Revision: 14.2
October 20, 2017

The Tao Manual

David Sagan

Introduction

Tao stands for “Tool for Accelerator Optics”. *Tao* is a general purpose program for simulating high energy particle beams in accelerators and storage rings. The simulation engine that *Tao* uses 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.

After the development of *Bmad*, 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.

This manual is divided into two parts. Part I is the reference section which defines the terms used by *Tao* and explains in detail the syntax of the configuration files that *Tao* uses to make a connection with a specific machine. Part II is a programmer’s guide which shows how to extend *Tao*’s capabilities and incorporate custom calculations.

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

```
http://www.lepp.cornell.edu/~dcs/bmad
```

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 that 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* or *Tao* manual pages and there will be a link for the tutorial.

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. In particular Jeff Smith who greatly contributed to this manual and Chris Mayes for his bug reports and suggestions for improvements to the program. Thanks also must go to Dave Rubin and Georg Hoffstaetter.

Contents

List of Figures

List of Tables

Part I

Reference Guide

Chapter 1

Starting Tao

1.1 Obtaining Tao

Instructions for setting up the appropriate environmental variables and for obtaining the source files can be found at:

```
http://www.lepp.cornell.edu/~dcs/bmad/
```

Briefly, you should be able to run *Tao* using the command

```
tao {-init <tao_input_file>} {-beam_all <beam_file>}
         {-beam0 <beam_file>} {-lat <lattice_file>}
```

`$ACC_EXE` is an environmental variable pointing to the directory the *Tao* executable is in. The root initialization file `<tao_input_file>` is the file that *Tao* reads to start *Tao*'s initialization process. If not present, `<tao_input_file>` defaults to `tao.init`. The `-beam_all` switch is for reading in data generated from beam tracking (§10.5). The `-beam0` switch is for specifying the initial beam distribution. The `-lat` switch is used to override the lattice file specified in the root initialization file. See section §2.1 for more details. Example:

```
tao -init my.init -lat slac.xsif
```

An initialization file is actually not needed. In this case, a `-lat` switch is manditory and *Tao* will use a set of default plot templates for plotting.

This tutorial uses the example set of input files that comes with the *Tao* library. If you are using a computer on the Cornell CLASSE linux cluster, you can get the files from:

```
$ACC_RELEASE_DIR/tao/examples/introduction_to_tao
```

If you are not on the cluster the example is at:

```
$ACC_ROOT_DIR/tao/examples/introduction_to_tao
```

In either case, copy this directory to your local area for use with the following sections.

1.2 Initializing Tao

Initialization occurs when *Tao* is started. The initialization information can reside in one file or it can be split into a number of files as discussed in Section §10.2. If no initialization files are found. *Tao* uses a default initialization.

Tao is started with the command:

```
  tao
```

Since no initialization file is specified on the command line, the default file `tao.init` (if it exists) is used. Using the example files in the `introduction_to_tao` directory *Tao* (§1.1), the `tao.init` there has the following lines:

```
  &tao_start
    plot_file = 'tao_plot.init'
  /
```

The plotting information will come from the file `tao_plot.init`. Since no other initialization files are specified (§10.2), *Tao* will look for the non-plotting information (except for the lattice file) in `tao.init`.

The lattice file is specified in the `tao_design_lattice` namelist in `tao.init`:

```
  &tao_design_lattice
    n_universes = 1
    design_lattice(1) = "bmad_L9A18A000-_MOVEREC.lat"
  /
```

Tao will setup a single universe since `n_universes = 1`. By default, *Tao* assumes that this lattice uses the *Bmad* lattice format. With the above information, *Tao* has the information on what files it needs to read to initialize itself.

1.3 Single Mode

Tao has a `single mode` in which 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.

Chapter 2

Running Tao

2.1 Initialization from the Command Line

The syntax of the command line for running *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` environmental variable then the directory specification may be omitted. The optional arguments are:

`-beam <beam_file>`
: Overrides the `beam_file` (§10.2) specified in the *Tao* initialization file.

`-beam_all <all_beam_file>`
: Overrides the `beam_all_file` (§10.5) specified in the `tao_beam_init` namelist.

`-beam0 <beam0_file>`
: Overrides the `beam0_file` (§10.5) specified in the `tao_beam_init` namelist.

`-building_wall <wall_file>`
: Overrides the `building_wall_file` (§10.2) specified in the *Tao* initialization file.

`-color_prompt`
: Sets the prompt string color to red. For different colors, use the `set global prompt_color` command (§11.26).

`-data <data_file>`
: Overrides the `data_file` (§10.2) 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 §10.10.3.

`-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 §10.10.

`-hook_init_file`
: Specifies an input file for customized versions of Tao. Default file name is `tao_hook.init`.

`-init <tao_init_file>`
: 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 `-lat` switch is mandatory and *Tao* will use a set of default plot templates for plotting.

`-lat <bmad_or_xsif_lattice_file>`
: Overrides the `design_lattice` lattice file specified in the *Tao* initialization file (§10.3). Example:

```
  tao -init my.init -lat slac.xsif
```

If there is more than one universe and the universes have different lattices, separate the different lattice names using a "|" character. Do not put any spaces in between. Example:

```
  tao -lat xsif::slac.lat|cesr.bmad
```

`-log_startup` If there is a problem with *Tao* is started, `-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 `-lat` switch is mandatory and *Tao* will use a set of default plot templates for plotting.

`-noplot`
: Suppresses the opening of the plot window.

`-plot <plot_file>`
: Overrides the `plot_file` (§10.2) specified in the *Tao* initialization file.

`-rf_on` Leaves `rfcavity` elements on. Normally *Tao* turns off these elements since Twiss and dispersion calculations do not make sense with them on.

`-startup <startup_command_file>` Overrides the `startup_file` (§10.2) specified in the *Tao* initialization file.

`-var <var_file>`
: Overrides the `var_file` (§10.2) specified in the *Tao* initialization file.

2.2 Aliases

Typing repetitive commands 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.

2.3 Command Files

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).

Do loops (§11.6) 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. Example

```
set global lattice_calc_on = F  ! Turn off lattice calculations
set global plot_on = F          ! Turn off plot calculations
... do some stuff ...
set global plot_on = T          ! Turn back on
set global lattice_calc_on = T  ! Turn back on
```

Additionally, the `global%command_file_print_on` switch controls whether printing is suppressed when a command file is called.

A `end_file` command (§11.7) can be used to signal the end of the command file.

The `pause` command (§11.13) can be used to temporarily pause the command file.

Chapter 3

Overall Organization and Structure

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].

Tao is based on the *Bmad* [Bma06] subroutine library. 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.

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.

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 15.

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 (§10.3).

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 (§11.26). When *Tao* starts up, the default universe is initially set to universe 1. Use the `show global` (§11.27) 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 (§4.1). 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) (§10.3). 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 (§11.26) is used to transfer information from the `design` or `model` lattices to the base lattice.

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 (§11.26). Initially, when *Tao* is started, the default branch is set to branch 0. Use the `show global` (§11.27) 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 (§10.4), 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 §10.4 and §10.5 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 `beam_start` in the *Bmad* lattice file (see the *Bmad* manual for more details on setting `beam_start`). Note: `beam_start` can be varied while running *Tao* using the `set beam_start` (§11.26) or `change beam_start` (§11.3) commands.

Chapter 4

Syntax

4.1 Element List Format

The syntax for specifying a set of lattice elements is called `element list` format. Each item of the list is one of:

Item Type	*Example*
An element name.	"5@q*"
An element index.	"23", "2»183"
A range of elements.	"b23w:67"
A class::name specification.	"sbend::b*"

Items in a list are separated by a blank character or a comma. Example:

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

An element name item is the name of an element or elements. 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. Element names may be prefixed by the universe number followed by the "" sign. If a universe is not specified, the current universe is used. Examples

```
"5@q*"        ! All elements whose name begins with "q" of universe 5.
"*@sex10w"    ! Element "sex10w" of all universes.
"b37"         ! Element "b37" of the current 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@2>>183   ! Element number 183 of branch # 2 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.
```

4.2 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 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)	Sine
`cos`(x)	Cosine
`tan`(x)	Tangent
`asin`(x)	Arc sine
`acos`(x)	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
`int(x)`	Nearest integer with magnitude less then x
`nint(x)`	Nearest integer to x
`floor(x)`	Nearest integer less than x
`ceiling(x)`	Nearset integer greater than x

Both `ran` and `ran_gauss` use a seeded random number generator. Setting the seed is described in Section §10.4.

See the following sections for the syntax for using data, variable, and lattice parameters in an expression.

4.3 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 (§4.2) 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.
  -1@data::orbit.x           ! Same as "orbit.x".
  *@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. §10.7 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. §6.8. See Sec. §4.5 for how to use "lattice parameters" that correspond to the data types given in Sec. §6.8 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.

4.4 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 (§4.2) 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. §10.6 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 varible` 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.

4.5 Specifying Lattice Parameters in Expressions

"Lattice parameters" are like `data` parameters (§4.3) 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 (§4.2) is:

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

where:

```
<universe(s)>       Optional universe specification (§3.3)
<parameter>         Name of the parameter.
<ref_ele>           Optional reference element.
<element_list>      Evaluation point or points.
<component>         Optional component.
```

correspond to the data types as listed in Sec. §6.8.

```
3@lat::orbit.x[34:37]         Array of orbits at element 34 through 37 in universe 3.
3@lat::orbit.x[34]|model      Orbit.x model value at element 34
lat::sigma.12[q10w]           Beam sigma matrix component at element q10w computed
                               from lattice parameters.
```

The list of possible lattice `<parameter>` names is given in Sec. §6.8. 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 prevenet 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`.

4.6 Specifying Beam Parameters in Expressions

Beam parameters are like lattice parameters (§4.5) 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 (§4.2) is:

```
{[<universe(s)>]@}beam::<parameter>[{<ref_ele>&}<element_list>]{|<component>}
```

where:

```
<universe(s)>      Optional universe specification (§3.3)
<parameter>        Name of the parameter
<ref_ele>          Optional reference element.
<eval_points>      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 `<parameter>` names is given in Sec. §6.8. The table 6.2 shows which data names are associated with beam tracking.

4.7 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 (§4.5). 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`
`c_mat.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 vaying 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 (§4.1) 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`.

Chapter 5

Variables

For the `model` lattice (or lattices if there are multiple `universes`) the `change` command (§11.3) can be used to vary lattice parameters such as element strengths, the initial Twiss parameters, etc. Additionally, `variables` can be defined in the *Tao* initialization files (§10.6) that can also be used to vary these `model` lattice parameters. A given *Tao* variable may control a single attribute of one element in one or more universes. There are a few reasons why one would want to setup such variables. For example, the optimizer (§8) will only work with *Tao* variables and blocks of these variables can be plotted for visual inspection.

Blocks of variables are associated with what is called a `v1_var` structure and each of these structures has a `name` with which to refer to them in *Tao* commands. For example, if `quad_k1` is the name of a `v1_var`, then `quad_k1[5]` referees to the variable with index 5 in the block.

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`.

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
```

A given variable may control a single attribute of one element 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     ! For use by extension code
  useit_opt     ! Variable is to be used for optimizing.
  useit_plot    ! Variable is to be used for plotting.
```

attrib_name
Name of the element attribute to vary. To see a list of attributes for a given element consult the *Bmad* manual or use the `show element -attributes` command.

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.

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

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 top10` command to set the variables and data which contribute most to the merit function.

merit_type
T'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 (§8.1.

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. §4.4.

Use the `show var` (§11.27) 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` (§11.33), `restore` (§11.22), and `use` (§11.31) 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_plot = exists & good_user & good_opt & good_var
```

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

If the `useit_plot` component is True, the variable is used when when plotting variables and is not used when `useit_plot` is False. `useit_plot` is set by *Tao* using the state of three other components:

```
  useit_plot = exists & good_plot & good_user
```

Chapter 6

Data

The term ``data'' denotes anything that can be calculated by *Tao*. This includes the vertical orbit at a particular position or the horizontal emittance of a storage ring. Data can be plotted or used in lattice correction and design (§8). This chapter explains how data is organized in *Tao* while Section §10.7 explains how to define the structures that hold the data in the initialization files. When running *Tao*, the `show data` (§11.27) 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

```
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.

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.

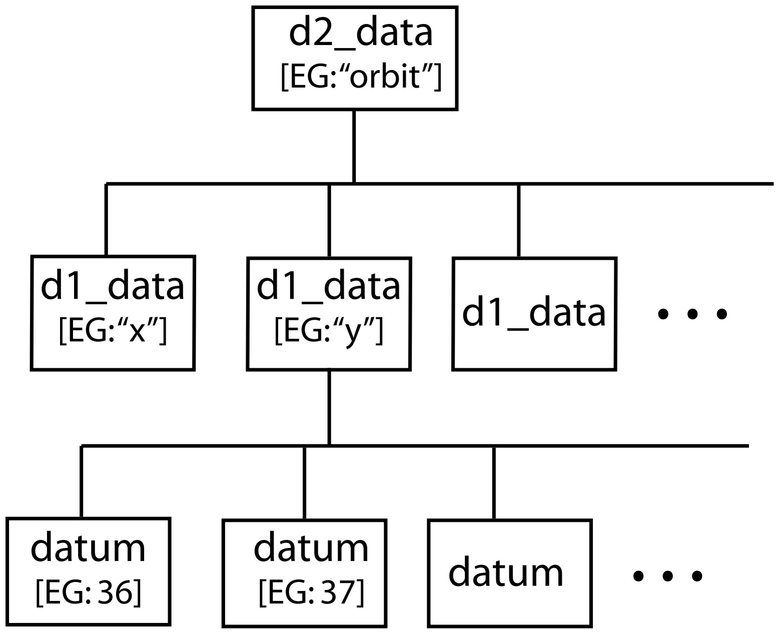


Figure 6.1: A `d2_data` structure holds a set of `d1_data` structures. A `d1_data` structure holds an array of datums.

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, then, 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.
  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: Measured datum value.
  ref               ! Real: 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.
  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.
  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: Can be used in Tao extensions.
  good_plot         ! Logical: Can be used in Tao extensions.
  useit_plot        ! Logical: Is this datum to be used in plotting?
  useit_opt         ! Logical: Is this datum to be used for optimization?
```

When running *Tao*, the `show data` (§11.27) command can be used to view the components of a datum. The `set` command (§11.26) can be used to set some of these components.

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 (§6.8). 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 (§10.7).

delta_merit
Difference used to calculate the contribution of the datum to the merit function (§8.1).

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 (§6.5). 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).

eval_point
Used with `s_offset` to determine where the datum is evaluated at (§6.4).

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 `design` value valid?

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

good_model
Set by the user. Is the `meas` value valid?

good_opt
Set by the user. Is the datum valid for optimization?

good_plot
Set by the user. Is the datum to be used in plotting?

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?

invalid
The value used for `delta_merit` if good_model = False.

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 (§8.1).

merit_type
The type of merit (§8.3).

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).

s
Longitudinal s-position of the lattice element.

s_offset
Offset of the evaluation point when there is an associated lattice element (§6.4).

useit_opt
Datum is possibly valid for optimization. `useit_opt` is set by *Tao* using the other `logicals` components. A datum is used in the optimization if both `useit_opt` and `good_meas` are true.

useit_plot
Datum is valid for plotting

weight
Weight used in evaluating the contribution of the datum to the merit function (§8.1).

6.3 Datum values

A given datum has six values associated it:

meas
The value of the datum as obtained from some measurement. This is the target or limit value that is used when running the optimizer. When doing lattice design, the measured value corresponds to a constraint value (8).

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
The reference 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.

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 is but the setting of `s_offset` is ignored.

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` §10.7) 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` (§11.33), `restore` (§11.22), and `use` (§11.31) 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 `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.

When doing optimization, the `delta_merit` component is set to the `delta` value used in computing the contribution to the merit function (§8.3). If the datum's value cannot be computed, that is, `good_model` is False, or, if the design or base values are being used in the merit calculation, `good_base` or `good_design` is False, then the `invalid` component is used for `delta_merit`.

`good_meas` is set True if the `meas` component value is set in the data initialization file (§10.7) or is set using the `set` command (§11.26). 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` (§10.4).

Finally `good_opt` is meant for use in custom versions of *Tao* (§15) and is always left True by the standard *Tao* code.

Example of using a `show data` (§11.27) 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.

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. 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 region from the exit end of the element with name `q12` to the exit end of the element with name `q45`. Notice that when a range of elements is used, a `merit_type` of `target` 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.

6.8 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 (§10.7). 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 s/he sees fit.

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 for more details.

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 for more details.

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 for more details.

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 for more details.

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 for more details.

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

c_mat.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. Old names: `chrom.dtune.a` and `chrom.dtune.b`

chrom.dbeta.a, .dbeta.b
The normalized change of the beta function with energy $(1/\beta_{a,b})\partial\beta_{a,b}/\partial\delta$. Unlike the standard chromaticities,`chrom.a` and `chrom.b`, the these chromaticities are evaluated at individual elements.

chrom.deta.a, .deta.b
The chromatic dispersion $\partial\eta_{x,y}/\partial\delta$. Unlike the standard chromaticities,`chrom.a` and `chrom.b`, the these chromaticities are evaluated at individual elements.

chrom.detap.a, .detap.b
The chromatic dispersion derivatives $\partial\eta'_{x,y}/\partial\delta$. Unlike the standard chromaticities,`chrom.a` and `chrom.b`, the these chromaticities are evaluated at individual elements.

chrom.dphi.a, .dphi.b
The chromatic betatron phase $\partial\phi_{a,b}/\partial\delta$. Unlike the standard chromaticities,`chrom.a` and `chrom.b`, the these chromaticities are evaluated at individual elements.

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

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

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.e_tot`.

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 (§10.10) 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'
  graph%n_curve = 1
  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

$$\epsilon_x = \sqrt{\widetilde{\sigma}_{xx}\,\widetilde{\sigma}_{p_x p_x} - \widetilde{\sigma}^2_{x p_x}} \tag{6.4}$$

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

$$\widetilde{\sigma}_{xx} = \langle x\,x\rangle - \frac{\langle x\,p_z\rangle\,\langle x\,p_z\rangle}{\langle p_z\,p_z\rangle} \tag{6.5}$$

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.

expression: <arithmetic_expression>

`<arithmetic_expression>` is an arithmetic expression (§4.2) 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"
```

To be valid, if an expression has a term with a `data` source, the expression must be evaluated after the `data` source components are evaluated. Data evaluation is done universe by universe starting with universe 1, then universe 2, etc. Within a given universe, the order of evaluation can be complicated but in this case a datum using an expression will always be evaluated after any datum that appears earlier in the initialization file. In the last example above, the expression terms involve an evaluation of `beta.a` in universe 2. Therefore, this expression datum should be in universe 2 or higher. Notice that while all datums must be assigned a universe, in this case, since all the terms explicitly give a universe number, the value of the datum will be independent of the universe it is in.

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
Position of the element in the global "floor" coordinate system. This is the nominal position ignoring any misalignments. See the *Bmad* manual for details on the global coordinate system. See also `rel_floor.` and `will.`.

floor.theta, .phi, .psi
Orientation of the element in the global "floor" coordinate system. This is the nominal position ignoring any misalignments. See the *Bmad* manual for details on the global coordinate system. See also `rel_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.

momentum_compaction
Momentum compaction factor. Also see `r56_compaction`.

multi_turn_orbit.x, .y, .z, .px, .py, .pz
Used for storing the orbit over many turns. Only used for plotting purposes. See §10.10.3 for more details.

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
Energy normalized apparent emittance. The normalization is the standard gamma factor:

$$\text{emit}_{\text{norm}} = \gamma \, \text{emit} \tag{6.6}$$

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

norm_emit.a, .b, .c
Energy normalized normal mode emittance. The normalization is the standard gamma factor:

$$\epsilon_{norm} = \gamma \, \epsilon \tag{6.7}$$

norm_emit.x, .y, .z
Energy normalized projected emittance. The normalization is the standard gamma factor:

$$\epsilon_{norm} = \gamma \, \epsilon \tag{6.8}$$

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:

$$M = A \circ \exp\left(: h :\right) \circ A^{-1}, \tag{6.9}$$

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,

$$M = A_1 \circ C \circ L \circ \exp\left(F \cdot \nabla\right) I \circ C^{-1} \circ A_1^{-1}. \tag{6.10}$$

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`.

orbit.amp_a, .amp_b
"Invariant" amplitude of the orbital motion.

orbit.norm_amp_a, .norm_amp_b Newline Energy normalized "invariant" amplitude of the orbital motion.

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

$$T_p \equiv (T_0 - I_4)^{-1} \tag{6.11}$$

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

$$I_4 \equiv \begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & 0 & \\ & & & & & 0 \end{pmatrix} \tag{6.12}$$

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.

orbit.e_tot

The `orbit.e_tot` 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.x, .y, .z, .px, .py, .pz

Orbit position and momenta

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 between the a and b normal modes. $-\pi < d\phi_{\text{frac}} < \pi$

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, .cos_y, .sin_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 `a`-mode response can be characterized by the equation

$$\begin{aligned} x_a(n) &= A_{ax} \sin(n\,\omega_a + \phi_{ax} + \phi_{a0}) \\ y_a(n) &= A_{ay} \sin(n\,\omega_a + \phi_{ay} + \phi_{a0}) \end{aligned} \tag{6.13}$$

where ω_a is the a-mode tune, $x_a(n)$ and $y_a(n)$ are the horizontal and vertical positions of the `a`-mode component on the nth turn. A_{ax} and A_{ay} are the response amplitudes, f_a is the `a`-mode frequency ϕ_{ax} and ϕ_{ay} are the oscillation 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.sin_y`	$\longrightarrow$	$A_{ay} \sin \phi_{ay}$
`ping_a.cos_y`	$\longrightarrow$	$A_{ay} \cos \phi_{ay}$

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. (7) of reference [Sag00a]

$$\begin{aligned} x_a(n) &= A_a \sqrt{\beta_a} \cos(n\omega_a), \\ y_a(n) &= -A_a \sqrt{\beta_b} \Big(\overline{C}_{22} \cos(n\omega_a) + \overline{C}_{12} \sin(n\omega_a) \Big) \end{aligned} \tag{6.14}$$

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. Since the design coupling in many machines is zero or very small, in such cases the `ping_a.sin_y` and `ping_a.cos_y` datums are better for analysis as opposed to the `ping_a.amp_y` and `ping_a.phase_y` datums since the value of `ping_a.phase_y` is meaningless when the local coupling, and hence `ping_a.amp_y`, is zero.

For the `design` and `model` values of a datum, Eq. (6.14) 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:

$$C_m \sum \text{ping\_a.amp\_x}_{\text{meas}} = \sum \text{ping\_a.amp\_x}_{\text{model}} \tag{6.15}$$

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, .sin_x, .cos_x

Similar to `ping_a` except this is for the `b`-mode component of the response. Here the `design` and `model` values are calculated from Eq. (8) of reference [Sag00a]:

$$\begin{aligned} x_b(n) &= A_b \sqrt{\beta_a} \Big(\overline{C}_{11} \cos(n\omega_b) - \overline{C}_{12} \sin(n\omega_b) \Big), \\ y_b(n) &= A_b \sqrt{\beta_b} \cos(n\omega_b). \end{aligned} \tag{6.16}$$

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*

Terms of the linear transfer map. $1 \le i, j \le 6$.

rad_int.i1, .i2, .i2_e4, .i3, .i3_e7, .i4a, .i4b, .i4z, .i5a, .i5b, .i5a_e6, .i5b_e6
rad_int1.i1, .i2, .i2_e4, .i3, .i3_e7, .i4a, .i4b, .i4z, .i5a, .i5b, .i5a_e6, .i5b_e6

Synchrotron radiation integrals. See the *Bmad* manual for details. The `rad_int1.xxx` datums are the radiation integrals over a single element. With the `rad_int.xxx` datums, the integral is from `ele_ref` to `ele`.

```
    .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
```

r56_compaction

This datum is defined to be

$$M_{5,6} + \sum_{i=1}^{4} M_{5,i}\,\eta_i \tag{6.17}$$

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.

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.`.

sigma.x, .y, .z, .px, .py, .pz, .ij, .Lxy

Beam sizes `sigma.x`, `sigma.px`, etc., and angular momentum `sigma.Lxy` $= < xp_x - yp_x >$ are calculated from the beam sigma matrix if the `data_source` is set to `beam` and calculated from the linear Twiss parameters if the `data_source` is set to `lat`.

`sigma.`ij with $1 \le i, j, k, \ldots \le 6$ are components of the beam sigma matrix.

Irregardless of the setting of `data_source`, the beam emittance and longitudinal sigma values will be taken from the `beam_init` structure (§10.5) and *not* any emittances or longitudinal sigma values specified in the lattice file.

t.ijk, tt.$ijklm\ldots$

Taylor map components between two points with $1 \le i, j, k, \ldots \le 6$. The difference between `t.`ijk and `tt.`$ijklm\ldots$ is that `t.`ijk is restricted to exactly three indices and `tt.`$ijklm\ldots$ is not. `t.`ijk is superfluous but is keep for backwards compatibility.

Calculation of `t.`ijk 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_step` attribute of that element (see the *Bmad* manual for more details). When a smooth curve (§10.10.2) 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_step`. 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
Tune in radians.

unstable.orbit
The `unstable.orbit` datum is used for linear lattices in an optimization to avoid unstable solutions (§8.3).

For single particle tracking, the value of an `unstable.orbit` datum is zero if the tracked particle survives (has not been lost) up to the evaluation element and, if it has been lost, is set to

$$1 + i_{\text{ele}} - i_{\text{lost}} + \frac{1}{2}\left[\tanh\left(\frac{r_{orbit}}{r_{lim}} - 1\right) - E\right] \tag{6.18}$$

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

$$E = \begin{cases} 1 & \text{if the particle is lost at the exit end of the element.} \\ 0 & \text{if the particle is lost at the entrance end of the element.} \end{cases} \tag{6.19}$$

In the abouve 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 choisen so that the datum value will be monotonic with increasing stability.

The default for the evaluation element, if `ele_name` nor `ix_ele` is not specified, is to use the last element in the lattice.

When tracking beams, the value of `unstable.orbit` is the averaged value over all particles in the bunch.

unstable.ring
`unstable.ring` is used for storage rings. The value of an `unstable.ring` datum is zero if the ring is stable and set to the largest growth rate of all the normal modes of oscillation if the ring is unstable. `unstable.ring` is used in an optimization to avoid unstable solutions (§8.3).

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 (§10.8). 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. $\widetilde{z}$ is the projection of the local z-axis onto the (Z, X) plane and $\widetilde{x}$ is the coordinate in the (Z, X) plane perpendicular to $\widetilde{z}$. In the typical situation, where a machine is planer (no out-of-plane bends), the $\widetilde{z}$-axis corresponds to the local z-axis and the $\widetilde{x}$-axis corresponds to the 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 $\widetilde{x}$ axis (has $\widetilde{z} = 0$) as shown in Figure 6.2.

By definition, the ``` ``left side'' ``` of the machine corresponds to be the $+\widetilde{x}$ side and the ``` ``right side'' ``` corresponds to be the $-\widetilde{x}$ side. That is, left and right are relative to someone looking in

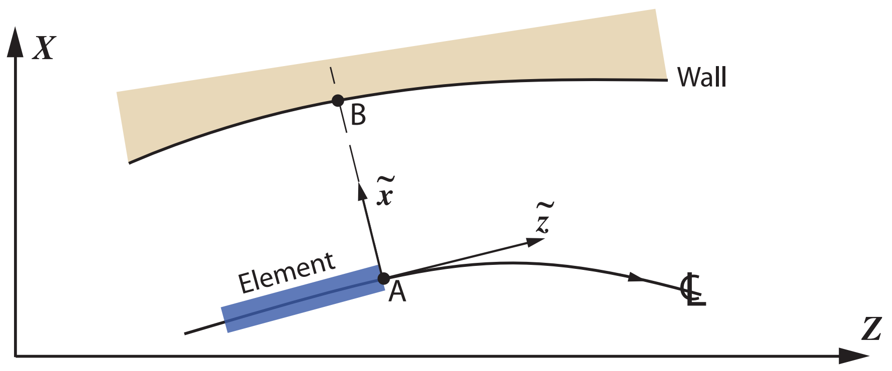


Figure 6.2: A `wall.` datum is a measure of the distance between the centerline of a machine and the walls of the containment building.

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. 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 $\tilde{z} = 0$, the datum value is set to a large number.

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

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.

Table 6.2: Predefined Data Types in Tao

Data_ Type	*Description*	`data_source`	`Can use s_offset?`
`alpha.a, .b`	`Normal-Mode alpha function`	`lat`	`Yes`
`apparent_emit.x, .y`	`Apparent emittance`	`beam, lat`	`No`
`beta.a, .b, .c`	`Normal-mode beta function`	`beam, lat`	`Yes`
`beta.x, .y, .z`	`Projected beta function`	`beam, lat`	`No`
`bpm_cbar.22a, .12a, .11b, .12b`	`Measured coupling`	`lat`	`Yes`
`bpm_eta.x, .y`	`Measured dispersion`	`lat`	`Yes`
`bpm_orbit.x, .y`	`Measured orbit`	`lat`	`Yes`
`bpm_phase.a, .b`	`Measured betatron phase`	`lat`	`Yes`
`bpm_k.22a, .12a, .11b, .12b`	`Measured coupling`	`lat`	`Yes`
`bunch_max.x, .px, .y, .py, .z, .pz`	`Max relative to centroid`	`beam`	`No`
`bunch_min.x, .px, .y, .py, .z, .pz`	`Min relative to centroid`	`beam`	`No`
`c_mat.11, .12, .21, .22`	`Coupling`	`lat`	`Yes`
`cbar.11, .12, .21, .22`	`Coupling`	`lat`	`Yes`
`chrom.a, .b`	`Chromaticities for a ring`	`lat`	`No`

Table 6.2: (continued)

Data_ Type	*Description*	data_source	Can use s_offset?
chrom.dbeta.a, .dbeta.b	Normalized Chromatic beta	lat	No
chrom.deta.x, .deta.y	Chromatic dispersions	lat	No
chrom.detap.x, .detap.y	Chromatic dispersion slopes	lat	No
chrom.dphi.a, .dphi.b	Chromatic betatron phase	lat	No
damp.j_a, .j_b, .j_z	Damping partition number	lat	No
dpx_dx, dpx_dy, etc.	Bunch <x px> / <x^2> & Etc...	beam	No
e_tot_ref	Lattice reference energy (eV)	lat	No
element_attrib.<attrib_name>	lattice element attribute	lat	No
emit.a, .b, .c	Emittance	beam, lat	No
eta.x, .y, .z	Lab Frame dispersion	beam, lat	Yes
eta.a, .b	Normal-mode dispersion	beam, lat	Yes
etap.x, .y	Lab Frame dispersion derivative	beam, lat	Yes
etap.a, .b	a & b-mode dispersion derivative	beam, lat	Yes
expression:<expression>	See text above	lat	No
floor.x, .y, .z	Global (“floor”) position	lat	Yes
floor.theta, .phi, .psi	Global (“floor”) orientation	lat	Yes
gamma.a, .b	Normal-mode gamma function	lat	Yes
k.11b, .12a, .12b, .22a	Coupling	lat	Yes
momentum	Momentum: P*C_light (eV)	lat	Yes
momentum_compaction	Momentum compaction factor	lat	No
multi_turn_orbit.x, .y, .z multi_turn_orbit.px, .py, .pz	Store orbit over many turns	lat	No
n_particle_loss	Number of particles lost	beam	No
norm_apparent_emit.x, .y	Normalized apparent emittance	beam, lat	No
norm_emit.a, .b, .c	Normalized beam emittance	beam, lat	No
norm_emit.x, .y, .z	Normalized projected emittance	beam, lat	No
normal.<type>.i.<monomial>	Normal Form map component	lat	No
orbit.e_tot	Beam energy (eV)	beam, lat	Yes
orbit.x, .y, .z	Orbit position	beam, lat	Yes
orbit.px, .py, .pz	Orbit Momenta	beam, lat	Yes
orbit.amp_a, .amp_b	Orbit amplitude	lat	Yes
orbit.norm_amp_a, .norm_amp_b	Energy normalized amplitude	lat	Yes
periodic.tt.$ijklm\ldots$ $1 \le i, j, k, \ldots \le 6$	Taylor term of the periodic map	lat	No
phase.a, .b	Betatron phase	lat	Yes
phase_frac.a, .b	Fractional betatron phase $-\pi < \phi_{\texttt{frac}} < \pi$	lat	No
phase_frac_diff	Phase diff between a and b modes	lat	No
photon.intensity	Photon total intensity	beam, lat	No
photon.intensity_x, photon.intensity_y	Photon intensity components	beam, lat	No
photon.phase_x, .phase_y	Photon phase	beam, lat	No
ping_a.amp_x, .phase_x, ping_a.amp_y, .phase_y	Amp & phase of a-mode response	lat	No
ping_b.amp_x, .phase_x ping_b.amp_y, .phase_y	Amp & phase of b-mode response	lat	No
r.ij $1 \le i, j \le 6$	Term in linear transfer map	lat	No
r56_compaction	R56 like compaction factor.	lat	No
rad_int.i1, .i2, etc.	Lattice Radiation integrals	lat	No
rad_int1.i1, .i2, etc.	Element radiation integrals	lat	No

Table 6.2: (continued)

Data_ Type	*Description*	data_source	Can use s_offset?
ref_time	Reference time	beam, lat	Yes
rel_floor.x, .y, .z, .theta	Relative global floor position	lat	No
s_position	longitudinal length constraint	lat	Yes
sigma.x, .y, .z sigma.px, px, .pz sigma.ij $1 \le i, j \le 6$, sigma.Lxy	Bunch size	beam, lat	No
spin.amp, .theta, .phi spin.x, .y, .z	Particle spin	beam, lat	No
time	Particle time (sec)	beam, lat	Yes
t.ijk $1 \le i, j, k \le 6$	Term in 2^{nd} order transfer map	lat	No
tt.$ijklm \ldots$	Term in n^{th} order transfer map	lat	No
tune.a, .b	Tune	lat	No
unstable.orbit	Nonzero if particles are lost in tracking	lat	No
unstable.ring	Nonzero if a ring is unstable	lat	No
wall.left_side, .right_side	Building wall constraint	lat	No
wire.<angle>	Wire scanner at <angle>	beam	No

Chapter 7

Plotting

Some definitions:

Curve
: A `curve` is a set of (x,y) points to be plotted.

Graph
: A `graph` consists of horizontal and vertical axes along with a set of `curve`s that are plotted within the graph.

Plot
: A `plot` is essentially a collection of `graphs`.

Page
: The `page` refers to the X11 window where graphics are displayed or the corresponding printed graphics page.

Region
: The `page` is divided up into a number of rectangles called `regions`. `Regions` may overlap.

The plot initialization file (cf. Chapter 10) defines a set of `template plots`. A `template` defines what type of data is to be plotted (orbit, beta function, etc.), how many `graphs` there are, what the scales are for the `graph` axes, how the `graph`s are laid out, etc. The plot initialization file also defines a set of `region`s within the `page`. Any `template plot` can be placed in any region. Using the `place` command (see Chapter 11 for a full descriptions of all commands) one can assign a particular `template plot` to a particular region for plotting. The relationship between `region`, `plot`, `graph`, and `curve` is shown graphically in Figure 7.1.

Figures 7.2 and 7.3 show examples of a plot `page`. Figure 7.2 was generated by defining two regions called `top` and `bottom` in the plot initialization file. The `top` region was defined to cover the upper half of the `page` and the `bottom` region was defined to cover the bottom half. `Template plots` were defined to plot phase and orbit data from a defined set of detector elements in the lattice. Each `template plot` defined two graphs which in both cases where assigned the names `x` and `y`. The orbit `template plot` was placed in the `top` region and the phase `template plot` was placed in the `bottom` region. The horizontal axis numbering is by detector `index`. Displayed plots are referred to by the `region` name (`top` and `bottom` in this case). Individual graphs and curves are referred to using the nomenclature `region.graph.curve`. Thus, in this example, the horizontal orbit graph would be referred to as `top.x`. Using the `set plot`, `set graph`, or `set curve` commands (§11.26) one can then specify what

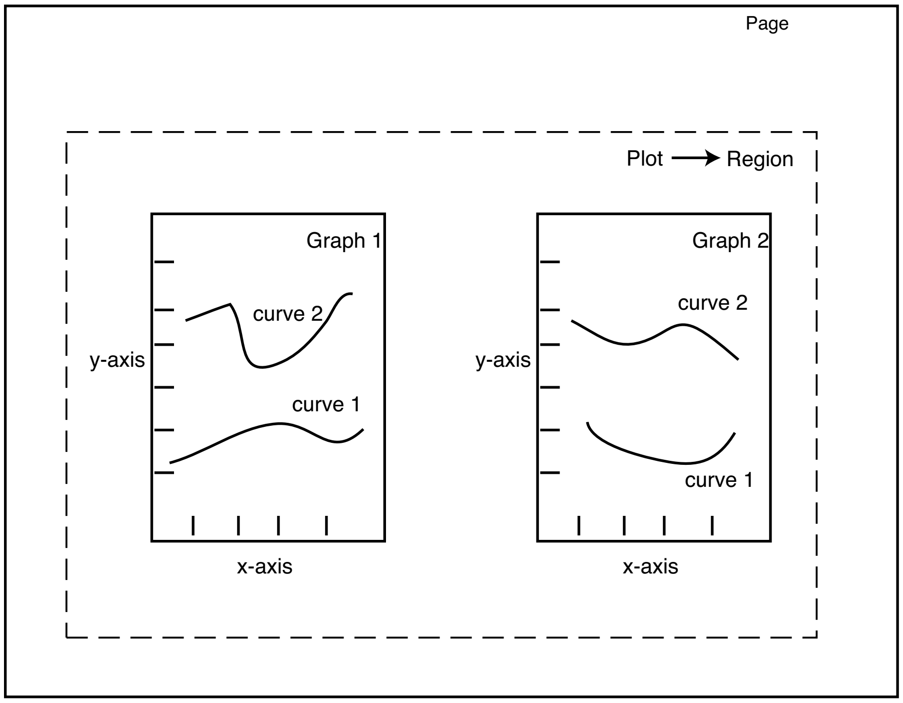


Figure 7.1: A plot has a collection of graphs and a graph has a collection of curves. A plot becomes visible when it is associated with some region on the page using the `place` command. Note that on the actual page the plot/region border is not visible.

`components` are plotted. "`component`" refers to `measured`, `reference`, `model`, `base`, and/or `design` data (§10.10.3). Notice that the same `template plot` can be assigned to different `regions` and the plots in different `regions` can have different scales for their axes or different `components`. In the example in Figure 7.2, the `component` for the `top` plot is `model` and for the `bottom` plot it is `model - design`.

Plots may be referred to by their template name or by the name of the region they are placed in. For example, the orbit plot in Figure 7.2 may be referred to using the region name (`top`) or the template name (`orbit`). A template may be placed in multiple regions. For example, you may wish to plot the `model` data for the orbit in one region and the `design` data for the orbit in another region. In this case the command `scale orbit` would scale the plots in both regions while to scale the plot in only one of the regions you would need to use the region name.

A graph of a plot is specified using the format `plot_name.graph_name` where `plot_name` is a template or region name and `graph_name` is the name of the graph. For example, if the horizontal orbit graph of the `orbit` plot is named `x` then it would be referred to as `orbit.x` or `top.x`. If a plot has only one graph, the graph may be specified by just using the plot name.

A curve within a graph is specified using the format `plot_name.graph_name.curve_name`. If a graph has only one curve, the curve may be specified using only the graph name `plot_name.graph_name`. Additionally, if the there is only one curve in a plot, the curve can be specified by just using the `plot_name`.

The `use`, `veto`, `restore`, and `clip` commands are used to control what data is used in fitting the model to the data in the optimization process (see Chapter 8). The general rule is that these commands only affect measured and reference data. If plotting `model`, `design` and/or `base` data then the data will be displayed irregardless. If plotting meas and/or ref data then the data displayed will vary with these commands. meas or ref data vetoed for display is also vetoed for fitting. However, measured data that is off the vertical or horizontal scale may still be used by the optimizer unless vetoed with the veto or clip command. If there are data points off the vertical scale then "**Limited**" will appear in the upper right-hand corner of the graph. If plotting measured data then these points off scale will still be used by the optimizer.

The x_axis and x_scale commands are used to set the axis type and scale for each graph. The axis type can be either index, ele_index or s which corresponds to the data index number, element index number and longitudinal position in the lattice (from element 0) respectively.

Figure 7.3 shows another example of a plot page. In this case the page was generated by again defining two vertically stacked regions but in this case the regions have different heights. A template plot with a single graph was placed in the bottom most region. This graph contains a key_table. A key_table is used in conjunction with single mode and is explained in Chapter §12. A template plot containing five graphs was placed in the uppermost region. The uppermost graph of this template plot contains a lat_layout which shows the placement of lattice elements. What elements are displayed in a lat_layout and what shapes they are represented by is specified in the initialization file. The horizontal scale is longitudinal position (s). The remaining four graphs show dispersion and beta data from two different universes representing the low energy and high energy transport in an energy recovery linac. The individual data points here (hard to see in this example) have been slaved to the lat_layout and represent the beta and dispersion at the edges of the displayed elements in the lat_layout.

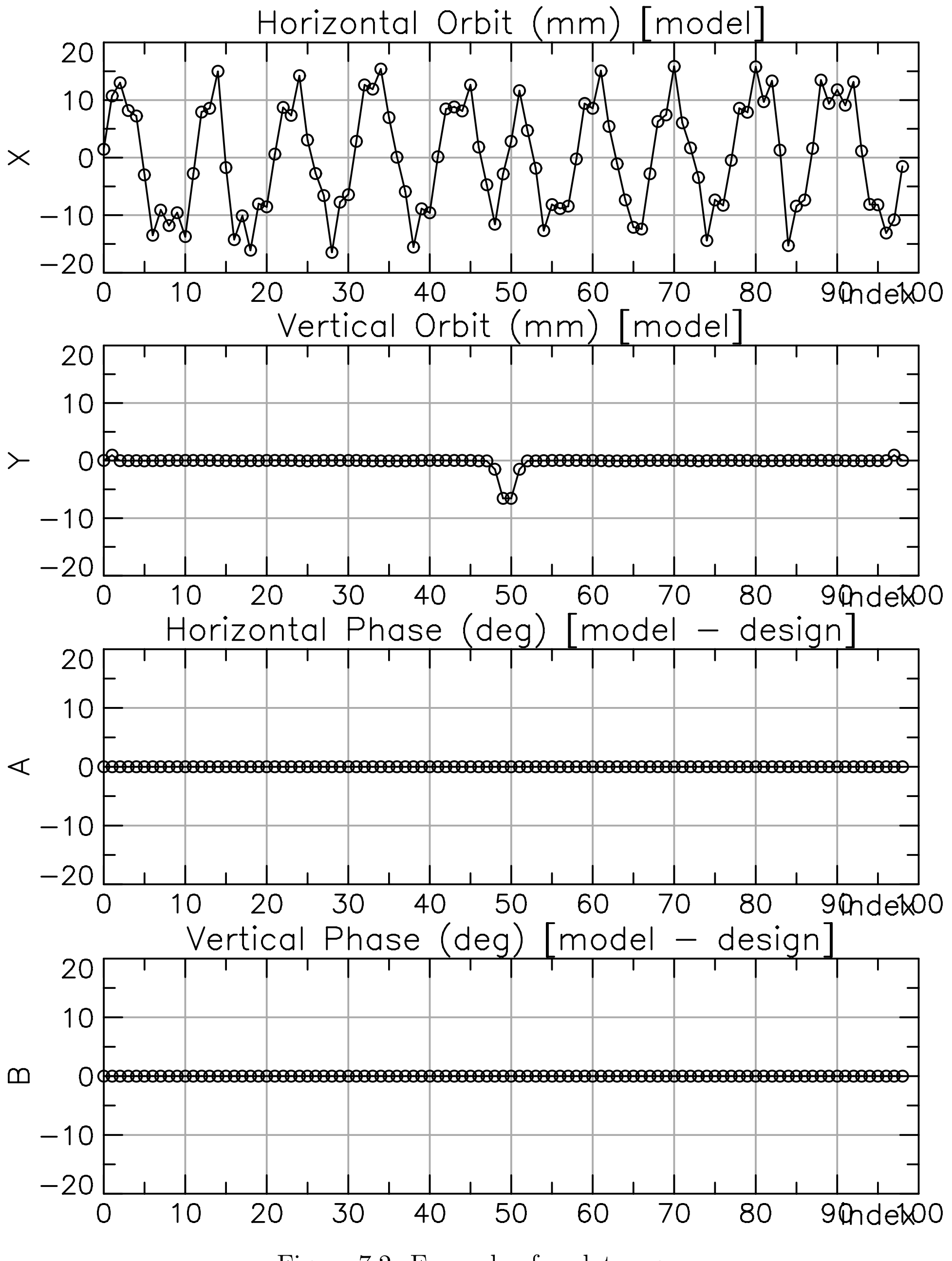
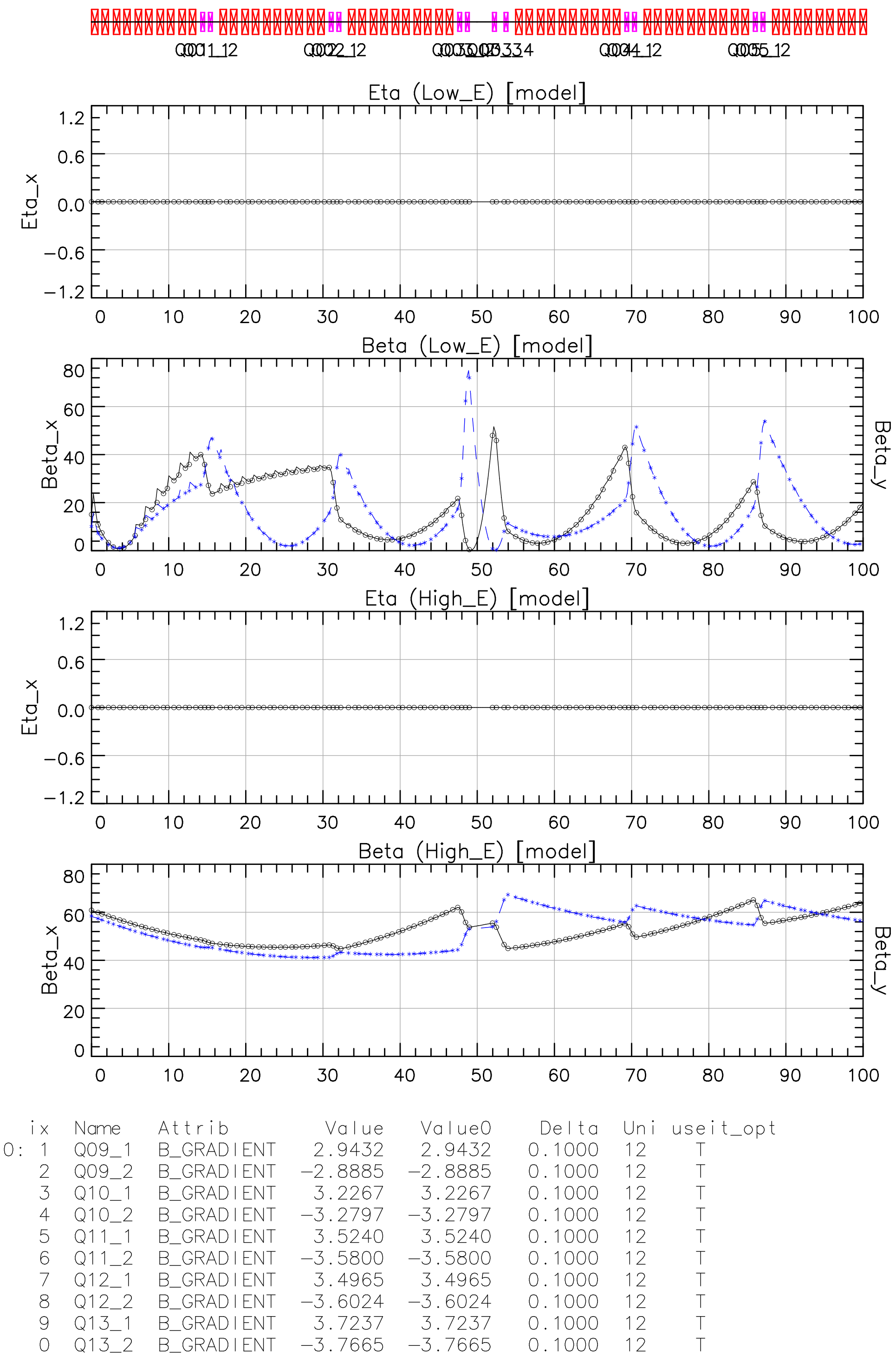


Figure 7.2: Example of a plot page



```
  ix  Name    Attrib        Value    Value0    Delta  Uni  useit_opt
0: 1  Q09_1   B_GRADIENT    2.9432    2.9432   0.1000  12      T
   2  Q09_2   B_GRADIENT   -2.8885   -2.8885   0.1000  12      T
   3  Q10_1   B_GRADIENT    3.2267    3.2267   0.1000  12      T
   4  Q10_2   B_GRADIENT   -3.2797   -3.2797   0.1000  12      T
   5  Q11_1   B_GRADIENT    3.5240    3.5240   0.1000  12      T
   6  Q11_2   B_GRADIENT   -3.5800   -3.5800   0.1000  12      T
   7  Q12_1   B_GRADIENT    3.4965    3.4965   0.1000  12      T
   8  Q12_2   B_GRADIENT   -3.6024   -3.6024   0.1000  12      T
   9  Q13_1   B_GRADIENT    3.7237    3.7237   0.1000  12      T
   0  Q13_2   B_GRADIENT   -3.7665   -3.7665   0.1000  12      T
```

Figure 7.3: Another example of a plot page.

Chapter 8

Optimization: Lattice Correction and Design

This chapter covers the process of `optimization` which involves minimization of a `Merit Function`. Optimization can be used to correct or to design lattices. Examples of `lattice corrections` include flattening the orbit and adjusting quadrupoles to correct the measured betatron phase. `Lattice design` involves creating a lattice that conforms to a set of desirable properties. For example, requiring that the beta function in a certain region never exceeds a given value. In this chapter, Section §8.1 presents the merit function in the context of lattice corrections while Section §8.2 discuses the merit function in the context of lattice design. Since the concepts used in `lattice corrections` and `lattice design` are similar, *Tao* combines the two into one generalized process as discussed in Section §8.3.

8.1 Lattice Corrections

Consider the problem of problem of modifying the orbit of a beam through a lattice to conform to some desired orbit (typically a "flat" orbit running through the centers of the quadrupoles). The process generally goes through three stages: First the orbit is measured, then corrections to the steering elements are calculated and finally the corrections are applied to the machine. Since these are necessarily machine specific, *Tao* has no specific routines to measure orbits or to load steering corrections but they could be implemented with some custom coding as discussed in Chapter §15. What *Tao* does, however, is to implement a generalized algorithm procedure for minimizing a `merit function` which can be used to calculate the corrections. The idea is to vary a set of variables (steerings in the case of an orbit correction) within the `model` lattice (§3.3) with the aim to make the `measured data` (position data for an orbit correction) correspond to the values as calculated from the `model` lattice. Once the model lattice the `model` and `measured` data agree, the difference between the `model`, which represents the state of the machine when the measurement is made, and the `design`, which represents the desired state of the machine, is used to calculate corrections. In the case of flattening an orbit, the difference between the `model` steering strengths and the `design` steering strengths (typically the `design` steering strengths are zero) is what the real steerings need to be changed by to flatten the orbit.

The merit function `M` that is a measure of how well the data as calculated from the `model`, fits the measured data. *Tao* uses a merit function of the form

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

where

$$\begin{aligned} \delta D &= \text{data\_model} - \text{data\_meas} \\ \delta V &= \text{var\_model} - \text{var\_meas} \end{aligned} \tag{8.2}$$

`data_model` is the data as calculated from the `model` and `data_meas` is the measured data. `var_model` is the value of a variable in the `model` and `var_meas` is the value as measured at the time the data was taken (for example, by measuring the current through a steering and using a calibration factor to calculate the kick) and the sum `j` runs over all variables that are allowed to be varied to minimize `M`. The second term in the merit function prevents degeneracies (or near degeneracies) in the problem which would allow *Tao* to find solutions where `data_model` matches `data_measured` with the `var_model` having "unphysical" values (values far from `var_meas`). The weights w_i and w_j need to be set depending upon how accurate the measured data is relative to how accurate the calibrations for measuring the `var_meas` values are. With the second term in the merit function, the number of constraints (number of terms in the merit function) is always larger than the number of variables and degeneracies can never occur.

In a correction one wants to change the machine variables so that the measured data corresponds to the design values `data_design`. Thus the change in the data that one wants is

```
  data_change = data_design - data_meas
```

Once a fit has been made, and presuming that the `data_model` is reasonably close to the `data_meas` this data change within the `model` lattice can be accomplished by changing the variables by

```
  var_change = var_design - var_model
```

This assumes the system is linear. For many situations this is true since typically `var_change` is "small". Since the variables have a measured value of `var_meas` the value that the variables should be set to is

```
  var_final = var_meas + (var_design - var_model)
```

Notice that the fitting process is independent of the `design` lattice. It is only when calculating the corrections to the variables that the `design` lattice plays a role.

Sometimes it is desired to fit to changes in data as opposed to the absolute value of the data. For example, when closing an orbit bump knob what is important is the difference in orbits before and after the bump knob is varied. Designating one of these orbit the `reference`, the appropriate deltas to be used in Eq. (8.1) are

$$\begin{aligned} \delta D &= (\text{data\_model} - \text{data\_design}) - (\text{data\_meas} - \text{data\_ref}) \\ \delta V &= (\text{var\_model} - \text{var\_design}) - (\text{var\_meas} - \text{var\_ref}) \end{aligned} \tag{8.3}$$

where `data_ref` and `var_ref` refer to the reference measurement. These deltas are acceptable if the reference data is taken with the machine reasonably near the design setup so that nonlinearities can be ignored. If this is not the case then the fitting becomes a two step process: The first step is to fit the `model` to the `reference` data using the deltas of Eq. (8.2). The `base` lattice is then set equal to the `model` lattice. The second step is to fit the model using the deltas

$$\begin{aligned} \delta D &= (\text{data\_model} - \text{data\_base}) - (\text{data\_meas} - \text{data\_ref}) \\ \delta V &= (\text{var\_model} - \text{var\_base}) - (\text{var\_meas} - \text{var\_ref}) \end{aligned} \tag{8.4}$$

Control of what data and what variables are to be used in the fitting process is controlled by the `use`, `veto`, `restore`, and `clip` commands.

8.2 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. In this case we can proceed as was done for lattice corrections and use Eq. (8.1). In this case, the deltas are computed to limit values to some range so a typical delta would be of the form

$$\delta D \text{ or } \delta V = \begin{cases} \text{model} - \text{limit} & \text{model} > \text{Limit} \\ 0 & \text{otherwise} \end{cases} \tag{8.5}$$

or a constraint is used to keep the `model` at a certain value so the form of the constraint would be

$$\delta D \text{ or } \delta V = \text{model} - \text{target} \tag{8.6}$$

Here `model` is the value as calculated from the `model` lattice. `target` and `limit` are given numbers. Part of the optimization process is in deciding what the values should be for any `target` or `limit`.

8.3 Generalized Design

The form of the deltas used in the merit function is determined by two global logicals called `opt_with_ref` and `opt_with_base` (§10.4) as shown in Table 8.1. An exception occurs when using a `common base lattice` (§8.7). In this case, the common universe does not have base or reference values associated with it. Thus all data and variables that are associated with the common universe calculate their `delta` as if both `opt_with_ref` and `opt_with_base` were set to `False`.

Opt_with_ref	Opt_with_base	delta
F	F	model - meas
T	F	model - meas + ref - design
F	T	model - meas - base
T	T	model - meas + ref - base

Table 8.1: The form of `delta`

Another exception occurs with data when the datum value cannot be computed (§6.5). In this case, the datum's `invalid` value is used for the `delta`. This is useful, for example, in a linear lattice when the particle trajectory results in the particle being lost.

The `Non-Zero-Condition` needed for a non–zero D_i is dependent upon the `merit_type` of the datum (§6.2). There are five `merit_type` constraint types as given in Table 8.2.

Merit_ Type	*Non-zero-Condition*
target	Any delta
min, abs_min	delta < 0
max, abs_max	delta > 0

Table 8.2: Constraint Type List.

For variables, the form of the terms V_i is determined by its `merit_type`. Here the `merit_type` may be:

```
target
limit
```

A `target` `merit_type` for a variable is the same as for datum. In this case `model` is just the value of the variable. A `limit` `merit_type` has the form

$$\delta V = \begin{cases} \text{model} - \text{high\_lim} & \text{model} > \text{high\_lim} \\ \text{model} - \text{low\_lim} & \text{model} < \text{low\_lim} \\ 0 & \text{Otherwise} \end{cases} \tag{8.7}$$

The default `merit_type` for a variable is `limit`.

Note: when doing lattice design `opt_with_ref` and `opt_with_base` are both set to `False` and the `target` and `limit` values are identified with `Meas`.

When optimizing a storage ring, If the ring is unstable so that the twiss parameters, closed orbit, etc. cannot be computed, the contribution to the merit function from the corresponding datums is set to zero. This tends to lower the merit function and in this case an optimizer will never leave the unstable region. To avoid this, an `unstable_ring` constraint (§6.8) must be set.

To see a list of constraints when running *Tao* use the `show constraints` command (§11.27). To see how a particular variable or datum is doing use the `show data` or `show variable` commands. See §6.5 for details on how datums are chosen to be included in an optimization.

8.4 Variable Limits and Optimization

High (`high_lim`) and low (`low_lim`) limiting values can be set for any variable (§10.6). If not explicitly set, `high_lim` defaults to 10^30 and `low_lim` defaults to -10^30. When running the optimizer, if the (model) value of a variable is outside of the range set by the limits, the value will be set to the value of the appropriate limit and the variable's `good_user` parameter (§5) is set to False so that no further variation by the optimizer is done.

If the parameter `global%var_limits_on` (§10.4) is set to `False`, limit settings are ignored.

By default, any variable value outside of the limit range will reset. Even those variables that are not varied by the optimizer. If this behavior is not desired, the 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.

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`.

8.5 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 (§10.4) 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 (§10.4).

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` (§10.4). 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 as implemented in `Numerical Recipes`[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 (§10.6). 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 (§10.4). 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.

`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` 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 (§11.26). 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 §10.4 for more details.

`svd`
: The `svd` optimizer uses a singular value decomposition calculation. See the description of `svdfit` from Numerical Recipes[NR92] for more details. 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 sigular. See the documentation for the Numerical Recipes routine `svdfit` for more details.

8.6 Optimization Troubleshooting Tips

Optimizations can behave in strange ways. Here are some tips on how to diagnose problems.

The `show optimizer` (§11.27.20) command will show global parameters associated with optimizations. This will show some of the parameters that can be varied to get better convergence. One quick thing to do is to increase the number of optimization loops and/or optimization cycles:

```
set global n_opti_loops = ...
set global n_opti_cycles = ...
```

One of the first things to check is the merit function, the top contributors can be seen with the command `show top10` (§11.27.24). And individual contributions can be viewed using the `show variable` and `show data` commands.

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 roundoff 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 top10 -derivative` (§11.27.24) or the `show derivative` (§11.27.8) command.

8.7 Common Base Lattice (CBL) Analysis

Some data analysis problems involve varying variables in a both the `model` and `base` lattices simultaneously. Such is the case with Orbit Response Matrix (`ORM`) analysis[Saf97]. With `ORM`, the analysis starts with a set of difference orbits. A given difference orbit is generated by varying a given steering by a known amount and the steering varied is different for different difference orbits. Typically, The number N of difference orbits is equal to the number of steering elements in the machine. In *Tao*, this will result in the creation of N universes, one for each difference measurement. The `model` lattice in a universe will correspond to the machine with the corresponding steering set to what it was when the data was taken. Conversely, the `base` lattices in all the universes all correspond to the common condition without any steering variation.

In *Tao*, this arrangement is called `Common Base Lattice` (`CBL`) analysis. To do a CBL analysis, the `common_lattice` switch must be set at initialization time (§10.3). With `CBL`, *Tao* will set up a `‘‘common’’` universe with index 0. The `model` lattice of this common universe will be used as the `base` lattice for all universes.

The variables (fit parameters) in a `CBL` analysis can be divided into two classes. One class consists of the parameters that were varied to get the data of the different universes. With `ORM`, these are the steering strengths. At initialization (§10.6), variables must be set up that control these parameters. A single variable will control that particular parameter in a particular universe, that was varied to create the data for that universe.

The second class of variables consists of everything that is to be varied in the common base lattice. With `ORM`, this generally will include such things as quadrupole and BPM error tilts, etc. That is, parameters that did *not* change during data taking. The *Tao* variables that are created for these parameters will control parameters of the `model` lattice in the common universe.

To cut down on memory usage when using `CBL` (the number of data sets, hence the number of universes, can be very large), *Tao* does not, except for the common `model` lattice, reserve separate memory for each `model` lattice. Rather, it reserves memory for a single ``working'' lattice and the `model` lattice for a particular universe is created by first copying the common `base` lattice to the `working` lattice and then applying the variable(s) (a steering in the case of `ORM`) appropriate for that universe. As a result, except for the common `model` lattice, it is not possible to vary a parameter of a `model` lattice unless that parameter has a *Tao* variable that associated with it. The `change` command (§11.3) is thus restricted to always vary parameters in the common `model` lattice.

With `CBL`, the `opt_with_base` and `opt_with_ref` (§8.3) global logicals are generally set to True. Since `opt_with_base`, and `opt_with_ref` do not make sense when applied to the data in the common universe, The contribution to the merit function from data in this universe is always calculated as if `opt_with_base` and `opt_with_ref` were set to False.

With `opt_with_base` set to True, the `base` value for a datum is evaluated by looking for a corresponding datum in the common universe and using its `model` value. To simplify the bookkeeping, it is assumed that the structure of the data arrays is identical from universe to universe. That is, the `show data` command gives identical results independent of the default universe.

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`	`Quadrupolar errors`
`Beta function`	`Quadrupolar errors`
`Coupling`	`Skew quadrupolar errors`
`Dispersion`	`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 the beta function analysis that is presented.

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,

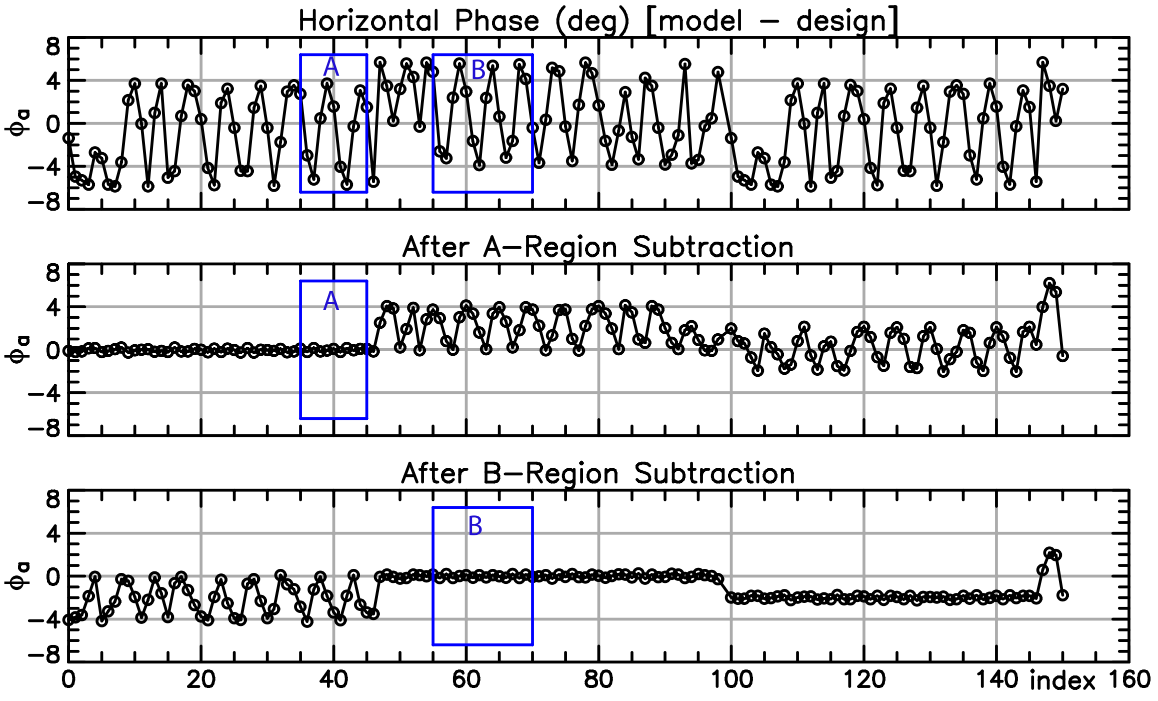


Figure 9.1: Example wave analysis for betatron phase data.

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 data9.1. 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.

9.2.1 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’’` (§10.7). 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 (§10.7):

```
  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
```

9.2.2 Wave Analysis Commands and Output

The `wave` command (§11.34) sets which plotted data curve is used for the wave analysis. The `set wave` command (§11.26) is used for setting the A and B region locations. Finally the `show wave` command (§11.27) 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

Tao 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 the "standard" variables should be.
* What the "standard" data is.
* What to plot and where to plot it.
```

Example initialization files can be found in the *Tao* distribution in the directory

```
tao/examples
```

10.1 Format

Initialization parameters are read in from a 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:

```
&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. Notice the notation `n*number` does not denote multiplication but instead can be used to denote multiple values. Also note that the compiler may be picky about blanks so that "2*34" will be accepted but "2 * 34" may not.

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.

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? Without going into detail, string variables are used in place of logical variables when *Tao* needs to know if the variable has been explicitly set.

10.2 Beginning Initialization

The initialization starts with the *Tao* initialization file. The default name for this file is `tao.init` (See §2.1). The first namelist block read in from this 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 initialization file.
    building_wall_file = "<file_name>"  ! No Default.
    data_file          = "<file_name>"  ! Default = Tao initialization file.
    var_file           = "<file_name>"  ! Default = Tao initialization file.
    plot_file          = "<file_name1> {<file_name2>} ..."
                                       ! Default = Tao initialization file.
    single_mode_file   = "<file_name>"  ! Default = Tao initialization 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* initialization file (as opposed to being present on the command line) is always relative to the directory that the *Tao* 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.

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.

Namelist	*File Name*	*Initialized here*	*Section*
lat_layout_drawing	plot_file*	Plotting	§10.10.8
floor_plan_drawing	plot_file*	Plotting	§10.10.8
tao_beam_init	beam_file	Particle beam	§10.5
tao_building_wall	building_wall_file	Plotting	§10.8
tao_design_lattice	"tao.init"	lattice files	§10.3
tao_d1_data	data_file	Data	§10.7
tao_d2_data	data_file	Data	§10.7
tao_params	"tao.init"	Global Variables	§10.4
tao_plot_page	plot_file*	Plotting	§10.10
tao_template_graph	plot_file	Plotting	§10.10
tao_template_plot	plot_file	Plotting	§10.10
tao_var	var_file	Variables	§10.6

*This namelist taken from the first file in the plot_file array.

Table 10.1: Table of `tao` Initialization Namelists.

10.3 Lattice Initialization

In the `tao_start` namelist, the `lattice_file` variable gives the name of the file that contains the `tao_design_lattice` namelist. This namelist defines where the lattice input files are. The variables that are set in the `tao_design_lattice` namelist are:

```
&tao_design_lattice
  n_universes        = <integer>      ! Number of universes. Default = 1.
  unique_name_suffix = "<string>"
  combine_consecutive_elements_of_like_name = <logical>
  common_lattice = <logical>                             ! Default = False
  design_lattice(i) = "<lattice_file>", {"<lattice2_file>"}
  design_lattice(i)%one_turn_map_calc = <logical>     ! Default = False
  design_lattice(i)%dynamic_aperture_calc = <logical> ! Default = False
/
```

`n_universes` is the number of universes to be created not counting the possible common universe created when using `CBL` analysis. The default is 1. `design_lattice(i)` gives the lattice file name for universe `i`. The syntax for `<lattice_file>` is:

```
  {<parser>::}<lattice_file>{#reverse}{@<use_line>}
```

Possible choices for the <parser> are:

```
  bmad      ! For a standard bmad lattice file. This is the default.
  xsif      ! For an xsif lattice file.
  digested  ! For a digested BMAD file.
```

If `#reverse` is present, tracking will be in the reverse direction from the end of the lattice to the beginning. The sign of the charge of the tracked particle will also be reversed. This is useful for simulating beams that go in the backward direction.

Example:

```
  &tao_design_lattice
    n_universe = 4
    design_lattice(1) = "this.lat#reverse"  ! Default: Bmad format lattice file.
    design_lattice(2) = "xsif::that.lat", "floor_coords.bmad"
                                            ! XSIF file. 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`. The `"xsif::"` prefix for `design_lattice(2)` indicates that the xsif parser is to be used. Alternatively, a `".xsif"` suffix signals that a file uses the `xsif` format. `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. This secondary lattice file must only have statements that are valid post lattice expansion. See the *Bmad* manual manual for a discussion of lattice expansion. A secondary lattice file must be in Bmad standard format. This can be especially useful if `lattice_file` is not a bmad file. For example, a `lattice2_file` can be used to set non-zero floor coordinates to an XSIF lattice file.

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(i)%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.` data type is populated. See Eq. 6.9 and Eq. 6.10. After startup, the map calculation can be toggled on/off by using the `set universe one_turn_map_calc` command (§11.26).

The `design_lattice(i)%dynamic_aperture` component sets whether the dynamic aperture calculation (§10.9) will be done. After startup, this calculation can be toggled on/off by using the `set universe dynamic_aperture_calc` command (§11.26).

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 (§4.1). 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
```

Using `##?"` as the suffix is convenient since it corresponds to the *Bmad* standard convention for distinguishing elements of the same name. See the *Bmad* manual for more details on this.

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.

The `common_lattice` switch can be used when there is a baseline lattice that is common to all universes. See §8.7 for more details.

10.4 Initializing Globals

Global variables are initialized in the `data_and_var_file` using a namelist block named `tao_params` The syntax of this block is:

```
  &tao_params
    global        = <tao_global_struct>     ! global parameters.
    bmad_com      = <bmad_com_struct>       ! Bmad global parameters.
    csr_param     = <csr_parameter_struct>  ! CSR global parameters.
    opti_de_param = <opti_de_param_struct>  ! de optimizer parameters.
  /
```

Example:

```
  &tao_params
    global%optimizer = "lm"  ! Set the default optimizer.
  /
```

The `tao_global_struct` structure contains *Tao* global parameters.

```
type tao_global_struct
  real(rp) y_axis_plot_dmin = 1e-4    ! Minimum y_max-y_min allowed for a graph.
  real(rp) lm_opt_deriv_reinit = -1   ! Derivative matrix cutoff. -1 => ignore this.
  real(rp) de_lm_step_ratio = 1       ! Step sizes between DE and LM optimizers.
  real(rp) de_var_to_population_factor = 5
  real(rp) svd_cutoff = 1e-5          ! SVD singular value cutoff limit.
  real(rp) random_sigma_cutoff = -1   ! Cut-off in sigmas.
  real(rp) merit_stop_value = -1      ! Value below which an optimizer will stop.
  integer n_opti_cycles = 20          ! number of optimization cycles
  integer n_opti_cycles = 1           ! number of optimization loops
  integer n_lat_layout_label_rows = 1 ! How many rows with a lat_layout
  integer phase_units = radians$      ! Phase units on output.
  integer bunch_to_plot = 1           ! Which bunch to plot
  integer random_seed = 0             ! use system clock by default
  character(16)  random_engine = "pseudo"         ! Random number engine to use
  character(16)  random_gauss_converter = "exact" ! Uniform to gauss conversion method
  character(16)  track_type = "single"            ! "single" or "beam"
  character(16)  prompt_string = "Tao"
  character(16)  optimizer     = "de"              ! optimizer to use.
  character(40)  print_command = "lpr"
  character(80)  var_out_file  = "var#.out"
  logical command_file_print_on = T     ! Toggle printing when using a command file.
  logical derivative_recalc = T         ! Recalc derivatives before each optimizer loop?
  logical derivative_uses_design = F    ! Derivative matrix uses the design lattice?
  logical disable_smooth_line_calc = F  ! Disable the plotting smooth line calc?
  logical draw_curve_off_scale_warn = T ! Display warning on graphs when any part of the
                                        !   curve is out-of-bounds
  logical init_opt_wrapper = T
  logical init_plot_needed = T          ! reinitialize plotting?
  logical label_lattice_elements = T    ! For lat_layout plots
  logical label_keys = T                ! For lat_layout plots
  logical lattice_calc_on = T           ! Master switch.
  logical opt_with_ref = F              ! use reference data in optimization?
  logical opt_with_base = F             ! use base data in optimization?
  logical optimizer_var_limit_warn = T  ! Warn when vars reach a limit when optimizing?
  logical plot_on = T                   ! Do plotting?
  logical rf_on = F                     ! RF cavities on?
  logical svd_retreat_on_merit_increase = T
  logical var_limits_on = T             ! Respect the variable limits?
  logical only_limit_opt_vars = F       ! Apply limits only if variable is used in optimization?
  logical single_step = F               ! Single step through a command file?
  logical optimizer_allow_user_abort = T ! See below.
end type
```

All global parameters can be changed from their initial value using the `set` command (§11.26).

`global%command_file_print_on`
: The switch controls whether printing is suppressed when a command file is called.

`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 §10.10.3.

`global%lattice_calc_on`
: `global%lattice_calc_on` controls whether lattice calculations are done. 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 (§11.26)), 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.

`global%merit_stop_value`
: The `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.

`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 (§8.5). 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.

`Random Number Generation`
: 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 (§10.5). For everything else, these parameters are accessed through the `tao_global_struct`.

`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_gauss_cutoff`.

`global%random_gauss_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 `0` (the default) causes the seed number to be picked based upon the system clock. Use the `show global` command to see what the seed number is.

`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.

`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.

`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 (§10.6). 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.

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_step    = 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.
  integer taylor_order = 3                ! 3rd order is default
  integer default_integ_order = 2         ! PTC integration order.
  integer ptc_max_fringe_order = 2        ! PTC max fringe order (2 => Quadrupole !).
                                          !   Must call set_ptc after changing.
  logical use_hard_edge_drifts = T        ! Insert drifts when tracking through cavity?
  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 space_charge_on = F             ! Space charge switch
  logical coherent_synch_rad_on = F       ! Longitudinal csr
  logical spin_tracking_on = T            ! Do particle spin tracking
  logical radiation_damping_on = F        ! Damping toggle.
  logical radiation_fluctuations_on = F   ! Fluctuations toggle.
  logical conserve_taylor_maps = T        ! Enable bookkeeper to set ele%map_with_offsets = F?
  logical absolute_time_tracking_default = F  ! Default for lat%absolute_time_tracking
  logical rf_auto_scale_phase_default = T     ! Default for lat%rf_auto_scale_phase
  logical rf_auto_scale_amp_default = T       ! Default for lat%rf_auto_scale_amp
  logical use_ptc_layout_default = F          ! Default for lat%use_ptc_layout
end type
```

See the *Bmad* manual for more details.

The `csr_parameter_struct` holds global variables for the coherent synchrotron radiation calculations.

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

```
    integer particle_bin_span = 2       ! Longitudinal particle length / dz_bin
    integer n_shield_images = 0         ! Chamber wall shielding. 0 = no shielding.
    integer ix1_ele_csr = -1            ! Start index for csr tracking
    integer ix2_ele_csr = -1            ! Stop index for csr tracking
    logical lcsr_component_on = T       ! Longitudinal csr component
    logical lsc_component_on = T        ! Longitudinal space charge component
    logical tsc_component_on = T        ! Transverse space charge component
    logical small_angle_approx = T      ! Use lcsr small angle approximation?
  end type
```

See the *Bmad* manual on the `csr_parameter_struct` for more details. In *Tao*, Besides setting the `csr_parameter_struct` components, the following must be done to enable CSR computations:

- The `global%track_type` (see above this section) must be set to `"beam"` and the appropriate beam initialization parameters (§10.5) must be set.
- The parameter `bmad_com%coherent_synch_radiation` (see above this section) must be set to `True`.
- In the *Bmad* lattice file, `csr_calc_on` must be set for the elements where CSR tracking is to be done (see the *Bmad* manual).

The `opti_de_param_struct` holds parameters that influence the behavior of the `de` optimizer (§8.5)

```
                         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 §8.1 for more details on `global%n_opti_cycles` and `global%n_opti_loops`.

10.5 Initializing Particle Beams

A particle beam is initialized in the `tao_beam_init` namelist block. The syntax is as follows:

```
  &tao_beam_init
    ix_universe                 = <integer>    ! Universe to apply to.
    beam0_file                  = <string>     ! File used in place of beam_init.
    beam_all_file               = <string>     ! File used in place of beam tracking.
    beam_saved_at               = "<ele_list>" ! Where to save the beam info.
    track_start                 = "<ele_name>" ! Element start tracking name or index.
    track_end                   = "<ele_name>" ! Element end tracking name or index.
    beam_init%distribution_type(3) = "<type>"  ! "ELLIPSE", "KV", "GRID", "" (default)
    beam_init%ellipse(3)%...    = ...          ! Parameters for an ellipse type distribution.
    beam_init%KV%...            = ...          ! Parameters for a KV distribution
    beam_init%grid(i)%...       = ...          ! 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_e             = <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_e_jitter      = <real>       ! bunch energy spread rms jitter (dE/E)
    beam_init%spin%polarization = <real>       ! spin polarization (1.0 = 100%)
    beam_init%spin%theta        = <real>       ! spin orientation  (polar coordinate)
    beam_init%spin%phi          = <real>       ! spin orientation  (polar coordinate)
    beam_init%init_spin         = <logical>    ! Initialize the spin (default: False)
    beam_init%preserve_dist     = <logical>    ! Use the same particle distribution.
    beam_init%random_engine     = "pseudo"     ! 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)?
  /
```

`ix_universe` refers to the universe index. See the *Bmad* documentation on the `beam_init_struct` for what the `beam_init` parameters refer to. The charge per particle is set to `bunch_charge/n_particle` and is used when calculating wakefield effects.

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.

The `beam0_file` component specifies a beam data 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 `-beam0` argument on the command line (§2.1). The file can either be in binary format (binary files can be created by the `write beam` command), or written in ASCII. The ASCII file format is:

```
  <ix_ele>          ! Lattice element index. This is ignored.
  <n_bunch>         ! Number of bunches.
  <n_particle>      ! Number of particles per bunch to use
  [bunch loop: ib = 1 to n_bunch]
    BEGIN_BUNCH     ! Marker to mark the beginning of a bunch specification block.
    <species_name>  ! Species of particle
    <bunch_charge>  ! Charge of bunch. 0 => Use <particle_charge>.
    <z_center>     ! z position at center of bunch.
    <t_center>     ! t position at center of bunch.
    [particle loop: Stop when END_BUNCH marker found]
      <x> <px> <y> <py> <z> <pz> <charge> <state> <spin_x> <spin_y> <spin_z>
    [end particle loop]
    END_BUNCH      ! Marker to mark the end of the bunch specification block
  [end bunch loop]
```

Example:

```
  0       ! ix_ele
  1       ! n_bunch
  25000   ! n_particle
  BEGIN_BUNCH
    POSITRON
    3.2E-9   ! bunch_charge
    0.0      ! z_center
    0.0      ! t_center
   -6.5E-3  9.6E-3 -1.9E-2  8.8E-3  2.2E-2 -2.4E-2  1.2E-13  1 1.0 0.0 0.0
    8.5E-3  5.5E-3  4.0E-2 -1.9E-2 -4.9E-3  2.1E-2  1.2E-13  1 1.0 0.0 0.0
    1.1E-2 -1.9E-2 -2.5E-2  1.0E-2 -1.8E-2 -7.1E-3  1.2E-13  1 1.0 0.0 0.0
   -3.4E-2 -2.7E-3 -4.1E-3  1.3E-2  1.3E-2  1.0E-2  1.2E-13  1 1.0 0.0 0.0
    6.8E-3 -4.5E-3  2.5E-3  1.4E-2 -2.3E-3  7.3E-2  1.2E-13  1 1.0 0.0 0.0
    1.2E-2 -9.8E-3  1.7E-3  6.4E-3 -9.8E-3 -7.2E-2  1.2E-13  1 1.0 0.0 0.0
    1.1E-2 -3.5E-4  1.2E-2  1.8E-2  5.4E-3  1.4E-2  1.2E-13  1 1.0 0.0 0.0
       ... etc. ...
  END_BUNCH
```

The first line of the file gives `ix_ele`, the index of the lattice element at which the distribution was created. This is ignored when the file is Read. The second line gives `<n_bunch>`, the number of bunches. The third line gives `n_particle` the number of particles in a bunch. After this, there are `<n_bunch>` blocks of data, one for each bunch. Each one of these blocks starts with a `BEGIN_BUNCH` line to mark the beginning of the block and ends with a `END_BUNCH` marker line. In between, the first four lines give the `species` name, `bunch_charge`, `z_center`, and `t_center` values. The `species` name may be one of:

```
  positron  ! default
  electron
  proton
  antiproton
  muon
  antimuon
  photon
```

The lines following the `t_center` line specify particle coordinates. One line for each particle. Only the first six number which are the phase space coordinates need to be specified for each particle. if `<particle_charge>` is not present, or is zero, it defaults to `bunch_charge/n_particle`. The `<state>` parameter indicates whether a particle is alive or dead. Values are

```
  1     ! Alive
  2-7   ! Dead
```

The particle spin is specified by x, y and z components.

The number rows specifying particle coordinates may be more then `<n_particle>`. In this case, particles will be discarded so that the the beam has `<n_particle>` particles. If `beam_init%n_particle`, if set in the *Tao* input file, this will override the setting of `<n_particle>` in the beam file.

Each particle has an associated `<particle_charge>`. If `<bunch_charge>` is set to a non-zero value, the charge of all the particles will be scaled by a factor to make the bunch charge equal to `<bunch_charge>`.

Additionally, if `beam_init%bunch_charge` is set in the *Tao* input file, this will override the setting of `bunch_charge` in the beam file.

When the particle coordinates are read in from the `beam0_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 (§11.26) can be used.

The `beam_all_file` component specifies a beam data file (which can be created with the `write beam` command) which contains the particle coordinates of the tracked beam at every element. This causes *Tao* to use the data from the file in lieu of actual tracking. This can be helpful when the time for *Tao* to track a bunch through the lattice becomes long. The file name can be overridden by using the `-beam_all` argument on the command line (§2.1). Note: *Tao* will set the variable `use_saved_beam_in_tracking` to `True` to prevent actual tracking. Note: A `beam_all_file` will supersede a `beam0_file`

When there is no `beam0_file` 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`.

`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` and `track_end` are not present, is to track through the entire root lattice branch. `track_start` and `track_end` is ignored for lattice branches other than the root branch (branch 0).

If spin tracking is desired then `beam_init%init_spin` must be set to true. If it is desired to use the exact same distribution of particles for each time the beam is tracked then set `beam_init%preserve_dist` to True. Otherwise, a new random distribution will be generated. The initialization routine does attempt to renormalize the beam to the specified parameters, nevertheless if tracking a small number of particles the distribution is subject to small random fluctuations unless `beam_init%prserve_dist` is True.

Tao re-tracks the beam through the lattice every time a lattice parameter is changed. For example, during optimizations or when the `set` command (§11.26) is used. For the re-tracking, the particle distribution at the beginning of the lattice is fixed. That is, the a new random distribution is *n*ot generated. To force a new distribution, use the `reinitialize beam` command (§11.23).

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"
  /
```

`beam_saved_at` is used to specify at what elements the beam distribution is to be saved at. The syntax used is element list format as explained in §4.1. The `BEGINNING` element (with index 0 in the lattice list) and the last element are automatically saved.

```
  &tao_beam_init
    beam_saved_at = "marker::m* *34w*" ! Save beam at all markers starting with "m"
                                       !   and all elements that have "34w" in their name.
  /
```

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` (§10.4). 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 §10.4.

10.6 Initializing Variables

`Variable`s are initialized using the `tao_var` namelist. The format for this is

```
&tao_var
  v1_var%name          = "<var_array_name>"  ! Variable array name.
  use_same_lat_eles_as = "<d1_name>"         ! Reuse a previous element list.
  search_for_lat_eles  = "<element_list>"    ! Find elements by name.
  default_universe     = "<integer>"         ! Universe variables belong in.
  default_attribute    = "<attribute_name>"  ! Attribute to control.
  default_weight       = <real>              ! Merit_function weight.
                                             ! default = 0.0
  default_step         = <real>              ! Small step value.
                                             ! default = 0.0
  default_merit_type   = "<merit_type>"      ! Sets how the merit is calculated.
                                             ! default = "limit"
  default_low_lim      = <real>              ! Lower variable value limit.
                                             ! default = -1e30
  default_high_lim     = <real>              ! Upper variable value limit.
                                             ! default =  1e30
  default_key_bound    = <logical>           ! Variables to be bound?
  default_key_delta    = <real>              ! Change when key is pressed.
  ix_min_var           = <integer>           ! Minimum array index.
  ix_max_var           = <integer>           ! Maximum array index.
  var(i)%ele_name      = "<ele_name>"        ! Element to be controlled.
  var(i)%attribute     = "<attrib_name>"     ! Attribute to be controlled.
  var(i)%universe      = "<uni_list>"        ! Universe containing variable to
                                             !    be controlled. "*" => All.
  var(i)%weight        = <real>              ! Merit function weight.
  var(i)%step          = <real>              ! Small step size.
  var(i)%low_lim       = <real>              ! Lower variable value limit
  var(i)%high_lim      = <real>              ! Upper variable value limit
  var(i)%merit_type    = "<merit_type_name>" ! Sets how the merit is calculated.
  var(i)%good_user     = <logical>           ! Good optimization variable?
  var(i)%key_bound     = <logical>           ! Variable bound to a key
  var(i)%key_delta     = <real>              ! Change when key is pressed.
/
```

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
  ix_min_var        = 0
  ix_max_var        = 99
  var(0:99)%ele_name  = "v00w", "v01w", "v02w", "    ", "v04w", ...
/
```

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(i)` array of variables has an index `i` that runs from `ix_min_var` to `ix_max_var`. A lattice element name `var(i)%ele_name` and the element's attribute to vary `var(i)%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(i)%attribute` is not specified in the namelist the `default_attribute` will be used.

`var(i)%key_bound` and `var(i)%key_delta` are used to bind variables to keys on the keyboard. 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(i)%step` establishes what a "small" variation of the variable is. This is used, for example, by some optimizers when varying variables. If `var%step(i)` is not given for a particular variable then the default `default_step` is used.

`var(i)%good_user` is a logical that the user can toggle when running *Tao* (§5). The initial default value of `%good_user` is True.

`var(i)%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(i)%universe` is not present, or is blank, the value of `default_universe` is used instead. If both `var(i)%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(i)%universe` may not be set in the namelist. The default if both `default_universe` and all `var(i)%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(i)%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(i)%low_lim` and `var(i)%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(i)%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 (§4.1). 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.

Note: `search_for_lat_eles` and `use_same_lat_eles_as` cannot be used together.

10.7 Initializing Data and Constraints

A set of data (§6) is initialized using a `tao_d2_data` namelist block and one or more `tao_d1_data` namelist blocks. 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"
```

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_data_source    = "<source>"           ! "lat" (dflt), "data", "var", or "beam".
    ix_min_data            = <integer>            ! Minimum array index.
    ix_max_data            = <integer>            ! Maximum array index.
    datum(j)%data_source     = "<source>"         ! "lat" (dflt), "data", "var", or "beam".
    datum(j)%data_type       = "<type_name>"      ! Eg: "orbit.x", etc.
    datum(j)%ele_name        = "<ele_name>"       ! Lattice element name.
    datum(j)%ele_start_name = "<ele_start_name>" ! Start element name.
    datum(j)%ele_ref_name    = "<ele_ref_name>"   ! Reference element names.
    datum(j)%merit_type      = "<merit_type>"     ! Sets how the merit is calculated.
    datum(j)%meas            = "<real>  "         ! Datum "measured" value
    datum(j)%weight          = "<weight>"         ! Merit function weight.
    datum(j)%good_user       = <logical>          ! Use for optimization and plotting?
    datum(j)%ix_bunch        = <integer>          ! Bunch index. Dflt: 0 = all bunches.
    datum(j)%eval_point      = "<where>"          ! "beginning", "center", or "end" (dflt).
    datum(j)%s_offset        = <real>             ! Default: 0.
  /
```

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", ...
  /
```

Alternatively, one can specify a datum 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) = "phase.b"  "Q09_1"  "B22"     "Q16_1"  "max"     30    0.1     T  ...
    datum( 3) = "floor.x"  ""        ""       "end"    "target"   3    0.01    T  ...
    datum( 4) = "floor.x"  "B1"      ""       "B2"     "target"   3    0.01    T  ...
    ... etc. ...
  /
```

When specifying data one line at a time, 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.

`ix_min_data` and `ix_max_data` give the bounds for the `datum(i)` structure array that is associated with the `d1_data` structure. `datum(:)%ele_name` gives the lattice element names associated with the data points.

`datum(i)%good_user` is a logical that the user can toggle when running *Tao* (§6.2). The initial default value of `%good_user` is True.

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 beta in the range between the end of element `B22` to the end of element `Q16_1`. 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>`.

The `datum(:)%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 §6.8).

If elements in the `data` array do not exist the corresponding `data%ele_name` should be left blank. 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"
  /
```

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 (§4.1). 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.

If `datum(j)%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 8.2. Custom data types not specified in this table must have a corresponding definition in `tao_hook_load_data_array.f90`. See Chapter 15 for details.

`datum(:)%weight` gives the weight coefficient for a datum in the merit function. If not present then the default weight of `default_weight` is used.

10.7.1 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
  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...
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...
```

10.8 Initializing a Building Wall

A two dimensional outline of the building containing the machine under simulation may be defined in *Tao*. This may be useful when drawing floor plans of the machine (§10.10.7) or to design a machine to

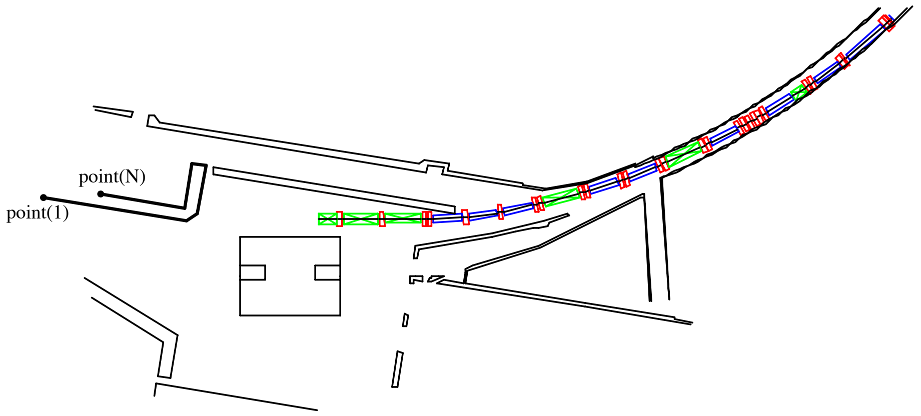


Figure 10.1: Floor plot showing the walls of the building (along with a section of a recirculation arc). Defining building walls can be useful for such things as floor plots and designing a machine to fit in an existing building.

fit within an existing building by optimizing (§8) the position of the machine to be within the building's walls.

The walls of a building are defined by a set of "sections" which are just curves that mark the wall boundaries. 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 (§10.2). 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
  {constraint = <type>}
  point(1) = <z1>, <x1>
  point(2) = <z2>, <x2>, {<r2>}
  point(3) = <z3>, <x3>, {<r3>}
  ... etc ...
  point(N) = <zN>, <xN>, {<rN>}
/
```

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 $+x$ side of the machine or the $-x$ 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 §6.8 for more details. Possible values for `constraint` are:

```
  "right_side"  ! Section is to be used with wall.x+ constraints
  "left_side"   ! Section is to be used with wall.x- constraints
  "none"        ! Default. Section is ignored in any constraint calculation.
```

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.

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.

10.9 Initializing Dynamic Aperture

For rings, the dynamic aperture can be calculated if `tao_dynamic_aperture` is defined:

```
  &tao_dynamic_aperture
   da_init(ix_uni)%pz = 0, 0.01, ...     ! List of particle energies to use
   da_init(ix_uni)%n_angle = 64          ! Number of angles in scan of each energy
   da_init(ix_uni)%min_angle = 0         ! Starting scan angle.
   da_init(ix_uni)%max_angle = 3.14159   ! Ending scan angle.
   da_init(ix_uni)%n_turn = 100          ! Number of turns a particle must survive
   da_init(ix_uni)%x_init = 1e-3_rp      ! initial estimate for horizontal aperture
   da_init(ix_uni)%y_init = 1e-3_rp      ! initial estimate for vertical aperture
   da_init(ix_uni)%accuracy = 1e-5_rp    ! resolution of bracketed aperture (meters)
  /
```

where `ix_uni` indicates the universe number. Here `pz` is a list of relative momenta to calculate the aperture for. If the RF is off, then a new closed orbit will be calculated for each of these momenta.

Optionally parameters `n_angle`, `min_angle`, and `max_angle` can be set to indicate the angle in the $x-y$ plane to scan about the closed orbit.

By default, the dynamic aperture calculation is off for all universes. To turn it on, use the `set` command (§11.26):

```
  set universe 1 dynamic_aperture_calc on
```

If Tao is compiled with the appropriate OpenMP flags, then this calculation will be done in parallel.

The results can be plotted. See §10.10.9.

10.10 Initializing Plotting

10.10.1 Plot Window

Plotting is defined by an initialization file whose name is defined by the `tao_start` namelist (§10.2). The first namelist block in the file has a block name of `tao_plot_page`. This block sets the size of the plot window (also called the plot page) and defines the "regions" where plots go. The syntax of this block is:

```
&tao_plot_page
  plot_page%plot_display_type        = <string>  ! Display type: 'X' or 'TK'
  plot_page%size                     = <x_size>, <y_size>         ! size in POINTS
  plot_page%border                   = <x1_b>, <x2_b>, <y1_b>, <y2_b>, "<units>"
  plot_page%text_height              = <real>   ! height in POINTS. Def = 12
  plot_page%main_title_text_scale    = <real>   ! Relative to text_height. Def = 1.3
  plot_page%graph_title_text_scale   = <real>   ! Relative to text_height. Def = 1.1
  plot_page%axis_number_text_scale   = <real>   ! Relative to text_height. Def = 0.9
  plot_page%axis_label_text_scale    = <real>   ! Relative to text_height. Def = 1.0
  plot_page%legend_text_scale        = <real>   ! Relative to text_height. Def = 0.8
  plot_page%key_table_text_scale     = <real>   ! Relative to text_height. Def = 0.9
  plot_page%floor_plan_shape_scale   = <real>   ! Floor_plan shape size scaling.
  plot_page%lat_layout_shape_scale   = <real>   ! Lat_layout shape size scaling.
  plot_page%title(i)                 = <string>, <x>, <y>, "<units>", "<justify>"
  plot_page%n_curve_pts              = <intger> ! Num points used to construct a
                                                !   smooth curve. Default = 401
   = <T/F>    ! Used with "show plot" command.
  plot_page%box_plots                = <T/F>    ! For debugging. Default = F.
  include_default_plots              = <T/F>    ! Include default templates? Def = T.
  region(i) = "<region_name>" <x1_r>, <x2_r>, <y1_r>, <y2_r>
  place(i)  = "<region_name>", "<template_name>"
  default_plot%...                            ! See below.
  default_graph%...                           ! See below.
/
```

For example:

```
&tao_plot_page
  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
  plot_page%title(1)    = "CESR Lattice", 0.5, 0.996, "%PAGE", "CC"
  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_plot%x%min = 100
  default_plot%x%max = 200
/
```

`plot_page%size` sets the horizontal and vertical size of the plot window in `points` units (72 points = 1 inch. Roughly 1 point = 1 pixel).

`plot_page%text_height` sets the overall height of the text that is drawn. Relative to this, various parameters can be used to scale individual types of text:

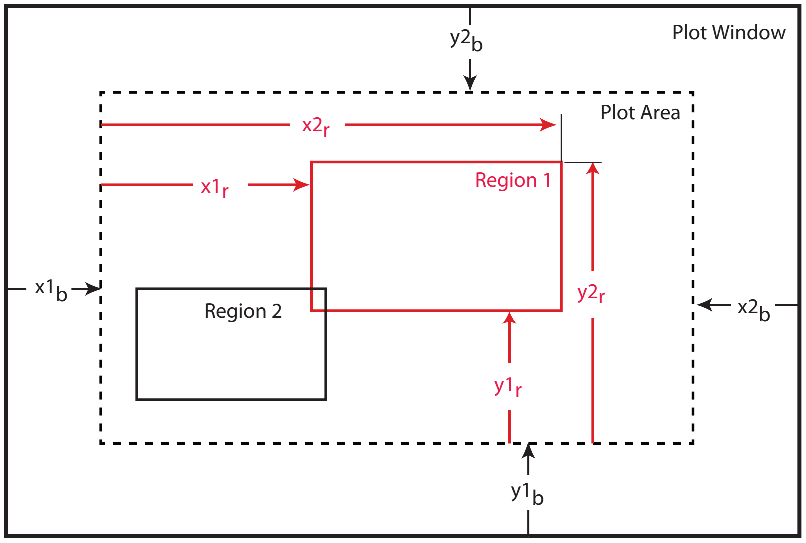


Figure 10.2: The plot window has a boarder whose position is determined by the `plot_page%border` parameter in the tao_plot_page namelist. Plots are placed in "`regions`" whose location is determined by the setting of the `region(i)` parameters in the same namelist. Regions may overlap.

```
  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 (§10.10.12).
  plot_page%legend_text_scale      = 0.9 ! Lat Layout or floor plan text.
```

The default values for these scales are given above.

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 the default PGPLOT)
```

Note: The environmental 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%border` sets a border around the edges of the window. As shown in Figure 10.2 $x1_b$, $x2_b$ are the right and left border widths and $y1_b$ and $y2_b$ are the bottom and top border widths respectively. The rectangle within this border is called the plot area.

`plot_page%title(i)` set the page title. There are two title areas (i = 1,2). If only the title string is given then the other variables are set to the defaults `x` = 0.5, `y` = 0.995, `justify` = "CC" and `units` = `"%PAGE"`. See the quickplot documentation for the `justify` variable syntax.

The plot area is divided up into rectangular regions where plots may be placed (what defines a plot is discussed below). `region(i)` in the `tao_plot_page` namelist is an array of five elements that defines the ith region. The first element of this array is the name of the region. This name may not contain a dot ".". The last four elements of the `retion(i)` array, $x1_r$, $x2_r$, $y1_r$ and $y2_r$ define the location of the region as illustrated in Figure 10.2. $x1_r$ and $x2_r$ are normalized to the width of the plot area and $y1_r$ and $y2_r$ are normalized to the height of the plot area. That is, these four number should be in the range $[0, 1]$. Regions may overlap any one can define as many regions as one likes.

Besides the regions that the user sets up in the `tao_plot_page` namelist, *Tao* defines a number of default regions whose names begin with the letter 'r'. Use the `show plot` command (§11.15) to view a list of these plots.

When `plot_page%delete_overlapping_plots` is True (the default), Placing a plot (using the `place` command §11.14) causes any existing plots that overlap the placed plot to become invisible.

The `plot_page%n_curve_pts` sets the default number of points to use for drawing "smooth" curves. Default is 401. This default may be overridden by setting the `plot%n_curve_pts` component of a plot (§10.10.2).

`place(i)` determines the initial placement of plots.

`default_plot` sets the defaults for any `plots` defined in the `tao_template_plot` namelists (§10.10.2). Similarly, `default_graph` sets defaults for the `graph` structure defined in the `tao_template_graph` namelist (§10.10.2). In the example above, the default x-axis min and max are set to 100 and 200 respectively.

If `include_default_plots` is set to `False`, the collection of default template plots (§10.10.2) that *Tao* uses by default are not used along with the template plots defined in the plotting file.

10.10.2 Plot Templates

As shown in Figure 7.1, a "plot" is made up of a collection of "graphs" and a graph consists of axes plus a set of "curves". In the `tao_plot.init` file there needs to be defined a set of "template plots". A template plot specifies the layout of a plot: How the graphs are placed within a plot, what curves are associated with what graphs, etc. When running *Tao*, the information in a template plot may then be transferred to a region using the `place` command and this will produce a visible plot.

Template plots are defined using namelists with a name of `tao_template_graph`. The general syntax is:

```
  &tao_template_plot
    plot%name        = "<plot_name>"
    plot%x           = <qp_axis_struct>
    plot%x_axis_type = "<x_axis_type>"   ! "index", "ele_index" "s", "lat", or "var".
                                         ! Default is "index".
    plot%n_graph     = <n_graphs>
    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.
    default_graph%...                    ! See below
  /
```

For example:

```
  &tao_template_plot
    plot%name                 = "orbit"
    plot%x%min                =   0
    plot%x%max                = 100
    plot%x%major_div_nominal = 10
    plot%x%label              = "Index"
    plot%n_graph              = 2
    default_graph%y%max       = 10
  /
```

`default_graph` sets defaults for the `graph` structure defined in the `tao_template_graph` namelist (§10.10.2). This overrides `default_graph` settings made in the `tao_template_plot` namelist (§10.10) but only for graphs associated with the `tao_template_plot` the `default_graph` is defined in.

`plot%x` sets the properties of the horizontal axis. For more information see the `Quick Plot` documentation on the `qp_axis_struct` in the Bmad manual. The major components are

```
  min         ! Left edge value.
  max         ! Right edge value.
  major_div   ! Number of major divisions.
              !  Number of major tick marks is one less.
  major_div_nominal ! Nominal number of major divisions
  minor_div   ! Number of minor divisions. 0 = auto choose.
  label       ! Axis label.
```

If `min` and `max` are absent, then *Tao* will autoscale the axis. If it is desired to have differing scales for different graphs, the `graph%x` component can be used (see below).

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` the number of major divisions is fixed at the set value and with `major_div_nominal` the number of major divisions can vary from the set value when *Tao* scales a graph. If `major_div_nominal` is set, this will override any setting of `major_div`. If neither `major_div` nor `major_div_nominal` is set, 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.

Plots with `plot%autoscale_x` and/or `plot%autoscale_y` logicals, set to true will automatically rescale after any calculation. The `plot%autoscale_gang_x` and `plot%autoscale_gang_y` components set how the `x_scale` (§11.37) and `scale` (§11.25) commands behave when autoscaling entire plots. See these individual commands for more details.

`plot%name` is the name that is used with *Tao* commands to identify the plot. 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` 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.
    "data"       ! From a data array
    "lat"        ! Lattice variable. See §10.10.5.
    "var"        ! Tao variable value. See §10.10.5.
```

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.

`n_graph` 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` namelists. The general format of the `tao_template_graph` namelist is:

```
  &tao_template_graph
    graph_index             = <integer>
    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%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%clip              = <logical>        ! Clip curves at boundary? Default = T
    graph%draw_axes         = <logical>        ! Default = T
    graph%draw_grid         = <logical>        ! Default = T
    graph%allow_wrap_around = <logical>        ! Wrap curves around lattice ends?
    graph%component         = "<string>"       ! Eg: "model - design"
    graph%symbol_size_scale      = <real>      ! Phase_space plots symbol scale factor
    graph%correct_xy_distortion  = <logical>   ! For Floor Plan plots: Default = F
    graph%ix_universe            = <integer>   ! Default = -1 => Use default universe
    graph%floor_plan_rotation    = <real>      ! Rotation of floor plan plot: 1.0 -> 360 deg.
    graph%floor_plan_view        = <string>    ! View plane for floor plan plot. default = 'zx'
    graph%floor_plan_orbit_scale = <real>      ! Scale for drawing orbits. Default: 0 -> Do not draw
    graph%floor_plan_orbit_color = <string>    ! Color of orbit. Default = 'RED'
    graph%floor_plan_size_is_absolute = <logical>     ! Shape sizes scaled to absolute dimensions?
    graph%floor_plan_draw_only_first_pass = <logical> ! Draw only first pass with multipass elements
    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>   ! number of curves
    curve(i)%name                = "<string>"  ! Default is "c<i>", <i> = curve num.
    curve(i)%data_source         = "<string>"  ! Source for the data curve points
    curve(i)%data_type_x         = "<string>"  ! Used with plot%x_axis_type = "data" or "var".
    curve(i)%data_type           = "<string>"  ! Default = plot%name.graph%name
    curve(i)%component           = "<string>"  ! Eg: "model - design". Overrides graph%component.
    curve(i)%data_index          = "<string>"  ! Index number for data points.
    curve(i)%legend_text         = "<string>"  ! Text for curve legend.
                                               !   Default is the data_type.
    curve(i)%y_axis_scale_factor = <factor>    ! Scale the y-axis by this.
    curve(i)%use_y2              = <logical>   ! Use left-axis scale?
    curve(i)%draw_line           = <logical>   ! Connect data with lines?
    curve(i)%draw_symbols        = <logical>   ! Draw data symbols?
    curve(i)%draw_symbol_index   = <logical>   ! Print index number next to the data symbol?
    curve(i)%ix_universe         = <integer>   ! Default = -1 => Use graph%ix_universe.
    curve(i)%ix_branch           = <integer>   ! Default = 0  => Use main lattice.
    curve(i)%ix_bunch            = <integer>   ! Bunch index. Default = 0 (all bunches).
    curve(i)%line        = <qp_line_struct>    ! Line spec (color, width, etc.)
    curve(i)%symbol      = <qp_symbol_struct>  ! Symbol spec (color size, etc.)
    curve(i)%symbol_every     = <integer>      ! Plot symbol every # datums
    curve(i)%ele_ref_name     = "<string>"     ! Name of reference element.
    curve(i)%ix_ele_ref       = <integer>      ! Index number of reference element.
    curve(i)%smooth_line_calc = <Logical>      ! Calc data between symbol points?
    curve(i)%units            = "<string>"     ! Data units
  /
```

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%max               =  4
    graph%y%min               = -4
    graph%y%major_div_nominal = 4
    graph%n_curve             = 1
    graph%component           = "model - design"
    curve(1)%data_source      = "data"
    curve(1)%data_type        = "orbit.x"
    curve(1)%units_factor     = 1000
    curve(1)%use_y2           = F
  /
```

`graph%title` is the string just above the graph. 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.

If there are multiple curves drawn with a graph then a curve legend showing what lines are associated with what data will be drawn. The default is to draw this 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`.

`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%box` sets the layout of the box which the `graph` is placed in. For a definition of what a box is see the Quick Plot documentation in the *Bmad* reference manual. In the above example the graph divides the region into two vertically stacked boxes and places itself into the bottom one.

`graph%allow_wrap_around` sets if, for a lattice with closed geometry, the curves contained in the graph are "wrapped" around the ends of the lattice. The default is `True`.

`graph%margin` sets the margin between the `graph` and the `box` it is drawn in.

`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%type` is the type of graph. *Tao* knows about the following types:

```
  "data"               ! Data and/or variable plots (default) (§10.10.3).
  "floor_plan"         ! A 2-dimensional birds-eye view of the machine (§10.10.7).
  "histogram"          ! Histogram of plot (§10.10.10).
  "key_table"          ! Key binding table for single mode (§10.10.12).
  "lat_layout"         ! Schematic showing placement of the lattice elements (§10.10.6).
  "phase_space"        ! Phase space plots (§10.10.13).
```

With `graph%type` set to `"beam_chamber_wall"` (§10.10.11), the beam chamber wall is drawn if it has been defined in the *Bmad* lattice file.

With `graph%type` set to `"data"` (§10.10.3), 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` (§10.10.7), the two dimensional layout of the machine is drawn.

With `graph%type` set to `histogram` (§10.10.10), such things such as beam densities can be histogrammed.

With `graph%type` set to `"key_table"` (§10.10.12), 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` (§10.10.6), 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` (§10.10.13), phase space plots are produced.

10.10.3 Data and Variable plotting

A `graph` (§10.10.2), with `graph%type` equal to `"data"`, is used to draw "data" such as orbits and/or variable values such as quadrupole strengths. A data `graph` will have a number of associated `curves` with each curve defining a particular data type to plot.

The data values will depend upon where the data comes from. This is determined, in part, by the setting of `graph%component` and `curve%component`. `graph%component` and `curve%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, `graph%component` may be set to plot a linear combination of the above. For example:

```
  graph%component = "model - design"
```

This will plot the difference between the `model` and `design` values.

If `curve%component` is set, it will override `graph%component`. If `graph%component` is not set in the initialization file, and if there are curves of the graph that have not been set, `graph%component` will be given a default setting of `model`.

The `curve` structure is used to define the data that is plotted in each graph. `curve%data_source` is the type of information for the source of the data points. `curve%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 tracking a beam of particles.
  "multi_turn_orbit"   ! Computation is from multi-turn tracking.
```

The default for `curve%data_source` is `"lat"`. With `curve%data_source` set to `data`, the values of the curve points come from the `d1_data` array structure named by `curve%data_type`. Thus in the above example the curve point values are obtained from `orbit.x` data. To be valid the data structure named by `curve%data_type` must be set up in an initialization file. If not given, the default `curve%data_type` is

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

If `curve%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. `curve%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 `curve%ix_bunch`. The default is `0` which combines all the bunches of the beam for the calculation.

Example: With `curve%data_type` set to `beta.x`, the setting of `curve%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.

`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 (§11.27). `curve%draw_symbol_index` determines whether the index number is printed besides the symbol. Use the `set curve` command (§11.26) to toggle the drawing of symbols. The default value for `curve%draw_symbol` is False if `plot%x_axis_type` is `"s"` and True otherwise. The default`curve%draw_symbol_index` is always False.

`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 `graph%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 symbols. The default of `curve%smooth_line_calc` is True. Use the `set curve` command (§11.26) to toggle the drawing of lines. Alternatively, the `-disable_smooth_line_calc` switch can be used on the command line (§2.1) or the global variable `global%disable_smooth_line_calc` can be set in the *Tao* initialization file (§10.4).

The `graph%draw_only_good_user_data_or_vars` switch determines whether datums (§10.7) or variables (§10.6) 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.

A graph has two vertical axes. The one on the left is called `"y"` and the one on the right is called `"y2"`. For example, `graph%y%label` sets the axis label for the `y` axis and `graph%y2%label` sets the axis label for the `y2` 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.

Typically, a graph's horizontal scale is set by the `plot%x` component. If it is desired to have differing scales for different graphs, the `graph%x` component can be used.

10.10.4 Graphing a Data Slice

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 plotbe 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 `graph%component` Example:

```
&tao_template_plot
  plot%name = "at_bpm"
  plot%x%label = "x"
  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%component = "meas - ref"
  graph%type = "data"
  graph%n_curve = 1
  graph%x_axis_scale_factor = 1000
  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 (§11.26) can be used to change `curve%ele_ref_name` and `graph%component` strings.

`curve%data_index` sets the index number for the symbol points (§10.10.2). 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` (§11.27) will print the symbol index number along with the (x, y) coordinates of the symbols.

Arithmetic expressions (§4.2) 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 `graph%component` settings are ignored for that expression.

10.10.5 Plotting With a Variable Parameter on the X-Axis

Data can be plotted as a function of a lattice parameter by setting `plot%x_axis_type` to `"lat"` (for lattice variables) 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.

Example:

```
&tao_template_plot
  plot%x_axis_type = "lat"
  plot%n_curve_pts = 50
  ...
/

&tao_template_graph
  ...
  curve(1)%data_type_x = "beam_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 `graph%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]"
```

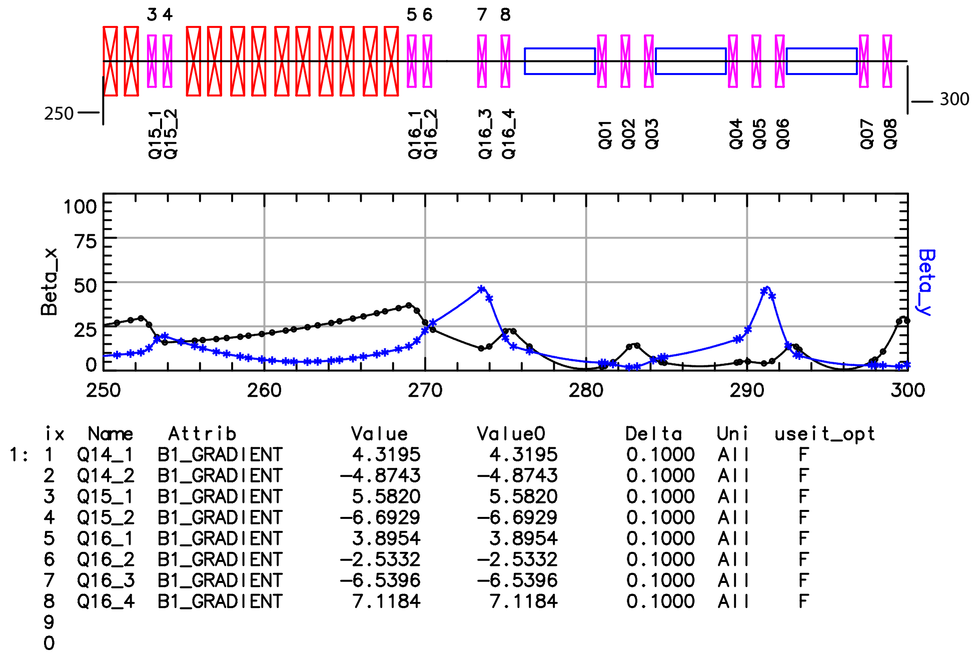
where the universe #2 `model` lattice would be setup to be equal to the universe #1 `design` lattice.

10.10.6 Drawing a Lattice Layout

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%x%min       = <real>
  plot%x%max       = <real>
  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%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%x%min       =   0
  plot%x%max       = 100
  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
  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 §10.10.8 for more details.

The longitudinal distance markers at either end of the lattice layout can be suppressed by setting

```
graph%x%draw_numbers = F
```

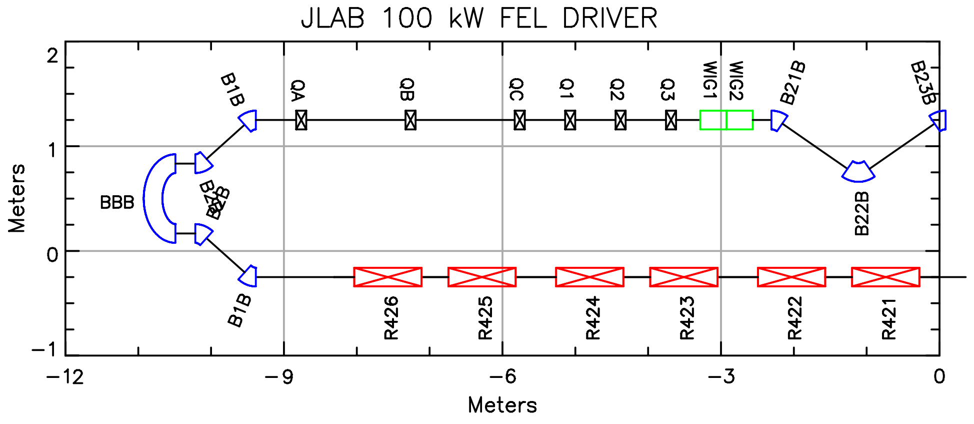


Figure 10.4: Example Floor Plan drawing.

10.10.7 Drawing a Floor Plan

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` (§10.10.6), Elements are represented by colored rectangles and which elements are drawn is determined by an `floor_plan_drawing` namelist (see §10.10.8).

The placement of an element in the drawing is determined by the element's coordinates in `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.

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 are 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. Etc.

An overall rotation of the floor plan can be controlled by setting `graph%floor_plan_rotation`. A setting of 1.0 corresponds to 360°. Positive values correspond to counter-clockwise rotations. Example: (§10.10)

```
&tao_plot_page
  graph%floor_plan_rotation = 0.5   ! Rotate 180 degrees
/
```

The beam orbit can be drawn upon the floor plan. This is done by setting `graph%floor_plan_orbit_scale` to something nonzero. A value of zero suppresses the drawing of the orbit. This scale is used to scale the distance between the centerline of the elements and the orbit. Note: If `graph%floor_plan_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.

The color of a drawn beam orbit is controlled by `graph%floor_plan_orbit_color`. The default color is 'RED'.

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.

Example Floor Plan template:

```
&tao_template_plot
  plot%name = "floor"
  plot%x%min = -12
  plot%x%max = 0
  plot%x%major_div_nominal = 4
  plot%x%minor_div = 3
  plot%x%label = "Meters"
  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 = 1
  graph%y%label = "Meters"
  graph%y%max = 2
  graph%y%min = -1
  graph%correct_xy_distortion = T
  graph%floor_plan_view = 'xz'  ! Looking from beneath
/
```

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. If `graph%correct_xy_distortion` is set to `False`, this scaling will not be done.

Note: The `show ele -floor` command (§11.27) can be used to view an element's global coordinates.

10.10.8 Defining Shapes for Lat_layout and Floor_plan Drawings

`Floor plan` (§10.10.7) 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 namelist syntax is the same for both:

```
&lat_layout_drawing
  ele_shape(i) = "<name>" "<shape>" "<color>" "<size>" "<label>" <draw> <multi>
/

&floor_plan_drawing
  ... same as lat_layout_drawing ...
/
```

For Example:

```
&floor_plan_drawing
  !               ele_id                     Shape      Color     Size      Label Draw? Multi?
  ele_shape(1) = "quadrupole::q*"           "box"      "red"     0.75      "name"  T
  ele_shape(2) = "quadrupole::*"            "xbox"     "red"     0.75      "none"
  ele_shape(3) = "sbend::sb*"               "box"      "blue"    0.37      "none"  T    T
  ele_shape(3) = "sbend::*"                 "box"      "blue"    0.37      "none"
  ele_shape(4) = "wiggler::*"               "xbox"     "green"   0.50      "name"  F
  ele_shape(5) = "var::quad_k1"             "circle"   "purple"  0.25      "name"
  ele_shape(6) = "data::orbit.x|design"     "vvar_box" "orange"  0.25      "name"
  ele_shape(7) = "wall::building"           "-"        "black"    0        "-"
/
```

A figure is drawn for each lattice element in the lattice that matches the `<ele_id>` specification (§4.1) of any `ele_shape(:)`. 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. 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 for the first shape matched, *Tao* will look for additional matches. Each time a match is found, the setting of `<multi>` for that shape will be used to determine whether additional shapes are searched for.

For a floor plan, for `wigglers` or `undulators` that have an `x_ray_line_len` attribute (see the *Bmad* manual), The X-ray line will be drawn if an `ele_shape` for a `photon_branch` is present.

Use the `show plot -shape` command to see the defined shapes. use the `set shape` command (§11.26)) to set shape parameters on the command line.

Data and variables can also be specified to be drawn by using a `<ele_id>` beginning with `data::` for drawing data and `var::` for drawing variable locations. In the above example, it is assumed that a `quad_k1` variable array and a `orbit.x` data array have been setup. A circle will be drawn at each element under control of a `quad_k1` variable. For the `orbit.x` data, an "x" will de drawn where the data is being evaluated but only for datums whose `useit_opt` parameter is True.

For `floor_plan` drawings, the building wall (§10.8) can be drawn by specifying an `ele_shape` whose name is `"wall::building"`. For the building wall, the only `ele_shape` attribute that is relevant is the `color`.

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 For floor_plan drawings, to determine the size of a shape, `<size>` is combined with the `graph` parameter

```
  floor_plan_size_is_absolute  ! Default: False.
```

If `floor_plan_size_is_absolute` is False (the default), `<size>` is taken to be the size of the shape in points (1 point is approximately 1 pixel). If `floor_plan_size_is_absolute` is True, `<size>` is taken to be the size in meters. That is, if `floor_plan_size_is_absolute` is False, zooming in or out will not affect the size of an element shape while if `floor_plan_size_is_absolute` is True then the size of an element will scale with the zoom.

The `graph%floor_plan_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.

The overall size of all the shapes can be scaled using the `plot_page` (§10.10) 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 `<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 (§11.26).

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. Valid Shapes are:

```
  "asym_var_box"   -- Like var_box but is not symmetric about the center line.
  "asym_vvar_box"  -- Like asym_var_box except scaled to variable strength.
  "box"            -- Rectangular box
  "var_box"        -- Rectangular box with variable height.
                       The box is symmetric about the center line.
  "vvar_box"       -- Like var_box except scaled to variable strength.
  "bow_tie"        -- Bow-tie shape.
  "circle"         -- Circle centered at center of element.
  "diamond"        -- Diamond shape.
  "pattern:<pattern_name>"
                   -- Custom curve specified by <name>.
  "x"              -- "X" centered at center of element
  "xbox"           -- Rectangular box with an x through it.
```

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. 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 `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>"
    curve(1)%pt(1) = <s>, <x>, <radius>
    curve(1)%pt(2) = <s>, <x>, <radius>
    curve(1)%line%color = <color_name>
    curve(1)%line%width = <line_width>
    curve(1)%scale = "none"
    curve(2)%...
  /
```

Example:

```
  &floor_plan_drawing
    ...
    ele_shape(2) = "quadrupole::*"     "pattern:q_pat"     "red"     0.75     "none"
    ...
  /

  &shape_pattern
    name = "q_pat"
    curve(1)%pt(1) = 0, -1
    curve(1)%pt(2) = 1, -1
    curve(1)%pt(3) = 0.9, 1
    curve(1)%pt(4) = 0.1, 1
    curve(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 a series of curves. Each curve is specified by a number of points. Between the points, an arc is drawn with the given `radius` or a line segment if the `radius` is zero (which is the default). If there is only one point specified with a non-zero `radius`, the `s, x` value is taken to be the center of a circle with the given radius. In the above example, there is one curve with five points which represents 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 `x` coordinate is scaled by the `size` attribute of the `ele_shape`. The default color for a given curve is taken to be the `color` attribute of the `ele_shape` ("red" in the example).

10.10.9 Drawing a Dynamic Aperture

A `dynamic_aperture` drawing displays the results of the dynamic aperture calculation. For example, the template

```
&tao_template_plot
```

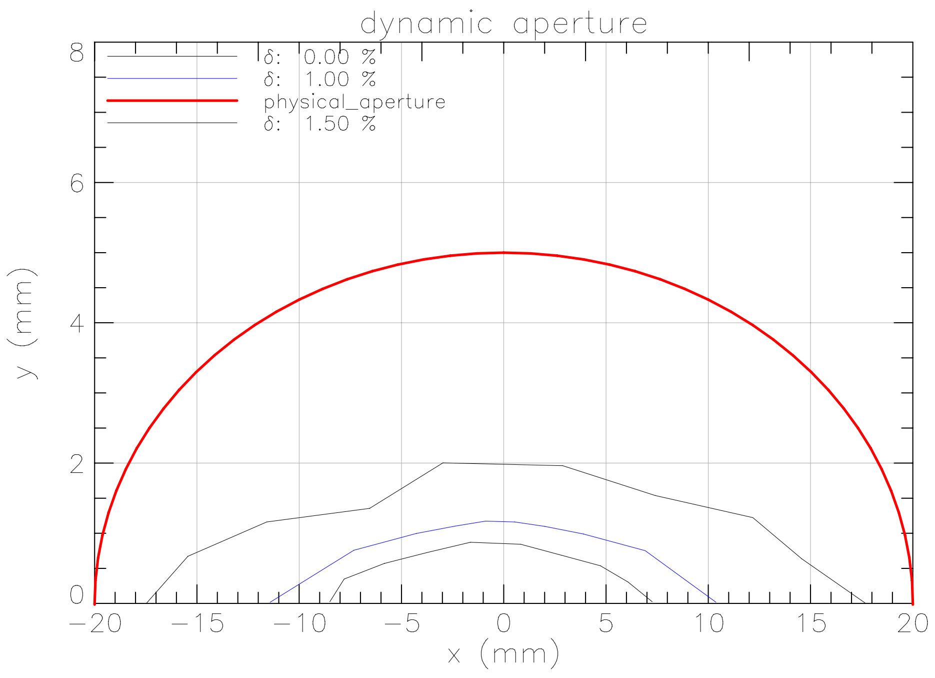


Figure 10.5: Example dynamic aperture plot.

```
  plot%name = 'da'
  plot%x%min = -20
  plot%x%max =  20
  plot%x%major_div_nominal = 10
  plot%x%label = 'x (mm)'
  plot%x_axis_type = 'phase_space'
  plot%n_graph = 1
/

&tao_template_graph
  graph%name = 'g1'
  graph%type = 'dynamic_aperture'
  graph_index = 1
  graph%title = 'dynamic aperture'
  graph%margin =  0.15, 0.06, 0.12, 0.12, '%BOX'
  graph%x_axis_scale_factor = 1000
  graph%y%label = 'y (mm)'
  graph%y%label_offset = .2
  graph%y%max = 0
  graph%y%min = 0
  graph%y%major_div = 4
  graph%n_curve = 3
  curve(1)%y_axis_scale_factor = 1000
  curve(2)%y_axis_scale_factor = 1000
  curve(3)%y_axis_scale_factor = 1000
  curve(1)%draw_symbols = F
  curve(2)%draw_symbols = F
  curve(3)%draw_symbols = F
  curve(3)%data_type = 'physical_aperture'
  curve(3)%line%color = 2
  curve(3)%line%width = 5
/
```

produces the plot on Fig. 10.5. Each curve represents a single momentum calculation according to §10.9. If there are more momenta than curves (as in this case), additional curves will automatically be created using the styles of the previous curves. Note that apertures are calculated at element 0. If `curve(i)%ix_ele_ref` is nonzero, then the aperture will be propagated to this element.

Dynamic aperture curves can have the following `%data_type`:

```
  'dynamic_aperture' or ''     ! (default) points include the reference orbit
  'dynamic_aperture_centered' ! points are centered (relative to) the reference orbit
  'physical_aperture'         ! draws the physical aperture based on x1_limit, etc.
```

10.10.10 Drawing a Histogram

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 (§10.10.13). An example is shown in Fig. 10.6, using the example histogram template:

```
&tao_template_plot
  plot%name = 'zhist'
  plot%x%min = -6
  plot%x%max =  6
  plot%x%label = 'z (mm)'
  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'
  graph%y%label = 'Current (A)'
  graph%n_curve = 1
  graph%y%label_offset = .1
  graph%x_axis_scale_factor = 1000.00 !m->mm

  curve1%hist%density_normalized = T
  curve1%hist%weight_by_charge = T
  curve1%hist%number = 100
  curve1%line%color = 4
  curve1%line%pattern = 2
  curve1%y_axis_scale_factor = 299792458  !Q/m * c_light
  curve1%data_type = 'z'
  curve1%data_source = 'beam_tracking'
  curve1%ele_ref_name = "BEGINNING"
```

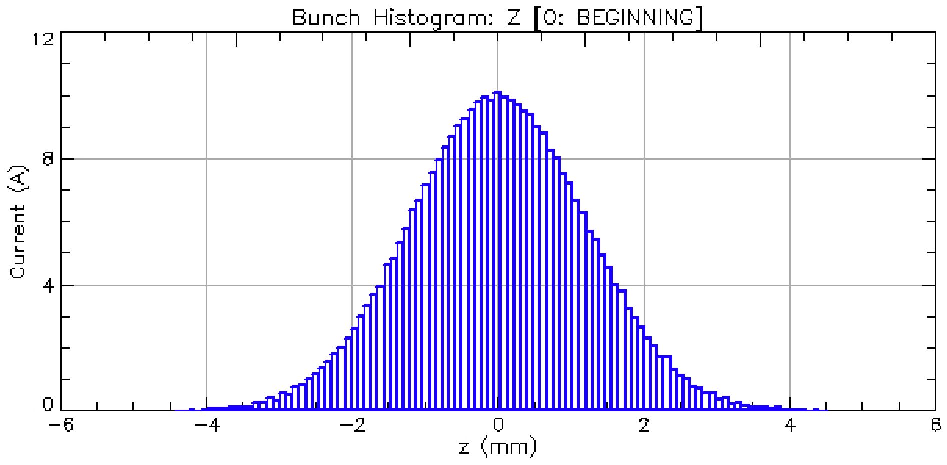


Figure 10.6: Example histogram plot.

```
  curve1%symbol%type = 1
/
```

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"
  "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` or `set curve ix_ele_ref` commands.

If `graph%type` is "histogram" then `curve%data_source` must be either:

```
  "beam"
  "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. `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 histogram planes to be plotted. For example, if the histogram 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 (§10.7).

10.10.11 Drawing the Beam Chamber Wall

If a beam chamber wall has been defined in the lattice file, This wall can be drawn in a `curve` by setting `curve%type` to `"beam_chamber_wall"`.

Beam chamber walls are drawn, like a `lat_layout`, on a one dimensional line as a function of longitudinal position along the machine centerline.

Note: Use the command `show ele -wall` to print information about the beam chamber wall for a particular element.

10.10.12 Drawing a Key Table

The `key table` is explained more fully in Section §12.1. 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
    graph%n_curve = 0
  /
```

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:

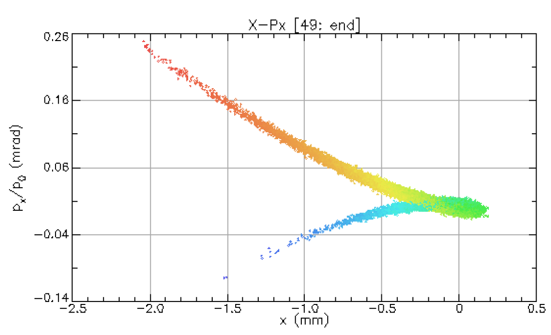
```
  Ix         ! Key index.
  Name       ! Element name whose attribute is bound.
  Attrib     ! Name of the element attribute that is bound.
  Value      ! Current value of bound attribute.
  Value0     ! Initial value of bound attribute.
  Delta      ! Change in value when the appropriate key is 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.

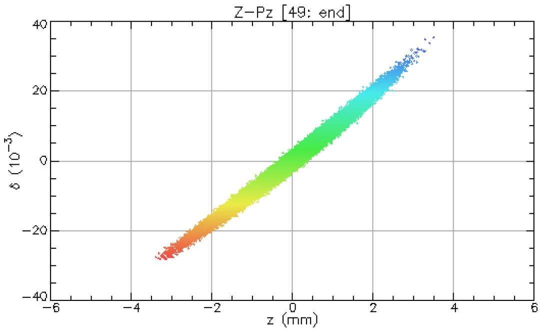
The `key_table` is drawn with respect to the upper left hand corner of the region in which it is placed.

10.10.13 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



(a) Horizontal phase space



(b) Longitudinal phase space

Figure 10.7: Example Phase Space plot, with points colored by the `pz` coordinate.

here are similar to data plotting (§10.10.3). An example is show in Figure 10.7. Example Phase Space template:

```
&tao_template_plot
  plot%name = "xphase"
  plot%x%min =   -2.5
  plot%x%max = 0.5
  plot%x%label = "x (mm)"
  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_axis_scale_factor = 1000.00 !m->mm
  graph%y%label =  "p\dx\u/p\d0\u (mrad)"
  graph%y%major_div = 4
  graph%n_curve = 1
  graph%y%label_offset=.4
  curve(1)%data_type = "x-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%type = 1
  curve(1)%data_type_z = "pz"
  curve(1)%use_z_color = T
  /
```

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. The phase space coordinates are:

```
  "x"
  "px"
  "y"
  "py"
  "z"
  "pz"
  "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 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` or `set curve ix_ele_ref` commands.

Points can be colored by another phase space coordinate by activating `use_z_color = T`. The available curve options and defaults are:

```
  use_z_color = F
  data_type_z = ""
  z_color0 = 0
  z_color1 = 0
  autoscale_z_color = T
```

These can be the init file, or in Tao using the `set curve` command. The `data_type_z` can be set to any of the available phase space coordinates. `z_color0` and `z_color1` 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_z_color=T`, then these will be set automatically based on the limits of the `data_type_z` 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 (§10.7). 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 1e-6 m-rad is used.

Chapter 11

Tao Line Mode 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 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 (§11.28).

Commands are case sensitive. The list of commands is shown in Table 11.1. 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
```

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
```

Command	*Section*	*Command*	*Section*
`alias`	§11.1	`re_execute`	§11.20
`call`	§11.2	`read`	§11.21
`change`	§11.3	`restore`	§11.22
`clip`	§11.4	`reinitialize`	§11.23
`continue`	§11.5	`run_optimizer`	§11.24
`derivative`	§11.9	`scale`	§11.25
`do, enddo`	§11.6	`set`	§11.26
`end_file`	§11.7	`show`	§11.27
`exit`	§11.8	`single_mode`	§11.28
`flatten`	§11.10	`spawn`	§11.29
`help`	§11.11	`timer`	§11.30
`misalign`	§11.12	`use`	§11.31
`pause`	§11.13	`veto`	§11.33
`place`	§11.14	`value`	§11.32
`plot`	§11.15	`wave`	§11.34
`ptc`	§11.16	`write`	§11.35
`python`	§11.17	`x_axis`	§11.36
`quiet`	§11.18	`x_scale`	§11.37
`quit`	§11.19	`xy_scale`	§11.38

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]]". arguments that do not have a corresponding "[[i]]" are placed at the end of `<string>`.

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 (§2.3) and executes the commands in it. Format:

```
call <filename> {<arg_list>}
call -ptc <filename>
```

Tao first looks in the current directory for the command file.

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 Section §2.3 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.

If the command file has the `quiet` command in it, output to the terminal is surpresed (but only for the duration of the execution of the file).

Command loops can be implemented in a command file. See Section §11.6 for more details.

Other useful commands to put in a command file are to speed up execution are:

```
set global lattice_calc_on = F   ! Stop lattic calculations
set global plot_on = F           ! Halt replotting
```

If set, at the end of the command file these logicals should be toggled back to True.

Examples:

```
  call 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 element <element_list> <attribute> {prefix>} <number>
change {-silent} variable <name>[<locations>] {<prefix>} <number>
change  {n@}beam_start <coordinate> {prefix>} <number>
```

The `change` is used for changing real (as opposed to integer or logical) parameters. Also see the `set` command (§11.26) 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 (§4.1), without any embedded blanks, is used for the `<element_list>` argument.

For `change beam_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 beam_start <coordinate> <number>` can be used to vary the starting coordinates for single particle tracking or the centroid coordinates for beam tracking. Here `<coordinate>` is one of:

```
  x, px, y, py, z, pz, t
```

For photons, `<coordinate>` may also be:

```
  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`.

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 `-silent` switch, if present, suppresses the printing of what variables are changed.

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@beam_start x @0.001         ! set beginning x position in universe 2 to 1 mm.
```

11.4 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 top region
  clip bottom       ! Clip the graphs in the bottom region
  clip -g orbit.x   ! Clip the
```

11.5 continue

The `continue` command is used to continue reading of a suspended command file (§2.3) after a `pause` command (`s:pause`). Format:

```
  continue
```

11.6 do, enddo

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 increamented 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 beam_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>]]".

11.7 end_file

The `end_file` command is used in command files (§2.3) 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
```

11.8 exit

The `exit` command exits the program. Same as `Quit`. Format:

```
  exit
```

11.9 derivative

The `derivative` command calculates the `dModel_Data/dVar` derivative matrix needed for the `lm` optimizer. Format:

```
  derivative
```

11.10 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>}
```

11.11 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.
```

Note: The `help` command works by parsing the file `$TAO_DIR/doc/command-list.tex` which is the LaTeX file for the Line Mode Commands chapter of the *Tao* manual. Thus, for the `help` command to work properly, the environmental variable `TAO_DIR` must be appropriately defined. Generally, `TAO_DIR` will be defined if the appropriate 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.

11.12 misalign

The `misalign` command misaligns a set of lattice elements. Format:

```
   misalign <wrt> <ele_type> <range> <ele_attrib> <misalign_value>
```

`<ele_type>` is the type of element to misalign. Only elements of type `<ele_type>` will be misaligned within the range. If `<ele_type>` begins with "*@" then choose all universes. If `<ele_type>` begin with "n@" then choose universe n. Otherwise the default universe (§3.3)) is used.

A lattice element will only be misaligned if its lattice index falls within a range given by `<range>`. `<range>` is of the form `nnn:mmm` or the word `ALL`.

The element attribute `<ele_attrib>` is "misaligned" by the rms value `<misalign_value>` with respect to the setting of `<wrt>`. Any element attribute can be misaligned provided the attribute is free to vary.

If `<misalign_value>` is prepended by 'x' then the misalignment value will be a relative misalignment with respect to the `<wrt>` value. Otherwise, it's an absolute rms value about the `<wrt>` value.

In the special case where sbend strengths are misaligned then use `<ele_attrib> = g_err`. However, if a relative error is specified it will be relative to 'g'.

The possible values of `<wrt>` are:

```
  wrt_model          ! Misalign about the current model value
  wrt_design         ! Misalign about the design value
  wrt_survey         ! Misalign about the zero value
```

Examples
```
  ! The following will misalign all quadrupole vertical positions in the default
  ! universe within the lattice element range 100:250 with respect to the zero
  ! value by 300 microns
misalign wrt_survey quadrupole 100:250 y_offset 300e-6
  ! The following will misalign all quadrupole strengths in all universes for
  ! the entire lattice with respect to the design value by 1%.
misalign wrt_design *@quadrupole ALL k1 x0.01
```

11.13 pause

The `pause` command is used to pause *Tao* when executing a command file (§2.3). 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 (§11.5) 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.
```

11.14 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 <region> <template>
place <region> none
place * none
```

To erase a plot from a region use `none` in place of a template name. Notice that by using multiple `place` commands a `template` can be associated with more than one region. `place * none` will erase all plots.

Examples:
```
place top orbit   ! place the orbit template in the top region
place top none    ! erase any plots in the top region
```

11.15 plot

The `plot` command is used to determine what "components" (§10.10.3) are plotted in the specified graphs or plots. Format:

```
  plot <plot_or_graph> <component>
```

components are a property of a graph (or curve) so when <plot_or_graph> specifies a plot, all the graphs associated with the plot are assigned the `<component>`.

Note: The plot command is a shortcut for the commands:

```
  set plot <plot_name> component = <component>     ! and
  set graph <graph_name> component = <component>
```

Also see the `set curve` command.

Use a "-" for baselines.

Examples:

```
  plot bottom.g1 model - design        ! Plot model - design in the g1 graph of the bottom region
  plot top meas - model + design - ref ! Set the components for the graphs in the top region.
```

11.16 ptc

The `ptc` command is used manipulating PTC layouts associated with Bmad lattices. Format:

```
  ptc init             ! Init associated PTC layout.
```

The `ptc init` command must be run before running any other `ptc` command is used.

Also see:

```
  call -ptc <file>         ! Run a PTC script
  read ptc                 ! Read a PTC lattice
  write ptc                ! Write a PTC lattice
```

Examples:

```
  ptc init
```

11.17 python

The `python` command is like the `show` command in that the `python` 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 `python` command is meant for easy parsing. Format:

```
  python {-append <file_name>} {-noprint} <what_to_print>
  python {-write <file_name>} {-noprint} <what_to_print>
```

The `python` command has `-append` and `-write` optional arguments which can be used to write the results to a file. The `python -append` command will appended to the output file. The `python -write` command will first erase the contents of the output file. Example:

```
  python -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 `python` 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 Python, Perl, Tcl, etc. In particular, see §13 for details on how to run *Tao* as a Python subprocess.

For long term maintainability of python scripts, the advantage of using the `python` command in the scripts over the `show` command comes from the fact that the output syntax of the `show` command can (and does) change.

For further documentation on the python command, please look at the file tao/code/tao_python.f90.

Note: At this point in time, the `python` command is still in development. Please contact David Sagan if needed.

11.18 quiet

Format:

```
quiet
```

The `quiet` command can only be used in command files (§11.2). When placed in a command file, output to the terminal is surpresed (but only from the `quiet` command for the duration of the execution of the file).

Other useful commands to put in a command file are to speed up execution are:

```
set global lattice_calc_on = F   ! Stop lattic calculations
set global plot_on = F           ! Halt replotting
```

If set, at the end of the command file these logicals should be toggled back to True.

11.19 quit

`Quit` exits the program. Same as `exit`. Format:

```
quit
```

11.20 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.

Examples

```
re_exe 34   ! Re-execute command number 34.
re_exe set  ! Re-execute last ``set'' command.
```

11.21 read

The `read` command is used to modify the (*Bmad*) `model` lattice or the associated `PTC` lattice. Format:

```
read lattice <file_name>
read ptc {-old} <file_name>
```

With the `read lattice` command, the `model` lattice contained in the default universe (§3.3) is modified using a "secondary" lattice file. [See the *Bmad* manual for the definition of secondary.]

For example, with the appropriate file, the `read` command can be used to misalign the lattice elements. The input file must be in Bmad standard lattice format.

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 §10.3.

The `read ptc` command reads in a PTC lattice. WARNING: This command is untested. Please contact David Sagan if you want to use it.

11.22 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
```

11.23 reinitialize

The `reinitialize` command reinitializes various things. Format:

```
  reinitialize beam
  reinitialize data
  reinitialize tao {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 section §2.1 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*.

Examples:

```
  reinit tao                         ! Reinit using previous arguments
  reinit tao -init tao_special.init  ! Reinitializes Tao with the initialization file
                                     !   tao_special.init
```

11.24 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` (§11.27.20) 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:

```
de            ! Differential Evolution
geodesic_lm   ! ''Geodesic'' Levenburg-Marquardt
lm            ! Levenburg-Marquardt from Numerical Recipes
lmdif         ! Levenburg-Marquardt
```

See 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
```

11.25 scale

The `scale` command scales the vertical axis of a graph or set of graphs. Format:

```
scale {-y} {-y2} {-gang} {-nogang} {<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` (§10.10.2) 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.

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              ! Scale everything
```

11.26 set

The `set` command is used to set values for data, variables, etc. Format:

```
  set beam_init {n@}<component> = <value>                  ! §11.26.1
  set beam_start {n@}<coordinate> = <value>                ! §11.26.2
  set bmad_com <component> = <value>                       ! §11.26.3
  set csr_param <component> = <value>                      ! §11.26.4
  set curve <curve> <component> = <value>                  ! §11.26.5
  set data <data_name>|<component> = <value>               ! §11.26.6
  set default <parameter> = <value>                        ! §11.26.7
  set element <element_list> <attribute> = <value>         ! §11.26.8
  set floor_plan <component> = <value>                     ! §11.26.9
  set geodesic_lm <component> = <value>                    ! §11.26.10
  set global <component> = <value>                         ! §11.26.11
  set graph <graph> <component> = <value>                  ! §11.26.12
  set key <key> = <command>                                ! §11.26.13
  set lat_layout <component> = <value>                     ! §11.26.14
  set lattice {n@}<destination_lat> = <source_lat>         ! §11.26.15
  set opti_de_param <component> = <value>                  ! §11.26.16
  set plot <plot> <parameter> = <value>                    ! §11.26.17
  set plot_page <parameter> = <value1> {<value2>}          ! §11.26.18
  set ran_state = <random_number_generator_state>          ! §11.26.19
  set universe <what_universe> <on/off>                    ! §11.26.20
  set universe <what_universe> recalculate                 ! §11.26.20
  set universe <what_universe> mat6_recalc <on/off>        ! §11.26.20
  set universe <what_universe> track_recalc <on/off>       ! §11.26.20
  set variable <var_name>|<component> = <value>            ! §11.26.21
  set wave <component> = <value>                           ! §11.26.22
```

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 (§11.27) 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
```

11.26.1 set beam_init

Format:
```
  set beam_init {n@}<component> = <value>
```
For `set beam_init`, the `<component>`s that can be set can be found in section §10.5. 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 (§11.27) to see the current values of the `beam_init` structure.

Examples:
```
  set beam_init 3@center(2) = 0.004  ! Set px center of beam for universe 3.
  set beam_init [1,2]@sig_e = 0.02   ! Set sig_e for universes 1 and 2.
```

11.26.2 set beam_start

Format:
```
  set beam_start {n@}<coordinate> = <value>
```
The optional `n@` universe specification (§3.3) may be used to specify the universe or universes to apply the set command to.

For lattices with an open geometry, `set beam_start <coordinate> <number>` can be used to vary the starting coordinates for single particle tracking or the centroid coordinates for beam tracking. Here `<coordinate>` is one of:
```
  x, px, y, py, z, pz, t
```
For photons, `<coordinate>` may also be:
```
  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`.

To see the values for `beam_start` use the command `show element 0`.

Examples:
```
  set beam_start 2@x = 0.001         ! set beginning x position in universe 2 to 1 mm.
```

11.26.3 set bmad_com

Format:
```
  set bmad_com <component> = <value>
```
For `set bmad_com`: The `show global` command will give a list of `<component>`s.

Example:
```
  set bmad_com radiation_fluctuations_on = T ! Turn on synchrotron radiation fluctuations.
```

11.26.4 set csr_param

Format:
```
  set csr_param <component> = <value>
```
Sets coherent synchrotron radiation parameters. Use the `show global -csr_param` command to see a list of `<component>`s.

Example:
```
  set csr_param n_bin = 30  ! Set number of bins used in the csr calc.
```

11.26.5 set curve

Format:

```
set curve <curve> <component> = <value>
```

For `set curve`, the `<component>`s that can be set are:

```
ele_ref_name        = <string>  ! Name of reference element
component           = <string>  ! §10.10.3
ix_branch           = <number>  ! Branch index.
ix_bunch            = <number>  ! Bunch index.
ix_ele_ref          = <number>  ! Index of reference element
ix_universe         = <number>  ! Universe index.
symbol_every        = <number>  ! Symbol skip number.
y_axis_scale_factor = <number>  ! Scaling of y axis
draw_line           = <logical>
draw_symbols        = <logical>
draw_symbol_index   = <logical>
```

See Section §10.10.2 for a description of these attributes. Use the `show curve` (§11.27) to see the settings of the attributes.

Examples:

```
set curve top.x.c1 ix_universe = 2  ! Set universe number for curve
```

11.26.6 set data

Format:

```
set data <data_name>|<component> = <value>
```

For `set data`, the `<component>`s that can be set are:

```
base        ! Base model value
design      ! Design model value
meas        ! Measured data value.
ref         ! Reference data value.
weight      ! Weight for the merit function.
exists      ! Valid datum for computations?
good_meas   ! A valid measurement has been taken?
good_ref    ! A valid reference measurement has been taken?
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.
merit_type  ! How merit contribution is calculated.
```

Besides a numeric value `<value>` can be any of the above along with:

```
meas        ! Measured data value.
```

Examples:

```
set data *|ref = *|meas            ! Set ref data = measured in current universe.
set data 2@orbit.x|base = 2@orbit.x|model
                                   ! Set the base orbit.x in universe 2 to model
set data beta.x[10]|weight = 1e-5  ! Set weight of datum.
```

11.26.7 set default

Format:

```
set default <parameter> = <value>
```

The parameters that can be set are:

```
branch            ! See: §3.4
universe          ! See: §3.3
```

Use the `show global` (§11.27) command to see the current default values.

Example:

```
set default universe = 3
```

11.26.8 set element

Format:

```
set 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.

Note: If an element in the `<element_list>` does not sepcify 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.
```

11.26.9 set floor_plan

Format:

```
set floor_plan <component> = <value>
```

Sets parameters for `floor_plan` plots (§10.10.8). Possible `<components>` are:

```
<shape_name>%<shape_component>
draw_beam_chamber_wall
beam_chamber_wall_scale
```

Where `<ele_shape_name>` is of the form "`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 shape2%draw = F   ! Veto drawing of ele_shape(2)
set floor_plan beam_chamber_scale = 0.5
```

11.26.10 set geodesic_lm

Format:
```
  set geodesic_lm <component> = <value>
```
For `set geodesic_lm`: The `show optimizer geodesic_lm` command will give a list of `<component>`s.

Example:
```
  set geodesic_lm imethod = 10
```

11.26.11 set global

Format:
```
  set global <component> = <value>
```
For `set global`: The `show global` command will give a list of `<component>`s.

Example:
```
  set global n_opti_loops = 30  ! Set number of optimization cycles
  set global rf_on = T          ! Turn on the RF cavities.
```

11.26.12 set graph

Format:
```
  set graph <graph> <component> = <value>
```
For `set graph`, the `components` that can be set are:
```
  component   = <string>     ! §10.10.3
  clip        = <logical>
  ix_universe = <number>
  margin      = <qp_rect_struct>
  x           = <qp_axis_struct>
  y           = <qp_axis_struct>
  y2          = <qp_axis_struct>
```
For setting the `component` attribute see also the commands:
```
  plot                    ! §11.15
  set plot component      ! §11.26.17
  set curve component     ! §11.26.5
```
Example:
```
  set graph orbit.x component = model - design
                          ! Plot model orbit - design orbit in the graph
```

11.26.13 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
```

11.26.14 set lat_layout

Format:
```
  set lat_layout <component> = <value>
```
Sets parameters for `lat_layout` plots (§10.10.8). 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 shape2%draw = F  ! Veto drawing of shape #2
```

11.26.15 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.
```

11.26.16 set opti_de_param

Format:
```
  set opti_de_param <component> = <value>
```
For `set opti_de_param`: The `show global` command will give a list of `<component>`s.

Example:
```
  set opti_de_param binomial_cross = T  ! Use binomial crossovers
```

11.26.17 set plot

Format:

```
  set plot <plot_or_region> <parameter> = <value>
```

For `set plot`, `<component>`s that can be set are:

```
  autoscale_x = <logical>
  autoscale_y = <logical>
  visible     = <logical>
  component   = <string>    ! §10.10.3
```

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 relavent when *Tao* is interfaced to a `gui` (§13.3).

Note: If the `component` parameter is set, the `<value>` is stored in each of the graphs of the plot since the `component` attribute is associated with individual graphs and not the plot as a whole. Other commands that involve `component` are:

```
  plot                    ! §11.15
  set graph component     ! §11.26.12
  set curve component     ! §11.26.5
```

Example:

```
  set plot orbit visible = F        ! Hide orbit plot
  set plot beta component = design  !
```

11.26.18 set plot_page

Format:

```
  set plot_page <component> = <value1> {<value2>}
```

For `set plot_page`, the `<component>`s that can be set are:

```
  title        = <string>          ! Set the plot title text
  subtitle     = <string>          ! Set the subtitle text
  subtitle_loc = <number> <number> ! Set the subtitle location (%PAGE)
```

The `subtitle_loc` component can be used to place the subtitle anywhere on the plot page. This can be useful for referencing a noteworthy part of a graph data.

Example:

```
  set plot_page title = 'XYZ'  ! Set plot page title string
```

11.26.19 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.

11.26.20 set universe

Format:

```
        set universe <what_universe> <on/off>
        set universe <what_universe> recalculate
        set universe <what_universe> mat6_recalc <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_recalc <on/off>
```

The `set universe <what_universe> <on/off>` command will turn the specified universe(s) on or off. Turning a universe off is useful to speed up lattice calculations when this universe is not being used. Or, if many changes are to be performed to a universe and there is no need to do any lattice calculations between commands then turning off all universes will speed things up. To specify the currently default universe (§3.3), you can use `-1` as an index. To specify all universes, use `*`.

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 §10.9. To enable the dynamic aperture calculation at startup, set the `design_lattice(i)%dynamic_aperture` component (§10.3).

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. 6.9 and Eq. 6.10 in the `normal.` data type. To enable the map calculation at startup, set the `design_lattice(i)%one_turn_map_calc` component (§10.3).

The commands

```
  set universe <what_universe> mat6_recalc  and
  set universe <what_universe> track_recalc
```

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 (Warning: The transfer matrices are needed to compute the Twiss parameters). Use the `show universe` command to see the state of these switches are.

Example:

```
  set universe 1 off
  set universe -1 on    ! Set on currently default universe.
  set universe * recalc ! Recalculate in all universes.
```

11.26.21 set variable

Format:

```
  set variable <var_name>|<component> = <value>
```

For `set var`, the `<component>`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.
```

11.26.22 set wave

Format:

```
  set wave <component> = <value>
```

The `set wave` command sets the boundaries of the A and B regions for the wave analysis (§9). The components 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 (§11.34) first to setup the display of the wave analysis.

11.27 show

The `show` command is used to display information. Format:

```
  show {-append <file_name>} {-noprint} <what_to_show>
  show {-write <file_name>} {-noprint} <what_to_show>
```

`<what_to_show>` may be one of:

```
  show alias                                                  ! §11.27.1
  show beam {<element_name_or_index>}                         ! §11.27.2
  show branch {-universe <universe>}                          ! §11.27.3
  show building_wall                                          ! §11.27.4
  show constraints                                            ! §11.27.5
  show curve {-line} {-no_header} {-symbol} <curve_name>      ! §11.27.6
  show data {<data_name>}                                     ! §11.27.7
  show derivative <data_name(s)> <var_name(s)>                ! §11.27.8
  show dynamic_aperture                                       ! §11.27.9
  show element {-attributes} {-base} {-data} {-design} {-all} {-field}
        {-floor_coords} {-no_slaves} {-ptc} {-taylor} {-wall}
        {-xfer_mat} <ele_name>                                ! §11.27.10
  show field <ele> <x> <y> <z> {<t>}                            ! §11.27.11
  show global {-bmad_com} {-csr_param} {-optimization} {-ran_state}   ! §11.27.12
```

```
show graph <graph_name>                                          ! §11.27.13
show history {-no_num} {<num_to_display>}                      ! §11.27.14
show hom                                                         ! §11.27.15
show key_bindings                                                ! §11.27.16
show lattice {-0undef} {-all} {-attribute <attrib>} {-base}
      {-blank_replacement <string>} {-branch <name_or_index>}
      {-custom <file_name>} {-design} {-floor_coords} {-lords} {-middle}
      {-no_label_lines} {-no_tail_lines} {-no_slaves_lines} {-orbit}
      {-radiation_integrals} {-remove_line_if_zero <column #>} {-s <s1>:<s2>}
      {-spin} {-tracking_elements} {<elements>}                  ! §11.27.17
show matrix {-s} {loc1 {loc2}}                                   ! §11.27.18
show normal_form {-order <n_order>} <type>                     ! §11.27.19
show optimizer {-geodesic_lm}                                  ! §11.27.20
show particle {-bunch <bunch_name_or_index>} {-particle <particle_index>
      {-element <element_name_or_index>} {-lost} {-all}        ! §11.27.22
show plot {-floor_plan} {-lat_layout } {<plot_region_name>}
      {<template_plot_name>}                                   ! §11.27.23
show taylor_map {-order <n_order>} {-s} {loc1 {loc2}}          ! §11.27.25
show top10 {-derivative}                                       ! §11.27.24
show twiss_and_orbit {-base} {-branch <name_or_index>} {-design}
      {-universe <ix_uni>} <s_position>                        ! §11.27.26
show universe {universe_number}                                ! §11.27.27
show use                                                       ! §11.27.28
show value <expression>                                        ! §11.27.29
show variable {-no_label_lines} {-universe <universes>}
       {-good_opt_only} {-bmad_format} {<var_name>}             ! §11.27.30
show wakes                                                     ! §11.27.31
show wall {-branch <name_or_index>} {-section <index>} {-angle <angle>}
      {-s <s1>:<s2>} {<n1>:<n2>}                               ! §11.27.32
show wave                                                      ! §11.27.33
```

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 orb.dat orbit    ! Write orbit data to the file "orb.dat".
```

The `-noprint` option suppresses printing and is useful when writing large amounts of data to a file.

Note: When running *Tao* as a subprocess, consider using the `python` command (§11.17) instead of the `show` command for communicating with the parent process.

11.27.1 show alias

Syntax:

```
show alias
```

Shows a list of defined aliases. See the `alias` command for more details.

11.27.2 show beam

Syntax:

```
show beam {<element_name_or_index>}
```

If `<element_name_or_index>` is absent, `show beam` shows parameters used with beam tracking including the number of particles in a bunch, etc. If `<element_name_or_index>` is present, `show beam` will show beam parameters at the selected element. Also see `show particle`. Use the `set beam_init` command to set values of the `beam_init` structure.

11.27.3 show branch

Syntax:

```
show branch {-universe <universe>}
```

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
```

11.27.4 show building_wall

Syntax:

```
show building_wall
```

List all building wall (§10.8) sections along with the points that define the sections.

For vacuum_chamber, capillary, and diffraction_plate walls use the "show wall" command.

11.27.5 show constraints

Syntax:

```
show constraints
```

Lists data and variable constraints.

11.27.6 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.

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".
```

11.27.7 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
```

11.27.8 show derivative

Syntax:

```
show derivative <data_name(s)> <var_name(s)>
```

Shows the derivative dModel_Value/dVariable. This derivative is used by the optimizers `lm` and `svd`. Note: Wild card characters can be used to show multiple derivatives.

Example:

```
show deriv orbit.x[23] k1[34] ! Show dModel_Value/dVariable Derivative.
show deriv * *                ! Show all derivatives. Warning! The output
                              ! may be large.
```

11.27.9 show dynamic_aperture

Syntax:

```
show dynamic_aperture
```

Shows parameters and results of the dynamic aperture calculation.

11.27.10 show element

Syntax:

```
show element {-attributes} {-base} {-data} {-design} {-all} {-field}
    {-floor_coords} {-no_slaves} {-ptc} {-taylor} {-wall} {-xfer_mat} <ele_name>
```

This shows information on lattice elements. The syntax for `<ele_name>` is explained in section §4.1. 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_slaves` switch is present, `super_slave` and `multipass` slave elements will not be listed.

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
-field
-floor_coords
-taylor
-wall
-xfer_mat
```

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.
```

11.27.11 show field

Syntax:

```
show field <ele> <x> <y> <z> {<t>}
```

The `show field` command shows the electric and magnetic field at a point in space-time. The `<z>` coordinate is with respect to the beginning of the element specified by `<ele>`. The syntax for `<ele>` is explained in section §4.1. In this case, `<ele>` must specify a single element. The `<t>` argument is optional and will be set to zero if not specified.

11.27.12 show global

Syntax:

```
show global {-bmad_com} {-csr_param} {-optimization} {-ran_state}
```

The `show global` command, used without any optional switches, shows information on the global parameter structure (§10.4). The optional switches are

-bmad_com
Displays `bmad_com` components (§10.4).

-csr_param
Displays `csr_param` components (§10.4).

-optimization
Displays optimization parameters.

-ran_state
Displays the state of the random number generator. Use this with the `set ran_state` command.

11.27.13 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.

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".
```

11.27.14 show history

Syntax:

```
  show history {-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.

The number of commands printed is, by defulat, 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` (§11.20) to re-execute a command.

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.

Examples

```
  show -write cmd_file hist all -no    ! Create a command history file
  show hist 30                         ! Show the last 30 commands.
```

11.27.15 show hom

Syntax:

```
  show hom
```

Shows long–range higher order mode information for linac accelerating cavities.

11.27.16 show key_bindings

Syntax:

```
  show key_bindings
```

Shows all key bindings.

11.27.17 show lattice

Syntax:

```
  show lattice {-0undef} {-all} {-attribute <attrib>} {-base}
      {-blank_replacement <string>}  {-branch <name_or_index>}
      {-custom <file_name>} {-design} {-floor_coords} {-lords} {-middle}
      {-no_label_lines} {-no_tail_lines} {-no_slaves} {-orbit} {-radiation_integrals}
      {-remove_line_if_zero <column #>} {-s <s1>:<s2>} {-spin} {-tracking_elements}
      {<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.

The `-all` switch shows all tracking and lord elements.

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 beam_start to vary the starting position in this case. If the -spin switch is also present, the particle's spin will also be displayed.

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 beam_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.

The -radiation_integrals switch, if present, will cause the radiation integrals for each lattice element to be displayed instead of the standard Twiss and orbit data.

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. The -no_tail_lines just suppress the header lines at the bottom of the table.

If present, the -no_slaves switch will veto from the list of element to print all elements that are either a super slave or a multipass slave.

If present, the -lords switch will print a list of lord elements only.

If present, the -floor_coords switch will print the global floor (laboratory) coordinates for each element.

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).

The locations to show are specified either by specifying a longitudinal position range with -s, or by specifying a list <element_list> of elements. The syntax used for specifying this list is given in section §4.1. In this case there should be no blank characters in the list.

The -tracking_elements switch can be used to show all the elements in the tracking part of the lattice. If neither -tracking_elements, nor a range is given, the first 200 elements are shown.

To customize the output use the command show lattice -custom <file_name>. A customization file looks like:

```
  &custom_show_list
    column(1)  = "#",                        "i6",     6
    column(2)  = "x",                        "x"       1   ! blank space
    column(3)  = "ele::#[name]",             "a",      0
    column(4)  = "ele::#[key]",              "a16",   16
    column(5)  = "ele::#[s]",                "f10.3", 10
    column(6)  = "ele::#[l]",                "f10.3", 10
    column(7)  = "ele::#[beta_a]",           "f7.2",   7
    column(8)  = "1e3 * ele::#[orbit_x]",    "f8.3",   8, "Orbit_x| (mm)"
    column(9)  = "lat::unstable.orbit[#]",   "f8.3",   8
    column(10) = "beam::n_particle_loss[#]", "i8",     8
  /
```

each column(1) has four components. The first component is what is to be displayed in that column. Algebraic expressions are permitted (§4.2). Note: Use of ele:: and beam::, etc sources is accepted but these constructs cannot be evaluated at the center of an element. That is the -middle switch will have no effect on such constructs.

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. The default format is `es12.4`. Example descriptors:

```
  From   Example
  ----   -------
  Fw.d   F13.5    ! For real numbers. No exponent.
  ESw.d  ES13.4   ! For real numbers. Has exponent
  Lw     L4       ! For logical numbers
  Iw     I8       ! For integers
  wX     4x       ! Whitespace.
```

In the above, "w" is the width of the field and "d" is the number of digits to the right of the decimal. 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.

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).
```

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` switch is present. In this case, a zero, "0", will be printed.

Additionally, The `-blank_replacement <string>` 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 -- 1:100
```

This will replace any blank fields with "–".

Note: Data can be used in custom output but data is always evaluated at the exit end of an element even when the `-middle` switch is used.

The second component is the Fortran edit descriptor. The third column is the total width of the field. 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 component is column title name. This component 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.

If present, the `-remove_line_if_zero <column #>` 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.

Example:

```
  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 markers 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.
```

11.27.18 show matrix

Syntax:

```
  show matrix {-s} {loc1 {loc2}}
```

Shows the transfer matrix for the `model` lattice of the default universe (set by `set default universe`). This command is equivalent to `show taylor_map -order 1`. See `show taylor_map` for documentation on the `show matrix` arguments.

11.27.19 show normal_form

Syntax:

```
  show normal_form {-order <n_order>} <type>
```

Shows normal form taylor maps (optionally truncated to <n_order>) from Eq. 6.9 and Eq. 6.10. <type> can be: `M`, `A`, `A_inv`, `dhdj`, `F`, `L`.

11.27.20 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`.

11.27.21 show opt_vars

Syntax:

```
  show opt_vars
```

Shows the settings of the variables used in the optimization using the Bmad standard lattice input format.

11.27.22 show particle

Syntax:

```
    show particle {-bunch <bunch_index>} {-particle <particle_index>
    {-element <element_name_or_index>} {-lost} {-all}
```

Shows individual particle information. The default for the optional `-bunch` index is set by the global variable `global%bunch_to_plot`. The default `-element` is 0 (the starting position). Also see `show beam`.

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.

Examples:

```
  show part -bun 3 -part 47 -ele 8 ! Shows information on particle #47 of
                                   !   bunch #3 at lattice element #8.
  show part -bun 47 -part 8        ! Same as above except the default bunch is used.
  show part -lost -bun 3           ! Show lost particle positions for bunch #3
```

11.27.23 show plot

Syntax:

```
  show plot {-floor_plan} {-lat_layout } {<plot_or_template_name>}
```

A simple `show plot` displays which templates are being plotted and in which regions and also all available templates. See §7 for the syntax on plot, graph, and curve names. A `show plot <plot_or_region_name>` will display information on a particular plot.

The `show plot -floor_plan` and `show plot -lat_layout` commands show the parameters associated with the `floor_plan` or `lat_layout` plots (§10.10.8). Use the `set floor_plan` or `set lat_layout` commands to set these parameters.

Also see `show graph` and `show_curve`.

Examples:

```
  show plot      ! General plot info
  show plot r13
```

11.27.24 show top10

Syntax:

```
  show top10 {-derivative}
```

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.

Note: To set the number of top contributors shown, use the command `set global n_top10 = nnn` where `nnn` is the desired number to be shown.

11.27.25 show taylor_map

Syntax:

```
show taylor_map {-order <n_order>} {-s} {loc1 {loc2}}
```

Shows the Taylor transfer map for the `model` lattice of the default universe (set by `set default universe`). See also `show matrix`.

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`

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.

Examples:

```
show taylor -order 1 q10w q10e  ! 0th and 1st order maps from q10e to q10w
show taylor 45                  ! Transfer map of element #45
show taylor -s 13 23            ! Transfer map from s = 13 meters to 23 meters.
```

11.27.26 show twiss_and_orbit

Syntax:

```
  show twiss_and_orbit {-base} {-branch <name_or_index>} {-design}
  {-universe <ix_uni>} <s_position>
```

Show Twiss and orbit information at a given longitudinal position `<s_position>`.

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 root branch (branch 0) is the default.

Examples:

```
show twiss -uni 2 23.7     ! Show parameters in universe 2 at s = 23.7 meters.
```

11.27.27 show universe

Syntax:

```
show universe {universe_number}
```

Shows various parameters associated with a given

universe. If no universe is specified, the current default universe is used. Parameters displayed include tune, chromaticity, radiation integrals, etc.

11.27.28 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.

11.27.29 show value

Syntax:

```
  show value <expression>
```

Shows the value of an expression. Examples:

```
  show value sqrt(3@lat:orbit.x[34]|model) + sin(0.35)
```

11.27.30 show variable

Syntax:

```
  show variable {-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 components 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.
```

11.27.31 show wakes

Syntax:

```
  show wakes
```

The `show wakes` 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 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 wakefieds is desired, a different program have to be used.

11.27.32 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.
```

11.27.33 show wave

Syntax:
```
  show wave
```
The `show wave` command shows the results of the current wave analysis (§9).

11.28 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”.

11.29 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.

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.
```

11.30 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 stop
```

Note: `timer start` will toggle `beam timing` off.

11.31 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.
```

11.32 value

The `value` command is used to evaluate a value. This command is in development...

11.33 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.

11.34 wave

The `wave` command sets what data is to be used for the wave analysis (§9). Format:

```
wave <curve> {<plot_location>}
```

The `<curve>` argument specifies what plot curve is to be used in the analysis. The specified curve must be visible in the plot window. Possible `<curve>`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.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 `<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` (§11.26.22) command to set the boundries 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.
```

11.35 write

The `write` command creates various files. Format:

```
write 3d_model {<file_name>}           ! Write a blender script for a 3D lattice display.
write bmad_lattice {<file_name>}       ! Write a Bmad lattice file of the model
write beam {-ascii} -at <element_list> {<file_name>}
                                       ! Write beam distribution data (§10.5).
write blender {<file_name>}            ! Write a blender script (Same as 3d_model).
write covariance_matrix {file_name}    ! Write the covariance and alpha matrices
                                       !   from the Levenburg (lm) optimization.
write curve <curve_name> {<file_name>} ! Write the curve data
write derivative_matrix {file_name}    ! Write the dModel_Data/dVar matrix.
write digested {<file_name>}     ! Write a digested Bmad lattice file of the model.
write gif {<file_name>}          ! create a gif file of the plot window.
```

```
  write hard                        ! Print the plot window to a printer.
  write hard-l                      ! Like "hard" except use landscape orientation.
  write mad8_lattice {<file_name>}  ! Write a MAD-8 lattice file of the model
  write madx_lattice {<file_name>}  ! Write a MAD-X lattice file of the model
  write ps {-scale <scale>} {<file_name>}
                                    ! Create a PS file of the plot window.
  write ps-l {-scale <scale>} {<file_name>}
                                    ! Create a PS file with landscape orientation.
  write ptc {-all} {-old} {-branch <name_or_index} {<file_name>}
  write variable {-good_var_only} {<file_name>}
                                    ! Create a Bmad file of variable values.
```

If `<file_name>` is not given then the defaults are:

Command	Default File Name
write 3d_model	blender_lat_#.py
write bmad_lattice	lat_#.bmad
write beam	beam_#.dat
write blender	blender_lat_#.py
write curve	curve
write derivative_mat	derivative_matrix.dat
write digested	lat_#.digested
write gif	tao.gif
write mad8_lattice	lat_#.mad8
write madx_lattice	lat_#.madx
write ps	tao.ps
write ptc	ptc.flatfile
write variable	global%var_out_file

where `#` is replaced by the universe number. `write curve` will produce two or three files:

```
  <file_name>.symbol_dat    ! Symbol coordinates file
  <file_name>.line_dat      ! Curve coords.
  <file_name>.particle_dat  ! Particle data file
```

The particle data file is only produced if particle data is associated with the curve. The curve coordinates are the the set of points that are used to draw the (possibly smooth) curve through the symbols.

For `ps` and `ps-l`, the optional `-scale` switch sets the scale for the postscript file. The default is 0 which autoscales to fit an 8-1/2 by 11 sheet of paper. A value of 1.0 will result in no scaling, 2.0 will double the size, etc.

The `write 3d_model` or `write blender` creates a script which can then be run by the `blender` program[Blender]. `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:

```
  https://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 environmental variables:

```
BMAD_BASE_DIR
DIST_BASE_DIR
ACC_RELEASE_DIR
```

Generally, one of the latter two environmental variables will be defined. If not, a copy of the *Bmad* directory must be created and then `BMAD_BASE_DIR` be appropriately defined.

The `write variable` command has an optional `-good_var_only` switch. If present, only the information on variables that are currently used in the optimization is written.

`write beam` will create a file of the particle positions when beam tracking is being used. The `-at` switch specifies at what elements the particle positions are written. Element list format (§4.1), without any embedded blanks, is used for the `<element_list>` argument to the `-at` switch. The `-ascii` switch is for writing text files. The default is to write with a compressed binary format. Note: Beam files can be used to initialize *Tao* (§2.1). Example

```
write beam -at *   ! Output beam at every element.
```

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 lattice branch # 0. 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.

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.

11.36 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
```

11.37 x_scale

The `x_scale` command scales the horizontal axis of a graph or set of graphs. Format:

```
  x_scale {-gang} {-nogang} {<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 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 is present then an `autoscale` will be invoked to give the largest bounds commensurate with the data. If an autoscale is performed upon an entire plot, and if `plot%autoscale_gang_x` (§10.10.2) 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.

Note: The `x_scale` command will vary the number of major divisions (set by plotcan be that if two plots have the same range of data but differing major division settings, the `x_scale` command can produce differing results.

Example:

```
  x_scale                ! Autoscale all x-axes.
  x_scale * 0 100        ! Scale all x-axes to go from 0 to 100.
```

11.38 xy_scale

The `xy_scale` command sets horizontal and vertical axis bounds. Format:

```
  xy_scale {<where>} {<value1> }<value2>}}}
```

`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.

Example:

```
  xy_scale          ! Autoscale all axes.
  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.

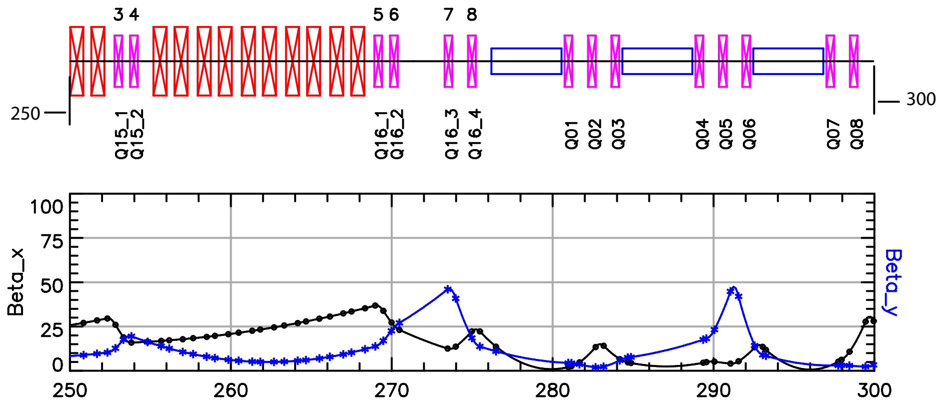
From `line mode` use the `single_mode` command (§11.28) to get into `single mode`. To go back to `line mode` type "`Z`".

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. 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 §10.6). After startup, associated variables with keyboard keys can be done using the `set variable` command (§11.26).

The variables are divided into banks of 10. The 0^{th} bank uses the first ten variables that have their `key_bound` attribute (§10.6) set to True. the 1^{st} 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 (§10.10.12)can be setup as shown in Figure 12.1. 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	("(")



```
     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 12.1: 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.

10 + 10*ib	P	p	0	shift-0	(")")

In the above table ib is the bank number (0 for the 0^{th} bank, etc.), and the change is in multiples of the `step` (§10.6. value for a variable. Note: In `line mode`, the command `show key_bindings` (§11.27) may be used to show the entire set of bound keys.

Initially the 0^{th} 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 12.1, the `"1:"` in the upper left corner of the `Key Table` shows that the 1^{st} 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 §11.26.

? 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 (§8.5). 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 `plot%x%min` and `plot%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/GUI Interface

13.1 Python Interface Via Pexpect

A python module, `tao_pipe.py`, for interfacing *Tao* to `Python` is provided in the `tao/python` directory.

The `tao_pipe` module uses the `pexpect` module. 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 `http://www.noah.org/wiki/pexpect`.

Example:

```
  >>> import tao_pipe                                        # import module
  >>> p = tao_pipe.tao_io("../bin/tao -lat my_lat.bmad")    # init session
  >>> p.cmd_in("show global")                # Command to Tao
  >>> print(p.output)                        # print the output from Tao
  >>> p.cmd("show global")                   # Like p.cmd_in() excepts prints the output too.
```

After each call to `tao_io.cmd` and `tao_io.cmd_in`, the `tao_io.output` variable is set to the multiline output string returned by *Tao*. To chop this string into lines, use the splitlines() string method.

To get information from *Tao* into Python, the output from *Tao*, contained in `tao_io.output`, needs to be parsed. For long term maintainability of python scripts, use the `python` (§11.17) command as opposed to the `show` command . See the `python` command for more details.

13.2 Tao Python command

13.3 Plotting Issues

When using *Tao* with a `gui`, and when the `gui` is doing the plotting, the `-noplot` option (§2.1) should be used when starting *Tao*. The `-noplot` option prevents *Tao* from opening a plotting window.

Even though *Tao* is not displaying the plot page when the `-noplot` option is used, *Tao* will still calculate the points needed for plotting curves for use by the `gui`. In this case, a few points must be kept in mind: First the names of the default plot regions are simplified to be 'r1', 'r2', etc. Use the `show plot` command (§11.27.23) to view a list. Second, to prevent unneeded computation, the `visible` parameter of template plots that are placed (§11.14) is set to False and must be set to True, using the `set plot` command (§11.26.17), to enable computation of the curve points.

Part II

Programmer's Guide

Chapter 14

Programming Tao

This chapter gives an overview of the coding structure of Tao. Knowledge of this structure is needed in order to create custom versions of *Tao*. See Chapter §15 for details of how to create custom versions.

14.1 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 §15, 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

14.2 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
  end type
```

An instance of this structure called `s` is defined in `tao_struct.f90`:

```
  type (tao_super_universe_struct), save, target :: s
```

Thus `s` is common to all of *Tao*'s routines.

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 §10.4 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.

%v1_var(:)
The `%v1_var(:)` component is an array of all the `v1_var` blocks (§5) that the user has defined. The range of valid blocks goes from 1 to `%n_v1_var_used`.

%var(:) The `%var(:)` array holds a list of all variables (§5) that the user has defined. The range of valid variables goes from 1 to `n_var_used`.

%u(:)
The `%u(:)` component is an array of universes (§3.3).

%mpi
The `%mpi` component holds parameters needed for parallel processing.

%key(:)
The `%key(:)` component is an array of indexes used for key bindings (§12.1). The value of `%key(i)` is the index in the `%var(:)` array associated with the *i*þkey.

%building_wall
The `%building_wall` component holds parameters associated with a building wall (§10.8).

%wave
The `%wave` component holds parameters needed for the wave analysis (§9).

Chapter 15

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.

15.1 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 §15.5 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.`

   Similarly, the command `mkd` will create a debug executable

   `ROOT/debug/bin/custom_tao`

A debug executable only needs to be created if you a debugging the code.

15.2 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 it'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.

15.3 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 (§10.4 or may be set on the command line using the `-hook_init_file` option (§2.1).

15.4 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.

15.4.1 tao_hook_graph_setup

Use this to setup custom graph data for a plot.

15.4.2 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. §15.6 for an example of how to use this hook.

15.4.3 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` sould be set to `False` for all other datums.

15.4.4 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.

15.4.5 tao_hook_init_design_lattice

This will do a custom lattice initialization. The standard lattice initialization just calls `bmad_parser` or `xsif_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.

15.4.6 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 is placed here. A custom lattice calculation can be performed on any lattice separately, this allows for the possibility of, for example, tracking a single particle for one lattice and beams in another.

15.4.7 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.

15.4.8 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.

15.4.9 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.

15.4.10 tao_hook_plot_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).

15.4.11 tao_hook_plot_data_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.

15.4.12 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.

15.5 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 (§15.1).

The formula for single particle emittance is

$$\epsilon = \gamma x^2 + 2\alpha x x' + \beta x'^2 \tag{15.1}$$

Place the following code in `tao_hook_evaluate_a_datum.f90` in the `case select` construct (also add the necessary type declarations)

```
  type (coord_struct), pointer :: orbit(:)
  ...
  orbit => tao_lat%tao_branch(0)%orbit
  ...
  case ('particle_emittance.x')
    datum_value =  (ele%a%gamma * orbit(ix1)%vec(1)**2 + &
     2 * ele%a%alpha * orbit(ix1)%vec(1) * orbit(ix1)%vec(2) + &
     ele%a%beta * orbit(ix1)%vec(2)**2)

  case ('particle_emittance.y')
    datum_value = (ele%b%gamma * orbit(ix1)%vec(3)**2 + &
     2 * ele%b%alpha * orbit(ix1)%vec(3) * orbit(ix1)%vec(4) + &
     ele%b%beta * orbit(ix1)%vec(4)**2)
```

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 `tao/example` 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%min =   0
    plot%x%max = 100
    plot%x%major_div = 10
    plot%x%label = ' '
    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
    graph%n_curve = 1
    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
    graph%n_curve = 1
    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 15.1.

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`.

15.6 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

```
  tao/examples/custom_tao_with_measured_data
```

The files in this directory are as follows:

CMakeLists.txt, cmake.ping_tao
    Script files for creating `ping_tao`. See Sec. §15.1.

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 (§10.2)

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.f9`
    (§15.4.2).

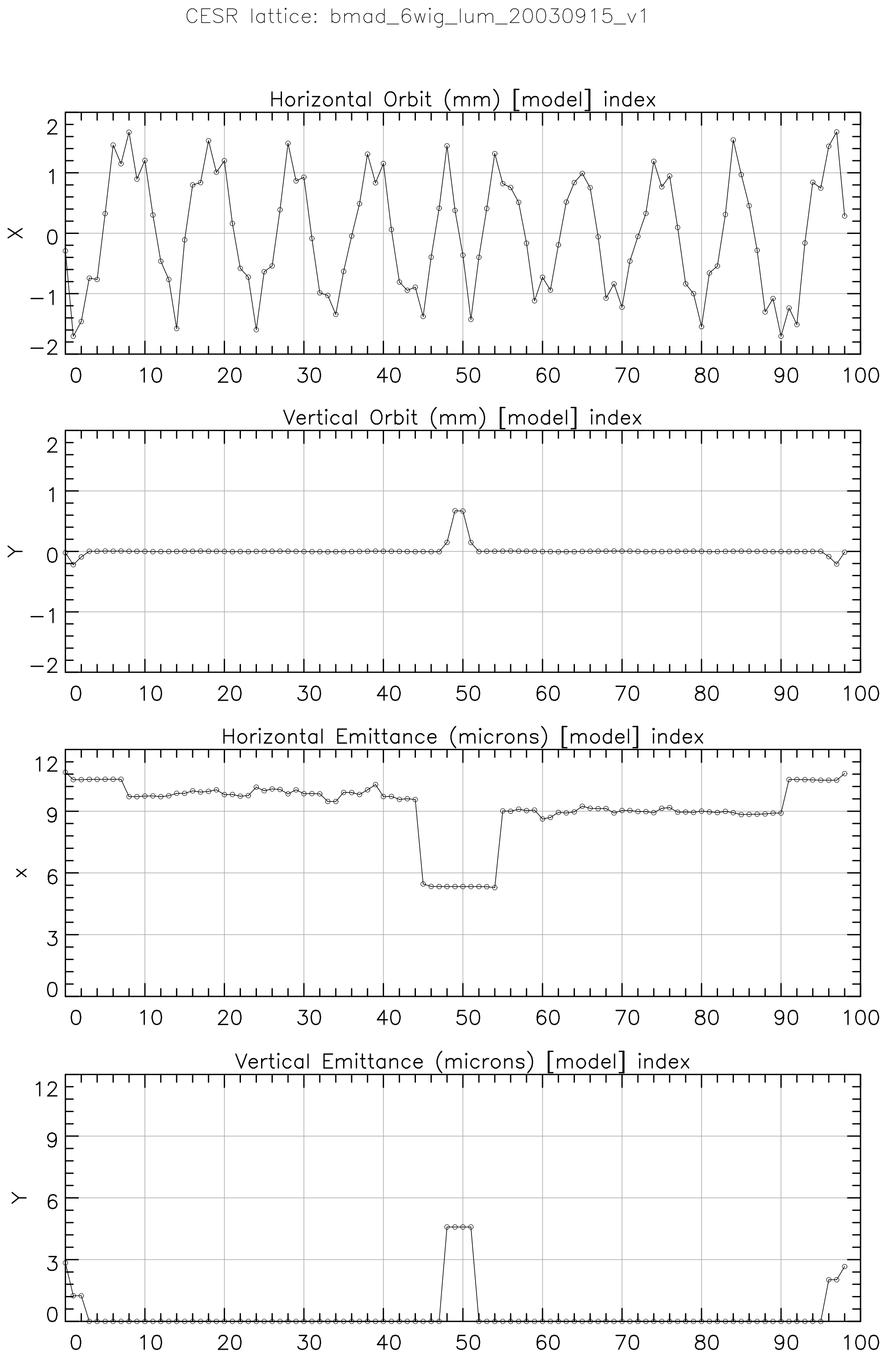


Figure 15.1: Custom data type: non-normalized emittance

tao_plot.init
File for defining plot parameters (§10.10).

tao_program.f90
copy of the `tao/program/tao_program.f90` file (§15.1).

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. §6.8 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`.

15.6.1 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 (§15.4.4).

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)
```

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:
`http://www.lepp.cornell.edu/~dcs/bmad`

[Blender] `Blender` web page:
`https://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).
`http://link.aps.org/doi/10.1103/PhysRevSTAB.14.034002`

[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)

[Saf97] J. Safranek, “Experimental determination of storage ring optics using orbit response measurements”, NIM-A388, p. 27 (1997).

[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).
`http://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).
`http://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).

[Wil00] Klaus Wille, *The Physics of Particle Accelerators: An Introduction*, Translated by Jason McFall, Oxford University Press (2000).

Index