data_source` is set to `"beam"`, the data is calculated using multiparticle tracking. If `data_source` is set to `"lat"`, the data is calculated using the "lattice" which here means everything *but* multiparticle tracking In particular, the `"lat"` `data_source` includes data derived from single particle tracking. For example, the `"beam"` based calculation of the emittance uses the bunch sigma matrix obtained through multiparticle tracking. The `"lat"` based calculation of the emittance uses radiation integrals.

Some data types may be restricted as to which `data_source` is possible. For example, a datum with `data_type` set to `n_particle_loss` must use `"beam"` for the `data_source`. Table 6.2 lists which `data_source` values are valid for what data types.

6.7 Datum Evaluation and Associated Lattice Elements

Datums can be divided up into two classes. In one class are the datums that are ``` ``local'' ```, like the beam orbit, which need to be evaluated at either a particular point are evaluated over some finite region of the machine. Other datums, like the emittance, are ``` ``global'' ``` and do not have associated evaluation points.

As mentioned, `local` datums may be evaluated at a specific point or over some evaluation region, an evaluation region is used when, for example, the maximum or minimum value over a region is wanted. To specify an evaluation point, an `evaluation element` must be associated with a datum. The evaluation point will be at the exit end of this element. To specify an evaluation region, a `start element` must also be associated with a datum along with the `evaluation element`. The evaluation region is from the exit end of the `start element` to the exit end of the `evaluation element`.

In addition to the `evaluation element` and the `start element`, a `local` datum may have an associated `reference element`. A `reference element` is used as a fiducial point and the datum value is calculated relative to that point. For example, a datum value may be the position of the `evaluation element` relative to the position of the `reference element`. The evaluation point of a `reference element` is the exit end of that element.

The components in a datum corresponding to the `evaluation element`, the `reference element`, and the `start element`. are shown in Table 6.1. These three elements may be specified for a datum by either setting the name component or the index component of the datum. Using the element index over the element name is necessary when more than one element in the lattice has the same name.

	Data Component	
Element	*name*	*index*
Reference Element	`ele_ref_name`	`ix_ele_ref`
Start Element	`ele_start_name`	`ix_ele_start`
Evaluation Element	`ele_name`	`ix_ele`

Table 6.1: The three lattice elements associated with a datum may be specified in the datum by setting the appropriate name component or by setting the appropriate index component.

If a datum has an associated `evaluation` element, but no associated `start` or `reference` elements, the `model` value of that datum is the value of the `data_type` at the `evaluation` element. For example, if a datum has:

```
data_type      = "orbit.x"
ele_name       = "q12"
```

here the `model` value of this datum will be the horizontal orbit at the element with name `q12`.

If a datum has an associated `start` element, specified by either setting the `ele_start_name` or `ix_ele_start` datum components, the datum is evaluated over a region from the exit end of the `start` element to the exit end of `evaluation` element. For example, if a datum has:

```
data_type      = "beta.a"
ele_name       = "q12"
ele_start_name = "q45"
merit_type     = "max"
```

then the `model` value of this datum will be the maximum value of the a-mode beta function in the `model` lattice in the region from the exit end of the element with name `q12` to the exit end of the element with name `q45`. The `base` and `design` values are computed similarly using the `base` and `design` lattices.

If the lattice branch associated with the datum has a closed geometry, and if the lattice element associated with `ele_start_name` is after the element specified by `ele_name`, the evaluation region will be the region from `ele_start_name` to the end of the branch along with the region from the beginning of the branch to the element specified by `ele_name`. That is, the evaluation region “wraps” around the end of the lattice.

It does not make sense to specify an evaluation range when the datum’s `merit_type` is set to `"target"`. In this case, the `model`, `base`, and `design` values are the value of the datum at the evaluation element. Also notice that for datums that do not have an evaluation element, for example, if `data_type` is set to `"emit.a"`, specifying an evaluation range does not make sense.

Typically, in evaluating a datum over some region to find the maximum or minimum, *Tao* will only evaluate the datum at the ends of the elements with the assumption that this is good enough. If this is not good enough, marker elements can be inserted into the lattice at locations that matter. For example, the maximum or minimum of the beta function typically occurs near the middle of a quadrupole so inserting marker elements in the middle of quadrupoles will improve the accuracy of finding the extremum beta.

If a datum has an associated `reference` element, specified by either setting the `ele_ref_name` or `ix_ele_ref` datum components, the `model` value of the datum is the value at the `evaluation` element (or the value over the range `ele_start` to the `evaluation` element if `ele_start` is specified), minus the `model` value at `ele_ref`. For example, if a datum has:

```
data_type      = "beta.a"
ele_name       = "q12"
ele_start_name = "q45"
ele_ref_name   = "q1"
merit_type     = "max"
```

then the `model` value of the datum will be the same as the previous example minus the value of the a-mode beta function at the exit end of element `q1`. There are a number of exceptions to the above rule and datum types treat the `reference` element in a different manner. For example, the `r.` data type uses the `reference` element as the starting point in constructing a transfer matrix.

Do not confuse the `ele_ref_name` component (see preceding paragraph) with the `ref` component. The `ref` component is set by the User and typically represents the measured value of the datum. For example, if two orbit measurements are made, the `meas` component of a datum can be set to the measured orbit for one of the measurements while the `ref` component can be set to the measured orbit for the other measurement. This way orbit differences can be analyzed.

6.8 Data Types Table

This section lists in table form all the data types defined by *Tao* and section §6.9 describes in detail these data types.

Table 6.2: Predefined Data Types in Tao

Pg#	*Data_ Type*	*Description*	*data_ source*	*Can use s_offset?*
[58]	alpha.a, .b	Normal-Mode alpha function	lat	Yes
[58]	apparent_emit.x, .y	Apparent emittance	lat, beam	No
[58]	beta.a, .b, .c	Normal-mode beta function	lat, beam	Yes
[58]	beta.x, .y, .z	Projected beta function	beam	No
[59]	bpm_cbar.22a, .12a, .11b, .12b	Measured coupling	lat	Yes
[59]	bpm_eta.x, .y	Measured dispersion	lat	Yes
[59]	bpm_orbit.x, .y	Measured orbit	lat, beam	Yes
[59]	bpm_phase.a, .b	Measured betatron phase	lat	Yes
[59]	bpm_k.22a, .12a, .11b, .12b	Measured coupling	lat	Yes
[59]	bunch_charge.live, .live_relative	Charge of live particles	beam	No
[59]	bunch_max.x, .y, .z, .px, .py, .pz	Max relative to centroid	beam	No
[59]	bunch_min.x, .y, .z, .px, .py, .pz	Min relative to centroid	beam	No
[59]	cmat.11, .12, .21, .22	Coupling matrix elements	lat	Yes
[59]	cbar.11, .12, .21, .22	Normalized coupling matrix	lat	Yes
[59]	chrom.a, .b	Chromaticities for a ring	lat	No
[60]	chrom.dbeta.a, .dbeta.b	Normalized Chromatic beta	lat	No
[60]	chrom.deta.x, .deta.y	Chromatic dispersions	lat	No
[60]	chrom.detap.x, .detap.y	Chromatic dispersion slopes	lat	No
[60]	chrom.dphi.a, .dphi.b	Chromatic betatron phase	lat	No
[60]	chrom.w.a, .w.b	Chromatic W-functions	lat	No
[60]	chrom_ptc.a.N, chrom_ptc.b.N, $N = 0, 1, \ldots$	Chromaticity Taylor terms	lat	no
[61]	curly_h.a, .b	Radiation integrals curly H function	lat	Yes
[61]	damp.j_a, .j_b, .j_z	Damping partition number	lat	No
[61]	deta_ds.a, .b	Dispersion derivatives	lat	Yes
[61]	deta_ds.x, .y	Dispersion derivatives	lat	Yes
[61]	dpx_dx, dpx_dy, etc.	Bunch <x px> / <x^2> & Etc...	beam	No
[61]	dynamic_aperture.N, $N = 1, 2, 3 \ldots$	Dynamic aperture	lat	No
[61]	e_tot_ref	Lattice reference energy (eV)	lat	No
[62]	element_attrib.<attr_name>	lattice element attribute	lat	No
[62]	emit.a, .b, .c	Emittance	lat, beam	No
[62]	emit.x, .y, .z	Projected emittance	lat, beam	No
[62]	eta.x, .y, .z	Dispersions	lat, beam	Yes
[62]	eta.a, .b	Normal-mode dispersions	lat, beam	Yes

Table 6.2: (continued)

Pg#	*Data_ Type*	*Description*	*data_ source*	*Can use s_ offset?*
[62]	etap.x, .y	Momentum dispersions	lat, beam	Yes
[62]	etap.a, .b	Momentum dispersions	lat, beam	Yes
[62]	expression:<expression>	See text above	lat	No
[63]	floor.x, .y, .z, theta, .phi, .psi	Lattice element global position	lat	Yes
[63]	floor_actual.x, .y, .z, .theta, .phi, .psi	Lattice element misaligned global position	lat	Yes
[63]	floor_orbit.x, .y, .z	global position of orbit	lat, beam	Yes
[63]	floor_orbit.theta, .phi, .psi	global position of orbit	lat, beam	Yes
[63]	gamma.a, .b	Normal-mode gamma function	lat	Yes
[63]	k.11b, .12a, .12b, .22a	Coupling	lat	Yes
[63]	momentum	Momentum: P*C_light (eV)	lat	Yes
[64]	momentum_compaction	Momentum compaction factor	lat	No
[64]	momentum_compaction_ptc.N $N = 0, 1, 2, \ldots$	Momentum compaction Taylor terms	lat	No
[64]	n_particle_loss	Number of particles lost	beam	No
[64]	norm_apparent_emit.x, .y	Normalized apparent emittance	lat, beam	No
[64]	norm_emit.a, .b, .c	Normalized beam emittance	lat, beam	No
[64]	norm_emit.x, .y, .z	Normalized projected emittance	lat, beam	No
[65]	normal.<type>.i.<monomial>	Normal form map component	lat	No
[64]	normal.h.<monomial>	Normal form driving term	lat	No
[66]	null	Data without model evaluation	lat, beam	No
[66]	orbit.amp_a, .amp_b	Orbit amplitude	lat	Yes
[66]	orbit.norm_amp_a, .norm_amp_b	Energy normalized amplitude	lat	Yes
[66]	orbit.energy	Total energy (eV)	lat, beam	Yes
[66]	orbit.kinetic	Kinetic energy (eV)	lat, beam	Yes
[66]	orbit.x, .y, .z .px, .py, .pz	Phase space orbit	lat, beam	Yes
[66]	periodic.tt.$ijklm\ldots$ $1 \le i, j, k, \ldots \le 6$	Periodic map Taylor terms	lat	No
[66]	phase.a, .b	Betatron phase	lat	Yes
[67]	phase_frac.a, .b	Fractional betatron phase	lat	No
[67]	phase_frac_diff	a - b mode phase difference $-\pi < \phi_{\texttt{frac}} < \pi$	lat	No
[67]	photon.intensity	Photon total intensity	lat, beam	No
[67]	photon.intensity_x, .intensity_y	Photon intensity components	lat, beam	No
[67]	photon.phase_x, .phase_y	Photon phase	lat, beam	No
[67]	ping_a.amp_x, .phase_x, .amp_y, .phase_y, .amp_sin_y, .amp_cos_y, .amp_sin_rel_y, .amp_cos_rel_y	Pinged beam a-mode response	lat	No

Table 6.2: (continued)

Pg#	*Data_ Type*	*Description*	*data_ source*	*Can use s_ offset?*
[68]	ping_b.amp_x, .phase_x, .amp_y, .phase_y, .amp_sin_x, .amp_cos_x, .amp_sin_rel_x, .amp_cos_rel_x	Pinged beam b-mode response	lat	No
[69]	r.ij $1 \le i,j \le 6$	Term in linear transfer map	lat	Yes
[69]	r56_compaction	R56 like compaction factor.	lat	No
[69]	rad_int.i1, .i2, etc.	Lattice Radiation integrals	lat	No
[69]	rad_int1.i1, .i2, etc.	Element radiation integrals	lat	No
[69]	ref_time	Reference time	lat, beam	Yes
[69]	rel_floor.x, .y, .z, .theta	Relative global floor position	lat	No
[69]	s_position	longitudinal length constraint	lat	Yes
[70]	sigma.x, .y, .z, .px, px, .pz, .ij $1 \le i,j \le 6$, .Lxy	Bunch size	lat, beam	No
[70]	slip_factor_ptc.N $N = 0, 1, 2, \ldots$	Slip factor Taylor terms	lat	No
[70]	spin.depolarization_rate	Spin depolarization rate	lat	No
[70]	spin.polarization_rate	Spin polarization rate	lat	No
[70]	spin.polarization_limit	Spin polarization limit	lat	No
[70]	spin.x, .y, .z .amp	Particle spin	lat, beam	No
[70]	spin_dn_dpz.x, .y, .z	Spin dn/dpz components	lat	No
[71]	spin_g_matrix.ij $1 \le i \le 2$, $1 \le j \le 6$	Spin G-matrix components	lat	No
[71]	spin_res.a.sum, .a.diff, .b.sum, .b.diff, .c.sum, .c.diff	Spin resonance strengths	lat	No
[71]	spin_map_ptc.$ijklmn$, i,j,k,l,m,n are digits	Spin Map Taylor terms	lat	no
[71]	spin_tune	Spin tune	lat	no
[71]	spin_tune_ptc.N, $N = 0, 1, \ldots$	Spin Tune Taylor terms	lat	no
[72]	srdt.h<monomial>.{r,i,a}	Normal form driving terms calculated by summation	lat	No
[72]	time	Particle time (sec)	lat, beam	Yes
[72]	t.$ijklm\ldots$, tt.$ijklm\ldots$, $1 \le i,j,k,\ldots \le 6$	Term in nth order transfer map	lat	No
[72]	tune.a, .b, .z	Tune	lat	No
[73]	unstable.eigen, .eigen.a, .eigen.b, .eigen.c	Maximum eigenvalue amplitude	lat	No
[73]	unstable.lattice	Positive if lattice is unstable	lat	No
[73]	unstable.orbit	Nonzero if particles are lost in tracking	lat	No
[74]	velocity	Normalized velocity v/c	lat, beam	Yes

Table 6.2: (continued)

Pg#	*Data_ Type*	*Description*	*data_ source*	*Can use s_ offset?*
[74]	`velocity.x, .y, .z`	`Normalized velocity component`	`lat, beam`	`Yes`
[74]	`wall.left_side, .right_side`	`Building wall constraint`	`lat`	`No`
[75]	`wire.<angle>`	`Wire scanner at <angle>`	`beam`	`No`

6.9 Tao Data Types

The `data_type` component of datum specifies what type of data the datum represents. For example, a datum with a `data_type` of `orbit.x` represents the horizontal orbit. Table 6.2 lists what data types *Tao* knows about.

It is important to note the difference between the `d2.d1` name that is used to refer to a datum and the actual type of data, given by `data_type`, of the datum. The `d2.d1` name is arbitrary and is specified in the *Tao* initialization file (§10.10). Often, these names do reflect the actual type of data. However, there is no mandated relationship between the two. For example, it is perfectly possible to set create a data set with a `d2.d1` name of `orbit.x` to hold, say, global floor position data. In fact, the datums in a given `d1` array do not all have to be of the same type. Thus the user is free to group data as wanted.

Description of the data types:

alpha.a, .b
Twiss function `alpha`.

apparent_emit.x, .y
The apparent emittance is the emittance that one would calculate based upon a measurement of the beam size[Fra11]. It can be useful to compare this to the true normal mode emittance. Also See the `norm_apparent_emit`, `emit.` and `norm_emit.` data types. With `data_source` set to `"beam"`, `apparent_emit.x` is

$$\text{emit}_x = \frac{\sigma_{xx} - \eta_x^2\, \sigma_{p_z p_z}}{\beta_a} \tag{6.1}$$

with a similar equation for `apparent_emit.y`. Here σ is the beam size matrix

$$\sigma_{r_1 r_2} \equiv \langle r_1\, r_2 \rangle \tag{6.2}$$

With `data_source` set to `"lat"`, The apparent emittance is calculated from the true normal mode emittance and the Twiss parameters (Cf. Eqs (4) and (5) of [Fra11]).

beta.a, .b, .c
Lattice normal mode betas.

beta.x, .y, .z
Beam projected beta functions. `beta.x` is defined by

$$\beta.x = \frac{< x^2 >}{\sqrt{< x^2 >< x'^2 > - < xx' >^2}}. \tag{6.3}$$

with similar equations for the other planes. The average <> is over all the particles in the beam.

Note: If the beta function is calculated from the beam distribution, the initial beam emittance must be set to something non-zero.

bpm_cbar.22a, .12a, .11b, .12b
The normalized Cbar coupling parameters. The computed `model` values include detector misalignments, rotations, gain errors, etc. This type of datum is useful for simulating how well actual coupling corrections are. See the *Bmad* manual on "Instrumental Measurement Attributes" for more details. Note: This type of datum can only be used with `detector`, `instrument`, `monitor` or `marker` elements

bpm_eta.x, y
The horizontal and vertical dispersion components. The computed `model` values include detector misalignments, rotations, gain errors, etc. This type of datum is useful for simulating how well actual dispersion corrections are. See the *Bmad* manual on "Instrumental Measurement Attributes" for more details. Note: This type of datum can only be used with `detector`, `instrument`, `monitor` or `marker` elements

bpm_orbit.x, y
Beam Orbit. The computed `model` values include detector misalignments, rotations, gain errors, etc. This type of datum is useful for simulating how well actual orbit corrections are. See the *Bmad* manual on "Instrumental Measurement Attributes" for more details. Note: This type of datum can only be used with `detector`, `instrument`, `monitor` or `marker` elements

bpm_phase.a, b
Betatron phase. The computed `model` values include detector misalignments, rotations, gain errors, etc. This type of datum is useful for simulating how well actual orbit corrections are. See the *Bmad* manual on "Instrumental Measurement Attributes" for more details. Note: This type of datum can only be used with `detector`, `instrument`, `monitor` or `marker` elements

bpm_k.22a, .12a, .11b, .12b
Measured beam coupling components. The computed `model` values include detector misalignments, rotations, gain errors, etc. This type of datum is useful for simulating how well actual coupling corrections are. See the *Bmad* manual on "Instrumental Measurement Attributes" for more details. Note: This type of datum can only be used with `detector`, `instrument`, `monitor` or `marker` elements

bunch_charge.live, .live_relative
The charge of the live particles in a bunch as expressed unnormalized or normalized by the total charge.

bunch_max.x, .px, .y, .py, .z, .pz
Maximum phase space coordinate in a bunch, relative to its centroid.

bunch_min.x, .px, .y, .py, .z, .pz
Minimum phase space coordinate in a bunch, relative to its centroid.

cmat.11, .12, .21, .22
Coupling matrix components. The 2x2 C matrix describe the x-y coupling of the beam. See the *Bmad* manual for more details.

cbar.11, .12, .21, .22
Normalized coupling matrix components. The 2x2 C matrix describe the x-y coupling of the beam. The normalized matrix is normalized by factors of β. See the *Bmad* manual for more details.

chrom.a, .b
Chromaticities. Also see `chrom_ptc`. The calculation uses finite differences with the variation of pz being set by `global%delta_e_chrom`.

Chromaticities will be calculated even if the geometry of the lattice branch has an open geometry. In this case, dbeta/dpz and dalpha/dpz at the beginning of the branch can be set in the lattice file by setting `beginning[dbeta_dpz_a]`, `beginning[dbeta_dpz_b]`, `beginning[dalpha_dpz_a]` and `beginning[dalpha_dpz_b]`.

chrom.dbeta.a, .dbeta.b

The normalized change of the beta function with energy $(1/\beta_{a,b})\partial\beta_{a,b}/\partial p_z$. Unlike the standard chromaticities,`chrom.a` and `chrom.b`, the these chromaticities are evaluated at individual elements. The calculation uses finite differences with the variation of pz being set by `global%delta_e_chrom`.

Chromaticities and the W-function will be calculated even if the geometry of the lattice branch has an open geometry. In this case, dbeta/dpz and dalpha/dpz at the beginning of the branch can be set in the lattice file by setting `beginning[dbeta_dpz_a]`, `beginning[dbeta_dpz_b]`, `beginning[dalpha_dpz_a]` and `beginning[dalpha_dpz_b]`.

chrom.deta.x, .deta.y

The chromatic dispersion $\partial\eta_{x,y}/\partial p_z$. This is the same as *Bmad* `deta_dpz_x` and `deta_dpz_y`. If the geometry of the lattice branch has an open geometry, the values of these parameter at the start of the branch by setting `beginning[deta_dpz_x]` and `beginning[deta_dpz_y]`. The calculation uses finite differences with the variation of pz being set by `global%delta_e_chrom`.

Chromaticities will be calculated even if the geometry of the lattice branch has an open geometry.

chrom.detap.x, .detap.y

The chromatic momentum dispersion derivatives $\partial\eta'_{x,y}/\partial p_z$. This is the same as *Bmad* `detap_dpz_x` and `detap_dpz_y`. If the geometry of the lattice branch has an open geometry, the values of these parameter at the start of the branch by setting `beginning[detap_dpz_x]` and `beginning[detap_dpz_y]`. The calculation uses finite differences with the variation of pz being set by `global%delta_e_chrom`.

Chromaticities will be calculated even if the geometry of the lattice branch has an open geometry.

chrom.dphi.a, .dphi.b

The chromatic betatron phase $\partial\phi_{a,b}/\partial p_z$. Unlike the standard chromaticities,`chrom.a` and `chrom.b`, these chromaticities are evaluated at individual elements. The calculation uses finite differences with the variation of pz being set by `global%delta_e_chrom`.

Chromaticities and the W-function will be calculated even if the geometry of the lattice branch has an open geometry. In this case, dbeta/dpz and dalpha/dpz at the beginning of the branch can be set in the lattice file by setting `beginning[dbeta_dpz_a]`, `beginning[dbeta_dpz_b]`, `beginning[dalpha_dpz_a]` and `beginning[dalpha_dpz_b]`.

chrom.w.a, chrom.w.b

The `chrom.w.a` and `chrom.w.b` data types are the so called chromatic amplitude `W`-functions introduced by Montague[Montague]. `chrom.w.a` is the W-function amplitude for the a-mode and `chrom.w.b` is the W-function for the b-mode. Dropping the mode subscript, the W-function amplitude is defined by

$$W = \sqrt{A^2 + B^2} \tag{6.4}$$

where

$$A = \frac{\partial\alpha}{\partial p_z} - \frac{\alpha}{\beta}\frac{\partial\beta}{\partial p_z}, \qquad B = \frac{1}{\beta}\frac{\partial\beta}{\partial p_z} \tag{6.5}$$

Chromaticities and the W-function will be calculated even if the geometry of the lattice branch has an open geometry. In this case, dbeta/dpz and dalpha/dpz at the beginning of the branch can be set in the lattice file by setting `beginning[dbeta_dpz_a]`, `beginning[dbeta_dpz_b]`, `beginning[dalpha_dpz_a]` and `beginning[dalpha_dpz_b]`.

Note: p_z is the local p_z at the evaluation point (as opposed to the p_z at the start of the lattice).

chrom_ptc.a.N, chrom_ptc.b.N, $N = 0, 1, 2, \ldots$

Terms in the Taylor expansion of the chromaticity which is a function of phase space p_z as calculated from Etienne Forest's PTC code. [See the *Bmad* manual for documentation on PTC.] $.a$ denotes the a normal mode of oscillation and $.b$ denotes the b-mode. N is an integer which gives the order of the Taylor term

$$Q(p_z) = Q_0 + Q_1\, p_z + Q_2\, p_z^2 + Q_3\, p_z^3 + \ldots \tag{6.6}$$

where $Q(p_z)$ is the tune in units of 2π. $N = 0$, that is `chrom_ptc.a.0` and `chrom_ptc.b.0` are the fractional tunes themselves in the range $[-0.5, 0.5]$, and $N = 1$ gives the zeroth order chromaticity.

Since there are differences in the tracking between PTC and *Bmad*, The chromaticities `chrom.a` and `chrom.b` which are calculated using *Bmad* will not exactly agree with the PTC values.

The chromatic Taylor series coefficients Q_N are calculated up to order $N_T - 1$ where N_T is the Taylor map order set in the lattice file by `parameter[taylor_order]` (the default is $N_T = 3$).

To save time when the calculation of chromatic terms is not needed, the `one_turn_map_calc` parameter (§10.4) for a universe can be toggled True or False as desired. The default is False.

Note: The `show chromaticity` (§11.30.5) command can be used to see the series coefficients.

curly_h.a, .b
The `curly_h` datum is the standard "curly H" function seen in the formulas for the I_{5a} and I_{5b} radiation integrals. See the *Bmad* manual section on "Synchrotron Radiation Integrals" for more details.

damp.j_a, .j_b, .j_z
Damping partition numbers.

deta_ds.a, deta_ds.b
Normal mode dispersion derivative with respect to s – $d\eta_a/ds$ and $d\eta_b/ds$. Also see `etap.a` and `etap.b`. These datums are useful when optimizing to minimize the dispersion in a region.

deta_ds.x, deta_ds.y
Horizontal and vertical dispersion derivative with respect to s – $d\eta_x/ds$ and $d\eta_y/ds$. Also see `etap.x` and `etap.y`. These datums are useful when optimizing to minimize the dispersion in a region.

dpx_dx, dpy_dy, etc.
Bunch sigma matrix ratios, <x px> / $<x^2>$ & Etc.

dynamic_aperture.N, $N = 1, 2, 3\ldots$
The value of `dynamic_aperture.`N is the "size" S_e of the maximal ellipse that will fit within the N^{th} dynamical aperture curve. The scan index starts at "1" for the first value in the `pz` array set in the `tao_dynamic_aperture` namelist (§10.12). The maximal ellipse size is determined by the minimum of the size calculated for each point of the N^{th} scan $S_e = \min(S_e(i))$ where the ellipse size $S_e(i)$ for the i^{th} point is

$$S_e(i) = \sqrt{\frac{x^2(i)}{\beta_a\, \epsilon_a} + \frac{y^2(i)}{\beta_b\, \epsilon_b}} \tag{6.7}$$

where $x(i), y(i)$ is the i^{th} scan point, β_a and β_b are the beta functions at the starting point, and ϵ_a and ϵ_b are the emittances set for the dynamic aperture calculation (§10.12). The above equation is valid when there is no horizontal-vertical coupling at the starting point. If there is coupling, the above equation is modified and $x(i)$ and $y(i)$ are replaced by the corresponding a and b normal mode phase space coordinates (See the section on "Coupling and Normal Modes" in the *Bmad* manual).

e_tot_ref
The reference energy of the lattice. This is the same as the `E_tot` attribute of a lattice element. For the actual particle energy, use `orbit.energy`.

element_attrib.<attrib_name>

The `element_attrib.<attrib_name>` data type is associated with the lattice element attribute named `<attrib_name>`. See the *Bmad* ([Bma06]) manual for information on attribute names. For example, to plot the dipole bend strength `g`, the following plot template (§10.13) can be used:

```
&tao_template_plot
  plot%name = 'bend_g'
  plot%n_graph = 1
  plot%x_axis_type = 'index'
/

&tao_template_graph
  graph%name = 'g'
  graph%type = 'data'
  graph_index = 1
  graph%y%label = 'g'
  curve(1)%name = 'g'
  curve(1)%data_type = 'element_attrib.g'
  curve(1)%draw_line = F
/
```

emit.a, .b, .c

True normal mode (eigen) emittances. With `data_source` set to `"beam"`, the emittance is calculated from the beam sigma matrix. With `data_source` set to `"lat"`, the normal mode emittance is calculated using the standard radiation integrals.

emit.x, .y, .z

"Projected" emittances[Fra11]. For a linear lattice, the emittance varies along the length of the line while for a circular lattice there is a single emittance number.

With `data_source` set to `"beam"`, the emittance is calculated from the beam sigma matrix. The formula for ϵ_x is

$$\epsilon_x = \sqrt{\widetilde{\sigma}_{xx}\,\widetilde{\sigma}_{p_x p_x} - \widetilde{\sigma}^2_{x p_x}} \tag{6.8}$$

With a similar equation for ϵ_y. Here $\widetilde{\sigma}$ is the energy normalized beam size:

$$\widetilde{\sigma}_{xx} = \langle x\, x\rangle - \frac{\langle x\, p_z\rangle\,\langle x\, p_z\rangle}{\langle p_z\, p_z\rangle} \tag{6.9}$$

with similar definitions for the other $\widetilde{\sigma}$ components. Note that the projected emittance is sometimes defined using x' and y' in place of p_x and p_y. However, in the vast majority of cases, this does not appreciably affect the numeric results.

See also the `norm_emit.`, `apparent_emit.`, and `norm_apparent_emit.` data types.

eta.a, .b

Normal mode dispersion.

eta.x, .y, .z

Horizontal, vertical, and longitudinal dispersion.

etap.a, .b

Normal mode momentum derivative dp_a/dp_z and dp_b/dp_z. Also see `deta_ds.a` and `deta_ds.b`.

etap.x, .y

Horizontal and vertical momentum derivative dp_x/dp_z and dp_y/dp_z. Also see `deta_ds.x` and `deta_ds.y`.

expression: <arithmetic_expression>

`<arithmetic_expression>` is an arithmetic expression (§4.4) which is evaluated to get the value of the datum. For example:

```
datum(i)%data_type = "expression: 1@ele::q10w[beta_a] - 2@ele::q10w[beta_a]"
```
With this, the value of the datum will be the difference between the a-mode beta at element `q10w` for universe 1 and universe 2. In this example, the source of both terms in the expression is explicitly given as `ele`. This is not necessary if the `datum%data_source` is set to `ele`
```
datum(i)%data_type = "expression: 1@q10w[beta_a] - 2@q10w[beta_a]"
datum(i)%data_source = "ele"
```
An expression can also be used as the `default_data_type`. In this case, the evaluation point is implicit. For example:
```
default_data_source = "data"
default_data_type = "expression: 1@beta.a - 2@beta.a"
```
which is equivalent to:
```
default_data_type = "expression: 1@data::beta.a - 2@data::beta.a"
```
Tao evaluates datums that contain expressions last after all lattice parameters and all non-expression datums have been evaluated. The evaluation for datums that contain expressions start with universe 1 and evaluates the expression containing datums of universe 1 in the order that can be seen with the `show data` command (which is the same order as the datums are defined in the data init file). After this, expression containing datums in universe 2 are evaluated, etc. It is important to keep this in mind since, if an expression containing datum references another datum that itself contains an expression, and if the referenced datum is evaluated after the the first datum, the evaluation of the first datum can be invalid.

In the above examples, the lattice elements involved were explicitly specified. To apply an expression to the lattice element associated with a datum use the syntax “`ele::#`” to represent the associated lattice element. Example:
```
default_data_type = "expression: ele::#[k1] * ele::#[l]"
datum(1:4)%ele_name = "Q01", "Q02", "Q03", "Q04"
```
In this example the values of the four datums will the integrated quadrupole strength K1*L of the associated lattice elements `Q01` for the first datum, etc.

floor.x, .y, .z, .theta, .phi, .psi
Position and orientation of the element in the global “floor” coordinate system. This is the nominal position ignoring any misalignments. That is, this is the “laboratory” coordinates that define the curvilinear reference orbit. See the *Bmad* manual for details on the global coordinate system. Also see the documentation on `floor_actual.` `rel_floor.`, and `floor_orbit.` datum types.

floor_actual.x, .y, .z, .theta, .phi, .psi
Position and orientation of the element with misalignments in the global “floor” coordinate system. That is, this is the “element body” coordinates”. See the *Bmad* manual for details on the global coordinate system. See also the documentation on `floor.` `rel_floor.`, and `floor_orbit.` datum types.

floor_orbit.x, .y, .z
Position of the orbit in the global “floor” coordinate system. See the *Bmad* manual for details on the global coordinate system. See also `floor.`.

floor_orbit.theta, .phi, .psi
Orientation of the orbit in the global “floor” coordinate system. See the *Bmad* manual for details on the global coordinate system. See also `floor.`.

gamma.a, .b
Normal mode Twiss gamma function.

k.11b, .12a, .12b, .22a
Measured beam coupling parameters. See also `bpm_k.11b, ...`.

momentum
Particle momentum amplitude.

momentum_compaction
Momentum compaction factor. Also see `r56_compaction` and `slip_factor_ptc`.

momentum_compaction_ptc.N, $N = 1, 2, 3, \ldots$
Momentum compaction factor. Also see `r56_compaction` and `slip_factor_ptc`.

The momentum compaction Taylor series is a function of phase space p_z. This Taylor series is calculated from Etienne Forest's PTC code. [See the *Bmad* manual for documentation on PTC.] N is an integer which gives the order of the Taylor term

$$\frac{\Delta L(p_z)}{L_0} = \alpha_1 \, p_z + \alpha_2 \, p_z^2 + \alpha_3 \, p_z^3 + \ldots \tag{6.10}$$

Where L is the transit distance over one turn and L_0 is reference transit distance. $N = 1$, that is `momentum_compaction_ptc.1` (α_1) is the momentum compaction at $p_z = 0$, etc.

Since there are differences in the tracking between PTC and *Bmad*, the momentum compaction as calculated with *Bmad* will not exactly agree with the PTC values.

The momentum compaction Taylor series coefficients α_N are calculated up to order N_T where N_T is the Taylor map order set in the lattice file by `parameter[taylor_order]` (the default is $N_T = 3$).

To save time when the calculation of momentum compaction terms is not needed, the `one_turn_map_calc` parameter (§10.4) for a universe can be toggled True or False as desired. The default is False.

Note: The `show chromaticity` (§11.30.5) command can be used to see the series coefficients.

n_particle_loss
If the reference element is not specified, `n_particle_loss` gives the number of particles lost at the evaluation element. If the reference element is specified, `n_particle_loss` gives the cumulative loss between the exit end of the reference element and the exit end of the evaluation element. That is, this sum does not count any losses at the reference element itself. If neither reference nor evaluation element is given then the total number of lost particles is given.

norm_apparent_emit.x, .y
Apparent emittance normalized with the standard gamma factor:

$$\text{emit}_{\text{norm}} = \beta \, \gamma \cdot \text{emit} \tag{6.11}$$

See the `apparent_emit.x, .y` data type for more details.

norm_emit.a, .b, .c
Normal mode emittance normalized with the standard gamma factor:

$$\epsilon_{norm} = \beta \, \gamma \cdot \epsilon \tag{6.12}$$

norm_emit.x, .y, .z
Projected emittance normalized with the standard gamma factor:

$$\epsilon_{norm} = \beta \, \gamma \cdot \epsilon \tag{6.13}$$

normal.h.<monomial>.{r,i,a}
Resonance driving terms à la [Bengt97]. For example: `h210000`. These are the coefficients of the complex polynomial h in Eq. (6.14). The suffix `.r`, `.i`, and `.a` specifies the real part, imaginary part, or absolute value.

The order of the term is the sum of the digits in its monomial. For example, `h210000` is 3rd order and `h201100` is 4th order. If the term order exceeds the map order, then the term will be set to zero. The map order is set in the lattice file using `parameter[taylor_order] = <order>` or in `tao.init` using `bmad_com%taylor_order = <order>`. The order set in `tao.init` overrides that in the lattice file.

To save time when the calculation of RDTs is not needed, the `one_turn_map_calc` parameter (§10.4) for a universe can be toggled True or False as desired.

Commonly optimized terms and their effect on the map are located in Tab. 6.3. These terms are typically minimized for dynamic aperture optimization.

Term	Effect
h_{110001}	horizontal chromaticity
h_{001101}	vertical chromaticity
h_{200001}	vertical sychrobetatron resonance
h_{002001}	horizontal synchrobetatron resonance
h_{100002}	second order dispersion
h_{210000}	drives Q_x
h_{300000}	drives $3Q_x$
h_{101100}	drives Q_x
h_{100200}	drives $Q_x - 2Q_y$
h_{102000}	drives $Q_x + 2Q_y$
h_{220000}	$\partial Q_x/\partial J_x$
h_{111100}	$\partial Q_{x,y}/\partial J_{y,x}$
h_{002200}	$\partial Q_y/\partial J_y$
h_{310000}	drives $2Q_x$
h_{112000}	drives $2Q_y$
h_{400000}	drives $4Q_x$
h_{200200}	drives $2Q_x - 2Q_y$
h_{201100}	drives $2Q_x$
h_{202000}	drives $2Q_x + 2Q_y$
h_{003100}	drives $2Q_y$
h_{004000}	drives $4Q_y$

Table 6.3: Driving terms and their effect on the map.

normal.<type>.i.<monomial>
Components of the normal form decomposition of the one-turn-map M for a ring. Possible settings for `type` is
```
M, A, A_inv, dhdj, ReF, or ImF
```
i is an integer between 1 and 6, and `monomial` is a six digit number that specifies a monomial. For example: `100001`.

In the symplectic case:

$$M = A \circ \exp\left(:h:\right) \circ A^{-1}, \tag{6.14}$$

where A is the nonlinear normalizing map, and h is a function of the amplitudes $J_i = (1/2)(x_i^2 + p_i^2)$ only. The amplitude dependent tune shifts are given by $\mu_i = -dh/dJ_i$, and can be accessed through `normal.dhdj`. Terms of A and A^{-1} can be accessed through `normal.A` and `normal.A_inv`. In the general case,

$$M = A_1 \circ C \circ L \circ \exp\left(F \cdot \nabla\right) I \circ C^{-1} \circ A_1^{-1}. \tag{6.15}$$

Here C is the linear map to the resonance basis: $h_\pm = x \pm ip$, L is a complex linear map, A_1 is the (real) first order normalizing map, and I is the identity map. All of the nonlinearities are therefore in the complex vector field F. The real and imaginary parts of L and F can be accessed through `normal.ReF`, `normal.ImF`, `normal.ReL`, and `normal.ImL`.

null

A `null` data type is used for data where *Tao* is not able to calculate a model value. Such data cannot be used in an optimization. For example, in a linac where the beam intensity is measured at the BPMs, *Tao* is not able to calculate current variations down the Linac. Nevertheless, it could be useful to read in the measured values and plot them.

orbit.amp_a, .amp_b

"Invariant" amplitude of the orbital motion.

orbit.norm_amp_a, .norm_amp_b

Energy normalized "invariant" amplitude of the orbital motion.

orbit.energy

The `orbit.energy` data type gives the total energy of a tracked particle (with `data_source` = `lat`) or the average energy of a beam (with `data_source` = `beam`).

Notice that this is different from the `E_tot` attribute of a lattice element which is the reference energy at that element.

orbit.kinetic

The `orbit.kinetic` data type gives the kinetic energy of a tracked particle (with `data_source` = `lat`) or the average energy of a beam (with `data_source` = `beam`).

orbit.x, .y, .z, .px, .py, .pz

Orbit position and momenta.

periodic.tt.$ijklm\ldots$ $1 \le i, j, k, \ldots \le 6$

This is like the `tt.` datum except here the terms are from the periodic Taylor map defined by

$$T_p \equiv (T_0 - I_4)^{-1} \tag{6.16}$$

Here T_p is the periodic map, T_0 is the one-turn map from some point back to that point, and I_4 is a linear map defined by the matrix

$$I_4 \equiv \begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & 0 & \\ & & & & & 0 \end{pmatrix} \tag{6.17}$$

The periodic map give information about the closed orbit, dispersion, etc. For example, the zeroth order terms are the closed orbit, the r16 term gives the horizontal dispersion, etc.

If a reference lattice element is specified, the map T_0 will be the transfer map from the reference element to the evaluation element.

Note: If the reference element is not specified, or if the reference element is the same as the evaluation element, this data type cannot be used with a linear lattice.

phase.a, .b

Betatron phase. If a `d1_data` array has a set of `phase` datums, and if the reference element is *not* specified, the average phase used for optimizations (D in Eq. (8.1)) and plotting for all the datums within a `d1_data` array are set to zero by adding a fixed constant to all the datums. This is done since, without a reference point that defines a zero phase, the overall average phase is arbitrary and so the average phase is taken in *Tao* to be zero. This can be helpful in optimizations since one does not have to worry about arbitrary offsets between the `model` and `measured` values. If the reference element is specified then there is no arbitrary constant in the evaluation.

phase_frac.a, .b
Fractional betatron phase. Also see the discussion under `phase.a, .b`.

phase_frac_diff
Fractional betatron phase difference a-mode minus b-mode with the difference in the region $-\pi < d\phi_{\text{frac}} < \pi$.

photon.intensity
Photon total intensity.

photon.intensity_x, .intensity_y
Photon intensity components in the horizontal and vertical planes.

photon.phase_x, .phase_y
Photon phases in the horizontal and vertical planes.

ping_a.amp_x, .phase_x, .amp_y, .phase_y, .amp_sin_y, .amp_cos_y, .amp_sin_rel_y, .amp_cos_rel_y

Phase and amplitude response at a BPM from turn-by-turn data acquired after the beam is pinged. Ignoring damping, the beam response will be the sum of three components, one for each beam oscillation eigenmode. `ping_a` data is for the response at the `a`-mode frequency.

At each BPM, the response will have a component in the `x` (horizontal) and `y` (vertical) planes. If there is no coupling, vertical response for the `a`-mode component is zero. The horizontal $x_a(s,n)$ and vertical $y_a(s,n)$ `a`-mode response at position s and turn n is

$$\begin{aligned} x_a(s,n) &= A_{ax}(s)\,\cos(n\,\omega_a + \phi_{ax}(s) + \phi_{a0}) \\ y_a(s,n) &= A_{ay}(s)\,\cos(n\,\omega_a + \phi_{ay}(s) + \phi_{a0}) \end{aligned} \tag{6.18}$$

where ω_a is the a-mode tune, A_{ax} and A_{ay} are the response amplitudes, ϕ_{ax} and ϕ_{ay} are the horizontal and vertical phases, and ϕ_{a0} is an overall phase dependent upon how turn $n = 0$ is defined. In terms of *Tao*'s data parameters, the correspondence is

`ping_a.amp_x`	$\longrightarrow$	A_{ax}
`ping_a.phase_x`	$\longrightarrow$	ϕ_{ax}
`ping_a.amp_y`	$\longrightarrow$	A_{ay}
`ping_a.phase_y`	$\longrightarrow$	ϕ_{ay}
`ping_a.amp_sin_y`	$\longrightarrow$	$A_{ay}\cdot\sin(\phi_{ay})$
`ping_a.amp_cos_y`	$\longrightarrow$	$A_{ay}\cdot\cos(\phi_{ay})$
`ping_a.amp_sin_rel_y`	$\longrightarrow$	$A_{ay}\cdot\sin(\phi_{ay}-\phi_{ax})$
`ping_a.amp_cos_rel_y`	$\longrightarrow$	$A_{ay}\cdot\cos(\phi_{ay}-\phi_{ax})$

In terms of how *Tao* analyses ping data, only differences in phases are important so ϕ_{a0} is ignorable.

The response can be related to the lattice Twiss parameters as given by Eq. (54) of reference [Sag99]

$$\begin{aligned} x_a(s,n) &= A_a(s)\,\sqrt{\beta_a(s)}\,\cos(\theta_{ax}(s,n)), \\ y_a(s,n) &= -A_a(s)\,\sqrt{\beta_b(s)}\Big(\overline{C}_{22}\,\cos(\theta_{ax}(s,n)) + \overline{C}_{12}\,\sin(\theta_{ax}(s,n))\Big) \end{aligned} \tag{6.19}$$

where

$$\theta_{ax}(s,n) = n\,\omega_a + \phi_{ax}(s) + \phi_{a0} \tag{6.20}$$

Roughly, if the coupling is not large, the “in-plane” `x` oscillation is insensitive to any coupling so that `ping_a.amp_x` and `ping_a.phase_x` can be directly related to the Twiss parameters computed without coupling. On the other hand, the “out-of-plane” `y` oscillation is a direct measure of the coupling. This can be used to measure and correct skew-quadrupole errors. For coupling data, whether to use `ping_a.amp_sin_y` and `ping_a.amp_cos_y` or `ping_a.amp_sin_rel_y` and `ping_a.amp_cos_rel_y` depends upon how the data is obtained. If the BPMs in the machine can only measure the orbit in one plane then `ping_a.amp_sin_y` and `ping_a.amp_cos_y` is probably the better choice. One the other hand, if the BPMs can measure in both planes, `ping_a.amp_sin_rel_y` may give cleaner data due to the fact that, since the `ping_a.amp_sin_rel_y` signal is out-of-phase with the main horizontal signal, the `ping_a.amp_sin_rel_y` data is insensitive to BPM tilts and cross-talk between the horizontal and vertical signals. In fact, for the CESR ring at Cornell University, with BPMs that can measure in both planes, best coupling correction results are obtained by using the `ping_a.amp_sin_rel_y` data and ignoring the `ping_a.amp_cos_rel_y` data.

The `ping_a.amp_y` and `ping_a.phase_y` data types are not useful for data analysis when the coupling is small since, in the limit of no coupling, `ping_a.phase_y` meaningless.

For the `design` and `model` values of a datum, Eq. (6.19) is used with A_a taken to be unity. To be able to compare the `design` and/or `model` values with the actual data stored in `meas` and/or `ref`, the `meas` values will be multiplied by a constant C_m computed so that the average `meas` value is equal to the average `model` value:

$$C_m \sum \text{ping\_a.amp\_x}_{\text{meas}} = \sum \text{ping\_a.amp\_x}_{\text{model}} \tag{6.21}$$

where the sums are over all `ping_a.amp_x` data points where the `exists`, `good_model`, `good_user`, and `good_meas` components (§6.2) are all true. The `ping_a.amp_y` data points are not used for the computation of C_m since, with a decoupled lattice, the `model` values are zero.

There is a similar multiplier defined for the reference data. The values of these two multipliers are shown with the `show data` command.

ping_b.amp_y, .phase_y, .amp_x, .phase_x, .amp_sin_x, .amp_cos_x, .amp_sin_rel_x, .amp_cos_rel_x

Similar to `ping_a` except this is for the `b`-mode component of the response which. The response is (see Eq. (6.18))

$$\begin{aligned} x_b(s,n) &= A_{bx}(s)\,\cos(n\,\omega_b + \phi_{bx}(s) + \phi_{b0}) \\ y_b(s,n) &= A_{by}(s)\,\cos(n\,\omega_b + \phi_{by}(s) + \phi_{b0}) \end{aligned} \tag{6.22}$$

where ω_b is the a-mode tune, A_{bx} and A_{by} are the response amplitudes, ϕ_{bx} and ϕ_{by} are the horizontal and vertical phases, and ϕ_{b0} is an overall phase dependent upon how turn $n = 0$ is defined. In terms of *Tao*'s data parameters, the correspondence is

`ping_b.amp_x`	$\longrightarrow$	A_{bx}
`ping_b.phase_x`	$\longrightarrow$	ϕ_{bx}
`ping_b.amp_y`	$\longrightarrow$	A_{by}
`ping_b.phase_y`	$\longrightarrow$	ϕ_{by}
`ping_b.amp_sin_x`	$\longrightarrow$	$A_{bx} \cdot \sin(\phi_{bx})$
`ping_b.amp_cos_x`	$\longrightarrow$	$A_{bx} \cdot \cos(\phi_{bx})$
`ping_b.amp_sin_rel_x`	$\longrightarrow$	$A_{bx} \cdot \sin(\phi_{bx} - \phi_{by})$
`ping_b.amp_cos_rel_x`	$\longrightarrow$	$A_{bx} \cdot \cos(\phi_{bx} - \phi_{by})$

Here the `design` and `model` values are calculated from Eq. (8) of reference [Sag00a]:

$$\begin{aligned} x_b(n) &= A_b\,\sqrt{\beta_a}\Big(\overline{C}_{11}\,\cos(n\omega_b) - \overline{C}_{12}\,\sin(n\omega_b)\Big), \\ y_b(n) &= A_b\,\sqrt{\beta_b}\,\cos(n\omega_b). \end{aligned} \tag{6.23}$$

with `A_b` taken to be unity for the evaluation.

The corresponding multiplicative values are derived from `ping_b.amp_y` in an analogous fashion to the multiplicative values for the a-mode ping data.

r.ij $1 \le i,j \le 6$
Terms of the linear transfer matrix from the reference point to the evaluation point. If the reference element is not specified, the reference point is the beginning of the lattice. When the reference point is the evaluation point, the transfer matrix is the unit matrix.

r56_compaction
This datum is defined to be

$$M_{5,6} + \sum_{i=1}^{4} M_{5,i}\,\eta_i \tag{6.24}$$

where $\mathbf{M}$ is the transfer matrix between the reference element and the element where the datum is evaluated and η is the dispersion vector evaluated at the reference element.

This datum is closely related to the momentum compaction. When `r56_compaction` is evaluated from the start of the lattice to the end, the value of `r56_compaction` will be related to the momentum compaction via:

```
  r56_compaction = -momentum_compaction * L
```

where `L` is the length of the lattice.

rad_int.i0, .i1, .i2, .i2_e4, .i3, .i3_e7, .i4a, .i4b, .i4z, .i5a, .i5b, .i5a_e6, .i5b_e6
Synchrotron radiation integrals over all elements in a lattice branch from `ele_ref` to `ele`. If `ele` is not specified, the radiation integral is over the entire associated lattice branch. See the *Bmad* manual for more details. Also see the `rad_int1` datum type.

```
  .i0         ! I0 radiation integral
  .i1         ! I1 radiation integral
  .i2         ! I2 radiation integral
  .i2_e4      ! Energy normalized I2 radiation integral
  .i3         ! I3 radiation integral
  .i3_e7      ! I3 radiation integral
  .i4a        ! a-mode I4 radiation integral
  .i4b        ! b-mode I4 radiation integral
  .i4z        ! Sum of I4a, and I4b radiation integrals
  .i5a        ! a-mode I5 radiation integral
  .i5b        ! b-mode I5 radiation integral
  .i5a_e6     ! Energy normalized I5a
  .i5b_e6     ! Energy normalized I5b
```

rad_int1.i0, .i1, .i2, .i2_e4, .i3, .i3_e7, .i4a, .i4b, .i4z, .i5a, .i5b, .i5a_e6, .i5b_e6
Similar to the `rad_int` datum type but here the radiation integrals are just over a single element given by `ele`. If `ele_ref` is given, the value of the datum is the difference between the integral over `ele` and over `ele_ref`.

ref_time
This is the time the reference particle passes the exit end of the element. If the particle is ultra-relativistic then this is just $c * s$ where s is the longitudinal distance from the start of the lattice.

rel_floor.x, .y, .z, .theta
This is the global floor position at the exit end of the evaluation element relative to the exit end of the reference element in a global coordinate system where the exit end of the reference element is taken to be at `x = y = z = theta = phi = 0`. See the *Bmad* manual for details on the global coordinate system. See also `floor.` and `wall.`.

s_position
The longitudinal position of `ele`. This type of datum is potentially useful when lattice elements are shifting in space. Also see `wall.left_side` and `wall.right_side`.

sigma.x, .y, .z, .px, .py, .pz, .Lxy, .*ij* $1 \le i, j \le 6$

The 6×6 sigma matrix `sigma.`ij with $1 \le i, j \le 6$ describes the beam size in phase space. That is

$$\text{sigma}.ij = \big\langle (r_i - \overline{r}_i)\,(r_j - \overline{r}_j) \big\rangle \tag{6.25}$$

where $< \ldots >$ is an average of the particle's phase space vector $\mathbf{r} = (x, p_x, y, p_y, z, p_z)$ with $\bar{\mathbf{r}}$ being the average position.

The datums `sigma.x`, `sigma.px`, etc., are just a shorthand notation for `sigma.11`, `sigma.22`, etc., and the angular momentum `sigma.Lxy` is shorthand for $< x\,p_y - y\,p_x >$

The sigma matrix is calculated from beam tracking if the `data_source` is set to `beam` and calculated from the Twiss parameters if the `data_source` is set to `lat` (§10.8).

Irregardless of the setting of `data_source`, the beam emittance and longitudinal sigma values will be taken from the `beam_init` structure (§10.7) and *not* any emittances or longitudinal sigma values specified in the lattice file.

slip_factor_ptc.*N*, $N = 1, 2, 3, \ldots$

Phase slip factor. Also see `r56_compaction` and `momentum_compaction`.

Phase slip factor Taylor series is a function of phase space p_z. The Taylor series is calculated from Etienne Forest's PTC code. [See the *Bmad* manual for documentation on PTC.] N is an integer which gives the order of the Taylor term

$$\frac{\Delta T(p_z)}{T_0} = \alpha_1\,p_z + \alpha_2\,p_z^2 + \alpha_3\,p_z^3 + \ldots \tag{6.26}$$

Where T is the transit time over one turn and T_0 is reference transit time. $N = 1$, that is `slip_factor_ptc.1` (α_1) is the slip factor at $p_z = 0$, etc.

Since there are differences in the tracking between PTC and *Bmad*, The slip factor as calculated with *Bmad* will not exactly agree with the PTC values.

The slip factor Taylor series coefficients α_N are calculated up to order N_T where N_T is the Taylor map order set in the lattice file by `parameter[taylor_order]` (the default is $N_T = 3$).

To save time when the calculation of momentum compaction terms is not needed, the `one_turn_map_calc` parameter (§10.4) for a universe can be toggled True or False as desired. The default is False.

Note: The `show chromaticity` (§11.30.5) command can be used to see the series coefficients.

spin.polarization_limit, .polarization_rate, .depolarization_rate

The spin depolarization limit, P_{dk}, is calculated from the Derbenev-Kondratenko-Mane formula as discussed in the `Spin Dynamics` chapter of the *Bmad* manual. The `Baier-Katkov-Strakhovenko` polarization rate, τ_{bks}^{-1}, and the depolarization rate, τ_{dep}^{-1}, are also discussed in the this chapter. Note: τ_{bks}^{-1} is generally a good approximation to the `Sokolov-Ternov` polarization rate τ_{st}^{-1}.

spin.x, .y, .z, .amp

The `spin.x`, `spin.y` and `spin.z` datums are the spin polarization (x, y, z) components and the `spin.amp` datum is the amplitude of the spin. For a beam, this is the spin averaged over the beam. For particles with a lattice with an open geometry, the spin is calculated by propagating the spin from the beginning of the lattice. The beginning spin is set in the lattice file by setting `beginning[spin_x]`, `beginning[spin_y]` and `beginning[spin_z]` as explained in the *Bmad* manual. For a lattice with a closed geometry, the calculated spin is the closed orbit invariant spin with the amplitude of the spin set at unity.

spin_dn_dpz.x, .y, .z

This datum gives the x, y, and z components of the $d\mathbf{n}/dp_z$ vector where $\mathbf{n}$ if the invarient spin field and p_z is the sixth component of the particle's phase space coordinates.

spin_g_matrix.*ij* $1 \le i \le 2, \quad 1 \le j \le 6$

The **G**-matrix is the 2×6 sub-matrix of the 8×8 spin-orbital transfer matrix $\widetilde{\mathbf{M}}$ that describes the coupling between orbital motion and spin precession as discussed in the section on the SLIM Formalism in the chapter on spin in the *Bmad* manual. Lattice design generally involves minimizing components of this matrix. The transfer matrix of which the G-matrix is a component of is computed from the datum's `ref_ele` reference element to the `ele` evaluation element. If the associated lattice branch has a closed geometry, and if the reference element is not specified, the reference element will be taken to be evaluation element and the 1-turn spin/orbital transfer matrix will be computed.

When using this data type, if the `spin_axis` component is not set, *Tao* will try to set appropriate values. If `spin_axis%n0` is not set, and if the lattice branch has a closed geometry, the value of this parameter will be set to the value of the closed orbit n_0 at the reference element. If `spin_axis%n0` is not set, and if the lattice branch has an open geometry, the value of this parameter will be set to be the same as the direction of the tracked particle's spin.[1] If neither `spin_axis%l` nor `spin_axis%m` is set, these axes are chosen such that $(\mathbf{l}, \mathbf{n}_0, \mathbf{m})$ form a right handed coordinate system.

Once the $(\mathbf{l}, \mathbf{n}_0, \mathbf{m})$ coordinate system has been set at the reference element, the $(\mathbf{l}, \mathbf{n}_0, \mathbf{m})$ coordinates at the evaluation element are determined such that a particle on the reference orbit with spin pointing along any of the three axes at the reference element will, after propagation to the evaluation element, maintain its spin along the axis. With this, the 2×2 submatrix $\mathbf{D}$ of the spin-orbital transfer matrix $\widetilde{\mathbf{M}}$ will be the unit matrix. An exception is made if the reference element is the same as the evaluation element. That is, if the 1-turn **G**-matrix is being computed. In this case, the final $(\mathbf{l}, \mathbf{n}_0, \mathbf{m})$ will be taken to be the same as the initial coordinates and, in this case, $\mathbf{D}$, will be a rotation matrix with a rotation angle equal to the spin tune.

spin_res.a.sum, .a.diff, .b.sum, .b.diff, .c.sum, .c.diff

Spin resonance values. `.a.sum`, `.a.diff` are the sum and difference resonance values for the a-mode, etc. The `show spin -ele ...` command will print the values. If not present, the evaluation element for the datum is the beginning element of the lattice branch.

spin_tune

Spin tune as calculated by Bmad and not using PTC code. Also see `spin_tune_ptc`.

spin_map_ptc.*ijklmn*, where i, j, k, l, m, n are 6 digits

Terms in the taylor map of the spin tune. See also `spin_tune_ptc.`N.

spin_tune_ptc.*N*, $N = 0, 1, 2, \ldots$

Terms in the taylor series of the spin tune expanded as a function of p_z as calculated from Etienne Forest's PTC code. [See the *Bmad* manual for documentation on PTC.] N is an integer which gives the order of the Taylor term

$$Q_s(p_z) = Q_{s0} + Q_{s1}\, p_z + Q_{s2}\, p_z^2 + Q_{s3}\, p_z^3 + \ldots \tag{6.27}$$

where $Q_s(p_z)$ is the spin tune in units of 2π.

The spin tune Taylor series coefficients Q_N are calculated up to order N_T where N_T is the Taylor map order set in the lattice file by `parameter[taylor_order]` (the default is $N_T = 3$).

To save time when the calculation of chromatic terms is not needed, the `one_turn_map_calc` parameter (§10.4) for a universe can be toggled True or False as desired. The default is False.

Note: `spin_tune_ptc.`N is the same as `spin_map_ptc.00000`N.

Note: The `show chromaticity` (§11.30.5) command can be used to see the series coefficients.

[1] The initial spin for particle tracking in an open geometry can be set either by setting in the lattice file parameters like `particle_start[spin_x]` or, when *Tao* is running, by using the `set particle_start` command.

srdt.h<monomial>.{r,i,a}

Resonance driving term summations based on summation formulas by Bengtsson[Bengt97] for 3rd order terms and Wang[Wang12] for 4th order terms. This is independent from the PTC derived `nomial.` data types.

The 3rd order monomials for which summations have been implemented are

```
h21000, h30000, h10110, h10020, h10200,
h11001, h20001, h00111, h00201,
h10002
```

The 4th order monomials are

```
h31000, h40000, h22000,
h00220, h00310, h00400,
h11110, h20110, h11200, h20020, h20200
```

Suffixes `.r`, `.i`, and `.a` signify the real part, imaginary part, or absolute value.

For higher orders and monomials for which summations are not available, see the `normal.h.<monomial>.{r,i,a}` data type. Additional details on driving terms are listed there and in Tab. 6.3

Three `tao.init` `tao_params` are important for `srdt` data. They are,

```
global%srdt_use_cache = {.true. (default), .false.}
global%srdt_sxt_n_slices= <integer, default 20>
global%srdt_gen_n_slices= <integer, default 10>
```

`srdt_sxt_n_slices` and `srdt_gen_n_slices` set the number of steps to take through sextupole and non-sextupole elements, respectively. More steps improve accuracy, but are slower and increase the size of the srdt cache.

`srdt_use_cache` generates a cache of the cross-products of the linear optics quantities used for the srdt summations. Provided the linear optics are not changing, using the cache can greatly speed up subsequent srdt calculations (i.e. during optimization of sextupole moments). Note that whenever the linear optics change, e.g. a quadrupole is adjusted, the cache must be regenerated. Also note that the cache can be rather large and grows as with the squares of `srdt_sxt_n_slices` and `srdt_gen_n_slices`. If there is insufficient RAM available, `tao` will generate a warning message and revert to a `srdt_use_cache=.false.` state.

t.$ijklm\ldots$, tt.$ijklm\ldots$ $1 \le i, j, k, \ldots \le 6$

Taylor map components between two points. `tt.`$ijk\ldots$ is analogous to the `tt` syntax for `taylor` elements (see the *Bmad* manual description of `taylor` elements). For example, `tt.34` corresponds to the linear M_{34} matrix element. The `t.`$ijklm\ldots$ notation, which is superfluous but is keep for backwards compatibility, is equivalent to `tt.`$ijklm\ldots$ Also see `r.`ij.

Calculation of `t.`$ijklm\ldots$ and `tt.`$ijklm\ldots$ datums involve symplectic integration through lattice elements. One point to be kept in mind is that results will be dependent upon the integration step size through an element set by the `ds_save` attribute of that element (see the *Bmad* manual for more details). When a smooth curve (§10.13.2) is plotted for `t.`ijk and `tt.`$ijklm\ldots$ data types, and the longitudinal (`"s"`) position is used for the x-axis, the integration step used in generating the points that define this curve will be decreased if the s-distance between points is smaller than the `ds_save`. In this case, discrepancies between the plot and datum values may be observed.

time

Time (in seconds) a particle or the bunch centroid is at the evaluation element.

tune.a, .b, .z

Tune in radians.

unstable.eigen, unstable.eigen.a, .eigen.b, .eigen.c

The `unstable.eigen` data type give the maximum eigenvalue amplitude of the three normal modes of oscillation. This calculation uses the transfer matrix calculated from the beginning of the specified lattice branch to the end of the branch. The geometry of the lattice branch does not have to be set to closed. If the transfer matrix represents stable motion, the value of `unstable.eigen` will be one. Also see `unstable.orbit` and `unstable.lattice`.

The `unstable.eigen.a`, `unstable.eigen.b`, and `unstable.eigen.c` are like `unstable.eigen` except that these data types are the maximum of the two eigenvalue amplitudes for the `a`, `b`, and `c` modes respectively.

unstable.lattice

`unstable.lattice` is used in an optimization to avoid unstable solutions (§8.1). Also see `unstable.orbit` and `unstable.eigen`. The value of `unstable.lattice` will always be zero or positive.

For lattices with a `closed` geometry such as a storage ring: The value `unstable.lattice` is zero if the ring is stable. If the closed orbit cannot be found, the value of `unstable.lattice` is set 1. And if the ring is unstable, the value is set to the largest growth rate of the transverse (but not longitudinal) normal modes of oscillation.

At the borderline between stability and instability the growth rate is zero. This means that an optimizer may find an optimum that is close to the borderline and unstable. To prevent this, the value of `global%unstable_penalty`, which has a default value of 0.001, is added to the computed growth rate when the lattice is unstable.

For lattices with an `open` geometry: The value of `unstable.lattice` will be non-zero if there is a problem computing the reference energy. This can happen, for example, if the accelerating voltage of an `lcavity` is set large and negative such that the reference particle is cannot make it to the exit end.

unstable.orbit

The `unstable.orbit` datum is used in an optimization to avoid losing particles during tracking. The idea is that `unstable.orbit` will be zero if there is no loss and be positive if there is. Including `unstable.orbit` in the merit function will thus tend to stabilize the lattice. Also see `unstable.eigen` and `unstable.lattice`.

For a lattice branch with an open geometry, and with single particle tracking (datum's `data_source` set to "lat"), if the tracked particle survives (has not been lost) up to the evaluation element, the value of `unstable.orbit` is zero. If it has been lost, the value is set to

$$1 + i_{\text{ele}} - i_{\text{lost}} + \frac{1}{2}\left[\tanh\left(\frac{r_{orbit}}{r_{lim}} - 1\right) - E\right] \tag{6.28}$$

where i_{ele} is the index of the evaluation element in the lattice and i_{lost} is the index of the element where the particle was lost. In the above equation, E is the function

$$E = \begin{cases} 1 & \text{if the particle is lost at the exit end of the element.} \\ 0 & \text{if the particle is lost at the entrance end of the element.} \end{cases} \tag{6.29}$$

In the above equation, r_{orbit} is the particle amplitude at the point of loss and r_{lim} is the aperture limit. The form of the above equation has been chosen so that the datum value will be monotonic with increasing stability.

If `ele_name` is not specified, the default for the evaluation element is to use the last element in the lattice branch. *This is the recommended way to use this datum.*

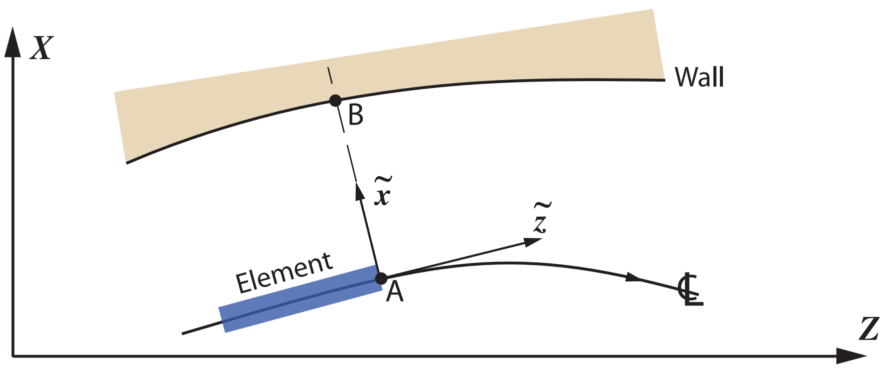


Figure 6.2: A `wall.` datum is a measure of the distance between the centerline of a machine and the walls of the containment building.

With a closed geometry lattice and with single particle tracking, `unstable.orbit` is set to zero if the orbit is stable and set to one if the orbit is unstable.

When tracking beams (datum's `data_source` set to "beam"), a beam is tracked a maximum of one turn in a branch. This being the case, the geometry of the branch is not pertinent and the value of `unstable.orbit` is value as shown above for a single particle in an open branch but here averaged over all particles in the bunch. In this case, if `ele_name` is not specified, it is taken to be the bunch tracking end point (which may not be the end of the branch).

One potential hindrance to optimization occurs if no apertures are set and the default aperture set by `bmad_com%max_aperture_limit` of 100 meters is used. In this case, the orbit can become so convoluted with the orbit oscillating with large amplitude, that it is hard for the optimizer to find a stable solution. The fix is either to set individual apertures or `bmad_com%max_aperture_limit` to prevent the large orbit oscillations. Apertures of 0.1 meter or less are generally advised.

velocity, velocity.x, .y, .z

The velocity normalized by the speed of light c.

wall.left_side, .right_side

The `wall` data data type is used to constrain the shape of a machine to fit inside a building's walls (§10.11). The general layout is shown in Figure 6.2. The machine centerline is projected onto the horizontal (Z, X) plane in the Global (floor) coordinate system. Point `A` is an evaluation point at the exit end of some element. $\tilde{z}$ is the projection of the local z-axis onto the (Z, X) plane and $\tilde{x}$ is the coordinate in the (Z, X) plane perpendicular to $\tilde{z}$. In the typical situation, where a machine is planer (no out-of-plane bends), the $\tilde{z}$-axis corresponds to the local laboratory z-axis and the $\tilde{x}$-axis corresponds to the local laboratory x-axis (see the *Bmad* manual for an explanation of local and global coordinate systems).

The distance from the machine at point `A` to the wall is defined to be the distance from `A` to a point `B` on the wall where point `B` is along the $\tilde{x}$ axis (has $\tilde{z} = 0$) as shown in Figure 6.2.

By definition, the **`"left side"`** of the machine corresponds to be the $+\tilde{x}$ side and the **`"right side"`** corresponds to be the $-\tilde{x}$ side. That is, left and right are relative to someone looking in the same direction as the beam is propagating. Correspondingly, there are two wall data types: `wall.left_side` and `wall.right_side`. With the `wall.left_side` data type, the datum value is positive if point `B` is on the left side and negative if on the right. Vice versa for a `wall.right_side` datum. That is, there is interference with with wall when the datum value is negative. If there are multiple wall points `B`, that is, if there are multiple points on the wall with $\tilde{z} = 0$, the datum value will be the minimum value. Notice that only wall sections that have a `constraint` matching the datum will be used when searching for possible points `B`. If there are no wall points with $\widetilde{z} = 0$, the datum will be marked invalid.

For `wall` data there can be no reference element since this does not make sense.

Note: To constrain the machine vertically, use the `floor.y` datum type. To constrain the length of the machine, use the `s_position` datum type.

wire.<angle>
`wire` data simulates the measurement of a wire scanner. The angle specified is the angle of the wire with respect to the horizontal axis. The measurement then measures the second moment $< uu >$ along an axis which is 90 degrees off of the wire axis. For example, `wire.90` is a wire scanner oriented in the vertical direction and measures the second moment of the beam along the horizontal axis, $< xx >$. The resultant data is not the beam size, but the beam size squared.

Chapter 7

Plotting

Tao has a graphical display window within which such things as lattice functions, machine layout, beam positions, etc., can be plotted. An example is shown in Fig. 7.1 where there are plots of the beta function and orbit along with a "lattice layout which shows the longitudinal positions of lattice elements.

Tao organizes the display window using a number of concepts which are explained in the sections below

```
  plot_page      ! The display window containing the graphics (§7.1).
  regions        ! A set of rectangles on the plot_page that plots can be put in (§7.2).
  plot           ! A collection of graphs (§7.3).
  box            ! Rectangular area within a plot that a graph is placed in (§7.4).
  graph          ! A diagram of some sort (§7.5).
  curve          ! Data displayed within a graph (§7.6).
```

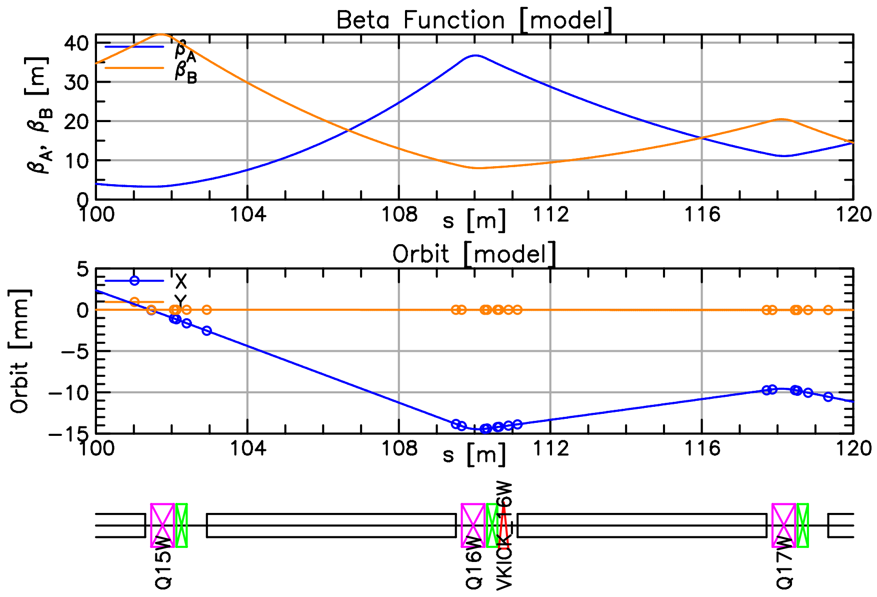


Figure 7.1: An example of a plot display. In this example there are three graphs: A graph displaying the beta function, a graph displaying the orbit, and a graph displaying the "lattice layout" which shows the longitudinal positions of lattice elements.

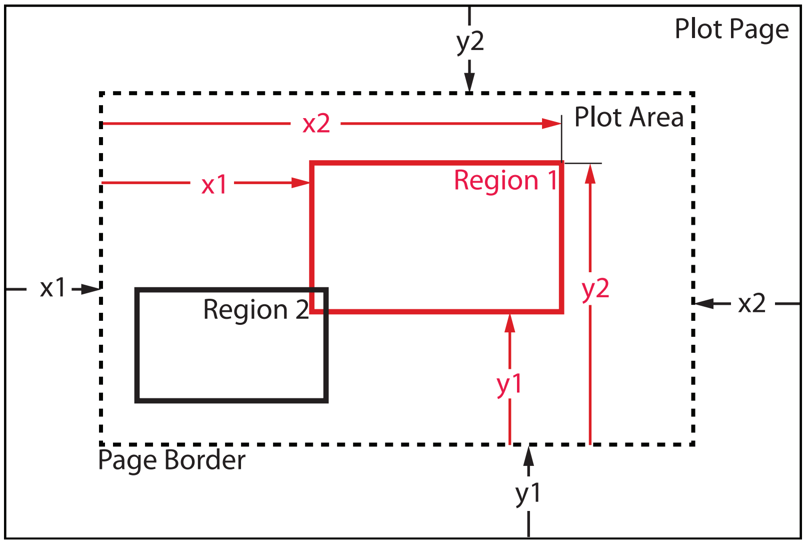


Figure 7.2: The `plot page` is the entire display window area. The `plot area` is the region within the boarders of the `plot page` within which "`regions`" are placed. The location of a `region` is defined by four offsets with respect to the `plot area`. Regions may overlap.

Underlying all this is the `quick_plot` software toolkit (§7.7) which was developed for *Bmad* and *Tao* for graphics plotting.

7.1 Plot Page

The `plot page`, sometimes called the `plot window`, refers to the window or the corresponding printed graphics page where graphics are displayed. A `plot page` is shown schematically in Fig. 7.2. Parameters associated with the `plot page` are discussed in §10.13.1. These parameters may be set in an initialization file or may be set on the *Tao* command line using the `set plot_page` (§11.29.21) command. Examples:

```
  set plot_page text_height = 11   ! 11 point font size
  set plot_page border%x1 = 0.2    ! Set left page border to 20% of width.
```

The size of the `plot page` is set by the `plot_page%size` parameter which is an array of two numbers which set the width and height. The `plot page` size can also be set when invoking *Tao* using the `-geometry` option (§10.1)

```
  > tao -lattice lat.bmad -geometry 300x500
```

This starts *Tao* with the `plot page` size set to 300 points wide by 500 points high. It is also sometimes convenient to start *Tao* without the plotting window. In this case, the `-noplot` option can be used on the startup command line. In a *Tao* initialization file, display of the plot window can be set using the `global%plot_on` parameter set in the `tao_params` namelist (§10.6).

In some cases, the screen resolution reported to *Tao* can be off. This has happened with some high resolution displays where the reported resolution is 96 dpi when in fact the actual resolution is much larger. In such a case, the size of the plot window created by *Tao* will be off. This can be corrected by setting `plot_page%size` appropriately but this in turn can create font size problems. To avoid this problem, the environmental variable `ACC_DPI_RESOLUTION` can be set to the correct resolution before running *Tao*. The shell command line would be something like

```
  > export ACC_DPI_RESOLUTION=168
```

The `plot page` has a border within which `regions` (§7.2) are defined. The area withing the plot page border is called the `plot area`

The `show plot -page` (§11.30.26) command may be used to view the page parameters.

7.2 Region

The `plot area` is the area within the border of the `plot page` as shown in Fig. 7.2. In this `plot area`, "`regions`" can be defined which are invisible rectangles where a `plot` (§7.3) can be placed. This is shown schematically in Fig. 7.2. Each region has a name and four numbers which specifies the location of the region within the plot area. Regions may be defined by the user. In addition, for convenience, *Tao* will define a number of regions. *Tao* defined regions will either begin with the letter "`r`" or begin with the string "`layout`" or the string "`scratch`". Regions may overlap. How to define regions is explained in §10.13.1. The `show plot` command will show the region list. Example:

```
  Tao> show plot

Plot Region          <-->  Plot                  x1     x2     y1     y2      Visible
-----------                 ---------------------------------------------------------
layout               <-->  lat_layout           0.00   1.00   0.00   0.15          T
r11                  <-->                       0.00   1.00   0.15   1.00
r12                  <-->                       0.00   1.00   0.58   1.00
r22                  <-->                       0.00   1.00   0.15   0.58
r13                  <-->  beta                 0.00   1.00   0.72   1.00          T
r23                  <-->  dispersion           0.00   1.00   0.43   0.72          T
r33                  <-->  orbit                0.00   1.00   0.15   0.43          T
r14                  <-->                       0.00   1.00   0.79   1.00
```

The `Plot` column shows what `plot` (if any) is associated with the region (§7.3). The next four columns show the values of `x1`, `x2`, `y1`, and `y2` set for the region. As shown in Sec. §10.13.1, `x1` and `x2` are the offsets from the left `plot area` edge to the left and right edges of the region. Similarly, `y1` and `y2` are the offsets from the bottom edge of the `plot area` to the bottom and top edges of the region. `x1` and `x2` are normalized by the `plot area` width and `y1` and `y2` are normalized by the `plot area` height so all four numbers should be in the range $[0, 1]$. Using the above example, the `r23` region spans the full width of the `plot area` (since `x1` $= 0$ and `x2` $= 1$), and occupies approximately the middle third vertically of the `plot area` (since `y1` $= 0.43$ and `y2` $= 0.72$).

The last column in the above shows if the `plot` associated with the `region` is visible. Normally everything is visible. Invisibility is used in some special cases. For example, when using a Graphical User Interface (GUI).

The `set region` command can be used to set region parameters. Example:

```
  set region r13 y1 = 0.8  ! Sets lower edge vertical position
```

7.3 Plot

A `plot` is essentially a collection of `graphs`. This is shown schematically in Fig. 7.3 which shows a plot with two graph side by side.

Plots are divided into two groups. A `template` plot defines how a `displayed` plot is to be constructed. That is, a `template` plot defines what the associated `graphs` are, defines graph placement within the plot, etc. When a `template` plot is `placed` in a `region`, either by using the `place` command (§11.19)

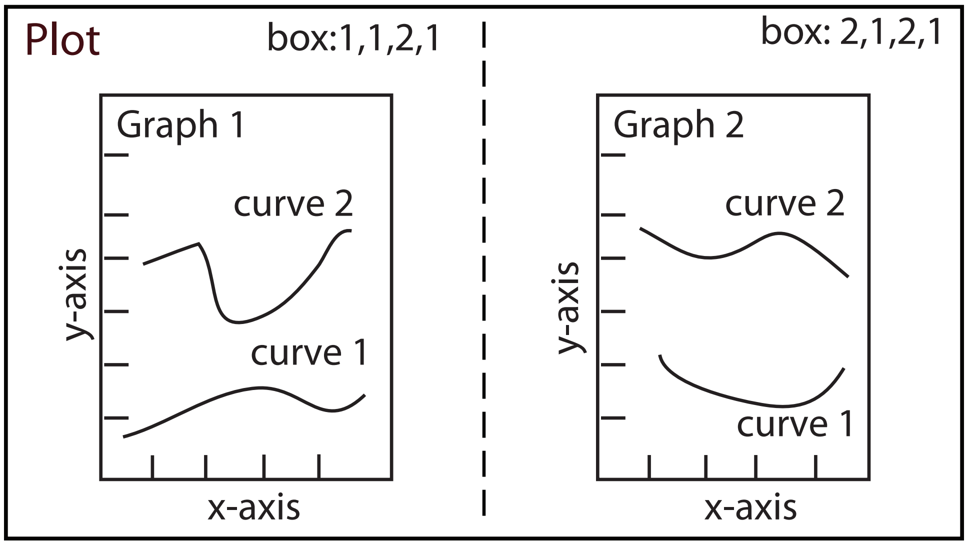


Figure 7.3: A plot has a collection of graphs and a graph has a collection of curves. A graph is located within a plot by defining the "`box`" associated with the `graph`. Illustrated here is a plot with two graphs placed side by side.

or by placement defined in an initialization file (§10.13.1), the information of the `template` is copied in order to construct a `displayed` plot. A given `template` plot may be placed in multiple `regions` to give multiple `displayed` plots and then, using `set` commands, the data displayed in each of these plots may be manipulated separately. For example, one displayed orbit plot could show the orbit of the `model` lattice while another orbit plot could show the orbit difference between the `model` and `design` lattices. When a `plot` is displayed in a given `region`, everything drawn is scaled to the region size.

Use the `show plot` to see what displayed plots are associated with what regions. Use the `show plot -templates` command to see a list of `template` plots. *Tao* defines a number of default `template` plots. Section §10.13.2 discusses how to define custom template plots in an initialization file. Use the `set plot` command (§11.29.20) to modify either `template` or `displayed` plots.

All plots have a name. A `displayed` plot will inherit the same name of the `template` plot it came from. If a given `template` plot is used to create multiple `displayed` plots. All of these plots will have the same name. A `displayed` plot can also be referred to by using the associated `region` name. This can be used to remove ambiguity if there are multiple `displayed` plots of the same name. Additionally, a `template` plot can unambiguously be referred to by adding the prefix "`T::`" to the plot name. Examples:

```
show plot           ! Show plots associated with regions
show plot -template ! Show template plots
place r13 orbit     ! Put orbit template into r13 region
```

Some commands, for example, the `scale` command by default will ignore `template` plots unless the plot name has the `T::` prefix. Other commands, for example the `show plot` command, will preferentially show displayed plot info but will show template plot info if there are no matching displayed plots. Examples:

```
scale orbit -10 10    ! Scale all displayed orbit plots. Ignore template.
scale r33 -10 10      ! Scale only plot in r33 region.
scale T::orbit -10 10 ! Scale template orbit plot.
show plot e_field     ! Will show displayed e_field plot info. If no
                      ! displayed plot exists, will show template info.
```

7.4 Box

To determine where a `graph` is drawn with respect to the boundaries of its associated `plot`, each `graph` is associated with a given "`box`". A `box` is a rectangular sub-region of the `plot`. Boxes are defined by dividing the `plot` into a rectangular grid and then choosing one of the grid rectangles to be the `box` associated with the `graph`. The is illustrated in Fig. 7.3 where `Graph 1` is associated with the `box` labeled "`1,1,2,1`" and `Graph 2` is associated with the `box` labeled `2,1,2,1`. The last two digits of a `box` label (`2,1` for both graphs) specify the number of rectangles the grid has horizontally and vertically (2 horizontally, 1 vertically here). The first two digits (`1,1` for `graph 1` and `2,1` for `graph 2`) specify the particular rectangle associated with the `box` with `1,1` designating the lower left rectangle. Different `graphs` do not have to use the same grid division to select a box from.

Setting the `box` for a given `graph` in a *Tao* initialization file is covered in §10.13.2. The `set graph` and `show graph` commands can be used to set and show the box parameters. Examples:

```
  set graph myplot.g1 box = 2 1 2 2  ! Set box of graph myplot.g1
  set graph myplot.g2 box = 1 1 1 2  ! Different graphs can use different grids
                                     !  for box selection
```

7.5 Graph

7.5.1 Overview

A `graph` is a diagram of some sort. Most `graphs` consists of horizontal and vertical axes along with one or more `curves`. `Floor_plan` (§10.13.8) and `lat_layout` (§10.13.7) `graphs`, on the other hand, shows the placement in space of the lattice elements and do not have any associated `curves`.

Every `plot` has at least one `graph`. How many `graphs` are associated with a `plot` is a matter of convenience and different `graphs` of a `plot` may display different types of information. For example, it would be possible to have a single `plot` contain three `graphs` and look like what is shown in Fig. 7.1. In actuality, the figure was constructed using three `plots` each one containing one `graph`.

How to define `graphs` when defining `template` plots is given in §10.13.2. The `show graph` command can be used to show graph parameters. The `set graph` command can be used to modify `graph` parameters.

7.5.2 Graph Name

All graphs have a name. For example, the graph of the standard `orbit` plot is simply "`g`". `Graphs` may be referred to using the syntax:

```
  <plot>.<graph>
```

where `<plot>` is the plot name (or the `region` name associated with the `plot`) and `<graph>` is the graph name. If the `.<graph>` ending is omitted, all graphs of the named `plot`(s) are selected. Examples:

```
  show graph beta   ! Show info of all graphs in all the displayed beta plots.
  show graph r13.g1 ! Show info on ‘‘g1’’ graph of region r13.
```

7.5.3 Curve Legend of a Graph

The `curve legend` is the legend identifying what curves are associated with what perimeters. In Fig. 7.1 the top two graphs have a curve legend in the upper left hand corner of the graph. By default, the `data_type` of each curve will be used as the text for that curve's line in the legend. This default can be changed by setting a curve's `curve%legend_tex`. Parameters that affect the curve legend are:

```
  plot_page%legend_text_scale        ! Also affects lat_layout and floor_plan text size.
  curve(N)%legend_text               !
  graph%curve_legend_origin
  graph%draw_curve_legend
  graph%curve_legend%line_len    ! curve_legend is a qp_legend_struct (§7.7.7).
  graph%curve_legend%text_offset !
```

The curve legend is distinct from the `text legend` (§7.5.4).

7.5.4 Text Legend

The `text legend` is a legend that can be setup by either the user or by *Tao* itself. *Tao* uses the text legend in conjunction with phase space plotting or histogram displays. The `text legend` is distinct from the `curve legend`. Parameters that affect the text legend are:

```
  graph%text_legend(:)      ! Array of strings to print
  graph%text_legend_origin  ! Position of legend.
```

7.5.5 Graph Types

Tao defines several kinds of graphs. The `graph%type` in the `tao_template_graph` (§10.13.2) sets the type.

"data"
: "Data" plotting is the plotting of a dependent variable on the y-axis vs an independent variable on the x-axis. Typically the independent variable will be the longitudinal position s-position as in the upper two graphs in Fig. 7.1. Also see Sec. §10.13.10 for an example where beam apertures are added to the graph.

  A "`data slice`" graph is plotting one data array on the y-axis versus another data array on the x-axis (§10.13.4). Also see `parametric plotting` (§10.13.5).

  With a `parametric` plot both the x and y values of the points on a curve are dependent upon an independent parameter (§10.13.5). This is similar to a `data slice` plot (§10.13.4).

"dynamic_aperture"
: A `dynamic aperture` graph (§10.13.11) draws the results from a dynamic aperture calculation (§10.12).

"floor_plan"
: A `floor plan` graph shows the physical layout of the machine (§10.13.8). A table maps lattice elements to a shape that is drawn (§10.13.9). The user may override the default mapping. Besides the lattice elements. the outline of the building or tunnel that the machine is in can be drawn (§10.11).

"histogram"
: Currently `histograms` (§10.13.12) are limited to displaying phase space data.

"key_table"
: The `key table` displays information about variables bound to keyboard keys §12.1. Key bindings are used in `single mode`.

"lat_layout"
: A `lattice layout` graph displays the lattice elements as a series of shapes as a function of the longitudinal position s (§10.13.7). The lowest graph in Fig. 7.1 is an example of a `lattice layout`. A table maps lattice elements to a shape that is drawn (§10.13.9). The user may override the default mapping.

"phase_space"
A `phase space graph` (§10.13.14) displays particle positions in phase space after a beam of particles has been tracked (§10.7).

"wave.0", "wave.a", "wave.b"
Wave analysis plotting (§9).

7.5.6 Graph Axes

Data graphs (§7.5.5) have three axes as shown in Fig. 7.4. The bottom axis is called `x`, and the left and right axes are called `y` and `y2` respectively. The `qp_axis_struct` structure (§7.7.6) is used to store axis parameters which can be accessed via the `graph%x`, `graph%y`, and `graph%y2` components in the `tao_template_graph` namelist (§10.13.2) or by using the `set graph` command (§11.29.14.

The `scale` command (§11.28) can be used to set the vertical axes. The `x_scale` (§11.41) command can be used to set the horizontal axis.

Normally there is only one vertical scale for a graph and this is associated with the `y` axis. However, if any curve of a given graph has `curve%use_y2` set to `True` then the `y2` axis will have an independent second scale. In this case, the `y2` axis numbers will be drawn. Notice that simply giving the `y2` axis a label does *not* make the `y2` axis scale independent of the `y` axis scale.

The following `tao_plot_page` namelist (§10.13.1) parameters affect the drawing of the axes:

```
text_height = 12              ! In points. Scales the height of all text
axis_number_text_scale = 0.9  ! Relative to text_height
axis_label_text_scale = 1.0   ! Relative to text_height
```

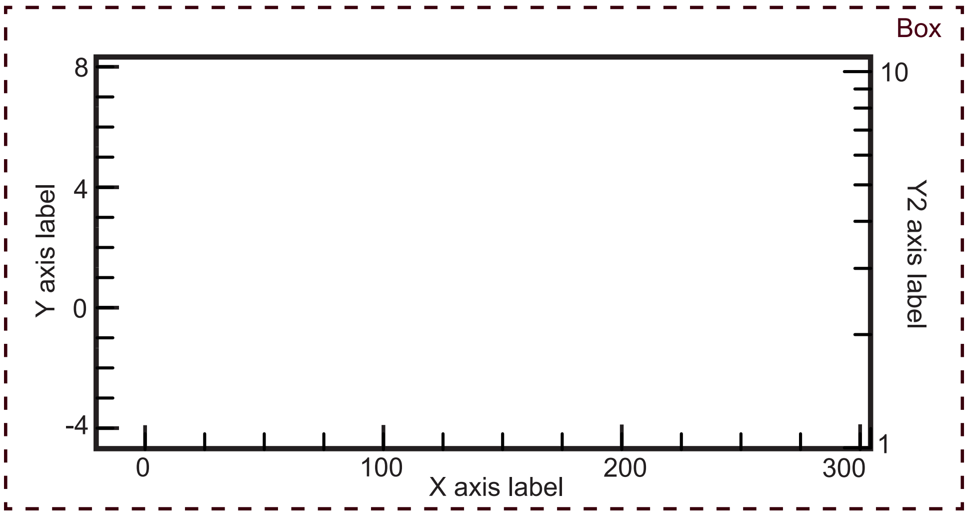


Figure 7.4: A data graph has three axes called `x` (bottom edge), `y` (left edge), and `y2` (right edge).

7.6 Curve

7.6.1 Overview

A `curve` is a data set to be displayed within a `graph`. For example, a `curve` may be the beta function of the `model` lattice. `Curves` have an associated set of points at which a symbol can be drawn. A curve also can have an associated curved line that can be drawn. For example, in Fig. 7.1 only the line is drawn with the two curves of the beta plot while both symbols and line are drawn for the two curves of the orbit plot (here the data points where symbols are drawn are the orbit at the edges of the lattice elements).

Some `graphs` do not have any associated curves. For example, a `lat_layout` graph does not have associated curves.

How to define `curves` when defining `template` plots is given in §10.13.2. The `show curve` command can be used to show curve parameters. The `set curve` command can be used to modify `curve` parameters.

7.6.2 Curve Name

All curves have a name. `Curves` may be referred to using the syntax:

```
  <plot>.<graph>.<curve>
```

where `<plot>` is the plot (or `region`) name, `<graph>` is the graph name and `<curve>` is the curve name. If the `.<curve>` ending is omitted, all curves of the named `graph`(s) are selected. If the `.<graph>.<curve>` ending is omitted, all curves of the named `plot`(s) are selected. Examples:

```
  show curve beta   ! Show info of all curves in all the displayed beta plots.
  show curve r13.g1 ! Show info on curves in ‘‘g1’’ graph of region r13.
  set graph orbit.g curve_legend_origin = 0.1 -0.2 "%BOX/LT"  ! Set curve legend origin
```

The last example sets the `curve legend` (§10.13.2) of the graph so that the curve legend of the graph is drawn with respect to the left top corner of the box.

7.6.3 Curve Line

Each curve may have an associated line that is drawn. The line may be a set of line segments connecting curve symbol points (§7.6.4) or may be a “smooth” curve calculated by evaluating the curve at a number of points.

`curve%draw_line` determines whether a curve is drawn through the data point symbols. The thickness, style (solid, dashed, etc.), and color of the line can be controlled by setting `curve%line`. If `plot%x_axis_type` is `"s"`, and `curve%component` does not contain `"meas"` or `"ref"`, *Tao* will attempt to calculate intermediate values in order to draw a smooth, accurate curve is drawn. Occasionally, this process is too slow or not desired for other reasons so setting `curve%smooth_line_calc` to False will prevent this calculation and the curve will be drawn as a series of lines connecting the symbol points. The default of `curve%smooth_line_calc` is True. Use the `set curve` command (§11.29) to toggle the drawing of lines. Alternatively, the `-disable_smooth_line_calc` switch can be used on the command line (§10.1) or the global variable `global%disable_smooth_line_calc` can be set in the *Tao* initialization file (§10.6).

The number of points to evaluate at when constructing a smoothed line is set by `plot_page%n_curve_pts` in the `tao_plot_page` namelist (§10.13.1) or by using the `set plot_page` command (§11.29.21). To override this value for a particular plot the `plot%n_curve_pts` parameter can be set in the `tao_template_plot` namelist or using the `set plot` command (§11.29.20). More evaluation points may give a more accurate curve at the expense of computation time.

7.6.4 Curve Symbol

`curve%draw_symbols` determines whether a symbol is drawn at the data points. The size, shape and color of the symbols is determined by `curve%symbol`. A given symbol point that is drawn has three numbers attached to it: The (x, y) position on the graph and an index number to help identify it. The index number of a particular symbol is the index of the datum or variable corresponding the symbol in the `d1_data` or `v1_var` array. These three numbers can be printed using the `show curve -symbol` command (§11.30). `curve%draw_symbol_index` determines whether the index number is printed besides the symbol. Use the `set curve` command (§11.29) to toggle the drawing of symbols. The default value for `curve%draw_symbol` is False if `plot%x_axis_type` is `"s"`, `"curve"`, `"lat"`, or `"var"` and True otherwise. The default `curve%draw_symbol_index` is always False.

The `graph%draw_only_good_user_data_or_vars` logical determines whether datums (§10.10) or variables (§10.9) with a `good_user` component set to `False` are drawn. The default is to not draw them which means that data or variables not used in an optimization are not drawn.

7.6.5 Curve Component

A "`data`" graph (§7.5.5) is used to draw lattice parameters such as orbits, or *Tao* data (§6), or variable values such as quadrupole strengths. The data values will depend upon where the data comes from. This is determined, in part, by the setting of the `component` parameter in the `tao_template_graph` namelist (§10.13.2). The `component` may be one of:

```
"model"             ! model values. Default.
"design"            ! design values.
"base"              ! Base values
"meas"              ! data values.
"ref"               ! reference data values.
"beam_chamber_wall" ! Beam chamber wall
```

Additionally, `component` may be set to plot a linear combination of the above. For example:

```
&tao_template_graph
  curve(2)%component = "model - design"
  ...
```

This will plot the difference between the `model` and `design` values. The default value of `%component` is `"model"`.

7.6.6 Curve Data Source

The `data_source` parameter of a curve is the type of information for the source of the data points. `data_source` must be one of:

```
"data"              ! A d1_data array is the source of the curve points.
"var"               ! A v1_var array is the source of the curve points.
"lat" (Default)     ! The curve points are computed directly from the lattice.
"beam"              ! The curve points are computed from tracking a beam of particles.
"multi_turn_orbit"  ! Computation is from multi-turn tracking.
```

The default for `data_source` is `"lat"`. With `data_source` set to `"data"`, the values of the curve points come from the `d1_data` array structure named by the curve's `data_type` parameter (§7.6.7).

If `data_source` is set to `var`, the values of the curve points come from a `v1_var` array structure. If it is set to `lat` the curve data points are calculated from the lattice without regard to any data structures. `data_source` can be set to `beam` when tracking beams of particles. In this case, the curve points are calculated from the tracking. With `beam`, the particular bunch that the data is extracted from can be specified via `ix_bunch`. The default is `0` which combines all the bunches of the beam for the calculation.

Used in conjunction with `data_type` and `component` (§7.6.5). For example (§7.6.6), a curve of the orbit with `data_source` set to `"beam"` would use the beam centroid computations. If the `data_source` was set to `"lat"` the computed orbit using single particle tracking is used.

Example: With `data_type` set to `beta.x`, the setting of `data_source` to `lat` gives the beta as calculated from the lattice and `beam` gives the beta as calculated from the shape of the beam.

7.6.7 Curve Data Type

The `data_type` of a curve specifies what is being plotted. What the valid settings for `data_type` are depends upon the type of graph (§7.5.5).

graph%type = "data", or "histogram"

Valid settings for `data_type` are any *Tao* datum type (§6.8), *Tao* variable (§5), and the following electric and magnetic field components:

```
b0_field.x,  b0_field.y,  b0_field.z,  b0_curl.x,  b0_curl.y,  b0_curl.z,  b0_div
e0_field.x,  e0_field.y,  e0_field.z,  e0_curl.x,  e0_curl.y,  e0_curl.z,  e0_div
```

The field data types with names starting with "b_" and "e_" evaluate the field along the single particle trajectory while the field data types with names starting with "b0_" and "e0_" are evaluated along a constant transverse position specified by the curve's `orbit` parameter.

graph%type = "dynamic_aperture"

Valid settings for `data_type` are:

```
"beam_ellipse"
"dynamic_aperture"
```

graph%type = floor_plan", "lat_layout", or "key_table"

There are not curves associated with these graph types.

graph%type = "phase_space"

Valid settings for `data_type` are:

```
"x",  "px",  "y",  "py",  "z",  "pz",
"intensity",  "intensity_x",  "intensity_y"     ! Photon intensity
"phase_x", "phase_y"                            ! Photon coherent phase
```

For example, with `graph%type` set to `dynamic_aperture` the

Thus in the above example the curve point values are obtained from `orbit.x` data. To be valid the data structure named by `data_type` must be set up in an initialization file. If not given, the default `data_type` is

```
<plot%name>.<graph%name>
```

7.7 Quick_Plot Plotting

`Quick_plot` is a software library developed for *Bmad* and *Tao* for graphics plotting.

7.7.1 Length and Position Units

Positions and lengths with `quick_plot` generally have an associated "`units`" string which determines how (x, y) positions or (dx, dy) lengths are to be interpreted. The syntax of the `units` parameter is:

```
  "unit_type/ref_object/corner"
```

A `units` string has a `unit_type`, `ref_object` and `corner` components separated by slashes "/".

The `unit_type` component is the type of units which can be one of:

```
   "%"       - Percent.
   "DATA"    - Data units associated with a graph.
   "MM"      - millimeters.
   "INCH"    - Inches.
   "POINTS"  - Printers points (72 points = 1 inch, 1 pt ~ 1 pixel).
```

Note: If `unit_type` is set to `"DATA"`, `ref_object`, if present, must be `"GRAPH"` and `corner`, if present, must be `"LB"`.

The `ref_object` component is a reference object which can be one of:

```
   "PAGE"  -- Relative to the plot display window.
   "BOX"   -- Relative to the box the graph is associated with.
   "GRAPH" -- Relative to the graph rectangle.
```

The `ref_object` component is optional if a relative length is being specified and the `unit_type` is anything other than `%`. If `unit_type` is `%`, the slash between the `unit_type` and the `ref_object` may be omitted.

Note: The `"PAGE"` reference is the entire `plot page` and not the `plot area`. The `plot area` is only used for defining the placement of `regions`.

The `corner` component is the origin location of the reference object. `corner` can be one of:

```
   "LB" -- Left Bottom of reference object. Default.
   "LT" -- Left Top.
   "RB" -- Right Bottom.
   "RT" -- Right Top.
```

The `ref_object` component is optional if a relative length is being specified.

Examples:

```
  "DATA"          -- Equivalent to "DATA/GRAPH/LB"
  "DATA/GRAPH/LB" -- Same as above.
  "DATA/BOX/RT"   -- ILLEGAL: DATA must always go with GRAPH/LB.
  "%/PAGE/LT"     -- Equivalent to "%PAGE/LT"
  "%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. Equivalent to "INCH/PAGE/LB"
```

Units can be set in an initialization file or with the `set` command. Example:

```
  set plot_page title%units = '%PAGE'
```

7.7.2 Text Justification Units

Text justification units is a two character string that sets where a line of text is to be printed with respect to the text (x, y) position. The first character of the justification string gives the horizontal alignment:

```
   "L" -- Left justify
   "C" -- Center justify
   "R" -- Right justify
```

The second character of the justification string gives the vertical alignment:

```
  "B" -- Bottom justify
  "C" -- Center justify
  "T" -- Top justify
```

Example:

```
plot_page%title%justify = 'CC'
```

7.7.3 qp_point_struct

QuickPlot defines a number of structures to parameterize such things like line and symbol properties.

The qp_point_struct defines where a point is:

```
type qp_point_struct:
  x     = <real>     ! Horizontal offset of point from fiducial point
  y     = <real>     ! Vertical offset of point from fiducial point
  units = "<units>"  ! Units of x & y (§7.7.1).
```

Example:

```
graph%curve_legend_origin = 5.0, -2.0, "POINTS/GRAPH/LT"
```

In this example the fiducial point the left-top point on the graph rectangle. The curve_legend_origin is positioned 5.0 points horizontally to the left and 2.0 points vertically downward from this fiducial point.

7.7.4 qp_line_struct

The parameters associated with data lines drawn in a graph are contained in the qp_line_struct:

```
type qp_line_struct:
  width    = <integer>  ! Default = 1
  color    = <string>   ! Default = "black" (§7.7.9).
  pattern  = <string>   ! Default = "solid" (§7.7.10).
```

7.7.5 Symbols

The parameters associated with symbols that are drawn are contained in the qp_symbol_struct:

```
type qp_symbol_struct:
  type          = <string>  ! Default = "dot"
  height        = <real>    ! Size in points. Default = 10
  color         = <string>  ! Default = "black" (§7.7.9)
  fill_pattern  = <string>  ! Default = "solid_fill"
  line_width    = <integer> ! Default = 1.
```

The symbol types are:

```
square                 triangle                   square_concave
dot                    circle_plus                diamond
plus                   circle_dot                 star5
times                  square_filled              triangle_filled
circle                 circle_filled              red_cross
x                      star5_filled               star_of_david
```

These symbols are illustrated in Table 7.1. Symbol type names are case insensitive.

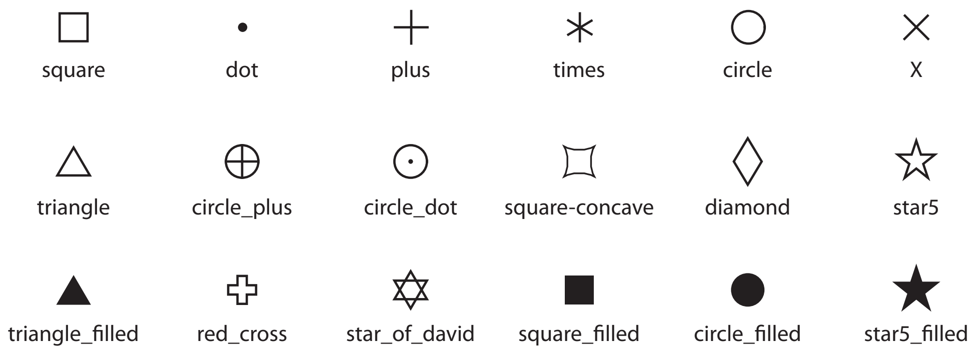


Table 7.1: Plotting Symbols.

7.7.6 qp_axis_struct

The `qp_axis_struct` structure defines the properties of a graph axis

```
  type qp_axis_struct::
    label              = "<string>" ! Axis label string.
    min                = <real>     ! Min is the left or bottom axis number.
    max                = <real>     ! Max is the right or top axis number.
    number_offset      = <real>     ! Offset from axis line in inches.
    label_offset       = <real>     ! Offset from numbers in inches.
    major_tick_len     = <real>     ! Major tick length in inches.
    minor_tick_len     = <real>     ! Minor tick length in inches.
    label_color        = <string>   ! Color of the label string (§7.7.9)
    major_div          = <integer>  ! Number of major divisions
    major_div_nominal  = <integer>  ! Major divisions nominal value.
    minor_div          = <integer>  ! Minor divisions. 0 = Tao will choose.
    minor_div_max      = <integer>  ! Max minor div number if Tao chooses.
    places             = <integer>  ! Number of digits to print
    type               = <string>   ! Axis type: "LINEAR" or "LOG".
    bounds             = <string>   ! Axis bounds: "GENERAL", "ZERO_AT_END", etc.
    tick_side          = <integer>  ! 1 = draw to the inside, 0 = both, -1 = outside.
    number_side        = <integer>  ! 1 = draw to the inside, -1 = outside.
    draw_label         = <logical>  ! Draw the label string
    draw_numbers       = <logical>  ! Draw the numbers.
```

The `%bounds` parameter sets how the axes min and max values are calculated when plots are initially instantiated and when `scale`, `x_scale`, and `xy_scale` commands are used. Possible settings are:

```
  "ZERO_AT_END"      ! Min or max value is set to zero.
  "ZERO_SYMMETRIC"   ! Min and max chosen so that max = -min.
  "GENERAL"          ! No restrictions (default).
  "EXACT"            ! The User min/max is used.
```

If input min and max values are specified by the User, *Tao* will take the specified values as the starting point to find “nice” min and max values to use. For example, with the command

```
  scale all 0 19
```

and with `bounds` set to `"GENERAL"`, the min and max values will be set to 0 and 20. The exception is when `bounds` is set to `"EXACT"`. In this case the User supplied min and max values will be used as is.

Examples:

```
Tao> set graph r13 y%bounds = "zero_at_end"
Tao> scale r13 200 280   ! Graph bounds set to [0, 300]

Tao> set graph r13 y%bounds = "zero_symmetric"
Tao> scale r13 200 280   ! Graph bounds set to [-300, 300]

Tao> set graph r13 y%bounds = "general"
Tao> scale r13 20 190    ! Graph bounds set to [0, 200]

Tao> set graph r13 y2%bounds = "exact"
Tao> scale r13 -y2 20 190    ! Y2 graph bounds set to [20, 190]
```

Both `major_div` and `major_div_nominal` set the number of major divisions in the plot. The difference between the two is that with `major_div` set positive and `major_div_nominal` set zero or negative, the number of major divisions is fixed at the value of `major_div`. With `major_div_nominal` positive, the value of `major_div` is ignored, and the number of major divisions will be chosen to be a "nice" value near the value of `major_div_nominal`. If neither `major_div` nor `major_div_nominal` is set positive, a value will be chosen for `major_div_nominal` by *Tao*. If you are unsure which to set, it is recommended that `major_div_nominal` be used.

The `places` parameter set the number of places to display a number. *Tao* will automatically calculate this number and it is not user settable.

The `label` parameter may include Greek letters, subscripts, superscripts, and special characters. Encoding for these are given in Table 7.2.

7.7.7 qp_legend_struct

The parameters associated with drawing a curve legend (§7.3) are contained in the parameter `plot_page%curve_legend` (§10.13.1). This parameter is an instance of a `qp_legend_struct` which has the structure:

```
  type qp_legend_struct
    row_spacing = <real>           ! Spacing between rows. Default = 1.0.
    line_length = <real>           ! Length of the line in points.
    text_offset = <real>           ! Horizontal offset in points between the line and the text.
    logical draw_line = <logic>    ! Draw lines?
    logical draw_symbol = <logic>  ! Draw symbols?
    logical draw_text = <logic>    ! Draw text?
  end type
```

7.7.8 String Escape Sequences

Table 7.3 shows how the character string `"\g<r>"`, where `"<r>"` is a Roman letter, map onto the Greek character set.

7.7.9 Color Names

Possible settings for color parameters are:

```
  White   (actually the background color)      Orange
  Black   (actually the foreground color)      Yellow_Green
  Red                                          Light_Green
  Green                                        Navy_Blue
```

\u	Start a superscript or end a subscript
\d	Start a subscript or end a superscript. \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 (×)
\\.	Centered dot (·)
\A	Angstrom symbol (Å)
\gx	Greek letter corresponding to roman letter x. See Table 7.3.
\mN \mNN	Graph marker number `N` or `NN` (1-31)
\\(NNNN)	Character number NNNN (1 to 4 decimal digits) from the Hershey character set which includes a number of special characters including mathematical, musical, astronomical, and cartographical symbols.

Table 7.2: Escape Sequences for Labels.

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	ο	π	ρ	σ	τ	υ	ϕ	χ	ψ	ω
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	Ο	Π	Ρ	Σ	Τ	ϒ	Φ	Χ	Ψ	Ω

Table 7.3: Conversion for the string `"\g<r>"` where `"<r>"` is a Roman character to the corresponding Greek character.

```
  Blue                                        Purple
  Cyan                                        Reddish_Purple
  Magenta                                     Dark_Grey
  Yellow                                      Light_Grey
```

Color names are case insensitive.

7.7.10 Line Pattern Names

Possible settings for line patterns are:

```
  solid      ! Solid line               dotted     ! Dotted line
  dashed     ! Dashed line              dash_dot3  ! Dash--dot--dot--dot line
  dash_dot   ! Dash--dot line
```

Pattern names are case insensitive.

7.7.11 Fill Pattern Names

Possible fill pattern settings for symbols are:

`solid_fill`	`hatched`
`no_fill`	`cross_hatched`

Fill pattern names are case insensitive.

Chapter 8

Optimization: Lattice Correction and Design

"`Optimization`" is the process of varying `model` (§3.4) lattice parameters so that a given set of properties are as close to a given set of "desired" values as possible. Optimization problems generally fall into one of two categories. One category involves "`lattice correction`" (§8.4). Problems in this category involve matching the `model` lattice to actual measured data. For example, orbit flattening involves varying steerings in the `model` lattice so that the orbit calculated from the `model` lattice matches the measured orbit. The steering strengths in the `model` lattice can then be used to calculate what changes are needed to correct the orbit in the actual machine.

The other category of optimization problems involves "`lattice design`" (§8.5) where lattice parameters are varied to achieve some set of ideal properties. For example, varying sextupole magnet strengths in order to give maximum dynamic aperture.

Note: The *Bmad* and *Tao* tutorial (§1.2) has examples of how to construct both lattice correction and lattice design optimizations.

8.1 Optimization Overview

Optimization involves "`data`" and "`variables`". `Data` are the parameters to be optimized. For example, orbit positions when flattening an orbit or the value of the beta function at a certain point when designing a lattice. `Variables` are what is to be varied which can be steering strengths, magnet positions, etc. Any lattice parameters can be used with optimization as long as varying the variables will affect the data. For example, orbit flattening is not restricted to varying steerings and can be done by, say, varying magnet misalignments.

How data is organized in *Tao* is discussed in Chapter §6. How to define datums in a *Tao* initialization file is discussed in Section §10.10. In brief, each datum has a set of associated parameters. For example, each datum has "`model`" and "`design`" values which are the values of the datum as calculated from the `model` and `design` lattices. Each datum also has a "`measured`" value which is set by the User. This value can be from an actual measurement which is typical when doing lattice correction or may be the desired value of the datum which is typical when doing lattice design. Which of the defined datums are varied during optimization is controlled by setting the `good_user` component of a datum (§6.5).

How variables are organized in *Tao* is discussed in Chapter §5. How to define variables in a *Tao* initialization file is discussed in Section §10.9. In brief, like datums, each variable has a number of associated parameters. For example, each variable has a "`model`" value which controls the corresponding value or values (a variable can control multiple parameters simultaneously) in the `model` lattice. There are also "`low_lim`" and "`high_lim`" parameters that can be set by the User that are used to keep the variable `model` value within a given range.

Typically, optimization involves minimizing one or more "objectives" or "merit functions". *Tao* itself implements "single objective" optimization where there is only one merit function[1]. The general form of the merit function $\mathcal{M}$ in *Tao* is

$$\mathcal{M} \equiv \sum_i w_i \left[\delta D_i\right]^2 + \sum_j w_j \left[\delta V_j\right]^2 \tag{8.1}$$

The first term on the RHS is a sum over the `data` and the second term is a sum over the `variables`. The w_i and w_j are weights, specified by the User and stored in the `weight` component of a datum or variable. The value of δD (§8.2) for a datum or the value of δV (§8.3) for a variable is put in the `delta_merit` component of the datum or variable. The contribution to the merit function, which is $[\delta D]^2$ for a datum and $[\delta V]^2$ for a variable, is put in the `merit` component of the datum or variable.

In some cases it might be desired a datum or variable's value. For example, maximizing the dynamic aperture. In this case, the datum or variable weight w needs to be set negative and the contribution to the merit function will be negative. Note: The `lmdif` optimizer (§8.6) will not function correctly if there are negative weights.

In the case where it is not possible to calculate `delta_merit` for a datum, for example, in the case where a datum is associated with a Twiss parameter but the lattice is unstable, the value of `delta_merit` is set equal to the value of the datum's `invalid` component. This is useful for ensuring that the optimization does not get stuck in an unstable state. Additionally, the following `data_type` (§6.9) can be used to drive the optimization away from an instability:

```
unstable.eigen
unstable.eigen.a, .b, .c
unstable.lattice
unstable.orbit
```

There are a number of `show` commands (§11.30) that can be used to show optimization parameters:

```
show constraints    ! §11.30.6 List of constraints
show data           ! §11.30.9 Show data info
show derivative     ! §11.30.10 Show derivative info
show merit          ! §11.30.23 Top contributors to the merit function
show optimizer      ! §11.30.24 Optimizer parameters
show variable       ! §11.30.39 Show variable info
```

8.2 Calculation of δD for a Datum

The δD terms in the merit function (Eq. (8.1)) (the value of which is put in the datum's `delta_merit` component) are used to drive the `model` lattice parameters towards a certain set of desirable properties. How δD is calculated for a given datum is determined, in part, by the setting of the datum's `merit_type` component. The possible settings of this component are:

```
"target"                       ! Default
```

[1]For "multiple objective" optimization, there is a separate program called `moga` that can be used (§1.5).

```
  "average"                          ! Uses evaluation range
  "integral"                         ! Uses evaluation range
  "rms"                              ! Uses evaluation range
  "min", "max"
  "abs_min", "abs_max"
  "max-min"                          ! Uses evaluation range
```

As noted, some of these settings need an `evaluation range` (§6.4). An evaluation range occurs when a datum's `ele_start_name` is set and the evaluation range covers the region from the lattice element identified by `ele_start_name` and ending at the `ele_name` element. Note: It does not make sense for a datum with a `data_type` that does not have an associated evaluation element (for example, a datum with an emittance related `data_type`), to have an evaluation range. Therefore, for such datums, it is not possible to use any one of the four `merit_type` settings that use an evaluation range.

The calculation of δD is a three step process. The first step is calculating the `model`, `base` and `design` values for a datum which are calculated from the `model`, `base`, and `design` lattices. The first step calculation depends upon the setting of `merit_type` as well as whether there is an associated evaluation range.

For datums that do not have an evaluation range, the value of the datum's `model`, `base`, or `design` component is the appropriate value of the `data_type`. For example, the b-mode emittance for a `data_type` setting of `"emit.b"` or the horizontal dispersion (for `data_type` set to `"eta.x"`) at the evaluation element (named by `ele_name`). If there is an associated reference element, the value will be modified by subtracting off the `data_type` value at the reference element. The exception is that for `merit_type` set to `"abs_min"` or `"abs_max"`, the absolute value is taken (this is done after any reference value is subtracted).

For datums that do have an an evaluation region, the `model`, `base`, or `design` datum values are computed as described below. In all cases, if there is an associated reference element, the value will be modified by subtracting off the `data_type` value at the reference element.

"target"
: With `merit_type` set to`"target"`, an evaluation range is not permitted since it does not make sense to have one.

"average"
: With `"average"` as the `merit_type`, the value is the mean of the `data_type` over the evaluation region. This is just the integral normalized by the length of the region. To save time, the data is only evaluated at the ends of elements and the average is evaluated assuming linear variation between points.

"integral"
: With `"integral"` as the `merit_type`, the value is the integral of the `data_type` over the evaluation region. To save time, the data is only evaluated at the ends of elements and the integral is evaluated assuming linear variation between points.

"min" or "max"
: With `"min"` or `"max"` as the `merit_type`, the value is the minimum or maximum value of the `data_type` over the evaluation region.

"abs_min" or "abs_max"
: With `"abs_min"` or `"abs_max"` as the `merit_type`, the value for a lattice is the minimum or maximum value of the absolute value of the `data_type` over the evaluation region.

"max-min"
: With `"max-min"` as the `merit_type`, the value for a lattice is the maximum value of the `data_type` over the evaluation region minus the minimum value over the evaluation region.

`Opt_with_ref`	`Opt_with_base`	`Composite Value Expression`
F	F	`model - meas`
F	T	`model - meas - base`
T	F	`(model - meas) - (design - ref)`
T	T	`(model - meas) - (base - ref)`

Table 8.1: The expression used to combine the values of `model`, `base`, `design`, `meas`, and `ref` into one "composite" value is determined by the settings of `opt_with_ref` and `opt_with_base` global parameters. These expressions are the same as Table 8.2.

"rms"
: With `"rms"` as the `merit_type`, the value is the RMS of the `data_type` over the evaluation region. To save time, the data is only evaluated at the ends of elements and the RMS is evaluated assuming linear variation between points.

After a datum's `model`, `base` and `design` values are calculated, these values are combined together along with the `meas` and `ref` values set by the User[2]. The formula used to combine these values into one "`composite`" value is determined by the setting of two global logicals `opt_with_ref` and `opt_with_base`. These parameters are in the `tao_global_struct` structure (§10.6.1). Table 8.1 shows the expressions.

After the `composite` value is calculated, δD will be set equal to the `composite` value except if the `merit_type` is `"min"`, `"max"`, `"abs_min"`, or `"abs_max"`. For `"min"` and `"abs_min"`, δD will be

$$\delta D = \begin{cases} 0 & \texttt{composite} > 0 \\ \texttt{composite} & \texttt{otherwise} \end{cases} \tag{8.2}$$

That is, the datum only contributes to the merit function if the `composite` is value is negative. For `"max"` and `"abs_max"`, δD will be

$$\delta D = \begin{cases} 0 & \texttt{composite} < 0 \\ \texttt{composite} & \texttt{otherwise} \end{cases} \tag{8.3}$$

That is, the datum only contributes to the merit function if the `composite` is value is positive.

8.3 Calculation of δV for a Variable

The δV terms in the merit function (Eq. (8.1)) serve one of two purposes. Such terms can be used to keep variables within certain limits or can be used to guide the optimization towards a solution where a variable has a certain value. How δV is calculated for a given variable is determined by the setting of the variable's `merit_type` component. This component can have one of two values:

```
"limit"     ! Default
"target"
```

For `merit_type` set to `"limit"`, the value of δV is determined by the setting of the variable's `high_lim` and `low_lim` components. If not explicitly set, `high_lim` defaults to 10^{30} and `low_lim` defaults to -10^{30}.

[2]keep in mind that the `ref` component is different from the `ele_ref_name` component (§6.2). `ele_ref_name` is used to set a reference element which is used when evaluating `model`, `base`, and `design` values (§8.4). The value of the `ref` component is set by the user.

The value of δV in this case is

$$\delta V = \begin{cases} \texttt{model} - \texttt{high\_lim} & \texttt{model} > \texttt{high\_lim} \\ \texttt{model} - \texttt{low\_lim} & \texttt{model} > \texttt{low\_lim} \\ 0 & \texttt{otherwise} \end{cases} \tag{8.4}$$

Note: When running the optimizer, if the parameter `global%var_limits_on` (§10.6) is `True`, and if the `model` value of a variable is outside of the range set by the limits, *Tao* will do two things: First, the `model` value of the variable will be set to the value of the nearest limit and, second, the variable's `good_user` parameter (§5) is set to False so that no further variation by the optimizer is done. This is done independent of the setting of `merit_type` for the variable and independent of whether the variable is being used in the optimization. Sometimes it is convenient to not set the `model` value to the nearest limit for variables that are not being used in the optimization. In this case, the global parameter `global%only_limit_opt_vars` may be set to `True`. If this is done, only variables that the optimizer is allowed to vary are restricted.

Note: The `global%optimizer_var_limit_warn` parameter controls whether a warning is printed when a variable value goes past a limit. The default is `True`.

Opt_with_ref	Opt_with_base	Delta_Merit (δV) Formula
F	F	model - meas
F	T	model - meas - base
T	F	(model - meas) - (design - ref)
T	T	(model - meas) - (base - ref)

Table 8.2: When `merit_type` is set to `"target"`, the formula for evaluating δV is determined by the settings of `opt_with_ref` and `opt_with_base` global parameters as shown in the table. These expressions are the same as Table 8.1.

For `merit_type` set to `"target"`, the formula used to compute δV is determined by the setting of two global logicals `opt_with_ref` and `opt_with_base`. These parameters are in the `tao_global_struct` structure (§10.6.1). Table 8.2 shows the expressions used to evaluate δV In the table, `model`, `base` and `design` are the values for the variable as set in the `model`, `base`, and `design` lattices, and the variable's `meas` and `ref` values are set by the User.

8.4 Lattice Corrections

Lattice correction is the process of varying a set of parameters in a machine achieve some desirable state. Typically, there are three stages. First there is a measurement. After this, corrections can be calculated (the optimization stage). Finally, the calculated corrections are loaded into the machine.

There are several variations of how optimization is done. To make the discussion concrete, the case where a beta function measurement is used to calculate quadrupole `K1` strength changes to correct the machine optics is considered.[3]. In this case, both data and variables will have their `merit_type` set to `"target"`,

[3]The beta function can be measured, for example, by pinging the beam and observing the oscillations at the beam position monitors.

the global `opt_with_ref` and `opt_with_base` will be False. With this, δD and δV in Eq. (8.1) will be

$$\begin{aligned} \delta\mathrm{D} &= \texttt{model\_beta} - \texttt{meas\_beta} \\ \delta\mathrm{V} &= \texttt{model\_K1} - \texttt{meas\_K1} \end{aligned} \tag{8.5}$$

The `meas_beta` data values will is the measured beta function and the variable `meas_K1` values will be the measured quadrupole `K1` at the time the data was taken. The quadrupole `K1` could be measured from, for example, from the currents through the quadrupoles combined with known current to field calibrations. The δV_j terms in the merit function prevents degeneracies (or near degeneracies) in the problem which would allow *Tao* to find solutions where `model_beta` matches `meas_beta` with the `model_k1` strengths having "unphysical" values far from the measured strengths. The weights w_i and w_j in Eq. (8.1) need to be set depending upon how accurate the measured data is relative to the measured magnet strengths.

If the fit is good, the beta function is corrected by changing the quadrupole strengths in the machine by an amount `dK1` given by

```
  dK1 = design_K1 - model_K1
```

where `design_K1` is the `design` value for the quadrupole strengths. The equation for `dK1` is derived using the following logic: Once a fit to the measured data has been made, the `model` lattice corresponds, more or less, to the actual state of the machine. On the other hand, the desired state is given by the `design` lattice . Thus the difference, `design - model`, represents the desired state minus the actual state. The final state after the correction will be

```
  Final_State = Initial_State + Change
              = Actual_State + (Desired_State - Actual_State)
              = Desired_State
```

which is what is wanted.

Notice that the fitting process is independent of the strengths of the parameters in the `design` lattice. That is, the fit involves the actual machine state independent of what the desired state is. It is not until the values needed for the correction are computed that the parameter strengths in the `design` lattice come into play.

To the extent that the measured beta function can be well fit determines the extent to which the beta function can be corrected. For example, if an unwanted quadrupole error is generated by some element at a spot that is far from any correctors (quadrupoles that can vary), it will not be possible to fit the measured beta function well and it will not be possible to make a good correction. If, on the other hand, an unwanted error is generated next to one corrector, the measured beta function can be well fit and the `model` lattice will have a strength change from the `design` for that one corrector. Varying that one corrector can then cancel out the unwanted kick.

Another point is that the correction algorithm will work with varying any set of parameters as long as the variation in the parameters affect the `model` data. Thus an analysis can be made of the beam orbit using dipole rolls and/or quadrupole offsets as variables or any combination thereof. If the fit is good, rolling the dipoles and moving the quadrupoles will correct the orbit. With *Tao*, the User has complete freedom to vary any parameters in the fitting process.

Typically, at the start of the fit, the `model` lattice is, by default, equal to the `design` lattice but this is not necessary. Generally, the actual machine state is near enough to the `design` machine state so that the machine will behave roughly linearly with the variation in the parameters (if the machine parameters are far from the design values, it may not be possible to store beam to take a measurement in the first place). This means that there will be only one minimum merit function state so that the parameter values (quadrupole strengths in this case) at the end of the fit are independent of the starting state. To put this in other terms, the User generally does not have to worry about the initial state of the `model` at the start of a fit. This is in contrast to lattice design (§8.5) where there are typically many local minima and it can take days of work to find a good operating point. With lattice correction, on the other hand, the near linear nature of the problem means that finding a solution in a machine with hundreds of correctors and hundreds of BPM readings can be done in seconds with the `lm` or `lmdif` optimizers (§8.6).

8.5 Lattice Design

Lattice design is the process of calculating `variable` strengths to meet a number of criteria called constraints. For example, one constraint could be that the beta function in some part of the lattice not exceed a certain value or a constraint is used to keep the `model` at a certain desired value. To keep the nomenclature consistent, the desired value is labeled `meas` even though it is not a "measured" value.

Lattice design is an art in that there is no known algorithm that will guarantee that a global minimum has been found. And indeed, there is no known general procedure for checking that a given minimum is the global minimum. *Tao* has one optimizer for global minimum searching and that is the `de` optimizer (§8.6). Typical strategies involve using the `de` optimizer to find a minimum, then using the `lm` or `lmdif` optimizer to refine the answer, and then switching back to the `de` optimizer to see if a better minimum can be found. This is repeated while varying the `de` parameters and/or weights in the merit function (Eq. (8.1)).

As an alternative, there is a *Bmad* based program separate from *Tao* called `moga` that can be used (§1.5). This program implements multi objective optimization.

8.6 Optimizers in Tao

The algorithm used to vary the `model` variables to minimize `M` is called an `optimizer`. In `command line mode` the `run` command is used to invoke an `optimizer`. In `single mode` the `g` key starts an optimizer. In both modes the period key (`‘‘.’’`) stops the optimization (however, the `global%optimizer_allow_user_abort` parameter (§10.6) can be set to False to prevent this). Running an optimizer is also called "fitting" since one is trying to get the `model` data to be equal to the `measured` data. With orbits this is also called "flattening" since one generally wants to end up with an orbit that is on–axis.

The optimizer that is used can be defined when using the `run` command but the default optimizer can be set in the *Tao* input file by setting the `global%optimizer` component (§10.6).

When the optimizer is run in *Tao*, the optimizer, after it initializes itself, takes a number of `cycles`. Each cycle consists of changing the values of the variables the optimizer is allowed to change. The number of steps that the optimizer will take is determined by the parameter `global%n_opti_cycles` (§10.6). When the optimizer initializes itself and goes through `global%n_opti_cycles`, it is said to have gone through one `loop`. After going through through `global%n_opti_loops` loops, the optimizer will automatically stop. To immediately stop the optimizer the period key `‘‘.’’` may be pressed. Note: In `single_mode` (§12), `n_opti_loops` is ignored and the optimizer will loop forever.

There are currently three optimizers that can be used:

`lm`
: `lm` is an optimizer based upon the Levenburg-Marquardt algorithm [NR92]. This algorithm looks at the local derivative matrix of `dData/dVariable` and takes steps in variable space accordingly. The derivative matrix is calculated beforehand by varying all the variables by an amount set by the variable's `step` component (§10.9). The `step` size should be chosen large enough so that round-off

errors will not make computation of the derivatives inaccurate but the step size should not be so large that the derivatives are effected by nonlinearities. By default, the derivative matrix will be recalculated each `loop` but this can be changed by setting the `global%derivative_recalc` global parameter (§10.6). The reason to not recalculate the derivative matrix is one of time. However, if the calculated derivative matrix is not accurate (that is, if the variables have changed enough from the last time the matrix was calculated and the nonlinearities in the lattice are large enough), the `lm` optimizer will not work very well. In any case, this method will only find local minimum. When running with `lm`, the value of a parameter called `a_lambda` will be printed. This "damping factor", which is always positive definite, is a measure of how well the variation of the δD_i terms with respect to the variables in Eq. (8.1) matches the computed derivative matrix. A small `a_lambda`, much less than one, indicates good agreement while a larger `a_lambda`, much greater than one, means that there is a mismatch. If `a_lambda` is large, the optimizer will not be able to make much progress. There are several reasons for a large `a_lambda`. First, the working point may have shifted enough so that the derivative matrix needs to be recalculated. If recalculating the derivative matrix does not fix the problem, it may be that one or more variable `step` sizes are either too small or two large. If the value of `step` is too small for a variable, round-off error can cause the calculation of the variable's derivatives to be off. If the value of the `step` is too large, nonlinearities can throw off the derivative calculation. To test if the step size is set correctly first use the `show merit -derivative` command to see what variables have the largest `dMerit/dVariable` values as computed from the derivative matrix. [Note: Generally the best thing is to concentrate on the variables with the largest derivatives since these give the "most bang for the buck".] These values should be small when at the true merit function minimum. The `change variable` command can now be used with varying step sizes to see if the actual `dMerit/dVariable` change matches this. The output from the `change` command will show the `dMerit/dVariable` value computed from varying the variable by the amount given in the change command. If the two do not agree, there may be a problem. Warning! At the $\mathcal{M}$ minimum the value of `dMerit/dVariable` is very sensitive to the value of the variables. To not get fooled, move away the minimum. For more information on `a_lambda`, see the Wikipedia article on the "`Levenberg-Marquardt algorithm`". In this article the variable is denoted λ.

`lmdif`

The `lmdif` optimizer is like the `lm` optimizer except that it builds up the information it needs on the derivative matrix by initially taking small steps over the first `n` cycles where `n` is the number of variables. The advantage of this is that you do not have to set a `step` size for the variables. The disadvantage is that for `lmdif` to be useful, the number of `cycles` (set by `set global n_cycles =<XXX>`) must be greater than the number of variables. Again, like `lm`, this method will only find local minimum.

`de`

The `de` optimizer stands for `differential evolution`[Sto96]. The advantage of this optimizer is that it looks for global minimum. The disadvantage is that it is slow to find the bottom of a local minimum. A good strategy sometimes when trying to find a global minimum is to use `de` in combination with `lm` or `lmdif` one after the other. One important parameter with the `de` optimizer is the `step` size. A larger step size means that the optimizer will tend to explore larger areas of variable space but the trade off is that this will make it harder to find minimum in the locally. One good strategy is to vary the `step` size to see what is effective. Remember, the optimal step size will be different for different problems and for different starting points. The `step` size that is appropriate of the `de` optimizer will, in general, be different from the `step` size for the `lm` optimizer. For this reason, and to facilitate changing the step size, the actual step size used by the `de` optimizer is the step size given by a variable's `step` component multiplied by the global variable `global%de_lm_step_ratio`. This global variable can be varied using the `set` command (§11.29). The number of trial solutions used in the optimization is

```
population = number_of_variables * global%de_var_to_population_factor
```

There are also a number of parameters that can be set that will affect how the optimizer works. See Section §10.6 for more details.

`svd`

The `svd` optimizer uses a singular value decomposition calculation. With the `svd` optimizer, the setting of the `global%n_opti_cycles` parameter is ignored. One optimization loop consists of applying svd to the derivative matrix to locate a new set of variable values. If the merit function decreases with the new set, the new values are retained and the optimization loop is finished. If the merit function increases, and if the global variable `global%svd_retreat_on_merit_increase` is True (the default), the variables are set to the original variable settings. In either case, an increasing merit function will stop the execution of additional loops.

The `global%svd_cutoff` variable can be used to vary the cutoff that SVD uses to decide what eigenvalues are singular.

8.7 Optimization Troubleshooting Tips

Optimization can be tricky. There are many parameters that affect the optimization and often it comes down to trial and error to find an acceptable an acceptable solution. And even in expert hands, optimizations can take days. The following are some tips if there are problems with an optimization.

Show commands

Commands that can be used to view optimizer parameters are:

```
show constraints
show data
show derivative
show merit
show optimizer
show variable
```

Set the optimizer to run longer

One quick thing to do is to increase the number of optimization loops and/or optimization cycles:

```
set global n_opti_loops = 30
set global n_opti_cycles = 50
```

The `show optimizer` (§11.30.24) command will show global parameters associated with optimizations.

Check merit function and weights

One of the first things to check is the merit function, the top contributors can be seen with the command `show merit` (§11.30.23). And individual contributions can be viewed using the `show variable` and `show data` commands. If the weight of an individual datum is too small, the optimizer will tend to ignore it. So one trick is to raise the weights for datums that are not being well optimized. When running *Tao* this is done with the `data data` command. For example:

```
set data twiss.a[1:2]|weight = 100*twiss.a[1:2]|weight
```

This example will increase the weight of datums `twiss.a[1]` and `twiss.a[2]` by a factor of 100.

Check step size

If using an optimizer that uses the derivative matrix (`lm`, `geodesic_lm` and `svd` optimizers), The variable `step` sizes that are used to calculate the derivative should be checked to make sure that the `step` is not too small so that round-off is a problem but yet not too large so that nonlinearities make the calculation inaccurate. One way to check that the step size is adequate for a given variable is to vary the variable using the command `change var` (§11.3). This command will print out the the change in the merit function per change in variable which can be compared to the derivatives as shown with the `show merit -derivative` (§11.30.23) or the `show derivative` (§11.30.10) command.

Rotate optimizer usage
Problems generally occur when there are many local minima. In this case, the `de` optimizer (§8.6) should be tried. This optimizer has several parameters which will need to be varied by trial and error to find values that are suitable for the problem at hand. Optimization strategies here include doing multiple optimizations one after another using the `de` optimizer every other optimization interlaced with one of the other optimizers. Also varying the merit function weights between optimizations can help guide the optimization process towards an acceptable solution.

Use a non-*Tao* optimizer Using, for example, a Python based optimizer interfaced to *Tao* is possible. There is also the `moga` program which implements multi-objective optimization.

Chapter 9

Wave Analysis

9.1 General Description

A “wave analysis” is method for finding isolated “kick errors” in a machine by analyzing the appropriate data. Types of data that can be analyzed and the associated error type is shown in Table 9.1.

The analysis works on difference quantities. For example, the difference between measurement and theory or the difference between two measurements, etc. Orbit and vertical dispersion measurements are the exception here since an analysis of, say, just an orbit measurement can be considered to be the difference between the measurement and a perfectly flat (zero) orbit.

Measurement Type	*Error Type*
`Orbit`	`Steering errors`
`Betatron phase differences`	`Quadrupolar errors`
`Beta function differences`	`Quadrupolar errors`
`Coupling`	`Skew quadrupolar errors`
`Dispersion differences`	`Sextupole errors`

Table 9.1: Types of measurements that can be used in a wave analysis and the types of errors that can be diagnosed.

The formulation of the wave analysis for quadrupolar and skew quadrupolar errors is presented by Sagan[Sag00b]. Although not discussed in the paper, the wave analysis for orbit and dispersion measurements is similar to analysis in the paper.

The wave analysis is similar for all the measurement types. How the wave analysis works is illustrated in Figure 9.1. Figure 9.1a shows the difference between `model` and `design` values for the a-mode betatron phase for the Cornell’s Cesr storage ring. In this example, one quadrupole in the model has been varied from it’s design value. The horizontal axis is the detector index.

For the wave analysis, two regions of the machine, labeled A and B in the figure, are chosen (more on this later). For each region in turn, the data in that region is fit using a functional form that assumes that there are no kick errors in the regions. For phase differences, this functional form is

$$\delta\phi(s) = D \sin(2\,\phi(s) + \phi_0) + C \tag{9.1}$$

where ϕ is the phase advance and the quantities C, D and ϕ_0 are varied to give the best fit. Once C, D,

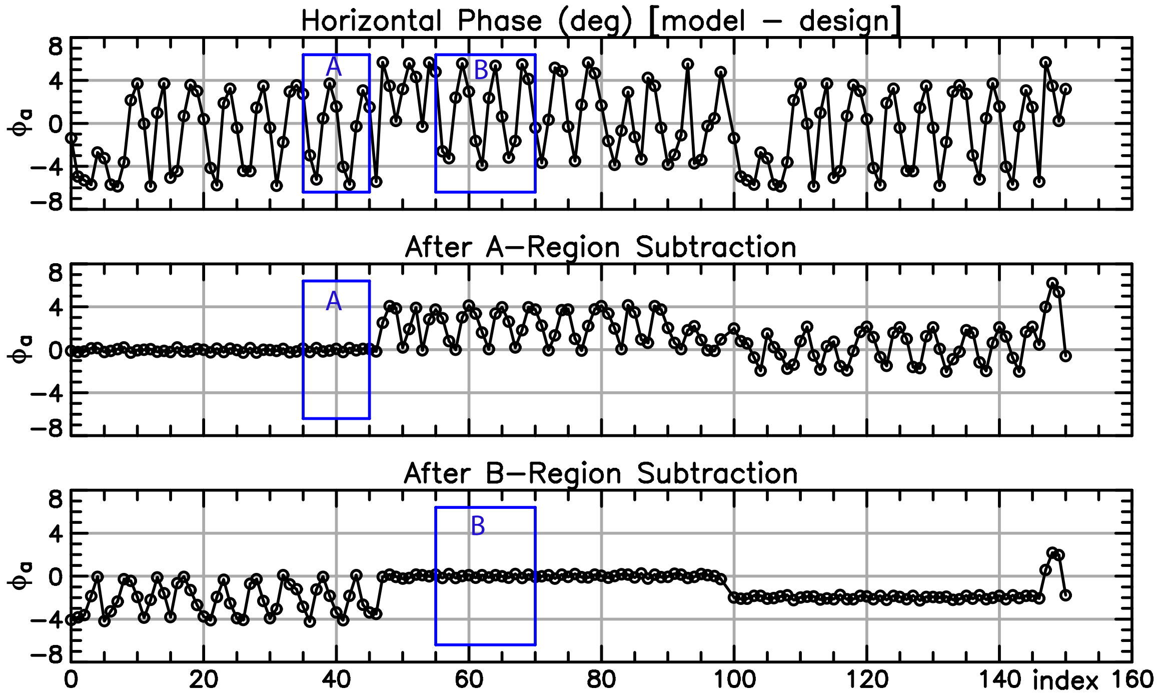


Figure 9.1: Example wave analysis for betatron phase data.

and ϕ_0 are fixed, Eq. (9.1) can be evaluated at any point. Figure 9.1b shows the orbit of 9.1a with the fit to the A region subtracted off. Similarly, Figure 9.1c shows the orbit of Figure 9.1a with the fit to the B region subtracted off. Concentrating on Figure 9.1b, since there are no kick errors in the A region, the fit is very good and hence the difference between the data and the fit is nearly zero. Moving to the right from the A region in Figure 9.1b, this difference is nearly zero up to where the assumption of no kick errors is violated. That is, at the location of the quadrupole error near detector 47. Similarly, since there are no kick errors in region B, the difference between the data and the B region fit is nearly zero in Figure 9.1c and this remains true moving leftward from region B up to the quadrupole near detector 47.

By taking the fitted values for C, D, and ϕ_0 for the regions A and B, the point between the regions where the kick is generated and the amplitude of the kick can be calculated. This calculation is similar to that used to find quadrupolar errors from beta data instead of phase data. The one difference is a factor of 2 that appears in the beta calculation due to the fact that a freely propagating beta wave oscillates at $2\phi(s)$.

The success of the wave analysis in finding a kick error depends upon whether there are regions of sufficient size on both sides of the kick that are kick error free. That is, whether the kick error is "isolated". The locations of the A and B regions are set by the user and the general strategy is to try to find, by varying the location of the regions, locations where the data is well fit within the regions. The data is well fit if the difference between data and fit is small compared to the data itself. If there are multiple isolated kick errors, then each error in turn can be bracketed and analyzed. If there are multiple errors so close together that they cannot be resolved, this will throw off the analysis, but it may still be possible to give bounds for the location where the kicks are at and an "effective" kick amplitude can be calculated.

For circular machines, to be able to analyze kicks near the beginning or end of the lattice, the wave analysis can be done by “wrapping” the data past the end of the lattice for another 1/2 turn. This is illustrated in Figure 9.1. In the Cesr machine, there are approximately 100 detectors labeled from 0 to 99. The detectors from 100 to 150 are just the detectors from 0 to 50 shifted by 100. Thus, for example, the detector labeled 132 in the figure is actually detector 32.

9.2 Wave Analysis in Tao

Performing a wave analysis in *Tao* is a three step process:

```
  1) Plot the data to be analyzed.
  2) Use the wave command to select the data.
  3) Use the set wave command to vary the fit regions.
```

In general, the accuracy of the wave analysis depends upon the accuracy with which the beta function and phase advances are known in the baseline lattice used. *Tao* uses the `model` lattice for the baseline. If possible, One strategy to improve the accuracy of the wave analysis is first use a measurement to calculate what the quadrupole strengths in the `model` lattice should be. Possible measurements that can give this information include an orbit response matrix (ORM) analysis, fits to beta or betatron phase measurements, etc.

9.3 Preparing the Data

At present (due to limited manpower to do the coding), the wave analysis is restricted to data that is stored in a `d1_data` array (§6). That is, the plotted curve to be analyzed must have its `data_type` parameter set to `‘‘data’’` (§10.10). The possible data types that can be analyzed are:

```
  orbit.x, orbit.y
  beta.a,  beta.b
  phase.a, phase.b
  eta.x, eta.y
  cbar.11, cbar.12, cbar.21      ! Analysis not possible for cbar.21
  ping_a.amp_x, ping_a.phase_x
  ping_a.sin_y, ping_a.cos_y
  ping_b.amp_y, ping_b.phase_y
  ping_b.sin_x, ping_b.cos_x
```

The curve to be analyzed must be visible. Any combination of data components may be used:. "meas", "meas-ref", "model", etc.

If data from a circular machine is being analyzed, the data is wrapped past the end of the lattice for another 1/2 turn. The translation from the data index in the wrapped section to the first 1/2 section of the lattice is determined by the values of `ix_min_data` and `ix_max_data` of the `d1_data` array under consideration (§10.10):

```
  index_wrap ⟶ index_wrap - (ix_max_data - ix_min_data + 1)
```

For example, for the Cesr example in the previous section, `ix_min_data` was 0 and `ix_max_data` was 99 to the translation was

```
  index_wrap ⟶ index_wrap - 100
```

9.4 Wave Analysis Commands and Output

The `wave` command (§11.38) sets which plotted data curve is used for the wave analysis. The `set wave` command (§11.29) is used for setting the *A* and *B* region locations. Finally the `show wave` command

(§11.30) prints analysis results.

Example wave analysis output with `show wave`:

```
ix_a:  35  45
ix_b:  55  70
A Region Sigma_Fit/Amp_Fit:     0.018
B Region Sigma_Fit/Amp_Fit:     0.015
Sigma_Kick/Kick:    0.013
Sigma_phi:          0.019
Chi_C:              0.037 [Figure of Merit]

Normalized Kick = k * l * beta [dimensionless]
   where k = quadrupole gradient [rad/m^2].
After Dat#     Norm_K       phi
       46      0.0705    30.431
       49      0.0705    33.573
       53      0.0705    36.715
```

This output is for analysis of betatron phase data but the output for other types of data is similar. The first two lines of the output show where the A and B regions are. The next two lines show σ_a/A_a and σ_b/A_b where σ_a and σ_b are given by Eq. (42) of Sagan[Sag00b] and

$$A_a \equiv \sqrt{\xi_a^2 + \eta_a^2} \tag{9.2}$$

with a similar equation for A_b. σ_a/A_a and σ_b/A_b are thus a measure of how well the data is fit in the A and B regions with a value of zero being a perfect fit and a value of one indicating a poor fit. Notice that a poor fit of one of the regions may simply be a reflection that the wave amplitude being there. The next three lines of the output are $\sigma_{\delta k}/\delta k$, σ_ϕ, and ξ_C, and are given by Eq. (39), (43), and (44) respectively of [Sag00b]. The last three lines of the analysis tell where the wave analysis predicts the kicks are and what the normalized kick amplitudes are. Thus the first of these three lines indicates that the kick may be somewhere after the location of datum #46 (but before the location of datum #47), The normalized quadrupole kick amplitude is 0.0705, and the betatron phase at the putative kick is 30.431 radians.

Chapter 10

Initialization

Tao is customized for specific machines and specific calculations using input files and custom software routines. Writing custom software is covered in the programmer's guide section. This chapter covers the input files.

In general, the input files tell *Tao*:

```
  * What Bmad lattice or lattices to use (§10.4).
  * What the variables and data should be when running optimizations (§8).
  * What to plot and how plots should be laid out in the plotting window (§10.13).
  * What kind of calculations are to be done. EG: a dynamic aperture calculation, etc.
  * Etc.
```

Example initialization files can be found in the *Tao* distribution in sub-directories of the directory:

```
  bmad-doc/tao_examples
```

10.1 Tao Initialization Command Line Arguments

`OpenMP` is a standard that enables programs to run calculations with multiple threads which will reduce computation time. Certain calculations done by *Tao*, including beam tracking and dynamic aperture calculations, can be run multithreaded via OpenMP if the *Tao* executable file has been properly compiled. See §2.3 for more details.

The syntax of the command line for starting *Tao* is:

```
  EXE-DIRECTORY/tao {OPTIONS}
```

where `EXE-DIRECTORY` is the directory where the tao executable lives. If this directory is listed in your `PATH` environment variable then the directory specification may be omitted.

The optional arguments, which will always supersede equivalent parameters set in an initialization file are:

-beam_file <file_name>
: Sets the name of the file containing the `tao_beam_init` namelist (§10.7). Overrides the setting of `beam_file` (§10.3) specified in the *Tao* initialization file.

-beam_init_position_file <file_name>
: Specifies the file containing initial particle positions. Overrides the setting of `beam_init%position_file` (§10.7) specified in the `tao_beam_init` namelist.

-building_wall_file <file_name>
Overrides the `building_wall_file` (§10.3) specified in the *Tao* initialization file.

-command <command_list>
List of commands to run at startup. This will be in addition to the commands run by the startup file (§10.3). The startup file commands will be run before the commands specified by `-command`. Put the `<command_list>` in quotes in order to embed blanks or if semicolons are used to separate multiple commands. The `-command` option is useful when running *Tao* from a script. Example:

```
tao -command "show lat 12:14; quit"
```

In this example *Tao* will print some information on lattice elements 12 through 14 and then quit. The output of the `show` command can be captured by a script and processed.

-data_file <file_name>
Overrides the `data_file` (§10.3) specified in the *Tao* initialization file.

-disable_smooth_line_calc
Disable computation of the "smooth curves" used in plotting. This can be used to speed up *Tao* as discussed in §7.6.3.

-external_plotting
This tells *Tao* that plotting is done externally to *Tao*. This is done, for example, when using a Graphics User Interface (GUI) (§13.3).

-geometry <width>x<height>
Overrides the plot window geometry. `<width>` and `<height>` are in Points. This is equivalent to setting `plot_page%size` in the `tao_plot_page` namelist §10.13. Also then environmental variable `ACC_DPI_RESOLUTION` (§7.1) can be used to vary the window size.

-hook_init_file
Specifies an input file for customized versions of Tao. Default file name is `tao_hook.init`.

-init_file <file_name>
Replaces the default *Tao* initialization file name (`tao.init`). Note: A *Tao* initialization file is actually not needed. If no *Tao* initialization file is used, the use of the `-lattice_file` switch is mandatory and *Tao* will use a set of default plot templates for plotting.

-lattice_file <file_name>
Overrides the `design_lattice` (§10.4) lattice file specified in the *Tao* initialization file (§10.4). Example:

```
tao -init my.init -lat slac.bmad
```

If there is more than one universe with different lattices, separate the different lattice file names using a "|" character. The general form is

```
tao -lat <file_name_uni1><file_name_uni2>...<file_name_uniN>
```

Do not put any spaces in between. Example:

```
tao -lat slac.bmad|cesr.bmad
```

In this example, `slac.bmad` would be used as the lattice file for universe 1 and `cesr.bmad` would be used as the lattice file for universe 2

If secondary lattice files (§10.4) are to be specified, separate the primary lattice file from the secondary ones using commas (and no spaces). Example:

```
tao -lat primary.bmad,secondary.bmad,another_secondary.bmad
```

The `file_name` can encode what line to use (instead of the using the line specified in the last `use` statement in the lattice file) and whether to read the digested (binary) lattice file or the normal ASCII lattice file. The syntax for `<file_name>` is

```
{<parser>::}<lattice_file>{@<use_line1>@<use_line2>...@<use_lineN>}
```

See section §10.4 for details.

-log_startup
: If there is a problem with starting *Tao*, `-log_startup` can be used to create a log file of the initialization process.

-no_stopping
: For debugging purposes. Prevents *Tao* from stopping where there is a fatal error.

-noinit
: Suppresses use of a *Tao* initialization file. In this case the use of the `-lattice_file` switch is mandatory and *Tao* will use a set of default plot templates for plotting.

-noplot
: Suppresses the opening of the plot window.

-nostartup
: Suppresses the use of a startup file.

-no_rad_int
: Suppresses the radiation integrals calculation. Radiation integrals are used to calculate such things as emittances, etc. Generally the calculation is not a problem but in some special circumstances the calculation can take appreciable time.

-plot_file <file_name>
: Overrides the `plot_file` (§10.3) specified in the *Tao* initialization file.

-prompt_color
: Sets the prompt string color to Blue. For different colors, use the `set global prompt_color` command (§11.29).

-quiet <level>
: Suppress warning messages. For details, see the documentation for the `call` command (§11.2).

-reverse
: Reverses the order of the lattice elements. Equivalent to setting `design_lattice(N)%reverse_lattice` to True (§10.4). If both `-reverse` and `design_lattice(N)%reverse_lattice` are set, they negate each other and the lattice will not be reversed.

-rf_on
: Leaves `rfcavity` elements on. RF on is currently the default so using `-rf_on` will not do anything. To turn the cavities off, use the negation (see below) `--rf_on`. Note: If you want to see orbit changes with RF frequency changes then you will need to set `parameter[absolute_time_tracking]` to True. See the “Relative Versus Absolute Time Tracking” section in the*Bmad* manual for more details.

-slice_lattice <element_list>
: If present, discard from the lattice all lattice elements that are not in the `<element_list>`. Overrides the setting of `design_lattice(N)%slice_lattice` (§10.4). Note: A `slice_lattice` command may also be put directly in the lattice file. See the *Bmad* manual for more documentation.

-start_branch_at <element>
: Overrides the setting of `design_lattice(N)%start_branch_at` (§10.4). If present, shift the starting point of a lattice branch while keeping the relative order of the elements the same. Elements that are, before the shift, before the starting element are shifted to the end of the branch. See the *Bmad* manual for more documentation. This is useful, for example, in storage rings. Note A `start_branch_at` command may also be put directly in the lattice file.

-startup_file <file_name>
: Overrides the `startup_file` (§10.3) specified in the *Tao* initialization file.

-symbol_import
: Import symbolic constants defined in any lattice files? (the default is not to). Symbols are imported lower cased. Also see `global%symbol_import` (§10.6.1) for more details.

-var_file <file_name>
: Overrides the `var_file` (§10.3) specified in the *Tao* initialization file.

To negate an argument, use a two dash prefix instead of a single dash prefix. For example:

```
tao -noplot --noplot
```

The `-noplot` argument turns off plotting and the following `--noplot` argument negates the effect of `-noplot` and turns plotting back on. This is useful with the `reinit tao` command (§11.25) to negate saved command line argument settings. Also `--rf_on` is used to turn off the RF.

10.2 Namelist Syntax

Parameters are read in from an initialization file using Fortran namelist input. Fortran namelist breaks up the input file into blocks. The first line of a namelist block starts with an ampersand "&" followed by the block identifying name. Variables are assigned using an equal sign "=" and the end of the block is denoted by a slash "/" For example:

```
&namelist_block_name
  var1 = 0.123   ! exclamation marks are used for comments
  var2 = 0.456
/
```

Variables that have default values can be omitted from the block. The order of the variables inside a block is irrelevant except if the same variable appears twice in which case the last occurrence is determinative. In between namelist blocks all text is ignored. Inside a block comments may be included by using an exclamation mark "!".

Care must be taken when setting arrays in a namelist as the following example shows:

```
&some_namelist_name
  var_array(8:11) = 34             ! Only sets var_array(8)
  var_array(8:11) = 34 34 81 81    ! OK. Sets all 4 values
  var_array(8:11) = 34, 34, 81, 81 ! OK. Same as above
  var_array(8:11) = 34, 34,        ! Lines may be continued ...
                    81, 81         !   ... like this.
  var_array(8:11) = 2*34 2*81      ! Equivalent to the preceding examples
  var_array(8:)   = 2*34 2*81      ! Also equivalent
  var_array(1:2) = 1 2 3           ! Error: Too many RHS values.
  string_arr = '1st' "2nd" '3rd'   ! Setting a string array.
  string_arr(1:3) = 1st 2nd 3rd    ! Same as above. [Not accepted by all compilers.]
  string_arr(1:3) = 1st,2nd,3rd    ! Same as above. [Not accepted by all compilers.]
  string_arr = 'A B' "2/" "&"      ! Quotes needed here.
/
```

The first line to set the `var_array` may look like it is setting the four values `var_array(8:11)` but the general rule is that with `n` values on the RHS, only `n` values in the array are set.

IMPORTANT: The notation `n*number` does not denote multiplication but instead can be used to denote multiple values. There should be no blank spaces here. Some compilers may accept something like "2 * 34" but you cannot count on it. Using "2*34" is safe. Also the GFortran compiler has a known repeat count bug.

For string input it is always best to use quotes. Some compilers will accept strings without quotes. Even those that do will generally not accept strings with special characters. Thus the following characters should not be used in unquoted strings:

```
  Blank or Tab character.
  Period if it is the first character in the string.
  &   ,   /   !   %   *   (   )   =   ?   '   "
```

Note: While there are exceptions, in general *Tao* string variables are case sensitive.

WARNING: Namelists cannot do expression evaluation. Thus the following will not work

```
  &some_namelist_name
    a = 3.7/148
    b = 5
  /
```

The slash in the intended expression "3.7/148" will be taken as the namelist terminator. This will result in variable `a` having the value 3.7 and the value of variable `b` will not be set!

WARNING: Currently there is a bug in the GCC/GFortran compiler up to version 9 (GCC Bugzilla #82086) where repeat counts used with structure components cause *Tao* to halt with an error message. For example:

```
  &tao_template_graph
    curve(1:3)%y_axis_scale_factor = 3*1e3  ! Will not work with gfortran!!!
  /
```

Here `curve` is a structure and `y_axis_scale_factor` is a component of that structure. The work around here is to eliminate the repeat count:

```
  &tao_template_graph
    curve(1:3)%y_axis_scale_factor = 1e3, 1e3, 1e3
  /
```

Logical variables should be set to `T` or `TRUE` when true and `F` or `FALSE` when false. This is case insensitive. It is possible to use the words `.true.` and `.false.` for logicals, however this may not always work. The reason for this is that a variable that is documented to be a logical may actually be a string variable! In this case a beginning period will cause problems. Why use string variables? String variables are used in place of logical variables when *Tao* needs to know if the variable has been explicitly set.

When setting an array in a namelist where the array components are a structure, the set can be structured in several ways. To make this clear, consider the `ele_shape(:)` array that can be set in the `lat_layout_drawing` namelist as explained in §10.13.9. Each component of the `ele_shape(:)` array is a structure and the elements of this structure are:

```
  ele_shape(N) = "<ele_id>" "<shape>" "<color>" "<size>" "<label>" <draw> <multi> <line_width>
```

Setting a given `ele_shape(:)` array component looks like:

```
  &lat_layout_drawing
    !              ele_id                  Shape     Color    Size  Label  ..etc..
    ele_shape(2) = "quadrupole::*"         "xbox"    "red"    0.75  "none"
  /
```

This sets the `ele_id` component of `ele_shape(2)` to `"quadrupole::*"`, etc.

Alternatively, a given structure component can be set for multipole array components. Example:

```
  &lat_layout_drawing
    ele_shape(5:6)%line_width = 5, 6
    ele_shape(3)%multi = T
  /
```

Here the `line_width` structure component for `ele_shape(5)` and `ele_shape(6)` is set along with the `multi` structure component for `ele_shape(3)`.

10.3 Beginning Initialization

The initialization starts with the `root` *Tao* initialization file. The default name for this file is `tao.init` but this default may be overridden when *Tao* is started using the `-init_file` switch (§10.1). The first namelist block read in from the root initialization file is a `tao_start` namelist. This block is optional (in which case the defaults are used). This namelist contains the variables:

```
  &tao_start
    beam_file          = "<file_name>"  ! Default = Tao root init file.
    building_wall_file = "<file_name>"  ! No Default.
    data_file          = "<file_name>"  ! Default = Tao root init file.
    var_file           = "<file_name>"  ! Default = Tao root init file.
    plot_file          = "<file_name1> {<file_name2>} ..."
                                        ! Default = Tao root init file.
    single_mode_file   = "<file_name>"  ! Default = Tao root init file.
    startup_file       = "<file_name>"  ! Default = "tao.startup"
    hook_init_file     = "<file_name>"  ! Default = "tao_hook.init"
    init_name          = "<init_name>"  ! Default = "Tao"
  /
```

Rule: A file name obtained from the *Tao* root initialization file (as opposed to being present on the command line) is always relative to the directory that the *Tao* root initialization file lives in. Example: If *Tao* is started from the system command line like:

```
    tao -data data.cl -init ../tao.init
```

And if the `tao_start` namelist in `../tao.init` looks like:

```
  &tao_start
    data_file = "dat.in"
    plot_file = "plot.in"
    var_file  = "/nfs/var.in"
  /
```

Then, relative to the current working directory, the files used will be

```
  data_file: "data.cl"      ! Command line arguments have preference
  plot_file: "../plot.in"   ! Relative to ../tao.init.
  var_file:  "/nfs/var.in"  ! Absolute paths are never modified.
```

`init_name` is for naming the initialization. This is useful to distinguish between multiple initialization files with custom versions of *Tao*. The other parameters specify which files to find the other initialization namelists. The `plot_file` variable can be an array of plot files.

Tao will open an execute a command file (§2.6) at startup if it exists. The default name is `tao.startup` but this name can be changed by setting the `startup_file` component in the `tao_start` namelist.

The following sections describe each of these initialization namelists and their locations are listed in table 10.1. Note: If `plot_file` specifies multiple files, the `tao_plot_page`, `lat_layout_drawing` and `floor_plan_drawing` namelists are taken from the first file on the list. All files, however, can contain `tao_template_plot` and `tao_template_graph` namelists.

10.4 Lattice Initialization

In the `tao_start` namelist (§10.3), the `lattice_file` variable gives the name of the file that contains the `tao_design_lattice` namelist. The default, if `lattice_file` is not present is to look in the *Tao* root initialization file. The `tao_design_lattice` namelist defines where the lattice input files are. The variables that are set in the `tao_design_lattice` namelist are:

Namelist	*Type of Parameters Initialized*	*Section*
`lat_layout_drawing`	`Plotting`	§10.13.9
`floor_plan_drawing`	`Plotting`	§10.13.9
`tao_beam_init`	`Particle beams`	§10.7
`building_wall_section`	`Building Walls`	§10.11
`symbolic_number`	`Symbolic Number`	§10.5
`tao_design_lattice`	`Lattice Files`	§10.4
`tao_d1_data`	`Data`	§10.10
`tao_d2_data`	`Data`	§10.10
`tao_dynamic_aperture`	`Dynamic Aperture`	§10.12
`tao_params`	`Global Parameters`	§10.6
`tao_plot_page`	`Plotting`	§10.13
`tao_template_graph`	`Plotting`	§10.13
`tao_template_plot`	`Plotting`	§10.13
`tao_var`	`Variables`	§10.9

Table 10.1: Table of `tao` Initialization Namelists.

```
&tao_design_lattice
  n_universes        = <integer>      ! Number of universes. Default = 1.
  unique_name_suffix = "<string>"
  combine_consecutive_elements_of_like_name = <logical>
  design_lattice(N) = "<lattice_file>", {"<lattice2_files>"}
  design_lattice(N)%one_turn_map_calc     = <logical>      ! Default = False
  design_lattice(N)%dynamic_aperture_calc = <logical>      ! Default = False
  design_lattice(N)%reverse_lattice       = <logical>      ! Default = False
  design_lattice(N)%slice_lattice         = "<element_list>"
  design_lattice(N)%start_branch_at =     = "<element>"
/
```

`n_universes` is the number of universes to be created. The default is 1. `design_lattice(N)` gives the lattice file name for universe `i`.

The syntax for `<lattice_file>` is:

```
{<parser>::}<lattice_file>{@<use_line1>@<use_line2>...@<use_lineN>}
```

Possible choices for the <parser> are:

```
bmad      ! For a standard ASCII Bmad lattice file. This is the default.
digested  ! For a digested Bmad file.
```

The `@<use_line1>@<use_line2>...@<use_lineN>` optional suffix is used to specify what `line` or lines in the lattice file to use as a basis for constructing the lattice. This overrides the `use` statement in the lattice file. Example:

```
design_lattice(1) = "cesr.bmad@ln1@ln2"
```

is equivalent to putting `use ln1,ln2` as the use statement in the `cesr.bmad` lattice file.

Note: If the `lattice_file` parameter is not set for the Nth universe, the parameters for the previous universe are used.

The optional `<lattice2_file>` specifies "secondary" lattice files that are parsed after the "primary" lattice file specified by `lattice_file`. The secondary lattice files must only have statements that are valid post lattice expansion. See the *Bmad* manual manual for a discussion of lattice expansion. If there is more than one secondary file, separate them using commas. Note: If a `%slice_lattice` parameter is used with a secondary lattice file then the paring specified by `%slice_lattice` is applied before the secondary lattice file is parsed.

If the `%reverse_lattice` logical is present, the lattice will be reversed. That is, the elements will be in reversed order. The sign of the tracked particle will be set to the anti-particle for proper tracking. This is useful for simulating beams that go in the backward direction. Note: If there are any electric fields, the orbit in the reversed lattice will not be the reverse of the trajectory in the unreversed lattice. Currently, lattice reversal only works if the lattice has a single branch. Lattice reversal can also be done using the `-reverse` option at startup (§10.1).

The `%slice_lattice` parameter, if set, specifies a list of elements to be used to pare down the lattice so that the only elements that appear in the list are kept in the lattice. In addition, any lord elements that control elements in the list are also retained. This is identical to putting a `slice_lattice` command directly in the lattice file. For example:

```
  design_lattice(1)%slice_lattice = "Q1:35"
```

In this example, everything outside of the range from element `Q1` to the element with index 35 will be discarded. See the *Bmad* manual for more details about the `slice_lattice` command. Note: There is also a `-slice_lattice` initialization argument (§10.1) that can be used.

The `%start_branch_at` parameter, if set, shifts the starting point of a lattice branch while keeping the relative order of the elements the same.

Example:

```
  &tao_design_lattice
    n_universe = 4
    design_lattice(1) = "this.lat"               ! Default: Bmad format lattice file.
    design_lattice(1)%slice_lattice = "Q1:Q2"    ! Discard element outside range [Q1:Q2]
    design_lattice(2) = "that.lat", "floor_coords.bmad"  ! For universe #2
    design_lattice(3) = "third.lat@my_line"      ! Specify a different line.
    design_lattice(3)%one_turn_map_calc = True   ! Calculate higher order maps.
  /
```

In this example, the lattice of universe 1 is given by the file `this.lat` and the lattice of universe 2 is given by the file `that.lat`. `design_lattice(2)` in the example also specifies a "secondary lattice file" called `floor_coords.bmad` which will be parsed after the "primary" `that.lat` file is read.

If there is no `design_lattice` specified for a given universe then the last `design_lattice` is used. Thus, in the above example, universes 4 use the same lattice as universe 3.

The `design_lattice(N)%one_turn_map_calc` sets whether a one-turn-map calculation for a ring using PTC will be done. If the calculation is made, the `normal.` and `chrom_ptc.` data types are populated. See Eq. 6.14 and Eq. 6.15. After startup, the map calculation can be toggled on/off by using the `set universe one_turn_map_calc` command (§11.29).

The `design_lattice(N)%dynamic_aperture` component sets whether the dynamic aperture calculation (§10.12) will be done. After startup, this calculation can be toggled on/off by using the `set universe dynamic_aperture_calc` command (§11.29).

Normally, a lattice file will specify which "line" will be used to specify the lattice. Occasionally, it is convenient to override this specification and to use a different line. To do this in *Tao*, the name of the line to be used to specify the lattice can be appended to the lattice file name. Thus, in the example above, universe 3 will have the lattice specified by the line "my_line" from the lattice "third.lat".

`global%combine_consecutive_elements_of_like_name` takes a lattice and combines all pairs of consecutive elements that have the same name and attributes. Why is this useful? Some programs, not based on *Bmad*, cannot generate the Twiss parameters inside the element. If the Twiss parameters at the center of an element are desired, a lattice where the element has been split into two identical pieces is needed. This, however, makes tasks like setting up lattice optimization cumbersome. Note: The recombination of like elements happens when the lattice is read in during initialization.

unique_name_suffix is used to append a unique character string to element names that are not unique. unique_name_suffix uses element list format (§4.3). The class is used to restrict which elements can have their names changed. The name part is used as a suffix. This suffix must have a single ''?'' character. When this suffix is applied to an element's name, a unique integer is inserted in place of the ''?''. For example, if unique_name_suffix is "quad::_?", and if the following quadrupoles are in the lattice:

```
QA    QB    QX    QA    QB     QB
```

then after initialization, the names will be:

```
QA_1  QB_1  QX    QA_2  QB_2   QB_3
```

Setting aperture_limit_on to False will turn off the aperture limits set in all lattices. This overrides the setting of parameter[aperture_limit_on] in a lattice file.

10.5 Symbolic Numbers

Symbolic numbers may be defined in the root initialization file using the symbolic_number namelist. Example:

```
&symbolic_number aaa = 37 /
&symbolic_number my_const = 17 * pi /
```

There may be multiple symbolic_number namelists and each namelist defines one and only one symbolic number. In this example, there are two namelists defining two numbers aaa and my_const.

The parsing of the symbolic_number namelist is handled differently from other namelists and here the value of a symbolic number may be an expression. This is unlike any other namelist in *Tao* where expressions will generate an error. Expressions for symbolic numbers are immediately evaluated. Note: Division can be used in expressions except that a division slash "/" may not be at the end of a line when an expression encompasses more than one line.

Once defined, symbolic constants may be used in expressions embedded in strings. For example:

```
&tao_d1_data
  ...
  datum(1)%data_type = "expression: my_const * data::beta.a"
  ...
/
```

Notice that here the "value" of datum(1)%data_type is a string which will be evaluated after the namelist is parsed.

Besides setting symbolic numbers in the main initialization file, symbolic numbers can be defined using the set symbolic_number command (§11.29.26) and a list of symbolic numbers can be printed using the show symbolic_number command (§11.30.32).

10.6 Global Parameter Initialization

Global parameters are grouped into a number of structures. Four global structures are of interest here:

Instance Name	*Structure*	*Notes*	
`global`	`tao_global_struct`	`Tao global parameters`	§10.6.1
`bmad_com`	`bmad_common_struct`	`Bmad global parameters`	§10.6.2
`space_charge_com`	`space_charge_common_struct`	`CSR global parameters`	§10.6.3
`opti_de_param`	`opti_de_param_struct`	`DE optimizer parameters`	§10.6.4

These instances are initialized in the root initialization file using a namelist named `tao_params`. Example:

```
&tao_params
  global%optimizer = "lm"                 ! Set the default optimizer.
  bmad_com%radiation_damping_on = True   ! Include radiation damping when tracking.
/
```

The `show global` command (§11.30.15) can be used to show global parameter values. The `set` command (§11.29) can be used to set global parameter values. The `show global` and `show optimizer` (§11.30) commands.

The `tao_params` namelist is read after reading of the lattice file so settings of `bmad_com` and `space_charge_com` structures in the lattice file will be overwritten by settings in the `tao_params` namelist. And settings in any startup command file (§10.3) will supersede everything else.

10.6.1 Tao_global_struct Structure

The `tao_global_struct` structure contains *Tao* global parameters. The components of this structure are:

```
type tao_global_struct:
  beam_dead_cutoff = 0.99             ! Dead particle cutoff for stopping beam tracking.
  de_lm_step_ratio = 1                ! Step sizes between DE and LM optimizers.
  de_var_to_population_factor = 5
  lm_opt_deriv_reinit = -1            ! Derivative matrix cutoff. -1 => ignore this.
  lmdif_eps = 1e-12                   ! Tolerance for lmdif optimizer.
  lmdif_negligible_merit = 1d-30      ! lmdif stops if merit is smaller.
  svd_cutoff = 1e-5                   ! SVD singular value cutoff limit.
  unstable_penalty = 1e-3             ! Used in unstable.lattice datum merit calculation.
  merit_stop_value = 0                ! Value below which an optimizer will stop.
  dmerit_stop_value = 0               ! Fractional change below which an optimizer will stop.
  random_sigma_cutoff = -1            ! Cut-off in sigmas.
  delta_e_chrom = 0                   ! Delta E used from chromaticity calc.
  n_opti_cycles = 20                  ! number of optimization cycles
  n_opti_loops = 1                    ! number of optimization loops
  n_lat_layout_label_rows = 1         ! How many rows with a lat_layout
  datum_err_messages_max = 10         ! Max number of error messages per cycle.
  phase_units = radians$            ! Phase units on output.
  bunch_to_plot = 1                   ! Which bunch to plot
  random_seed = -1                    ! Use system clock by default
  n_top10_merit = 10                  ! Number of top constraints to print.
  random_engine = "pseudo"            ! Random number engine to use
  random_gauss_converter = "exact"    ! Uniform to gauss conversion method
  track_type = "single"               ! "single" or "beam"
  prompt_string = "Tao"
  prompt_color = "DEFAULT"          ! See read_a_line routine for possible settings.
  optimizer     = "de"              ! optimizer to use.
  print_command = "lpr"
  var_out_file  = "var#.out"
  history_file = "~/.history_tao"  ! Command history file.
  beam_timer_on = F                 ! For timing the beam tracking calculation.
  cmd_file_abort_on_error = T       ! Close all open command files when there is an error?
  concatenate_maps = F              ! False => tracking using DA.
  derivative_recalc = T             ! Recalc derivatives before each optimizer loop?
  derivative_uses_design = F        ! Derivative matrix uses the design lattice?
  disable_smooth_line_calc = F      ! Disable the plotting smooth line calc?
  draw_curve_off_scale_warn = T     ! Display warning on graphs when any part of the
                                    !   curve is out-of-bounds
  lat_sigma_calc_uses_emit_from = "lat_or_beam_init
                                    ! Init sigma mat set equal to beam distribution?
  label_lattice_elements = T        ! For lat_layout plots
  label_keys = T                    ! For lat_layout plots
  lattice_calc_on = T               ! Turn on/off beam and single particle calculations.
  only_limit_opt_vars = F           ! Apply limits only if variable is used in optimization?
  opt_with_ref = F                  ! use reference data in optimization?
  opt_with_base = F                 ! use base data in optimization?
  opti_var_write_file = T           ! ``run'' command writes var_out_file
  optimizer_allow_user_abort = T    ! See below.
  optimizer_var_limit_warn = T      ! Warn when vars reach a limit when optimizing?
  plot_on = T                       ! Do plotting?
  quiet = "off"                     ! Suppress informational output to terminal.
  rf_on = F                         ! RF cavities on?
  svd_retreat_on_merit_increase = T
  single_step = F                   ! Single step through a command file?
  stop_on_error = T                 ! For debugging: True -> Tao will not exiting on an error.
  symbol_import = F                 ! Import symbols defined in any lattice files?
  var_limits_on = T                 ! Respect the variable limits?
```

In an initialization file, this structure is set in the `tao_params` namelist (§10.6) using "`global`" as the instance name. All global parameters can be changed from their initial value using the `set` command (§11.29).

`global%beam_dead_cutoff`
: Percentage of dead particles at which beam tracking is stopped.

`global%cmd_file_abort_on_error`
: The default is `True`. With this setting, if there is an error, any open command files (and there can be multiple ones since command files can call other files), are closed if a command in a command file generates an error. If this parameter is set `False`, most errors will not result in any open command files being closed. The exception is if infinite command file recursion is detected.

`global%concatenate_maps`
: When constructing transfer Taylor maps the default method, used with `global%concatenate_maps` set to False, is to use Differential Algebra (DA) to integrate the map from the starting point to the ending point. Alternatively, with `global%concatenate_maps` = True, if an element within the integration region has an associated map, that map is concatenated with the map under construction. This saves time but the potential drawback is a loss of accuracy. Note that a lattice element will only have an associate map if the `tracking_method` or `make_mat6_method` components

of the lattice element are such that a map is needed for tracking (see the *Bmad* manual for more details).

`global%datum_err_messages_max`
: Sets the maximum number of error messages per cycle generated when evaluating all datums. A "cycle", which generally happens after most commands or every optimization cycle, consists of the reevaluation of lattice parameters and subsequent datum evaluations. Limiting the number of error messages is useful when many essentially similar error messages are being generated.

`global%derivative_recalc`
: The `global%derivative_recalc` logical determines whether the derivative matrix is recalculated every optimization loop. The `global%derivative_uses_design` logical determines if the design lattice is used in the derivative matrix calculation instead of the model lattice.

`global%disable_smooth_line_calc`
: The `global%disable_smooth_line_calc` is used to disable computation of the "smooth curves" used in plotting. This can be used to speed up *Tao* as discussed in §7.6.3.

`global%dmerit_stop_value`
: When optimizing, if the fractional change in the merit function over one `loop` (set by `global%n_opti_loops`) is below the value of `global%dmerit_stop_value`, optimization will stop. The default value is zero. Also see `global%merit_stop_value`.

`global%lattice_calc_on`
: `global%lattice_calc_on` controls whether lattice calculations are done when there are changes in the lattice. Lattice calculations include the calculation of orbits, Twiss parameters, beam tracking, etc. This also includes plotting calculations so while `%lattice_calc_on` is set to False, the plot window will not update. This switch is useful in controlling unnecessary calculational overhead. A typical scenario where this switch is used involves first setting `%lattice_calc_on` to `False` (using the `set` command (§11.29)), then executing a set of commands, and finally setting `%lattice_calc_on` back to `True`. This saves some of the calculational overhead that each command generates. Similarly, `global%plot_on` can be toggled to save even more time. Also see the `set universe` command (§11.29.28) for ways to suppress certain types of calculations (for example, calculating the Twiss parameters) that are not needed.

`global%force_plot_data_calc`
: Sometimes it is convenient to have *Tao* calculate plotting curve points even when *Tao* is not doing any plotting (that is, `global%plot_on` = F). For example, when *Tao* is run as a server by a client (such as a graphic user interface) program where the client program is taking care of the plotting but the data to be plotted is calculated by *Tao*. In this case by setting `global%force_plot_data_calc` to True will force *Tao* to always calculate curve data points even when `global%plot_on` = F.

`global%history_file`
: The commands typed in by a user are saved in a "history file" so that they can be recalled using the up-arrow key and eve recalled between run sessions. The default is to save the command history to the file `~/.history_tao`. Sometimes is is convenient to have multiple history files and in this case the setting of `global%history_file` can varied from init file to init file.

`global%lat_sigma_calc_uses_emit_from`
: In parallel to calculating the beam sigma matrix from the beam distribution, *Tao* can calculate the sigma matrix using lattice twiss parameters and transport matrices (§10.8). This calculation also needs the following parameters:

```
a-mode and b-mode emittances
size (sigma) of z and pz phase-space coordinates.
dpz/dz dependence of pz centroid as a function of z.
```

The `%lat_sigma_calc_uses_emit_from` switch determines how these parameters are computed. Possible values are:

```
"lat_or_beam_init"  ! Default. Calculated from the lattice For closed geometry branches and
                    !   from the beam_init structure of open geometry branches.
"beam"              ! Calculated from the beam.
"beam_init"         ! Calculated from the beam_init structure.
```

See §10.8 for more details.

`global%merit_stop_value`
The value of `global%merit_stop_value` establishes a point such that, during optimization, if the merit function falls below that value, the optimization stops. If the value is negative (the default), `global%merit_stop_value` is ignored. Also see `global%dmerit_stop_value`.

`global%opt_with_ref`
Use the `reference` data and variable values in the calculation of the merit function (§8.1)? Default is False.

`global%opt_with_base`
Use the `base` lattice data and variable values in the calculation of the merit function (§8.1)? Default is False.

`global%optimizer_allow_user_abort`
Normally `optimizer_allow_user_abort` defaults to True which allows the optimizer, when it is run, to look for user input from the terminal (§8.6). If the user types a period ".", the optimization is aborted cleanly. However, if *Tao* is started with standard input redirected from a file (using the "<" character) *Tao* will not be able to distinguish between input meant as a *Tao* command and input meant for aborting the optimization. In this case, `optimizer_allow_user_abort` will default to False so that the optimizer will not do any checking.

`global%quiet`
For use with command files. May be set to one of:

```
off       ! Normal verbose output
all       ! Suppress output except errors.
warnings  ! Suppress warnings only.
```

If set to `all`, output to the terminal will be suppressed except for error messages. If set in a command file, the setting will revert to what it was at the end of the command file.

`global%random_engine`
`global%random_engine` selects the algorithm used for generating the random numbers. `"pseudo"` causes *Tao* to use a pseudo-random number generator. `"quasi"` uses Sobel quasi-random number generator which generates a distribution that is smoother then the pseudo-random number generator. `"pseudo"` is the default.

`global%random_gauss_converter`
`global%random_gauss_converter` selects the algorithm used in the conversion from a uniform distribution to a Gaussian distribution. `"exact"` is an exact conversion and `"limited"` has a cut-off so that no particles are generated beyond. This cutoff is set by `global%random_sigma_cutoff`.

`global%random_sigma_cutoff`
See `global%random_gauss_converter`.

`global%random_seed`
`global%random_seed` sets the seed number for the pseudo-random number generator. A value of `-1` (the default) means the seed number used is the number set in the lattice file. Any other number overrides the number set in the lattice file. If the lattice file does not set the random number seed, a value of zero is used. A value of zero means that the the "true" random number seed used in generating random numbers is picked by *Bmad* based upon the system clock. Use the `show global` command to see what the input and true random number seeds are.

`global%rf_on`
: The rf cavities in circular lattices can be be toggled on or off using the `global%rf_on` switch. The default is False. Notice that with the RF off, the beam energy will be independent of the closed orbit which is not the case when the RF is on. Note: If you want to see orbit changes with RF frequency changes then you will need to set `parameter[absolute_time_tracking]` to True. See the "Relative Versus Absolute Time Tracking" section in the*Bmad* manual for more details.

`global%single_step`
: For use with command files. If set True, this is equivalent to putting a "pause -1" after each line in a command file. Useful for debugging or for talk demonstrations.

`global%symbol_import`
: Symbolic constants can be defined in lattice files. These symbols can be imported by setting `%symbol_import` to True (default is False). Alternatively, the `-symbol_import` switch may be used at startup (§10.1). In *Tao*, unlike *Bmad*, symbolic names are case sensitive. Imported symbolic names are lower cased.

  If the value of these symbols is subsequently modified in *Tao*, this will *not* affect the lattice. For example, control functions used in lattice control elements (`group` and `overlay` elements), will not be affected even if a control function expression used a symbolic constant.

`global%track_type`
: The setting of the `global%track_type` parameter can be

```
"single"
"beam"
```

  The `"single"` setting is used when single particle tracking is desired and `"beam"` is used when tracking with a beam of particles. Note that with `"single"` tracking, synchrotron radiation fluctuations (but not damping) is always turned off.

`global%var_limits_on`
: The `global%var_limits_on` switch controls whether a variable's model value is limited by the variable's `high_lim` and `low_lim` settings (§10.9). This is particularly important during optimization. If a variable's model value moves outside of the limits, the value is set at the limit and the variable's `good_user` parameter is set to `False` so it will not be further varied in the optimization.

`global%only_limit_opt_vars`
: The `global%only_limit_opt_vars` switch controls whether only the variables being optimized are limited or whether all variables are limited. The `global%optimizer_var_limit_warn` switch controls whether a warning is printed when a variable value goes past a limit.

`global%var_out_file`
: The `global%var_out_file` sets the name of the file that is written when running an optimizer that stores variable values. The format of the file is such that the file can be used to construct a lattice with the optimized variables. For example, if "`lat.bmad`" is the name of the unoptimized lattice and the name of the variable file is "`v.out`", the following file will can be used for the optimized lattice

```
call, file = lat.bmad  ! Read in original lattice.
call, file = v.out     ! Set optimized values.
```

  The default file name is "`var#.out`". If the file contains a hash ("#") symbol, a separate file will be generated for each universe with the universe index substituted for the hash symbol. For example, with the default file name, the name of the file for universe 1 will be "`var1.out`". If the file name is blank, the results will be printed on the screen and no file will be generated.

`global%opt_match_auto_recalc`
: There are some circumstances where having all `match` elements recalc their transfer map in each optimization cycle is desirable. Setting `global%opt_match_auto_recalc` to True will do this. The default is False.

`global%opti_write_var_file`
: Normally the optimizer writes to `global%var_out_file`. Setting `global%opti_write_var_file` to `F` prevents this.

Random number generation in *Tao* is divided into two categories: Random numbers used for generating the initial coordinates of the particles in a beam and random numbers used for everything else. As explained below, there are four parameters that govern how random numbers are generated. For beam particle generation, three of the four (everything except the random number seed) are accessed through the `beam_init` structure (§10.7). For everything else, these parameters are accessed through the `tao_global_struct`.

10.6.2 bmad_com_struct Structure

The `bmad_com_struct` holds bmad global variables.

```
  type bmad_com_struct:
    real(rp) max_aperture_limit = 1e3
    real(rp) d_orb(6) = 1e-5  ! for the make_mat6_tracking routine
    real(rp) default_ds_save    = 0.2_rp    ! Integration step size.
    real(rp) significant_length = 1e-10     ! meter
    real(rp) rel_tol_tracking = 1e-8
    real(rp) abs_tol_tracking = 1e-10
    real(rp) rel_tol_adaptive_tracking = 1e-8   ! Adaptive tracking relative tolerance.
    real(rp) abs_tol_adaptive_tracking = 1e-10  ! Adaptive tracking absolute tolerance.
    real(rp) init_ds_adaptive_tracking = 1e-3   ! Initial step size
    real(rp) min_ds_adaptive_tracking = 0       ! Min step size to take.
    real(rp) fatal_ds_adaptive_tracking = 1e-8  ! particle lost if step size is below this.
    real(rp) autoscale_amp_abs_tol = 0.1_rp     ! Autoscale absolute amplitude tolerance (eV).
    real(rp) autoscale_amp_rel_tol = 1d-6       ! Autoscale relative amplitude tolerance
    real(rp) autoscale_phase_tol = 1d-5         ! Autoscale phase tolerance.
    real(rp) electric_dipole_moment = 0         ! Particle's EDM.
    real(rp) ptc_cut_factor = 0.006             ! Cut factor for PTC tracking
    real(rp) sad_eps_scale = 5.0d-3             ! Used in sad_mult step length calc.
    real(rp) sad_amp_max = 5.0d-2               ! Used in sad_mult step length calc.

    integer space_charge_mesh_size(3) = [32, 32, 64]  ! Gird size for fft_3d space charge calc.
    integer sad_n_div_max = 1000                ! Used in sad_mult step length calc.
    integer taylor_order = 3                    ! 3rd order is default
    integer runge_kutta_order = 4               ! Runge Kutta order.
    integer default_integ_order = 2             ! PTC integration order.
    integer ptc_max_fringe_order = 2            ! PTC max fringe order (2  = > Quadrupole !).
    integer max_num_runge_kutta_step = 10000    ! Maximum number of RK steps before particle is consider

    logical rf_phase_below_transition_ref = F   ! Autoscale uses below transition stable point for rfcav
    logical sr_wakes_on = T                     ! Short range wakefields?
    logical lr_wakes_on = T                     ! Long range wakefields
    logical mat6_track_symmetric = T            ! symmetric offsets
    logical auto_bookkeeper = T                 ! Automatic bookkeeping?
    logical csr_and_space_charge_on = F         ! Space charge switch
    logical spin_tracking_on = F                ! spin tracking?
    logical backwards_time_tracking_on = F      ! Track backwards in time?
    logical spin_sokolov_ternov_flipping_on = F ! Spin flipping during synchrotron radiation emission?
    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_offse
    logical absolute_time_tracking_default = F  ! Default for lat%absolute_time_tracking
    logical convert_to_kinetic_momentum = F     ! Cancel finite vector potential edge kicks with symplec
    logical aperture_limit_on = T               ! use apertures in tracking?
    logical ptc_print_info_messages = F         ! Allow PTC to print informational messages?
    logical debug = F                           ! Used for code debugging.
```

See the *Bmad* manual for more details.

10.6.3 space_charge_common_struct Structure

The `space_charge_common_struct` holds global variables for space charge, including coherent synchrotron radiation (CSR), calculations.

```
  type space_charge_common_struct
    ds_track_step = 0                   ! Tracking step size
    dt_track_step = 0                   ! Time based space charge step
    beam_chamber_height = 0             ! Used in shielding calculation.
    cathode_strength_cutoff = 0.01      ! Cutoff for the cathode field calc.
    rel_tol_tracking = 1d-8
    abs_tol_tracking = 1d-10
    lsc_sigma_cutoff = 0.1              ! Cutoff for the lsc calc. If a bin sigma
                                        !  is < cutoff * sigma_ave then ignore.
    particle_sigma_cutoff = -1          ! Veto particles that are far from the bench with 3D SC.
    n_bin = 0                           ! Number of bins used
    particle_bin_span = 2               ! Longitudinal particle length / dz_bin
    n_shield_images = 0                 ! Chamber wall shielding. 0 = no shielding.
    sc_min_in_bin = 10                  ! Min number of particle needed to compute sigmas.
    space_charge_mesh_size = [32,32,64] ! Mesh size with fft_3d space charge calc.
    csr_3d_mesh_size = [32,32,64]       ! Mesh size for 3D CSR calc.
    print_taylor_warning = T            ! Print Taylor element warning?
    diagnostic_output_file = ""         ! Wake file name
  end type
```

It is important to note that space charge / CSR calculations also depend upon settings in the `bmad_common_struct` structure as well as individual lattice element parameters. See the *Bmad* manual for more details.

10.6.4 opti_de_param_struct Structure

The `opti_de_param_struct` holds parameters that influence the behavior of the `de` optimizer (§8.6)

```
                         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.
```

See the *Bmad* manual for more details.

If ix1_ele_csr and ix2_ele_csr are set, The effect of coherent synchrotron radiation is only included in tracking in the region from the exit end of the lattice element with index ix1_ele_csr through the exit end of the lattice element with index ix2_ele_csr. By restricting the CSR calculation, the calculational time to track through a lattice is reduced.

See §8.4 for more details on global%n_opti_cycles and global%n_opti_loops.

10.7 Particle Beam Initialization

Beam tracking involves tracking some number of particles through the lattice to gather statistics about the expected distribution of particles in an actual machine.

Beam tracking is started in root lattice branches (a root branch is a branch where no other branch forks to that branch). Beams will be propagated to through fork elements to all downstream branches provided that the downstream branch has the same reference particle as the upstream branch (this prevents, for example, an electron bunch being injected into an X-ray beam line).

The default is single particle tracking. To turn on particle tracking the global%track_type parameter must be set to "beam". This can be placed in the tao_params namelist above, for example,

```
&tao_params
  global%optimizer = "lm"  ! Set the default optimizer.
  global%track_type = "beam"
/
```

Particle beam initialization parameters are set in the tao_beam_init namelist block. The file that *Tao* looks in to find this namelist is set by the beam_file component of the tao_start namelist (§10.3). The default, if beam_file is not set, is the root initialization file. If the beam initial distribution is *not* being read in from a file, *Tao* calculates the beam initial distribution based upon the settings of the beam_init structure (see below) and the local Twiss and orbit values at the position the beam is initialized at. To avoid jitter due to random number fluctuations, recalculation of the beam initial distribution is not automatically done if the beam is tracked multiple times. Rather, recalculation is only done after a reinit beam command (§11.25) or after changes to the beam_init parameters.

The syntax of the tao_beam_init namelist is:

```
&tao_beam_init
  ix_universe                = <integer>     ! Universe to apply to.
  always_reinit              = <logical>     ! Always reinit the particle distribution?
  saved_at                   = "<ele_list>"  ! At what elements to save beam info.
  dump_file                  = "<file_name>" ! File for saving beam info.
  dump_at                    = "<ele_list>"  ! Save beam info at these elements.
  track_start                = "<ele_id>"    ! Beam tracking start element.
  track_end                  = "<ele_id>"    ! Beam tracking end element.
  comb_ds_save               = <real>        ! Step size for beam parameter evaluation.
  beam_init%position_file    = <string>      ! Beam position init file.
  beam_init%distribution_type(3) = "<type>"  ! "ELLIPSE", "KV", "GRID", or
                                             !   "RAN_GAUSS" (default).
  beam_init%ellipse(3)%...   = ...           ! Parameters for an ellipse type distribution.
  beam_init%KV%...           = ...           ! Parameters for a KV distribution
  beam_init%grid(N)%...      = ...           ! Parameters for a grid distribution.
  beam_init%a_norm_emit      = <real>        ! A-mode energy normalized emittance
```

```
    beam_init%b_norm_emit      = <real>        ! B-mode energy normalized emittance
    beam_init%a_emit           = <real>        ! A-mode emittance
    beam_init%b_emit           = <real>        ! B-mode emittance
    beam_init%dPz_dZ           = <real>        ! Energy-Z correlation
    beam_init%center           = <real>*6      ! Bunch center offset relative to
                                               !   reference particle (BMAD coords)
    beam_init%sig_pz           = <real>        ! e_sigma in dE/E0
    beam_init%sig_z            = <real>        ! Z sigma in m
    beam_init%n_bunch          = <integer>     ! Number of bunches
    beam_init%dt_bunch         = <real>        ! Time between bunches (meters)
    beam_init%n_particle       = <real>        ! Number of particles per bunch
    beam_init%bunch_charge     = <real>        ! charge per bunch (Coulombs)
    beam_init%renorm_center    = <logical>     ! Default is T
    beam_init%renorm_sigma     = <logical>     ! Default is F
    beam_init%center_jitter    = <real>*6      ! Bunch center rms jitter (meters)
    beam_init%emit_jitter      = <real>*2      ! Emittance rms jitter (dϵ/ϵ)
    beam_init%sig_z_jitter     = <real>        ! bunch length rms jitter (dz/z)
    beam_init%sig_pz_jitter    = <real>        ! bunch energy spread rms jitter (dE/E)
    beam_init%spin(3)          = <real>*3      ! (x, y, z) spin components.
    beam_init%init_spin        = <logical>     ! Initialize the spin (default: False)
    beam_init%random_engine    = ""            ! random number engine to use
    beam_init%random_gauss_converter = "exact" ! Uniform to gauss conversion method
    beam_init%random_sigma_cutoff = 4.0         ! Cut-off in sigmas.
    beam_init%use_t_coords     = <logical>     ! Use time coords (for e_guns)?
    beam_init%use_z_as_t       = <logical>     ! Use time instead of z (for e_guns)?
  /
```

always_reinit

Tao tracks the beam through the lattice every time a lattice parameter is changed. For example, during optimizations or when the `set` command (§11.29) is used. For the retracking, the default is that the particle distribution at the beginning of the lattice is not recalculated. The exception here is that if the `set beam` (§11.29.1) or `set beam init` (§11.29.2) is used (with the exception of the `set beam beginning` command), the initial beam distribution is automatically reinitialized. To force a new initial beam particle distribution, use the `reinitialize beam` command (§11.25). Also, if the `always_reinit` parameter of the `tao_beam_init` namelist is set to True, the initial distribution is always recalculated.

Not recalculating the initial distribution can be important since, if the initial distribution is constructed with the help of a random number generator,[1] variations of the initial distribution will cause values calculated from the beam to "jitter". This can be especially problematical when doing an optimization as it may hinder finding the merit function minimum.

comb_ds_save

When tracking a beam, the beam parameters (like centroid, beam size matrix, etc) can be saved at equally spaced points by setting `comb_ds_save` to a positive number representing the spacing between points. Default is -1. Any value less than zero means no comb will be calculated. Using a comb can give better detail when plotting beam parameters as a function of s. The comb points are index starting at zero and the actual spacing between points will be adjusted to give an integer number of points over the region of travel.

beam_init

The `beam_init` parameter is an instance of a `beam_init_struct` structure which holds parameters

[1]Note that if the initial distribution is read in from a file, no random number generator is used.

(for example, the beam emittances) from which a distribution of particles can be constructed. Documentation on this can be found in the *Bmad* manual in the `Beam Initialization` chapter. In particular, `beam_init%position_file` if it is non-blank, specifies a file (which can be created with the `write beam -at <ele_name>` command) which contains a beam's particle coordinates which are to be used at the start of the lattice. Note: The file name can be overridden by using the `-beam_init_position_file` argument on the command line (§10.1). The file can either be in binary format (binary files can be created by the `write beam` command), or written in ASCII. Note: When the particle coordinates are read in from the `beam_init%position_file` file, the centroid will be shifted by the setting of `beam_init%center`. To vary the centroid of the beam on the *Tao* command line, the `set beam_init center` command (§11.29) can be used.

The emittances used construct to the beam's particle distribution can be set using the energy normalized emittances `%a_norm_emit` and `%b_norm_emit` or the unnormalized ("geometric") `%a_emit` and `%b_emit`. If not set, the emittances set in the lattice file are used. These emittances are also used as the initial emittance in a linear lattice for the emittance calculation using the radiation integrals.

When `beam_init%position_file` is blank, the Twiss parameters at the beginning of the lattice are used in initializing the beam distribution. For circular lattices the Twiss parameters will be found from the closed orbit, and the emittance will be calculated using the *Bmad* routine `radiation_integrals`.

If spin tracking is desired then `beam_init%init_spin` must be set to true.

The three random number generator parameters (`%random_engine`, `%random_gauss_converter`, and `%random_sigma_cutoff`) used for initializing the beam are set in the `tao_global_struct` (§10.6). They may, however, be overridden for beam particle generation by setting the corresponding parameters in the `beam_init` structure. That is, separate parameters may be setup for beam particle generation verses everything else. These parameters are explained in Section §10.6.

dump_at
See documentation on the `dump_file` parameter below.

dump_file
If the beam size is large or the number of elements at which the beam is to be saved at is large, it may be problematic to store all the beam particle position information in memory until the end of tracking. If this is the case, the beam particle position information can be written directly to a file during tracking (and not saved in memory) by setting `dump_at` to a list of elements at which the position information is to be saved and setting `dump_file` to the name of the data file. The data file should have an ".h5" or ".hdf5" suffix to save the data in HDF5 format. Otherwise, an ASCII file will be produced. The syntax for `dump_at` is the same at `saved_at`. Saving directly to a file using `dump_at` is separate from saving to memory using `saved_at`. Example

```
&tao_beam_init
  dump_at = "marker::m* *34w*" ! Save beam at all markers starting with "m"
                               !  and all elements with "34w" in their name.
  dump_file = "beam_dump.h5"
/
```

ix_universe
Beam initialization parameters can be set on a universe-by-universe basis by having multiple `tao_beam_init` namelists. The universe that the namelist is applied to is set by the `ix_universe` component. If `ix_universe` is not present, or if set to -1, the beam initialization parameters will be applied to all universes. Universes where beam initialization parameters are not set will not have beams tracked through them.

saved_at
`saved_at` is used to specify at what elements the beam particle positions are saved at. Note that,

independent of the setting of `saved_at`, beam statistics (like the beam sigma matrix) are always saved at each lattice element. Element list format, as explained in §4.3, is used to specify a list of elements for `saved_at`. The beam is automatically saved at the beginning position and end position of beam tracking and at `fork` and `photon_fork` elements.

```
  &tao_beam_init
    saved_at = "marker::m* *34w*" ! Save beam at all markers starting with "m"
                                  !   and all elements with "34w" in their name.
  /
```

The elements where the beam is saved may be modified while *Tao* is running by using the `set beam saved_at`, `set beam add_saved_at`, and `set beam subtract_saved_at` commands (§11.29.1). To write the beam particle positions use the `write beam` command (§11.39.1).

track_start, track_end

`track_start` and `track_end` are used when it is desired to only track the beam through part of the root lattice branch. `track_start` gives the starting element name or index. Tracking will start at the exit end of this element so the beam *will not* be tracked through this element. The tracking will end at the exit end of the lattice element with name or index `track_end`. The default, if `track_start` is not given, is to start at the beginning of the branch The default for `track_end` is the end of the root branch if the branch has an open geometry or beam tracking is beginning at the start of the branch. For a root branch with a closed geometry and with the beam starting in the middle, the tracking will wrap around from the branch end to the beginning of the branch and will end up just before the starting point.

After initialization, the `set beam_init` (§11.29.2) command can be used to set `track_start` and `track_end`. Note: Deprecated names for `track_start` and `track_end` are `track_start` and `track_end` respectively.

10.8 Lattice Sigma Matrix Initialization

Tao will calculate a 6×6 "beam sigma matrix" from the lattice Twiss parameters and element transfer matrices. This can be useful for comparisons with the sigma matrix calculated from the distribution of a tracked beam (§10.7) or for fast optimizations (the sigma matrix as calculated from the lattice can be done faster than tracking a beam). The "`show beam -lattice`" command (§11.30.2) will display the lattice derived sigma matrix. For optimizations, the `sigma` data type (pg. [70]) with `data_source` is set to `lat` can be used.

The calculation of the lattice sigma matrix starts at some beginning element where, as detailed below, the sigma matrix is calculated. Once the initial sigma matrix has been calculated, the sigma matrix $\boldsymbol{\sigma}(s)$ at a point downstream is calculated using the standard formula

$$\boldsymbol{\sigma}(s) = \mathbf{M}\,\boldsymbol{\sigma}_0\,\mathbf{M}^t \tag{10.1}$$

where $\mathbf{M}$ is the transfer matrix from the beginning to s and the t exponent means transpose. Note: the calculation ignores radiation effects.

The initial sigma matrix is calculated as follows: If the lattice branch where the sigma matrix is being calculated is being forked to from a "base" branch, the sigma matrix as calculated at the fork element will be used. That is, the sigma matrix is propagated from the base branch to the forked-to branch.

If there is no base branch that is forked from, the initial sigma matrix is determined in one of three ways depending upon the setting of the parameter `global%lat_sigma_calc_uses_emit_from` (§10.6.1). Possible settings of this parameter are:

```
  "lat_or_beam_init"  ! Default. Calculated from the lattice For closed geometry branches and
```

```
                      !   from the beam_init structure of open geometry branches.
  "beam"          ! Calculated from the beam.
  "beam_init"     ! Calculated from the beam_init structure.
```

If `global%lat_sigma_calc_uses_emit_from` is set to `"beam"`, the initial sigma matrix is calculated by the initial beam distribution used with beam tracking (§10.7).

For the other two settings of `global%lat_sigma_calc_uses_emit_from`, the calculation of the initial sigma matrix needs the following parameters:

```
  a-mode and b-mode emittances
  size (sigma) of z and pz phase-space coordinates.
  dpz/dz dependence of pz centroid as a function of z.
```

With the `"beam_init"` setting, the above parameters are taken from the `beam_init` structure (§10.7):

```
  beam_init%a_emit or beam_init%a_norm_emit
  beam_init%b_emit or beam_init%b_norm_emit
  beam_init%sig_z
  beam_init%sig_pz
  beam_init%dpz_dz
```

Notice that the sigma matrix formed using the `beam_init` parameters may be quite different from the sigma matrix formed using the initial beam distribution. For example, if the initial beam distribution is read from a file (done if `beam_init%position_file` is set), there is generally no relationship between the `beam_init` parameters and the initial beam distribution.

The `"lat_or_beam_init"` setting of `global%lat_sigma_calc_uses_emit_from` (which is the default) calculates the emittances and other parameters from radiation integrals for lattice branches that have a closed geometry. For branches that are open, the `beam_init` structure (§10.6.4) will be used.

For everything besides closed branches with the `"lat_or_beam_init"` setting for `global%lat_sigma_calc_uses_emit_from`, the calculation of the sigma matrix starts at the lattice element defined by the `track_start` parameter in the `tao_beam_init` namelist (§10.7). With the `"lat"` setting, the initial sigma matrix is taken to be at the beginning of the lattice branch.

10.9 Variable Initialization

Tao `variables` (§5 are used in lattice correction or design (§8).

The file that *Tao* looks in to find information on *Tao* variables is set by the `var_file` component of the `tao_start` namelist (§10.3). The default, if `data_file` is not set, is the root initialization file.

Variables are initialized using the `tao_var` namelist. The format for this is

```
  &tao_var
    v1_var%name          = "<array_name>"  ! Variable array name.
    use_same_lat_eles_as = "<d1_name>"     ! Reuse a previous element list.
    search_for_lat_eles  = "<ele_list>"    ! Find elements by name.
    default_universe     = "<integer>"     ! Universe variables belong in.
    default_attribute    = "<attrib_name>" ! Attribute to control.
    default_weight       = <real>          ! Merit_function weight. Default: 0.
    default_meas         = <real>          ! Default ‘‘measured’’ value (§8.3).
    default_step         = <real>          ! Small step value. Default: 0.
    default_merit_type   = "<merit_type>"  ! Sets how the merit is calculated.
                                           !   Default = "limit"
    default_low_lim      = <real>          ! Lower var value limit. Default: -1e30
    default_high_lim     = <real>          ! Upper var value limit. Default 1e30
    default_key_delta    = <real>          ! Change when key is pressed.
    default_key_bound    = <logical>       ! Variable  to be bound?
    default_good_user    = <logical>       ! Vary for optimization?
    ix_min_var           = <integer>       ! Minimum array index.
    ix_max_var           = <integer>       ! Maximum array index.
    var(N)%ele_name      = "<ele_name>"    ! Name or index of element to be controlled.
    var(N)%attribute     = "<attrib_name>" ! Attribute to be controlled.
    var(N)%universe      = "<uni_list>"    ! Universe containing parameter to
                                           !    be controlled. "*" => All.
    var(N)%weight        = <real>          ! Merit function weight.
    var(N)%step          = <real>          ! Small step size.
    var(N)%low_lim       = <real>          ! Lower variable value limit
    var(N)%high_lim      = <real>          ! Upper variable value limit
    var(N)%merit_type    = "<merit_type>"  ! Sets how the merit is calculated.
    var(N)%good_user     = <logical>       ! Good optimization variable?
    var(N)%key_bound     = <logical>       ! Variable bound to a key
    var(N)%key_delta     = <real>          ! Change when key is pressed.
    var(N)%meas          = <real>          ! ‘‘Measured’’ value (§8.3).
  /
```

Example:

```
  &tao_var
    v1_var%name        = "v_steer"   ! vertical steerings
    default_universe   = "clone 2,3"
    default_attribute  = "vkick"     ! vertical kick attribute
    default_weight     = 1e3
    default_step       = 1e-5
    var(0:99)%ele_name = "v00w", "v01w", "v02w", "    ", "v04w", ...
    var(2)%attribute   = "hkick"     ! Override default
  /
```

A `tao_var` block is needed for each variable array to be defined. `v1_var%name` is the name of the array to be used with *Tao* commands. The `var(N)` array of variables has an index `i` that runs from `ix_min_var` to `ix_max_var`. If `ix_min_var` and/or `ix_max_var` is not present, *Tao* will choose the range based upon which elements in the array define a valid variable. A lattice element name `var(N)%ele_name` and the element's attribute to vary `var(N)%attribute` needs to specified. Not all elements need to `exist` and the element names of non–existent elements should be undefined or set to a name with only spaces in it. For those variables where `var(N)%attribute` is not specified in the namelist the `default_attribute` will be used.

`var(N)%key_bound` and `var(N)%key_delta` are used to bind variables to keys on the keyboard for use in single mode(§12). The default values for these parameters are set by `default_key_bound` and `default_key_delta`. If not set, `default_key_bound` is set to `False` and `default_key_delta` is set to `0`. See §12.1 for more details.

`var(N)%step` establishes what a "small" variation of the variable is. This is used, for example, by some optimizers when varying variables. If `var%step(N)` is not given for a particular variable then the default `default_step` is used.

`var(N)%good_user` is a logical that the user can toggle when running *Tao* (§5). The initial default value of `%good_user` is set by `default_good_user`. If `default_good_user` is not present, the default is True.

`var(N)%universe` gives the universe that the lattice element lives in. Multiple universes can be specified using a comma delimited list. For example:

```
var(10)%universe = "2, 3"
```

If `var(N)%universe` is not present, or is blank, the value of `default_universe` is used instead. If both `var(N)%universe` and `default_universe` are not present or blank then all universes are assumed. In addition to a number (or numbers), `default_universe` can have values:

```
"gang"     -- Multiple universe control (default).
"clone"    -- Make a var array block for each universe.
```

`"gang"` means that each variable will control the given attribute in each universe simultaneously. `"clone"` means that the array of variables will be duplicated, one for each universe. To differentiate variables from different universes, `_u<n>` will be appended to each `v1_var%name` where `<n>` is the universe number. For example, if `v1_var%name` is `quad_k1` then the variable block name for the first universe will be `quad_k1_u1`, second universe will be `quad_k1_u2`, etc. With `"clone"`, individual `var(N)%universe` may not be set in the namelist. The default if both `default_universe` and all `var(N)%universe` are not given is for `default_universe` to be `"gang"`. Examples:

```
default_universe = "gang"         ! Gang all universes together.
default_universe = "gang 2, 3"    ! Gang universes 2 and 3 together.
default_universe = "2, 3"         ! Same as "gang 2, 3".
default_universe = "clone 2, 3"   ! Make two var arrays.
                                  !   One for universe 2 and one for universe 3.
```

`var(N)%weight` gives the weight coefficient for the contribution of a variable to the merit function. If not present then the default weight of `default_weight` is used. `var(N)%low_lim` and `var(N)%high_lim` give the lower and upper bounds outside of which the value of a variable should not go. If not present `default_low_lim` and `default_high_lim` are used. If these are not present as well then by default

```
low_lim  = -1e30
high_lim =  1e30
```

`var(N)%merit_type` determines how the merit contribution is calculated. Possible values are:

```
"limit"       ! Default
"target"
```

For details on `limit` and `target` constraints see Chapter 8 on Optimization.

If elements in the `var` array do not exist the corresponding `var%ele_name` should be left blank. Lists of names can be reused using the syntax:

```
use_same_lat_eles_as = "<d1_name>"     ! Reuse a previous element list.
```

For example:

```
&tao_var
  v1_var%name      = "quad_tilt"
  default_attribute = "tilt"
  ...
  use_same_lat_eles_as = "quad_k1"
/
```

Instead of specifying a list of lattice element names for `var(:)%ele_name`, *Tao* can be told to search for the elements by name using the syntax:

```
 search_for_lat_eles = "-no_grouping <element_list>"
```

Where `<element_list>` is a list of elements using the element list format (§4.3). The searching will automatically exclude any superposition and multipass slaves elements. If the `-no_grouping` flag is not present, the default behavior is that all matched elements with the same name are grouped under a single variable. That is, a single variable can control multiple elements. On the other hand, if the `-no_grouping` flag is present, each element will be assigned an individual variable. For example:

```
search_for_lat_eles = "sbend::b*"
```

will search for all non-lord bend lattice elements whose names begins with `"B"` followed by any set of characters. In this example, if, for example, two bends have the name, say "bend0", then a single variable will be set up to control these two bends.

`Warning`: Generally `-no_grouping` should be used with `unique_name_suffix` (§10.4) to avoid the problem that if different lattice elements have the same name but differing parameter values, the `write bmad` command (§11.39) will not produce a valid lattice.

Note: `search_for_lat_eles` and `use_same_lat_eles_as` cannot be used together.

10.10 Data and Constraint Initialization

Tao `data` (§6) is used with lattice correction or design (§8). A set of data is initialized using a `tao_d2_data` namelist block and one or more `tao_d1_data` namelist blocks.

The file that *Tao* looks in to find these two namelists is set by the `data_file` component of the `tao_start` namelist (§10.3). The default, if `data_file` is not set, is the root initialization file.

The format of the `tao_d2_data` namelist is
```
  &tao_d2_data
    d2_data%name = "<d2_name>"          ! d2_data name.
    universe     = "<list>"             ! Universes data belong in.
                                        !   "*" => all universes (default).
    default_merit_type = "<merit_type>" ! Sets how the merit is calculated.
    n_d1_data          = <integer>      ! Number associated d1_data arrays.
  /
```
For example: For example:
```
  &tao_d2_data
    d2_data%name = "orbit"
    universe     = "1,3:5"  ! Apply to universes 1, 3, 4, and 5
    n_d1_data    = 2
  /
```
A `tao_d2_data` block is needed for each `d2_data` structure defined. The `d2_data%name` component gives the name of the structure. The `universe` component gives a list of the universes that the data is associated with. A value of `"*"` means that a `d2_data` structure is set up in each universe. Ranges of universes can be specified in the list using a `:`.

The `default_merit_type` component determines how the merit function terms are calculated for the individual datum points. Possibilities are:
```
  "target"
  "max",     "min"
  "abs_max", "abs_min"
  "max-min"                  ! Only used when datum specifies a range of elements.
  "average", "integral"      ! Only used when datum specifies a range of elements.
```
The `average` and `max-min` merit types are used when there is a range of elements associated the the datum. That is, `ele_start` is specified (see below). For the `average` data type, the datum value is the average of the values computed for all lattice elements in the specified range. With `max-min`, the value of the datum is the difference between the maximum value in the range minus the minimum in the range. See Chapter 8 on optimization for more details.

The associated `tao_d1_data` namelists must come directly after their associated `tao_d2_data` namelist. The `n_d1_data` parameter in the `tao_d2_data` namelist defines how many `d1_data` structures are associated with the `d2_data` structure. For each `n_d1_data` structure there must be a `tao_d1_data` namelist which has the form:

```
&tao_d1_data
  ix_d1_data             = <integer>            ! d1_data index
  use_same_lat_eles_as   = "<d1_name>"          ! Reuse previous element list.
  search_for_lat_eles    = "<element_list>"     ! Find elements by name.
  d1_data%name           = "<d1_name>"          ! d1_data name.
  default_data_type      = <type_name>          ! EG: orbit.x, e_tot, etc...
  default_weight         = <real>               ! Merit function weight. Dflt: 0.0
  default_meas           = <real>               ! Default datum "measured" value (§8.1).
  default_data_source    = "<source>"           ! "lat" (dflt), "data", "var", or "beam".
  default_merit_type     = "<merit_type>"       ! Set default for datums.
  ix_min_data            = <integer>            ! Minimum array index.
  ix_max_data            = <integer>            ! Maximum array index.
  datum(N)%data_source    = "<source>"          ! "lat" (dflt), "data", "var", or "beam".
  datum(N)%data_type      = "<type_name>"       ! Eg: "orbit.x", etc.
  datum(N)%ele_name       = "<ele_name>"        ! Evaluation lattice element name.
  datum(N)%ele_start_name = "<ele_start_name>"  ! Start lattice element name.
  datum(N)%ele_ref_name   = "<ele_ref_name>"    ! Reference lattice element name.
  datum(N)%merit_type     = "<merit_type>"      ! Sets how the merit is calculated.
  datum(N)%meas           = <real>              ! Datum "measured" value (§8.1).
  datum(N)%weight         = <weight>            ! Merit function weight.
  datum(N)%good_user      = <logical>           ! Use for optimization and plotting?
  datum(N)%ix_bunch       = <integer>           ! Bunch index. Dflt: 0 = all bunches.
  datum(N)%eval_point     = "<where>"           ! "beginning", "center", or "end" (dflt).
  datum(N)%s_offset       = <real>              ! Default: 0.
  datum(N)%spin_axis      = <struct>            ! For spin G-matix calculations.
/
```

For example:

```
&tao_d1_data
  ix_d1_data        = 1
  d1_data%name      = "x"
  default_weight    = 1e6
  ix_min_data       = 0
  ix_max_data       = 99
  datum(0:)%ele_name = "DET_00W", " ", "DET_02W", ...
  datum(0:)%weight   = 0.23,      0,   0.45, ...
  ... etc ...
/
```

This format is called “component-by-component” since different datum components (`ele_name`, `weight`, etc.) are specified together on one line. See §6.2 for a list of components that are user settable. Alternatively, one can use “datum-by-datum” format to specify individual datums in a single line. For example

```
&tao_d1_data
  ix_d1_data        = 1
  d1_data%name      = "t"
  !           data_      ele_ref  ele_start ele     merit    meas   weight good
  !           type       name     name       name    type     value         user ..
  datum( 1) = "beta.a"   "S:2.3"  ""        "Q16_1"  "max"     30    0.1     T  ...
  datum( 2) = "eta.x"    ""       "B22"     "Q16##2" "max"     30    0.1     T  ...
  datum( 3) = "floor.x"  ""       ""        "end"    "target"   3    0.01    T  ...
```

```
    datum( 4) = "floor.x"  "B1"     ""       "1>>32"  "target"   3    0.01    T  ...
    ... etc. ...
  /
```

When using the datum-by-datum format, the columns are:

```
  data_type
  ele_ref_name
  ele_start_name
  ele_name
  merit_type
  meas_value
  weight
  good_user
  data_source
  eval_point
  s_offset
  ix_bunch
```

Default values will be used if an individual line does not include all columns.

A given `tao_d1_data` namelist may mix both component-by-component and datum-by-datum formats. In particular, component-by-component format must be used for components that cannot be set by the datum-by-datum format.

d1_data%name
: The name of the `d1_data` array. If `datum(N)%data_type` is not set, the `d1_data%name` is used to construct a default data type. See the `datum(N)%data_type` documentation.

datum(N)%data_source
: The `datum(N)%data_source` component specifies where the data is coming from. Possible values are:

```
  "beam"        ! Value is from multiparticle beam tracking.
  "data"        ! Used with expressions.
  "lat"         ! Value is from the lattice.
  "var"         ! Used with expressions.
```

With `%data_source` set to `"beam"`, the particular bunch that the data is extracted from can be specified via `datum(:)%ix_bunch`. The default is `0` which combines all the bunches for the datum calculation. If the `%data_source` is not set, the value of the `default_data_source` is used. If both `%data_source` and `default_data_source` are not specified, `"lat"` is the default. A `%data_source` of `"data"` or `"var"` establishes the default data source for evaluating expressions (see `"expression:"` in §6.9).

datum(N)%data_type
: If `datum(N)%data_type` is not given, and `default_data_type` is not specified, then the `d2_data` name and the `d1_data` name are combined for each datum to form the datum's `type`. For example, if the `d2_data%name` is `orbit`, and the `d1_data%name` is `x`, then the `data_type` is `orbit.x`. The `data_types` recognized by *Tao.* are given by Table 8.1. Custom data types not specified in this table must have a corresponding definition in `tao_hook_load_data_array.f90`. See Chapter 14 for details.

datum(N)%ele_name
: The `datum(N)%ele_name` name may be set to the appropriate element name or may be specified using element branch/element index notation (EG: "456", "1»22", etc.). On input, the datum's `ele_name` component (§6.2) will be set to the element name irregardless of the setting in the `tao_d1_data` namelist and the `ix_ele` datum component will be set to the element index.

If elements in the `data` array do not exist the corresponding `data%ele_name` should be left blank.

datum(N)%ele_ref_name, datum(N)%ele_start_name

Like `datum(N)%ele_name`, the `%ele_start_name`, and `%ele_ref_name` names may be specifed using branch/element index notation and on input the datum's `ele_start_name` and `ele_ref_name` will be set to the actual element names with the datum's `ix_ele_start` and `ix_ele_ref` set to the appropriate element indexes.

A range of elements can be specified by giving an `ele_start_name` that is not a blank string. Thus, in the above example, the value of `datum(2)` is the maximum horizontal dispersion in the range between the end of element `B22` to the end of element `Q16##2`. Elements can be specified by name (EG: `Q16_1`) or by longitudinal position using the notation `"S:<s_distance>"`. This will match to the element whose longitudinal position at the exit end is closest to `<s_distance>`.

datum(N)%eval_point

See §6.2 for details.

datum(N)%good_user

`datum(N)%good_user` is a logical that the user can toggle when running *Tao* (§6.2). The initial default value of `%good_user` is True.

datum(N)%ix_bunch

Index of the particle bunch used for evaluating the datum. Only needed if the datum is indeed associated with a bunch.

datum(N)%meas

"`Measured`" datum value used to calculate the datum's contribution to the merit function. See §8.2 for details.

datum(N)%merit_type

Merit type for the datum. See §8.2 for details.

datum(N)%s_offset

See §6.2 for details.

datum(N)%spin_axis

The `datum(N)%spin_axis` structure defines the coordinate axes at the reference element about which the spin G-matrix is computed (when the `datum(N)%data_type` is set to `spin_g_matrix.`ij (§6.9)). The `%spin_axis` structure has three components

```
spin_axis%l(3)
spin_axis%n0(3)
spin_axis%m(3)
```

The chapter on spin in the *Bmad* manual has information on how these axes are defined. With one exception (§6.9) The `n0` axis must be specified. If `l` is also specified, `m` will be appropriately calculated such that the axes form a right handed coordinate system. If `m` is also specified, `l` will be appropriately calculated. If neither `l` nor `m` is specified, `l` and `m` will be calculated somewhat arbitrarily to form a right handed coordinate system. Note: the axis vectors will be normalized to unity. Example:

```
&tao_d1_data
  ...
  datum(2)%n0 = 0, 1, 0
  datum(2)%l = 1, 0, 0
  ...
```

datum(N)%weight

`datum(N)%weight` gives the weight coefficient for a datum in the merit function. If not present then the default weight of `default_weight` is used.

default_data_source

Set the default for `datum(N)%data_source`.

default_data_type
Set the default for `datum(N)%data_type`.

default_data_weight
Set the default for `datum(N)%weight`

default_meas
Set the default for `datum(N)%meas`

default_merit_type
Set the default for `datum(N)%merit_type`

ix_d1_data
The `ix_d1_data` number gives the index in the array of `d1_data` structures within a `d2_data` structure the first `&tao_d1_data` namelist after a `&tao_d2_data` namelist should have `ix_d1_data` set to 1, etc.

ix_min_data, ix_max_data
`ix_min_data` and `ix_max_data` give the bounds for the `datum(N)` structure array that is associated with the `d1_data` structure.

search_for_lat_eles
Tao can search for the elements in the lattice to be associated with each data type by using the syntax:

```
   search_for_lat_eles = "{-no_lords} {-no_slaves} <element_list>"
```

`<element_list>` specifies elements using the standard element list format (§4.3). The `-no_lords` and `-no_slaves` switches, if present, are used to restrict the counting of lord or slave elements. The `-no_lords` switch excludes all group, overlay, and girder elements. The `-no_slaves` switch vetoes superposition or multipass slave elements. For example:

```
  search_for_lat_eles = "-no_lords sbend::b*
```

This will search for all non-lord bend lattice elements whose names begins with `"B"` followed by any set of characters. `search_for_lat_eles` and `use_same_lat_eles_as` cannot be used together.

use_same_lat_eles_as
Lists of names can be reused using the syntax:

```
  use_same_lat_eles_as = "<d1_name>"     ! Reuse previous element list.
```

For example:

```
  &tao_d1_data
    ix_d1_data       = 2
    d1_data%name     = "y"
    ...
    use_same_lat_eles_as = "orbit.x"
  /
```

10.10.1 Old Data Format

In the present data format there are three elements that are associated with a given datum: `ele_ref`, `ele_start`, and `ele`. There exists an old, deprecated, data format where only two elements are given for a given datum. These elements are called `ele0` and `ele`. In this old format, `data` is used in place of `datum`. For example:

```
  &tao_d1_data
    ! OLD SYNTAX. DO NOT USE!
    !          data_      ele0_     ele_      merit_   meas_    weight good_
    !            type       name      name      type     value          user
    data( 1) = "beta.a"   "S:12.3"  "Q16_1"  "max"      30      0.1     T
```

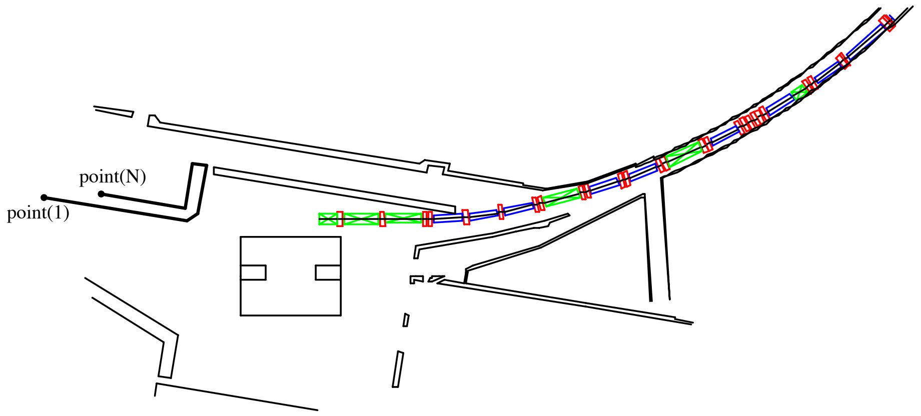


Figure 10.1: Floor_plan drawing showing the walls of the building (along with a section of a recirculation arc). Defining building walls can be useful for such things as floor plots and designing a machine to fit in an existing building.

```
    data( 2) = "phase.b"  "Q09_1"   "Q16_1"  "max"      30     0.1    T
    data( 3) = "floor.x"  " "       "end"    "target"    3     0.01   T
    data( 4) = "floor.x"  "B1"      "B2"     "target"    3     0.01   T
    ... etc. ...
  /
```

The interpretation of `ele0` was dependent upon the data type. With data types denoted as "`relative`", `ele0` was interpreted as `ele_ref`. For non-relative data types, `ele0` was interpreted as being equivalent to `ele_start`. The relative data types where:

`floor.x, floor.y, floor.z, floor.theta`
`momentum_compaction`
`periodic.tt.`$ijklm\ldots$
`phase.a, phase.b`
`phase_frac.a, phase_frac.b`
`phase_frac_diff`
`r.`ij
`rel_floor.x, rel_floor.y,`
`rel_floor.z, rel_floor.theta`
`s_position`
`t.`ijk
`tt.`$ijklm\ldots$

10.11 Building Wall Initialization

A two dimensional cross-section of the building containing the machine under simulation may be defined in *Tao*. This can be useful when drawing `floor_plan` plots of the machine (§10.13.8) or to design a machine to fit within an existing building by using optimization (§8).

The wall cross-sections are defined by a set of "`sections`". A section is a curve in the horizontal Z-X plane that defines where the face of a wall is. One such section is highlighted in Figure 10.1 starting at the point marked "point(1)" and ending at the point marked "point(N)". Each section is defined by a set of points which are connected together using straight lines or circular arcs.

The name of the file containing the building wall definition is given by the `building_wall_file` variable in the `tao_start` namelist (§10.3). In general, this file will contain a number of `building_wall_section` namelists. Each `building_wall_section` namelist defines a single wall section. The syntax of this namelist is

```
  &building_wall_section
    {name = <string>}
    {constraint = <type>}
    point(1) = <z1>, <x1>
    point(2) = <z2>, <x2>, {<r2>}
    point(3) = <z3>, <x3>, {<r3>}
    ... etc ...
    point(N) = <zN>, <xN>, {<rN>}
  /
```

The optional `name` component allows for matching wall sections to `floor_plan` shapes (§10.13.9) when drawing a `floor_plan` so that different portions of the wall can be drawn in different colors.

The global coordinate system in *Bmad* (see the *Bmad* manual) defines the (Z, X) plane as being horizontal. [Note: (Z, X) is used instead of (X, Z) since (Z, X, Y) forms a right handed coordinate system.] The points that define a wall section are specified in this coordinate system. In the `building_wall_section` namelist, the (Z, X) position of each point defining a wall section is given along with an optional radius r. If a non-zero radius is given for point j, then the segment between point $j - 1$ and j is a circular arc of the given radius. If no radius is given, or if it is zero, the segment is a straight line. A radius for the first point, number 1, cannot be specified since this does not make sense. Additionally, a radius must be at least half the distance between the two points that define the end points of the arc.

In general, given two end points and a radius, there are four possible arcs that can be drawn. The arc chosen follows the following convention:

1. The angle subtended by the arc is 180 degrees or less.
2. If the radius for the arc from $j - 1$ to j is positive, the arc curves in a clockwise manner. If the radius is negative, the arc curves counterclockwise. This convention mimics the convention used for `rbend` and `sbend` elements.

To define a wall that is circular, use three points with two 180 degree arcs in between.

When designing a machine to fit within the walls of a building, the `constraint` variable of the namelist is used to designate whether the given wall section is on the $+x$ (left) side of the machine or the $-x$ (right) side. Here x is the local reference frame transverse coordinate. See the write up of the `wall.right_side` and `wall.left_side` constraints in §6.9 for more details. Possible values for `constraint` are:

```
  "right_side"  ! Section is to be used with wall.right_side constraints
  "left_side"   ! Section is to be used with wall.left_side constraints
  "none"        ! Default. Section is ignored in any constraint calculation.
```

Using `"none"` for `constraint` is convenient for drawing building components on a `floor_plan` that are not used as an optimization constraint.

Example:

```
  &building_wall_section
    constraint = "left_side"
    point(1) =  23.2837,    8.2842
    point(2) = -10.9703,   13.8712,   107.345
    point(3) = -10.8229,   14.7737
  /
```

In this example, point 1 is at $(Z, X) = (23.2837, 8.2842)$, the segment between points 1 and 2 is an arc with a radius of 107.345 meters, and the segment between points 2 and 3 is a straight line. Also this wall section is to be used when evaluating any `wall.x+` constraint.

If the machine varies vertically (y-direction), vertical constraints may be imposed using the `floor.y` data type (§6.9).

To see a list of the building wall points when running *Tao*, use the `show building_wall` (§11.30.4) command .

Note: To position a machine in the global coordinate system, the starting point and starting orientation can be adjusted using `beginning[...]` statements as explained in the *Bmad* manual.

10.11.1 Building Orientation

It may be convenient to use a different two-dimensional coordinate system for the horizontal plane than the global coordinate system used by *Bmad* and *Tao*. For example, if the building wall coordinates are obtained from a blueprint. To help with this, an overall position and angle shift may be specified by a `building_wall_orientation` namelist in the same file with the `building_wall_section` namelists. The syntax of the `building_wall_orientation` namelist is:

```
  theta = <Real>       ! Angle rotation in radians. Default is 0.
  z_offset = <Real>    ! Z-offset. Default is 0.
  x_offset = <Real>    ! X-offset. Default is 0.
```

The transformation from the input coordinates of a wall point specified in a `build_wall_section` namelist to the global coordinate system is

$$\begin{pmatrix} z \\ x \end{pmatrix}_{global} = \begin{pmatrix} \text{z\_offset} \\ \text{x\_offset} \end{pmatrix} + \begin{pmatrix} \cos(\text{theta}) & -\sin(\text{theta}) \\ \sin(\text{theta}) & \cos(\text{theta}) \end{pmatrix} \begin{pmatrix} z \\ x \end{pmatrix}_{input} \tag{10.2}$$

10.12 Dynamic Aperture Calculation Initialization

For historical reasons, the `dynamic_aperture` program (another Bmad based program included in Bmad Distributions) is also capable of calculating the dynamic aperture. In fact both programs use the same underlying code for the aperture analysis. The basic difference is that the `dynamic_aperture` program is more flexible in terms of tracking. For example, the `dynamic_aperture` program can handle `ramper` elements and track with maps.

In a storage ring, the "`dynamic aperture`" is the region in phase space within which a particle will stably oscillate. That is, the region within which the motion is bounded. This is in contrast to the "`physical aperture`" which is defined by the vacuum chamber walls. Since it may take many turns for particle motion to become unstable, calculating the stability region for the full six-dimensional phase space is a time intensive process. In light of this, *Tao* uses a simplified calculation as discussed below.

In *Tao*, the motion of a particle is taken to be "stable" if the particle survives (does not hit the physical aperture or does not diverge to large amplitude) in tracking over some set number of turns.[2] A typical number is 1000 turns. A dynamic aperture "scan" is the calculation of the dynamic aperture in (x, y) space at some given initial phase space p_z as illustrated in Fig. 10.7. An (x, y) point represents the initial x and y phase space position of the particle with the initial p_x, p_y, and z values being set equal

[2]If the dynamic aperture is larger than the physical aperture the calculated boundary curve will reflect the physical aperture and not the dynamic aperture. In practice, this possible confusion is not a concern since if the dynamic aperture is outside the physical aperture there is no worry that the dynamic aperture will limit machine performance.

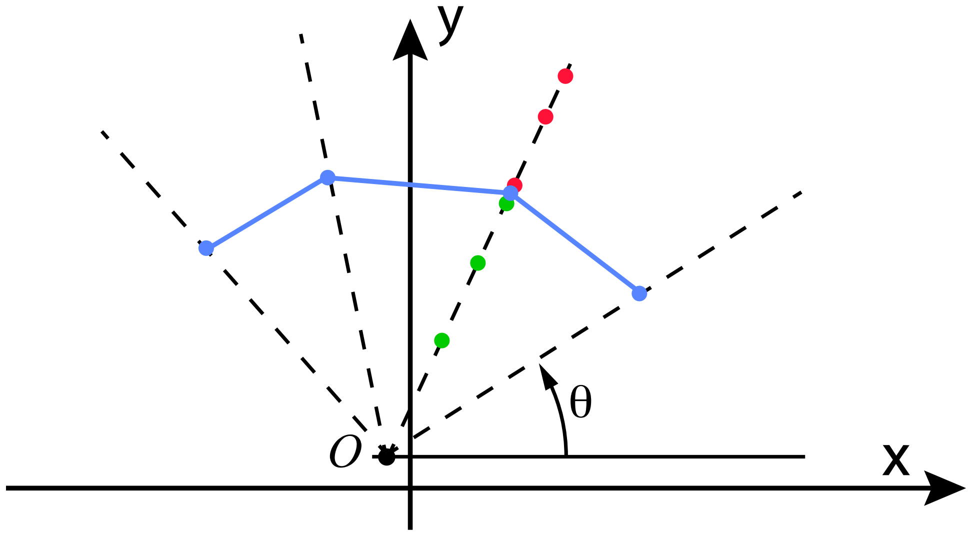


Figure 10.2: The calculation of a dynamic aperture curve in the x-y plane at a given initial p_z value involves calculating aperture curve points (blue dots) along a set of "rays" (dashed lines) having a common origin point ($\mathcal{O}$) which is taken to be the reference orbit. The line segments between points is simply for visualization purposes. The calculation of an aperture curve point along a given ray involves iteratively tracking particles with different starting (x, y) position values to find the boundary between stable (green dots) and unstable (red dots) motion.

to the closed orbit values. To calculate an aperture curve for a given initial p_z, a set of "rays" (dashed lines in the figure) are constructed. The rays have a common origin point ($\mathcal{O}$) which is the closed orbit (x, y) point. On each ray, the boundary point between stable and unstable motion (blue points in the figure) is found by iteratively tracking particles with initial (x, y) points on the ray (red and green dots) until the boundary point is determined with the specified accuracy.

The origin point of the rays is taken to be the closed orbit at the given p_z and RF off. This is true even if the RF is on for the tracking. The reason for this is that with the RF on, there is no well defined closed orbit at constant p_z (since p_z is not a constant of the motion with the RF on).

Having the RF off when tracking suppresses synchrotron oscillation effects which may be important. It is therefore recommended to have the RF on unless there is a good reason for ignoring synchrotron oscillations. It is also recommended that the lattice element at which the tracking begins be in a zero dispersion region.

To calculate the dynamic aperture for the i^{th} universe, the `design(N)%dynamic_aperture_calc` parameter must be set True in the `tao_design_lattice` namelist (§10.4). Example:

```
&tao_design_lattice
  design_lattice(1)%file = "lat.bmad"
  design_lattice(1)%dynamic_aperture_calc = T
/
```

Alternatively, the aperture calculation can be turned on during running using the `set` command (§11.29.28):

```
set universe 1 dynamic_aperture_calc on
```

Since aperture calculations take time, once an aperture calculation is done, the calculation is turned off so to perform multiple scans within a given session, the `set universe` command must be repeatedly done.

The `show dynamic_aperture` command (§11.30.11) shows parameter values and the `set dynamic_aperture` command (§11.29.9) can be used to change parameter values.

If Tao is compiled with `OpenMP` enabled (§2.3), the dynamic aperture calculation will be done in parallel.

Parameters for the dynamic aperture simulation are set in the `tao_dynamic_aperture` namelist (§10.3) in the *Tao* root initialization file. Multiple `tao_dynamic_aperture` namelists may be present if different universes need different parameter values. Example:

```
&tao_dynamic_aperture
  ix_universe = -1              ! Universe to apply to. -1 = all universes.
  pz = 0, 0.01, 0.15            ! List of phase space pz to start scans at.
  a_emit = 1e-11                ! A-mode emittance. Used for data calc.
  b_emit = 1e-13                ! B-mode emittance. Used for data calc.
  ellipse_scale = 10            ! Scale for drawing the ellipse in beam sigmas.
  da_param%start_ele = ''       ! Lattice element to start tracking at.
  da_param%n_angle = 21         ! Number of angles in scan of each energy
  da_param%min_angle = 0        ! Starting scan angle (rad).
  da_param%max_angle = 3.1416   ! Ending scan angle (rad).
  da_param%n_turn = 2000        ! Number of turns a particle must survive
  da_param%rel_accuracy = 1e-2 ! Relative accuracy of boundary point.
  da_param%abs_accuracy = 1e-5 ! Absolute accuracy of boundary point (meters).
  da_param%x_init = 1e-3        ! Initial horizontal aperture estimate. Default: 1e-3 meters.
  da_param%y_init = 1e-3        ! Initial vertical aperture estimate. Default: 1e-3 meters.
/
```

Parameters:

ellipse_scale
Scale for drawing the beam ellipse. The default value is 1 which will result in an ellipse drawn at 1 sigma.

ix_universe
The `ix_universe` parameter set which universe the parameters are applied to. Any universe index below a value of one results in the parameter values being applied to all universes.

pz
The `pz` parameter array is a list of p_z values to use. The number of scans (dynamic aperture curves) that are produced is equal to the number of `pz` values.

a_emit, b_emit
Emittance values for the a (“horizontal like”) and b (“vertical like”) normal modes. The emittance values do not affect particle tracking but are used to draw the beam sigma ellipse in dynamic aperture plots (§10.13.11) and to calculate the `dynamic_aperture.`N datum values (§6.9).

da_param%start_ele
This parameter sets the starting element for tracking. If not set, the beginning element of the root branch is used. `da_param%start_ele` may be set to either the element name or element index.

da_param%n_angle
The number of boundary points calculated for a scan is set by the `da_param%n_angle` parameter.

da_param%min_angle, da_param%max_angle
These parameters set the ray minimum and maximum angles, labeled θ in Fig. 10.7, in a scan. In the example above the angle ranges from 0 to pi. That is, the upper half-plane. These are typical settings since typically storage rings are vertically symmetric so the aperture curves should vertically symmetric as well.

The angles between adjacent rays is not uniform but are rather calculated to give a roughly equal spacing between boundary points. This is done by looking at the aperture points on a horizontal and a vertical ray and then scaling the ray angles appropriately).

da_param%rel_accuracy, da_param%abs_accuracy

These parameters set the relative and absolute accuracies that determine when the search for a boundary point is considered accurate enough.

If $r = \sqrt{(x - x_0)^2 + (y - y_0)^2}$ is the distance along any ray of the computed boundary point, where (x_0, y_0) are the coordinates of the origin point, the search for the boundary point will stop then the accuracy of the boundary point is below the desired accuracy σ_{cut} which is computed from

$$\sigma_{cut} = \sigma_a + r\,\sigma_r \tag{10.3}$$

with σ_a begin the absolute accuracy and σ_r being the relative accuracy.

da_param%x_init, da_param%y_init

These parameters set the initial x and y values used in the first two boundary point searches. The values of these parameters will not affect significantly affect the computed curve but will affect the computation time. If not set, these parameters will default to 0.001 meter.

To plot the results, an appropriate plot must be defined (§10.13.11) and `placed` in the plotting window (§10.13). An example dynamic aperture plot is shown in Fig. 10.7.

Example input files for calculating and plotting the dynamic aperture are at (§1.3):
`$ACC_ROOT_DIR/bmad-doc/tao_examples/dynamic_aperture`

10.13 Plotting Initialization

Tao has a graphical display window, called the `plot page` (§7.1) within which such things as lattice functions, machine layout, beam positions, etc., can be plotted. An example display is shown in Fig. 7.1.

Plotting is defined by an initialization file whose name is defined by the `plot_file` component of the `tao_start` namelist (§10.3).

10.13.1 Plot Page and Plot Regions

The `tao_plot_page` namelist (§10.2) in the plot initialization file (§10.3) sets `plot page` (§7.1) parameters including `region` definitions and the initial placement of plots. The syntax of this namelist is:

```
&tao_plot_page
  plot_page%title                    = "<string>", <x>, <y>, "<units>", "<justify>"
  plot_page%subtitle                 = "<string>", <x>, <y>, "<units>", "<justify>"
  plot_page%plot_display_type        = "<string>"  ! Display type: "X" or "TK"
  plot_page%size                     = <x_size>, <y_size>  ! Window size (POINTS)
  plot_page%border                   = <qp_rect_struct>    ! Border around edge
  plot_page%text_height              = <real>   ! height in POINTS. Def = 12
  plot_page%main_title_text_scale    = <real>   ! Rel to text_height. Def = 1.3
  plot_page%graph_title_text_scale   = <real>   ! Rel to text_height. Def = 1.1
  plot_page%axis_number_text_scale   = <real>   ! Rel to text_height. Def = 0.9
  plot_page%axis_label_text_scale    = <real>   ! Rel to text_height. Def = 1.0
  plot_page%legend_text_scale        = <real>   ! Rel to text_height. Def = 0.8
```

```
    plot_page%key_table_text_scale     = <real>   ! Rel to text_height. Def = 0.9
    plot_page%floor_plan_shape_scale   = <real>   ! Floor_plan shape size scaling.
    plot_page%floor_plan_text_scale    = <real>   ! Floor_plan shape text scaling.
    plot_page%lat_layout_shape_scale   = <real>   ! Lat_layout shape size scaling.
    plot_page%lat_layout_text_scale    = <real>   ! Lat_layout shape text scaling.
    plot_page%n_curve_pts              = <int>    ! Num points used to construct a
                                                  !   smooth curve. Default = 401
    plot_page%box_plots                = <T/F>    ! For debugging. Default = F.
    plot_page%delete_overlapping_plots = <T/F>    ! Default = T.
    plot_page%draw_graph_title_suffix  = <T/F>    ! Default = T.
    include_default_plots              = <T/F>    ! Include default templates? Def = T.
    region(N) = "<region_name>" <x1>, <x2>, <y1>, <y2>
    place(N)  = "<region_name>", "<template_name>"
    default_plot%...                            ! See below.
    default_graph%...                           ! See below.
  /
```

For example:

```
  &tao_plot_page
    plot_page%title = "CESR Lattice", 0.5, 0.996, "%PAGE", "CC"
    plot_page%plot_display_type = "X"        ! X11 window.  "TK" is alternative.
    plot_page%size        = 700, 800         ! Points
    plot_page%border      = 0, 0, 0, 50, "POINTS"
    plot_page%text_height = 12.0
    region(1) = "top"    0.0, 1.0, 0.5, 1.0
    region(2) = "bottom" 0.0, 1.0, 0.0, 0.5
    place(1)  = "top",    "orbit"
    place(2)  = "bottom", "phase"
    default_graph%x%min = 100
    default_graph%x%max = 200
  /
```

The `tao_plot_page` namelist has the following parameters:

default_graph

The `default_graph` parameter is used to set defaults for any `graph` component defined in any `tao_template_graph` namelist (§10.13.2). Example

```
  &tao_plot_page
    default_graph%x%min = 0
    default_graph%x%max = 100
    ...
```

This sets the default x-axis bounds. Also see `default_plot` below.

default_plot

The `default_plot` parameter is used to set defaults for any `plot` component defined in any `tao_template_plot` namelist (§10.13.2). Example:

```
  &tao_plot_page
    default_plot%x_axis_type = "index"
    ...
```

This sets the default `%x_axis_type`. Also see `default_graph` above.

include_default_plots

If `include_default_plots` is set to `False`, the collection of template plots (§10.13.2) that *Tao* constructs by default are not constructed. The default is True. Note: If `include_default_plots` is True, and if a user defines a template plot that has the same name as a default plot, the default plot will not be instantiated.

place(N)
The `place(N)` parameter, with `N` being an integer, determines the initial placement of plots (§7.3). If no `place` parameters are set, the default `orbit`, `beta`, `dispersion`, and `lat_layout` plots will be displayed. Each `place(N)` has the syntax:

```
  &tao_plot_page
    place(N) = "<region_name>" "<plot_template_name>"
    ...
```

`"<region_name>"` is the region name and `"<plot_template_name>"` is the name of the template plot to put in the region. Examples:

```
  &tao_plot_page
    place(1)  = "top",    "orbit"   ! Orbit plot placed in "top" region
    place(2)  = "bottom", "phase"   ! Phase plot placed in "bottom" region
    ...
```

plot_page%axis_number_text_scale
This along with `plot_page%text_height` sets the font size for the plot page title. See `plot_page%text_height` for more details.

plot_page%axis_label_text_scale
This along with `plot_page%text_height` sets the font size for the plot page title. See `plot_page%text_height` for more details.

plot_page%border
`plot_page%border` sets a border around the edges of the window. As shown in Figure 7.2, the offsets `x1`, `x2` in black (corresponding to `%border%x1` and `%border%x2`) are the right and left border widths and the offsets `y1` and `y2` in black (corresponding to `%border%y1` and `%border%y2`) are the bottom and top border widths respectively. The rectangle within this border is called the `plot area`.

plot_page%curve_legend
Sets parameters for the curve legend (§7.3). `%curve_legend` is an instance of a `qp_legend_struct` (§7.7.7). Note: When drawing a curve legend for a particular graph, the placement of the legend is given by `graph%curve_legend_origin`.

plot_page%delete_overlapping_plots
When `plot_page%delete_overlapping_plots` is True (the default), Placing a plot (using the `place` command §11.19) causes any existing plots that overlap the placed plot to become invisible.

plot_page%draw_graph_title_suffix
The `plot_page%draw_graph_title_suffix` is used to suppress the drawing of the string that is printed to the right of a graph title (set by `graph%title`). This string is set by *Tao* and has information on what is being plotted (typically the `curve%component`). To suppress the suffix, set `plot_page%draw_graph_title_suffix` to False.

plot_page%floor_plan_shape_scale
This parameter sets the overall scale for drawing shapes for a `floor_plan` drawing (§10.13.9). The default value is 1.

plot_page%floor_plan_text_scale
Sets the font size of `floor_plan` shape labels. The font size is the product

```
  size = plot_page%text_scale * plot_page%legend_text_scale * plot_page%floor_plan_text_scale
```

plot_page%graph_title_text_scale
This along with `plot_page%text_height` sets the font size for the plot page title. See `plot_page%text_height` for more details.

plot_page%key_table_text_scale
This along with `plot_page%text_height` sets the font size for the plot page title. See `plot_page%text_height` for more details.

plot_page%lat_layout_shape_scale
This parameter sets the overall scale for drawing shapes for a `lat_layout` drawing (§10.13.9). The default value is 1.

plot_page%lat_layout_text_scale
Sets the font size of `lat_layout` shape labels. The font size is the product

```
  size = plot_page%text_scale * plot_page%legend_text_scale * plot_page%lat_layout_text_scale
```

plot_page%legend_text_scale
Sets the font size of the graph curve legend (§7.5.3 relative to `plot_page%text_height`. See `plot_page%text_height` for more details. The setting of `plot_page%legend_text_scale` also affects the size of `lat_layout` and `floor_plan` plots.

plot_page%main_title_text_scale
Sets the font size of the the plot page title relative to the `plot_page%text_height`. See `plot_page%text_height` for more details.

plot_page%n_curve_pts
The `plot_page%n_curve_pts` parameter sets the default number of points to use for drawing "smooth" curves. The default is 401. This default may be overridden for individual plots by setting the `plot%n_curve_pts` component of a plot (§10.13.2). If `plot%n_curve_pts` is set for an individual plot, that value overrides the value of `plot_page%n_curve_pts`. Warning: *Tao* will cache intermediate calculations used to compute a smooth curve to use in the computation of other smooth curves. *Tao* will only do this for curves that have `plot_page%n_curve_pts` number of points. Depending upon the circumstances, setting `plot%n_curve_pts` for individual plots may slow down plotting calculations significantly.

plot_page%plot_display_type
The `plot_page%plot_display_type` component sets the type of plot display window used. possibilities are:

```
  "X"      X11 window
  "TK"     tk window
  "QT"     Available only when using PLPLOT (and not PGPLOT)
```

Note: The environment variable `ACC_PLOT_DISPLAY_TYPE` sets the default display type. You can set this variable in your login file to avoid having to setup a *Tao* init file to set this.

plot_page%size
The `plot_page%size` parameter sets the horizontal and vertical size of the plot window in `points` (§7.7.1). Also then environmental variable `ACC_DPI_RESOLUTION` (§7.1) can be used to vary the window size.

plot_page%subtitle
Subtitle text of the plot. See the description for `plot_page%title`. The defaults here are the same as `plot_page%title` except that `y` defaults to 0.97.

plot_page%text_height
The `plot_page%text_height` parameter sets the overall height of the text that is drawn. Relative to this, various parameters can be used to scale individual types of text:

```
  &tao_plot_page
    plot_page%main_title_text_scale  = 1.3 ! Main title height.
    plot_page%graph_title_text_scale = 1.1 ! Graph title height.
    plot_page%axis_number_text_scale = 0.9 ! Axis number height
    plot_page%axis_label_text_scale  = 1.0 ! Axis label height.
    plot_page%key_table_text_scale   = 0.8 ! Key Table text (§10.13.13).
    plot_page%legend_text_scale      = 0.9 ! Lat Layout or floor plan text.
    ...
```

The default values for these scales are given above.

plot_page%title

The `plot_page%title` sets the page title which is text that is generally printed at the top of the page. This parameter is a structure which has components:

```
string = ""          ! Text to print
x = 0.50             ! Horizontal position
y = 0.99             ! Vertical position
units = "%PAGE"      ! Units of x and y (§7.7.1)
justify = "CC"       ! Justification (§7.7.2)
```

The values shown are the defaults. Also see `page%subtitle`.

region(N)

The `region(N)` parameter, with `N` being an integer, is used to create custom regions (§7.2) in addition to the default regions defined by *Tao*. Each `region(N)` has the syntax:

```
region(N) = "<region_name>" <x1>, <x2>, <y1>, <y2>
```

`"<region_name>"` is the region name which may not contain a dot "." or a space. The other four elements `<x1>`, `<x2>`, `<y1>`, and `<y2>` define the region position on the plot page as discussed in Sec. §7.2. There is no upper limit to the number of regions that can be defined.

```
&tao_plot_page
  region(1) = "top"    0.0, 1.0, 0.5, 1.0
  region(2) = "bottom" 0.0, 1.0, 0.0, 0.5
  ...
```

10.13.2 Plot Templates

A plot `template` (§7.3) defines a set of parameters used for constructing a `displayed` plot. *Tao*, by default, defines a number of template plots. User defined template plots are constructed with a `tao_template_plot` namelist in the plot initialization file (§10.3) along with zero or more `tao_template_graph` namelists, one for each `graph` associated with the template plot. The syntax for the `tao_template_plot` is:

```
&tao_template_plot
  plot%name        = "<plot_name>"
  plot%x_axis_type = "<x_axis_type>"   ! "index", "ele_index" "s", "lat", "var", etc.
  plot%autoscale_gang_x = <logical>    ! Default: True.
  plot%autoscale_gang_y = <logical>    ! Default: True.
  plot%autoscale_x = <logical>         ! Default: False.
  plot%autoscale_y = <logical>         ! Default: False.
  plot%n_curve_pts = <integer>         ! Used to override plot_page%n_curve_pts.
  plot%n_graph     = <n_graphs>
  default_graph%...                    ! See below
  default_curve%...                    ! See below
/
```

For example:

```
&tao_template_plot
  plot%name                        = "orbit"
  default_graph%x%major_div_nominal = 10
  default_graph%x%label            = "Index"
  default_graph%y%max              = 10
  plot%n_graph                     = 2
/
```

The `tao_plot_page` namelist has the following parameters:

default_curve
The `default_curve` sets defaults for `curves` associated with the plot. The `default_curve` is a structure with the same components as the `curve` parameter of the `tao_template_graph` structure discussed below.

default_graph
The `default_graph` sets defaults for `graphs` associated with the plot. This is useful if there are multiple associated graphs. The `default_graph` is a structure with the same components as the `graph` parameter of the `tao_template_graph` structure discussed below.

Settings of `default_graph` in the `tao_template_plot` namelist overrides, for the graphs associated with the plot, any `default_graph` settings made in the `tao_template_plot` namelist (§10.13).

plot%autoscale_gang_x
The `plot%autoscale_gang_x` parameter is relavent if the plot has more than one associated graph. In this case, if set to True (the default), and if the `x_scale` command (§11.41) is applied to the plot (as opposed to being applied to an individual graph), the data of all the graphs is combined to compute a horizontal scale which is used for all the graphs. If `%autoscale_gang_x` is set to False, graphs are scaled individually.

plot%autoscale_gang_y
The `plot%autoscale_gang_y` parameter is relavent if the plot has more than one associated graph. In this case, if set to True (the default), and if the `scale` command (§11.28) is applied to the plot (as opposed to being applied to an individual graph), the data of all the graphs is combined to compute a vertical scale which is used for all the graphs. If `%autoscale_gang_y` is set to False, graphs are scaled individually.

plot%autoscale_x
Plots with `plot%autoscale_x` set to True will automatically rescale the horizontal axis after any calculation. Default is False.

plot%autoscale_y
Plots with `plot%autoscale_y` set to True will automatically rescale the vertical axes after any calculation. Default is False.

plot%n_curve_pts
The `plot%n_curve_pts` parameter sets the number of evaluation points to use for drawing "smooth" curves (§7.6). This overrides the setting of `plot_page%n_curve_pts` (§10.13). Warning: *Tao* will cache intermediate calculations used to compute a smooth curve to use in the computation of other smooth curves. *Tao* will only do this for curves that have `plot_page%n_curve_pts` number of points. Depending upon the circumstances, setting `plot%n_curve_pts` for individual plots may slow down plotting calculations significantly.

plot%n_graph
The `plot%n_graph` parameter sets the number of graphs associated with the plot and each one needs a `tao_template_graph` namelist to define it. These namelists should be placed directly after their respective `tao_template_graph` namelist.

plot%name
The `plot%name` parameter is the name that is used with *Tao* commands to identify the plot (§7.3). It is important that this name not contain any blank spaces since *Tao* uses this fact in parsing the command line.

plot%x_axis_type
The `plot%x_axis_type` parameter sets what is plotted along the `x_axis`. Possibilities are:

```
"index"          ! Data Index.
"ele_index"      ! Element lattice number index.
"s"              ! Longitudinal position in the lattice.
```

```
    "s_expression"  ! s-dependent expression involving lattice parameters.
    "data"          ! From a data array.
    "lat"           ! Lattice variable. See §10.13.6.
    "var"           ! Tao variable value. See §10.13.6.
    "phase_space"   ! Set by Tao if graph%type = "phase_space".
    "none"          ! Set by Tao if graph%type = "key_table".
    "floor"         ! Set by Tao if graph%type = "floor_plan".
```

The `ele_index` switch is used when plotting data arrays. In this case the `index` switch refers to the index of the data array and `ele_index` refers to the index of the lattice element that the datum was evaluated at.

The number of graphs associated with a template plot is specified by the setting of `plot%n_graph` in the `tao_template_plot` namelist. For each associated graph there needs to be a `tao_template_graph` namelist. These namelists need to be placed directly below the `tao_template_plot` namelist. Each `tao_template_graph` namelist must have a `graph_index` parameter with the first `tao_template_graph` namelist below the `tao_template_plot` namelist having `graph_index` set to 1, the next `tao_template_graph` having `graph_index` set to 2, etc.

The general format of the `tao_template_graph` namelist is:

```
  &tao_template_graph
    graph_index              = <integer>          ! Graph index. 1 = first graph, etc.
    graph%name               = "<string>"         ! Default is  "g<n>" <n> = graph_index.
    graph%type               = "<string>"         ! "data", "floor_plan", etc.
    graph%box                = <ix>, <iy>, <ix_tot>, <iy_tot>
    graph%title              = "<string>"         ! Title above the graph.
    graph%text_legend(n)     = "<string>"         ! Set legend text
    graph%text_legend_origin  = <qp_point_struct> ! Placement of the text legend
    graph%curve_legend_origin = <qp_point_struct> ! Placement of the curve legend
    graph%margin             =  <ix1>, <ix2>, <iy1>, <iy2>, "<Units>"
    graph%scale_margin       =  <ix1>, <ix2>, <iy1>, <iy2>, "<Units>"
    graph%x                  = <qp_axis_struct>   ! Horizontal axis.
    graph%y                  = <qp_axis_struct>   ! Left axis.
    graph%y2                 = <qp_axis_struct>   ! Right axis.
    graph%y2_mirrors_y       = <logical>          ! y2 min/max the same as y-axis? Default = T
    graph%clip               = <logical>          ! Clip curves at boundary? Default = T
    graph%draw_axes          = <logical>          ! Default = T
    graph%draw_grid          = <logical>          ! Default = T
    graph%draw_curve_legend  = <logical>          ! Default = T
    graph%draw_title         = <logical>          ! Default = T
    graph%allow_wrap_around  = <logical>          ! Wrap curves around lattice ends?
    graph%symbol_size_scale  = <real>             ! Phase_space plots symbol scale factor
    graph%ix_universe        = <integer>          ! Default = -1 => Use default universe
    graph%ix_branch          = <integer>          ! Lattice branch index. -1 => Use default branch
    graph%floor_plan         = <floor_plan_struct> ! Floor_plan parameters (§10.13.8).
    graph%draw_only_good_user_data_or_vars        ! Veto data or variables with good_user = F?
                                 = <logical>      !   Default = T.
    graph%x_axis_scale_factor    = <factor>       ! Scale the x-axis by this.
    graph%n_curve                = <integer>      ! Limit number of curves.
    curve(N)%name                = "<string>"     ! Default is "c<i>", <i> = curve num.
    curve(N)%data_type           = "<string>"     ! EG: "orbit.x"
    curve(N)%data_source         = "<string>"     ! Source for the data curve points
    curve(N)%data_type_x         = "<string>"     ! Used with plot%x_axis_type = "data" or "var".
    curve(N)%component           = "<string>"     ! Eg: "model - design".
    curve(N)%data_index          = "<string>"     ! Index number for data points.
    curve(N)%legend_text         = "<string>"     ! Text for curve legend.
    curve(N)%y_axis_scale_factor = <factor>       ! Scale the y-axis by this.
    curve(N)%use_y2              = <logical>      ! Use left-axis scale?
    curve(N)%draw_line           = <logical>      ! Connect data with lines?
    curve(N)%draw_symbols        = <logical>      ! Draw data symbols?
    curve(N)%draw_symbol_index   = <logical>      ! Print index number next to the data symbol?
    curve(N)%draw_error_bars     = <logical>      ! Draw error bars with data?
    curve(N)%ix_universe         = <integer>      ! Default = -1 => Use default uni.
    curve(N)%ix_branch           = <integer>      ! Default = -1  => Use default lat branch.
    curve(N)%ix_bunch            = <integer>      ! Bunch index. Default = 0 (all bunches).
    curve(N)%n_turn              = <integer>      ! For phase space multi_turn_orbit.
    curve(N)%line     = <qp_line_struct>          ! Line spec (color, width, etc.)
    curve(N)%symbol   = <qp_symbol_struct>        ! Symbol spec (color size, etc.)
    curve(N)%symbol_every      = <integer>        ! Plot symbol every # datums
    curve(N)%ele_ref_name      = "<string>"       ! Name of reference element.
    curve(N)%smooth_line_calc  = <Logical>        ! Calc data between symbol points?
    curve(N)%orbit    = <tao_graph_orbit_struct>  ! For E & B field plots
    curve(N)%hist     = <tao_histogram_struct>    ! For histograms
    curve(N)%z_color = <tao_curve_color_struct>   ! For phase space plotting
  /
```

For example:

```
  &tao_template_graph
    graph_index                = 1
    graph%name                 = "x"
    graph%type                 = "data"
    graph%box                  = 1, 1, 1, 2
    graph%title                = "Horizontal Orbit (mm)"
    graph%margin               =  60, 200, 30, 30, "POINTS"
    graph%y%label              = "X"
    graph%y%major_div_nominal  = 4
    curve(1)%component         = "model - design"
    curve(1)%data_source       = "data"
    curve(1)%data_type         = "orbit.x"
    curve(1)%units_factor      = 1000
    curve(1)%use_y2            = F
  /
```

The `tao_template_graph` namelist has the following parameters:

curve(N)%component
The `%component` sets from where data is derived from (§7.6.5).

curve(N)%data_index
When used with graphing a data slice (§10.13.4), the `%data_index` parameter sets the index number for the symbol points. The symbol index number can then be displayed next to the symbol.

When used with graphing the dynamic aperture (§10.13.11), the `%data_index` parameter is used to associate a curve with a given dynamic aperture scan.

curve(N)%data_source
The `%data_source` parameter sets where information is drawn in computing curve points (§7.6.6). Used in conjunction with `%data_type` and `%component`.

curve(N)%data_type
The `%data_type` parameter sets what is being computed (§7.6.7). Used in conjunction with `%data_type` and `%component`.

curve(N)%data_type_x
Used with data slices (§10.13.4).

curve(N)%draw_error_bars
The `%draw_error_bars` logical determines whether error bars are drawn when plotting data (`%data_source` set to `data`). The half height of the error bars is determined by the `error_rms` values of the data associated with the curve (§6.2). To keep things simple, *Tao* ignores the setting of `%component` when drawing error bars. This must be kept in mind since for example, the measurement error associated with a difference plot of measured data minus reference data (when `%component` is set to `meas-ref`) is different from just plotting measured data, which in turn is different from a plot of the data as calculated from the `model` (the measurement error associated with this is zero).

curve(N)%draw_line
Used to toggle drawing of the curved line associated with a curve (§7.6.3). Default is True.

curve(N)%draw_symbol_index
Used to toggle drawing of the symbol index. Default is False.

curve(N)%draw_symbols
Used to toggle drawing of symbols. Default is True.

curve(N)%ele_ref_name
The `%ele_ref_name` component is only used if `%data_source` is set to `"lat"`. If `%ele_ref_name` is set, the curve will be shifted by subtracting the value of the parameter being plotted evaluated at the reference element. For example, if `orbit.x` is being plotted, and `%ele_ref_name` is set to `"Q10W"`, the plotted curve will be shifted by subtracting the value of the horizontal orbit at Q10W. Notice that the shifting is done for each curve component. For example, if `%component` is set to `"model - design"`, the curve will be shifted by subtracting the difference between the `model` and `design` values evaluated at the reference element.

curve(N)%hist
The `%hist` parameter is a structure used for setting histogram parameters (§10.13.12).

curve(N)%ix_branch
The `%ix_branch` sets which lattice branch data (beta function, orbit, etc.) is taken. from. Default is -1 which translates to the default branch `global%default_branch`.

curve(N)%ix_bunch
The `%ix_bunch` parameter sets which particle bunch data is taken from if `%data_source` (§7.6.6) is set to `"beam"`. Default is 1.

curve(N)%ix_universe
The `%ix_universe` parameter sets which universe (§3.3) data is taken from. Default is -1 which means that the data will be drawn from the current default universe `global%default_uni`.

curve(N)%legend_text
The `%legend_text` parameter sets the text that is displayed in the curve legend (§7.5.3) for the curve.

curve(N)%line
The `%line` parameter sets parameters associated with curved line associated with the curve (§7.6.3). This parameter is a structure of type `qp_line_struct` (§7.7.4).

curve(N)%name
The identifying name of the curve (§7.6.2). Used in *Tao* commands that manipulate curves.

curve(N)%orbit

The `%orbit` parameter, used when plotting electric and magnetic fields, defines the orbit with constant transverse offset along which the fields are evaluated. This parameter is used when the `%data_type` is one of the following:

```
  b0_field.x,  b0_field.y,  b0_field.z,  b0_curl.x,  b0_curl.y,  b0_curl.z,  b0_div
  e0_field.x,  e0_field.y,  e0_field.z,  e0_curl.x,  e0_curl.y,  e0_curl.z,  e0_div
```

Note: The data types with names starting with "b_" and "e_" evaluate the field along the single particle trajectory.

The `%orbit` is a structure with the following subcomponents:

```
  x         ! horizontal $x$-position of orbit.
  y         ! vertical $y$-position of orbit.
  t         ! time to evaluate fields at.
```

curve(N)%smooth_line_calc

Sets if additional points are used to evaluate the curve at so that the drawn line has a "smooth" appearance (§7.6.3). Default is True.

curve(N)%symbol

The `%symbol` parameter sets parameters associated with the symbols to be drawn (§7.6.4). This parameter is a structure of type `qp_symbol_struct` (§7.7.5).

curve(N)%symbol_every

When drawing a set of symbols of a curve, if the density of symbols is too large so that the drawing is too crowded, The number can be reduced by a factor equal to the value of `%symbol_every`. For example, a setting of `3` will result in every third symbol being drawn. This is especially helpful in phase space plots. The default is `1`.

curve(N)%use_y2

Use the `y2` axis (§7.5.6) for the curve? Default is False.

curve(N)%y_axis_scale_factor

Curve vertical `y` and `y2` axes scale factor. For a given "datum" value, the plotted value will be:

```
  y(plotted) = scale_factor * y(datum)
```

The default value is `1`. For example, a `%y_axis_scale_factor` of 1000 will draw a 1.0 mm orbit at the 1.0 mark on the vertical scale. That is, the vertical scale will be in millimeters. `graph%x_axis_scale_factor`.

curve(N)%z_color

The `%z_color` parameter is a structure used for setting false color parameters for phase space plotting. See Sec. §10.13.14 for more details.

graph%allow_wrap_around

If `plot%x_axis_type` is set to `"s"`, and if the plotted data is from a lattice branch with a closed geometry, and if `graph%x%min` is negative, then the `graph%allow_wrap_around` parameter sets if the curves contained in the graph are "wrapped around" the beginning of the lattice so that the curves are not cut off at $s = 0$. The default is `True`.

graph%box

The `graph%box` parameter sets the layout of the box which the `graph` is placed in §7.4. In the above example, the graph divides the `region` the plot is placed in into two vertically stacked rectangles and the graph will be placed into the bottom one. The default is `1,1,1,1` which scales a graph to cover the entire `region` the plot is placed in.

graph%clip

Clip the graph curves at the graph top and bottom boundaries? Default is `True`.

graph%curve_legend_origin

The curve legend displays which curves are associated with which of the plotted lines and symbols. Two examples are given Fig. 7.1. The `%curve_legend_origin` defines where the upper left hand corner of the legend is. The default is:

```
graph%curve_legend_origin%x =  5.0
graph%curve_legend_origin%y = -2.0
graph%curve_legend_origin%units = "POINTS/GRAPH/LT"
```

The `%curve_legend_origin` is of type `qp_point_struct`. See §7.7.3 for details on this structure. Also see `graph%draw_curve_legend` and `plot_page%curve_legend`.

graph%draw_axes
Draw the graph axes? Default is `True`.

graph%draw_curve_legend
Draw the curve legend? Default is `True`. The curve legend displays which curves are associated with which of the plotted lines and symbols. Two examples are given Fig. 7.1. Also see `graph%curve_legend_origin`.

graph%draw_grid
Draw the graph grid? Default is `True`.

graph%draw_only_good_user_data_or_vars
When plotting *Tao* data (§6) or variables (§5): If `%draw_only_good_user_data_or_vars` is set to True (the default), symbol points of curves in the graph associated with data or variables whose `good_user` parameter is set to `False` will be ignored. That is, data and variables that will not be used in an optimization will be ignored. If `%draw_only_good_user_data_or_vars` is set to False, data or variables that have a valid value will be plotted.

graph%draw_title
Draw the graph title? Default is True.

graph%floor_plan
This parameter is a structure whose components are used when drawing a `floor_plan`. See Sec. §10.13.8 for more details.

graph%ix_branch
The `graph%ix_branch` parameter sets the default branch for curves of the graph. The default will be overridden by `curve(N)%ix_branch`.

graph%ix_universe
The `graph%ix_universe` parameter sets the default universe for curves of the graph.

graph%margin
`graph%margin` sets the margin between the `graph` and the `box` it is drawn in.

graph%n_curve
If not present, *Tao* will count the number of curves associated with a graph based on if `curve(N)%data_type` is set. In the case where `default_curve%data_type` in the `tao_template_plot` namelist is set, the `graph%n_curve` parameter can be set to limit the number of curves created.

graph%name
`graph%name` and `curve%name` define names to be used with commands. The default names are just the letter `g` or `c` with the index of the graph or curve. Thus, in the example above, the name of the curve defaults to `c1` and it would be referred to as `orbit.x.c1`. It is important that these names do not contain any blank spaces since *Tao* uses this fact in parsing the command line.

graph%scale_margin
`graph%scale_margin` is used to set the minimum space between what is being drawn and the edges of the `graph` when a `scale`, `x_scale`, or a `xy_scale` command is issued. Normally this is zero but is useful for `floor plan` drawings.

graph%symbol_size_scale

graph%text_legend

graph%text_legend_origin

graph%title

The `graph%title` component is the string printed just above the graph box. The full string will also include information about what is being plotted and the horizontal axis type. To fully suppress the title leave it blank. Note: A graph also has a `graph%title_suffix` which *Tao* uses to hold the string which is printed to the right of the `graph%title`. This string contains information like what `curve%component` is being plotted. The `graph%title_suffix` cannot be set by the user.

graph%type

`graph%type` is the type of graph. *Tao* knows about the following types:

```
  "data"               ! Lattice parameters, data and/or variable plots (default).
  "dynamic_aperture"   ! Dynamic aperture plot (§10.13.11).
  "floor_plan"         ! A 2-dimensional birds-eye view of the machine (§10.13.8).
  "histogram"          ! Histogram of plot (§10.13.12).
  "key_table"          ! Key binding table for single mode (§10.13.13).
  "lat_layout"         ! Schematic showing placement of the lattice elements (§10.13.7).
  "phase_space"        ! Phase space plots (§10.13.14).
```

With `graph%type` set to `"data"` (§7.5.5), data such as orbits and/or variable values such as quadrupole strengths are plotted. Here "data" can be data from a defined data structure (§6) or computed directly from the lattice, beam tracking, etc. A `"data"` graph type will contain a number of `curves` and multiple data and variable curves can be drawn in one graph.

With `graph%type` set to `floor_plan` (§10.13.8), the two dimensional layout of the machine is drawn.

With `graph%type` set to `histogram` (§10.13.12), such things such as beam densities can be histogrammed.

With `graph%type` set to `"key_table"` (§10.13.13), the key bindings for use in single mode (§12.1) are displayed. Note: The `"key_table"` graph type does not have any associated `curves`.

With `graph%type` set to `lat_layout` (§10.13.7), the elements of the lattice are symbolical drawn in a one dimensional line as a function of the longitudinal distance along the machine centerline.

With `graph%type` set to `phase_space` (§10.13.14), phase space plots are produced.

graph%x

The `%x` parameter sets parameters for the `x`-axis (§7.5.6). This parameter is a structure of type `qp_axis_struct`. See Sec. §7.7.6 for more details.

graph%x_axis_scale_factor

Sets the horizontal `x`-axis scale factor. For a given "datum" value, the plotted value will be:

```
  x(plotted) = scale_factor * x(datum)
```

The default value is `1`. For example, a `%x_axis_scale_factor` of 1000 will draw a 1.0 mm phase space z value at the 1.0 mark on the horizontal scale. That is, the horizontal scale will be in millimeters. Also see `curve(N)%y_axis_scale_factor`.

graph%y

The `%y` parameter sets parameters for the `y`-axis (§7.5.6). This parameter is a structure of type `qp_axis_struct`. See Sec. §7.7.6 for more details.

graph%y2

The `%y2` parameter sets parameters for the `y2`-axis (§7.5.6). This parameter is a structure of type `qp_axis_struct`. See Sec. §7.7.6 for more details.

graph_index
The first `tao_template_graph` namelist after the associated `tao_template_plot` namelist must have the `graph_index` component set to 1. The next must have `graph_index` set to 2, etc. *Tao* uses the `graph_index` component to check for errors.

10.13.3 Lattice Parameter Graphing

Templates for plotting lattice parameters such as Twiss parameters or the orbit can be defined by setting the `data_type` of a curve appropriately. Example:

```
&tao_template_plot
  plot%name = "my_orbit"
  plot%x_axis_type = "s"
  plot%n_graph = 1
  default_curve%y_axis_scale_factor = 1000  ! mm
/

&tao_template_graph
  graph_index = 1
  graph%name = "g"
  curve(1:2)%data_type = "orbit.x", "orbit.y"
/
```

lattice parameter names correspond to data type name as listed in Sec. §6.8.

If the `plot%x_axis_type` is set to `"index"`, the horizontal axis will be the lattice element index.

If the `curve(N)%source` parameter is set to `"beam"` (The default is `"lat"`), beam tracking must be done (§10.7) to have a visible plot. Additionally, due to the way *Tao* does beam tracking, *Tao* is only able to evaluate the plotted parameter at the boundaries between lattice elements. This means that *Tao* is not able to do the "smooth" line calculation.

10.13.4 Data Slice Graphing

Note: Data slicing is a type of parametric plotting. For parametric plotting using curve data see section §10.13.5.

The standard data graph, as presented in the previous subsection, plots data from a given `d1_data` array. It is also possible to graph data that has been "sliced" in other ways. For example, suppose a number of universes have been established, with each universe representing the same machine but with different steerings powered. If in each universe an `orbit d2_data` structure has been defined, an example of a data slice is the collection of points (x, y) where:

```
(x, y) = (<n>@orbit.x[23], <n>@orbit.y[23]),  <n> = 1, ..., n_universe
```

When defining a template for graphing a data slice, the `plot%x_axis_type` is set to `"data"`, and the `graph%type` must be set to `"data"`, the `curve(:)%data_source` must be set to `"data"` and the `curve(:)%data_type_x` and `curve%data_type` are used to define the x and y axes respectively. In the strings given by `<curve%data_type_x` or `<curve%data_type`, all substrings that look like `#ref` are eliminated and the string given by `curve%ele_ref_name` is substituted in its place. Similarly, a `#comp` string is used as a place holder for the `curve%component` Example:

```
&tao_template_plot
  plot%name = "at_bpm"
  plot%x_axis_type = "data"
```

```
    plot%n_graph = 1
  /

  &tao_template_graph
    graph_index = 1
    graph%title = "Orbit at BPM"
    graph%y%label = "y"
    graph%type = "data"
    graph%x_axis_scale_factor = 1000
    graph%x%label = "x"
    curve(1)%component   = "meas - ref"
    curve(1)%data_source = "data"
    curve(1)%data_type_x = "[2:57]@orbit.x[#ref]|#comp"
    curve(1)%data_type   = "[2:57]@orbit.y[#ref]|#comp"
    curve(1)%data_index  = "[2:57]@orbit.y[#ref]|ix_uni"
    curve(1)%y_axis_scale_factor = 1000
    curve(1)%ele_ref_name = "23"
    curve(1)%draw_line = F
  /
```

In this example, `curve(1)%data_type_x` expands to `"[2:57]@orbit.x[23]|meas-ref"`. That is, the `meas - ref` values of `orbit.x[23]` from universes 2 through 57 is used for the x-axis. Similarly, `orbit.y[23]` is used for the y-axis. The `set` command (§11.29) can be used to change `curve%ele_ref_name` and `curve(1)%component` strings.

`curve%data_index` sets the index number for the symbol points (§10.13.2). In the above example, `curve%data_index` is set to `"[2:57]@orbit.y[#ref]|ix_uni"`. The `|ix_uni` component will result in the symbol index number being the universe number. Additionally, the component `|ix_d1` can be used to specify the index in the `d1_data` array, and the component `|ix_ele` can be used to specify the lattice element index. Setting the symbol index number is important when `curve%draw_symbol_index` is set to True so that the symbol index is drawn with the curve. Additionally, the command `show curve -symbol` (§11.30) will print the symbol index number along with the (x, y) coordinates of the symbols.

Arithmetic expressions (§4.4) may be mixed with explicit datum components in the specification of `curve(:)%data_type_x` and `curve(:)%data_type`. Example:

```
  curve(1)%data_type_x = "[#ref]@orbit.x|model"
  curve(1)%data_type   = "[#ref]@orbit.x|meas-ref"
  curve(1)%ele_ref_name = "3"
```

The plots the `model` values of `orbit.x` verses `meas - ref` of `orbit.x` for the data in universe 3. Note: Whenever explicit components are specified, the `curve%component` settings are ignored for that expression.

10.13.5 Parametric Plotting

With parametric plotting, both the x and the y values of the points on a curve are dependent upon an independent parameter. An example could be plotting $\alpha_a(s)$ versus $\sqrt{\beta_b(s)}$ over some range of the independent parameter s. One way to do parametric plotting is to use data slices as discussed in section §10.13.4. Another way to do parametric plotting, which is discussed in this section, is to setup two plot curves whose y values are the desired dependent parameters ($\alpha_x(s)$ and $\beta_y(s)$ say) and then define a parametric curve which uses the data from these curves.

The two curves from which the data is to be taken must be in the same graph. The y values from the first curve will be taken to define the x coordinate of the parametric curve and the y values from the second curve will be taken to define the y coordinate of the parametric curve. The plot that holds these curves will be called the "`source`" plot. Example:

```
&tao_template_plot
 plot%name = "src"
 plot%x_axis_type = "s"
 plot%n_graph = 1
/

&tao_template_graph
 graph_index = 1
 graph%name = "g"
 curve(1)%data_source = "lat"
 curve(1)%data_type = "alpha.a"
 curve(2)%data_source = "lat"
 curve(2)%data_type = "expression: sqrt(beta.b)"
/
```

This defines a source plot called `src` with two curves which will be used in the parametric plot.

The parametric plot curve references the source curves by setting the parametric curve's `data_source` parameter equal to `"curve"` and the parametric curve's `data_type` to the graph in the source plot which contains the source curves. For example:

```
&tao_template_plot
  plot%name = "parametric"
  plot%n_graph = 1
  plot%x_axis_type = "curve"
/

&tao_template_graph
  graph_index = 1
  graph%name = "g1"
  curve(1)%data_source = "curve"
  curve(1)%data_type = "src.g"
/
```

The parametric plot's `x_axis_type` needs to be set to `"curve"` along with the parametric curve's `data_source`.

When the parametric plot is `placed` in the plot window, *Tao* will look for a suitable source plot to connect with. If *Tao* does not find a suitable source plot, *Tao* will place a source plot in an unused plot `region` and set the plot to be invisible. The region name will be set to

```
<source-plot-name>_<parametric-plot-region>
```

where `<source-plot-name>` is the name of the source plot and `<parametric-plot-region>` is the name of the region where the parametric plot has been placed. For example, if the above parametric plot is placed in a region called "`r12`", the name of the region where the source plot is placed will be named "`src_r12`". Note: The `show plot` command will show if a plot in a given region is visible. The `set plot` (§11.29.20) command can be used to toggle plot visibility.

10.13.6 X-Axis Variable Parameter Plotting

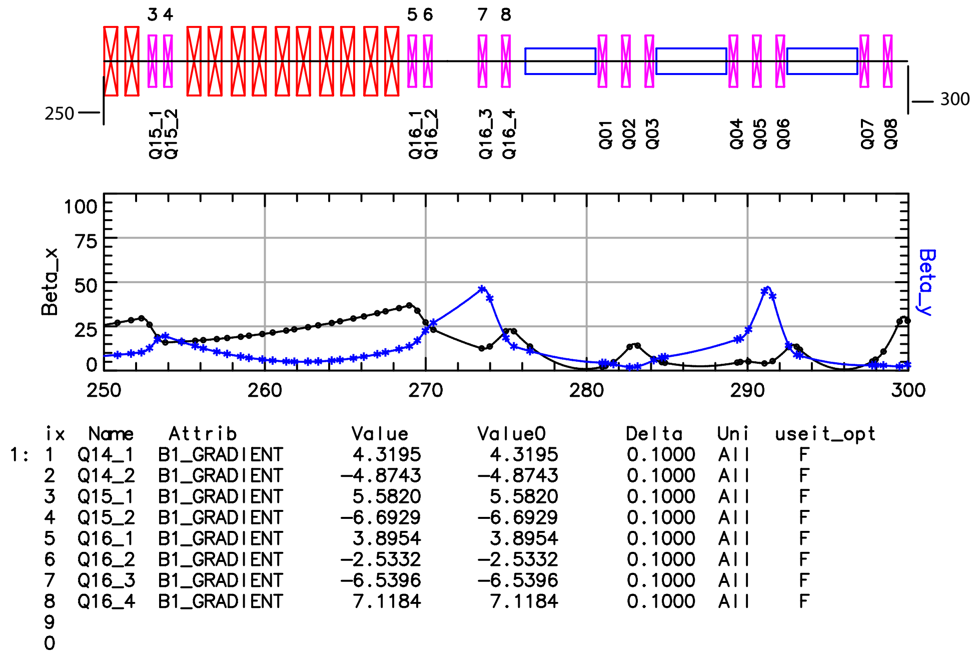
Data can be plotted as a function of a lattice parameter by setting `plot%x_axis_type` to `"lat"` (for lattice parameters including lattice element parameters) or `"var"` (for *Tao* variables) and setting `curve(:)%data_type_x` to the name of the variable. In this case, the `curve(:)%data_type` must evaluate to a single number (not a vector).

Example:
```
  &tao_template_plot
    plot%x_axis_type = "lat"
    plot%n_curve_pts = 50
    ...
  /

  &tao_template_graph
    ...
    curve(1)%data_type_x = "particle_start[x]"  ! X-axis values.
    curve(1)%data_type   = "orbit.x[10]"        ! Y-axis values.
    ...
  /
```
Here the number of curve points has been set to 50 to reduce the evaluation overhead.

Note: *Tao* treats the `design` and `base` lattices as static so that varying a variable will not affect these lattices. Thus, constructing a plot with `curve%component` set to, for example, `"model - design"` will *not* produce a plot that is the difference between varying a variable in both `model` and `design` lattices. In the case where such a plot is desired, a second universe needs to be established. In this case, one would set `curve(:)%data_type` to something like
```
    curve(1)%data_type   = "1@orbit.x[10] - 2@orbit.x[10]"
```
where the universe #2 `model` lattice would be setup to be equal to the universe #1 `design` lattice.

10.13.7 Lattice Layout Drawing

A lattice layout plot draws the lattice along a straight line with colored rectangles representing the various elements. An example is shown in Figure 10.3. The `tao_template_plot` needed to define a lattice layout looks like:
```
  &tao_template_plot
    plot%name        = "<plot_name>"
    plot%n_graph     = <integer>
    plot%x_axis_type = "s"
  /
  &tao_template_graph
    graph_index       = <integer>
    graph%name        = <name>
    graph%type        = "lat_layout"
    graph%title       = "Layout Title"
    plot%box          = <ix>, <iy>, <ix_tot>, <iy_tot>
    graph%ix_universe = <integer> ! -1 => use current default universe
    graph%ix_branch   = <integer> !  0 => use main lattice.
    graph%margin      = <ix1>, <ix2>, <iy1>, <iy2>, "<Units>"
    graph%x%min       = <real>
    graph%x%max       = <real>
    graph%y%min       = <real>    ! Default: -100
    graph%y%max       = <real>    ! Default:  100
  /
```
Example:



```
     ix  Name   Attrib         Value      Value0       Delta   Uni  useit_opt
  1: 1   Q14_1  B1_GRADIENT    4.3195     4.3195      0.1000  All     F
     2   Q14_2  B1_GRADIENT   -4.8743    -4.8743      0.1000  All     F
     3   Q15_1  B1_GRADIENT    5.5820     5.5820      0.1000  All     F
     4   Q15_2  B1_GRADIENT   -6.6929    -6.6929      0.1000  All     F
     5   Q16_1  B1_GRADIENT    3.8954     3.8954      0.1000  All     F
     6   Q16_2  B1_GRADIENT   -2.5332    -2.5332      0.1000  All     F
     7   Q16_3  B1_GRADIENT   -6.5396    -6.5396      0.1000  All     F
     8   Q16_4  B1_GRADIENT    7.1184     7.1184      0.1000  All     F
     9
     0
```

Figure 10.3: A lattice layout plot (top) above a data plot (middle) which in turn is above a key table plot (bottom). The points on the curves in the data plot mark the edges of the elements displayed in the lattice layout. Elements that have attributes that are varied as shown in the key table have the corresponding key table number printed above the element's glyph in the lattice layout.

```
&tao_template_plot
  plot%name        = "layout"
  plot%n_graph     = 1
  plot%x_axis_type = "s"
/

&tao_template_graph
  graph_index       = 1
  graph%name        = "u1"
  graph%type        = "lat_layout"
  graph%box         = 1, 1, 1, 1
  graph%ix_universe = -1  ! Use default universe
  graph%margin      = 0.12, 0.12, 0.30, 0.06, "%BOX"
/
```

Which elements are drawn is under user control and is defined using an `lat_layout_drawing` namelist. See Section §10.13.9 for more details.

Setting `graph%ix_universe` to -1 means the current default universe will be drawn. Normally, if there are element shapes that are associated with data or variable shapes (§10.13.9), these shapes will be drawn if there are lattice elements associated with the data or variables that live in the universe with index `graph%ix_universe` and if the associated elements fall within the range of elements plotted. The exception is that if `graph%ix_universe` is set to -2, the universe of the associated lattice elements is ignored. Using a value of -2 here only makes sense if the design lattices of all the universes is the same.

The longitudinal distance markers at either end of the lattice layout can be suppressed by setting

```
  graph%x%draw_numbers = F
```

10.13.8 Floor Plan Drawing

A `floor plan` drawing gives a display of the machine projected onto the horizontal plane. An example is shown in Figure 10.4. Like a `Lattice Layout` (§10.13.7), Elements are represented by colored rectangles and which elements are drawn is determined by a `floor_plan_drawing` namelist (see §10.13.9). Additionally, a cross-section of the walls of the building containing the machine (§10.11) can be drawn along with the reference orbit (which is the closed orbit for machines with a closed geometry). This is illustrated in Figure 10.5.

The placement of a lattice element in the drawing is determined by the element's coordinates in the `global reference system`. See the Bmad manual for more information on the `global reference system`. In the `global reference system`, the (Z, X) plane is the horizontal plane.

A floor plan orbit is associated with a `graph` of a `plot` (§10.13.2). A `graph` has a `floor_plan` parameter which is a structure of type `tao_floor_plan_struct`. Components of this structure can be set to control how a floor plan is drawn. The components of a `tao_floor_plan_struct` are:

```
  type tao_floor_plan_struct:
    rotation             = <real>      ! Rotation of floor plan plot: 1.0 -> 360 deg.
    view                 = "<string>"  ! View plane for floor plan plot. default = "zx"
    correct_distortion   = <logical>   ! Scale plot to use an equal aspect ratio so squares
                                       !   in real space are drawn as squares? Default = T
    flip_label_side      = <logical>   ! Draw element label on other side of element?
    size_is_absolute     = <logical>   ! Shape sizes scaled to absolute dimensions?
    draw_only_first_pass = <logical>   ! Draw only first pass with multipass elements?
    orbit_scale          = <real>      ! Scale for the orbit. Default = 0 => No orbit drawn.
    orbit_color          = "<color>"   ! Line color. Default = "red".
    orbit_pattern        = "<pattern>" ! Line pattern. Default = "solid_line".
    orbit_width          = <integer>   ! Line width. Default = 1.
    orbit_lattice        = "<string>"  ! May be "model" (default), "design", or "base".
```

A graph is initialized with a `tao_template_graph` namelist (§10.13.2). Example:

```
  &tao_template_graph
    ...
    graph%floor_plan%rotation = 0.5   ! Rotate 180 degrees
```

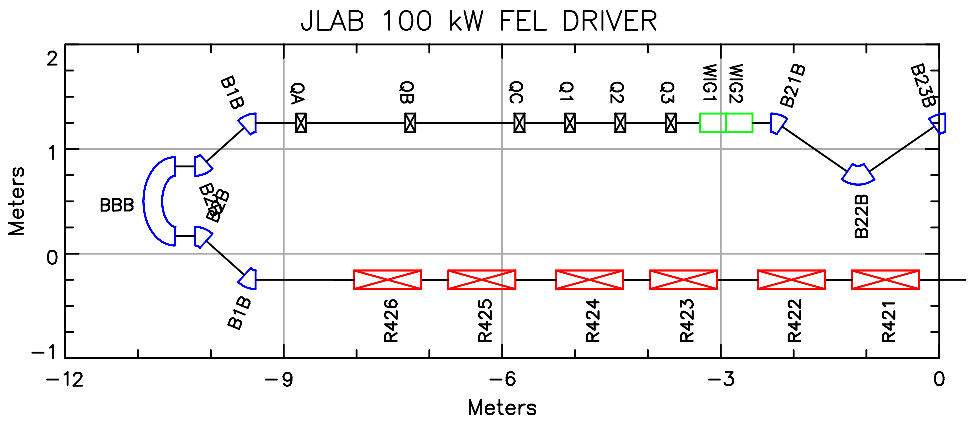


Figure 10.4: Example Floor Plan drawing.

```
    graph%floor_plan%orbit_scale = 100
    graph%floor_plan%orbit_color = "red"
    graph%floor_plan%orbit_width = 3
    graph%floor_plan%view = "zx"
  /
```

The `orbit_scale` component scales the displacement of the orbit from the lattice reference coordinate system (which is the centerline of the lattice elements if there are no misalignments). So a value of 100.0, a 1 cm orbit is drawn 1 meter from the centerline. A setting of zero (the default) means that the orbit is now drawn. Note: If `orbit_scale` is not unity, the plotted orbit when going through a `patch` element with a finite transverse offset will show a discontinuity due to the discontinuity of the reference orbit.

What plane a floor plan is projected onto is determined by the setting of the `graph%floor_plan%view` switch. This switch is a two character string. Each character is either "x", "y", or "z" and the characters must not be both the same. Default is "zx". The first character determines which global coordinate is mapped to the horizontal axis of the graph and the second character determines which global coordinate is mapped to the vertical axis of the graph. There are six possible two character combinations. The default "zx" setting represents looking at the horizontal plane from above. A setting of "xz" represents looking at the horizontal plane from below. The other combinations involving "y" are only potentially useful if the machine has a significant vertical extent.

To draw multiple orbits representing orbits from `model`, `design`, and/or `base` lattices, define multiple `graphs` within a `plot`, one for each type of orbit to be displayed and set the `floor_plan%orbit_lattice` appropriately for each graph.

If element labels are to be drawn, on which side the labels are drawn can be flipped by setting `graph%floor_plan%flip_label_side` to True.

The `size_is_absolute` logical is combined with the `<size>` setting for a shape to determine the size transverse to the center line curve of the drawn shape (§10.13.9). If `size_is_absolute` is False (the default), `<size>` is taken to be the size of the shape in points (72 points is approximately 1 inch). If `size_is_absolute` is True, `<size>` is taken to be the size in meters. That is, if `size_is_absolute` is

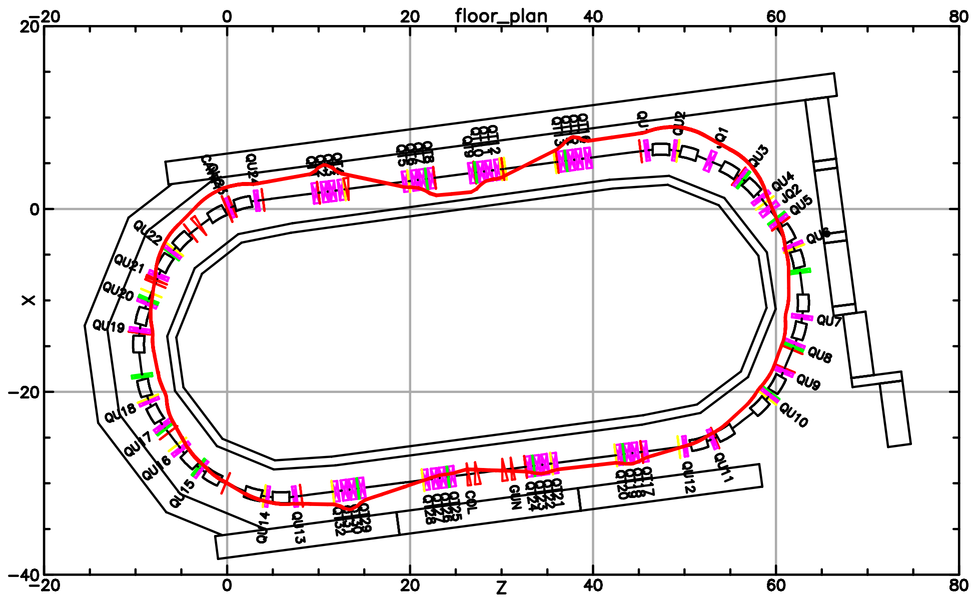


Figure 10.5: Example Floor plan drawing with the closed orbit (red line) and building walls included.

False, zooming in or out will not affect the size of an element shape while if `size_is_absolute` is True, the size of an element will scale when zooming.

An overall rotation of the floor plan can be controlled by setting `rotation` parameter. A setting of 1.0 corresponds to 360°. Positive values correspond to counter-clockwise rotations. Alternatively, the global coordinates at the start of the lattice can be defined in the lattice file and this can rotate the floor plan. Unless there is an offset specified in the lattice file, a lattice will start at $(x, y) = (0, 0)$. Assuming that the machine lies in the horizontal plane with no negative bends, the reference orbit will start out pointing in the negative x direction and will circle clockwise in the (x, y) plane.

The `draw_only_first_pass` logical, if set True, suppresses drawing of `multipass_slave` lattice elements that are associated with the second and higher passes. This logical defaults to False. Setting to True is only useful in some extreme circumstances where the plotting of additional passes leads to large pdf/ps file sizes.

Note: If `graph%ix_universe` is set to -1 the current viewed universe is used. If `graph%ix_universe` is set to -2, all universes are plotted.

Example Floor Plan template:
```
  &tao_template_plot
    plot%name = "floor"
    plot%n_graph = 1
  /

  &tao_template_graph
    graph_index = 1
    graph%name = "1"
    graph%type = "floor_plan"
    graph%box = 1, 1, 1, 1
    graph%margin = 0.10, 0.10, 0.10, 0.10, "%BOX"
    graph%ix_universe = -2   ! Draw all universes.
    graph%x%min = -12
    graph%x%max = 0
    graph%x%major_div_nominal = 4
    graph%x%minor_div = 3
    graph%x%label = "SMART LABEL"
    graph%y%label = "SMART LABEL"
    graph%y%max = 2
    graph%y%min = -1
    graph%floor_plan%correct_distortion = T
    graph%floor_plan%size_is_absolute = T
    graph%floor_plan%view = "xz"  ! Looking from beneath
    graph%floor_plan%orbit_scale = 100
  /
```
Having `graph%x%label` and `graph%y%label` set to "`SMART LABEL`" means that the actual axis labels will be picked appropriately based upon the setting of `graph%floor_plan%view`.

To prevent the drawing of the axes set `graph%draw_axes` to False. To prevent the drawing of a grid at the major division points set `graph%draw_grid` to False.

By default, the horizontal or vertical margins of the graph will be increased so that the horizontal scale (meters per plotting inch) is equal to the vertical scale. That is, both axes will have an equal aspect ratio and squares in real space will be drawn as squares in the plot. If `graph%floor_plan%correct_distortion` is set to `False`, this scaling will not be done.

Note: The `show ele -floor` command (§11.30) can be used to view an element's global coordinates.

10.13.9 Lat_layout and Floor_plan Drawings Shape Definition

`Floor plan` (§10.13.8) and `lattice layout` drawings use various shapes, sizes, and colors to represent lattice elements. The association of a particular element with a given shape is determined via two namelists: `lat_layout_drawing` for the lattice layout and `floor_plan_drawing` for floor plan drawings. Two different namelists are used since, for example, a size that is good for a layout will not necessarily be good for a floor plan.

The file that *Tao* looks in to find these two namelists is set by the first file specified in the `plot_file` array set in the `tao_start` namelist (§10.3). The default, if `plot_file` is not set, is the root initialization file.

The namelist syntax is the same for both:

```
&lat_layout_drawing
  include_default_shapes = <logical>
  ele_shape(N) = "<ele_id>" "<shape>" "<color>" "<size>" "<label>" <draw>
                                                    <multi> <line_width> <offset>
/

&floor_plan_drawing
  ... same as lat_layout_drawing ...
/
```

For Example:

```
&floor_plan_drawing
  include_default_shapes = T
  !               ele_id                    Shape         Color     Size  Label  ..etc..
  ele_shape(1) = "quadrupole::q*"          "box"         "red"     0.75  "name"
  ele_shape(2) = "quadrupole::*"           "xbox"        "red"     0.75  "none"
  ele_shape(3) = "sbend::sb*"              "box"         "blue"    0.37  "none"
  ele_shape(4) = "sbend::*"                "box"         "blue"    0.37  "none"
  ele_shape(5) = "wiggler::*"              "xbox"        "green"   0.50  "name"
  ele_shape(6) = "var::quad_k1"            "circle"      "purple"  0.25  "name"
  ele_shape(7) = "data::orbit.x|design"    "vvar:box"    "orange"  0.25  "name"
  ele_shape(8) = "building_wall::*"        "solid_line"  "black"    0    "none"
  ele_shape(3)%multi = T
  ele_shape(5:6)%line_width = 5, 6
/
```

A figure is drawn for each lattice element in the lattice that matches the `<ele_id>` field (§4.3). Thus, in the example above, `ele_shape(1)` will match to all quadrupoles whose name begins with "q" and `ele_shape(2)` will match all quadrupoles.

Besides the usual element class prefixes (`quadrupole::`, `sbend::`, etc.), other prefixes that can be used with an `<ele_id>` are

```
data::            ! Match to Tao datum name.
var::             ! Match to Tao variable name.
alias::           ! Match to lattice element alias parameter.
type::            ! Match to lattice element type parameter.
building_wall::   ! Used in floor_plan plots.
```

The `data::` prefix is used to match to data that will be used in an optimization. Thus, in the above example, `ele_shape(7)` specifies that an "x" will be drawn at points where there is valid `orbit.x` data. For this to work, an `orbit.x` data array must be defined (§10.10).

The `var::` prefix is used for drawing variable locations for variables used in an optimization. In the above example, it is assumed that a `quad_k1` variable array has been setup. A circle will be drawn at each element under control of a `quad_k1` variable.

For `floor_plan` drawings, the building wall (§10.11) can be drawn by specifying an `ele_shape` whose name is `"building_wall::<name>"` where `<name>` is used to match to the building wall section name. Use "`*`" for `<name>` to match to all names. For the building wall, the only attribute that is relevant is the `<color>` attribute.

The `alias::` and `type::` prefixes for `<ele_id>` are used to match to the `alias` and `type` string parameters of that can be set in the lattice file for each individual element.

If an element matches more than one shape, what is drawn depends upon the setting of `<multi>`. If `<multi>` is False (the default) for the first shape matched in the list of shapes, only this shape will be used. If `<multi>` is True, *Tao* will draw this shape and then look for additional matches. Each time an additional match is found, the shape is drawn and the setting of `<multi>` for that shape will be used to determine whether additional shapes are searched for. Thus `<multi>` can be use to draw, for example, a `circle` shape superimposed upon a `bow_tie` shape.

Tao defines a set of default shapes in case no shapes are defined in the plot file. If the optional `include_default_shapes` logical, which can be set for either `floor_plan` and/or `lat_layout` shape namelists, is set to False (the default), the default shapes are not used. If `include_default_shapes` is set to True, the default shapes are appended to the list of shapes.

Use the `show plot -floor_plan` and `show plot -lat_layout` commands to see the defined shapes. Use the `set floor_plan` and `set lat_layout` commands (§11.29)) to set shape parameters on the command line.

The width of a drawn shape is the width of the associated element. The exception is the `"x"` shape whose width is always the same as the height determined by the `<size>` setting.

`<size>` is the half height of the shape. That is, the size transverse to the longitudinal dimension. For `lat_layout` drawings, `<size>` = 1.0 corresponds to full scale if the default `graph%y%min` = -1 and `graph%y%max` = 1 are used. For floor_plan drawings, the drawn size is also affected by the setting of `graph%floor_plan%size_is_absolute` See §10.13.8 for more details.

The overall size of all the shapes can be scaled using the `plot_page` (§10.13) parameters
```
  floor_plan_shape_scale     ! For floor_plan drawings. Default = 1
  lat_layout_shape_scale     ! For lat_layout drawings. Default = 1
```
The text size in both `floor_plan` and `lat_layout` plots can be scaled by using the `plot_page` parameter

```
  legend_text_scale          ! Default = 1
```
Use the `show plot` command to view these parameters. Use the `set plot_page` command to set these parameters.

`<color>` is the color of the shape. Good colors to use are:
```
  "black"
  "blue"
  "cyan"
  "green"
  "magenta"
  "orange"
  "purple"
  "red"
  "yellow"
```

The `<line_width>` parameter is an integer that specifies the width of the lines drawn. The default is 1.

The `<offset>` parameter offsets the shape transverse to the reference orbit.

The `<label>` indicates what type of label to print next to the corresponding element glyph. Possibilities are:

```
  name            -- The element name (default).
  none            -- No label is drawn.
  s               -- Draw longitudinal s position.
```

The default is `"name"`

The `<draw>` field determines if a shape is drawn or not. The default is `T`. This can be useful for toggling on and off the drawing of shapes using the `set shape` command (§11.29).

Note: There is an old, deprecated syntax where both the lattice layout and floor plan drawings are specified via one `element_shapes` namelist.

The `<shape>` parameter is the shape of the figure drawn. The `<shape>` string will have the form:

```
  <shape-name>  or
  <prefix>:<shape-name>
```

Valid `shape-names` are:

```
  "box"             -- Rectangular box
  "bow_tie"         -- Bow-tie shape.
  "circle"          -- Circle centered at center of element.
  "diamond"         -- Diamond shape.
  <pattern_name>    -- Custom shape specified by <name>. Used with "pattern" prefix.
  "rbow_tie"        -- Bow-tie shape rotated 90 degrees.
  "d_triangle"      -- Triangle pointing ``down''.
  "l_triangle"      -- Triangle pointing ``left'' (upstream).
  "r_triangle"      -- Triangle pointing ``right'' (downstream).
  "u_triangle"      -- Triangle pointing ``up''.
  "x"               -- "X" centered at center of element
  "xbox"            -- Rectangular box with an x through it.
  "dashed_line"     -- Only used with ele_id set to "building_wall".
  "dash_dot_line"   -- Only used with ele_id set to "building_wall".
  "dotted_line"     -- Only used with ele_id set to "building_wall".
  "solid_line"      -- Only used with ele_id set to "building_wall".
```

Valid prefixes are:

```
  "asym_var"   -- Like "var" prefix but is not symmetric about the center line.
  "asym_vvar"  -- Like asym_var except scaled to associated variable or datum.
  "pattern"    -- Custom shape. <shape-name> here is a pattern name.
  "var"        -- Shape with variable height.
                       The shape size is symmetric about the center line.
  "vvar"       -- Like "var" prefix except scaled to associated variable or datum.
```

For example, if an element's shape is set to `var:box` or `asym_var:box`, the drawn size of the element is proportional to the element's magnetic or electric strength. The associated `<size>` setting is the multiplier used to scale from element strength to height. For example, for a quadrupole the height is proportional to the `K1` focusing strength. The difference between `var:box` or `asym_var:box` is that with `var:box` the drawn box is symmetric with respect to the centerline with a size independent of the sign of the element strength. On the other hand, with `asym_var:box`, the drawn box will terminate with one side on the centerline and the side on which it is drawn will depend upon the the sign of the element strength. Note: Not all lattice elements can be used with a `var:box` or `asym_var:box`.

A `vvar:box` shape is like a `var:box` and a `asym_vvar:box` is like a `asym_var:box`. The difference is that `vvar:box` and `asym_vvar:box` shapes may only be used when the `<ele_id>` is associated with data or variables. That is, when the `<ele_id>` string starts with "`data::`" or "`var::`". In this case, the height of the box, instead of being proportional to the strength of the element, is proportional to the value of the associated datum or variable. If no datum or variable component is specified in the `ele_id`, the model value will be used. Thus, in the above example, where `<ele_id>` was set to `"data::orbit.x|design"`, the design value is used.

The `solid_line`, `dashed_line`, `dash_dot_line`, and `solid_line` settings for `<shape>` is used when `<ele_id>` is set to `building_wall` to indicate what type of line is to be drawn.

The `pattern:<pattern_name>` shape allows for a custom pattern to be specified. Custom patterns are specified by a `shape_pattern` namelist:

```
&shape_pattern
  name = "<curve_name>"
  line%width = <line_width>
  pt(1) = <s>, <y>
  pt(2) = <s>, <y>
  pt(3) = ...
/
```

with `<s>` being the longitudinal coordinate and `<y>` begin the coordinate perpendicular to the longitudinal coordinate. Example:

```
&floor_plan_drawing
  ...
  ele_shape(2) = "quadrupole::*"     "pattern:q_pat"     "red"     0.75     "none"
  ...
/

&shape_pattern
  name = "q_pat"
  pt(1) = 0, -1
  pt(2) = 1, -1
  pt(3) = 0.9, 1
  pt(4) = 0.1, 1
  pt(5) = 0, -1
/
```

The `name` of the `shape_pattern` namelist (in this example it is "q_pat") must match the name given by `"pattern:<pattern_name>"`. The pattern is specified by a number of points. Between the points, a line segment is drawn. In the above example, the pattern is an isosceles trapezoid. When drawn, the `s` coordinate is scaled so that $s = 0$ corresponds to the entrance end of the element and $s = 1$ corresponds to the exit end. The `y` coordinate is scaled by the `size` attribute of the `ele_shape`. The color of the line segments is set by the definition and the width of the line segments is set by the pattern definition. Multiple `shape_pattern`s with the same name can be defined. In such a case, all patterns of a given name will be drawn. This allows the construction of more complex patterns. For example, a rectangle and a triangle drawn together.

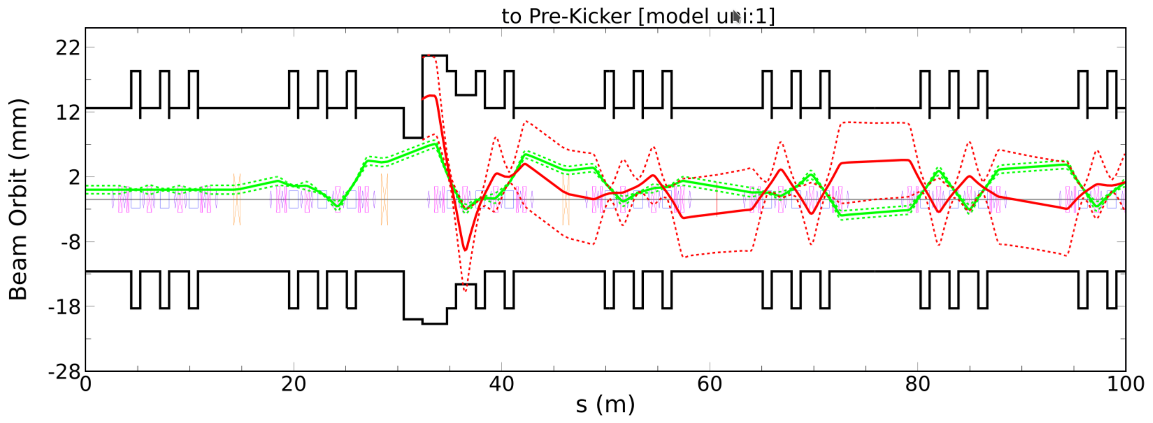


Figure 10.6: Example plot of the beam aperture. In this drawing, two turns of three injected particles are drawn. The particles start at different positions and illustrate what the size of an injected beam would be. Also drawn (a bit faint) is a `lat_layout` showing lattice element positions.

10.13.10 Aperture Drawing

Beam apertures can be defined in the *Bmad* lattice file. Apertures can be defined in one of three ways. The most common is to set limit or aperture parameters for an element. Another possibility is to use a `mask` element (which can be used to define an aperture of arbitrary shape). The third possibility involves defining a continuous three-dimensional wall. This third possibility is only used with Runge-Kutta type tracking.

To simplify things, the drawing of the beam aperture ignores any `mask` elements (since the geometry can be very complicated here) and ignores any three-dimensional walls (which are only used for Runge-Kutta type tracking). Fig. 10.6 shows an example of a aperture drawing.

To draw an aperture, a curve's `data_source` parameter must be set to `"aperture"` and the `data_type` parameter is set to one of

```
  "+x"     ! Aperture in +X direction
  "-x"     ! Aperture in -X direction
  "+y"     ! Aperture in +Y direction
  "-y"     ! Aperture in -Y direction
```

The apertures in the $+x$ and $+y$ directions will have positive values and the apertures in the $-x$ and $-y$ directions will have negative values. Set the curve's `y_axis_scale_factor` to scale the aperture curve if needed.

The following example will graphs the horizontal orbit along with the horizontal apertures.

```
  &tao_template_plot
    plot%name        = "x_orbit"
    plot%x_axis_type = "s"
    plot%n_graph     = 1
  /

  &tao_template_graph
    graph%name    = "x"
    graph_index   = 1

    curve(1)%data_source  = "aperture"
```

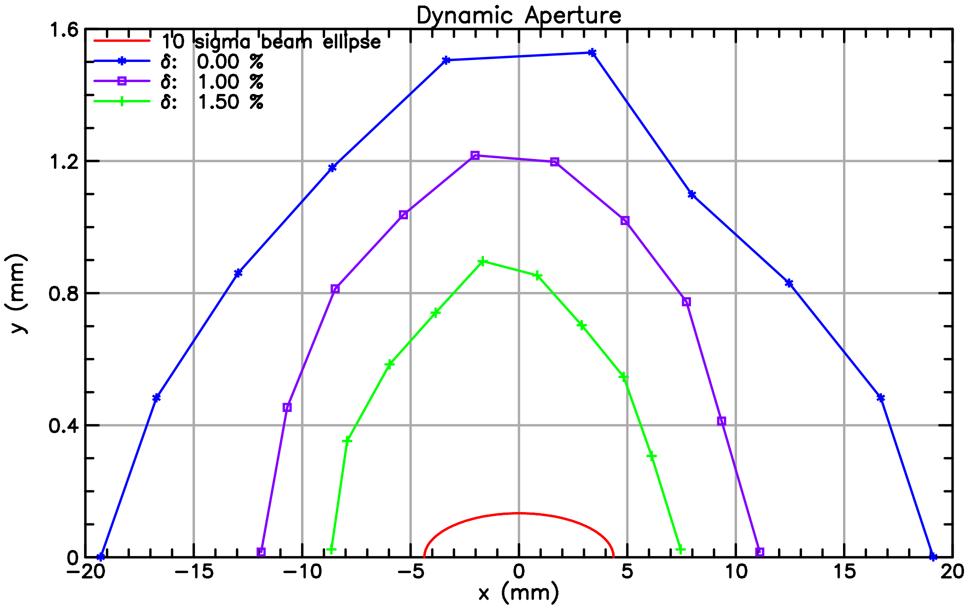


Figure 10.7: Example dynamic aperture plot.

```
    curve(1)%data_type    = "+x"
    curve(1)%draw_symbols = T
    curve(1)%draw_line    = F
    curve(1)%data_source  = "aperture"
    curve(2)%data_type    = "-x"
    curve(2)%draw_symbols = T
    curve(2)%draw_line    = F
    curve(3)%data_type = "orbit.x"
  /
```

Note: Aperture curves will ignore the `curve%component` parameter.

10.13.11 Dynamic Aperture Curve Drawing

A `dynamic_aperture` drawing displays the results of the dynamic aperture calculation (§10.12). Example plot setup:

```
&tao_template_plot
  plot%name = "da"
  plot%x_axis_type = "phase_space"
  plot%n_graph = 1
/

&tao_template_graph
  graph%name = "g1"
  graph%type = "dynamic_aperture"
  graph_index = 1
  graph%x%label = "x (mm)"
  graph%x_axis_scale_factor = 1000  ! Plot in mm.
  graph%y%label = "y (mm)"
  curve(1)%line%color = "red"
  curve(1)%data_type = 'beam_ellipse'
  curve(1)%line%width = 5
  curve(1)%legend_text = '10 sigma beam ellipse'
  curve(1)%draw_symbols = F
  curve(1:10)%y_axis_scale_factor = 10*1000  ! Plot in mm.
  curve(2:10)%draw_symbols = 9*T
  curve(2:10)%data_type = 9*"dynamic_aperture"
  curve(2:10)%data_index = 1, 2, 3, 4, 5, 6, 7, 8, 9
  curve(3)%symbol%color = "purple"
  curve(3)%line%color = "purple"
/
```

This will produce a plot similar to Fig. 10.7. Each curve represents a dynamic aperture scan at fixed initial momentum p_z.

Dynamic aperture curves can have the following `%data_type` parameters:

```
  "dynamic_aperture"        ! Curve points are with respect to the closed orbit (x,y).
  "dynamic_aperture_ref0"   ! Curve points are with respect to x = y = 0.
  "beam_ellipse"            ! Draws the beam ellipse within the [min_angle, max_angle] range.
  "beam_ellipse_full"       ! Draws the entire beam ellipse.
```

In the above example, the first curve has `%data_type` set to `"beam_ellipse"`. This results in the half ellipse in red since the [min_angle, max_angle] set in the `tao_dynamic_aperture` namelist for this example was [0, pi]. If the entire ellipse is desired to be drawn, the `%data_type` can be set to `"beam_ellipse_full"`. The scale of the ellipse drawn is set by the `ellipse_scale` parameter of the `tao_dynamic_aperture` namelist (§10.12). The default value is 10 so that a $10\,\sigma$ ellipse will be drawn by default. Also used in calculating the ellipse curve are the settings of `a_emit` and `b_emit` emittances also set in the same namelist.

There are 10 curves defined in the example. Curves 2 through 10 have `%data_type` set to `"dynamic_aperture"`. Each curve here represents one dynamic aperture scan. The scan index is set by a curve's `%data_index` parameter. An index of "1" denotes the first scan with the initial momentum set by the first value in the `pz` array set in the `tao_dynamic_aperture` namelist (§10.12). If a curve has an index that is greater than the number of scans, that curve is ignored.

10.13.12 Histogram Drawing

A `histogram` drawing displays a histogram of phase space beam density. Histogram plotting is associated with a `graph` by setting `graph%type` equal to `"histogram"`. The concepts here are similar to `phase space` plotting (§10.13.14). An example is shown in Fig. 10.8, using the example histogram template:

```
&tao_template_plot
  plot%name = "zhist"
  plot%n_graph = 1
/

&tao_template_graph
  graph_index = 1
  graph%name = "z"
  graph%type = "histogram"
  graph%box = 1, 1, 1, 1
  graph%title = "Bunch Histogram: Z"
  graph%margin =  0.15, 0.06, 0.12, 0.12, "%BOX"
```

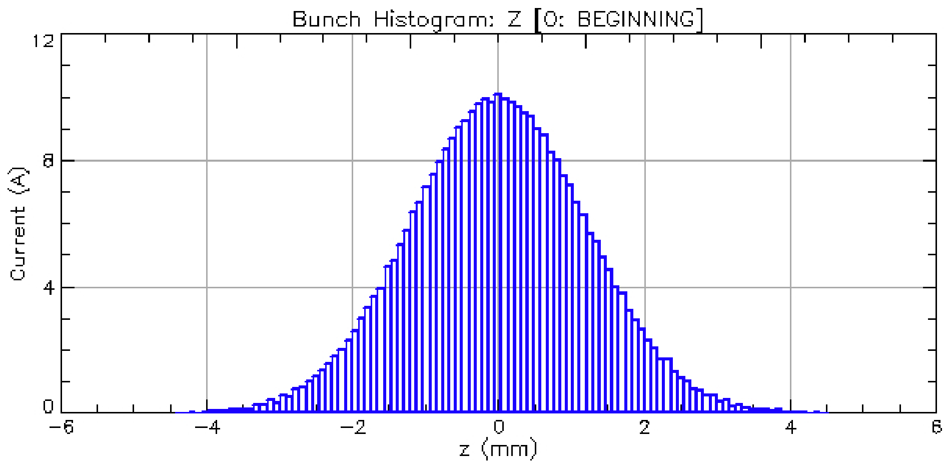


Figure 10.8: Example histogram plot.

```
  graph%x%min = -6
  graph%x%max =  6
  graph%x%label = "z (mm)"
  graph%y%label = "Current (A)"
  graph%y%label_offset = .1
  graph%x_axis_scale_factor = 1000.00 !m->mm

  curve(1)%hist%density_normalized = T
  curve(1)%hist%weight_by_charge = T
  curve(1)%hist%number = 100
  curve(1)%line%color = "blue"
  curve(1)%line%pattern = "dashed"
  curve(1)%y_axis_scale_factor = 299792458  !Q/m * c_light
  curve(1)%data_type = "z"
  curve(1)%data_source = "beam_tracking"
  curve(1)%ele_ref_name = "BEGINNING"
  curve(1)%symbol%type = "dot"
/
```

For a "histogram" type graph, curve%data_type determines what coordinate is plotted along the x-axis. Valid curve%data_type values are:

```
  "x", "px", "y", "py","z", "pz"   -- Phase space coordinates
  "intensity"                      -- Photon total intensity
  "intensity_x"                    -- Photon intensity along x-axis
  "intensity_y"                    -- Photon intensity along y-axis
  "phase_x"                        -- Photon phase along x-axis
  "phase_y"                        -- Photon phase along y-axis
```

In this example above, the x-axis of the plot will correspond to the z phase space coordinate.

The maximum and minimum of the bins is set automatically to fit the data. The curve%hist%number establishes the number of bins. Alternatively, if curve%hist%number = 0, then curve%hist%width establishes the width of the histogram bins and sets the number automatically.

If curve%hist%density_normalized = T, then the height of a bin will be divided by its width. If `curve%hist%weight_by_charge = T`, then the particle charge will be used to bin, otherwise the particle count will be used to bin.

The `curve%hist%center` will insure that a bin will be centered at this location.

To change the place in the lattice where the data for the `histogram` is evaluated, use the `set curve ele_ref_name` command.

If `graph%type` is `"histogram"` then `curve%data_source` must be either:

```
  "beam"
  "multi_turn_orbit"
  "rel_multi_turn_orbit"
```

`"beam"` indicates that the points of the histogram plot will be obtained correspond to the positions of the particles within a tracked beam.

Setting `curve%data_source` to `"multi_turn_orbit"` or `"rel_multi_turn_orbit"` is used for rings where a single particle is tracked multiple turns and the position of this particle is recorded each turn. The number of turns is determined by the setting of `curve%n_turn`. The starting position for the tracking is set by setting the `particle_start[x]`, `particle_start[px]`, etc. parameters in the lattice file (see the *Bmad* manual for details). The difference between `"multi_turn_orbit"` and `"rel_multi_turn_orbit"` is that `"rel_multi_turn_orbit"` is drawn relative to the phase space point

$$p_{z0} \cdot (\eta_x, \eta_x', \eta_y, eta_y', 0, 0) \tag{10.4}$$

where p_{z0} is the value of the particle's initial p_z and η is the dispersion at the reference element where the orbit is being evaluated for the plot.

10.13.13 Key Table Drawing

A `key table` displays information about variables bound to keyboard keys §12.1. Key bindings are used in `single mode`. An example is shown in Figure 10.3. A template to create a key table looks like:

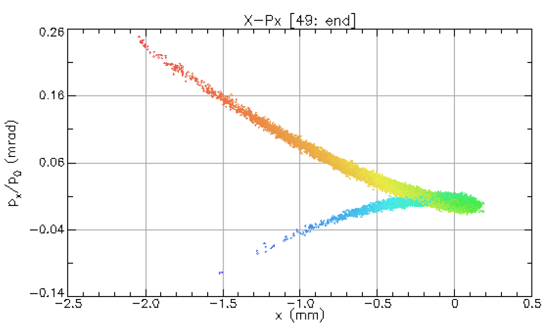
```
  &tao_template_plot
    plot%name = "table"
    plot%n_graph = 1
  /

  &tao_template_graph
    graph%type = "key_table"
    graph_index = 1
  /
```

The number in the upper left corner, to the left of the first column, (`1` in Fig. 10.3) shows the active `key bank`. The columns in the Key Table are:

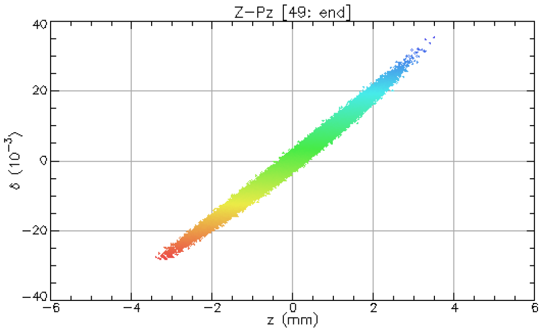
```
  Ix          ! Key index.
  Name        ! Element name whose attribute is bound.
  Attrib      ! Name of the element attribute that is bound.
  Value       ! Current value of bound attribute.
  Value0      ! Initial value of bound attribute.
  Delta       ! Change in value when the appropriate key is pressed.
  Uni         ! Universe that contains the element.
  Opt         ! Shows if bound attribute is used in an optimization.
```

Note that in a `Lattice Layout`, if a displayed element has a bound attribute, then the key index number will be displayed just above the element's glyph.

The `key_table` is drawn with respect to the upper left hand corner of the region in which it is placed.



(a) Horizontal phase space



(b) Longitudinal phase space

Figure 10.9: Example Phase Space plot, with points colored by the pz coordinate.

10.13.14 Phase Space Plotting

A phase space plot displays a particle or particles phase space coordinates at a given location. Phase space plotting is associated with a graph by setting graph%type equal to "phase_space". The concepts here are similar to data plotting (§7.5.5). An example is show in Figure 10.9. Example Phase Space template:

```
&tao_template_plot
  plot%name = "xphase"
  plot%n_graph = 1
/

&tao_template_graph
  graph_index = 1
  graph%name = "x"
  graph%type = "phase_space"
  graph%box = 1, 1, 1, 1
  graph%title = "X-Px"
  graph%margin =  0.15, 0.06, 0.12, 0.12, "%BOX"
  graph%x%min =   -2.5
  graph%x%max = 0.5
  graph%x%label = "x (mm)"
  graph%x_axis_scale_factor = 1000.00 !m->mm
  graph%y%label =  "p\dx\u/p\d0\u (mrad)"
  graph%y%major_div = 4
  graph%y%label_offset=.4
  curve(1)%data_type_x = "x"
  curve(1)%data_type   = "px"
  curve(1)%y_axis_scale_factor = 1000 !rad->mrad
  curve(1)%data_source = "beam_tracking"
  curve(1)%ele_ref_name = "END"
  curve(1)%symbol_every = 10
  curve(1)%symbol%type = 1
  curve(1)%z_color%data_type = "pz"
  curve(1)%z_color%is_on = T
  /
```

By setting %symbol_every to something greater than 1, only a subset of the particles are used for plotting.

For a "phase_space" type graph, curve%data_type_x determines what phase space coordinate is plotted along the x-axis and curve%data_type determines what phase space coordinate is plotted along the y-axis. Phase space coordinates are one of:

```
  "x",  "px",  "y",  "py",  "z",  "pz",           ! Phase space coordinates
  "time",                                         ! Particle time
  "bunch_index",                                  ! Index of bunch particle is in.
  "energy",                                       ! Total particle energy
  "Ja", "Jb"                                      ! Action coordinate in action-angle coords.
  "intensity",  "intensity_x",  "intensity_y"     ! Photon intensity
  "phase_x", "phase_y"                            ! Photon coherent phase
```

In this example above, the x-axis of the plot will correspond to the z phase space coordinate and the pz-axis will correspond to the px coordinate.

To change the place in the lattice where the data for the phase_space curve is evaluated, use the set curve ele_ref_name command.

Points can be colored by another phase space coordinate by activating z_color%is_on = T. The available curve options and defaults for curve(N)%z_color components are:

```
  %is_on = F
  %data_type = ""
  %min = 0
  %max = 0
  %autoscale = T
```

These can be the init file, or in Tao using the set curve command. The %data_type can be set to any of the available phase space coordinates. %min and %max specify the minimum and maximum of this coordinate to be used in the color range. Values above or below this range will be colored Black or Grey, respectively. If %autoscale = True, then these will be set automatically based on the limits of the %data_type coordinate.

If graph%type is "phase_space" then curve%data_source must be either:

```
  "beam"
  "multi_turn_orbit"
  "twiss"
```

"beam" indicates that the points of the phase space plot will be obtained correspond to the positions of the particles within a tracked beam. multi_turn_orbit" is used for rings where a single particle is tracked multiple turns and the position of this particle is recorded each turn. In this case, a d2_data structure must have been set up to hold the turn–by–turn orbit. This d2_data structure must be called multi_turn_orbit and must have d1_data data arrays for the phase space planes to be plotted. For example, if the phase space plot is x versus px, then there must be d1_data arrays named "x" and "px". The number of turns is determined by the setting of ix_max_data in the tao_d1_data namelist (§10.10). Using "twiss" as the curve%data_source indicates that the phase space plot will be an ellipse whose shape is based upon the Twiss and coupling parameters, and the normal mode emittances. If the normal mode emittances have not been computed then a nominal value of 1 μm-rad is used.

Chapter 11

Commands

Tao has two `modes` for entering commands. In `line mode`", described in this chapter, *Tao* waits until the `return` key is depressed to execute a command. That is, a command consists of a single line of input. Conversely, `Single Mode`, which is described in `Single Mode` chapter (§12), interprets each keystroke as a command. Single Mode is useful for quickly varying parameters to see how they affect a lattice but the number of commands in Single Mode is limited. To put *Tao* into `single mode` use the `single_mode` command (§11.31).

The syntax for `line mode` commands is discussed in Section §4.1. The list of commands is shown in Table 11.1.

This chapter uses the following special characters to define the command line syntax:

```
  {}        ! Identifies an optional argument.
            !   Arguments now enclosed in brackets are required
  <>        ! Indicates a non-literal argument.
```

Example:

```
  change {-silent} variable <name>[<locations>] <number>
```

Here the `-silent` argument is optional while the `variable` argument is mandatory. Appropriate values for `<name>`, `<locations>`, and `<number>` must be substituted. A possible

```
  change var steering[34:36] @1e-3  ! set the steering strength #34-36 to 0.001
```

When running *Tao*, use the `help` (§11.15) command to show documentation on any command. For example, `help plot` will show documentation on the `plot` command.

Command	*Section*	*Command*	*Section*
`alias`	§11.1	`quit`	§11.22
`call`	§11.2	`re_execute`	§11.23
`change`	§11.3	`read`	§11.24
`clear`	§11.4	`reinitialize`	§11.25
`clip`	§11.5	`restore`	§11.26
`continue`	§11.8	`run_optimizer`	§11.27
`create`	§11.6	`scale`	§11.28
`cut_ring`	§11.7	`set`	§11.29
`derivative`	§11.12	`show`	§11.30
`do, enddo`	§11.9	`single_mode`	§11.31
`end_file`	§11.10	`spawn`	§11.32
`exit`	§11.11	`taper`	§11.33
`fixer`	§11.13	`timer`	§11.34
`flatten`	§11.14	`use`	§11.35
`help`	§11.15	`veto`	§11.36
`ls`	§11.16	`view`	§11.37
`pause`	§11.17	`wave`	§11.38
`pipe`	§11.18	`write`	§11.39
`place`	§11.19	`x_axis`	§11.40
`ptc`	§11.20	`x_scale`	§11.41
`python`	§11.21	`xy_scale`	§11.42

Table 11.1: Table of *Tao* commands.

11.1 alias

The `alias` command defines command shortcuts. Format:

```
  alias {<alias_name> <string>}
```

`Alias` is like Unix aliases. Using the `alias` command without any arguments results in a printout of the aliases that have been defined. When using an alias up to 9 arguments may be substituted in the `<string>`. The ith argument is substituted in place of the sub-string "[[i]]" or "[<i>]". Arguments that do not have a corresponding "[[i]]" or "[<i>]" are placed at the end of `<string>`. The difference between "[[i]]" and "[<i>]" is that "[[i]]" is a required argument while "[<i>]" defines an optional argument. For example

```
  alias aaa show element [[1]] [[2]]
  alias zzz show element [[1]] [<2>]
```

This defines "`aaa`" as an alias for the `show element` command with two required arguments while "`zzz`" has only one requred argument.

Aliases can be set up for multiple commands using semicolons.

Examples:

```
  alias xyzzy plot [[1]] model   ! Define xyzzy
  alias                          ! Show all aliases
  xyzzy top                      ! Use an alias
  plot top model                 ! Equivalent to "xyzzy top"
  xyzzy top abc                  ! Equivalent to "plot top model abc"
  alias foo  show uni; show top ! "foo" equivalent to "show uni; show top"
```

In the above example “xyzzy” is the alias for the string “plot [[1]] model”. When the command xyzzy is used “top” is substituted for “[[1]]” in the string.

11.2 call

The `call` command opens a command file (§2.6) and executes the commands in it. Format:

```
call <filename> {<arg_list>}
call -no_calc {<arg_list>}
call -ptc <filename>
```

The `call` command without `-ptc` is for running a set of *Tao* commands. Up to 9 arguments may be passed to the command file. The ith argument is substituted in place of the string “[[i]]” in the file. Nesting of command files (command files calling other command files) is allowed. There is no limit to the number of nested files. See the `Command Files and Aliases` section (§2.6) for more details.

The `call -ptc` command passes the command file to PTC for processing. Previous to such a call, the command `ptc init` must be issued. This is for PTC wizards only.

If a command file calls another command file, and the name of the second command file has a relative (as opposed to absolute) path name, *Tao* will look for the second command file relative to the directory of the first command file. To have *Tao* look relative to your current working directory (where you started *Tao*), use the prefix `$PWD/`. For example, to call a command file that is one level up from your current working directory use

```
call $PWD/../second.cmd
```

Command loops can be implemented in a command file. See the documentation on `do/enddo` (§11.9) for more details.

The `-no_calc` option, can be use to speed up execution time by halting all lattice and plotting calculations. This is equivalent to setting the `global` parameter `lattice_calc_on` to False. at the beginning of the command file and then setting this parameter to its initial state after the file has been run.

To suppress all the output when running a command file use the command:

```
set global quiet = all       ! Suppress everything except errors
set global quiet = warnings  ! Suppress just warnings.
set global quiet = off       ! No suppression
```

Note: if `quiet` is set in a command file, the setting will persist to the end of the file and then revert to what it was before the command file was run.

Examples:

```
  call -no my_cmd_file abc def
```

In the above example the argument “abc” is substituted for any “[[1]]” appearing the file and “def” is substituted for any “[[2]]”.

11.3 change

The `change` command changes element attribute values or variable values in the `model` lattice. Format:

```
change {-update} element <element_list> <attribute> {prefix>} <number>
change {-silent} variable <name>[<locations>] {<prefix>} <number>
change {n@}particle_start <coordinate> {prefix>} <number>
change {-branch <branch_list>} {-listing} {-mask <veto_list>} tune {dQa} {dQb}
change {-branch <branch_list>} z_tune dQz
```

The `change` is used for changing real (as opposed to integer or logical) parameters. Also consider using the `set` command (§11.29) which is more general.

If `<prefix>` is not present, `<number>` is added to the existing value of the attribute or variable. That is:

```
final_model_value = initial_model_value + <number>
```

If `<prefix>` is present, it may be one of

```
@       final_model_value = <number>
d       final_model_value = design_value + <number>
%       final_model_value = initial_model_value * (1 + <number> / 100)
```

Element list format (§4.3), without any embedded blanks, is used for the `<element_list>` argument.

For `change particle_start`, The optional `n@` universe specification (§3.3) may be used to specify the universe or universes to apply the change command to.

For lattices with an open geometry, `change particle_start <coordinate> <number>` can be used to vary the starting coordinates for single particle tracking. If the `use_particle_start` of the `beam_init` structure (§10.7) is set to True, `particle_start` will also vary the beam centroid and the beam particle spin for tracking. Here `<coordinate>` is one of:

```
x, px, y, py, z, pz, t
```

For photons, `<coordinate>` may also be:

```
e_photon, field_x, field_y, phase_x, phase_y
```

For closed lattices only the `pz` component is applicable. For lattices that have an `e_gun` (which necessarily implies that the lattice has an open geometry), the time `t` coordinate must be varied instead of `pz`. Note: Setting `particle_start` will affect the orbit at the active `fixer` element and the default active fixer element is the `beginning` element.

For open lattices, `change element beginning <twiss>` can be used to vary the starting Twiss parameters where `<twiss>` is one of:

```
beta_a, beta_b, alpha_a, alpha_b
eta_a, eta_b,etap_a, etap_b
```

The `change z_tune` command will vary the longitudinal tune by `<dQz>`. The `<branch_list>` is used to select which lattice branches the tune is varied in. Each branch listed can have an optional universe prefix. The default is to vary branch 0 of the current default universe.

The `change tune` command will vary the transverse tunes by `<dQa>` and `<dQb>` and the `change z_tune` command will vary the longitudinal tune by `<dQz>`. Units are in radians/2pi With the `change tune` command, if `<dQa>` or `<dQb>` is not given, the value will be taken to be zero (that is, no change). The `<branch_list>` is a list of lattice branches with optional universe prefix, to vary the tunes. The `<veto_list>` of the `-mask` option gives a list of quadrupoles *not* to use for varying the tune. See the `set tune` (§11.29.27) command for more details. The `-listing` option, if present, will, in addition to the tune change, generate a list of quadrupoles varied along with variation coefficients.

The `-silent` switch, if present, suppresses the printing of what variables are changed.

The `-update` switch, if present, suppresses *Tao* from printing error messages if a "variable slave value mismatch" is detected (§5.4). Independent of whether `-update` is present or not, *Tao* will fix the mismatch using the changed value to set all of the slave values.

Note: The `change element` command can be used with `ramper` type elements.

Examples:
```
  change ele 3@124 x_offset 0.1        ! Offset element #124 in universe 3 by 0.1
  change ele 1,3:5 x_offset 0.1        ! Offset elements 1, 3, 4, and 5 by 0.1
  change ele q* k1 d 1.2e-2            ! Set the k1 strength of all elements starting with
                                       !   the letter "q" relative to the design
  change ele quadrupole::* k1 d 1.2e-2 ! Set the k1 strength of all quadrupole elements.
  change var steering[34:36] @1e-3     ! set the steering strength #34-36 to 0.001
  change var steering[*] %10           ! vary all steering strengths by 10%
  change 2@particle_start x @0.001     ! set beginning x position in universe 2 to 1 mm.
  change -mask Q1* tune 0 0.01         ! Change transverse tunes without using quadrupoles
                                       !   whose names start with "Q1".
  change -branch 2@1 z_tune 0.02       ! Change z-tune of branch #1 of universe #2.
```

11.4 clear

The `clear` command clears stored spin and orbital Taylor maps from all elements in a lattice with the exception of `Taylor` elements (which are specified in the lattice file as opposed to being calculated by *Bmad*). Format:
```
  clear maps
```
Clearing the Taylor maps may be needed if the maps are in use (for example, with a spin polarization calculation) and orbit excursions place the calculated orbit outside of the range of validity of the maps.

11.5 clip

The `clip` command vetoes data points for plotting and optimizing. That is, the `good_user` logical of the data associated with the out-of-bound plotted points are set to False. Format:
```
  clip {-gang} {<where> {<limit1> {<limit2>}}}
```

Which graphs are clipped is determined by the `<where>` switch. If `<where>` is not present, all graphs are clipped. If `where` is a plot name, then all the graphs of that plot are clipped. If `where` is the name of a `d2_data` (for example, `orbit`) or a `d1_data` (for example, `orbit.x`) structure, then those graphs that display this data are clipped.

The points that are clipped those points whose y values are outside a certain range defined by `<limit1>` and `<limit2>`. If neither `<limit1>` nor `<limit2>` are present, the clip range is taken to be outside the graph minimum and maximum y–axis values. If only `<limit1>` is present then the clip range is outside the region from `-<limit1>` to +`<limit1>`. If both are present than the range is from `<limit1>` to `<limit2>`.

The `-gang` switch is apply a clip to corresponding data in a `d2_data` structure. For example
```
  clip -g orbit.x   ! Clips both orbit.x and orbit.y
```
Here the `orbit.x` data is clipped and the corresponding data in `orbit.y` is also vetoed. For example, if datum number 23 in `orbit.x` is clipped, datum number 23 in `orbit.y` will be vetoed.

Examples:
```
  clip top.x -3  7  ! Clip the curves in the x graph in the region named "top".
  clip bottom       ! Clip the graphs in the "bottom" region
  clip -g orbit.x   ! Clip the orbit.x graph and also veto corresponding points
                    ! in other graphs of the orbit plot.
```

11.6 create

Format:

```
  create data ... ! §11.6.1
```

The `create` command constructs various tao objects that would have otherwise been created in tao initialization.

11.6.1 create data

The `create data` constructs a new `d2_data` and one or more `d1_data` for it. The primary purpose of this command is to make more data available for design and optimization in an interactive session; for repeated use, create data using data initialization namelists (§10.10). The resulting data must be initialized with the `set data` command (§11.29.7).

Syntax:

```
  create data d2_name d1_name[ix_min:ix_max] ...
```

You may have as many `d1_names` as needed. `ix_min` and `ix_max` must be literal integers, not expressions.

11.7 cut_ring

Format:

```
  cut_ring {-particle_start} {-static} {-zero}
```

The `cut_ring` command is used to toggle the geometry of the viewed `model` lattice between `closed` to `open`.

When the lattice is toggled to an open geometry, the `-particle_start`, `-static` or `-zero` options can be used to set the starting orbit. In all cases, the starting orbit is set equal to the setting of `particle_start` where `particle_start` can be set in the lattice file and/or using *Tao*'s `set particle_start` or `change particle_start` commands.

With the `-static` option (the default), `particle_start` is set to the same orbit as currently exists for the closed orbit. The exception is if no closed orbit is found. In this case, `particle_start` is not modified (same as with the `particle_start` option).

With the `-zero` option, the `particle_start` orbit is set to zero.

With the `-particle_start` option, the `particle_start` orbit is not modified.

```
  cut -zero   ! When lattice geometry is toggled open: zero initial orbit.
```

11.8 continue

The `continue` command is used to continue reading of a suspended command file (§2.6) after a `pause` command (`s:pause`). Format:

```
  continue
```

11.9 do/enddo command file looping

Command loops can be implemented in a command file files. Format:

```
do <var> = <l_bound>, <u_bound> {, <incr>}
  ...   ! use the syntax ‘‘[[<var>]]’’ to refer to a variable.
enddo
```

Note: “`enddo`” is one word and my not be split into two words. Loops can be nested and the number of levels is not unlimited.

A loop will execute the code in between the `do` and `enddo` lines a certain number of times. Each time trough the the the integer variable `<var>` will be incremented by `<incr>`, starting at `<l_bound>` and stopping before `<var>` is greater than `<u_bound>`. If `<incr>` is not present, the increment will be 1. Note: `<l_bound>`, `<u_bound>`, and `<incr>` must all be integers.

Example:

```
do j = 0, 10, 2
  set particle_start pz = 1e-3 * [[j]]
  ...
enddo
```

As shown in the above example, to refer to a loop variable in a command, use the syntax “[[<var>]]”.

11.10 end_file

The `end_file` command is used in command files (§2.6) to signal the end of the file. Everything after an `end_file` command is ignored. An `end_file` command entered at the command line will simply generate an error message. Format:

```
end_file
```

11.11 exit

The `exit` command exits the program. Same as `Quit`. Format:

```
exit
```

11.12 derivative

The `derivative` command calculates the `dModel_Data/dVar` derivative matrix needed for the `lm` optimizer. Format:

```
derivative
```

11.13 fixer

The `Fixer` command can be used to set the active fixer and can be used to transfer fixer parameters from the from the actual to the stored. See the *Bmad* manual for documentation on fixer elements. Command format:

```
  fixer activate <fixer>
  fixer save <fixer> {<parameters>}
  fixer write <fixer> {<file-name>}
```
here `<fixer>` is the name of a fixer element (Note: For the purposes of this discussion, the `BEGINNING` element is considered a fixer element).

The basic stored parameters are:
```
  spin_x_stored, spin_y_stored, spin_z_stored     ! Spin group

  x_stored, px_stored
  y_stored, py_stored                             ! phase space group
  z_stored, pz_stored

  beta_a_stored, alpha_a_stored, phi_a_stored     ! a-mode Twiss group
  dbeta_dpz_a_stored, dalpha_dpz_a_stored

  beta_b_stored, alpha_b_stored, phi_b_stored     ! b-mode Twiss group
  dbeta_dpz_b_stored, dalpha_dpz_b_stored

  eta_x_stored, etap_x_stored                     ! Horizontal dispersion group
  deta_dpz_x_stored, detap_dpz_x_stored

  eta_y_stored, etap_y_stored                     ! Vertical dispersion group
  deta_dpz_y_stored, detap_dpz_y_stored

  mode_flip_stored                                ! Logical: Normal modes flipped?
  cmat_11_stored, cmat_12_stored                  ! Coupling group (includes mode flip)
  cmat_21_stored, cmat_22_stored
```
In addition, the following groups of stored values are recognized:
```
  all (or blank string)   ! All parameters
  twiss                   ! All Twiss (non-orbit) parameters
  a_twiss                 ! a-mode Twiss group
  b_twiss                 ! b-mode Twiss group
  dispersion              ! Both horizontal and vertical dispersion groups
  x_dispersion            ! Horizontal dispersion group
  y_dispersion            ! Vertical dispersion group
  cmat                    ! Coupling group
  chromatic               ! dbeta, dalpha, deta, and detap parameters
  spin                    ! Spin group
  phase_space             ! Phase space group
  orbit                   ! Phase space and spin groups
  x_plane                 ! x and px
  y_plane                 ! y and py
  z_plane                 ! z and pz
```
The `activate` subcommand activates (turns on) a given fixer element. When a fixer element is activated, the stored Twiss and orbit parameters will be used to set the actual Twiss and orbit.

The `save` subcommand takes actual parameter values and saves them in the corresponding stored parameters. Either an individual or a parameter group may be used for `<parameters>`.

Note: To set stored parameters to arbitrary values, use the `set element` (§11.29.10), or `change element` (§11.3) commands. Note: Changing stored values does not affect the actual values. To load stored values to actual use the `fixer activate` command.

The `write` subcommand write a file of fixer stored values. This file is in Bmad lattice format. The default file name is `<fixer>.bmad` where `<fixer>` is the name of the fixer element.

For example:

```
fixer save f1 orbit    ! Save orbit parameters in the fixer named f1.
fixer activate f1      ! Set fixer f1 to be active.
fixer write f1         ! Write fixer stored values in Bmad lattice file f1.bmad.
set ele f1 x_stored = 0.001  ! Set x_stored parameter.
```

11.14 flatten

The `Flatten` command runs the optimizer to minimize the merit function. This is the same as the `run_optimizer` command. See the `run_optimizer` command for more details. Format:

```
flatten {<optimizer>}
```

11.15 help

The `help` command gives help on *Tao* commands. Format:

```
help {<command> {<subcommand>}}
```

The `help` command without any arguments gives a list of all commands. Some commands, like `show`, are so large that help on these commands is divided up by their subcommand.

Examples:

```
help            ! Gives list of commands.
help run        ! Gives help on the run_optimizer command.
help show       ! Help on the show command.
help show alias ! Help on the show alias command.
```

The `help` command works by parsing the file `$TAO_DIR/doc/command-list.tex` which is the LaTeX file for the `Tao Commands` chapter of the *Tao* manual. For the `help` command to work properly, the environment variable `TAO_DIR` must be appropriately defined. Generally, `TAO_DIR` will be defined if the appropriate *Bmad* setup script has been run. For "Distributions", this is the same setup script used to setup a distribution. See your local *Bmad* guru for details.

When the `help` command parses the `$TAO_DIR/doc/command-list.tex` file, LaTeX syntax will be modified to produce a reasonable looking output on the terminal. This translation is not perfect so reference should be made to the *Tao* manual if there is a problem in the translation.

11.16 ls

The `ls` command is the same as the standard UNIX `ls` command to display a list of files and directories. The standard ls switches are accepted. This is equivalent to the `spawn ls` command. Format:

```
ls {switches}
```

Example:

```
ls -lrt
```

11.17 pause

The `pause` command is used to pause *Tao* when executing a command file (§2.6). Format:

```
pause {<time>} ! Pause time in seconds.
```

If `<time>` is not present or zero, *Tao* will pause until the `CR` key is pressed. Once the `CR` key is pressed, the command file will be resumed. If `<time>` is negative, *Tao* will suspend the command file. Commands can now be issued from the keyboard and the command file will be resumed when a `continue` command (§11.8) is issued. Multiple command files can be simultaneously suspended. Thus, while one command file is suspended, a second command file can be run and this command file too can be suspended. A `continue` command will resume the second command file and when that command file ends, another `continue` command will be needed to complete the first suspended command file. Use the `show global` command to see the number of suspended command files.

Example:

```
pause 1.5    ! Pause for 1.5 seconds.
pause -1     ! Suspend the command file until a continue
             !   command is issued.
```

11.18 pipe

The `pipe` command is like the `show` command in that the `pipe` command prints information to the terminal. The difference is that the output from the `show` command is meant for viewing by the user while the output of the `pipe` command is meant for easy parsing. Format:

```
pipe {-append <file_name>} {-noprint} <subcommand> <arguments>
pipe {-write <file_name>} {-noprint} <subcommand> <arguments>
```

The `pipe` command has `-append` and `-write` optional arguments which can be used to write the results to a file. The `pipe -append` command will appended to the output file. The `pipe -write` command will first erase the contents of the output file. Example:

```
pipe -write d2.dat data_d2    ! Write to file "d2.dat"
```

The `-noprint` option suppresses printing and is useful when writing large amounts of data to a file. The `pipe` command can be used to pass information to a parent process when *Tao* is run as a subprocess. The parent process may be any scripting program like Pipe, Perl, Tcl, etc. In particular, see the `Python Interface` chapter (§13) for details on how to run *Tao* as a Python subprocess.

In terms of long term maintainability, the advantage of using the `pipe` command in the scripts over the `show` command comes from the fact that the output syntax of `show` commands can (and does) change.

Note to programmers: For debugging, the `show internal -pipe` command will show the `c_real` and `c_integer` arrays.

Possible `<subcommand>` choices are:

```
beam, beam_init, branch1, bunch_comb, bunch_params, bunch1, bmad_com,
building_wall_list, building_wall_graph, building_wall_point, building_wall_section,
constraints, da_params, da_aperture, data, data_d2_create, data_d2_destroy,
data_d_array, data_d1_array, data_d2, data_d2_array, data_set_design_value,
data_parameter, datum_create, datum_has_ele, derivative,
ele:ac_kicker, ele:cartesian_map, ele:chamber_wall, ele:control_var,
ele:cylindrical_map, ele:elec_multipoles, ele:floor, ele:gen_attribs,
ele:gen_grad_map, ele:grid_field, ele:head, ele:lord_slave, ele:mat6, ele:methods,
ele:multipoles, ele:orbit, ele:param, ele:photon, ele:spin_taylor, ele:taylor,
ele:twiss, ele:wake, ele:wall3d, em_field, enum, evaluate, floor_plan,
floor_orbit, global, global:opti_de, global:optimization, global:ran_state,
help, inum, lat_branch_list, lat_calc_done, lat_ele_list, lat_header, lat_list,
lat_param_units, lord_control, matrix, merit, orbit_at_s, place_buffer,
plot_curve, plot_curve_manage, plot_graph, plot_graph_manage, plot_histogram,
plot_lat_layout, plot_line, plot_list, plot_symbol, plot_template_manage,
plot_transfer, plot1, ptc_com, ring_general,
shape_list, shape_manage, shape_pattern_list, shape_pattern_manage,
shape_pattern_point_manage, shape_set, show, slave_control, space_charge_com,
species_to_int, species_to_str, spin_invariant, spin_polarization, spin_resonance,
super_universe, taylor_map, twiss_at_s, universe, var_v1_create, var_v1_destroy,
var_create, var_general, var_v1_array, var_v_array, var, wall3d_radius, wave
```

When running *Tao*, to see documentation on any of the subcommands, use the command:

```
help pipe <subcommand>
```

For example,

```
help pipe ele:orbit
```

will show information on the `pipe ele:orbit` subcommand.

11.19 place

The `place` command is used to associate a `<template>` plot with a `<region>` and thus create a visible plot in that region. Format:

```
place {-no_buffer} <region> <template>
place <region> none
place * none
```

If `<region>` is set to "`*`" then all regions are selected.

If `<template>` is set to "`none`" all selected regions are cleared of plots.

The `-no_buffer` optional switch is used when external plotting is being done (EG with a GUI) and is not of interest otherwise.

Notice that by using multiple `place` commands a `template` can be associated with more than one region. For example, if multiple orbit plots are desired.

Examples:

```
place * none     ! Erase all plots.
place top orbit  ! Place the orbit template in the top region
place top none   ! Erase any plots in the top region
```

11.20 ptc

The `ptc` command is used manipulating PTC layouts associated with Bmad lattices. Format:

```
ptc init           ! Init associated PTC layout.
ptc reslice        !
```

The `ptc init` command initializes a PTC layout.

The `ptc reslice` command calculates good values for lattice element `num_steps` and `integrator_order`. This command does not adjust the following elements since the algorithm for the calculation can be problematical when the field is varying longitudinally within an element:

```
rfcavity, lcavity, crab_cavity
wiggler, undulator
```

Also see:

```
call -ptc <file>         ! Run a PTC script
read ptc                 ! Read a PTC lattice
write ptc                ! Write a PTC lattice
```

Examples:

```
ptc init
```

11.21 python

`Python` is the old name for the `pipe` command. For backwards compatibility, the old name is still accepted.

11.22 quit

`Quit` exits the program. Same as `exit`. Format:

```
quit
```

11.23 re_execute

The `re_execute` command reruns prior commands. Format:

```
re_execute <index>   ! Re-execute a command with the given index number.
re_execute <string>  ! Re-execute last command that begins with <string>.
```

Every *Tao* command entered is recorded in a “history stack”. These commands can be viewed using the `show history` command. The `show history` command will also display the index number associated with each command.

Note: The up and down arrow keys on the keyboard can be used to scroll through the command history stack.

Examples

```
re_exe 34   ! Re-execute command number 34.
re_exe set  ! Re-execute last ‘‘set’’ command.
```

11.24 read

The `read` command is used to modify the (*Bmad*) `model` lattice. Format:

```
read lattice {-silent} {-universes <universe-list>} <file_name>
```

With the `read lattice` command, the `model` lattices contained in the universes specified by `<universe-list>` are modified using a "secondary lattice" file. [See the *Bmad* manual for the definition of "secondary lattice".] For example, with the appropriate file, the `read` command can be used to misalign the lattice elements. For the `read lattice` command, the input file must be in Bmad standard lattice format.

If `-universes` is not present, only the `model` lattice in the default universe is modified.

If, after the lattice file has been read in, a given *Tao* variable has slave parameters that have different values there is a problem. For example, if a *Tao* variable controls the `k2` value of sextupoles elements `S1` and `S2`, and if `S1` is set to a different value than `S2`, there is an inconsistency which needs to be corrected. This can be done in a number of ways. For example, by using the `set ele -update` command or using a further `read lattice` command with a lattice that corrects the problem.

If desired, the `-silent` switch can be used to suppress error messages about differing *Tao* variable slave parameter values.

Note: Due to bookkeeping complications, the number of lattice elements may not be modified. If it is desired to initiate *Tao* using both "primary" and secondary lattice files, this can be done as illustrated in §10.4.

Examples:

```
  read lat -uni * lat.bmad   ! Modify model lattice of all universes.
  read lat -uni 2,3 lat.bmad ! Modify model lattice universes 2 and 3.
```

11.25 reinitialize

The `reinitialize` command reinitializes various things. Format:

```
  reinitialize beam
  reinitialize data
  reinitialize tao {-clear} {command line optional arguments}
```

The `reinitialize beam` command reinitializes the beam at the start of the lattice. That is, a new random distribution is generated. Note: This also reinitializes the model data.

`reinitialize data` forces a recalculation of the model data. Normally, a recalculation is done automatically when any lattice parameter is changed so this command is generally only useful for debugging purposes.

`reinitializes tao` reinitializes *Tao*. This can be useful to reset everything to initial conditions or to perform analysis with more than one initialization file. See the Command Line Initialization section (§10.1) for a list of the optional arguments. If an argument is not set, the `reinitialize` command uses the same argument value that were used in the last `reinitialize` command, or, if this is the first reinitialization, what was used to start *Tao*. Exception: If the `-clear` switch is present, all initialization parameters are set to their default state before the command line arguments specified in the `reinitialize` command are parsed. The `-clear` switch, if used, should come before any command line arguments since if there are command line arguments before the `-clear` switch, these arguments will be cleared.

Examples:

```
  reinit tao                         ! Reinit using previous arguments
  reinit tao -init special.init      ! Reinitializes Tao with the initialization file
                                     !   special.init.
  reinit -clear -start my_start      ! Use default init values except for the start file.
```

11.26 restore

The `restore` command cancels data or variable vetoes. Format:

```
  restore data  <data_name> <locations>
  restore var <var_name> <locations>
```

See also the `use` and `veto` commands.

Examples:

```
  restore data orbit.x[23,34:56]   ! un-veto orbit.x 23 and 34 through 56.
  restore data orbit.x[23,34:56:2] ! un-veto orbit.x 23 and even data between 34
                                   !                                          and 56
  restore data *@orbit[34]         ! un-veto orbit data in all universes.
  restore var quad_k1[67]          ! un-veto variable
```

11.27 run_optimizer

The `run_optimizer` command runs an optimizer. Format:

```
  run_optimizer {<optimizer>}
```

If `<optimizer>` is not given then the default optimizer is used. Use the `show optimizer` (§11.30.24) command to see optimizer parameters. To stop the optimizer before it is finished press the period "." key. If you want the optimizer to run forever run the optimizer in `single mode`. Valid optimizers are:

```
  custom        ! Used when a custom optimizer has been implemented (§14).
  de            ! Differential Evolution (good for global optimizations).
  geodesic_lm   ! ``Geodesic'' Levenburg-Marquardt (good for local optimizations).
  lm            ! Levenburg-Marquardt (good for local optimizations).
  lmdif         ! Levenburg-Marquardt (alternative version) (good for local optimizations).
  svd           ! svd optimizer (good for local optimizations).
```

See the optimization chapter (§8) for details on how *Tao* structures optimization and for more details on the different optimizers.

Examples:

```
  run         ! Run the default optimizer
  run de      ! Run the de optimizer
```

11.28 scale

The `scale` command scales the vertical axis of a graph or set of graphs. Format:

```
  scale {-exact} {-gang} {-include_wall} {-nogang}
             {-y} {-y2} {<where> {<value1> {<value2>}}}
```

Which graphs are scaled is determined by the `<where>` switch. If `<where>` is not present or `<where>` is `all` then all graphs are scaled. `<where>` can be a plot name or the name of an individual graph withing a plot.

`scale` adjusts the vertical scale of graphs. If neither `<value1>` nor `<value2>` is present then an `autoscale` is performed and the scale is adjusted so that all the data points are within the graph region. If an autoscale is performed upon an entire plot, and if `plot%autoscale_gang_y` (§10.13.2) is

True, then the chosen scales will be the same for all graphs. That is, a single scale is calculated so that all the data of all the graphs is within the plot region. The affect of `plot%autoscale_gang_y` can be overridden by using the `-gang` or `-nogang` switches.

If only `<value1>` is present then the scale is taken to be from `-<value1>` to +`<value1>`. If both are present than the scale is from `<value1>` to `<value2>`.

A graph can have a `y2` (left) axis scale that is separate from the `y` (right) axis. Normally, the `scale` command will scale both axes. Scaling of just one of these axes can be achieved by using the `-y` or `-y2` switches.

How a graph is scaled is determined in part by the setting of the `bounds` parameter in the `y` and `y2` components of the graph. See `s:quick.plot` for more details. The `-exact` switch, if present, will set `bounds` to `"EXACT"` which means that *Tao* will use the min and max bounds as given by `<value1>` and `<value2>` and not try to find "nice" values near the given ones. If `<value1>` and `<value2>` are not given, and if `bounds` is set to `"EXACT"`, *Tao* will set `bounds` to `"GENERAL"`. Note: To set the axis `bounds` directly, use the `set graph` command.

For scaling `floor_plan` plots where there is a building wall to be drawn, if `-include_wall` is present and autoscaling is being done, then the plot bounds are extended to include the extent of the building wall.

Examples:

```
scale top.x -3  7   ! Scale the x graph in the top region
scale -y2 top.x     ! Scale only the y2 axis of the top.x graph.
scale bottom        ! Autoscale the graphs of the plot in the bottom region
scale -include      ! Scale everything and include the extent of any
                    !   building walls in the calculation of the plot bounds.
```

11.29 set

The `set` command is used to set values for data, variables, etc. Subcommands are:

```
set beam {n@}<parameter> = <value>                     ! §11.29.1
set beam_init {n@}<parameter> = <value>                ! §11.29.2
set bmad_com <parameter> = <value>                     ! §11.29.3
set branch <branch> <parameter> = <value>              ! §11.29.4
set calculate <on/off>                                 ! §11.29.5
set curve <curve> <parameter> = <value>                ! §11.29.6
set data <data_name>|<parameter> = <value>             ! §11.29.7
set default <parameter> = <value>                      ! §11.29.8
set dynamic_aperture {n@}<parameter = <value>          ! §11.29.9
set element <element_list> <attribute> = <value>       ! §11.29.10
set floor_plan <parameter> = <value>                   ! §11.29.11
set geodesic_lm <parameter> = <value>                  ! §11.29.12
set global <parameter> = <value>                       ! §11.29.13
set graph <graph> <parameter> = <value>                ! §11.29.14
set key <key> = <command>                              ! §11.29.15
set lat_layout <parameter> = <value>                   ! §11.29.16
set lattice {n@}<destination_lat> = <source_lat>       ! §11.29.17
set opti_de_param <parameter> = <value>                ! §11.29.18
set particle_start {n@}<coordinate> = <value>          ! §11.29.19
set plot <plot> <parameter> = <value>                  ! §11.29.20
```

```
  set plot_page <parameter> = <value1> {<value2>}          ! §11.29.21
  set ptc_com <parameter> = <value>                        ! §11.29.22
  set ran_state = <random_number_generator_state>          ! §11.29.23
  set region <region> <parameter> = <value>                ! §11.29.24
  set space_charge_com <parameter> = <value>               ! §11.29.25
  set symbolic_number <name> = <value>                     ! §11.29.26
  set tune <Qa> <Qb>                                       ! §11.29.27
  set universe <what_universe> <on/off>                    ! §11.29.28
  set universe <what_universe> <calc_name> <on/off>        ! §11.29.28
  set variable <var_name>|<parameter> = <value>            ! §11.29.29
  set wave <parameter> = <value>                           ! §11.29.30
  set z_tune <Qz>                                          ! §11.29.31
```

When running *Tao*, to see documentation on any of the subcommands, use the `help set <subcommand>` command. For example, `help set element` will show information on the `set element` subcommand.

Also see the `change` command (§11.3). The `change` command is specialized for varying real parameters while the `set` command is more general.

Note: The `show` command (§11.30) is able to display the settings of many variables that can be set by the `set` command.

To apply a set to all data or variable classes use "*" in place of `<data_name>` or `var_name`.

To set the prompt color, use the command

```
  set global prompt_color = <value>
```

Where `<value>` may be one of:

```
  "BLACK"
  "RED"
  "GREEN"
  "YELLOW"
  "BLUE"
  "MAGENTA"
  "CYAN"
  "GRAY"
  "DEFAULT"       ! Default foreground color
```

11.29.1 set beam

Format:

```
  set beam {n@}<parameter> = <value>
  set beam {n@}beginning = <ele-name>
  set beam {n@}add_saved_at = <ele-list>
  set beam {n@}subtract_saved_at = <ele-list>
```

The `set beam` command sets beam parameters such as the initial and final tracking positions. Use the `show beam` command (§11.30) to see the current values.

For the `set beam beginning <ele-name>` command, the element specified by `<ele-name>` must be an element where particle positions of the tracked beam have been stored. With this command, the initial distribution of the beam at the beginning of the lattice will be set to the distribution at the indicated element. This is useful to track the beam over many turns.

The `set beam {n@}add_saved_at` command adds to the list of elements where the beam distribution is saved at.

The `set beam {n@}subtract_saved_at` command subtracts from the list of elements where the beam distribution is saved at.

The optional `n@` allows the specification of the universe or universes the set is applied to. The current default universe (§3.3) will be used if no universe is given.

Also see the commands: `set beam_init` and `set particle_start`.

Examples:

```
set beam 2@track_start = q10w  ! Set the tracking start at element Q10W in universe 2.
set beam saved_at = "Q*, B*"   ! Save beam parameters (sigma matrix, etc.) at elements
                               !  whose names begin with "Q" or "B".
set beam add_saved_at = S10    ! Save beam parameters at element "S10" as well.
set beam beginning = end       ! Set the initial beam distribution equal to the distribution at
                               !  the lattice element named "end".
```

11.29.2 set beam_init

Format:

```
set beam_init {n@}<parameter> = <value>
```

The `set beam_init` command sets parameters of the `beam_init` structure (§10.7). Additionally, the `set beam_init` command can set the parameters (§10.7)

```
track_start  and
track_end
```

The optional `n@` allows the specification of the universe or universes the set is applied to. The current default universe (§3.3) will be used if no universe is given.

Use the `show beam` command (§11.30) to see the current values.

Also see the commands: `set beam` and `set particle_start`.

Examples:

```
set beam_init 3@center(2) = 0.004   ! Set px center of beam for universe 3.
set beam_init [1,2]@sig_pz = 0.02   ! Set sig_pz for universes 1 and 2.
set beam_init track_end = q10w      ! Set track_end parameter.
```

11.29.3 set bmad_com

Format:

```
set bmad_com <parameter> = <value>
```

Sets global *Bmad* parameters. Use the `show global -bmad_com` command to see a list of `<parameter>`s. See the *Bmad* manual for information on this structure.

Example:

```
set bmad_com radiation_fluctuations_on = T ! Turn on synchrotron radiation fluctuations.
```

11.29.4 set branch

Format:

```
  set branch <branch-id> <parameter> = <value>
```
Sets parameters associated with a lattice branch. The parameters that can be set are:
```
  particle                   = <species>   ! Reference particle
  default_tracking_species   = <species>   ! Particle that is tracked.
  geometry                   = open or closed
  live_branch                = T or F
```
Use the `show branch` command to see lattice branch information. `<branch-id>` may be the branch index or branch name. `<branch-id>` may also contain an optional `n@` prefix to specify a particular universe to apply the set to. The default is to only set the current viewed universe.

Note: When toggling a branch from closed to open the beginning orbit and Twiss parameters will not change. On the other hand, when toggling a branch from open to closed, the orbit and Twiss parameters will, in general, shift.

Examples:
```
  set branch 2@0 live_branch = F     ! Suppress calculations for branch # 0 of universe 2.
  set branch a_line geometry = open  ! Open geometry for branch named a_line.
  set branch default_tracking_species = positron
                                     ! Set the tracking species to positron.
```

11.29.5 set calculate

Format:
```
  set calculate {<on/off>}
```
Toggles the following on (True) or off (False) the parameter `global%lattice_calc_on`

Examples:
```
  set calc on    ! Sets lattice_calc_on to True
  set calc off   ! Sets lattice_calc_on to False
  set calc       ! Toggles lattice_calc_on
```

11.29.6 set curve

Format:
```
  set curve <curve> <parameter> = <value>
```
For `set curve`, the `<parameter>`s that can be set are:
```
  ele_ref_name        = <string>  ! Name or index of the reference element. Blank => No ref ele.
  ix_ele_ref          = <integer> ! Same as setting ele_ref_name. -1 => No ref ele.
  component           = <string>  ! §7.6.5
  ix_branch           = <integer> ! Branch index.
  ix_bunch            = <integer> ! Bunch index.
  ix_universe         = <integer> ! Universe index.
  symbol_every        = <integer> ! Symbol skip number.
  y_axis_scale_factor = <integer> ! Scaling of y axis
  draw_line           = <logical>
  draw_symbols        = <logical>
  draw_symbol_index   = <logical>
```
See the `Plot Templates` section (§10.13.2) for a description of these attributes. Use the `show curve` (§11.30) to see the settings of the attributes.

If there are visible plots with the same name as the plot parameter of `<curve>`, a template plot of the same name is ignored. To set template plot curve(s) in this case, add a “`T::`” prefix.

Examples:

```
  set curve top.x.c1 ix_universe = 2       ! Set universe number for curve.
  set curve T::orbit.x.c1 ix_universe = 2  ! Set curve in template plot.
```

11.29.7 set data

Format:

```
  set data {-silent} <data_name>|<component> = <value>
```

Set datum parameters.

parameters that are computed like the `model` value cannot be set. The list of parameter that `cannot` be set is:

```
  model, base, design, old
  good_model, good_base, good_design
  merit, delta_merit
  invalid, exists
  useit_opt, useit_plot
  ix_d1
```

The `-silent` switch, if present, prevents *Tao* from issuing an error message if *Tao* detects a malformed datum. This is useful when creating datums from scratch (via `pipe data_d2_create`) or when modifying multiple datum parameters, like a datum's `data_type` and `data_source`, where it is known that the datum will be in a malformed state before the final set.

Examples:

```
  set data *|ref = *|meas               ! Set ref data = measured in current universe.
  set data 2@orbit.x|ref = 2@orbit.x|model
                                        ! Set the ref orbit.x in universe 2 to model.
  set data beta.x[10]|weight = 1e-5     ! Set weight of datum.
  set -silent d[1]|data_type = beta.a   ! Set type_type.
```

11.29.8 set default

Format:

```
  set default <parameter> = <value>
```

The parameters that can be set are:

```
  branch              ! See: Lattices section (§3.4)
  universe            ! See: Universe section (§3.3)
```

Use the `show global` (§11.30) command to see the current default values.

Example:

```
  set default universe = 3
```

11.29.9 set dynamic_aperture

Format:

```
  set dynamic_aperture {n@}<parameter> = <value>
```

The `set dynamic_aperture` command sets parameters for dynamic aperture simulations (§10.12) Also see the `set universe dynamic_aperture` (§11.29.28) and `show dynamic_aperture` (§11.30.11).

To set the particle energy for the <n>th scan use `pz(<n>)`. Use a value less than -1 to remove the scan.

The optional `n@` prefix allows the specification of the universe or universes the set is applied to. The current default universe (§3.3) will be used if no universe is given.

Examples:

```
  set dy 2@n_angle = 20   ! Set number of scan points for universe 2.
  set dy accuracy = 1e-5  ! Set scan scan accuracy
  set dy pz(3) = -0.05    ! Set particle energy for the 3rd scan.
```

11.29.10 set element

Format:

```
  set {-update} element <element_list> <attribute> = <value>
```

The `set element` command sets the attributes of an element. Use the `show element` command to view the attributes of an element.

The reference species of a lattice branch can be set by setting `ref_species` of the `beginning` element of the branch.

The `-update` switch, if present, suppresses *Tao* from printing error messages if a "variable slave value mismatch" is detected (§5.4). Independent of whether `-update` is present or not, *Tao* will fix the mismatch using the changed value to set all of the slave values.

Note: `set element` can be used to set `ramper` type elements.

Note: If an element in the `<element_list>` does not specify a universe (or universes), only the element in the viewed universe is used. See the examples below.

Note: It is also possible to use the `change element` command to change real (as opposed to logical or integer) attributes.

Examples:

```
  set ele rfcav::* is_on = F         ! Turn off all rfcavity elements the viewed universe.
  set ele *@rfcav::* is_on = F       ! Turn off all rfcavity elements in all universes.
  set ele A:B track_method = linear  ! Set tracking_method for all elements between
                                     !   elements A and B
  set ele q10w k1 = 0.7              ! Set element q10w k1 of the viewed universe.
  set ele Q* k1 = ele::Q*[k1]|design ! Set model to design values.
```

11.29.11 set floor_plan

Format:

```
  set floor_plan <parameter> = <value>
```

Sets parameters for `floor_plan` plots (§10.13.9). Possible `<parameters>` are:

```
  <shape_name>%<shape_parameter>
  draw_beam_chamber_wall
  beam_chamber_wall_scale
```

Where `<ele_shape_name>` is of the form "`ele_shape(<n>)`" where `<n>` is the index of the `ele_shape` in the `floor_plan_drawing` namelist. Use "`show plot -floor_plan`" to see the current state of the `floor_plan` parameters

Example:

```
  set floor_plan ele_shape(2)%draw = F  ! Veto drawing of ele_shape(2)
  set floor_plan beam_chamber_scale = 0.5
```

11.29.12 set geodesic_lm

Format:

```
set geodesic_lm <parameter> = <value>
```

For `set geodesic_lm`: The `show optimizer geodesic_lm` command will give a list of `<parameter>`s.

Example:

```
set geodesic_lm method = 10
```

11.29.13 set global

Format:

```
set global <parameter> = <value>
```

The `set global` command sets global parameters of *Tao*. The `show global` command will give a list of global parameters.

Example:

```
set global n_opti_loops = 30  ! Set number of optimization cycles
set global rf_on = T          ! Turn on the RF cavities.
```

For users who have OpenMP support enabled for parallel calculations, the global parameter `n_threads` may be set at runtime to configure the `OMP_NUM_THREADS` on the fly.

Example:

```
set global n_threads = 1  ! Use only a single thread
set global n_threads = 4  ! Use four threads
```

11.29.14 set graph

Format:

```
set graph <graph> <parameter> = <value>
```

The `set graph` command is used to set parameters of a graph structure (§10.13.2).

If the `<graph>` name corresponds to a plot, the set is applied to all the graphs associated with the plot. If there are visible plots with the same name as the plot parameter of `<graph>`, a template plot of the same name is ignored. To set template plot graphs(s) in this case, add a "`T::`" prefix.

For setting the `parameter` attribute see also the commands:

```
set plot parameter      ! §11.29.20
set curve parameter     ! §11.29.6
```

Example:

```
set graph orbit.x component = model - design  ! Plot orbit (model - design).
set graph orbit component = model - design    ! Applies to all graphs of orbit plot.
set graph T::orbit.x component = design       ! Set template plot
set graph r11 floor_plan%orbit_scale = 100    ! To display an orbit.
set graph beta y%bounds = "zero_at_end"       ! §7.7.
```

11.29.15 set key

Format:

```
  set key <key> = <command>
```

Binds a custom command to a key for use in single mode (§12). This will override the default behavior (if there is one) of the key. The command `default` will reset the key to its default usage.

Example:

```
  set key h = veto var *
  set key j = default
```

11.29.16 set lat_layout

Format:

```
  set lat_layout <parameter> = <value>
```

Sets parameters for `lat_layout` plots (§10.13.9). Syntax for "`set lat_layout`" is identical to syntax of "`set floor_plan`". See "`set floor_plan`" for more details.

Use "`show plot -lat_layout`" to see a listing of all shapes.

Example:

```
  set lat_layout ele_shape(2)%draw = F  ! Veto drawing of shape #2
```

11.29.17 set lattice

Format:

```
  set lattice {n@}<destination_lat> = <source_lat>
```

The `set lattice` command transfers lattice parameters (element strengths, etc., etc.) from one lattice (the `source` lattice) to another (the `destination` lattice). Both lattices are restricted to be from the same universe. The optional `n@` prefix (§3.3) of the destination lattice can be used to specify which universe the lattices are in. If multiple universes are specified, the corresponding destination lattice will be set to the corresponding source lattice in each universe. Note: At this time, it is not permitted to transfer parameters between lattices in different universes.

The destination lattices that can be set are:

```
  model      ! Model lattice.
  base       ! Base lattice
```

The source lattice can be:

```
  model       ! model lattice.
  base        ! base lattice.
  design      ! design lattice
```

Note: *Tao* variables that control parameters in multiple universes can complicate things. If, for example, there are two universes, and a *Tao* variable controls, say, the quadrupole strength of quadrupoles in both universes, then a "set lat 2@model = design" will result in the quadrupole strengths of those quadrupoles controlled by the variable in universe 1 being changed.

Example:

```
  set lattice *@model = design  ! Set the model lattice to the design in
                                !   all universes.
  set lattice base = model      ! Set the base lattice to the model lattice in
                                !   the default universe.
```

11.29.18 set opti_de_param

Format:

```
set opti_de_param <parameter> = <value>
```

For `set opti_de_param`: The `show global` command will give a list of `<parameter>`s.

Example:

```
set opti_de_param binomial_cross = T  ! Use binomial crossovers
```

11.29.19 set particle_start

Format:

```
set particle_start {n@}<coordinate> = <value>
```

The `set particle_start` command sets the starting coordinates for single particle tracking for lattices with an open geometry. If the `use_particle_start` of the `beam_init` structure (§10.7) is set to True, `particle_start` will also vary the beam centroid and beam particle spin for beam tracking. Note: Setting `particle_start` will affect the orbit at the active `fixer` element and the default active fixer element is the `beginning` element.

The optional `n@` universe specification (§3.3) may be used to specify the universe or universes to apply the set command to.

`<coordinate>` is one of:

```
x, px, y, py, z, pz, t
spin_x, spin_y, spin_z
```

For photons, `<coordinate>` may also be:

```
field_x, field_y, phase_x, phase_y
e_photon
```

The `*` coordinate denotes the phase space vector (x, p_x, y, p_y, z, p_z). For closed lattices only the `pz` parameter is applicable. For lattices that have an `e_gun` (which necessarily implies that the lattice has an open geometry), the time `t` coordinate must be varied instead of `pz`.

For photons, the photon energy can be set by setting `e_photon` which sets the photon energy in eV or by setting `pz` which sets the relative difference between the photon energy and the reference energy:

```
photon_energy = reference_energy * (1 + pz)
```

To see the values for `particle_start` use the command `show element 0`.

Also see the commands: `set beam` (§11.29.1), `set beam_init` (§11.29.2), and `change particle_start` (§11.3).

Examples:

```
set particle_start 2@x = 0.001        ! Set beginning x position in universe 2 to 1 mm.
set particle_start field_x = 1        ! Set photon field
set particle_start spin_y = 0.37      ! Set spin parameter.
```

11.29.20 set plot

The `set plot` command set various parameters of a plot. Format:

```
set plot <plot_or_region> <parameter> = <value>
```

The `<parameters>`s that can be set are:

```
  autoscale_x        = <logical>
  autoscale_y        = <logical>
  visible            = <logical>
  component          = <string>    ! Sets curve component §7.6.5
  x%<axis_parameter> = <value>
  n_curve_pts        = <integer>
```

Use the `show plot <plot_name>` to see the settings of various parameters. See the `Plot Templates` section (§10.13.2) for information on the plotting parameters.

The `visible` parameter hides a plot but keeps the plot associated with the associate region. If the plot window is not enabled (`-noplot` option used at startup), the `visible` parameter is used by *Tao* to decide whether to calculate the points needed for plotting curves (saves time if the computation is not needed). This is relevant when *Tao* is interfaced to a `GUI` (§13.3).

The `n_curve_pts` parameters sets the number of points to use for drawing "smooth" curves. This overrides the setting of `plot_page%n_plot_pts` (§10.13). Warning: *Tao* will cache intermediate calculations used to compute a smooth curve to use in the computation of other smooth curves. *Tao* will only do this for curves that have `plot_page%n_curve_pts` number of points. Depending upon the circumstances, setting `plot%n_curve_pts` for individual plots may slow down plotting calculations significantly.

Note: If the `component` parameter is set, the `<value>` is stored in each of the curves of the plot since the `component` attribute is associated with individual curves and not the plot as a whole.

If `<plot_or_region>` is a plot name, and there are visible plots of that name, any template plot of the same name is ignored. To set a template plot in this case, add a "`T::`" prefix.

Example:

```
  set plot orbit visible = F           ! Hide orbit plot
  set plot beta component = design     ! Plot the design value.
  set plot T::beta component = design  ! Set the template plot instead of any displayed plots.
  set plot x%draw_label = False        ! Do not draw x-axis label.
```

11.29.21 set plot_page

Format:

```
  set plot_page <parameter> = <value1> {<value2>}
```

The `set plot_page` command sets `plot page` parameters (§7.1). use the `show plot -page` command to see a list of plot page parameters.

The `<value2>` value is needed for the plot window `size`.

Examples:

```
  set plot_page title = 'XYZ'  ! Set plot page title string
  set plot_page size = 500 600 ! Set plot window size in pixels.
```

11.29.22 set ptc_com

Format:

```
  set ptc_com <parameter> = <value>
```

Sets global PTC parameters. Use the `show global -ptc_com` command to see a list of `<parameter>`s. See the *Bmad* manual for information on this structure.

Note: to set the Taylor map order, use the command:

```
set bmad_com taylor_order = ...
```

Example:

```
set ptc_com exact_model = F ! Non-exact is not as accurate but faster.
```

11.29.23 set ran_state

Format:

```
set ran_state = <random_number_generator_state>
```

Sets the state of the random number generator to a specific state. Use `show global -ran_state` to show the random number generator state. Manipulating the state for generating random numbers is generally only used for debugging purposes and is not of interest to the typical user.

11.29.24 set region

Format:

```
set region <parameter> = <value>
```

Sets a plot region parameter. Parameters are:

```
x1, x2, y1, y2    ! Region rectangle placement
visible           ! Is plot in region visible?
```

Use the `show plot` command to see a listing of region parameters.

Example:

```
set region r13 y2 = 0.3  ! Set y2 parameter of region r13
```

11.29.25 set space_charge_com

Format:

```
set space_charge_com <parameter> = <value>
```

Sets global space charge (including CSR) parameters. Use the `show global -space_charge_com` command to see a list of `<parameter>`s. See the *Bmad* manual for information on this structure.

Example:

```
set space_charge_com n_bin = 30  ! Set number of bins used in the csr calc.
```

11.29.26 set symbolic_number

Format:

```
set symbolic_number <name> = <value>
```

Create a symbolic number that can be used in expressions. Use the `show symbolic_number` command to show a list of symbols that have been defined. Repeated `set` commands may be used to modify the value of a symbol if desired.

Example:

```
set sym aa = 23.4 * pi  ! Define the symbol "aa"
```

11.29.27 set tune

Format:

```
  set tune {-branch <branch_list>} {-listing} {-mask <veto_list>} {<Qa>} {<Qb>}
```

Set the two transverse tunes. Units for `<Qa>` and `<Qb>` are radians/2pi. If only the fractional part of `<Qa>` and `<Qb>` is given, the integer part will be taken to be the integer part of the tunes in the model lattice. If not given, `<Qa>` and `<Qb>` will default to the model lattice tunes.

The `<branch_list>` is a list of lattice branches with optional universe prefix.

The algorithm used to vary the tunes starts by selecting all quadrupole elements or overlay elements whose slave parameters are all quadrupole `k1`. From this list all quadrupoles that have a non-zero tilt are thrown out. The list is divided up into two groups: One group where $\beta_a > \beta_b$ at the quadrupole and the other group is where $\beta_a > \beta_b$. The elements in each group are assigned a weight. Currently the weights assigned is $+1$ for all the elements in one group and -1 for all elements in the other group. To get the desired tune, the `k1` strengths of the elements of each group are are varied such that the fractional change of `k1` for all quadrupoles in a group is proportional to the weights.

The `-listing` option, if present, will, in addition to the tune change, generate a list of quadrupoles varied along with variation coefficients.

It is sometimes desirable to veto from changing certain quadrupoles (or overlays). The `<veto_list>` of the `-mask` option gives a list of quadrupoles *not* to use. A tilde   in front of an element name means that the element will not be vetoed. This can be used to specify what quadrupoles to use (not veto). For example:

```
  set tune -mask *,~qf_%%,~qd_% 0.23 0.45
```

In this example, the mask string has three "words" separated by commas. The first word is "`*`" which will veto everything. The second word `qf_%` reinstates all elements whose name starts with `qf_` and has exactly two characters after the beginning `qf_`. The third word reinstates all elements that match the wild card pattern `qd_%`. The upshot is that only elements whose names match `qf_%%` or `qd_%` will be varied.

The tunes can also be varied using the `change tune` command. For the longitudinal tune there is the `set z_tune` and `change z_tune` commands. Note that with the present algorithms used for varying the transverse and longitudinal tunes, varying the transverse tunes will vary the longitudinal tune somewhat and vice versa.

Examples:

```
  set tune -mask qd* 0.45 0.67  ! Use all quads except with name starting with "qd".
```

11.29.28 set universe

Format:

```
  set universe <what_universe> <on/off>
  set universe <what_universe> recalculate
  set universe <what_universe> twiss_calc <on/off>
  set universe <what_universe> dynamic_aperture_calc <on/off>
  set universe <what_universe> one_turn_map_calc <on/off>
  set universe <what_universe> track_calc <on/off>
```

The `set universe <what_universe> ...` command will turn on or off specified lattice/tracking calculations for the specified universe(s). Turning specified calculations off for a universe is useful to speed up lattice calculations when the calculation is not necessary.

To specify the currently default universe (§3.3), you can use `-1` as the `<what_universe>` index. To specify all universes, use `*`. Use the `show universe` command to see the state of these switches are. The `<what_universe>` argument may be a list of universes enclosed in brackets "[...]". See below for an example.

Note: The global logical `lattice_calc_on` (§10.6) is separate from the logicals set by `set universe`. That is, toggling the state of `lattice_calc_on` will not affect the settings of the logicals set by `set universe`. If `lattice_calc_on` is set to `False` then no calculations are done in any universe independent of the settings of the `set universe` logicals. That is, `lattice_calc_on` acts as a master toggle that can be used to turn off all lattice/tracking calculations.

If optimizing while one or more universes are turned off, the variables associated with that universe will still be included in the merit function but not the data for that universe. The variables will still vary in the turned off universe.

The `set universe <what_universe> recalculate` command will recalculate the lattice parameters for that universe.

The `set universe <what_universe> dynamic_aperture_calc` command will enable the dynamic aperture calculation for a ring. See the `Initializing Dynamic Aperture` section (§10.12) for more details. To enable the dynamic aperture calculation at startup, set the `design_lattice(i)%dynamic_aperture` parameter (§10.4).

The `set universe <what_universe> one_turn_map_calc` command will enable a one-turn-map calculation for a ring using PTC, and populate the normal form taylor maps. See Eq. 6.14 and Eq. 6.15 in the `normal.` data type. To enable the map calculation at startup, set the `design_lattice(i)%one_turn_map_calc` parameter (§10.4).

The commands

```
  set universe <what_universe> twiss_calc  and
  set universe <what_universe> track_calc
```

will set whether the 6x6 transfer matrices and the central orbit (closed orbit for circular rings) is calculated for a given universe. Turning this off is useful in speeding up calculations in the case where the transfer matrices and/or orbit is not being used.

Example:

```
  set universe [1,3] off ! Set universes 1 and 3 off.
  set universe -1 on     ! Set on currently default universe.
  set universe * recalc  ! Recalculate in all universes.
```

11.29.29 set variable

Format:

```
  set variable <var_name>|<parameter> = <value>
```

For `set var`, the `<parameter>`s that can be set are:

```
  model       ! Model lattice value.
  base        ! Base model value
  design      ! Design model value
  meas        ! Value at the time of a measurement.
  ref         ! Value at the time of a reference measurement.
  weight      ! Weight for the merit function.
  exists      ! Does this variable actually correspond to something?
  good_var    ! The optimizer can be allowed to vary it
  good_opt    ! Good for using in the merit function for optimization?
  good_plot   ! Good for using in a plot?
  good_user   ! This is what is set by the use, veto, and restore commands.
  step        ! Sets what a "small" variation of the variable is.
  merit_type  ! How merit contribution is calculated.
  key_bound   ! Model value can be modified using keyboard?
  key_delta   ! Change in model value when key is pressed.
```

Example:

```
  set var quad_k1|weight = 0.1         ! Set quad_k1 weights.
```

11.29.30 set wave

Format:

```
  set wave <parameter> = <value>
```

The `set wave` command sets the boundaries of the A and B regions for the wave analysis (§9). The parameters are

```
  ix_a = <ix_a1> <ix_a2>  ! A-region left and right boundaries.
  ix_b = <ix_b1> <ix_b2>  ! B-region left and right boundaries.
```

Example:

```
  set wave ix_a = 15 27    ! Set A-region to span from datum #15 to #27
```

Note: Use the `wave` command (§11.38) first to setup the display of the wave analysis.

11.29.31 set z_tune

Format:

```
  set z_tune <Qz>
```

Set the longitudinal tune by varying RF cavity voltages.

Also see the `change z_tune` command as well as the `set tune` and `change_tune` commands. Note that with the present algorithms used for varying the transverse and longitudinal tunes, varying the transverse tunes will vary the longitudinal tune somewhat and vice versa.

Example:

```
  set z_tune 0.023
```

11.30 show

The `show` command is used to display information. Format:

```
  show {-append <file_name>} {-noprint} {-no_err_out} <subcommand>
  show {-write <file_name>} {-noprint} {-no_err_out} <subcommand>
```

`<subcommand>` subcommands may be one of:

```
  show alias                ! Show aliases §11.30.1.
  show beam ...             ! Show beam info §11.30.2.
  show bmad_com             ! Old syntax for show global -bmad_com §11.30.15.
  show branch ...           ! Show lattice branch info §11.30.3.
  show building_wall        ! Show building wall info §11.30.4.
  show chromaticity ...     ! Show chromaticity, momentum compaction, phase slip §11.30.5.
```

```
show constraints              ! Show optimization constraints §11.30.6.
show control ...              ! Show lords and slaves of a given element §11.30.7.
show csr_param                ! Old syntax for show global -csr_param §11.30.15.
show curve ...                ! Show plot curve info §11.30.8.
show data ...                 ! Show optimization data info §11.30.9.
show derivative ...           ! Show d_data/d_var optimization info §11.30.10.
show dynamic_aperture         ! Show DA info §11.30.11.
show element ...              ! Show lattice element info §11.30.12.
show emittance                ! Show normal mode emittances §11.30.13.
show floor_plan               ! Old syntax for show plot -floor_plan §11.30.26.
show field ...                ! Show EM field §11.30.14.
show global ...               ! Show Tao global parameters §11.30.15.
show graph ...                ! Show plot graph info §11.30.16.
show history ...              ! Show command history §11.30.17.
show hom                      ! Show Higher Order Mode info §11.30.18.
show internal ...             ! Used for code debugging §11.30.19.
show key_bindings             ! Show single mode key bindings §11.30.20.
show lattice ...              ! Lattice element-by-element table §11.30.21.
show matrix ...               ! Show transport matrix §11.30.22.
show merit ...                ! Show optimization merit function §11.30.23.
show optimizer ...            ! Show optimizer info §11.30.24.
show particle ...             ! Show tracked particle beam info §11.30.25.
show plot ...                 ! Show plot info §11.30.26.
show plot_page                ! Old syntax for show plot -page §11.30.26.
show ptc ...                  ! Show PTC calculated parameters §11.30.27.
show ptc_com                  ! Old syntax for show global -ptc_com. §11.30.15.
show radiation_integrals ... ! Show synchrotron radiation integrals §11.30.28.
show rampers                  ! Show ramper lord and slave information §11.30.29.
show space_charge_com         ! Old syntax for show global -space_charge_com §11.30.15.
show spin ...                 ! Show information on spin simulations §11.30.30.
show string ...               ! Print a string §11.30.31.
show symbolic_numbers ...     ! Show symbolic constants §11.30.32.
show taylor_map ...           ! Show transport Taylor map§11.30.33.
show track ...                ! Show phase space coords, Twiss, EM field,
                              !   and other info along the tracked orbit §11.30.34.
show twiss_and_orbit ...      ! Show Twiss and orbit info at given position including
                              !   synchrotron radiation related parameters §11.30.35.
show universe ...             ! Show universe info §11.30.36.
show use                      ! Show data and vars used in optimization §11.30.37.
show value ...                ! Show value of an expression §11.30.38.
show variables ...            ! Show optimization variable info §11.30.39.
show version                  ! Show Tao version.
show wake_elements            ! Show wake info §11.30.41.
show wall ...                 ! Show vacuum chamber wall info §11.30.42.
show wave                     ! Show wave analysis info §11.30.43.
```

When running *Tao*, to see documentation on any of the subcommands, use the `help show <subcommand>` command. For example, `help show element` will show information on the `show element` subcommand.

The `show` command has `-append` and `-write` optional arguments which can be used to write the results to a file. The `show -append` command will appended to the output file. The `show -write` command will first erase the contents of the output file. If `global%write_file` has a `*` character in it, a three digit number is substituted for the *. The value of the number starts at 001 and increases by 1 each time `show -write` is used. Example:

```
  show -write floor.dat lat -floor  ! Write floor positions to the file "floor.dat".
```

The `-noprint` option suppresses printing and is useful when writing large amounts of data to a file.

When writing to a file, if there are any error messages (for example, that something could not be computed), the error messages are reproduced in the file. If this behavior is not wanted, the `-no_err_out` switch may be used to block the error messages being written.

The `-append`, `-write`, `-noprint`, and `-no_err_out` switches must be placed before `<subcommand>`.

Note: When running *Tao* as a subprocess, use the `pipe` command (§13.2) instead of the `show` command for communicating with the parent process.

11.30.1 show alias

Syntax:

```
  show alias
```

Shows a list of defined aliases. See the `alias` command for more details.

11.30.2 show beam

Command to show beam parameters. Syntax:

```
  show beam {-comb} {-universe <uni_index>} {-lattice} {-z <z1> <z2>} {<element_id>}
```

If both `<element_id>` and `-lattice` are absent, `show beam` shows parameters used with beam tracking including the number of particles in a bunch, etc.

If no universe is given, the current default universe (§3.3) is used.

If `<element_id>` is present, and `-lattice` is not, `show beam` will show beam parameters at the selected element.

If `-lattice` is present, `show beam` will show the beam sigma matrix as calculated from the lattice at the position given by `<element_id>` (§10.8). If an element is not specified, the beginning element (with index 0) will be used.

If the `-comb` is present, `<element_id>` should be an integer which is the index of the comb of longitudinally equally spaced points where the beam parameters are evaluated at. Note: `comb_ds_save` (§10.6.1) is used to set the spacing between points. If `<element_id>` is not present, information about all the comb points is output. Also see the `write bunch_comb` (§11.39.4) and `pipe bunch_comb` (§13.4.5) commands.

The `-z <z1> <z2>` option shows bunch parameters for a slice of the beam. The slice boundaries are specified by `<z1>` and `<z2>`. Both values must be in the range [0.0, 1.0] with 0.0 indicating the back of the bunch and 1.0 indicating the front of the bunch. For example, "-z 0.0, 0.5" would give bunch parameters for the back half of the bunch. If the `-z` option is used, the `<element_id>` must be specified and the beam must have been saved at the corresponding element (§10.7).

Note: To show individual particle positions, see the `show particle` command (§11.30.25).

Note: Use the `set beam_init` command to set values of the `beam_init` structure.

Examples:

```
  show beam          ! Show beam initialization parameters.
  show beam -lat 37  ! Show sigma matrix, etc. calculated at element #37.
  show beam -comb 3  ! Show sigma matrix, etc. calculated at comb index #3.
  show beam -z 0.1 0.4 q3w  ! Show parameters of a slice of the bunch.
```

11.30.3 show branch

Syntax:
```
  show branch {-universe <uni_index>}
```
Lists the lattice branches of the lattice associated with the given universe along with information on the fork elements connecting the branches. If no universe is given, the current default universe (§3.3) is used.

Example:
```
  show branch -u 2     ! Show info on lattice branches associated with universe 2
```

11.30.4 show building_wall

Syntax:
```
  show building_wall
```
List all building wall (§10.11) sections along with the points that define the sections.

For vacuum_chamber, capillary, and diffraction_plate walls use the "show wall" command.

11.30.5 show chromaticity

Syntax:
```
  show chromaticity {-taylor} {-universe <uni_index>}
```
Shows chromaticity and derivatives as calculated from PTC normal form analysis. Also shown is momentum compaction and phase slip and derivatives.

If no universe is given, the current default universe (§3.3) is used.

The `-taylor` switch will show the Taylor series for the three normal mode tunes as functions of the phase space coordinates. The computation uses complex series. The imaginary part should be zero (or very small).

11.30.6 show constraints

Syntax:
```
  show constraints
```
Lists data and variable constraints. Also see `show merit`.

11.30.7 show control

Syntax:
```
  show control element-name-or-index
```
This command compiles a list of all lords (and lords of lords, etc.) of the given element as well as a list of all slaves (and slaves of slaves, etc.) of the given element. Then for each element in the lists, the lords and slaves of that element are displayed. Example:
```
  show control q1#2   ! Show lords/slaves of second instance of element named q1.
```

11.30.8 show curve

Syntax:

```
  show curve {-line} {-no_header} {-symbol} <curve_name>
```

Show information on a particular curve of a particular plot. See §7 for the syntax on plot, graph, and curve names. Use `show plot` to get a list of plot names. The `-symbol` switch will additionally print the (x,y) points for the symbol placement and the `-line` switch will print the (x,y) points used to draw the "smooth" curve in between the symbols. The line or symbol points from multiple curves can be printed by specifying multiple curves. Example:

```
  show curve -sym orbit
```

This will produce a three column table assuming that the orbit plot has curves `orbit.x.c1` and `orbit.y.c1`. When specifying multiple curves, each curve must have the same number of data points and it will be assumed that the horizontal data values are the same for all curves so the horizontal data values will be put in column 1.

The `-no_header` switch is used with `-line` and `-symbol` to suppress the printing of header lines. This is useful when the generated table is to be read in by another program.

If there are visible plots with the same name as the plot parameter of `<curve>`, a template plot of the same name is ignored. To show template plot curve(s) in this case, add a "`T::`" prefix.

Also see: `show plot` and `show graph` commands.

Example:

```
  show curve r2.g1.c3     ! Show the attributes of a curve named "c3" which is
                          !   in the graph "g1" which is plotted in region "r2".
```

11.30.9 show data

Syntax:

```
  show data {<data_name>}
```

Shows data information. If `<data_name>` is not present then a list of all `d2_data` names is printed.

Examples:

```
  show data                   ! Lists d2_data for all universes
  show data *@*               ! Same as above
  show data -1@*              ! Lists d2_data for the currently default universe.
  show data *                 ! Same as above.
  show data 2@*               ! Shows d2_data in universe 2.
  show data orbit             ! Show orbit data.
  show data orbit.x           ! list all orbit.x data elements.
  show data orbit.x[35]       ! Show details for orbit.x element 35
  show data orbit.x[35,86:95] ! list orbit.x elements 35 and 86 through 95
  show data orbit.x[1:99:5]   ! list every fifth orbit.x between 1 and 99
```

11.30.10 show derivative

Syntax:

```
  show derivative {-derivative_recalc} {<data_name(s)>} {<var_name(s)>}
```

Shows the derivative dData_Model_Value/dVariable. This derivative is used by the optimizers `lm` and `svd`. Note: Wild card characters can be used to show multiple derivatives. Default values for `<data_name(s)>` and `<var_name(s)>` is `"*"` (all data or variables).

The `-derivative_recalc` forces a recalculation of the derivative matrix. This is exactly the same as using `derivative` command (§11.12) before the `show derivative` command.

Note: Derivatives are only calculated for data and variables that are used in an optimization. That is, derivatives are only calculated for data and variables whose `useit_opt` parameter (see §6.2 and §5) is True.

The output of this command is a number of lines that look like:

```
Data                Variable               Derivative   ix_dat  ix_var
k.22a[98]           v_steer[92]           -7.63151E+01    1584     214
k.22a[98]           v_steer[93]           -1.81810E+00    1584     215
```

The first and second columns are the datum and variable names, the third column is the derivative, and the last two columns are the indexes of where the derivative is stored in *Tao*'s internal derivative matrix. These last two columns are for debugging purposes and can be ignored.

Example:

```
show deriv orbit.x[2] k1[3] ! Show dModel_Value/dVariable Derivative.
show deriv                  ! Show all derivatives. Warning! The output may be large.
```

11.30.11 show dynamic_aperture

Syntax:

```
show dynamic_aperture
```

Shows parameters and results of the dynamic aperture calculation (§10.12). See also the commands `set dynamic_aperture`, and `set universe dynamic_aperture`.

11.30.12 show element

Syntax:

```
show element {-attributes} {-base} {-data} {-design} {-all} {-field}
    {-floor_coords} {-no_slaves} {-no_super_slaves} {-ptc} {-taylor} {-wall}
    {-xfer_mat} <ele_name>
```

This shows information on lattice elements. The syntax for `<ele_name>` is explained in section §4.3. If `<ele_name>` contains a wild card or a class name then a list of elements that match the name are shown. If no wild–card or class name is present then information about the element whose name matches `<ele_name>` is shown.

If the `-ptc` switch is used, then the associated PTC fibre information will be displayed. If there is not associated PTC fibre (which will be true if PTC has not been used for tracking with this element), an associated PTC fibre will be created. In this case, only the PTC information will be displayed and the other switches will be ignored.

If the `-attributes` switch is present, then all of the element "attributes" will be displayed. The default is is to display only those attributes with non-zero values. "Attributes" here does not include such things as the cross-section, Taylor map and wiggler element parameters.

By default, the appropriate element(s) within the `model` lattice (§3.3) are used. This can be overridden by using the `-base` or the `-design` switches which switch the lattice to the `base` or `design` lattices respectively.

If the `-wall` switch is present, the wall information for the element, if it has been defined in the lattice file, is displayed. For an x-ray `capillary` element, the wall is the inner surface of the capillary. For all other elements, the wall is the beam chamber wall.

If the `-data` switch is present, information about the all the datums associated with the element will be listed.

If the `-floor_coords` switch is present, the global floor coordinates at the exit end of the element will be printed. See the *Bmad* manual for an explanation of the floor coordinates.

When using wild cards in the element name, if the `-no_super_slaves` switch is present then `super_slave` elements will not be included in the output. If the `-no_slaves` switch is present, both `super_slave` and `multipass_slave` elements will be ignored.

If the `-taylor` switch is present, the Taylor map associated with an element, if there is one, is also displayed. An element will have an associated Taylor map if tracking or transfer matrix calculations for the element call for one. For example, if an elements `tracking_method` is set to `Taylor`, it will have an associated Taylor map. To see the Taylor map for an element that does not have an associated map, use the `show taylor_map` command.

If the `-field` switch is present, any associated Electro-magnetic field maps or grid data is printed. For example, wiggler terms for a `map_type wiggler` element are printed.

If the `-xfer_mat` switch is present, the 6x6 transfer matrix (the first order part of the transfer map) along with the zeroth order part of the transfer map are printed.

The `-all` switch is equivalent to using:
```
  -attributes
  -floor_coords
  -taylor
  -wall
  -xfer_mat
```
If the element has a field map, the `-all` switch will print map parameters (such as the spacing between points) but not the entire field table itself. To print the field table as well, use the `-field` switch.

Example:
```
  show ele quad::z* -no_slaves   ! list all non-slave quadrupole elements with
                                 !   names beginning with "z".
  show ele q10w                  ! Show a particular lattice element.
  show ele -att 105              ! Show element #105 in the lattice.
```

11.30.13 emittance

Syntax:
```
  show emittance {-element <ele_id>} {-sigma_matrix} {-universe <uni_index>} {-xmatrix}
```
The `show emittance` command shows, for a given lattice branch, the three normal mode emittances as calculated by PTC, a full non-PTC based 6D calculation, and via radiation integrals evaluation (also non-PTC).

The `-element` switch is used to select what lattice element is used as the end points of the one-turn integrals that are used in the calculation. If the `-element` switch is not present, the beginning element of the default branch (set by `set default branch`) is used. With radiation damping, the emittance is not an exact invariant of the motion. Thus the calculated emittance will vary depending upon what lattice element is used.

If the `-sigma_matrix` switch is present, the sigma matrix at the given element will be displayed along with the emittances.

If the `-xmatrix` switch is present, instead of showing emittances, the damping and stochastic kick transfer matrices used for particle tracking are displayed for the element given by the `-element` switch.

Examples:

```
  show emit -ele 1>>q7  ! Use element q7 in branch #1
```

11.30.14 show field

Syntax:

```
  show field <ele> {-derivatives} {-absolute_s} {-percent_len} <x> <y> <s> {<t-or-z>}
```

The `show field` command shows the electric and magnetic field at a point in space-time and, if the `-derivatives` switch is present, the field derivatives as well as curl and divergence are also printed.

`<ele>` is the lattice element whose fields are to be displayed. The syntax for `<ele>` is explained in section §4.3. Wild card characters are permitted. If multiple elements are matched, the field for each will be printed.

`<x>`, and `<y>` are the transverse coordinates and `<s>` coordinate is the longitudinal position with respect to the beginning of the element.

The `<t-or-z>` argument is optional and specifies the time if absolute time tracking is being used or the phase space z value if relative time tracking is being used (use the `show universe` command to see if absolute time tracking is used or not). The `<t-or-z>` argument is only useful for elements with RF fields. If not set, `<t-or-z>` will default to zero.

Expressions can be used for all real quantities. An expression must be quoted if it contains any blank spaces or, simpler, any blank spaces can be removed.

If the `-absolute_s` switch is present, the `<s>` value will be relative to the start of the element's lattice branch instead of relative to the start of the element.

If the `-percent_len` switch is present, the `<s>` value will be taken as a percentage of the element length with 0.0 representing the upstream end of the element and 1.0 representing the downstream end.

```
  show field q1 0.1  0.2, 0.5*ele::q1[L]  ! Show field at (x, y) = (0.1, 0.2) and
                                          !    at s-center of q1 element.
  show field q1 -percent 0.1  0.2 0.5 ! Same as above.
  show field 2>>3 0 0 0 -deriv        ! Show field at start of 3rd element in branch 2.
```

11.30.15 show global

Syntax:

```
  show global {-bmad_com} {environment} {-optimization} {-ptc_com}
                                                 {-ran_state} {-space_charge_com}
```

The `show global` command prints lists of global parameters. Only one of the optional arguments may be used at one time.

The `-environment` switch shows environment variables defined in any of the *Bmad* lattice files. See the *Bmad* manual for details. If there are multiple universes, only the environment variables associated with the last universe are displayed.

Note: The state of the random number generator is only used for debugging purposes and is not of interest to the typical user.

Specifically:
```
  show global                   ! Displays Tao's global parameters.
  show global -bmad_com         ! Displays bmad_com parameters (§10.6).
  show global -environment      ! Displays environment variables defined in the lattice files.
  show global -optimization     ! Displays optimization parameters.
  show global -ran_state        ! Displays the state of the random number generator.
  show global -space_charge_com ! Displays space_charge_com parameters (§10.6).
```
Use the `set` command to set global parameters

11.30.16 show graph

Syntax:
```
  show graph <graph_name>
```
Show information on a particular graph of a particular plot. See §7 for the syntax on plot, graph, and curve names. Use `show plot` to get a list of plot names.

If there are visible plots with the same name as the plot parameter of `<graph>`, a template plot of the same name is ignored. To show template plot graphs(s) in this case, add a "`T::`" prefix.

Also see: `show plot` and `show curve` commands.

Example:
```
  show graph r2.g1         ! Show the attributes of graph "g1" which is
                           !   plotted in region "r2".
```

11.30.17 show history

Syntax:
```
  show history {-filed} {-no_num} {<num_to_display>}
```
Shows the command history. Each command is given an index number starting from 1 for the first command. This index is printed with the command unless the `-no_num` switch is present. If the `-filed` switch is present, the numbering is shifted so that the current `show history` command has index zero.

The number of commands printed is, by default, the last 50. Setting the `<num_to_display>` will change this. Setting `<num_to_display>` to `all` will cause all the commands to be printed.

Use the command `re_execute` (§11.23) to re-execute a command. Also the up and down arrow keys on the keyboard can be used to scroll through the command history stack.

If a command file has been called, the commands within the command file will be displayed but will be proceeded by an exclamation mark "!" to show that the command was not "directly" executed.

Commands from previous sessions of *Tao* are saved in the file `/.history_tao`. By default they are not displayed. Use the `-filed` switch to include commands from previous sessions.

Examples
```
  show -write cmd_file hist all -no    ! Create a command history file
  show hist 30                         ! Show the last 30 commands.
```

11.30.18 show hom

Syntax:

```
  show hom
```

Shows long–range higher order mode information for linac accelerating cavities.

11.30.19 show internal

The `show internal` command is for printing parameter values that are internal to *Tao*. This command is used for code debugging and not useful (nor understandable) to non-programmers. Note to programmers: Further information is contained in the code that executes the `show internal` command.

11.30.20 show key_bindings

Syntax:

```
  show key_bindings
```

Shows all key bindings (§12.1).

11.30.21 show lattice

Syntax:

```
  show lattice {-0undef} {-all} {-attribute <attrib>} {-base} {-beginning}
      {-blank_replacement <string>}  {-branch <name_or_index>} {-center}
      {-custom <file_name>} {-design} {-floor_coords} {-lords} {-middle}
      {-no_label_lines} {-no_slaves} {-no_super_slaves} {-no_tail_lines} {-orbit}
      {-pipe} {-radiation_integrals} {-remove_line_if_zero <column #>}
      {-rms} {-s <s1>:<s2>} {-spin} {-sum_radiation_integrals} {-tracking_elements}
      {-undef0} {-universe <uni_index>} {<element_list>}
```

Show a table of Twiss and orbit data, etc. at the specified element locations. The default is to show the parameters at the exit end of the elements. To show the parameters in the middle use the `-middle` switch.

By default, the appropriate element(s) within the `model` lattice (§3.3) are used. This can be overridden by using the `-base` or the `-design` switches which switch the lattice to the `base` or `design` lattices respectively.

-0undef
: See the `-undef0` attribute for a description. Also see the `blank_replacement` switch.

-all
: For lattices with a large number of elements, the `show lattice` command defaults to only showing the first 200 elements or so to prevent the accidental generation of possibly tens of thousands of lines. The `-all` switch overrides this default and shows all tracking and lord elements. Also see the `-lords`, `-no_slaves`, `no_super_slaves`, `-tracking_elements` switches.

-attribute <attrib>
: Instead of defining a custom file, the `-attribute <attrib>` switch can be used as a shortcut way for customizing the output columns. When using the `-attribute` switch, the first five columns are the the same default columns of `index`, `name`, `element key`, `s` and `length`. All additional

columns are determined by the `-attribute` switch. Multiple `-attribute` switches can be present and the number of additional columns will be equal to the number of times `-attribute` is used. The `<attrib>` parameter for each `-attribute` switch specifies what attribute will be printed. The general form of `<attrib>` is:

```
  attribute-name         or
  attribute-name@format
```

where `attribute-name` is the name of an attribute and `format` specifies the Fortran style edit descriptors to be used (§4.10). The default format is `es12.4`. Example:

```
  show lat -attrib is_on@l4 -attrib voltage rfcavity::*
```

In the above example, `-attribute` appears twice and the total number of columns of output will thus be 7 (= 5 + 2). The sixth column will have the `is_on` element attribute and will be printed using the `l4` format (logical with a field width of 4 characters). The seventh column will show the voltage attribute.

Note: Data can be used in custom output but data is evaluated independent of whether the `-middle` switch is used.

Also see the `-0undef`, `-undef0`, and `-blank_replacement` switches.

-base
Show values from the `base` lattice instead of the `model` lattice. Also see the `-design` switch.

-beginning
Show value evaluated at the beginning of the lattice elements instead of the default exit end. The `-beginning` switch is ignored when displaying "Intrinsic" element parameters such as the element's length or an element's field strength (which can be displayed using the `-attribute` switch as discussed below). Also the `-beginning` switch is ignored when displaying beam based parameters. Also see `-middle`.

-blank_replacement <string>
The `-blank_replacement` switch specifies that whenever a blank string is encountered (for example, the `type` attribute for an element can be blank), `<string>` should be substituted in its place. `<string>` may not contain any blank characters. Example:

```
  show lat -cust custom.file -blank zz 1:100
```

This will replace any blank fields with "zz".

-branch
The `-branch <name_or_index>` option can be used to specify the branch of the lattice. `<name_or_index>` can be the name or index of the branch. The default is the main branch (# 0).

-center
Same as `-middle`. See `-middle` documentation for more details.

-custom <file_name>
A table with customized columns may be constructed either by using the `-custom` switch which specifies a file containing a description of the custom columns or by using one or more `-attribute` switches. Example customization file:

```
  &custom_show_list
    col(1)  = "#",                    "i6"
    col(2)  = "x",                    "2x"    ! two blank spaces
    col(3)  = "ele::#[name]",         "a0"
    col(4)  = "ele::#[key]",          "a16"
    col(5)  = "ele::#[s]",            "f10.3"
    col(6)  = "ele::#[l]",            "f10.3"
    col(7)  = "ele::#[beta_a]",       "f7.2"
    col(8)  = "1e3 * ele::#[orbit_x]",  "f8.3", "Orbit_x| (mm)"
    col(9)  = "lat::unstable.orbit[#]", "f9.3"
    col(10) = "beam::n_particle_loss[#]", "i8"
  /
```

each `col(n)` line has three parameters. The first parameter is what is to be displayed in that column. Algebraic expressions are permitted (§4.4). The second parameter is the Fortran edit descriptor. Notice that strings (like the element name) are left justified and numbers are right justified. In the case of a number followed by a string, there will be no white space in between. The use of an "x" column can solve this problem. A field width of 0, which can only be used for an `ele::#[name]` column, indicates that the field width will be taken to be one greater then the maximum characters of any element name.

The last parameter is column title name. This parameter is optional and if not present then *Tao* will choose something appropriate. The column title can be split into two lines using `"|"` as a separator. In the example above, The column title corresponding to `"Orbit_x| (mm)"` will have "Orbit_x" printed in one row of the title and "(mm)" in the next row.

To encode the element index, use a `#` or `#index`. To encode the branch index, use `#branch`. Any element attribute is permitted ("show ele" will show element attributes or see the Bmad manual). Additionally, the following are recognized:

```
x                           ! Add spaces
#                           ! Index number of element.
ele::#[name]                ! Name of element.
ele::#[key]                 ! Type of element (‘‘quadrupole’’, etc.)
ele::#[slave_status]        ! Slave type (‘‘super_slave’’, etc.)
ele::#[lord_status]         ! Slave type (‘‘multipass_lord’’, etc.)
ele::#[type]                ! Element type string (see Bmad manual).
```

Note: Data can be used in custom output but data is evaluated independent of whether the `-middle` switch is used.

Also see the `-0undef`, `-undef0`, and `-blank_replacement` switches.

-design
Show values from the `design` lattice instead of the `model` lattice. Also see the `-base` switch.

<element_list>
The locations to show are specified either by specifying an element list or by specifying a longitudinal position range using the `-s` switch The syntax used for specifying the element list is given in the `Lattice Element List Format` section (§4.3). In this case there should be no blank characters in the list.

-floor_coords
If present, the `-floor_coords` switch will print the global floor (laboratory) coordinates for each element. If used with the `-orbit` option, the orbit in floor coordinates will be shown.

-lords
If present, the `-lords` switch will print a list of lord elements only. Also see the `-all`, `-no_slaves`, `no_super_slaves`, `-tracking_elements` switches.

-middle
Show value evaluated at the middle of the lattice elements instead of the default exit end. The `-middle` switch is ignored when displaying "Intrinsic" element parameters such as the element's length or an element's field strength (which can be displayed using the `-attribute` switch as discussed below). Also the `-middle` switch is ignored when displaying beam based parameters. Also see `-beginning`.

-no_label_lines
If present, the `-no_label_lines` switch will prevent the printing of the header (containing the column labels) lines at the top and bottom of the table. This is useful when the output needs to be read in by another program. Also see the `-no_tail_lines` switch.

-no_slaves
If the `-no_slaves` switch is present, all `super slave` and `multipass slave` elements will be ignored. Also see the `-all`, `-lords`, `no_super_slaves`, `-tracking_elements` switches.

-no_super_slaves
If present, the `-no_super_slaves` switch will veto from the list of elements to print all `super slave` elements. Also see the `-all`, `-lords`, `-no_slaves`, `no_super_slaves`, `-tracking_elements` switches.

-no_tail_lines
The `-no_tail_lines` just suppress the header lines at the bottom of the table. Also see the `-no_label_lines` switch.

-orbit
The `-orbit` switch will show the particle's phase space orbit which is the closed orbit if the lattice has a closed geometry and is the orbit beginning from the specified starting position for lattices with an open geometry. Use `set particle_start` to vary the starting position in this case. If the `-spin` switch is also present, the particle's spin will also be displayed. If used with the `-floor_coords` option, the orbit in floor coordinates will be shown.

-pipe
The `-pipe` switch gives a comma delimited table as output. This switch is used with the `pipe` command (§11.18).

-radiation_integrals
The `-radiation_integrals` switch, if present, will display the radiation integrals for each lattice element instead of the standard Twiss and orbit data. See the *Bmad* manual for the definitions of the radiation integrals. Also see `-sum_radiation_integrals`.

-remove_line_if_zero <column #>
If present, the `-remove_line_if_zero` switch will suppress any lines where the value in the column given by `<column #>` is zero or not defined. Notice that when specifying custom columns using the `-custom` switch, columns that only insert blank space are not counted. For example:
```
  show lat -custom cust.table -remove 5
```
Assuming that the file `cust.table` contains the example customization given above, the fifth visible column corresponds to `column(6)` which prints the element length. The `-remove 5` will then remove all lines associated with elements whose length is zero. Multiple `-remove_line_if_zero` may be present. In this case the row will be suppressed if all designated columns have a zero entry.

-rms
When the `-rms` switch is present, five additional lines will be added to the output showing the mean and RMS values for columns that had floating point numbers along with mean and RMS values integrated over the longitudinal s-position. A simple trapezoid integration is used for the integrated values. The fifth row shows the number of data points. Only valid values will be included in the calculation.

-s <s1>:<s2>
The locations to show are specified either by specifying a longitudinal position range with `-s`, or by specifying a list `<element_list>` of elements.

-spin
The `-spin` switch will show the particle's spin which is the invariant spin if the lattice has a closed geometry and is the spin beginning from the specified starting spin for lattices with an open geometry. Use `set particle_start` to vary the starting spin in this case. If the `-orbit` switch is also present, the particle's phase space orbit will also be displayed.

-sum_radiation_integrals
The `-sum_radiation_integrals` switch, if present, will display the radiation integrals integrated

from the start of the lattice to each lattice element. See the *Bmad* manual for the definitions of the radiation integrals. Also see the `-radiation_integrals` switch.

-tracking_elements
The `-tracking_elements` switch can be used to show all the elements in the tracking part of the lattice. Also see the `-all`, `-lords`, `-no_slaves`, `no_super_slaves` switches.

-undef0
If an attribute does not exist for a given element (for example, `quadrupoles` do not have a `voltage`), a series of dashes, "----", will be placed in the appropriate spot in the table. Additionally, an arithmetic expression that results in a divide by zero will result in dashes being printed. This behavior is changed if the `-0undef` or `-undef0` switch is present. In this case, a zero, "0", will be printed. The difference between `-0undef` and `-undef0` is that with `-undef0` the zero will be printed using the same format as the other numbers in the column. With the `-0undef` switch the zero will be printed as a right justified "0" which gives a visual clue to differentiate between a true zero value and a zero that represents an undefined parameter.

-universe <index>
The `-universe` switch specifies which universe is used. If not present, the current viewed universe is used.

Examples:
```
  show lattice 50:100         ! Show lattice elements with index 50 through 100
  show lat 45:76, 101, 106    ! Show element #45 through #76 and 101 and 106.
  show lat q34w:q45e          ! Show from element q34w through q45e.
  show lat q*                 ! Show elements whose name begins with "q"
  show lat marker::bpm*       ! Show marker elements whose name begins with "bpm"
  show lat -s 23.9:55.3       ! Show elements whose position is between
                              !   23.9 meters and 55.3 meters.
  show lat -att x_offset -rms ! Table will have a column of lattice element x_offset
                              !   values. Additionally, mean and RMS will be shown.
```

11.30.22 show matrix

The `show matrix` command is shorthand for `show taylor_map -order 1`. See the `show taylor_map` documentation for more details on optional arguments, etc.

Also see `write matrix`.

Examples:
```
  show matrix q10w q12e           ! 0th and 1st order maps from q10w to q12e
  show taylor -order 1 q10w q12e  ! Same as above.
  show matrix -ele *              ! Show matrices for all lattice elements.
```

11.30.23 show merit

Syntax:
```
  show merit {-derivative} {-merit_only}
```
If the `-derivative` switch is present, this command shows top dMerit/dVariable derivatives, and Largest changes in variable value. If not present, this command shows top contributors to the merit function.

Also see: `show constraints`.

If the `-merit_only` switch is present, only the value of the merit function is printed and nothing else. That is, it makes the output compact if only the value of the merit function is desired.

Note: To set the number of top contributors shown, use the command

```
  set global n_top10_merit = <number>
```

where `<number>` is the desired number of top contributors to the merit function to be shown.

Note: The `show merit` command was once called the `show top10` command.

Example:

```
  show merit -der     ! Show merit derivative info
```

11.30.24 show optimizer

Syntax:

```
  show optimizer {-geodesic_lm}
```

Shows parameters pertinent to optimization: Data and variables used, etc.

If `-geodesic_lm` option is present, parameters for the `geodesic_lm` optimizer will be shown. These parameters are shown in any case if the optimizer has been set to use `geodesic_lm`.

Also see:

```
  show constraints
  show data
  show derivative
  show merit
  show variables
```

11.30.25 show particle

Syntax:

```
    show particle {-bunch <bunch_index>} {-particle <particle_index>
                  {-element <element_id>} {-lost} {-all}
```

Shows individual beam particle information except if the the `-lost` or `-all` options are used.

The default for the optional `-bunch` index is set by the global variable `global%bunch_to_plot`. The default `-element` is `init` which is the initial beam distribution. The default `-particle` to show is the particle with index 1.

The `-lost` option shows which particles are lost during beam tracking. Note: Using the `-lost` option results in one line printed for each lost particle. It is thus meant for use with bunches with a small number of particles.

The `-all` option shows all particles at the given element.

The `dtime` column shows the time relative to the reference time $(t - t_{ref})$. The conversion between phase space z and `dtime` is $z = -\beta\, c\, (t - t_{ref})$ where $\beta = v/c$ is the normalized velocity. Since `dtime` is undefined if $\beta = 0$, zero will be displayed in this case.

Also see `show beam`.

Examples:

```
  show part -bun 3 -part 47 -ele 8 ! Shows information on particle #47 of
                                   !   bunch #3 at lattice element #8.
  show part -part 47 -ele 8        ! Same as above except the default bunch is used.
  show part -lost -bun 3           ! Show lost particle positions for bunch #3
```

11.30.26 show plot

Syntax:

```
show plot {-floor_plan} {-page} {-lat_layout} {-regions}
          {-templates} {<plot_or_template_name>}
```

The `show plot -floor_plan` and `show plot -lat_layout` commands show the parameters associated with the `floor_plan` or `lat_layout` plots (§10.13.9). Use the `set floor_plan` or `set lat_layout` commands to set these parameters.

The `show plot -page` command shows some plot page plotting parameters like the size of the plot window.

The `show plot -regions` command shows what plots are placed in which regions. Use the `place` command to change where plots are placed.

The `show plot -templates` command displays what plot templates have been defined for plotting. See §10.13 for information on setting up template plots.

The `show plot <plot_or_region_name>` command will display information on a particular plot. If there are visible plots with the same name, a template plot of the same name is ignored. To show a template plot in this case, add a "`T::`" prefix.

The various `show plot` options are mutually exclusive and only the last option is used. That is, a command like

```
show plot -lat_layout -regions
```

is equivalent to `show plot -regions`.

Also see `show graph` and `show_curve`.

Examples:

```
show plot         ! Show plot region information by default.
show plot r13     ! Show information on plot in region r13.
show plot T::beta ! Show template beta plot.
```

11.30.27 show ptc

Syntax:

```
show ptc
```

Show quantities as calculated by PTC. This command is under development. Currently emittances and tunes are shown.

11.30.28 show radiation_integrals

Syntax:

```
show radiation {-branch <branch_name>} {universe_number}
```

Show radiation integrals along with associated parameters computed from the integrals like the emittance, damping decrement, etc. Values for the associated parameters like the emittance will vary from the values shown by the `show universe` command since the `show universe` command uses a computation that derives from the transport matrices with included radiation effects. Differences between the two are due to differing approximations as explained in the *Bmad* manual (See the chapter on `Synchrotron Radiation`).

11.30.29 show rampers

Syntax:
```
  show rampers {-universe <ix_uni>} {-energy_show}
```
Shows information on ramper lords and slaves.

The `-universe` switch can be used to choose which universe to use.

By default, slave listings will omit references to energy ramp control (control of `p0c` or `E_tot`. The reason for this is that typically all elements will be involved in energy ramping and this would make the listing very long. To explicitly display the energy ramp control, use the `-energy_show` switch.

Examples:
```
  show ramp -energy_show  ! Show includes energy ramp info.
```

11.30.30 show spin

Syntax:
```
  show spin {-element {<ref_ele_name>} <ele_name>} {-flip_n_axis} {-g_map}
                {-ignore_kinetic <ele_list>} {-isf} {-spin_tune}
                {-l_axis <lx>, <ly>, <lz>} {-n_axis <nx>, <ny>, <nz>}
                {-x_zero <ele_list>} {-y_zero <ele_list>} {-z_zero <ele_list>}
```
Show spin related information. Note: To see the closed orbit invariant spin at any element, make sure spin tracking is on (if not, use: `set bmad_com spin_tracking_on = T`), and then the `show element` command will display n_0.

If `-element` is not present, `show spin` will show various quantities including polarization limits and polarization rates. See the `Spin Dynamics` chapter of the *Bmad* manual for a discussion of how these quantities are calculated.

If `-element` is present, the output will be the first order spin transfer map from the downstream end of the element given by `<ref_ele_name>` to the downstream end of the element given by `<ele_name>`. If `<ref_ele_name>` is not given, the default is the element given by `<ele_name>`. This gives the 1-turn spin transport from the downstream end of `<ele_name>`. The spin map can be displayed in two different forms. If the `-l_axis` or `-n_axis` is given, or if `-g_map` is present, the G-matrix form of the spin map will be printed. The G-matrix is dependent upon the $(\mathbf{l}, \mathbf{n}, \mathbf{m})$ axes used to define the spin vector (see the `SLIM Formalism` section of the Spin chapter in the Bmad manual). The **n** and **l** axes can be specified using the `-n_axis` and `-l_axis` switches. The **m**-axis is calculated from knowledge of the other two axes. If not given, the **n**-axes will be set to the reference orbit $\mathbf{n}_0$ value (the reference orbit is the closed orbit if the lattice geometry is closed). The **l**-axis, if not given, will be chosen to be perpendicular to **n**. Commas between axis parameters are optional. If `q_map` is present, or if the G-matrix is not being printed, the quaternion form of the map will be printed.

To see the linear spin-orbit resonance strengths, use the `-ele` switch with either `<ref_ele_name>` being blank or the same as `<ele_name>`. The spin-orbit resonance strength table has three rows for the three orbital modes a, b, and c. The columns of the table are:

- Col 1: Name of mode
- Col 2: Orbital tune.
- Col 3: Q_spin + Q_mode + N where N is an integer that minimizes this expression. The sum resonance occurs when this is zero.
- Col 4: Sum resonance strength. This number is only accurate if the value of column 3 is small compared to one.

- Col 5: Q_spin - Q_mode + N where N is an integer that minimizes this expression. The difference resonance occurs when this is zero.
- Col 6: Difference resonance strength. This number is only accurate if the value of column 5 is small compared to one.

See the Bmad manual section on the linear spin/orbit resonance analysis for more details. The `pipe spin_resonance` command can be used to extract resonance values when using running *Tao* with a script.

The `-flip_n_axis` switch, if present, will flip the direction of the displayed n-axis.

The `-isf` switch, if present, will print the invariant spin field which are three taylor series for the three components of the spin (S_x, S_y, S_z). The independent variables are the six orbital phase space coordinates (x, p_x, y, p_y, z, p_z).

The `-spin_tune` switch, if present, will print the amplitude-dependent spin tune. The output will be a Taylor series in the phasor's basis, i.e. $x_k = sqrt(J_k) * exp(i * phi_k)$. For example the monomial "[1 1 0 0 0 0]" corresponds to J_a, and if RF is on then "[1 1 2 2 3 3]" corresponds to $J_a * J_b^2 * J_c^3$.

The `-x_zero`, `-y_zero`, and `-z_zero` options are for testing if suppressing certain terms in the linear part of the spin transport map for a set of elements selected by the user will significantly affect the polarization. This is discussed in the section "`Linear dn/dpz Calculation`" in the *Bmad* manual. In particular, `-x_zero` will zero the $\mathbf{q}_1$ and $\mathbf{q}_2$ terms in the $\overrightarrow{\mathbf{q}}$ vector (equivalent to zeroing $\mathbf{G}_x$ in the SLIM formalism) for the spin transport map of the chosen lattice elements. Similarly, `-y_zero` and `-z_zero` will zero vertical and longitudinal components. Element list format (§4.3), without any embedded blanks, is used for the `<ele_list>` list of elements to apply to.

The `-ignore_kinetic` option can be used such that the kinetic term in the Derbenev-Kondratenko polarization formula coming from lattice elements specified by `<lat_list>` are not used in evaluating the polarization. Element list format (§4.3), without any embedded blanks, is used for the `<ele_list>` list of elements to apply to.

Example:
```
  show spin
  show spin -ele Q1-1 Q1 -n 0,1,0 -l 1,0,0  ! G-matrix for Q1
  show spin -ele S1       ! 1-turn analysis at downstream end of S1
  show spin -ele 4 7 -g   ! G-matrix from end of element 4 to end of element 7
  show spin -x_zer s1:s2 ! Zero q1 and q2 components for all lattice elements
                          ! between elements s1 and s2.
  show spin -ign B0%     ! Ignore the kinetic term for all elements whose name has three
                          !   characters and starts with "B0".
```

11.30.31 show string

Syntax:
```
  show string {string-to-print}
```
Print a string. This can be useful when creating a data file. Use "\n" to output multiple lines. Anything within backticks, \`...\`, will be evaluated. If an evaluated quantity is an array, the array is enclosed in brackets "[...]". Also if the evaluated quanty ends in `‘‘@@N’’` where `N` is an integer, this is used to determine the accuracy of the printed value(s). The default is 14. Also see `show value`.

Examples:
```
  show -append a.dat str 2 + 2 = `2+2` ! Writes the line "2 + 2 = 4" to a.dat
  show str `1e3*lat::orbit.x[3:5]@@2`  ! Prints something like "[3.4, 3.2, 2.7]"
```

11.30.32 show symbolic_numbers

Syntax:

```
  show symbolic_numbers {-physical_constants} {-lattice_constants}
```

Show the symbolic constants created using the `set symbolic_number` command.

If the `-physical_constants` switch is present, the predefined physical constants (like `c_light`) along with predefined mathematical constants (like `pi`) are displayed instead (Also see the *Bmad* manual for this list).

If the `-lattice_constants` switch is present, constants defined in the lattice are displayed. Note: To import these symbols into Tao, set `global%symbol_import` to True (or use the `-symbol_import` switch on the startup command line). The default is to not import lattice symbols.

Examples:

```
  set sym aaa = 23  ! Set a symbol.
  show sym          ! Show all user defined symbols.
  show sym -phys    ! Show predefined physical and mathematical constants.
```

11.30.33 show taylor_map

Syntax:

```
  show taylor_map {-angle_coordinates} {-eigen_modes} {-elements <ele_list>}
                 {-inverse} {-lattice_format} {-noclean} {-number_format <fmt>}
                 {-order <n_order>} {-ptc} {-radiation} {-s}
                 {-scibmad <suffix>} {-universe <uni>} {loc1 {loc2}}
```

Shows the Taylor transfer map for the `model` lattice of the default universe (set by `set default universe`). Maps are computed about the particle orbit. Not the zero orbit. Also see `write matrix`. Note: the `show matrix` command is equivalent to the `show taylor -order 1` command.

If the `-elements` switch is present, other switches are ignored except `-order` and the individual element transfer maps are printed for each element specified by `<ele_list>`. The default order is one. Element list format (§4.3), without any embedded blanks, is used for the `<ele_list>` list of elements.

If neither `loc1` nor `loc2` are present, the transfer map is computed for the entire lattice.

if `loc1` and `loc2` are the same, the 1-turn transfer map is computed. If the s-position of `loc1` is greater than the s-position of `loc2`, the map from `loc1` to the end of the lattice with the map from the beginning to `loc2` is computed.

If the `-s` switch is present, `loc1` and `loc2` will be interpreted as longitudinal s-positions. In this case, if `loc2` is not present, the map will be the 1-turn map if the lattice is circular and the map from the beginning to `loc1` if the map is not.

If the `-s` switch is not present, `loc1` and `loc2` will be interpreted as element names or indexes. The map will be from the exit end of the `loc1` element to the exit end of the `loc2` element. In this case, if `loc2` is not present, the map will be the for the element given by `loc1`

Expressions can be used for all real quantities (that is, `loc1` and `loc2` if `-s` is present). An expression must be quoted if it contains any blank spaces or, simpler, any blank spaces can be removed.

If the `-eigen_modes` switch is present, the first order part of the map will be treated as a 1-turn matrix and the corresponding eigen values and eigen vectors will be printed.

If the `-inverse` switch is present, the inverse of the map is displayed.

The `-noclean` switch, if present, prevents *Tao* from "cleaning" the Taylor map. Cleaning is the process of dropping terms that are very small.

The `-number_format` switch, if present, overrides the default format for the displayed format. Examples:

```
  show taylor -num f12.6  ! Fixed format. 12 char width, 6 digits after decimal point.
  show taylor -num es12.4 ! Float format. 12 char width, 4 digits after decimal point.
```

The `-order` switch, if present, gives the limiting order to display. In any case, the maximum order of the map is limited to the order set by the lattice file.

The `-ptc` switch is used with `-order 1`. By default, order 1 maps (matrices) are calculated using native Bmad code. If the `-ptc` switch is present, the matrix is calculated using the `PTC` code (see the *Bmad* manual for details on PTC). Since PTC is always used to calculate maps of order higher than 1, the `-ptc` switch is ignored for higher orders.

If used with the `-radiation` switch, the `-ptc` switch will cause the radiation to be calculated around the closed orbit as calculated from PTC as opposed to the orbit calculated by Bmad native code. This is used as a check that the Bmad and PTC closed orbits are not significantly different from one another.

If the the `-radiation` switch is present, displayed will be the linear map with radiation damping and excitation. See the `Synchrotron Radiation` chapter in the *Bmad* manual for how the matrices displayed here are defined. The RF cavities should, in general be powered since these affect the reference orbit and transfer matrix. The damping matrix will be computed independent of whether radiation damping is on or off for tracking. The difference is that the reference orbit is affected by having damping on or off. Having radiation damping off in fact may be the preferable since in an actual machine the "sawtooth" orbit which comes with having damping on will tend to be compensated by tuning of the machine to meet the design conditions. If the `-ptc` switch is used with `-radiation`, the PTC calculated closed orbit will be used as a reference instead of the *Bmad* one.

To toggle radiation damping and RF use the commands:

```
  set bmad_com radiation_damping_on = T
  set global rf_on = T
```

Note that the calculation does not depend upon the radiation excitation being turned on (since radiation excitation will not affect the reference orbit or transfer matrix).

The `-angle_coordinates` switch, if present, causes the output to be displayed using "angle" phase space coordinates (x, x', y, y', z, p_z) in place of the standard *Bmad* canonical coordinates (x, p_x, y, p_y, z, p_z). (Note: The conversion between the two coordinate systems is given in the Bmad manual.

The `-lattice_format` switch, if present, causes the output to be displayed in a format suitable for using in a *Bmad* lattice file.

The `-scibmad <suffix>` switch, if present, causes the output to be displayed in SciBmad format. The `<suffix>` string is used to form the name of the variable holding the Taylor map. The variable name uses the string `v_` as the prefix. For example, if the `<suffix>` is `z2`, the variable name will be `v_z2`.

Examples:

```
  show taylor -order 1 q10w q12e  ! 0th and 1st order maps from q10w to q12e
  show taylor 45                  ! Transfer map of element #45
  show taylor -s 13 23            ! Transfer map from s = 13 meters to 23 meters.
  show taylor -ele quad::*        ! Show transfer matrices for all quadrupole elements.
```

11.30.34 show track

Syntax:

```
show track {-b_field {<fmt>}} {-base} {-branch <name_or_index>} {-design}
    {-dispersion {<fmt>}} {-e_field {<fmt>}} {-element <ele_id>} {-momentum {<fmt>}}
    {-no_label_lines} {-points <num>} {-position {<fmt>}} {-energy {<fmt>}}
    {-range <s1> <s2>} {-s {<fmt>}} {-spin {<fmt>}} {-time {<fmt>}}
    {-twiss {<fmt>}} {-universe <ix_uni>} {-velocity {<fmt>}}
```

The `show track` command shows a table of phase space coords, Twiss parameters, EM fields, and other info at equally spaced points along the tracked orbit. Also see the `show twiss_and_orbit` command.

Command arguments that toggle whether a certain quantity is displayed have an optional `<fmt>` format specifier that can be used to set the format of the displayed quantities. The format uses Fortran edit descriptor syntax (§4.10). If "`no`" is used as the format then the associated quantity will not be displayed. If there is no format specified then *Tao* will use a default format. Example:

```
show track -b_field      ! Display magnetic field parameters using the default format
show track -position no  ! Do not display position information.
show track -s 3pf12.1    ! Display S-position with decimal point shifted by 3 places.
                         !   That is, display the S-position in millimeters.
```

When the value of quantities are shifted, using the "`P`" prefix, the header string for the corresponding column(s) will be appropriately marked.

`{-b_field {<fmt>}}`
: Set the format for the three parameters of the magnetic field (in Tesla). The default, if `-b_field` is not present, is not to print the field.

`{-base}`
: If present, use the `base` lattice for evaluating quantities. The default is the `model` lattice.

`{-branch <name_or_index>}`
: Lattice branch to use. The default is the default branch (§3.4)

`{-design}`
: If present, use the `design` lattice for evaluating quantities. The default is the `model` lattice.

`{-dispersion {<fmt>}}`
: Set the format for the dispersion and dispersion derivative columns ($\eta_x, \eta'_x, \eta_y, \eta'_y$). The default is not to print these columns.

`{-e_field {<fmt>}}`
: Set the format for the three parameters of the electric field (in V/m). The default is not to print the field.

`{-element <ele_id>}`
: The `-element` switch can be used instead of `-range` to set the s-position. With `-element` the track range is the extent of the element given by `<ele_id>`. Note: If the element is a `beambeam` element, the track will show the before and after particle positions at each strong beam slice along with the initial position after tracking through the previous lattice element along with the final position which is used to start tracking through the next element.

`{-momentum {<fmt>}}`
: Set the format for the three phase space momentum parameters (p_x, p_y, p_z). Notice that these are canonical momenta and are dimensionless as explained in the *Bmad* manual. In particular, p_z is the momentum deviation from the reference momentum. The default is to print the momenta using the default format.

`{-no_label_lines}`
: If present then suppress the output header lines.

`{-points <num>}`
: Set the number of evaluation points. That is, set the number of rows in the table.

`{-position {<fmt>}}`
: Set the format for the three phase space position parameters (x, y, z). See the *Bmad* manual for details on phase space coordinates. The default is to print the position using the default format. The default format is `3PF14.6` so the output will be in mm.

`{-energy {<fmt>}}`
: Set the format for the column showing the total energy (in eV) of the particle. The default is not to print this.

`{-range <s1> <s2>}`
: Set the S-position min/max bounds for the table. Default is beginning and ending s-positions of the lattice.

`{-s {<fmt>}}`
: Set the format for the S-position column. The default, if `-s` is not present, is to print the column.

`{-spin {<fmt>}}`
: Set the format for the three parameters of the particle's spin. The default, if `-spin` is not present, is not to print the spin.

`{-time {<fmt>}}`
: Set the format for the time column. The default, if `-s` is not present, is to not the column.

`{-twiss {<fmt>}}`
: Set the format for the Beta and Alpha functions of the two transverse normal modes. The default, if `-twiss` is not present, is not to print the Twiss parameters

`{-universe <ix_uni>}`
: Set the universe to use. The default is the default universe (§3.3).

`{-velocity {<fmt>}}`
: Set the format for the three particle velocity parameters $(v_x/c, v_y/c, v_z/c)$ normalized by the speed of light. The default is not to print the velocity.

11.30.35 show twiss_and_orbit

Syntax:

```
show twiss_and_orbit {-base} {-branch <name_or_index>} {-design}
{-universe <ix_uni>} <s_position>
```

The `show twiss_and_orbit` shows Twiss and orbit information at a given longitudinal position `<s_position>` including synchrotron radiation related parameters. Also see `show track`.

The default universe to use is the current default universe. This can be changed using the `-universe` switch.

The default is to show the `model` Twiss and orbit parameters. The use of `-base` or `-design` switches can be used to show parameters for the `base` or `design` lattices.

The particular branch used in the analysis can be selected by the `-branch` switch. The default is the default branch (§3.4).

Examples:

```
show twiss -uni 2 23.7     ! Show parameters in universe 2 at s = 23.7 meters.
```

11.30.36 show universe

Syntax:

```
show universe {-branch <branch_name>} {universe_number}
```

Shows various parameters associated with a given branch of a given universe. If no universe is specified, the current default universe is used. If no branch is given, the current default branch is used. Parameters displayed include tune, emittances, etc.

Here quantities like the emittance or momentum compaction factor are calculated from the transport matrices with included radiation effects. Previously (pre April 2022), such quantities where calculated from evaluation of the synchrotron radiation integrals. To see the radiation integral derived values use the command `show radiation_integrals`. Differences between the results of the transport matrix and radiation integral treatments are due to differing approximations as explained in the *Bmad* manual (See the chapter on `Synchrotron Radiation`).

Example:

```
show universe -branch 1 3  ! Show info on branch 1 of universe 3.
```

11.30.37 show use

Syntax:

```
show use
```

Shows what data and variables are used in a format that, if saved to a file, can be read in with a `call` command.

11.30.38 show value

Syntax:

```
show value {#format <format-string>} <expression>
```

Shows the value of an expression. The `#format` switch (notice that `"#"` is used instead of `"-"` to avoid confusion with negative signs) can be used to set the number format. The default format is `es25.17` which results in number printed in scientific notation format with a field width of 25 and 17 digits displayed after the decimal place. Use "`f`" for fixed point numbers. For example, "`f10.3`" will give display numbers in fixed point format with a field width of 10 and 3 digits displayed after the decimal place. Rule: The format string must not contain an embedded blank space.

If the expression is a vector, values will be printed one per line. The exception is that if there is a format specified and if the format has a repeat count[1] or a comma (example "`f9.3,es9.1`"), the values will be printed on one line.

Also see `show string`.

Examples:

```
show value sqrt(3@lat::orbit.x[34]|model) + sin(0.35)
show value #form f10.4 ran_gauss()
show value #form 2es10.2 [ele::q20w[hkick], lat::beta.a[4]] ! Output on one line
```

The last example shows how to

[1]for example "`4f10.3`" has a repeat count of 4.

11.30.39 show variables

Syntax:

```
  show variables {-no_label_lines} {-universe <universes>}
         {-good_opt_only} {-bmad_format} {<var_name>}
```

Shows variable information. If `<var_name>` is not present, a list of all appropriate `v1_var` classes is printed.

The `-universe` switch is used to select only variables what control parameters in a given universe or universes. Use `-universe @` to select the current viewed universe.

If the `-bmad_format` switch is used then the Bmad lattice parameters that the *Tao* variables control will be printed in Bmad lattice format. This is the same syntax used in generating the variable files when an optimizer is run. If `-good_opt_only` is used in conjunction with `-bmad_format` then the list of variables will be restricted to ones that are currently being used in the optimization.

If present, the `-no_label_lines` switch will prevent the printing of the header (containing the column labels) lines. This switch is ignored if `-bmad_format` is present.

Examples:

```
  show var              ! List all v1 variables.
  show var quad_k1      ! List variables in the quad_k1[*] array.
  show var quad_k1[10] ! List detailed information on the variable quad_k1[10].
  show var -uni 2       ! List all variables that control attributes in universe 2.
  show var -bmad        ! List variables in Bmad Lattice format.
```

11.30.40 show version

Syntax:

```
  show version
```

The `show version` command will show the “version” of *Tao* corresponding to *Tao* executable being run. Since *Bmad* and *Tao* are under continuous development, the standard semantic versioning scheme using major and minor numbers does not make sense. Instead, the version string associated with *Tao* is a date encoded in the form

```
  YYYY_MMDD
```

where `YYYY` is the four digit year, `MM` is the two digit month, and `DD` is the two digit day. The version string is stored in the file `$TAO_DIR/VERSION`. For the `show version` command to work properly, the environment variable `TAO_DIR` must be appropriately defined. Generally, `TAO_DIR` will be defined if the appropriate *Bmad* setup script has been run. For “Bmad Distributions”, this is the same setup script used to setup a distribution. See your local *Bmad* guru for details.[2]

11.30.41 show wake_elements

Syntax:

```
  show wake_elements
```

The `show wake_elements` command will list the lattice elements that have associated wake fields. Use the `show ele` command to get more details on a given element. Note that wakes only affect particle

[2]

tracking when tracking with a beam of particles (not when tracking just a single particle which is the default for *Tao*).

At this point in time, *Tao* is not setup to do multiturn tracking with bunches which means that if simulations with wakefields is desired, a different program have to be used like `long_term_tracking`.

11.30.42 show wall

Syntax:

```
show wall {-branch <name_or_index>}{-section <index>} {-angle <angle>}
{-s <s1>:<s2>} {<n1>:<n2>}
```

The `show wall` command shows the vacuum chamber wall associated with a lattice branch.

For the building wall, use the "show building_wall" command.

For showing the wall associated with a given element, use the "show ele -wall" command.

The `-branch` switch is used to select a particular branch.

The `-section` switch is used to show information about a specific chamber wall cross-section. In this case, all the other options are ignored except for `-branch`.

If `-section` is not present, a list of vacuum chamber wall sections is presented. In this case, the range of wall sections shown is given by `<n1>:<n2>` except if `-s` is present in which case all sections within a range of `s` values is given within the range `<s1>` to `<s2>`. With each section, a wall radius is given. The angle in the (x, y) plane at which the radius is computed is determined by the `-angle` option. The default angle is 0 which corresponds to the $+x$ direction.

Examples:

```
show wall 45:100       ! Show vacuum chamber wall sections 45 through 100.
show wall -s 10.0:37.5 ! Show wall sections that have S-position between 10 and 37.5.
show wall -section 49  ! Show chamber wall section 49.
```

11.30.43 show wave

Syntax:

```
show wave
```

The `show wave` command shows the results of the current wave analysis (§9).

11.31 single_mode

The `single_mode` command puts *Tao* into `single mode` (§12). For on-line help when running *Tao* go to `single mode` and type "?". To get out of single mode type "Z".

11.32 spawn

The `spawn` command is used to pass a command to the command shell. Format:

```
spawn <shell_command>
```

The users default shell is used. `spawn` only works in Linux and Unix environments. Note: Shell aliases will not be recognized.

Examples:

```
spawn gv quick_plot.ps &      ! view a postscript file with ghostview
                              ! (and return to the TAO prompt)
spawn tcsh                    ! launch a new tcsh shell
                              ! (type 'exit' to return to TAO)
spawn ls                      ! Get a directory listing.
```

11.33 taper

The `taper` command is used to vary magnet strengths to eliminate the transverse orbital and Twiss changes due to the radiation damping induced "sawtooth" effect. Format:

```
taper {-universe <ix_uni>} {-except <ele_list>}
```

The sawtooth effect is the variation of the energy as a function of longitudinal position due to radiation damping. This affects the transverse closed orbit and Twiss parameters. The `taper` command will adjust magnet strengths in the `model` universe to counteract this making strengths weaker or stronger in proportion to the local closed orbit momentum deviation.

Note: Another way of handling the sawtooth effect is to set `bmad_com%radiation_zero_average` to True. See the *Bmad* manual for more details.

The magnet strengths that are tapered are the "multipole" like parameters:

```
dg, k1, k2, k3, ks, hkick, vkick, kick, a1, a2, ..., b1, b2, ...
```

Notice that something like wiggler strengths are not included.

The `-except` switch is used select lattice elements to not vary their magnetic strength.

Note: Tapering must be done with Rf on.

Important! The scaling of the magnet strengths is done with respect to the `base` lattice (which is the same as the `design` lattice if the `set lattice base = ...` command has not been issued). That is, if two `taper` commands are issued one after another, the second `taper` command will not affect magnet strengths. The reason why this is done is so that successive `taper` commands do not cause the magnet strengths to "walk". That is, there is no unique solution to the taper problem since a changing all magnet strengths along with a corresponding shift in particle energy will not change the orbit.

The `-universe <ix_uni>` switch can be used select the desired universe to taper. If not present, the default universe will be used. Set <ix_uni> to "*" to choose all universes.

The taper command will vary magnet strengths independent of the element's name. That is, the magnet strengths of elements which have the same name will be different. If a *Bmad* lattice file is created after tapering using the `write bmad` command, The created lattice file will have the proper magnet strengths (the lattice file will use the `##N` construct to set strengths for individual elements with the same same). However, creating a lattice file in a non-*Bmad* (MAD, etc.) format will be problematical.

No tapering will be applied to strengths that are controlled by an `overlay` controller element.

Examples:

```
taper -uni *            ! Taper all universes.
taper -exc solenoid::*  ! Taper default uni except solenoid elements.
```

11.34 timer

The `timer` command is used to show computation time. Format:

```
  timer start      ! Start (reset) the timer
  timer read       ! Display the time from the last timer start command.
  timer beam       ! Toggle beam timing mode on/off.
```

The timer has a `beam timing` mode which can be toggled using the `timer beam` command. The initial state, when *Tao* is started, is for `beam timing` to be off. With `beam timing` mode on, when *Tao* is tracking a particle beam through the lattice, *Tao* will print, about once a minute, the element number and the elapsed time.

The `timer start` and `timer read` commands can be used to time execution times. Example:

```
  timer start ; call my_cmd_file ; timer read
```

Note: `timer start` will toggle `beam timing` off.

11.35 use

The `use` command un-vetoes data or variables and sets a veto for the rest of the data. Format:

```
  use data  <data_name>
  use var <var_name>
```

See also the `restore` and `veto` commands.

Examples:

```
  use data orbit.x               ! use orbit.x data in the default universe.
  use data *@orbit[34]           ! use element 34 orbit data in all universes.
  use var quad_k1[67]            ! use variable.
  use var quad_k1[30:60:10]      ! use variables 30, 40, 50 and 60.
  use data *                     ! use all data in the default universe.
  use data *@*                   ! use all data in all universes.
```

11.36 veto

The `veto` command vetoes data or variables. Format:

```
  veto data <data_name> <locations>
  veto var <var_name> <locations>
```

See also the `restore` and `use` commands.

Examples:

```
  veto data orbit.x[23,34:56]  ! veto orbit.x data.
  veto data *@orbit.*[34]      ! veto orbit data in all universes.
  veto var quad_k1[67]         ! veto variable
  veto var quad_k1[30:60:10]   ! veto variables 30, 40, 50 and 60
  veto data *                  ! veto all data
  veto data *[10:20]           ! veto all data from index 10 to 20 (see note)
```

Note: The command '`veto data *.*[10:20]`' will veto all `d1_data` elements within the range 10:20 *using the index convention for each `d1_data` structure separately.* This may produce curious results if the indexes for the `d1_data` structures do not all point to the same lattice elements.

11.37 view

The `view` command is just a shortcut for the `set default universe` command. Format:

```
  view <universe-index>
```

Example:

```
  view 2   ! Same as "set default uni = 2".
```

11.38 wave

The `wave` command sets what data is to be used for the wave analysis (§9). Format:

```
  wave <curve-or-data_type> {<plot_location>}
```

The `<curve-or-data-type>` argument specifies what plot `curve` or `data_type` is to be used in the analysis. Possible `<data_type>`s that can be analyzed are:

```
  orbit.x, orbit.y
  beta.a,  beta.b
  phase.a, phase.b
  eta.x, eta.y
  cbar.11, cbar.12, cbar.22      ! Analysis not possible for cbar.21
  ping_a.amp_x, ping_a.phase_x
  ping_a.sin_y, ping_a.cos_y
  ping_a.amp_sin_y, ping_a.amp_cos_y
  ping_a.amp_sin_rel_y, ping_a.amp_cos_rel_y
  ping_b.amp_y, ping_b.phase_y
  ping_b.sin_x, ping_b.cos_x
  ping_b.amp_sin_x, ping_b.amp_cos_x
  ping_b.amp_sin_rel_x, ping_b.amp_cos_rel_x
```

If there is more than one displayed curve that has the `data_type` to be analyzed, use the `curve` name instead (§7).

The `<plot_location>` argument specifies the plot region where the results of the wave analysis is to be plotted. If not present, the region defaults to the region of the plot containing the curve used for the analysis.

Note: use the `set wave` (§11.29.30) command to set the boundaries of the fit regions.

Examples:

```
  wave orbit.x      ! Use the orbit.x curve for the wave analysis.
  wave top.x bottom ! Use the curve in top.x and the results of the
                    !  wave analysis are put in the bottom region.
```

11.39 write

The `write` command creates various files. Format:

```
  write beam ...                ! §11.39.1
  write blender ...             ! §11.39.2
  write bmad ...                ! §11.39.3
  write bunch_comb ...          ! §11.39.4
```

```
write covariance_matrix ...     ! §11.39.5
write derivative_matrix ...     ! §11.39.6
write digested ...              ! §11.39.7
write elegant ...               ! §11.39.8
write field ...                 ! §11.39.9
write gif ...                   ! §11.39.10
write hard                      ! §11.39.11
write mad8 ...                  ! §11.39.12
write madx ...                  ! §11.39.13
write matrix ...                ! §11.39.14
write namelist ...              ! §11.39.15
write opal ...                  ! §11.39.16
write pals ...                  ! §11.39.17
write pdf ...                   ! §11.39.18
write plot_commands             ! §11.39.19
write ps ...                    ! §11.39.20
write ptc ...                   ! §11.39.21
write sad ...                   ! §11.39.22
write scibmad ...               ! §11.39.23
write spin_mat8                 ! §11.39.24
write variable ...              ! §11.39.25
write xsif ...                  ! §11.39.26
```

11.39.1 write beam

The `write beam` command writes beam particle information to a file. Syntax:

```
    write beam {-ascii} {-floor_position} -at <element_list> {<file_name>}
```

The `write beam` command creates a file of beam particle positions at a given lattice element(s). The `-at` switch specifies at what elements the particle positions are written. Element list format (§4.3), without any embedded blanks, is used for the `<element_list>` argument to the `-at` switch.

The default, if `-floor_position` is not present, is to write particle phase space positions. If `-floor_position` is present. The particle position in global coordinate space is written. In this case, a ASCII file is always produced.

Note: Non-floor position beam files can be used to initialize *Tao* (§10.1).

If `-floor_position` is not present, the default is to write a binary HDF5 file. See the `Beam Initialization` chapter in the *Bmad* manual for a discussion of the syntax. This default can be overridden by using the `-ascii` switch.

If `-floor_position` is present, the default file name is `beam_floor_#.dat` where `#` is replaced by the universe number. If `-floor_position` is not present, the default ASCII file name is `beam_#.dat` and the default HDF5 binary file name is `beam_#.hdf5`.

Examples:

```
  write beam -at *         ! Output beam at every element.
  write beam -floor end    ! Output beam floor coords at element named ‘‘end’’.
```

11.39.2 write blender

The `write blender` creates a script which can then be run by the `blender` program[Blender]. Syntax:

```
    write blender {<file_name>}           ! Write a blender script (Same as 3d_model).
```

The default file name is `blender_lat_#.py` where `#` is replaced by the universe number.

`Blender` is a free, open source, program for creating, among other things, 3D images. This script will create a 3D model of the lattice in the current default universe (§3.3). The suffix must by '.py' and if this suffix is not present it will be added. To run the script in `blender`, use the following on the operating system command line:

```
  <path-to-blender-exe>/blender -P <script-file-from-tao>
```

To learn how to pan, zoom, etc. in `blender`, consult any one of a number of online tutorials and videos. A good place to start is:

```
  www.blender.org/support/tutorials/
```

Note: In order of the script to work, the script must be able to find the "base" file `blender_base.py`. This base file lives in the `bmad/scripts` directory and the `bmad` directory is found using one of the following environment variables:

```
  BMAD_BASE_DIR
  DIST_BASE_DIR
  ACC_RELEASE_DIR
```

Generally, one of the latter two environment variables will be defined. If not, a copy of the *Bmad* directory must be created and then `BMAD_BASE_DIR` be appropriately defined.

11.39.3 write bmad

The `write bmad` command will create a bmad lattice file. Syntax:

```
    write bmad {-format <type>} {<file_name>}
```

The default file name is `lat_#.bmad` where `#` is replaced by the universe number.

The `-format` switch is used set how field description parameters of a lattice element are stored. The `-format` switch can be set to one of:

```
  one_file      ! One lattice file.
  ascii         ! Separate ASCII field files.
  binary        ! [Default] Separate ASCII field files with the exception that
                !   grid_field files use the HDF5 binary format.
```

Lattice elements may have associated field descriptions. There are four types as explained in the *Bmad* manual:

```
  cartesian_map
  cylindrical_map
  gen_grad_map
  grid_field
```

Since the data associated with these may be largish, there is the option of storing the data is separate secondary lattice files. This is done by setting `format` to either `ascii` or `binary`. The difference between `ascii` and `binary` is that for `grid_fields`, which may have a huge amount of associated data, the `binary` format stores `grid_fields` using HDF5. The other three field description types are always stored using ASCII files.

```
  write bmad -format one lat.bmad  ! Single lattice file lat.bmad created.
```

11.39.4 write bunch_comb

The `write bunch_comb` command writes to a file bunch parameters (bunch sigma matrix, etc.) at the "comb" points where these aggregate bunch parameters are saved (§10.7). Also see the `show beam -comb` (§11.30.2 and `pipe bunch_comb` (§13.4.5) commands. Syntax:

```
    write bunch_comb {-branch <ix_branch> } {-centroid} {-ix_bunch <ix_bunch>}
                        {-min_max} {-sigma} {-universe <ix_uni>} {file_name}
```

There are three file types described below. For all types the file will contain a table. The rows of the table correspond to different comb points. The columns of the table correspond to various bunch parameters calculated at the comb points.

-centroid
The table generated when the `-centroid` switch is present has columns showing phase space and spin centroid values averaged over the bunch. Additionally, the sigmas for the six phase space variables are given along with emittances for the three normal modes.

-min_max
The table generated when the `-min_max` switch is present has columns showing the minimum and maximum values for the six phase space coordinates of all the particles of the bunch.

-sigma
The table generated when the `-sigma` switch is present has columns giving values of the bunch beam size sigma matrix.

If no file type switch is given, `-sigma` is the default.

The `file_name` option gives the name of the output file. If not present, the default name is "`bunch_comb.XXX`" where "`.XXX` is suffix that is set dependent upon the type of output file. "`.centroid`" for the centroid table, etc.

The `-branch` switch specifies which branch to use the comb from. The default is the default branch.

The `-ix_bunch` switch specifies which bunch comb to use (if there are multiple bunches in a beam).

The `-universe` switch specifies which universe to look in for the comb. The default is the current default universe.

Examples:

```
  write bunch_c -uni 2 -cent  ! Centroid type table for universe 2
```

11.39.5 write covariance_matrix

Syntax:

```
    write covariance_matrix {file_name}    ! Write the covariance and alpha matrices
```

The default file name is `covar.matrix`.

11.39.6 write derivative_matrix

Syntax:

```
    write derivative_matrix {file_name}    ! Write the dModel_Data/dVar matrix.
```

The default file name is `derivative_matrix.dat`.

11.39.7 write digested

Syntax:
```
    write digested {<file_name>}      ! Write a digested Bmad lattice file of the model.
```
The default file name is `lat_#.digested` where `#` is replaced by the universe number.

11.39.8 write elegant

Syntax:
```
    write elegant {<file_name>}           ! Elegant lattice file using the model lattice.
```
Write a lattice file in `Elegant` format. The default file name is `lat_#.lte` where `#` is replaced by the universe number.

11.39.9 write field

The `write field` command creates a file with a table of magnetic and electric field values for a given lattice element. Syntax:
```
    write field {-nmax <nx_max>, <ny_max> <nz_max>} {-nmin <nx_min>, <ny_min> <nz_min>}
                {-rmax <rx_max>, <ry_max> <rz_max>} {-rmin <rx_min>, <ry_min> <rz_min>}
        {-dr <dr_x> <dr_y <dr_z>} -ele <eleID> {<file_name>}
```
The default file name is `field.dat`.

The fiducial point for the grid is the coordinates at the entrance end of the the lattice element given by the `-ele` switch. Laboratory (not body) coordinates are used.

The field grid indexes runs from `nx_min` to `nx_max` for x, etc. The distance between points is set with the `-dr` switch and the extent of the grid is set by `-rmin` and `rmax` switches.

If `-nmin` is present, so must `-nmax` be present. If `-nmin` is not present but `-nmax` is, the default values for `-nmin` are [-nx_max, -ny_max, 0].

If `-rmin` is present, so must `-rmax` be present. If `-rmin` is not present but `-rmax` is, the default values for `-rmin` are [-rx_max, -ry_max, 0].

`-dr`, `-nmax`, and `-rmax` values are interdependent. For example:
```
  rx_max = nx_max * dr_x
```
Given this, exactly two of the three needs to be present.

Examples:
```
  write field -ele Q1 -dr 0.01 0.02 0.05 -nmax 30 20, 200
  write field -ele Q1 -dr 0.01 0.02 0.05 -rmax 0.3 0.4 10.0   ! Same as above
```
The above examples both specify the same grid which will have an index range in x of $[-30, 30]$, and range in y of $[-20, 20]$,

11.39.10 write gif

Syntax:
```
    write gif {<file_name>}           ! Create a gif file of the plot window.
```
Write a `gif` file. The default file name is `tao.gif`.

Note: PGPLOT, if being used, does a poor job producing gif files so consider making a postscript file instead and using a ps to gif converter.

11.39.11 write hard

Syntax:

```
    write hard                        ! Print the plot window to a printer.
```

11.39.12 write mad8

Syntax:

```
    write mad8 {<file_name>}  ! Write a MAD-8 lattice file of the model
```

The default file name is `lat_#.mad8` where `#` is replaced by the universe number.

11.39.13 write madx

Syntax:

```
    write madx {<file_name>}  ! Write a MAD-X lattice file of the model
```

The default file name is `lat_#.madx` where `#` is replaced by the universe number.

11.39.14 write matrix

Syntax:

```
    write matrix {-branch <branch>} {-single} {-from_start} {-combined}
               {-universe <ix_uni>} {<file_name>}  ! Write transport matrices
```

The `write matrix` command writes transport matrices to a file for a particular lattice branch determined by the `-universe` and `-branch` switches. The default is the current viewed universe and default lattice branch.

If `<file_name>` is not present, the default file name to write to is `matrix.dat`.

What is written is determined by the `-single`, `-from_start`, and `-combined` switches. If `-single` is present, the transfer matrices through each lattice element is recorded. If `-from_start` is present, the transfer matrices from the start of the branch to each element is written. If `-combined` is present, both the element matrices and the matrices from the start of the branch are written. The default is to write the element matrices.

Examples:

```
  write mat -uni 3 -br 1  ! Write matrices from universe 3, branch 1.
  write mat -from m.dat   ! Write matrices in m.dat from branch beginning to elements
```

11.39.15 write namelist

Syntax:

```
    write namelist {-append} {-data} {-plot} {-variable} {file_name}
```

The default file name is `tao.namelist`.

11.39.16 write opal

Syntax:

```
    write opal {<file_name>}  ! Write a OPAL lattice file of the model
```

The default file name is `lat_#.opal` where `#` is replaced by the universe number.

11.39.17 write pals

Syntax:

```
  write pals {<file_name>}  ! Write a PALS lattice file of the model in YAML format.
```

The default file name is `lat_#.pals.yaml` where `#` is replaced by the universe number.

11.39.18 write pdf

The `write pdf` command produces PDF output. Syntax:

```
write pdf {-scale <scale>} {<file_name>}
```

This command is not available When using PGPLOT as the plotting backend.

The default file name is `tao.pdf`.

The optional `-scale` switch sets the scale for the postscript file. A value of 1.0 (the default) will result in no scaling, 2.0 will double the size, etc.

11.39.19 write plot_commands

The `write plot_commands` command writes all the plotting commands that have been issued since *Tao* was started. The syntax of this command is:

```
write plot_commands {<file_name>}
```

If `file_name` is not set, the default file name is `plot_commands.tao`.

Plot commands are commands that effect how plots look. For example:

```
set graph r11 component = model - design
```

Commands that effect values of the data being plot are not considered to be plot commands. For example, A command to change lattice element magnet strengths, which changes, Twiss values, is not considered a plot command.

The command file generated by the `write plot_commands` command can be read into *Tao* using the `call` command. The `write plot_commands` comand is useful for generating a file that can be used to configure the plot window. The alternative to using this command is to modify the plotting startup file (§10.13).

Example:

```
write plot p1.tao   ! Write plotting commands to p1.tao.
```

11.39.20 write ps

The `write ps` command produces postscript output. Syntax:

```
write ps {-scale <scale>} {<file_name>}
```

When using PLPLOT as the plotting backend, it is recommended to use the `write pdf` command.

The default file name is `tao.ps`.

The optional `-scale` switch sets the scale for the postscript file. A value of 1.0 (the default) will result in no scaling, 2.0 will double the size, etc.

11.39.21 write ptc

Syntax:
```
  write ptc {-all} {-old} {-branch <name_or_index} {<file_name>}
```
The default file name is `ptc.flatfile`

The `write ptc` command creates PTC lattice files (called “flat” files). If the `-all` switch is present, there will be two main flat files generated. The `-all` switch needs to be used when there are multiple lattice branches that need to be translated to PTC. For example, in a dual colliding ring machine with two storage rings. Both `M_u` and `M_t` mad_universe structures will be generated. The two main files generated will have the suffixes `.m_u` and `.m_t` appended to the file names. In this case, the setting of `-branch` is ignored.

If `-all` is not present, only one main flat file is generated. In this case, if `-old` is present, the flat file generated will be of the “old” syntax. Generally there is no reason to generate old style flat files. When generating a single flat file (no `-all` switch present), the flat file will contain the information for a single lattice branch. The lattice branch used can be specified by the `-branch` switch. The default, if `-branch` is not present, is to use default branch. The `-old` switch will generate an “old style” version.

In all cases, the `write ptc` command can only be used after a `ptc init` command has been used to setup PTC.

11.39.22 write sad

Syntax:
```
  write sad {<file_name>}  ! Write a SAD lattice file of the model
```
The default file name is `lat_#.sad` where `#` is replaced by the universe number.

11.39.23 write scibmad

Syntax:
```
  write scibmad {<file_name>}  ! Write a SciBmad lattice file of the model
```
SciBmad files have a Julia format.

The default file name is `lat_#.jl` where `#` is replaced by the universe number.

11.39.24 write spin_mat8

The `write spin_mat8` writes the 8x8 matrices spin/orbit transport matrices, element-by-element, for a given branch of the model lattice of the default universe. Syntax:
```
  write spin_mat8 -l_axis <lx> <ly> <lz> {-branch <name_or_index>} {<file_name>}
```
See the *Bmad* manual for details on how the 8x8 spin/orbit matrices are defined.

The default file name if `<file_name>` is not present is `spin_mat8.dat`.

The computation starts at the beginning of the lattice. The n_0-axis is computed by *Bmad*. The l_0-axis must be given in the `write spin_mat8` command. The m_0-axis will be computed so that (l_0, n_0, m_0) form a right handed coordinate system.

Example:
```
  write spin -l 1 0 0    ! l-axis is (1, 0, 0).
```

11.39.25 write variable

The `write variable` command writes *Tao* variable values to a file or files. Syntax:

```
  write variable {-good_opt_only} {-tao_format} {<file_name>}
```

This is useful, for example, for recording changes when a lattice is optimized.

If the `-tao_format` switch is absent, the output is a list of lines with each line of the form:

```
  slave_ele[slave_param] = value
```

where `slave_param` is a parameter of *Bmad* element `slave_ele` that is controlled by a *Tao* variable. And `value` is the value of that parameter. For example, if `quad_rot[14]` is a *Tao* variable that controls the `tilt` parameter of *Bmad* lattice element `q_arc_12`, the output will contain a line like:

```
  q_arc12[tilt] = 0.23465e-4
```

This output can be used construct a lattice with optimized values. For example, if the output file name of the `write variable` command is called `var1.out`, a three line lattice file with the optimized values can be constructed which looks like:

```
  call, file = original_lattice.bmad
  expand_lattice
  call, file = var1.out
```

where "`original_lattice.bmad`" is the name of the original unoptimized lattice file. The `expand_lattice` command may not be needed if the controlled *Bmad* elements have unique names.

If the `tao_format` switch is present, the output is a set of lines of the form:

```
  set tao_var[index]|model = value
```

where `tao_var[index]` designates a *Tao* variable which has the given `model` value. For example, if `quad_rot[14]` is a *Tao* variable, the output will contain a line like:

```
  set quad_rot[14]|model = 0.23465e-4
```

With this format, the output of `write variable` can be read back into *Tao* using a `call` command.

Note: after the *Bmad* or *Tao* compatible set lines, there will be an `end_file` command (so *Bmad* or *Tao* will ignore the rest of the file), and following this will be information on the optimization state.

When there are multiple universes, and if `-tao_format` is not present, the `write variable` command writes a number of files, one for each universe.

The default file name is `var#.out` where `#` is replaced by the universe number.

If the optional `-good_opt_only` switch is present, only the information on variables that are currently used in the optimization is written.

Example

```
  write var -good this_var.dat  ! Write to file "this_var.dat".
```

11.39.26 write xsif

Syntax:

```
    write xsif {<file_name>}  ! Write an XSIF lattice file of the model
```

The default file name is `lat_#.xsif` where `#` is replaced by the universe number. XSIF is a version of MAD-8 customized by SLAC to handle `lcavity` elements.

11.40 x_axis

The `x_axis` command sets the data type used for the x-axis coordinate. Format:

```
  x_axis <where> <axis_type>
```

The `x_axis` command sets the `plot%x_axis_type`. This determines what data is used for the horizontal axis. Possibilities for `<axis_type>` are:

```
  index     -- Use data index
  ele_index -- Use data element index
  s         -- Use longitudinal position.
```

Note that `index` only makes sense for data that has an index associated with it.

Examples:

```
  x_axis * s
  x_axis top index
```

11.41 x_scale

The `x_scale` command scales the horizontal axis of a graph or set of graphs. Format:

```
  x_scale {-exact} {-gang} {-nogang} {-include_wall} {<where> {<value1> <value2>}}
```

Which graphs are scaled is determined by the `<where>` switch. If `<where>` is not present or `<where>` is `*` then all graphs are scaled. `<where>` can be a plot name or the name of an individual graph withing a plot. If `<where>` is `s` then the scaling is done only for the plots where the x-axis scale is the longitudinal s-position.

`x_scale` sets the lower and upper bounds for the horizontal axis. If `<bound1>` and `<bound2>` are present, `<bound1>` is taken to be the lower (left) bound and `<bound2>` is the upper (right) bound. If neither is present, an `autoscale` will be invoked to give the largest bounds commensurate with the data. If an autoscale is performed upon an entire plot. In the case where there is an autoscale, if `plot%autoscale_gang_x` (§10.13.2) is True, then the chosen scales will be the same for all graphs. That is, a single scale is calculated so that all the data of all the graphs is within the plot region. The affect of `plot%autoscale_gang_x` can be overridden by using the `-gang` or `-nogang` switches.

How a graph is scaled is determined in part by the setting of the `bounds` parameter in the `x` parameter of the graph. See `s:quick.plot` for more details. The `-exact` switch, if present, will set `bounds` to `"EXACT"`. which means that *Tao* will use the min and max bounds as given by `<value1>` and `<value2>` and not try to find “nice” values near the given ones. If `<value1>` and `<value2>` are not given, and if `bounds` is set to `"EXACT"`, *Tao* will set `bounds` to `"GENERAL"`. Note: To set the axis `bounds` directly, use the `set graph` command.

Note: The `x_scale` command will vary the number of major divisions (set by `graph%x%major_divisions` (§10.13.2)) to try to give a nice looking axis. The result can be that if two plots have the same range of data but differing major division settings, the `x_scale` command can produce differing results.

For scaling `floor_plan` plots where there is a building wall to be drawn, if `-include_wall` is present and autoscaling is being done, then the plot bounds are extended to include the extent of the building wall.

Example:

```
  x_scale -include      ! Autoscale all x-axes and include the extent of any
                        !   building walls in the calculation of the plot bounds.
  x_scale * 0 100       ! Scale all x-axes to go from 0 to 100.
  x_scale orbit -10 10  ! This "wraps around" the beginning of the lattice.
  x_scale s             ! Scale all graphs using x_axis = "s".
```

11.42 xy_scale

The `xy_scale` command sets horizontal and vertical axis bounds. Format:

```
  xy_scale {-include_wall} {<where> {<bound1> <bound2>}}}
```

`xy_scale` is equivalent to an `x_scale` followed by a `y-scale`.

Which graphs are scaled is determined by the `<where>` switch. If `<where>` is not present or `<where>` is `*` then all graphs are scaled. `<where>` can be a plot name or the name of an individual graph withing a plot.

`xy_scale` sets the lower and upper bounds for both the horizontal and vertical axes. This is just a shortcut for doing an `x_scale` followed by a `scale`. If both `<bound1>` and `<bound2>` are present then `<bound1>` is taken to be the lower (left) bound and `<bound2>` is the upper (right) bound. If only `<bound1>` is present then the bounds will be from `-<bound1>` to `<bound1>`.

If neither {<bound1>} nor {<bound2>} is present then an `autoscale` will be invoked to give the largest bounds commensurate with the data.

For scaling `floor_plan` plots where there is a building wall to be drawn, if `-include_wall` is present and autoscaling is being done, then the plot bounds are extended to include the extent of the building wall.

Example:

```
  xy_scale -include ! Autoscale all axes and include the extent of any
                    !   building walls in the calculation of the plot bounds.
  xy_scale * -1 1   ! Scale all axes to go from -1 to 1.
```

Chapter 12

Single Mode

Tao has two `modes` for entering commands. In `Single Mode`, described in this chapter, each keystroke represents a command. That is, the user does not have to press the carriage control key to signal the end of a command (there are a few exceptions which are noted below). Conversely, in `Line Mode`, which is described in Chapter §11, *Tao* waits until the `return` key is depressed to execute a command. That is, in Line Mode a command consists of a single line of input. Single Mode is useful for quickly varying parameters to see how they affect a lattice but the number of commands in Single Mode is limited.

From `line mode` use the `single_mode` command (§11.31) to get into `single mode`. To go back to `line mode` type `"Z"`.

12.1 Key Bindings

The main purpose of Single Mode is to associate certain keyboard keys with certain variables so that the pressing of these keys will change their associated model value of the variable as illustrated in Figure 12.1. This is called a `key binding`. Key bindings are established in a startup file by setting the `var(i)%key_bound` and `var(i)%key_delta` parameters (see Section §10.9). After startup, associated variables with keyboard keys can be done using the `set variable` command (§11.29).

The variables are divided into banks of 10. The 0th bank uses the first ten variables that have their

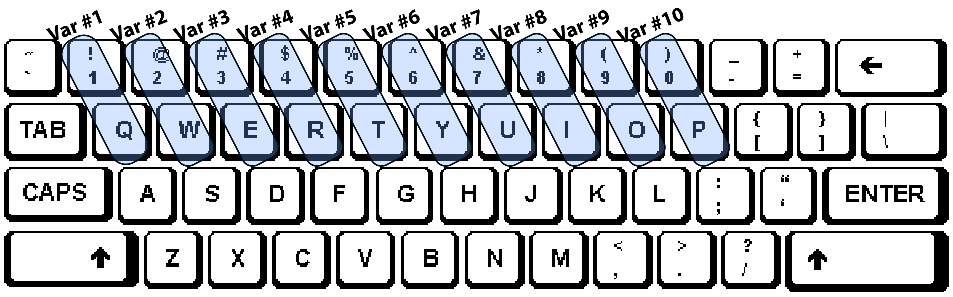
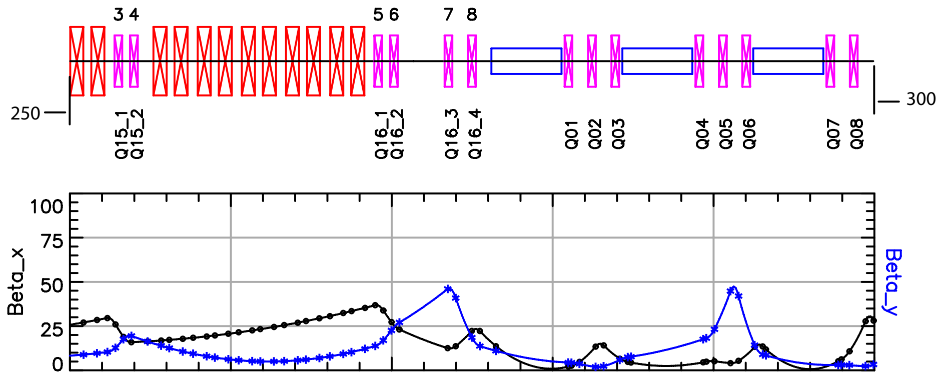


Figure 12.1: Ten pairs of keys on the keyboard are bound to ten variables so that pressing a key of a given pair will either increment or decrement the associated variable. The first key pair bound to variable number 1 are the `1` and `Q` keys, etc.



```
     ix  Name   Attrib         Value     Value0      Delta   Uni   useit_opt
1:   1  Q14_1  B1_GRADIENT     4.3195    4.3195     0.1000  All      F
     2  Q14_2  B1_GRADIENT    -4.8743   -4.8743     0.1000  All      F
     3  Q15_1  B1_GRADIENT     5.5820    5.5820     0.1000  All      F
     4  Q15_2  B1_GRADIENT    -6.6929   -6.6929     0.1000  All      F
     5  Q16_1  B1_GRADIENT     3.8954    3.8954     0.1000  All      F
     6  Q16_2  B1_GRADIENT    -2.5332   -2.5332     0.1000  All      F
     7  Q16_3  B1_GRADIENT    -6.5396   -6.5396     0.1000  All      F
     8  Q16_4  B1_GRADIENT     7.1184    7.1184     0.1000  All      F
     9
     0
```

Figure 12.2: A lattice layout plot (top) above a data plot (middle) which in turn is above a key table plot (bottom). Elements that have attributes that are varied as shown in the key table have the corresponding key table number printed above the element's glyph in the lattice layout.

`key_bound` attribute (§10.9) set to True. the 1st bank uses the next ten, etc. At any one time, only one bank is active. To see the status of this bank, a `key_table` plot (§10.13.13)can be setup as shown in Figure 12.2. The relationship between the keys and a change in a variable is:

```
                 Change by factor of:
   Variable      -10   -1    1     10
  ----------     ---   ---   ---   -------
   1 + 10*ib      Q     q     1    shift-1    ("!")
   2 + 10*ib      W     w     2    shift-2    ("@")
   3 + 10*ib      E     e     3    shift-3    ("#")
   4 + 10*ib      R     r     4    shift-4    ("$")
   5 + 10*ib      T     t     5    shift-5    ("%")
   6 + 10*ib      Y     y     6    shift-6    ("^")
   7 + 10*ib      U     u     7    shift-7    ("&")
   8 + 10*ib      I     i     8    shift-8    ("*")
   9 + 10*ib      O     o     9    shift-9    ("(")
  10 + 10*ib      P     p     0    shift-0    (")")
```

In the above table ib is the bank number (0 for the 0th bank, etc.), and the change is in multiples of the `step` (§10.9). value for a variable. Note: In `line mode`, the command `show key_bindings` (§11.30) may be used to show the entire set of bound keys.

Initially the 0th bank is active. The left arrow and right arrow are used to decrease or increase the bank number. Additionally the "<" and ">" keys can be used to change the deltas for the variables.

For example, looking at Figure 12.2, the `"1:"` in the upper left corner of the `Key Table` shows that the 1st bank is active. `key(14)` is associated with the `"4"` key and from the `Key Table` it is seen that the bound attribute is the `b1_gradient` of the element named `Q15_2`. Thus, if the `"4"` key is depressed in single mode, the value of the `b1_gradient` of element `Q15_2` will be increased by the given Delta (0.1000 in this case). Pressing the `"r"` key (which is just below the `"4"` key) will decrease the value of the `b1_gradient` by 0.1000. Using the shift key, which is shift-4 (`"$"`) will increase `b1_gradient` by 10 times the given delta (1.000 in this case) and `"R"` will decrease, by a factor of 10, the given delta.

Since element `Q15_2` is also displayed in the `Lattice Layout`, there is a `"4"` drawn above this element that reflects the fact that the element contains a bound attribute. Since, in this case, the Lattice Layout only shows part of the lattice, not all key indexes are present.

12.2 List of Key Strokes

In the following list, certain commands use multiple key strokes. For example, the `"/v"` command is invoked by first pressing the slash (`"/"`) key followed by the `"v"` key. `"a <left_arrow>"` represents pressing the `"a"` key followed by the left-arrow key.

Additionally, custom commands can be associated with any key using the `set key` command §11.29. Example:

```
  set key h = veto var *  ! This sets the "h" key to the command "veto var *"
```

? Type a short help message.

a <left_arrow> Pan plots left by half the plot width.

a <right_arrow> Pan plots right by half the plot width.

a <up_arrow> Pan plots up by half the plot height.

a <down_arrow> Pan plots down by half the plot height.

s <left_arrow> Scale x-axis of plots by a factor of 2.0.

s <right_arrow> Scale x-axis of plots by a factor of 0.5

s <up_arrow> Scale y-axis of plots by a factor of 2.0.

s <down_arrow> Scale y-axis of plots by a factor of 0.5

z <left_arrow> Zoom x-axis of plots by a factor of 2.0.

z <right_arrow> Zoom x-axis of plots by a factor of 0.5

z <up_arrow> Zoom y-axis of plots by a factor of 2.0.

z <down_arrow> Zoom y-axis of plots by a factor of 0.5

c Show constraints.

g Go run the default optimizer (§8.6). The optimizer will run until you type a '.' (a period). Periodically during the optimization the variable values will be written to files, one for each universe, whose name is `tao_opt_vars#.dat`. where `#` is the universe number.

v Show Bmad variable values in bmad lattice format. See also the `/v` command. Equivalent to `show vars -bmad` in line mode.

V Same an `v` except only variables currently enabled for optimization are shown. This is equivalent to `show vars -bmad -good` in line mode.

Z Go back to `line mode`

< Reduce the deltas (the amount that a variable is changed when you use the keys 0 through 9) of all the variables by a factor of 2.

> Increase the deltas (the amount that a variable is changed when you use the keys 0 through 9) of all the variables by a factor of 2.

<**left_arrow**> Shift the active key bank down by 1: ib -> ib - 1

<**right_arrow**> Shift the active key bank up by 1: ib -> ib + 1

/<**up_arrow**> Increase all key deltas by a factor of 10.

/<**down_arrow**> Decrease all key deltas by a factor of 10.

<**CR**> Do nothing but replot.

-p Toggle plotting. Whether to plot or not to plot is initially determined by `plot%enable`.

'<**command**> Accept a Line Mode (§11) command.

/**b** Switch the default lattice branch (§3.4).

/**e** <**Index or Name**> Prints info on a lattice element. If there are two lattices being used and only the information of an element from one particular lattice is wanted then prepend with "n@" where n is the lattice index.

/**l** Print a list of the lattice elements with Twiss parameters.

/**u** <**Universe Index**> Switch the default universe (§3.3).

/**v** Write variable values to the default output file in Bmad lattice format. The default output file name is set by `global%var_out`. See also the `V` command.

/**x** <**min**> <**max**> Set the horizontal scale min and max values for all the plots. This is the same as setting `default_graph%x%min` and `default_graph%x%max` in the *Tao* input file. If `min` and `max` are not given then the scale will be chosen to include the entire lattice.

/**y** <**min**> <**max**> Set the y-axis min and max values for all the plots. This is the same as setting `plot%y%min` and `plot%y%max` in the *Tao* input file. If `min` and `max` are not given then an autoscale will be done.

=**v** <**digit**> <**value**> Set variable value. `<digit>` is between 0 and 9 corresponding to a variable of the current bank. `<value>` is the value to set the variable to.

=<**right_arrow**> Set saved ("value0") values to variable values to saved values. The saved values (the value0 column in the display) are initially set to the initial value on startup. There are saved values for both the manual and automatic variables. Note that reading in a TOAD input file will reset the saved values. If you want to save the values of the variables in this case use "/w" to save to a file. Use the "`/<left_arrow>`" command to go in the reverse direction.

=<**left_arrow**> Paste saved (`value0` column in the display) values back to the variable values. The saved values are initially set to the initial value on startup. Use the "`/<right_arrow>`" command to go in the reverse direction.

Chapter 13

Python Interface to Tao

It is sometimes convenient to interface *Tao* to a scripting language like Python or interface to some external program. Applications include analyzing Tao generated data or to interface *Tao* to an online control system environment.

To aid in interfacing, *Tao* has the `pipe` command (§13.2).[1] The `pipe` command defaines a standardized syntax with which to communicate with *Tao*.

Another aid is the `PyTao` package which is an interface layer to be used between *Tao* and `Python`. See §13.1 for more details.

13.1 PyTao Interface

The `PyTao` package is an interface layer to be used between *Tao* and `Python`. `PyTao` is hosted on `GitHub` (independent of *Bmad* distributions) at:

`https://bmad-sim.github.io/pytao`

Documentation for setup and using PyTao is at:

`bmad-sim.github.io/pytao/`

See the `PyTao` documentation for installation instructions, examples, etc. In this chapter, some simple examples will be given.

The `PyTao` package uses *Tao*'s `pipe` command to ease integration with `Python`.

There are two ways to interface with Python/PyTao. One way is using the Python `ctypes` library. The other way is using the `pexpect` module. A Web search will point to documentation on `ctypes` and `pexpect`.

`ctypes` is a foreign function library for Python which can be used to link to a *Tao* shared library. The `pexpect` module is a general purpose tool for interfacing Python with programs like *Tao*. If `pexpect` is not present your system, it can be downloaded from `www.noah.org/wiki/pexpect`.

The advantage of `ctypes` is that it directly accesses *Tao* code which makes communication between Python and *Tao* more robust. The disadvantage of `ctypes` is that it needs a shared-object version

[1] Formally this command was called `python` but the name was changed to avoid confusion with the scripting language Python.

of the `Tao` library. [See the Bmad web site for information on building shared-object libraries.] The disadvantage of `pexpect` is that it is slower and it is possible for `pexpect` to time out waiting for a response from *Tao*.

13.1.1 Python/PyTao Via Pexpect

For communicaiton via `pexpect` (§13.1), the python module `tao_pipe.py`, is provided by `PyTao` in the directory `pytao/tao_pexpect`.

Example Python session:

```
>>> from pytao.tao_pexpect import tao_pipe  # import module
>>> p = tao_pipe.tao_io("-lat my_lat.bmad") # init session
>>> out = p.cmd_in("show global")           # Command to Tao
>>> print(out)                              # print the output from Tao
>>> p.cmd("show global")                    # Like p.cmd_in() excepts prints the output too.
```

13.1.2 Python/PyTao Interface Via Ctypes

A `ctypes` based Python module `pytao.py` for interfacing *Tao* to `Python` is provided by `PyTao` (§13.1) in the directory `pytao/tao_pexpect`.

A test driver script named `pytao_example.py` is in the same directory. See the documentation in both of these files for further information.

13.2 Tao's Pipe Command

Tao's `pipe` (§11.18) command was developed to:

- Standardize output of information (data, parameters, etc.) from *Tao* to simplify the task of interfacing *Tao* to external programs especially scripting languages like `Python`.
- Act as an intermediate layer for the control of *Tao* by such things as machine online control programs or the planned graphical user interface for *Tao*.

Using the `pipe` command to control *Tao* will not be covered here. The interested reader is invited to read the sections of this manual on the coding of *Tao* and look at the *Tao* code itself (which is heavily documented).

Using the `pipe` command is far superior to using the `show` command when interfacing to an external program. For one, the `pipe` command is formatted for ease of parsing. Another reason to use the `pipe` command is that, as *Tao* is developed over time, the output format of the `pipe` command is much more stable than output from the `show` command.[2] Thus the risk of User developed interface code breaking is much reduced by using the `pipe` command.

The general form of the `pipe` command is:

```
pipe <subcommand> <arguments>
```

[2]The output of the `pipe` command will change when *Tao*'s or *Bmad*'s internal structures are modified. This is in contrast to the `show` command whose output is formated to be human readible and whose output format may change on a whim.

The `pipe` command has a number of `subcommands` (over 100) that are listed in Sec. §13.4. The subcommands can be divided into two categories. One category are the "`action`" subcommands which allow the user to control *Tao* (for example, creating variables and data for use in an optimization). The other category are the "`output`" subcommands which output information from *Tao*.

The output of the `pipe` command are semi-colon delimited lists. Example: With the `pipe global` the output looks like:

```
lm_opt_deriv_reinit;REAL;T; -1.0000000000000000E+00
de_lm_step_ratio;REAL;T;  1.0000000000000000E+00
de_var_to_population_factor;REAL;T;  5.0000000000000000E+00
unstable_penalty;REAL;T;  1.0000000474974513E-03
n_opti_cycles;INT;T;20
track_type;ENUM;T;single
derivative_uses_design;LOGIC;T;F
... etc ...
```

Most `output` subcommands use "`parameter list form`" format where each line has four fields separated by semicolons:

```
name;type;variable;value(s)
```

The fields are:

```
  name:       The name of the parameter

  type:       The type of the parameter:
      INT           Integer number
      REAL          Real number
      COMPLEX       Complex number. A complex number is output as Re;Im
      REAL_ARR      Real array
      LOGIC         Logical: "T" or "F".
      INUM          Integer whose allowed values can be obtained
                      using the "pipe inum" command.
      ENUM          String whose allowed values can be obtained
                      using the "pipe enum" command.
      FILE          Name of file.
      CRYSTAL       Crystal name string. EG: "Si(111)"
      DAT_TYPE      Data type string. EG: "orbit.x"
      DAT_TYPE_Z    Data type string if plot%x_axis_type = 'data'.
                      Otherwise is a data_type_z enum.
      SPECIES       Species name string. EG: "H2SO4++"
      ELE_PARAM     Lattice element parameter string. EG "K1"
      STR           String that does not fall into one of the above string categories.
      STRUCT        Structure. In this case component_value(s) is of the form:
                      name1;type1;value1;name2;type2;value2;...
      COMPONENT     For curve component parameters.

  can_vary:   Either 'T', 'F', or 'I', indicating whether or not the
              user may change the value of the parameter. 'I' indicates
              that the parameter is to be ignored by a GUI when displaying parameters.

  value(s):   The value or values of the the parameter. If a parameter has multiple
              values (EG an array), the values will be separated by semicolons.
```

13.3 Plotting Issues

When using *Tao* with a `GUI`, and when the `GUI` is doing the plotting, the `-noplot` and `-external_plotting` options (§10.1) should be used when starting *Tao*. The `-noplot` option (which sets `global%plot_on`) prevents *Tao* from opening a plotting window. Note: Both of these options can also be set, after startup, with the `set global` command and the setting of both can be viewed using the `show global` command.

With `-external_plotting` set, the external code should handle how plots are assigned to plot regions and it would be potentially disruptive if a user tired to place plots (which could inadvertently happen when running command files). To avoid this, with `-external_plotting` set, the `place` command will not do any placement but rather save the `place` arguments (which is the name of a template plot and a region name) to a buffer which then can be read out by the external code using the `pipe place_buffer` command. The external code may then decide how to proceed. The external code is able to bypass the buffering and perform placements by using `place` with the `-no_buffer` switch (§11.19). Notice: *Tao* never processes place command information put in the buffer. It is up to the external code to decide on a course of action.

Normally when *Tao* is not displaying the plot page when the `-noplot` option is used, *Tao* will, to save time, not calculate the points needed for plotting curves. The exception is if `-external_plotting` is turned on. In this case, to make plot references unambiguous, a plot can be referred to by its index number. The plot index number can be viewed using the `pipe plot_list` command. Template plots can be referenced using the syntax "`@Tnnn`" where `nnn` is the index number. For example, `@T3` referrers to the template plot with index 3. Similarly, the displayed plots (plots that are associated with plot regions) can be referred to using the syntax "`@Rnnn`".

13.4 Pipe subcommands

The `pipe` command has the following subcommands:

13.4.1 pipe beam

Output beam parameters that are not in the beam_init structure.

```
   pipe beam {ix_uni}@{ix_branch}
```

```
Where:
  {ix_uni} is a universe index. Defaults to s%global%default_universe.
  {ix_branch} is a lattice branch index. Defaults to s%global%default_branch.

Note: To set beam_init parameters use the "set beam" command.
```

13.4.2 pipe beam_init

Output beam_init parameters.

```
   pipe beam_init {ix_uni}@{ix_branch}
```

```
Where:
  {ix_uni} is a universe index. Defaults to s%global%default_universe.
  {ix_branch} is a lattice branch index. Defaults to s%global%default_branch.

Note: To set beam_init parameters use the "set beam_init" command
```

13.4.3 pipe bmad_com

Output bmad_com structure components.

```
pipe bmad_com
```

13.4.4 pipe branch1

Output lattice branch information for a particular lattice branch.

```
pipe branch1 {ix_uni}@{ix_branch}
```

```
Where:
  {ix_uni} is a universe index. Defaults to s%global%default_universe.
  {ix_branch} is a lattice branch index. Defaults to s%global%default_branch.
```

13.4.5 pipe bunch_comb

Outputs bunch parameters at a comb point. Also see the "write bunch_comb" and "show bunch -comb" commands.

```
pipe bunch_comb {flags} {who} {ix_uni}@{ix_branch} {ix_bunch}
```

```
Where:
  {flags} are optional switches:
      -array_out : If present, the output will be available in the
             tao_c_interface_com%c_real array.
  {ix_uni} is a universe index. Defaults to s%global%default_universe.
  {ix_branch} is a branch index. Defaults to s%global%default_branch.
  {ix_bunch} is the bunch index. Defaults to 1.
  {who} is one of:
      x, px, y, py, z, pz, t, s, spin.x, spin.y, spin.z, p0c, beta     -- centroid
      x.Q, y.Q, z.Q, a.Q, b.Q, c.Q where Q is one of: beta, alpha, gamma, phi,
                                      eta, etap, sigma, sigma_p, emit, norm_emit
      t.sigma
    sigma.IJ where I, J in range [1,6]
    rel_min.I, rel_max.I where I in range [1,6]
    charge_live, n_particle_live, n_particle_lost_in_ele, ix_ele

  Note: If ix_uni or ix_branch is present, "@" must be present.

Example:
  pipe bunch_comb py 2@1 1
```

13.4.6 pipe bunch_params

Outputs bunch parameters at the exit end of a given lattice element.

```
   pipe bunch_params {ele_id}|{which}

Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"

Example:
  pipe bunch_params end|model  ! parameters at model lattice element named "end".
```

13.4.7 pipe bunch1

Outputs Bunch parameters at the exit end of a given lattice element.

```
   pipe bunch1 {ele_id}|{which} {ix_bunch} {coordinate}

Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"
  {ix_bunch} is the bunch index.
  {coordinate} is one of: x, px, y, py, z, pz, "s", "t", "charge", "p0c",
                                                                "state", "ix_ele"

For example, if {coordinate} = "px", the phase space px coordinate of each particle
of the bunch is displayed. The "state" of a particle is an integer.
A value of 1 means alive and any other value means the particle has been lost.
```

13.4.8 pipe building_wall_list

Output List of building wall sections or section points

```
   pipe building_wall_list {ix_section}

Where:
  {ix_section} is a building wall section index.

If {ix_section} is not present, a list of building wall sections is given.
If {ix_section} is present, a list of section points is given.
```

13.4.9 pipe building_wall_graph

Output (x, y) points for drawing the building wall for a particular graph.

```
   pipe building_wall_graph {graph}

Where:
  {graph} is a plot region graph name.

Note: The graph defines the coordinate system for the (x, y) points.
```

13.4.10 pipe building_wall_point

add or delete a building wall point

```
   pipe building_wall_point {ix_section}^^{ix_point}^^{z}^^{x}^^{radius}^^
                                                            {z_center}^^{x_center}


Where:
  {ix_section}    -- Section index.
  {ix_point}      -- Point index. Points of higher indexes will be moved up
                       if adding a point and down if deleting.
  {z}, etc...     -- See tao_building_wall_point_struct components.
                  -- If {z} is set to "delete" then delete the point.
```

13.4.11 pipe building_wall_section

Add or delete a building wall section

```
   pipe building_wall_section {ix_section}^^{sec_name}^^{sec_constraint}


Where:
  {ix_section}      -- Section index. Sections with higher indexes will be
                         moved up if adding a section and down if deleting.
  {sec_name}        -- Section name.
  {sec_constraint}  -- A section constraint name or "delete". Must be one of:
      delete          -- Delete section. Anything else will add the section.
      none
      left_side
      right_side
```

13.4.12 pipe constraints

Output optimization data and variable parameters that contribute to the merit function.

```
   pipe constraints {who}


Where:
  {who} is one of: "data" or "var"

Data constraints output is:
  data name
  constraint type
  evaluation element name
  start element name
  end/reference element name
  measured value
  ref value (only relavent if global%opt_with_ref = T)
  model value
  base value (only relavent if global%opt_with_base = T)
  weight
  merit value
  location where merit is evaluated (if there is a range)
Var constraints output is:
  var name
  Associated varible attribute
  meas value
  ref value (only relavent if global%opt_with_ref = T)
  model value
  base value (only relavent if global%opt_with_base = T)
  weight
  merit value
  dmerit/dvar
```

13.4.13 pipe da_aperture

Output dynamic aperture data

```
pipe da_aperture {ix_uni}
```

```
Where:
  {ix_uni} is a universe index. Defaults to s%global%default_universe.
```

13.4.14 pipe da_params

Output dynamic aperture input parameters

```
pipe da_params {ix_uni}
```

```
Where:
  {ix_uni} is a universe index. Defaults to s%global%default_universe.
```

13.4.15 pipe data

Output Individual datum parameters.

```
pipe data {ix_uni}@{d2_name}.{d1_name}[{dat_index}]
```

```
Where:
  {ix_uni} is a universe index. Defaults to s%global%default_universe.
  {d2_name} is the name of the d2_data structure the datum is in.
  {d1_datum} is the name of the d1_data structure the datum is in.
  {dat_index} is the index of the datum.

Use the "pipe data-d1" command to get detailed info on a specific d1 array.

Example:
  pipe data 1@orbit.x[10]
```

13.4.16 pipe data_d_array

Output list of datums for a given d1_data structure.

```
   pipe data_d_array {ix_uni}@{d2_name}.{d1_name}

Where:
  {ix_uni} is a universe index. Defaults to s%global%default_universe.
  {d2_name} is the name of the containing d2_data structure.
  {d1_name} is the name of the d1_data structure containing the array of datums.

Example:
  pipe data_d_array 1@orbit.x
```

13.4.17 pipe data_d1_array

Output list of d1 arrays for a given data_d2.

```
   pipe data_d1_array {d2_datum}

{d2_datum} should be of the form
  {ix_uni}@{d2_datum_name}
```

13.4.18 pipe data_d2

Output information on a d2_datum.

```
   pipe data_d2 {ix_uni}@{d2_name}

Where:
  {ix_uni} is a universe index. Defaults to s%global%default_universe.
  {d2_name} is the name of the d2_data structure.
```

13.4.19 pipe data_d2_array

Output data d2 info for a given universe.

```
   pipe data_d2_array {ix_uni}

Where:
  {ix_uni} is a universe index. Defaults to s%global%default_universe.

Example:
  pipe data_d2_array 1
```

13.4.20 pipe data_d2_create

Create a d2 data structure along with associated d1 and data arrays.

```
   pipe data_d2_create {ix_uni}@{d2_name}^^{n_d1_data}^^{d_data_arrays_name_min_max}
```

```
Where:
  {ix_uni} is a universe index. Defaults to s%global%default_universe.
  {d2_name} is the name of the d2_data structure to create.
  {n_d1_data} is the number of associated d1 data structures.
  {d_data_arrays_name_min_max} has the form
    {name1}^^{lower_bound1}^^{upper_bound1}^^....
                                           ^^{nameN}^^{lower_boundN}^^{upper_boundN}
  where {name} is the data array name and
  {lower_bound} and {upper_bound} are the bounds of the array.

Example:
  pipe data_d2_create 2@orbit^^2^^x^^0^^45^^y^^1^^47
This example creates a d2 data structure called "orbit" with
two d1 structures called "x" and "y".

The "x" d1 structure has an associated data array with indexes in the range [0, 45].
The "y" d1 structure has an associated data arrray with indexes in the range [1, 47].

Use the "set data" command to set created datum parameters.

Note: When setting multiple data parameters,
      temporarily toggle s%global%lattice_calc_on to False
  ("set global lattice_calc_on = F") to prevent Tao trying to
      evaluate the partially created datum and generating unwanted error messages.
```

13.4.21 pipe data_d2_destroy

Destroy a d2 data structure along with associated d1 and data arrays.

```
pipe data_d2_destroy {ix_uni}@{d2_name}
```

```
Where:
  {ix_uni} is a universe index. Defaults to s%global%default_universe.
  {d2_name} is the name of the d2_data structure to destroy.

Example:
  pipe data_d2_destroy 2@orbit
This destroys the orbit d2_data structure in universe 2.
```

13.4.22 pipe data_parameter

Output an array of values for a particular datum parameter for a given array of datums,

```
pipe data_parameter {data_array} {parameter}
```

```
{parameter} may be any tao_data_struct parameter.
Example:
  pipe data_parameter orbit.x model_value
```

13.4.23 pipe data_set_design_value

Set the design (and base) values of all datums to the values as calculated for the design lattice. This is useful for newly created datums that do not yet have the design and base values set.

```
   pipe data_set_design_value


Example:
  pipe data_set_design_value

Note: Use the "data_d2_create" and "datum_create" first to create datums.
```

13.4.24 pipe datum_create

Create a datum.

```
   pipe datum_create {datum_name}^^{data_type}^^{ele_start_name}^^{ele_ref_name}^^
                     {ele_name}^^{merit_type}^^{meas}^^{good_meas}^^{ref}^^
                     {good_ref}^^{weight}^^{good_user}^^{data_source}^^
                     {eval_point}^^{s_offset}^^{ix_bunch}^^{invalid_value}^^
                     {spin_axis%n0(1)}^^{spin_axis%n0(2)}^^{spin_axis%n0(3)}^^
                     {spin_axis%l(1)}^^{spin_axis%l(2)}^^{spin_axis%l(3)}


Note: The 3 values for spin_axis%n0, as a group, are optional.
      Also the 3 values for spin_axis%l are, as a group, optional.
Note: Use the "pipe data_d2_create" command first to create a d2 structure
      with associated d1 arrays.
Note: After creating all your datums, use the "pipe data_set_design_value" routine
      to set the design (and model) values.
```

13.4.25 pipe datum_has_ele

Output whether a datum type has an associated lattice element

```
   pipe datum_has_ele {datum_type}
```

13.4.26 pipe derivative

Output optimization derivatives

```
   pipe derivative


Note: If a universe is off, there will be no derivatives for this universe.
```

13.4.27 pipe ele:ac_kicker

Output element ac_kicker parameters

```
   pipe ele:ac_kicker {ele_id}|{which}


Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"

Example:
  pipe ele:ac_kicker 3@1>>7|model
This gives element number 7 in branch 1 of universe 3.
```

13.4.28 pipe ele:cartesian_map

Output element cartesian_map parameters

```
   pipe ele:cartesian_map {ele_id}|{which} {index} {who}


Where:
  {ele_id} is an element name or index
  {which} is one of: "model", "base" or "design"
  {index} is the index number in the ele%cartesian_map(:) array
  {who} is one of: "base", or "terms"

Example:
  pipe ele:cartesian_map 3@1>>7|model 2 base
This gives element number 7 in branch 1 of universe 3.
```

13.4.29 pipe ele:chamber_wall

Output element beam chamber wall parameters

```
   pipe ele:chamber_wall {ele_id}|{which} {index} {who}


Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"
  {index} is index of the wall.
  {who} is one of:
    "x"       ! Return min/max in horizontal plane
    "y"       ! Return min/max in vertical plane
```

13.4.30 pipe ele:control_var

Output list of element control variables. Used for group, overlay and ramper type elements.

```
   pipe ele:control_var {ele_id}|{which}
```

```
Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"

Example:
  pipe ele:control_var 3@1>>7|model
This gives control info on element number 7 in branch 1 of universe 3.
```

13.4.31 pipe ele:cylindrical_map

Output element cylindrical_map

```
   pipe ele:cylindrical_map {ele_id}|{which} {index} {who}
```

```
Where
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"
  {index} is the index number in the ele%cylindrical_map(:) array
  {who} is one of: "base", or "terms"

Example:
  pipe ele:cylindrical_map 3@1>>7|model 2 base
This gives map #2 of element number 7 in branch 1 of universe 3.
```

13.4.32 pipe ele:elec_multipoles

Output element electric multipoles

```
   pipe ele:elec_multipoles {ele_id}|{which}
```

```
Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"

Example:
  pipe ele:elec_multipoles 3@1>>7|model
This gives element number 7 in branch 1 of universe 3.
```

13.4.33 pipe ele:floor

Output element floor coordinates. The output gives four lines. "Reference" is without element misalignments and "Actual" is with misalignments. The lines with "-W" give the W matrix. The exception is that if ele is a multipass_lord, there will be 4*N lines where N is the number of slaves.

```
   pipe ele:floor {ele_id}|{which} {where}
```

```
Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"
  {where} is an optional argument which, if present, is one of
```

```
    beginning  ! Upstream end.
    center     ! Middle of the element. This is the surface of element when used
               !  with photonic reflecting elements such as crystal and mirror elements.
    end        ! Downstream end (default).

Example:
  pipe ele:floor 3@1>>7|model
This gives element number 7 in branch 1 of universe 3.
```

13.4.34 pipe ele:gen_attribs

Output element general attributes

```
   pipe ele:gen_attribs {ele_id}|{which}

Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"

Example:
  pipe ele:gen_attribs 3@1>>7|model
This gives element number 7 in branch 1 of universe 3.
```

13.4.35 pipe ele:gen_grad_map

Output element gen_grad_map

```
   pipe ele:gen_grad_map {ele_id}|{which} {index} {who}

Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"
  {index} is the index number in the ele%gen_grad_map(:) array
  {who} is one of: "base", or "derivs".

Example:
  pipe ele:gen_grad_map 3@1>>7|model 2 base
This gives element number 7 in branch 1 of universe 3.
```

13.4.36 pipe ele:grid_field

Output element grid_field

```
   pipe ele:grid_field {ele_id}|{which} {index} {who}

Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"
  {index} is the index number in the ele%grid_field(:) array.
  {who} is one of: "base", or "points"
```

```
Example:
  pipe ele:grid_field 3@1>>7|model 2 base
This gives grid #2 of element number 7 in branch 1 of universe 3.
```

13.4.37 pipe ele:head

Output "head" Element attributes

```
   pipe ele:head {ele_id}|{which}

Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"

Example:
  pipe ele:head 3@1>>7|model
This gives element number 7 in branch 1 of universe 3.

Output: Various element parameters. See the code for the list of what is returned.

Note: For super_slave elements, the "type", "descrip", and "alias" parameters of the element are never s
If the element used in this command is a super_slave with exactly one super_lord, what will be returned
for these three parameters is the values stored in the super_lord. This is done for convenience sake.
```

13.4.38 pipe ele:lord_slave

Output the lord/slave tree of an element.

```
   pipe ele:lord_slave {ele_id}

Where:
  {ele_id} is an element name or index.

Example:
  pipe ele:lord_slave 3@1>>7
This gives lord and slave info on element number 7 in branch 1 of universe 3.

The output is a number of lines.
Each line gives information on an element (element index, etc.).
Some lines begin with the word "Element".
After each "Element" line, there are a number of lines (possibly zero)
that begin with the word "Slave or "Lord".
These "Slave" and "Lord" lines are the slaves and lords of the "Element" element.
```

13.4.39 pipe ele:mat6

Output element mat6

```
   pipe ele:mat6 {ele_id}|{which} {who}
```

```
Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"
  {who} is one of: "mat6", "vec0", or "err"

Example:
  pipe ele:mat6 3@1>>7|model mat6
This gives element number 7 in branch 1 of universe 3.
```

13.4.40 pipe ele:methods

Output element methods

```
pipe ele:methods {ele_id}|{which}
```

```
Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"

Example:
  pipe ele:methods 3@1>>7|model
This gives element number 7 in branch 1 of universe 3.
```

13.4.41 pipe ele:multipoles

Output element multipoles

```
pipe ele:multipoles {ele_id}|{which}
```

```
Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"

Example:
  pipe ele:multipoles 3@1>>7|model
This gives element number 7 in branch 1 of universe 3.
```

13.4.42 pipe ele:orbit

Output element orbit

```
pipe ele:orbit {ele_id}|{which}
```

```
Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"

Example:
  pipe ele:orbit 3@1>>7|model
This gives element number 7 in branch 1 of universe 3.
```

13.4.43 pipe ele:param

Output lattice element parameter

```
   pipe ele:param {ele_id}|{which} {who}


Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"
  {who} values are the same as {who} values for "pipe lat_list".
        Note: Here {who} must be a single parameter and not a list.

Example:
  pipe ele:param 3@1>>7|model e_tot
This gives E_tot of element number 7 in branch 1 of universe 3.

Note: On output the {variable} component will always be "F" (since this
command cannot tell if a parameter is allowed to vary).

Also see: "pipe lat_list".
```

13.4.44 pipe ele:photon

Output element photon parameters

```
   pipe ele:photon {ele_id}|{which} {who}


Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"
  {who} is one of: "base", "material", or "curvature"

Example:
  pipe ele:photon 3@1>>7|model base
This gives element number 7 in branch 1 of universe 3.
```

13.4.45 pipe ele:spin_taylor

Output element spin_taylor parameters

```
   pipe ele:spin_taylor {ele_id}|{which}


Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"

Example:
  pipe ele:spin_taylor 3@1>>7|model
This gives element number 7 in branch 1 of universe 3.
```

13.4.46 pipe ele:taylor

Output element taylor map

```
   pipe ele:taylor {ele_id}|{which}

Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"

Example:
  pipe ele:taylor 3@1>>7|model
This gives element number 7 in branch 1 of universe 3.
```

13.4.47 pipe ele:twiss

Output element Twiss parameters

```
   pipe ele:twiss {ele_id}|{which}

Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"

Example:
  pipe ele:twiss 3@1>>7|model
This gives element number 7 in branch 1 of universe 3.
```

13.4.48 pipe ele:wake

Output element wake.

```
   pipe ele:wake {ele_id}|{which} {who}

Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"
  {Who} is one of:
      "sr_long"        "sr_long_table"
      "sr_trans"       "sr_trans_table"
      "lr_mode_table"  "base"

Example:
  pipe ele:wake 3@1>>7|model
This gives element number 7 in branch 1 of universe 3.
```

13.4.49 pipe ele:wall3d

Output element wall3d parameters.

```
   pipe ele:wall3d {ele_id}|{which} {index} {who}
```

```
Where:
  {ele_id} is an element name or index.
  {which} is one of: "model", "base" or "design"
  {index} is the index number in the ele%wall3d(:) array
            The size of this array is obtained from "pipe ele:head".
  {who} is one of: "base", or "table".
Example:
  pipe ele:wall3d 3@1>>7|model 2 base
This gives element number 7 in branch 1 of universe 3.
```

13.4.50 pipe evaluate

Output the value of an expression. The result may be a vector.

```
   pipe evaluate {flags} {expression}
```

```
Where:
  Optional {flags} are:
      -array_out : If present, the output will be available in the
                   tao_c_interface_com%c_real array.
  {expression} is expression to be evaluated.

Example:
  pipe evaluate 3+data::cbar.11[1:10]|model
```

13.4.51 pipe em_field

Output EM field at a given point generated by a given element.

```
   pipe em_field {ele_id}|{which} {x} {y} {z} {t_or_z}
```

```
Where:
  {which} is one of: "model", "base" or "design"
  {x}, {y}  -- Transverse coords.
  {z}       -- Longitudinal coord with respect to entrance end of element.
  {t_or_z}  -- time or phase space z depending if lattice is setup for
            --   absolute time tracking.
```

13.4.52 pipe enum

Output list of possible values for enumerated numbers.

```
   pipe enum {enum_name}
```

```
Example:
  pipe enum tracking_method
```

13.4.53 pipe floor_plan

Output (x,y) points and other information that can be used for drawing a floor_plan.

```
   pipe floor_plan {graph}
```

13.4.54 pipe floor_orbit

Output (x, y) coordinates for drawing the particle orbit on a floor plan.

```
pipe floor_orbit {graph}
```

13.4.55 pipe global

Output global parameters.

```
pipe global
```

```
Output syntax is parameter list form. See documentation at the beginning of this file.

Note: The follow is intentionally left out:
  optimizer_allow_user_abort
  quiet
  single_step
  prompt_color
  prompt_string
```

13.4.56 pipe global:optimization

Output optimization parameters. Also see global:opti_de.

```
pipe global:optimization
```

```
Output syntax is parameter list form. See documentation at the beginning of this file.
```

13.4.57 pipe global:opti_de

Output DE optimization parameters.

```
pipe global:opti_de
```

```
Output syntax is parameter list form. See documentation at the beginning of this file.
```

13.4.58 pipe help

Output list of "help xxx" topics

```
pipe help
```

13.4.59 pipe inum

Output list of possible values for an INUM parameter. For example, possible index numbers for the branches of a lattice.

```
   pipe inum {who}
```

13.4.60 pipe lat_calc_done

Output if a lattice recalculation has been proformed since the last time "pipe lat_calc_done" was called.

```
   pipe lat_calc_done
```

13.4.61 pipe lat_ele_list

Output lattice element list.

```
   pipe lat_ele_list {branch_name}
```

```
{branch_name} should have the form:
  {ix_uni}@{ix_branch}
```

13.4.62 pipe lat_header

Output lattice "header" info like the lattice and machine names.

```
   pipe lat_header {ix_uni}
```

```
Output syntax is parameter list form. See documentation at the beginning of this file.
```

13.4.63 pipe lat_branch_list

Output lattice branch list

```
   pipe lat_branch_list {ix_uni}
```

```
Output syntax:
  branch_index;branch_name;n_ele_track;n_ele_max
```

13.4.64 pipe lat_list

Output list of parameters at ends of lattice elements

```
   pipe lat_list {flags} {ix_uni}@{ix_branch}>>{elements}|{which} {who}



Where:
 Optional {flags} are:
  -no_slaves   - If present, multipass_slave and super_slave elements will not
               -   be matched to.
  -track_only  - If present, lord elements will not be matched to.
  -index_order - If present, order elements by element index instead of the
               -   standard s-position.
  -array_out   - If present, the output will be available in the
    tao_c_interface_com%c_real or tao_c_interface_com%c_integer arrays.
    See the code below for when %c_real vs %c_integer is used.
    Note: Only a single {who} item permitted when -array_out is present.

  {which} is one of: "model", "base" or "design"

  {who} is a comma deliminated list of:
    orbit.floor.x, orbit.floor.y, orbit.floor.z    ! Floor coords at particle orbit.
    orbit.spin.1, orbit.spin.2, orbit.spin.3,
    orbit.vec.1, orbit.vec.2, orbit.vec.3, orbit.vec.4, orbit.vec.5, orbit.vec.6,
    orbit.t, orbit.beta,
    orbit.state,     ! Note: state is an integer. alive$ = 1, anything else is lost.
    orbit.energy, orbit.pc,
    ele.name, ele.key, ele.ix_ele, ele.ix_branch
    ele.a.beta, ele.a.alpha, ele.a.eta, ele.a.etap, ele.a.gamma, ele.a.phi,
    ele.b.beta, ele.b.alpha, ele.b.eta, ele.b.etap, ele.b.gamma, ele.b.phi,
    ele.x.eta, ele.x.etap,
    ele.y.eta, ele.y.etap,
    ele.ref_time, ele.ref_time_start
    ele.s, ele.l
    ele.e_tot, ele.p0c
    ele.mat6      ! Output: mat6(1,:), mat6(2,:), ... mat6(6,:)
    ele.vec0      ! Output: vec0(1), ... vec0(6)
    ele.c_mat     ! Output: c_mat11, c_mat12, c_mat21, c_mat22.
    ele.gamma_c   ! Parameter associated with coupling c-matrix.
    ele.XXX       ! Where XXX is a Bmad syntax element attribute.
                  !   EG: ele.beta_a, ele.k1, etc.

  {elements} is a string to match element names to.
    Use "*" to match to all elements.

Examples:
  pipe lat_list -track 3@0>>Q*|base ele.s,orbit.vec.2
  pipe lat_list 3@0>>Q*|base real:ele.s

Also see: "pipe ele:param"
```

13.4.65 pipe lat_param_units

Output units of a parameter associated with a lattice or lattice element.

```
   pipe lat_param_units {param_name}
```

13.4.66 pipe lord_control

Output lord information for a given slave element.

```
   pipe lord_control {ele_id}
```

```
Where:
  {ele_id} is the slave element.

Example:
  pipe lord_control 2@1>>q01w

The output is a number of lines with each line giving the information:
    Lord-index;Lord-name;Lord-type;Attribute-controlled;Control-expression;Value

Note: The last three fields will only be non-blank for ramper, overlay and group lords with
the value field being blank for rampers.

Note: For control expressed as a set of knot points (as opposed to an expression),
the control-expression will be "knots".

Note: The Value field is the contribution to the slave attribute value due to the lord.
```

13.4.67 pipe matrix

Output matrix value from the exit end of one element to the exit end of the other.

```
   pipe matrix {ele1_id} {ele2_id}
```

```
Where:
  {ele1_id} is the start element.
  {ele2_id} is the end element.
If {ele2_id} = {ele1_id}, the 1-turn transfer map is computed.
Note: {ele2_id} should just be an element name or index without universe,
      branch, or model/base/design specification.

Example:
  pipe matrix 2@1>>q01w|design q02w
```

13.4.68 pipe merit

Output merit value.

```
   pipe merit
```

13.4.69 pipe orbit_at_s

Output twiss at given s position.

```
   pipe orbit_at_s {ix_uni}@{ele}->{s_offset}|{which}


Where:
  {ix_uni}   - Universe index. Defaults to s%global%default_universe.
  {ele}      - Element name or index.
                 Default at the Beginning element at start of branch 0.
  {s_offset} - Offset of the evaluation point from the downstream end of ele.
                 Default is 0. If {s_offset} is present, the preceeding "->" sign
                 must be present. EG: Something like "23|model" will {which} is
                 one of: "model", "base" or "design".

Example:
  pipe orbit_at_s Q10->0.4|model   ! Orbit at 0.4 meters from Q10 element exit end in model lattice.
```

13.4.70 pipe place_buffer

Output the place command buffer and reset the buffer. The contents of the buffer are the place commands that the user has issued. See the Tao manual for more details.

```
   pipe place_buffer
```

13.4.71 pipe plot_curve

Output curve information for a plot.

```
   pipe plot_curve {curve_name}
```

13.4.72 pipe plot_graph

Output graph info.

```
   pipe plot_graph {graph_name}


{graph_name} is in the form:
  {p_name}.{g_name}
where
  {p_name} is the plot region name if from a region or the plot name if a template plot.
  This name is obtained from the pipe plot_list command.
  {g_name} is the graph name obtained from the pipe plot1 command.
```

13.4.73 pipe plot_histogram

Output plot histogram info.

```
   pipe plot_histogram {curve_name}
```

13.4.74 pipe plot_lat_layout

Output plot Lat_layout info

```
   pipe plot_lat_layout {ix_uni}@{ix_branch}

Note: The returned list of element positions is not ordered in increasing
      longitudinal position.
```

13.4.75 pipe plot_list

Output list of plot templates or plot regions.

```
   pipe plot_list {r_or_g}

where "{r/g}" is:
  "r"      ! list regions of the form ix;region_name;plot_name;visible;x1;x2;y1;y2
  "t"      ! list template plots of the form ix;name
```

13.4.76 pipe plot_template_manage

Template plot creation or destruction.

```
   pipe plot_template_manage {template_location}^^{template_name}^^
                       {n_graph}^^{graph_names}

Where:
  {template_location} - Location to place or delete a template plot.
                          Use "@Tnnn" syntax for the location.
  {template_name}     - The name of the template plot.
                          If deleting a plot this name is immaterial.
  {n_graph}           - The number of associated graphs.
                          If set to -1 then any existing template plot is deleted.
  {graph_names}       - Names of the graphs. graph_names should be in the form:
                            graph1_name^^graph2_name^^...^^graphN_name
                          where N=n_graph names
```

13.4.77 pipe plot_curve_manage

Template plot curve creation/destruction

```
   pipe plot_curve_manage {graph_name}^^{curve_index}^^{curve_name}

If {curve_index} corresponds to an existing curve then this curve is deleted.
In this case the {curve_name} is ignored and does not have to be present.
If {curve_index} does not not correspond to an existing curve, {curve_index}
must be one greater than the number of curves.
```

13.4.78 pipe plot_graph_manage

Template plot graph creation/destruction

```
   pipe plot_graph_manage {plot_name}^^{graph_index}^^{graph_name}

If {graph_index} corresponds to an existing graph then this graph is deleted.
In this case the {graph_name} is ignored and does not have to be present.
If {graph_index} does not not correspond to an existing graph, {graph_index}
must be one greater than the number of graphs.
```

13.4.79 pipe plot_line

Output points used to construct the "line" associated with a plot curve.

```
   pipe plot_line {region_name}.{graph_name}.{curve_name} {x_or_y}

Optional {x-or-y} may be set to "x" or "y" to get the smooth line points x or y
component put into the tao_c_interface_com%c_real array buffer.
Note: The plot must come from a region, and not a template, since no template plots
      have associated line data.
Examples:
  pipe plot_line r13.g.a   ! String array output.
  pipe plot_line r13.g.a x ! x-component of line points put in array buffer.
  pipe plot_line r13.g.a y ! y-component of line points put in array buffer.
```

13.4.80 pipe plot_symbol

Output locations to draw symbols for a plot curve.

```
   pipe plot_symbol {region_name}.{graph_name}.{curve_name} {x_or_y}

Optional {x_or_y} may be set to "x" or "y" to get the symbol x or y
positions put into the real array buffer.
Note: The plot must come from a region, and not a template,
      since no template plots have associated symbol data.
Examples:
  pipe plot_symbol r13.g.a       ! String array output.
  pipe plot_symbol r13.g.a x     ! x-component of the symbol positions
                                     loaded into the real array buffer.
  pipe plot_symbol r13.g.a y     ! y-component of the symbol positions
                                     loaded into the real array buffer.
```

13.4.81 pipe plot_transfer

Output transfer plot parameters from the "from plot" to the "to plot" (or plots).

```
   pipe plot_transfer {from_plot} {to_plot}

To avoid confusion, use "@Tnnn" and "@Rnnn" syntax for {from_plot}.
If {to_plot} is not present and {from_plot} is a template plot, the "to plots"
 are the equivalent region plots with the same name. And vice versa
 if {from_plot} is a region plot.
```

13.4.82 pipe plot1

Output info on a given plot.

```
   pipe plot1 {name}
```

```
{name} should be the region name if the plot is associated with a region.
Output syntax is parameter list form. See documentation at the beginning of this file.
```

13.4.83 pipe ptc_com

Output Ptc_com structure components.

```
   pipe ptc_com
```

13.4.84 pipe ring_general

Output lattice branch with closed geometry info (emittances, etc.)

```
   pipe ring_general {ix_uni}@{ix_branch}|{which}
```

```
where {which} is one of:
  model
  base
  design
Example:
  pipe ring_general 1@0|model
```

13.4.85 pipe shape_list

Output lat_layout or floor_plan shapes list

```
   pipe shape_list {who}
```

```
{who} is one of:
  lat_layout
  floor_plan
```

13.4.86 pipe shape_manage

Element shape creation or destruction

```
   pipe shape_manage {who} {index} {add_or_delete}
```

```
{who} is one of:
  lat_layout
  floor_plan
{add_or_delete} is one of:
  add     -- Add a shape at {index}.
             Shapes with higher index get moved up one to make room.
  delete  -- Delete shape at {index}.
             Shapes with higher index get moved down one to fill the gap.

Example:
  pipe shape_manage floor_plan 2 add
Note: After adding a shape use "pipe shape_set" to set shape parameters.
This is important since an added shape is in a ill-defined state.
```

13.4.87 pipe shape_pattern_list

Output list of shape patterns or shape pattern points

```
   pipe shape_pattern_list {ix_pattern}

If optional {ix_pattern} index is omitted then list all the patterns.
If {ix_pattern} is present, list points of given pattern.
```

13.4.88 pipe shape_pattern_manage

Add or remove shape pattern

```
   pipe shape_pattern_manage {ix_pattern}^^{pat_name}^^{pat_line_width}

Where:
  {ix_pattern}      -- Pattern index. Patterns with higher indexes will be moved up
                                      if adding a pattern and down if deleting.
  {pat_name}        -- Pattern name.
  {pat_line_width}  -- Line width. Integer. If set to "delete" then section
                                      will be deleted.
```

13.4.89 pipe shape_pattern_point_manage

Add or remove shape pattern point

```
   pipe shape_pattern_point_manage {ix_pattern}^^{ix_point}^^{s}^^{x}

Where:
  {ix_pattern}      -- Pattern index.
  {ix_point}        -- Point index. Points of higher indexes will be moved up
                                    if adding a point and down if deleting.
  {s}, {x}          -- Point location. If {s} is "delete" then delete the point.
```

13.4.90 pipe shape_set

Set lat_layout or floor_plan shape parameters.

```
   pipe shape_set {who}^^{shape_index}^^{ele_name}^^{shape}^^{color}^^
                  {shape_size}^^{type_label}^^{shape_draw}^^
                  {multi_shape}^^{line_width}
```

```
{who} is one of:
  lat_layout
  floor_plan
```

13.4.91 pipe show

Output the output from a show command.

```
   pipe show {line}
```

```
{line} is the string to pass through to the show command.
Example:
  pipe show lattice -pipe
```

13.4.92 pipe slave_control

Output slave information for a given lord element.

```
   pipe slave_control {ele_id}
```

```
Where:
  {ele_id} is the lord element.

Example:
  pipe slave_control 2@1>>q01w

The output is a number of lines with each line giving the information:
    Slave-branch;Slave-index;Slave-name;Slave-type;Attribute-controlled;Control-expression;Value

Note: The last three fields will only be non-blank for ramper, overlay and group lords with
the value field being blank for ramper lords.

Note: For control expressed as a set of knot points (as opposed to an expression),
the control-expression will be "knots".

Note: The Value field is the contribution to the slave attribute value due to the lord.
```

13.4.93 pipe space_charge_com

Output space_charge_com structure parameters.

```
   pipe space_charge_com
```

```
Output syntax is parameter list form. See documentation at the beginning of this file.
```

13.4.94 pipe species_to_int

Convert species name to corresponding integer

```
   pipe species_to_int {species_str}
```

```
Example:
  pipe species_to_int CO2++
```

13.4.95 pipe species_to_str

Convert species integer id to corresponding

```
   pipe species_to_str {species_int}

Example:
  pipe species_to_str -1     ! Returns 'Electron'
```

13.4.96 pipe spin_invariant

Output closed orbit spin axes n0, l0, or m0 at the ends of all lattice elements in a branch. n0, l0, and m0 are solutions of the T-BMT equation. n0 is periodic while l0 and m0 are not. At the beginning of the branch, the orientation of the l0 or m0 axes in the plane perpendicular to the n0 axis is chosen a bit arbitrarily. See the Bmad manual for more details.

```
   pipe spin_invariant {flags} {who} {ix_uni}@{ix_branch}|{which}

Where:
  {flags}       - Optional flags (currently there is only one):
                    -array_out  If present, the output will be available in
                                            the tao_c_interface_com%c_real.
  {who}         - One of: l0, n0, or m0
  {ix_uni}      - A universe index. Defaults to s%global%default_universe.
  {ix_branch}   - A branch index. Defaults to s%global%default_branch.
  {which}       - Switch which is one of:
                     model
                     base
                     design

Example:
  pipe spin_invariant 1@0|model

Note: This command is under development. If you want to use please contact David Sagan.
```

13.4.97 pipe spin_polarization

Output spin polarization information

```
   pipe spin_polarization {ix_uni}@{ix_branch}|{which}

Where:
  {ix_uni} is a universe index. Defaults to s%global%default_universe.
  {ix_branch} is a branch index. Defaults to s%global%default_branch.
  {which} is one of:
    model
    base
    design

Example:
  pipe spin_polarization 1@0|model

Note: This command is under development. If you want to use please contact David Sagan.
```

13.4.98 pipe spin_resonance

Output spin resonance information

```
   pipe spin_resonance {ix_uni}@{ix_branch}|{which} {ref_ele}


Where:
  {ix_uni} is a universe index. Defaults to s%global%default_universe.
  {ix_branch} is a lattice branch index. Defaults to s%global%default_branch.
  {which} is one of: "model", "base" or "design"
  {ref_ele} is an element name or index.
This will return a string_list with the following fields:
  spin_tune                   -- Spin tune
  dq_X_sum, dq_X_diff         -- Tune sum Q_spin+Q_mode and tune difference
                                   Q_spin-Q_mode for modes X = a, b, and c.
  xi_res_X_sum, xi_res_X_diff -- The linear spin/orbit sum and difference resonance
                                   strengths for X = a, b, and c modes.
```

13.4.99 pipe super_universe

Output super_Universe parameters.

```
   pipe super_universe
```

13.4.100 pipe taylor_map

Output Taylor map between two points.

```
   pipe taylor_map {ele1_id} {ele2_id} {order}


Where:
  {ele1_id}   - The start element.
  {ele2_id}   - The end element.
  {order}     - The map order. Default is order set in the lattice file.
                  {order} cannot be larger than what is set by the lattice file.

If {ele2_id} = {ele1_id}, the 1-turn transfer map is computed.
Note: {ele2_id} should just be an element name or index without universe,
      branch, or model/base/design specification.
Example:
  pipe taylor_map 2@1>>q01w|design q02w  2
```

13.4.101 pipe twiss_at_s

Output twiss parameters at given s position.

```
   pipe twiss_at_s {ix_uni}@{ele}->{s_offset}|{which}
```

```
Where:
  {ix_uni}    - A universe index. Defaults to s%global%default_universe.
  {ele}       - An element name or index. Default is the Beginning element of branch 0.
  {s_offset}  - Evaluation point offset from the downstream end of ele. Default is 0.
                  If {s_offset} is present, "->" must also be present.
  {which}     - One of: "model", "base" or "design".
```

13.4.102 pipe universe

Output universe info.

```
   pipe universe {ix_uni}

Use "pipe global" to get the number of universes.
```

13.4.103 pipe var

Output parameters of a given variable.

```
   pipe var {var} {slaves}

Note: use "pipe var_general" to get a list of variables.
```

13.4.104 pipe var_create

Create a single variable

```
   pipe var_create {var_name}^^{ele_name}^^{attribute}^^{universes}^^
                     {weight}^^{step}^^{low_lim}^^{high_lim}^^{merit_type}^^
                     {good_user}^^{key_bound}^^{key_delta}

{var_name} is something like "kick[5]".
Before using var_create, setup the appropriate v1_var array using
the "pipe var_v1_create" command.
```

13.4.105 pipe var_general

Output list of all variable v1 arrays

```
   pipe var_general

Output syntax:
  {v1_var name};{v1_var%v lower bound};{v1_var%v upper bound}
```

13.4.106 pipe var_v_array

Output list of variables for a given data_v1.

```
   pipe var_v_array {v1_var}

Example:
  pipe var_v_array quad_k1
```

13.4.107 pipe var_v1_array

Output list of variables in a given variable v1 array

```
   pipe var_v1_array {v1_var}
```

13.4.108 pipe var_v1_create

Create a v1 variable structure along with associated var array.

```
   pipe var_v1_create {v1_name} {n_var_min} {n_var_max}
```

```
{n_var_min} and {n_var_max} are the lower and upper bounds of the var
Example:
  pipe var_v1_create quad_k1 0 45
This example creates a v1 var structure called "quad_k1" with an associated
variable array that has the range [0, 45].

Use the "pipe var_create" and "set variable" commands to set variable parameters.
Note: When setting multiple variable parameters, first set
  set global lattice_calc_on = F")
to prevent Tao trying to evaluate the
partially created variable and generating unwanted error messages.
```

13.4.109 pipe var_v1_destroy

Destroy a v1 var structure along with associated var sub-array.

```
   pipe var_v1_destroy {v1_datum}
```

13.4.110 pipe wall3d_radius

Output vaccum chamber wall radius for given s-position and angle in (x,y) plane. The radius is with respect to the local wall origin which may not be the (x,y) = (0,0) origin.

```
   pipe wall3d_radius {ix_uni}@{ix_branch} {s_position} {angle}
```

```
Where:
  {ix_uni} is a universe index. Defaults to s%global%default_universe.
  {ix_branch} is a lattice branch index.
  {s_position} is the s-position to evaluate at.
  {angle} is the angle to evaluate at.
```

13.4.111 pipe wave

Output Wave analysis info.

```
   pipe wave {who}
```

```
Where {who} is one of:
  params
  loc_header
  locations
  plot1, plot2, plot3
```

Part II

Programmer's Guide

Chapter 14

Customizing Tao

Tao has been designed to be readily extensible with a minimum of effort when certain rules are followed. This chapter discusses how this is done. This is separate from using *Tao*'s `pipe` command (§13.2) to control *Tao*.

14.1 Initial Setup

Creating a custom version of *Tao* involves creating custom code that is put in a directory that is distinct from the `tao` directory that contains the standard *Tao* code files.

It is important to remember that the code in the `tao` directory is not to be modified. This ensures that, as time goes on, and as *Tao* is developed by the "Taoist" developers, changes to the code in the `tao` directories will have a minimal chance to break your custom code. If you do feel you need to change something in the `tao` directory, please seek help first.

To setup a custom *Tao* version do the following:

1. Establish a base directory in which things will be built. This directory can have any name. Here we will call this directory `ROOT`.
2. Make a subdirectory of `ROOT` that will contain the custom code. This directory can have any name. Here this directory will be called `tao_custom`.
3. Copy the files from the directory `tao/customization` to `ROOT/tao_custom`. The `tao` directory is part of the *Bmad* package. If you do not know where to find it, ask your local Guru where it is. Along with a `README` file, there are two CMake[1] script files in the `customization` directory:
   ```
   CMakeLists.txt
   cmake.custom_tao
   ```
   These scripts are setup to make an executable called `custom_tao`. This name can be changed by modifying the `cmake.custom_tao` file.
4. Copy the file `tao/program/tao_program.f90` to `ROOT/tao_custom`.
5. Copy as needed `hook` files from `tao/hook` to `ROOT/tao_custom`. The hook files you will need are the hook files you will want to modify to customize *Tao*. See below for details. See §14.6 for an example.

[1] CMake is a program used for compiling code.

6. Go to the `ROOT/tao_custom` directory and use the command `mk` to create the executable
   ```
   ROOT/production/bin/custom_tao.
   ```
   If a debug executable is wanted, the command `mkd` will create one at:
   ```
   ROOT/debug/bin/custom_tao
   ```

A debug executable is only needed if you are debugging the code. The debug exe will run much slower than the production version.

14.2 It's All a Matter of Hooks

The golden rule when extending *Tao* is that you are only allowed to customize routines that have the name "hook" in them. These files are located in the directory `tao/hook`. To customize one of these files, copy it from `tao/hook` to `ROOT` and then make modifications to the copy.

The reason for this golden rule is to ensure that, as time goes by, and revisions are made to the *Tao* routines to extend *Tao*'s usefulness and to eliminate bugs, these changes will have a minimum impact on the specialized routines you write. What happens if the modification you want to do cannot be accomplished by customizing a hook routine? The answer is to contact the *Tao* programming team and we will modify *Tao* and provide the hooks you need so that you can then do your customization.

14.3 Implementing a Hook Routine in Tao

Function pointers are used by *Tao* to call customized hook routines. *Tao* uses the same system as *Bmad* where an abstract interface with a `_def` suffix in the name is defined along with a function pointer with a `_ptr` suffix. For example, the `tao_hook_command` routine has the function pointer (defined in `/tao/code/tao_interface.f90`):

```
procedure(tao_hook_command_def), pointer :: tao_hook_command_ptr => null()
```

To use a customized `tao_hook_command` routine, the following can be put in the `tao_program.f90` that was copied to your area:

```
tao_hook_command_ptr => tao_hook_command
```

Important: To not duplicate documentation, full details on setting up a hook routine is in the section "Custom and Hook Routines" in the *Bmad* manual. Please read this.

14.4 Initializing Hook Routines

One way to initialize a hook routine is to read in parameters from an initialization file. If an initialization file is used, the filename may be set using the `s%global%hook_init_file` string. This string may be set in the `tao_params` namelist (§10.6) or may be set on the command line using the `-hook_init_file` option (§10.1).

14.5 Hook Routines

To get a good idea of how *Tao* works it is recommended to spend a little bit of time going through the source files. This may also provide pointers on how to make customizations in the hook routines. Of particular interest is the module `tao_lattice_calc_mod.f90` where tracking and lattice parameters are computed.

Plotting is based upon the `quick_plot` subroutines which are documented in the *Bmad* reference manual. If custom plotting is desired this material should be reviewed to get familiar with the concepts of "graph", "box", and "page".

The following is a run through of each of the hook routines. Each routine is in a separate file called `tao/hook/<hook_routine_name>.f90`. See these files for subroutine headers and plenty of comments throughout the dummy code to aid in the modification of these subroutines.

14.5.1 tao_hook_branch_calc

This hook routine is called by tao_lattice_calc when tracking, twiss calculations, etc are done. This subroutine can be used, for example, to do custom calculations on a lattice branch.

Also see `tao_hook_lattice_calc` and `tao_hook_universe_calc_post_process`.

14.5.2 tao_hook_command

Any custom commands are placed here. The dummy subroutine already has a bit of code that replicates what is performed in `tao_command`. Commands placed here are searched before the standard *Tao* commands. This allows for the overwriting of any standard *Tao* command.

By default, there is one command included in here: `'hook'`. This is just a simple command that doesn't really do anything and is for the purposes of demonstrating how a custom command would be implemented.

The only thing needed to be called at the end of a custom command is `tao_cmd_end_calc`. This will perform all of the steps listed in Section §3.6.

See Sec. §14.7 for an example of how to use this hook.

14.5.3 tao_hook_data_sanity_check

Hook routine to check if a custom datum is internally consistent. This routine is called by tao_data_-sanity_check. See this routine for more details.

14.5.4 tao_hook_draw_floor_plan

Routine to customize the plotting of the floor_plan. Also see: tao_hook_draw_graph.

14.5.5 tao_hook_draw_graph

This will customize the plotting of a graph. See the *Tao* module `tao_plot_mod` for details on what it normally done. You will also need to know how `quick_plot` works (See the *Bmad* manual).

14.5.6 tao_hook_evaluate_a_datum

Any custom data types are defined and calculated here. If a non-standard data type is listed in the initialization files, then a corresponding data type must be placed in this routine. The tutorial uses this hook routine when calculating the emittance.

Dependent lattice parameters (such as closed orbits, beta functions, etc.) are recalculated every time *Tao* believes the lattice has changed (for example, after a `change` command). This is done in `tao_lattice_calc`. `tao_lattice_calc` in turn calls `tao_evaluate_a_datum` for each datum. `tao_evaluate_a_datum` in turn calls `tao_hook_evaluate_a_datum` to allow for custom data evaluations.

See the `tao_evaluate_a_datum` routine as an example as how to handle datums. The arguments for `tao_hook_evaluate_a_datum` is

```
  tao_hook_evaluate_a_datum (found, datum, u, tao_lat, datum_value, valid_value)
```

The `found` logical argument should be set to `True` for datums that are handled by this hook routine and `found` should be set to `False` for all other datums.

14.5.7 tao_hook_graph_postsetup

14.5.8 tao_hook_graph_setup

Use this to setup custom graph data for a plot.

14.5.9 tao_hook_init1 and tao_hook_init2

After the `design` lattice and the global and universe structures are initialized, `tao_hook_init1` is called from the `tao_init` routine. Here, any further initializations can be added. In particular, if any custom hook structures need to be initialized, here's the place to do it.

Further down in `tao_init`, `tao_hook_init2` is called. Normally you will want to use `tao_hook_init1`. However, `tao_hook_init2` can be used, for example, ! to set model variable values different from design variable values since when `tao_hook_init1` is called the `model` lattice has not yet been initialized.

14.5.10 tao_hook_init_beam

14.5.11 tao_hook_init_data

14.5.12 tao_hook_init_global

14.5.13 tao_hook_init_lattice_post_parse

This will do a custom lattice initialization. The standard lattice initialization just calls `bmad_parser`. If anything more complex needs to be done then do it here. This is also where any custom overlays or other elements would be inserted after the parsing is complete. But in general, anything placed here should, in principle, be something that can be placed in a lattice file.

This is the only routine that should insert elements in the ring. This is because the *Tao* data structures use the element index for each element associated with the datum. If all the element indexes shift then the data structures will break. If new elements need to be inserted then modify this routine and recompile. You can alternatively create a custom initialization file used by this routine that reads in any elements to be inserted.

14.5.14 tao_hook_init_plotting

14.5.15 tao_hook_init_read_lattice_info

14.5.16 tao_hook_init_var

14.5.17 tao_hook_lattice_calc

The standard lattice calculation can be performed for single particle, particle beam tracking and will recalculate the orbit, transfer matrices, twiss parameters and load the data arrays. If something else needs to be performed whenever the lattice is recalculated then it can be done here. A custom lattice calculation can be performed in any universe separately, this allows for the possibility of, for example, tracking a single particle for one lattice and beams in another.

See also `tao_hook_branch_calc` and `tao_hook_universe_calc_post_process`.

14.5.18 tao_hook_merit_data

A custom data merit type can be defined here. Table 8.2 lists the standard merit types. If a custom merit type is used then `load_it` in `tao_hook_load_data_array` may also need to be modified to handle this merit type, additionally, all standard data types may need to be overridden in `tao_hook_load_data_array` in order for the custom `load_it` to be used. See `tao_merit.f90` for how the standard merit types are calculated.

14.5.19 tao_hook_merit_var

This hook will allow for a custom variable merit type. However, since there is no corresponding data transfer, no `load_it` routine needs to be modified. See `tao_merit.f90` for how the standard merit types are calculated.

14.5.20 tao_hook_optimizer

If a non standard optimizer is needed, then it can be implemented here. See the `tao_*_optimizer.f90` files for how the standard optimizers are implemented.

14.5.21 tao_hook_parse_command_args

The `tao_hook_parse_command_args` routine can be used to set the names of initialization files. The file names are stored in the `s%com` structure. For example, in the hook file, the following changes the default plot initialization file:

```
  s%com%hook_plot_file = '/nfs/acc/user/dcs16/my_plot_init.tao'
```

Note that if an initialization file name is given on the command line or in the root *Tao* initialization file, that name will supersede the hook name.

14.5.22 tao_hook_plot_setup

Use this routine to override the `tao_plot_data_setup` routine which essentially transfers the information from the `s%u(:)%data` arrays to the `s%plot_page%region(:)%plot%graph(:)%curve(:)` arrays. This may be useful if you want to make a plot that isn't simply the information in a data or variable array.

14.5.23 tao_hook_post_process_data

Here can be placed anything that needs to be done after the data arrays are loaded. This routine is called immediately after the data arrays are called and before the optimizer or plotting is done, so any final modifications to the lattice or data can be performed here.

14.5.24 tao_hook_show_cmd

14.5.25 tao_hook_universe_calc_post_process

The standard lattice calculation can be performed for single particle, particle beam tracking and will recalculate the orbit, transfer matrices, twiss parameters and load the data arrays. If something else needs to be performed whenever the lattice is recalculated then it can be done here. A custom lattice calculation can be performed in any universe separately, this allows for the possibility of, for example, tracking a single particle for one universe and beams in another.

See also `tao_hook_branch_calc` and `tao_hook_lattice_calc`.

14.6 Adding a New Data Type Example

As an example of a customization, let's include a new data type called `particle_emittance`. This will be the non-normalized x and y emittance as found from the Courant-Snyder invariant. This data type will behave just like any other data type (i.e. `orbit`, `phase` etc...).

This example will only require the modification of one file: `tao_hook_evaluate_a_datum.f90`. This file should be copied from the `tao/hook` directory and put in your `ROOT/code` directory (§14.1).

The formula for single particle emittance is

$$\epsilon = \gamma x^2 + 2\alpha x x' + \beta x'^2 \tag{14.1}$$

Place the following code in `tao_hook_evaluate_a_datum.f90` in the `case select` construct. Also add the necessary type declarations. See the routine `tao_evaluate_a_datum` as an example.

```
type (coord_struct), pointer :: orbit(:)
type (ele_struct), pointer :: ele
type (lat_struct), pointer :: lat
integer ix_ele
...
lat => tao_lat%lat
orbit => tao_lat%tao_branch(0)%orbit
ele => tao_pointer_to_datum_ele (lat, datum%ele_name, datum%ix_ele, datum, &
                                                        valid_value, why_invalid)
...
```

```
  select case (datum%data_type)
  case ('particle_emittance.x')
    datum_value =  (ele%a%gamma * orbit(ix_ele)%vec(1)**2 + &
     2 * ele%a%alpha * orbit(ix_ele)%vec(1) * orbit(ix_ele)%vec(2) + &
     ele%a%beta * orbit(ix_ele)%vec(2)**2)

  case ('particle_emittance.y')
    datum_value = (ele%b%gamma * orbit(ix_ele)%vec(3)**2 + &
     2 * ele%b%alpha * orbit(ix_ele)%vec(3) * orbit(ix_ele)%vec(4) + &
     ele%b%beta * orbit(ix_ele)%vec(4)**2)
  end select
```

This defines what is to be calculated for each `particle_emittance` datum. There are two transverse coordinates, so two definitions need to be made, one for each dimension.

Now you just need to declare the data types in the `tao.init` and `tao_plot.init` files. For the sake of this example, modify the example files found in the `bmad-doc/tao_examples` directory

```
mkdir ROOT/my_example
  cp tao/example/*.init ROOT/my_example
  cp tao/example/*.lat ROOT/my_example
```

In `ROOT/my_example/tao.init` add the following lines to the data declarations section

```
  &tao_d2_data
    d2_data%name = "particle_emittance"
    universe = 0
    n_d1_data = 2
  /

  &tao_d1_data
    ix_d1_data = 1
    d1_data%name = "x"
    default_weight = 1
    use_same_lat_eles_as = 'orbit.x"
  /

  &tao_d1_data
    ix_d1_data = 2
    d1_data%name = "y"
    default_weight = 1
    use_same_lat_eles_as = 'orbit.x"
  /
```

In `ROOT/my_example/tao_plot.init` add the following lines to the end of the file

```
  &tao_template_plot
    plot%name = 'particle_emittance'
    plot%x_axis_type = 'index'
    plot%n_graph = 2
  /

  &tao_template_graph
    graph%name = 'x'
    graph_index = 1
    graph%box = 1, 2, 1, 2
    graph%title = 'Horizontal Emittance (microns)'
    graph%margin =  0.15, 0.06, 0.12, 0.12, '%BOX'
    graph%y%label = 'x'
    graph%y%max =  15
    graph%y%min =  0.0
    graph%y%major_div = 4
    curve(1)%data_source = 'data'
    curve(1)%data_type   = 'particle_emittance.x'
    curve(1)%y_axis_scale_factor = 1e6 !convert from meters to microns
  /

  &tao_template_graph
    graph%name = 'y'
    graph_index = 2
    graph%box = 1, 1, 1, 2
    graph%title = 'Vertical Emittance (microns)'
    graph%margin =  0.15, 0.06, 0.12, 0.12, '%BOX'
    graph%y%label = 'Y'
    graph%y%max =  15
    graph%y%min =  0.0
    graph%y%major_div = 4
    curve(1)%data_source = 'data'
    curve(1)%data_type = 'particle_emittance.y'
    curve(1)%units_factor = 1e6 !convert from meters to microns
  /
```

These namelists are described in detail in Chapter 10.

We are now ready to compile and then run the program. The *Tao* library should have already been created so all you need to do is

```
cd ROOT/code
mk
  cd ROOT/my_example
  ../production/bin/custom_tao
```

After your custom *Tao* initializes type

```
  place bottom particle_emittance
  scale
```

Your plot should look like Figure 14.1.

The emittance (as calculated) is not constant. This is due to dispersion and coupling throughout the ring. *Bmad* provides a routine to find the particle emittance from the twiss parameters that includes dispersion and coupling called `orbit_amplitude_calc`.

14.7 Reading in Measured Data Example

This section shows how to construct a customized version of *Tao*, called `ping_tao`, to read in measured data for analysis. This example uses data from the Fermilab proton recirculation. The data is obtained by measuring the orbit turn-by-turn of a beam that has been initially pinged to give it a finite oscillation amplitude.

The files for constructing `ping_tao` can be found in the directory

```
  bmad-doc/tao_examples/custom_tao_with_measured_data
```

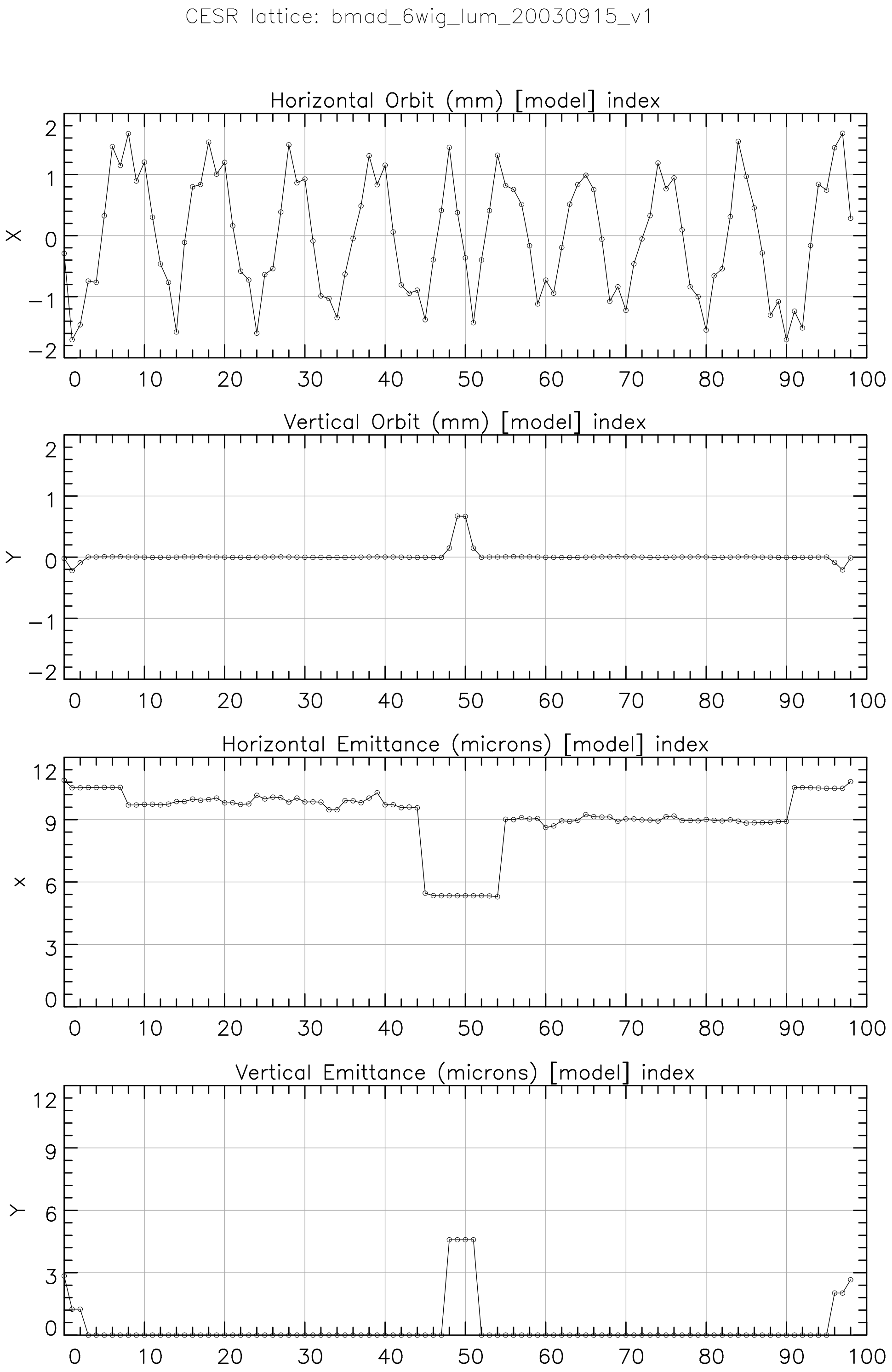


Figure 14.1: Custom data type: non-normalized emittance

The files in this directory are as follows:

CMakeLists.txt, cmake.ping_tao
: Script files for creating `ping_tao`. See Sec. §14.1.

README
: The `README` file gives some instructions on how to create `ping_tao`

RRNOVAMU2E11172016.bmad
: Lattice file for the proton recirculation ring.

data
: Directory where some ping data is stored

tao.init
: *Tao* initialization file defining the appropriate data and variable structures (§10.3)

tao.startup
: File with some command that are executed when *Tao* is started. These commands will read in and plot some data.

tao_hook_command.f90
: Custom code for reading in ping data. The template used to construct this file is at `tao/hook/tao_hook_command.f90` (§14.5.2).

tao_plot.init
: File for defining plot parameters (§10.13).

tao_program.f90
: copy of the `tao/program/tao_program.f90` file (§14.1).

After creating the `ping_tao` program (see the `README` file), the program can be run by going to the custom_tao_with_measured_data directory and using the command:

```
../production/bin/ping_tao
```

The customized `tao_hook_command` routine implements a custom command called `pingread`. This command will read in ping data. Ping data is the amplitude and phase of the beam oscillations at a BPM for either the `a-mode` or `b-mode` oscillations. See the write up on ping data types in Sec. §6.9 under `ping_a.amp_x`, and `ping_b.amp_x` for more details.

The data files in the `data` directory contain data for either the `a-mode` or `b-mode` ping at either the horizontal or vertical BPMs.

The syntax of the `pingread` command is:

```
  pingread <mode> <filename> <data_or_ref>
```

The first argument, `<mode>`, should be either "`a_mode`" "`b_mode`" indicating wether the data is for the `a-mode` `b-mode` analysis (a better setup would encode this information in the data file itself). The second argument, `filename` is the name of the data file, and the third argument, `data_or_ref` should be "`data`" or "`reference`" indicating that the data is to be read into the `meas_value` or `ref_value` of the appropriate `tao_data_struct`.

14.7.1 Analysis of the tao_hook_command.f90 File

The first part of the `tao_hook_command` routine parses the command line to see if the `pingread` command is present. The relevant code, somewhat condensed, is:

```
  subroutine tao_hook_command (command_line, found)
```

```
!!!! put your list of hook commands in here.

character(16) :: cmd_names(1) = [character(16):: 'pingread']

! "found" will be set to TRUE if the command is found.

found = .false.

! strip the command line of comments

call string_trim (command_line, cmd_line, ix_line)
ix = index(cmd_line, '!')
if (ix /= 0) cmd_line = cmd_line(:ix-1)        ! strip off comments

! blank line => nothing to do

if (cmd_line(1:1) == '') return

! match first word to a command name
! If not found then found = .false.

call match_word (cmd_line(:ix_line), cmd_names, ix_cmd, .true., .true., cmd_name)
if (ix_cmd < 0) then
  call out_io (s_error$, r_name, 'AMBIGUOUS HOOK COMMAND')
  found = .true.
  return
endif

found = .true.
call string_trim (cmd_line(ix_line+1:), cmd_line, ix_line)
```

Note: To quickly find information on routines and structures, use the `getf` and `listf` scripts as explained in the *Bmad* manual. For example, typing "`getf string_trim`" on the system command line will give information on the string_trim subroutine.

The above code tests to see if the command is `pingread` and, if not, returns without doing anything.

If the `pingread` command is found, the rest of the command line is parsed to get the `<mode>`, `<filename>`, and `<data_or_ref>` arguments.

In the `tao.init` file, a `tune` d2 datum is setup to have two `d1` datum arrays One for the `a`-mode tune and one for the `b`-mode tune:

```
&tao_d2_data
  d2_data%name = "tune"
  universe = '*'  ! apply to all universes
  n_d1_data = 2
/

&tao_d1_data
  ix_d1_data = 1
  d1_data%name = "a"
  default_weight = 1e6
```

```
    ix_min_data = 1
    ix_max_data = 1
  /

  &tao_d1_data
    ix_d1_data = 2
    d1_data%name = "b"
    default_weight = 1e6
    ix_min_data = 1
    ix_max_data = 1
  /
```

And each `d1` array has only one datum since the `a`-mode and `b`-mode tunes have only one value associated with them (as opposed to, say an orbit which will have multiple values from different BPMs).

In a data file there is a header section which, among other things, records the tune. In a line beginning with the word "`Tune`". Example:

```
                   Horz          Vert          Sync.
   Tune           ( .452444)   ( .404434)   ( 0       ) 2p
```

In the `tao_hook_command` file, after the arguments are parsed, the header part of the data file is read to extract the tune datums:

```
  type (tao_d2_data_array_struct), allocatable :: d2(:)
  ...
  if (line(1:4) == 'Tune') then
    call tao_find_data (err, 'tune', d2_array = d2)
    if (size(d2) /= 1) then
      call out_io (s_fatal$, r_name, 'NO TUNE D2 DATA STRUCTURE DEFINED!')
      return
    endif
```

The call to `tao_find_data` looks for a `d2` data structure named `tune`. This structure is setup in the `tao.init` file. Alternatively, the `ping_tao` program could be configured to automatically setup the appropriate data and/or variable structures via the `tao_hook_init1` routine (§14.5.9).

The returned value from the call to `tao_find_data` is an array called `d2` of type `tao_d2_data_array_struct`. `d2` holds an array of pointers to all `d2_data_struct` structures it can find. In general, there could be multiple such structures if multiple universes are being used or if the match string, in this case `'tune'`, contained wild card characters. In this case, the expectation is that there will only one universe used and thus there should be one and only one structure that matches the name `tune`. This structure will be pointed to by `d2(1)%d2`. The appropriate datums, will be:

```
  d2(1)%d2%d1(1)%d(1)   ! a-mode tune
  d2(1)%d2%d1(1)%d(2)   ! b-mode tune
```

The values read from the data file are put in these datums via the code:

```
  if (data_or_ref == 'data') then
    d2(1)%d2%d1(1)%d(1)%meas_value = twopi * (data_tune_a + nint(design_tune_a))
    d2(1)%d2%d1(1)%d(1)%good_meas = .true.
    d2(1)%d2%d1(2)%d(1)%meas_value = twopi * (data_tune_b + nint(design_tune_b))
    d2(1)%d2%d1(2)%d(1)%good_meas = .true.
  else
    d2(1)%d2%d1(1)%d(1)%ref_value = twopi * (data_tune_a + nint(design_tune_a))
    d2(1)%d2%d1(1)%d(1)%good_ref = .true.
    d2(1)%d2%d1(2)%d(1)%ref_value = twopi * (data_tune_b + nint(design_tune_b))
    d2(1)%d2%d1(2)%d(1)%good_ref = .true.
  endif
```

The next step is to setup pointers to the appropriate data arrays to receive the ping data. In the data file the ping data looks like:

```
BPM            Phase    Ampl.   RMSdev     Beta  bml_psi *Calib Old_Cal
R:HP222    -0.27314  0.46085    0.078    1.863  0.35183
R:HP224    -0.05939  0.28277    0.143    0.701 -0.43442
R:HP226     0.23140  0.31712    0.075    0.882 -0.14363
... etc ...
```

The "H" in `R:HP222`, etc. indicates that the data is from BPMs that only measure the horizontal displacement of the beam. Alternatively, a "V" would indicate data from vertical measurement BPMs.

In the `tao_hook_command` file the data pointers are setup by the code:

```
type (tao_d1_data_array_struct), allocatable, target :: d1_amp_arr(:), d1_phase_arr(:)
...
if (line(3:3) == 'H') then
  if (mode == 'a_mode') then
    call tao_find_data (err, 'ping_a.amp_x', d1_array = d1_amp_arr)
    call tao_find_data (err, 'ping_a.phase_x', d1_array = d1_phase_arr)
  else
    call tao_find_data (err, 'ping_b.amp_x', d1_array = d1_amp_arr)
    call tao_find_data (err, 'ping_b.phase_x', d1_array = d1_phase_arr)
  endif
elseif (line(3:3) == 'V') then
  if (mode == 'a_mode') then
    call tao_find_data (err, 'ping_a.amp_y', d1_array = d1_amp_arr)
    call tao_find_data (err, 'ping_a.phase_y', d1_array = d1_phase_arr)
  else
    call tao_find_data (err, 'ping_b.amp_y', d1_array = d1_amp_arr)
    call tao_find_data (err, 'ping_b.phase_y', d1_array = d1_phase_arr)
  endif
```

`line(3:3)` is either `H` or `V` indicating horizontal or vertical orbit measuring BPMs. In this case, the call to the `tao_find_data` routine returns `d1` data arrays to the amplitude data (`d1_amp_arr`) and phase data (`d1_phase_arr`). Just like the tune data, since it is assumed only one universe is being used, there should be one and only `d1` structure for the phase and only one `d1` structure for the amplitude:

```
d1_amp_arr(1)%d1      ! d1 struucture for the amplitude data
d1_phase_arr(1)%d1    ! d1 struucture for the phase data
```

To save on typing, and make the code clearer, pointers are used to point to these structures:

```
type (tao_d1_data_struct), pointer :: d1_phase, d1_amp
...
d1_amp => d1_amp_arr(1)%d1
d1_phase => d1_phase_arr(1)%d1
```

The array of datums for the amplitude and phase data will be `d1_amp%d(:)` and `d1_phase%d(:)` respectively.

After the `d1_amp` and `d1_phase` pointers have been set, there is a loop over all the lines in the file to extract the ping data. One problem faced is that the order of the data in the file is not the same as the order of the data in `d1` structures. [The data in the file is sorded in increasing numberical order in the BPM name while the order in the `d1` structures is sorted by increasing logitudinal s-position.] To get around this problem, the BPM name in the file is used to locate the appropriate datum (the associated BPM element name is stored in the `%ele_name` component of the datums):

```
character(140) :: cmd_word(12), ele_name
...
```

```
  call tao_cmd_split (line, 4, cmd_word, .false., err)
  read (cmd_word(2), *) r1
  read (cmd_word(3), *) r2
  ele_name = cmd_word(1)
  datum_amp => tao_pointer_to_datum(d1_amp, ele_name(3:))
  datum_phase => tao_pointer_to_datum(d1_phase, ele_name(3:))
```

The `line` string holds a line from the data file, the call to `tao_cmd_split` splits the line into word chunks and puts them into the array `cmd_word(:)`. `cmd_word(1)` holds the first word which is the BPM name with "`R:`" prepended to the name. The calls to `tao_pointer_to_datum` return pointers, `datum_amp` and `datum_phase`, to the approbriate datums given the BPM name.

After the appropriate datums have been identified, the ping data values read from the data file, `r1` and `r2`, are used to set the appropriate components:

```
  if (data_or_ref == 'data') then
    datum_phase%good_meas = .true.
    datum_amp%meas_value = r2
    datum_amp%good_meas = .true.
  else
    datum_phase%good_ref = .true.
    datum_amp%ref_value = r2
    datum_amp%good_ref = .true.
  endif
```

One problem is that individual data phase data points can be off by factors of 2π. To correct this, the measured phase values are shifted by factors of 2π so that they are within $\pm\pi$ of the design values. There is an added "branch cut" problem here in that, even without the factors of 2π problem, the measured phases will be off from the design values by some arbitrary amount (determined by how the zero phase is defined in the program that created the data file). If this difference between the zero phase of the data and the zero phase of design lattice (in the design lattice, the phase is taken to be zero at the beginning of the lattice) is close enough to π, the shifting of the phases by factors of 2π will not be correct. For this reason, a best guess as to what the offset is is used in the calculation to avoid the branch cut problem:

```
  rms_best = 1e30

  do i = 1, 20
    offset = i / 20.0
    data = data + nint(design + offset - data)
    rms = sum((data - design - offset)**2, mask = ok)
    if (rms < rms_best) then
      offset_best = offset
      rms_best = rms
    endif
  enddo

  data = data + nint(design + offset_best - data)
```

Chapter 15

Tao Structures

This chapter gives an overview of the structures (classes) used in *Tao*. Knowledge of the structures is needed in order to create custom versions of *Tao*. See Chapter §14 for details of how to create custom *Tao* versions.

15.1 Overview

The *Tao* code files are stored in the following directories:

```
tao/code
tao/hooks
tao/program
```

Here `tao` is the root directory of *Tao*. Ask your local guru where to find this directory.

The files in `tao/code` should not be modified when creating custom versions of *Tao*. The files in `tao/hooks`, as explained in Chapter §14, are templates used for customization. Finally, the directory `tao/program` holds the program file `tao_program.f90`.

The structures used by tao are defined in the file `tao_struct.f90`. All *Tao* structures begin with the prefix `tao_` so any structure encountered that does not begin with `tao_` must be defined in some other library The `getf` and `listf` commands can be used to quickly get information on any structure. See the *Bmad* manual for more details.

15.2 tao_super_universe_struct

The "root" structure in *Tao* is the `tao_super_universe_struct`. The definition of this structure is:

```
type tao_super_universe_struct
  type (tao_global_struct) global                    ! Global variables.
  type (tao_common_struct) :: com                    ! Global variables
  type (tao_plotting_struct) :: plotting             ! Plot parameters.
  type (tao_v1_var_struct), allocatable :: v1_var(:)  ! V1 Variable array
  type (tao_var_struct), allocatable :: var(:)       ! Array of all variables.
  type (tao_universe_struct), allocatable :: u(:)    ! Array of universes.
  type (tao_mpi_struct) mpi
```

```
    integer, allocatable :: key(:)
    type (tao_building_wall_struct) :: building_wall
    type (tao_wave_struct) :: wave
    integer n_var_used
    integer n_v1_var_used
    type (tao_cmd_history_struct) :: history(1000)        ! command history
  end type
```

An instance of this structure called `s` is defined in `tao_struct.f90`:

```
  type (tao_super_universe_struct), save, target :: s
```

This `s` variable is common to all of *Tao*'s routines and serves as a giant common block for *Tao*.

The components of the `tao_super_universe_struct` are:

%global
: The `%global` component contains global variables that a user can set in an initialization file. See §10.6 for more details.

%com
: The `%com` component is for global variables that are not directly user accessible.

%plot_page
: The `%plot_page` component holds parameters used in plotting (§15.3).

%v1_var(:)
: The `%v1_var(:)` component is an array of all the `v1_var` blocks (§5) that the user has defined (§15.4).

%var(:) The `%var(:)` array holds a list of all variables (§5) that the user has defined (§15.5).

%u(:)
: The `%u(:)` component is an array of universes (§3.3) (§15.6).

%mpi
: The `%mpi` component holds parameters needed for parallel processing (§15.7).

%key(:)
: The `%key(:)` component is an array of indexes used for key bindings (§12.1).

%building_wall
: The `%building_wall` component holds parameters associated with a building wall (§10.11).

%wave
: The `%wave` component holds parameters needed for the wave analysis (§9).

%history
: The `%history` component holds the command history (§15.11).

15.3 s%plot_page Component

The `s%plot_page` component of the `super universe` (§15.2) holds plotting information and is initialized in the routine `tao_init_plotting`. `s%plot_page` is a `tao_plot_page_struct` structure which has components:

```
  type tao_plot_page_struct
    type (tao_title_struct) title            ! Title at top of page.
    type (tao_title_struct) subtitle         ! Subtitle at top of page.
    type (qp_rect_struct) border             ! Border around plots edge of page.
    type (tao_drawing_struct) :: floor_plan
    type (tao_drawing_struct) :: lat_layout
    type (tao_shape_pattern_struct), allocatable :: pattern(:)
    type (tao_plot_struct), allocatable :: template(:)  ! Templates for the plots.
    type (tao_plot_region_struct), allocatable :: region(:)
    character(8) :: plot_display_type = 'X'   ! 'X' (X11) or 'TK'
    real(rp) size(2)                          ! width and height of window in pixels.
    real(rp) :: text_height = 12              ! In points. Scales the height of all text
    real(rp) :: main_title_text_scale  = 1.3  ! Relative to text_height
    real(rp) :: graph_title_text_scale = 1.1  ! Relative to text_height
    real(rp) :: axis_number_text_scale = 0.9  ! Relative to text_height
    real(rp) :: axis_label_text_scale  = 1.0  ! Relative to text_height
    real(rp) :: legend_text_scale      = 0.7  ! Relative to text_height
    real(rp) :: key_table_text_scale   = 0.9  ! Relative to text_height
    real(rp) :: curve_legend_line_len  = 50   ! Points
    real(rp) :: curve_legend_text_offset = 10 ! Points
    real(rp) :: floor_plan_shape_scale = 1.0
    real(rp) :: lat_layout_shape_scale = 1.0
    integer :: n_curve_pts = 401              ! Default number of points for plotting a smooth curve.
    integer :: id_window = -1                 ! X window id number.
    logical :: delete_overlapping_plots = .true. ! Delete overlapping plots when a plot is placed?
  end type
```

%template(:)
The `%template(:)` array contains the array of plot templates defined by the user (§10.13.2) and/or the default plot templates which are created in the routine `tao_init_plotting`.

%region(:)
The `%region(:)` array contains the plot regions. Each element in the array is a `tao_plot_region_struct` structure:

```
  type tao_plot_region_struct
    character(40) :: name = ''     ! Region name. Eg: 'r13', etc.
    type (tao_plot_struct) plot    ! Plot associated with this region
    real(rp) location(4)           ! [x1, x2, y1, y2] location on page.
    logical :: visible = .false.   ! To draw or not to draw.
    logical :: list_with_show_plot_command = .true.  ! False used for default plots to
                                                     !  shorten the output of "show plot"
  end type
```

Then `place` command finds the appropriate plot in the `s%plot_page%template(:)` array and copies it to the `s%plot_page%region(i)%plot` component where `i` is the index of the region specified by the `place` command.

15.4 s%v1_var Component

The `s%v1_var(:)` array holds the list of `v1` variable blocks (§5). This array is initialized in the routine `tao_init_variables`. The range of valid elements in this array goes from 1 to `s%n_v1_var_used`. Each element of this array is a `tao_v1_var_struct` structure:

```
  type tao_v1_var_struct
    character(40) :: name = ''       ! V1 variable name. Eg: 'quad_k1'.
    integer ix_v1_var                ! Index to s%v1_var(:) array
    type (tao_var_struct), pointer :: v(:) => null()
                                     ! Pointer to the appropriate section in s%var.
  end type
```

The `%ix_v1_var` component is the index of the element in the `s%v1_var(:)` array. That is, `s%v1_var(1)%ix_v1_var` = 1, etc. This is useful when debugging.

The `%v(:)` component is a pointer to the appropreiate block in the `s%var(:)` array (§15.5) which contain the individual variables associated with the particular `v1` variable block.

15.5 s%var Component

The `s%var(:)` array holds the list complete list of all variables (§5). This array is initialized in the routine `tao_init_variables`. The range of valid variables goes from 1 to `s%n_var_used`. Each element in the `s%v1_var(:)` array (§15.4) has a pointer to the section of the `s%var(:)` array holding the variables associated with `v1` block. Using a single array of variables simplifies code where one wants to simply loop over all variables (for example, during optimization).

Each element of the `s%var(:)` array is a `tao_var_struct` structure:

```
  type tao_var_struct
    character(40) :: ele_name = ''    ! Associated lattice element name.
    character(40) :: attrib_name = '' ! Name of the attribute to vary.
    character(40) :: id = ''          ! Used by Tao extension code. Not used by Tao directly.
    type (tao_var_slave_struct), allocatable :: slave(:)
    type (tao_var_slave_struct) :: common_slave
    integer :: ix_v1 = 0              ! Index of this var in the s%v1_var(i)%v(:) array.
    integer :: ix_var = 0             ! Index number of this var in the s%var(:) array.
    integer :: ix_dvar = -1           ! Column in the dData_dVar derivative matrix.
    integer :: ix_attrib = 0          ! Index in ele%value(:) array if appropriate.
    integer :: ix_key_table = 0       ! Has a key binding?
    real(rp), pointer :: model_value => null()     ! Model value.
    real(rp), pointer :: base_value => null()      ! Base value.
    real(rp) :: design_value = 0      ! Design value from the design lattice.
    real(rp) :: scratch_value = 0     ! Scratch space to be used within a routine.
    real(rp) :: old_value = 0         ! Scratch space to be used within a routine.
    real(rp) :: meas_value = 0        ! The value when the data measurement was taken.
    real(rp) :: ref_value = 0         ! Value when the reference measurement was taken.
    real(rp) :: correction_value = 0  ! Value determined by a fit to correct the lattice.
    real(rp) :: high_lim = -1d30      ! High limit for the model_value.
    real(rp) :: low_lim = 1d30        ! Low limit for the model_value.
    real(rp) :: step = 0              ! Sets what is a small step for varying this var.
    real(rp) :: weight = 0            ! Weight for the merit function term.
    real(rp) :: delta_merit = 0       ! Diff used to calculate the merit function term.
    real(rp) :: merit = 0             ! merit_term = weight * delta^2.
    real(rp) :: dMerit_dVar = 0       ! Merit derivative.
    real(rp) :: key_val0 = 0          ! Key base value
    real(rp) :: key_delta = 0         ! Change in value when a key is pressed.
    real(rp) :: s = 0                 ! longitudinal position of ele.
    character(40) :: merit_type = ''  ! 'target' or 'limit'
    logical :: exists = .false.       ! See above
    logical :: good_var = .false.     ! See above
    logical :: good_user = .true.     ! See above
    logical :: good_opt = .false.     ! See above
    logical :: good_plot = .false.    ! See above
    logical :: useit_opt = .false.    ! See above
    logical :: useit_plot = .false.   ! See above
    logical :: key_bound = .false.    ! Variable bound to keyboard key?
    type (tao_v1_var_struct), pointer :: v1 => null() ! Pointer to the parent.
  end type tao_var_struct
```

%exists
The variable exists. Non-existent variables can serve as place holders in the `s%var array`.

%good_var
The variable can be varied. Used by the lm optimizer to veto variables that do not change the merit function.

%good_user
What the user has selected using the use, veto, and restore commands.

%good_opt
Not modified by Tao. Setting is reserved to be done by extension code.

%good_plot
Not modified by Tao. Setting is reserved to be done by extension code.

%useit_opt
Variable is to be used for optimizing:
```
  %useit_opt = %exists & %good_user & %good_opt & %good_var
```
%useit_plot
If True variable is used in plotting variable values:
```
  %useit_plot = %exists & %good_plot & %good_user
```

15.6 s%u Component

The `s%u(:)` array holds the *Tao* universes (§3.3). Each element of this array is a `tao_universe_struct` structure:

```
  type tao_universe_struct
    type (tao_universe_struct), pointer :: common => null()
    type (tao_lattice_struct), pointer :: model, design, base
    type (tao_beam_struct) beam
    type (tao_dynamic_aperture_struct) :: dynamic_aperture
    type (tao_universe_branch_struct), pointer :: uni_branch(:) ! Per element information
    type (tao_d2_data_struct), allocatable :: d2_data(:)   ! The data types
    type (tao_data_struct), allocatable :: data(:)         ! Array of all data.
    type (tao_ping_scale_struct) ping_scale
    type (lat_struct) scratch_lat                          ! Scratch area.
    type (tao_universe_calc_struct) calc                   ! What needs to be calculated?
    real(rp), allocatable :: dModel_dVar(:,:)              ! Derivative matrix.
    integer ix_uni                        ! Universe index.
    integer n_d2_data_used                ! Number of used %d2_data(:) components.
    integer n_data_used                   ! Number of used %data(:) components.
    logical is_on                         ! universe turned on
    logical picked_uni                    ! Scratch logical.
  end type
```

15.7 s%mpi Component

The `s%mpi` component holds information that is used when running *Tao* multi-threaded.

15.8 s%key Component

The value of `%key(i)` is the index in the `%var(:)` array associated with the *i*þkey.

15.9 s%building_wall Component

15.10 s%wave Component

15.11 s%history Component

Part III

Bibliography

Bibliography

[Bma06] D. Sagan, "Bmad: A Relativistic Charged Particle Simulation Library" Nuc. Instrum. & Methods Phys. Res. A, **558**, pp 356-59 (2006).

The Bmad Manual can be optained at:
www.classe.cornell.edu/bmad

[Bengt97] J. Bengtsson, "The Sextupole Scheme for the Swiss Light Source (SLS): An Analytic Approach," SLS Note 9/97, Paul Scherrer Institut, (1997).

[Blender] `Blender` web page:
blender.org/

[Fra11] A. Franchi, L. Farvacque, J. Chavanne, F. Ewald, B. Nash, K. Scheidt, and R. Tom; "Vertical emittance reduction and preservation in electron storage rings via resonance driving terms correction", Phys. Rev. ST Accel. Beams, **14**, 3, 034002, (2011).
link.aps.org/doi/10.1103/PhysRevSTAB.14.034002

[CBETA19] C. Gulliford, A. Bartnik, J. Scott Berg, J. Dobbins, A. Nunez-delPrado, and D. Sagan, "Experience With CBETA Online Modeling Tools", 13th International Computational Accelerator Physics Conference, 2019.

[NR92] W. Press, B. Flannery, S. Teukolsky, W. Wetterling, *Numerical Recipes in Fortran, the Art of Scientific Computing*, Second Edition, Cambridge University Press, New York (1992)

[Montague] B. W. S. L. Montague, "Linear Optics For Improved Chromaticity Correction," CERN, Geneva, Tech. Rep. CERN-LEP-Note-165. LEP-Note-165, 1979. http://cds.cern.ch/record/443342

[Saf97] J. Safranek, "Experimental determination of storage ring optics using orbit response measurements", NIM-A388, p. 27 (1997).

[Sag99] D. Sagan, and D. Rubin, "Linear Analysis of Coupled Lattices," Phys. Rev. ST Accel. Beams 2, 074001 (1999).
https://journals.aps.org/prab/abstract/10.1103/PhysRevSTAB.2.074001

[Sag00a] D. Sagan, R. Meller, R. Littauer, and D. Rubin, "Betatron phase and coupling measurement at the Cornell Electron/Positron Storage Ring", Phys. Rev. ST Accel. Beams 3, 092801 (2000).
link.aps.org/doi/10.1103/PhysRevSTAB.3.092801

[Sag00b] D. Sagan, "Betatron phase and coupling correction at the Cornell Electron/Positron Storage Ring", Phys. Rev. ST Accel. Beams 3, 102801 (2000).
link.aps.org/doi/10.1103/PhysRevSTAB.3.102801

[Sto96] 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).

[Wang12] C. Wang, “Explicit formulas for 2nd-order driving terms due to sextupoles and chromatic effects of quadrupoles,” ANL/APS/LS-330, Argonne National Laboratory, (2012).

[Wil00] Klaus Wille, *The Physics of Particle Accelerators: An Introduction*, Translated by Jason McFall, Oxford University Press (2000).