Revision: 9.5
January 30, 2010

# The


# Manual

**By David Sagan**
**and**
**Jeffrey C. Smith**

# 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[4]. *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 three parts. Part I gives an introduction and tutorial to *Tao* with examples of how to set-up tracking, do plotting, etc. Part II 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. Finally, Part III is a programmer's guide which shows how to extend *Tao*'s capabilities and incoporate 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`

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>`

# Contents

# List of Figures

# List of Tables

# Part I

# Concepts and Tutorial

# Chapter 1

# Tao Concepts

*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 *The Physics of Particle Accelerators* by Klaus Wille[1].

*Tao* is based on the *Bmad* [4] 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 commisioning 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*, there is a hands-on tutorial in Chapter §2. After you get more familiar with *Tao*, you might be interested to exploit its versatility by extending *Tao* to do custom calculations. For this, see Chapter 9.

## 1.1 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 (§1.4).

**Data**
Anything that can be measured. For example: The orbit of a particle or the lattice beta functions, etc. (§3)

**Variables**
Essentially, any lattice parameter or initial condition that can be varied. For example: quadrupole strengths, etc. (§1.5).

**Plotting**
Information used to draw graphs, display the lattice floor plan, etc. (§1.8).

**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.

## 1.2 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` (§1.3), 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.

## 1.3 The Universe

The *Tao* `super_universe` (§1.2) contains one or more `universes`. A `universe` contains a `lattice` (§1.4) plus whatever data (§3) 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 (§6.4).

There are several situations in which multiple universes are useful. One case is where there are multiple machines. For example, a transfer line connected to a storage ring. In this case, one universe will correspond to the transfer line and another universe will correspond to the storage ring.

Another 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 (§1.4) 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.]

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 (§1.6). When there is only a single Universe specified, the brackets `[...]` are optional. When the universe prefix is not present, the current "viewed" (default) universe (§7.29) is assumed. The current `viewed` 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 viewed universe.
-1@orbit.x      ! Same as "orbit.x".
*@orbit.x       ! orbit.x data in all the universes.
*@*             ! All the data in all the universes.
```

## 1.4 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) (§6.4). In many instances, this is the particular lattice that one wants the actual physical machine to conform to. The `design` lattice is fixed. Nothing is allowed to vary in this lattice.

**Model Lattice**
: 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 (§7.23) is used to transfer information from the `design` or `model` lattices to the base lattice.

## 1.5 Variables

For the `model` lattice (or lattices if there are multiple `universes`) the `change` command (§7.3) 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 (§6.8) 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 (§2.5) 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[*]|model ! All model 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:

**Measured Value**
: The Value as obtained at the time of the data measurement.

**Reference Value**
: The Value as obtained at the time of the reference data measurement.

**Model Value**
: The value as given in the model lattice.

**Design Value**
: The value as given in the design lattice.

**Base Value**
: The value as given in the base lattice.

These components and others can be refereed to using the notation |name where name is the appropriate name for the component. The list of components that can be set or refereed to are:

```
quad_k1[1]|meas       ! Value at time of data measurement
quad_k1[1]|ref        ! Value at time of the reference data measurement
quad_k1[1]|model      ! Value in the model lattice
quad_k1[1]|base       ! Value in the base lattice
```

```
quad_k1[1]|design     ! Value in  the design lattice
quad_k1[1]|weight     ! Weight used in the merit function.
quad_k1[1]|old        ! Scratch value.
quad_k1[1]|step       ! For fitting/optimization: What is considered a small change.
quad_k1[1]|exists     ! Logical
quad_k1[1]|good_var   ! Logical
quad_k1[1]|good_user  ! Logical
quad_k1[1]|good_opt   ! Logical
quad_k1[1]|good_plot  ! Logical
quad_k1[1]|useit_opt  ! Logical
quad_k1[1]|useit_plot ! Logical
```

Use the `show var` (§7.24) command to view variable information

When using optimization for lattice correction or lattice design (§4), Individual datums can be excluded from the process using the `veto` (§7.28), `restore` (§7.19), and `use` (§7.27) commands. These set the `good_user` component of a datum. 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. The settings of everything but `good_user` is determined by *Tao*

## 1.6 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.<br>[Can include wildcard characters "*" or "%"] | "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 wildcard characters "*" and/or "%" can be used. The "*"wildcard matches any number of characters, The "%" wildcard matches a single character. Thus, 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.

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. Example

```
2>>183   ! Element number 183 of branch # 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 wildcard characters "%" and "*". Essentially this is an extension of the `element name` format. Example:

```
"quad::q*"
```

will match to all quadrupole elements whose name starts with a ''q''.

## 1.7 Arithmetic Expressions

*Tao* is able to handle arithmetic expressions within commands (§7) 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:

| | |
|---|---|
| `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`(x) | Arc Tangent |
| `abs`(x) | Absolute Value |
| `ran`() | Random number between 0 and 1 |
| `ran_gauss`() | Gaussian distributed random number with unit RMS |

Both `ran` and `ran_gauss` use a seeded random number generator. Setting the seed is described in Section §6.5.

Data (§3.1) and variable values (§1.5), along with lattice parameters can be referenced in expressions. The general form of a term for data and variables is

```
{[<universe(s)>]@}<source>::<var_or_dat_name>[<index_list>]{|<component>}
```

For lattice and beam parameters the format is

```
{[<universe(s)>]@}<source>::<param_name>[{<ref_element>&}<element_list>]{|<component>}
```

And for element parameters the format is

```
{<universe_range>@}ele::<element_id>[<param_name>]{|<component>}
{<universe_range>@}ele_mid::<element_id>[<param_name>]{|<component>}
```

`ele::` will evaluate the parameter at the exit end of the element and `ele_mid::` will evalute the parameter at the middle of the element. The components of this syntax are:

```
<universe_range>    Optional universe specification (§1.3)
<var_or_dat_name>   Variable or data name.
```

```
<element_id>        Element identifyer: Name or lattice index.
<source>            Source of the data.
<param_name>        Name of the parameter, datum, or variable.
<index_list>        List of indexes.
<ref_point>         Optional reference element (with lat or beam source only).
<eval_points>       Evaluation point or points.
<component>         Optional component.
```

Examples:

```
3@lat::orbit.x[34:37]          Array of orbits at element 34 through 37 in universe 3.
3@lat::orbit.x[34]|model       Same as above.
beam::sigma.x[q10w]            Beam sigma at element q10w.
lat::n_particle_loss[2&56]     Particle loss between elements 2 and 56.
3@ele::quad::q*[k1]            k1 of all quads in the model lattice of universe 3
                                 with names beginning with "q".
ele::3[l]|design               Length of the design lattice element #3
                                 in the viewed universe.
dat::orbit.x[2:7,8]|model
```

The <source> field may be one of:

```
beam        value is from beam tracking.
dat         Value is from a Tao datum in a data array (§3.1).
lat         Value is from the lattice.
var         Value is from a Tao variable (§1.5).
```

When a term has a dat source, The value of the term comes from a data structure. With a dat source, <var_or_dat_name> is the name of a d2_name.d1_name data array (§3.1) and <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. Finally, the optional <component> indicates what component of the datum is to be used. Possible components are listed in Section §3.3. The default is model.

When a term has a var source, the value of the term comes from a *Tao* variable (§1.5). With a var source, <var_or_dat_name> is the name of a v1_var variable array. Like terms with a dat source, <index_list> is a list of indexes. The optional <component> indicates what component of the variable is to be used. Possible components are listed in Section §1.5. The default is model.

When a term has a lat or beam source, the value of the term comes from evaluation of the lattice or beam tracking. The <param_name> will be a datum type from Table 3.5 with a lat source and Table 3.5 for a beam source. Element list format (§1.6) is used for the element_list. The optional <ref_element> specifies a reference element for the evaluation. For example, "lat::r.56[q0&qa:qb]" is an array of the $r(5,6)$ matrix element of the transport map between element q0 and each element in the range from element qa and qb. The optional <component> indicates what lattice is to be used. Possible components are

```
model (default)
base
design
```

The default is model.

Notice the difference between, say, "lat::orbit.x[10]" and "dat::orbit.x[10]". With the "lat::" source, the element index, in this case 10, refers to the 10th lattice element. With the "dat::" 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 and which may not even refer to orbit data.

When a term has a ele source, the value of the term comes from an element parameter such as the element strength. Element list format (§1.6) is used for the <element_id> so an array of elements can

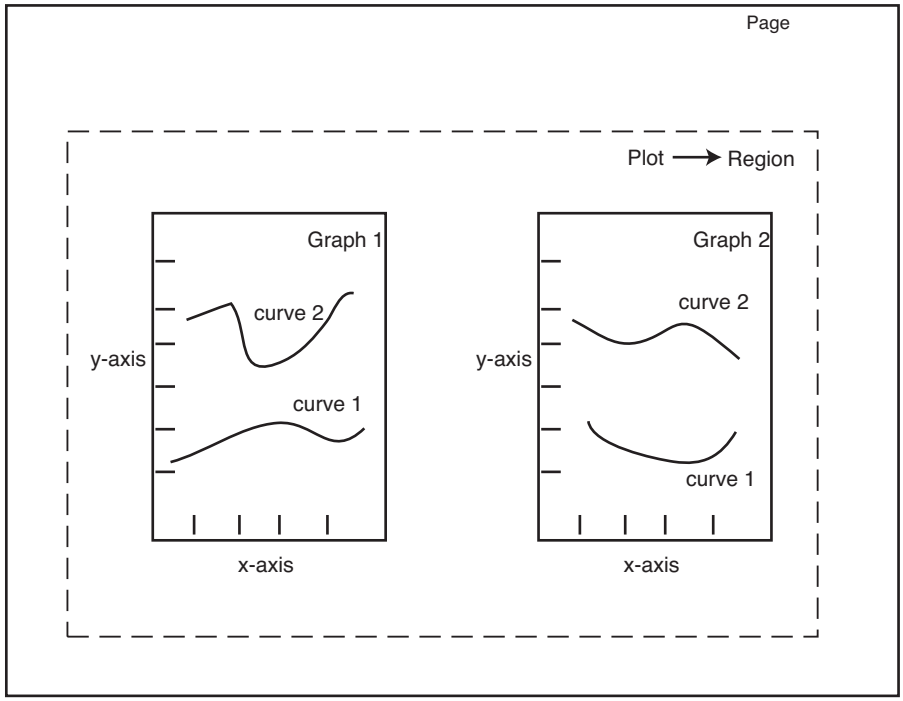



Figure 1.1: A plot has a collection of graphs and a graph has a collection of curves. A plot becomes visible when it is associated with some region on the page using the `place` command. Note that on the actual page the plot/region border is not visible.

be defined. The optional `<component>` indicates what lattice is to be used and is the same as for terms with a `lat` or `beam` source (see above).

Generally, if the `<source>` field is not present, for example, "`orbit.x[10]`", *Tao* will search for a match assuming that the source is either `dat` or `var`. Exceptions will be noted in the manual.

## 1.8 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 6) 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 graphs are laid out, etc. The plot initialization file also defines a set of regions within the page. Any template plot can be placed in any region. Using the place command (see Chapter 7 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 1.1.

Figures 1.2 and 1.3 show examples of a plot page. Figure 1.2 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 plot command one can then specify what components are plotted. component refers to measured, reference, model, base, and/or design data. 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 1.2, 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 1.2 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 4). 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 1.3 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 8. 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`.

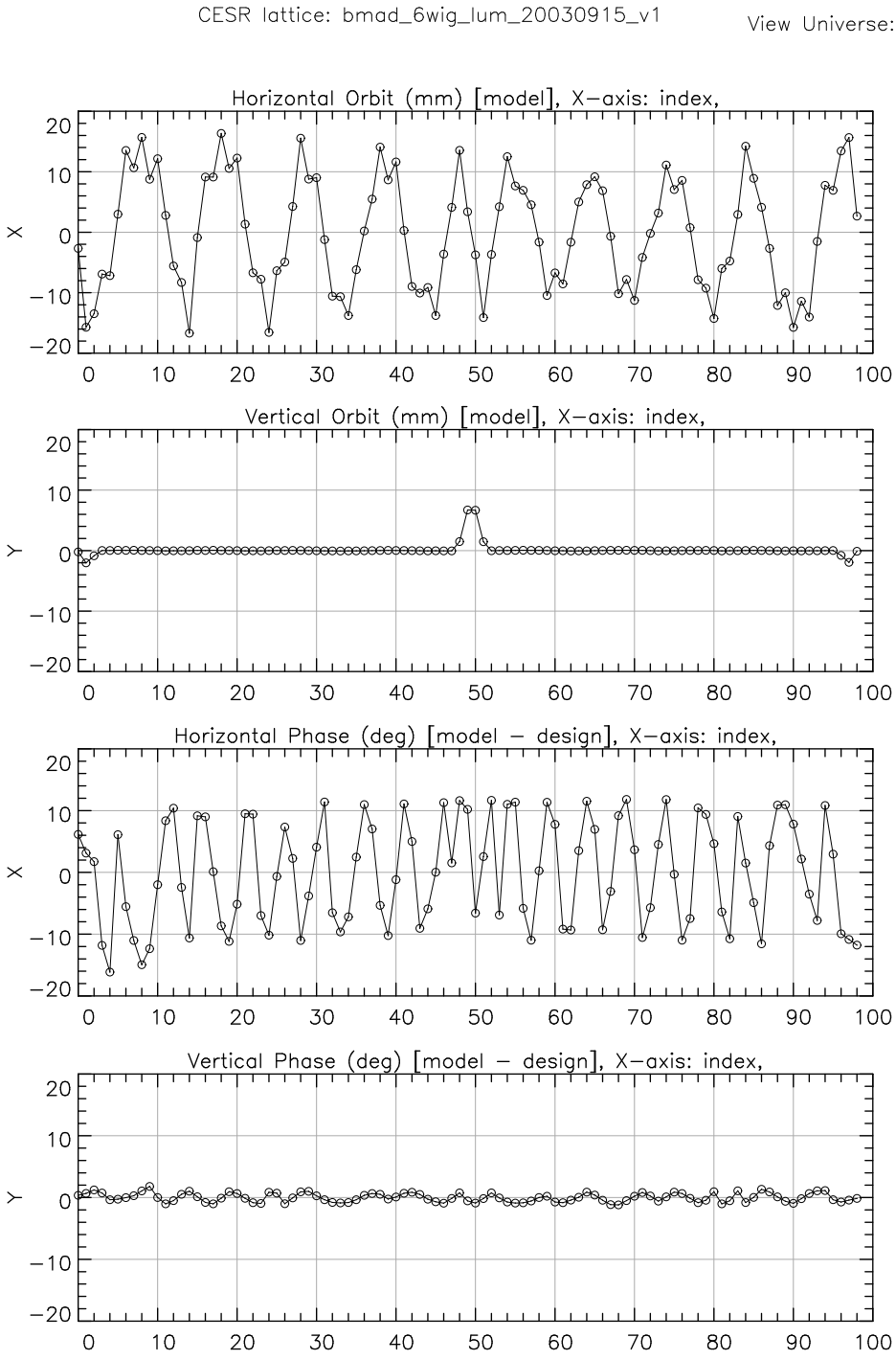
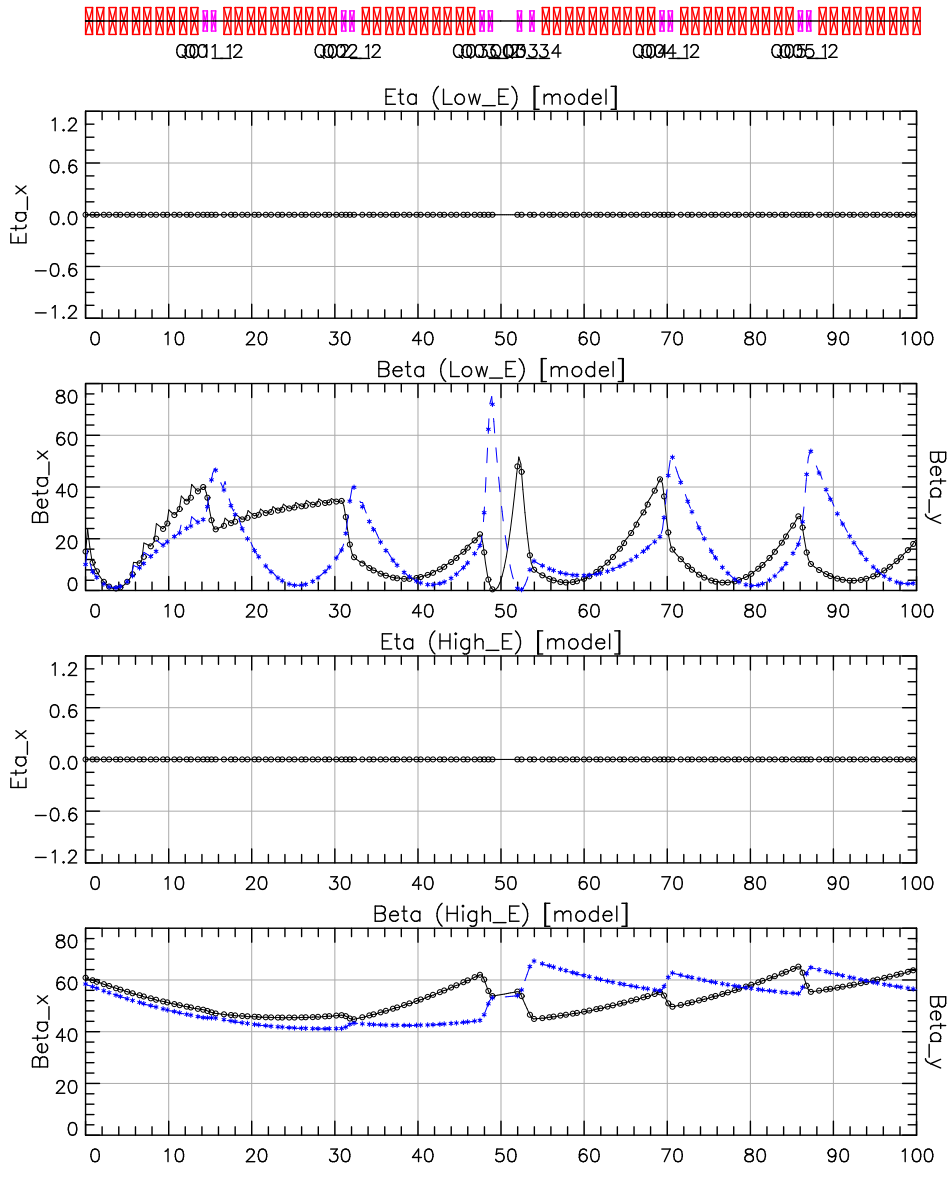



Figure 1.2: Example of a plot page




| | 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 1.3: Another example of a plot page.

## 1.9 Single Character Input

Sometimes it is convenient to be able to vary variables using single key strokes without having to type a carriage return. With *Tao*, this is possible using what is called `single mode`. This is distinct from `line mode` where commands to *Tao* are typed at the command line with a carriage return signaling the end of the command.

The `single mode` initialization file associates variables with certain keyboard keys so that when these keys are pressed the value of the variable is varied. This association between variables and keys is called a `key table`. See Chapter §8 for more details.

## 1.10 Aliases and Command Files

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 §7.1 for more details.

Command files are like Unix shell scripts. A series of commands are put in a file and then that file can be called using the `call` command (§7.2).

Do loops 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 (§7.6) can be used to signal the end of the command file.

The `pause` command (§7.14) can be used to temporarily pause the command file.

## 1.11 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 (§6.5), 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 §6.5 and §6.7 for details on setting up spin tracking.

## 1.12 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 4 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.

# Chapter 2

# Tutorial

This chapter gives a brief tutorial of *Tao* to get you, the user, up and running with *Tao* without needing to dredge through the entire reference manual. See Chapter §1 for a discussion of *Tao* terminology.

## 2.1 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

```
$ACC_EXE/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 (§6.7). 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 §6.2 for more details. Example:

```
$ACC_EXE/tao -init my.init -lat xsif::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. These files can be viewd and/or downloaded at:

```
https://accserv.lepp.cornell.edu/cgi-bin/view.cgi/trunk/src/tao/program
```

On a unix machine with the Subversion `svn` client program, these files can be obtained more simply via the command:

```
svn co https://accserv.lepp.cornell.edu/svn/trunk/src/tao/program
```

this will create a sub-directory `program` of your current directory with the appropriate files.

## 2.2 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 §6.3.

Using the example files provided with *Tao* (§2.1), *Tao* is started with the command:

```
$ACC_EXE/tao
```

Since no initialization file is specified on the command line, the default file `tao.init` is used. In the example `tao.init` there is the following:

```
&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 (§6.3), *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.

## 2.3 Getting information from Tao

### 2.3.1 The Plotting Window

When *Tao* first starts up, *Tao* will create a new window for displaying plots. This window will be called the `plot` window. Commands to *Tao* can be typed in from the original window from which *Tao* was invoked. This will be called the `command` window.

As a first step, let us command *Tao* to display some plotting information using the command:

```
show plot
```

The information will be displayed in the command window. The first section of the `show plot` output shows information on font sizes:

```
plot_page parameters:
%size                      =   4.00000000E+02  5.00000000E+02
%n_curve_pts               =      401
%text_height               = 12.000
%main_title_text_scale     = 1.300
%graph_title_text_scale    = 1.100
%axis_number_text_scale    = 0.900
%axis_label_text_scale     = 1.000
%key_table_text_scale      = 0.900
%legend_text_scale         = 0.800
%shape_height_max          = 40.000
```

See section §6.10 for more details.

The second section of the `show plot` output lists the plot `templates` (§1.8) and §6.10.2) that *Tao* knows about:

```
Templates:
  Plot                .Graph
  ------------------  ----------
```

```
orbit               .x  .y
phase               .a  .b
beta                .a  .b
eta                 .x  .y
cbar                .22  .12  .11
quad_k1             .k1
floor               .this
```

In this case, these templates are defined in the `tao_plot.init` file. There is an `orbit` plot template which has two associated graphs: `orbit.x` and `orbit.y`, etc. To see more information on, for example, the `phase` template plot, use the command

```
show plot phase
```

and to see more information on, for example, the `orbit.y` graph use the command

```
show plot orbit.y
```

Similarly, the `show plot` command can be used to display information about the curves within each graph.

The next block of information in the `show plot` command is:

```
Visible  Plot Region         <-->  Template              x1    x2    y1    y2
-------  -----------              ----------------------------------------
   T     top                  <-->  orbit               0.00  1.00  0.48  0.95
   T     bottom               <-->  phase               0.00  1.00  0.00  0.48
```

This shows that two plot regions have been defined called `top` and `bottom`. Each region contains a visible plot and the plot template associated with the `top` region is the `orbit` template, etc.

Figure 2.1 shows what you will see in the plot window. In the top two plots you see the `x` and `y` model lattice orbit data. The horizontal axis is the *CESR* BPM index. The horizontal pretzel and L03 vertical bump in CESR can be clearly seen. The slight vertical displacement due to the solenoid compensation can also be seen around the IP. The orbit data is for a closed orbit electron (this being a storage ring). The bottom two plots show the relative particle phase, that is, the difference between the model and design phases (as documented in the plot title as [model - design]). Two plot regions are defined in *Tao* `top` and `bottom`.

As a first step let's view the absolute model phase. Use the command:

```
plot bottom model
```

This will change the data plotted in the bottom two graphs to just the model. The plots are now way off scale. Let *Tao* automatically set the scale by typing:

```
scale bottom
```

As expected, the phase increases approximately linearly as the particle travels through the ring. Zero phase is halfway through the ring (at L03 in *CESR* lingo). This is always true. Absolute phase is arbitrary so *Tao* sets the average phase to zero when generating the data. To set this back to relative phase type:

```
plot bottom model - design
```

Let's now look at the beta function by typing

```
place bottom beta
```

Again, we need to rescale the plots by typing

```
scale bottom
```

We see the periodic FODO beta function where large horizontal beta corresponds to small vertical beta and vice versa.

Likewise, we can look at the dispersion in the top two graphs by typing

```
place top eta
scale top
```

The plot window should now look like Figure 2.2.

Now let's look at the coupling (C-matrix) by typing

```
place bottom cbar
scale bottom
```

We see that there is strong coupling within the CLEO solenoid and virtually no coupling anywhere else. To zoom in the scale so that we can see the residual coupling outside the interaction region type

```
scale bottom -0.01 0.01
```

The `**Limited**` displayed in red on the bottom plots tells us that there are data points outside the plotted region. We now see that there is a small amount of coupling at the L03 region (BPM indexes 45-55) and a few other places along the ring. Your plot window should now look like Figure 2.3.

The x-axis is currently the BPM index number. It is sometimes convenient to plot the data versus longitudinal position. This is done by typing

```
x-axis * s
```

The `all` will apply the change to all plot areas (both top and bottom). In any of the above commands `top` or `bottom` could have been replaced with `all`.

Variables can also be plotted provided the proper plot template has been set up in the plot initialization file (See Section §6.10 for details on initializing plotting). Type the following to view the quadrupole k1 values:

```
place bottom quad_k1
```

### 2.3.2 The Show Command

Anything in the super-universe can be displayed using the `show` command (§7.24). To get a list of the data elements currently defined in *Tao* type

```
show data
```

the output should look like:

```
Name                                    Using for Optimization

orbit.x[0:99]
orbit.y[0:99]

phase.a[0:99]
phase.b[0:99]

eta.x[0:99]
eta.y[0:99]

beta.a[0:99]
beta.b[0:99]

cbar.11[0:99]
cbar.12[0:99]
cbar.21[0:99]
cbar.22[0:99]
```

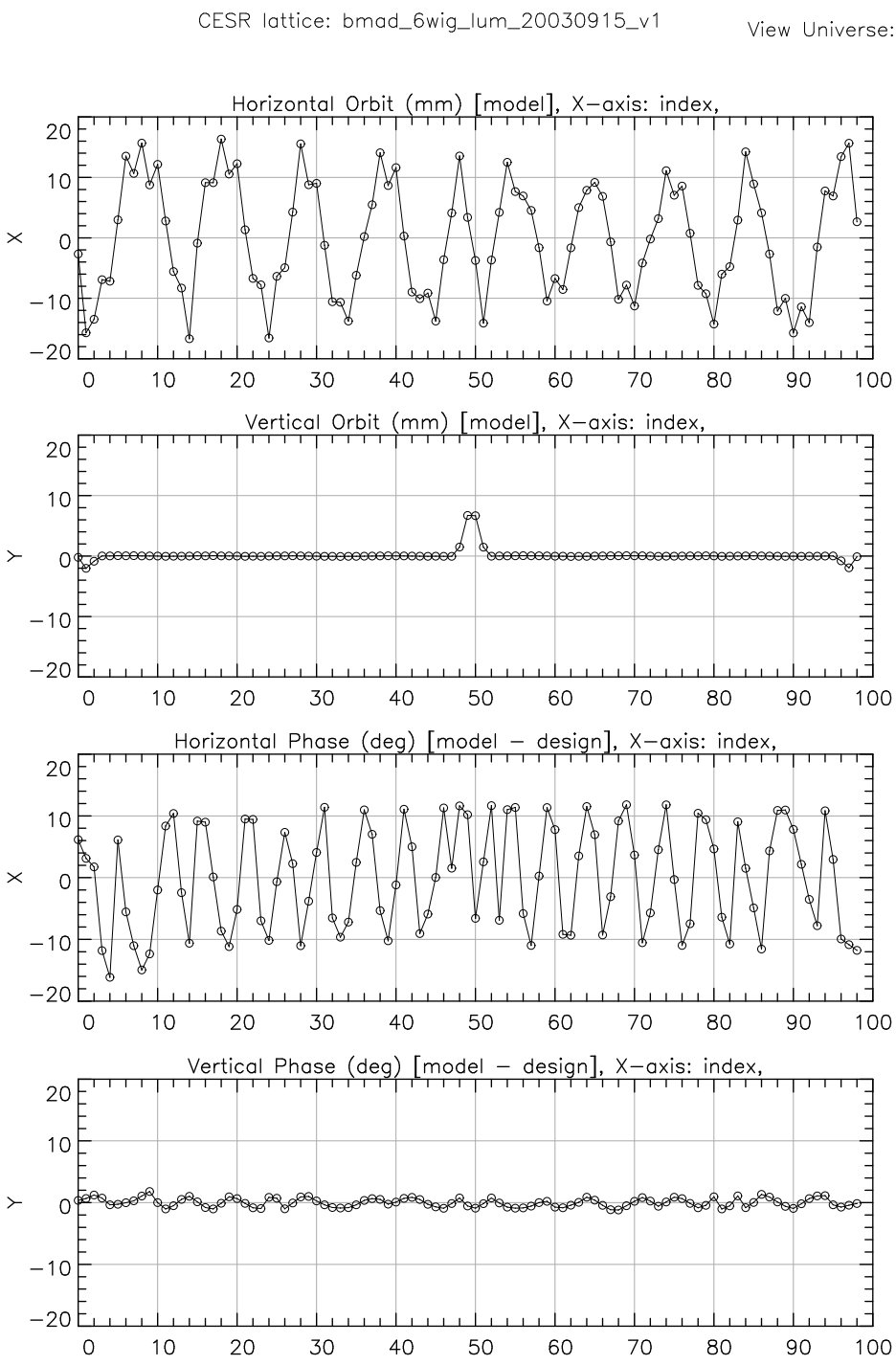



Figure 2.1: The plot window at startup

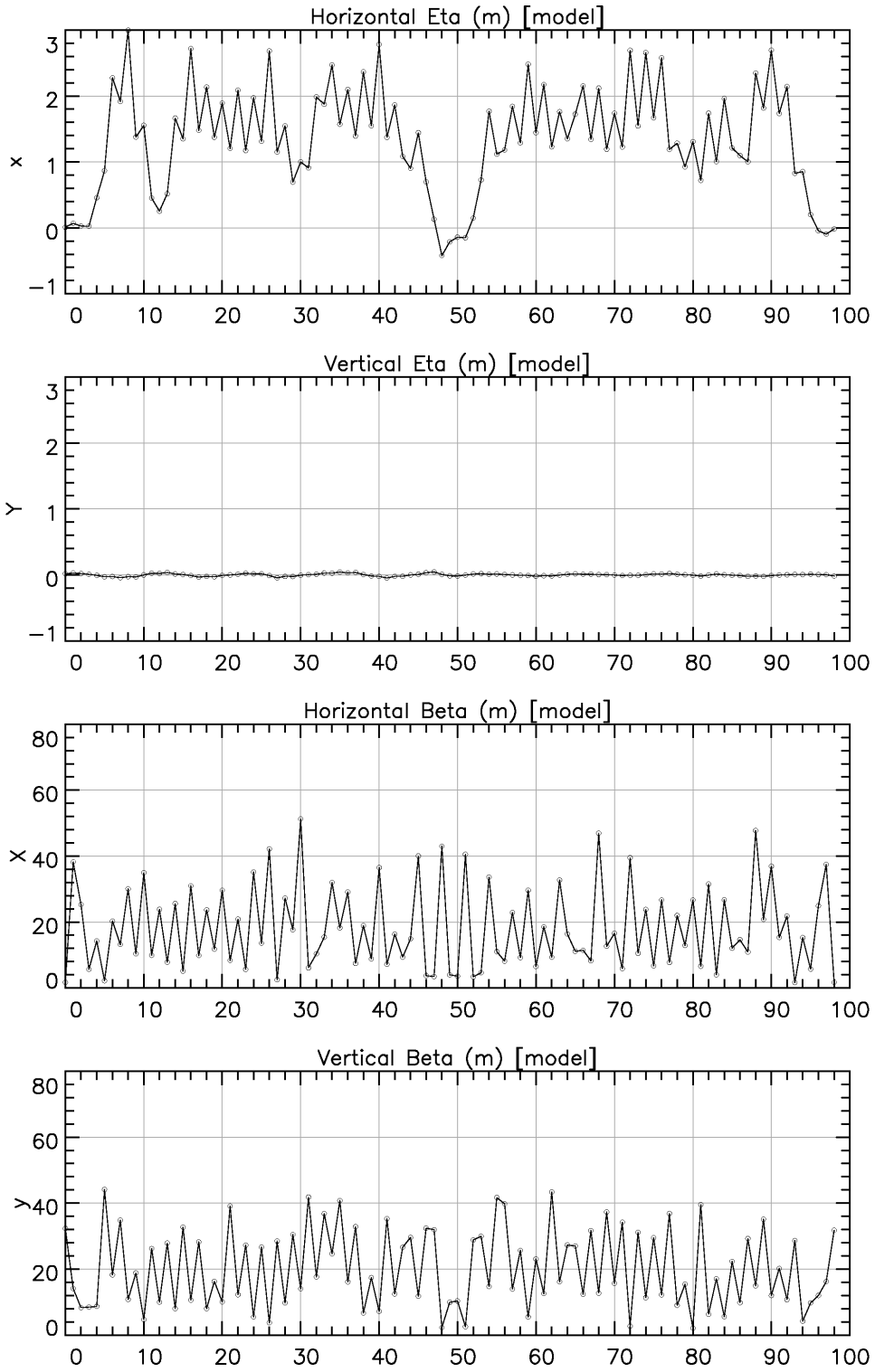



Figure 2.2: Plotting dispersion and beta function

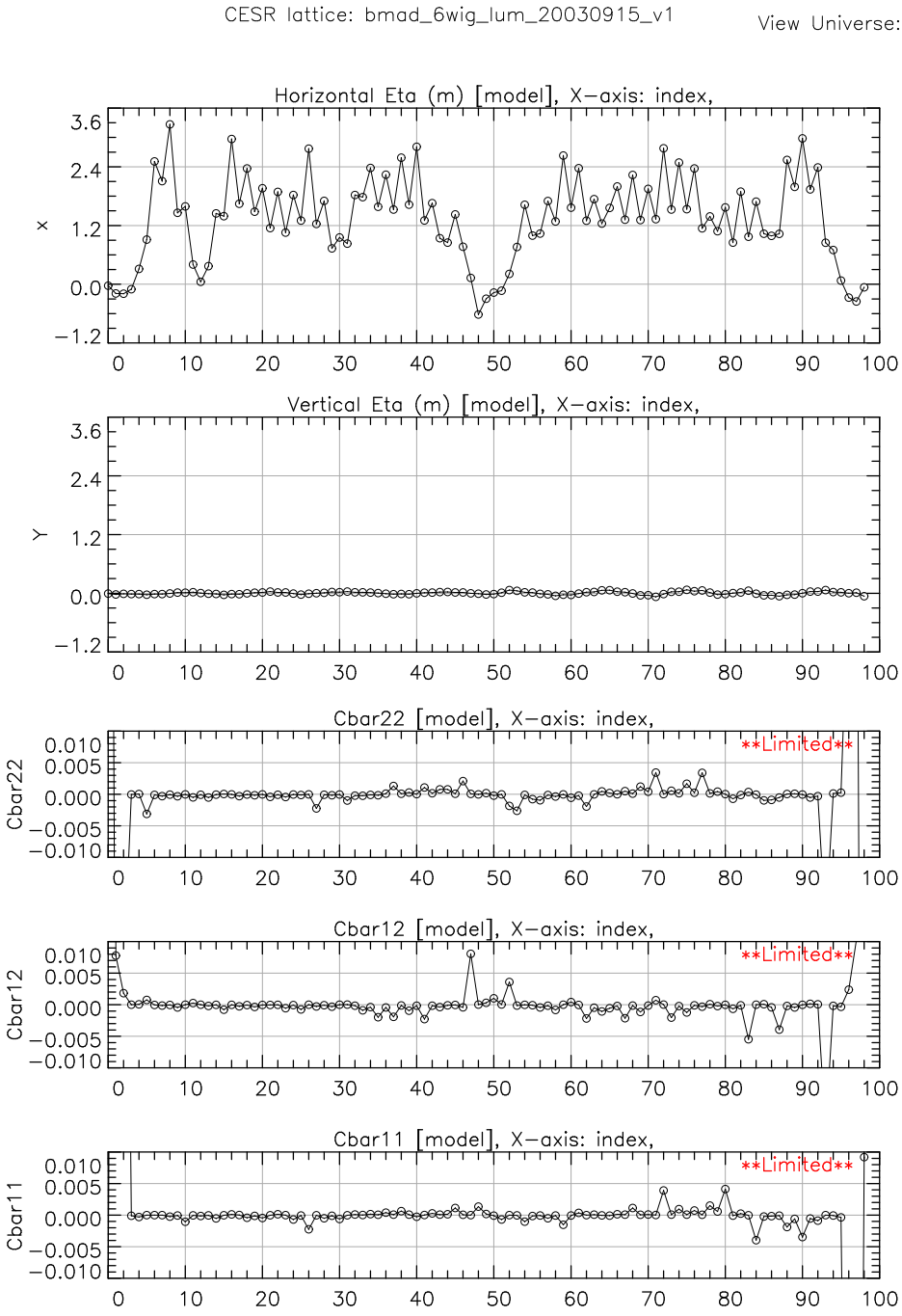



Figure 2.3: Zooming in on the residual coupling outside the IR.

```
k.11b[0:99]
k.12a[0:99]
k.12b[0:99]
k.22a[0:99]
```

There are six `d2` data types (§3.1) defined in the initialization file: `orbit`, `phase`, etc. The `beta` data is further subdivided into two `d1` data arrays labeled `beta.a` and `beta.b`. Each of these arrays has 100 data points indexed from 0 to 99.

The ‘‘`Using for Optimization`’’ column is blank indicating that no data would be used in an optimization (§4).

To see the data values for the horizontal beta function for *CESR* BPMs 1 through 50 type

```
show data beta.a[1:50]
```

Since we haven't changed any elements in the lattice yet the model values equal the design values. Also note that `beta.a` is actually the a-mode betatron function. In regions with little or no coupling, the a-mode is almost completely in the horizontal plane.

A significant point: The convention in *Bmad* is to label the twiss parameters as `x` and `y` but they are actually the `a` and `b` normal modes. So in regions of strong coupling `beta.a` does not correspond to `orbit.x` which is always in the true horizontal lab frame. However, if you wish, you can re-label your twiss data planes as `a` and `b`. Section 6 shows you how to do this. Keep in mind that lattice twiss parameters are defined *only for uncoupled betatron motion* so this is all that is provided as data types for single particle tracking. However, true lab-frame `x` and `y` twiss parameters can be defined for a distribution of particles so for beam tracking true lab-frame `x` and `y` twiss parameters can be calculated and are provided as data types for those tracking types. See the *Bmad* manual for how to convert from normal mode coordinates to lab-frame coordinates.

This tutorial uses single particle tracking and the twiss parameters are found about the orbit of the tracked particle. There is another tracking type called particle beam tracking. This tracking type will not be explored in this tutorial.

You can also view variables by typing

```
show var
```

To view the quadrupole k1 values for *CESR* quadrupoles 5 and 20 through 30 type

```
sho var quad_k1[5,20:30]
```

Again, since we haven't changed any quadrupoles the model values are all at their design values.

You can also see the details of a particular lattice element. To view the details for quadrupole Q05W type

```
sho ele Q05W
```

`show var` and `show ele` show two completely different types of structures in *Tao*. Elements are the actual lattice elements as known to *Bmad*. Variables are native *Tao* structures that act kind of like *Bmad overlays* and only indirectly control the lattice elements.

A list of lattice elements between two elements can be shown by typing

```
sho lattice 0:20
```

This will show a list of all the lattice elements between and including elements 0 and 20. Twiss parameters and orbit information, at the exit end of each element, is also shown.

Anything printed to the display using the `show` command can also be printed to a file by typing

```
show -write <file_name> <what_to_show>
```

A list of all intrinsic `tao` commands can be found by typing

```
help
```

This will not list any custom commands. Detailed help on any individual command can be found with

```
help <command_name>
```

where `<command_name>` is the command you want help with.

## 2.4 Modifying the Lattice

### 2.4.1 Changing a Variable

Let's change a variable and see what happens to the lattice. We are going to change a quadrupole strength so we should plot the change in beta and phase. Type the following (everything after the '!' are just comments and can be omitted):

```
x-axis * index        ! let the data index be the x-axis
place top beta          ! plot Beta data on top plot
place bottom phase      ! plot phase data on bottom plot
plot * model - design ! plot the difference between model and design data
scale                   ! scale all plots
```

The k1 value can be increased by 0.01 units for quadrupole Q05W by typing

```
change var quad_k1[5] 0.01
scale
```

Note the information returned on the command line after the command and the relative changes in beta and phase in the plot window. This is a vertically focusing quadrupole so the vertical beta and phase are affected more than the horizontal. The `0.01` at the end of the command tells *Tao* to change this variable by 0.01 units. If you want to set a variable to a particular value then use a "@" before the value. So, to change this quadrupole k1 to -0.348 type

```
change var quad_k1[5] @-0.348
```

### 2.4.2 Putting things back where you found them

Let's put this quadrupole back where we found it. We can also modify the quadrupole by modifying the lattice element directly by typing

```
change ele Q05W k1 d0.0
```

By modifying the element directly with the `change ele` command you can modify almost any attribute of the element listed in the output of `show ele Q05W`. The "d" before the value is used to set the variable relative to the design value.

If you've changed the lattice around a lot using variables, a great way to set all variables back to their design values is to type

```
set var *|model = *|design
```

This only works if you just changed variables. If you changed any elements directly with the `change ele` command then this will not work. To set every attribute of every element back to the design type

```
set lattice model = design
```

Note that this will also recalculate the data and variable values associated with the the model lattice to reflect the change so all the bookkeeping is done for you.

## 2.5 Using the Optimizer for Lattice Correction and Design

There are three non-linear optimizers included with *Tao*: Two optimizers are based on the Levenburg-Marquardt method. These are referred to as '`lm`', and `lmdif`', the third optimizer is based upon `Differential Evolution` and is called '`de`'. This example will use the Levenburg-Marquardt optimizer which first uses steepest decent to zero in on the region containing the minimum then uses the inverse-Hessian to converge on the minimum. See Numerical Recipes in Fortran (or C or C++) for a

detailed explanation. There's no need to know the details in order to use either optimizer. Once you set up the problem *Tao* has the proper wrapper routines to do the optimization. Of course, you are not limited to using the included optimizers. Custom analysis can be done using custom routines but these two optimizers have been integrated 'out of the box' with the *Tao* data and variable structures to make quick optimization possible.

Basically, the '`lm`' is typically faster since it uses a Jacobian or "dmerit" matrix to find the data derivatives versus each variable before starting the optimization process. However it assumes the second derivative is fairly smooth, so for very complex function spaces the '`de`' may work better. But because '`lm`' typically converges much faster (for functions it can handle) it is recommended to try this one first and only use '`de`' if it fails.

### 2.5.1 Fix a Messed Up lattice

Let's mess the lattice up a little and see if the optimizer can "fix" the lattice. First transfer the "correct" or `design` lattice to the `meas` data area.

```
set data *.*|meas = *.*|design
```

Now mess up the lattice a bit. We'll be messing with quadrupoles so plot beta and phase.

```
place top beta
place bottom phase
plot * meas - model
change var quad_k1[10] 0.001
change var quad_k1[21] -0.001
change var quad_k1[67] -0.005
scale
```

The lattice is now sufficiently screwed up.

Now specify what variables and data to use in the optimization. First type

```
show top10
```

to see what data is effecting the merit function the most. The merit function is defined by

$$\mathcal{M} \equiv \sum_i w_i \left[\text{data}_{\text{model}}(i) - \text{data}_{\text{meas}}(i)\right]^2 + \sum_j w_j \left[\text{var}_{\text{model}}(j) - \text{var}_{\text{meas}}(j)\right]^2 \tag{2.1}$$

where $w_i$ and $w_j$ are the weights given to each component. The optimizer tries to minimize the merit function by changing the model to look like the measured data. From the `top10` output we see that the beta function is effecting the merit function the most. Since we are looking at beta and phase let's only use that data in the optimization.

```
veto data *         ! Veto all the data
use  data beta      ! Use all the beta data
use  data phase     ! And use all the phase data
```

We also know that we need to change quadrupoles to correct the lattice.

```
veto var *           ! veto all the variables
restore var quad_k1 ! restore just the quad_k1 variables
```

Note that we need to specify what data and variables we will be using beforehand in the initialization files. This is already taken care of in the demo initialization files. You can view these files to see how the data and variables were initialized. Raw lattice elements cannot be used by the included optimizer but there is no such restriction on custom optimizers.

Now let's see if we have the optimizer set up correctly.

```
sho optimizer
```

Whoops! we want to use the Levenburg - Marquardt optimizer so

```
set global optimizer = lm
show opti
```

The second command is short-hand. Most *Tao* commands can be shortened to the least number of characters needed to distinguish the command from all others.

Now we're ready to run the optimizer or "fit" the model to the 'measured' data.

```
run
```

You see the optimizer going through its cycles and it did it! The model is now "fitted." We can see what changes where done to the quadrupoles by typing

```
sho var quad_k1
```

The optimizer came very close to finding the "design" lattice. However, it changed more quadrupoles than just 10, 21 and 67. This isn't surprising. The optimizer finds the minimum of the merit function and there are potentially many minima, or degeneracies. It does it's best not to get stuck in a local minimum and as we can see by the plotted data, the minimum found is very close – virtually identical – to the design lattice optics. A good hint as to what variables will be adjusted is the output of `show top10`. The top 3 derivatives were not the quadrupoles we adjusted. Nevertheless, the final result was a darn near perfect match!

### 2.5.2 Now Not Using all of the Variables

Alternatively, we could have used only a subset of the quadrupoles. Say we know approximately which quadrupoles should be adjusted. We can then specify these variables ranges.

```
change var quad_k1[10]   0.001
change var quad_k1[21]  -0.001
change var quad_k1[67]  -0.005
scale
use var quad_k1[8:12,20:25,65:70]
run
show constraints
sho var quad_k1[8:12,20:25,65:70]
```

Different quadrupoles than the ones we initially changed were still adjusted by the optimizer. The end result is again very close to the design lattice.

### 2.5.3 Lattice Design

In `lattice design` (§4.2) it is generally easiest to specify constraint data using the "`single line`" input format (§6.9). For example:

```
&tao_d2_data
  d2_data%name = 'c1'
  universe = '1'
  default_merit_type = "max"
  n_d1_data = 1
/

&tao_d1_data
  ix_d1_data = 1
  d1_data%name = 'xx'
```

```
  default_weight = 0.1
  ix_min_data = 1
  ix_max_data = 50
  data( 1) = 'beta.a'  'end_arc' 'end_lin3' 'max'    60   1.0   T
  data( 2) = 'beta.b'  'end_arc' 'end_lin3' 'max'    60   1.0   T
  data( 3) = 'beta.a'  ''        'end_lin3' 'max'    60   1.0   T
  data( 4) = 'beta.b'  ''        'end_lin3' 'max'    60   1.0   T
  data( 5) = 'alpha.a' ''        'end_lin3' 'max'   -0.6  1e3   T
  data( 6) = 'alpha.b' ''        'end_lin3' 'max'   -0.6  1e3   T
/
```

## 2.6 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 §8 for more details.

## 2.7 Where to go from here

You now have an understanding of the basic abilities of *Tao*. After this tutorial, Part II of the *Tao* Manual should be legible and useful. The Reference Guide will provide the details of everything mentioned in this tutorial. It goes into detail of setting up your own initialization files and how to use the optimizer. It also includes a complete command reference with command syntax.

However, you're not yet ready to customize *Tao*, but this is where the true versatility of *Tao* lies. So, onward to the next section and learn how to write your own custom routines to perform whatever accelerator calculations that strikes your fancy!

# Part II

# Reference Guide

# Chapter 3

# Data in Tao

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 (§4). This chapter explains how data is organized in *Tao* while Section §6.9 explains how to define the structures that hold the data in the initialization files. When running *Tao*, the `show data` (§7.24) command can be used to view information about the data.

## 3.1 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 3.1. 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 hetogeneous 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.

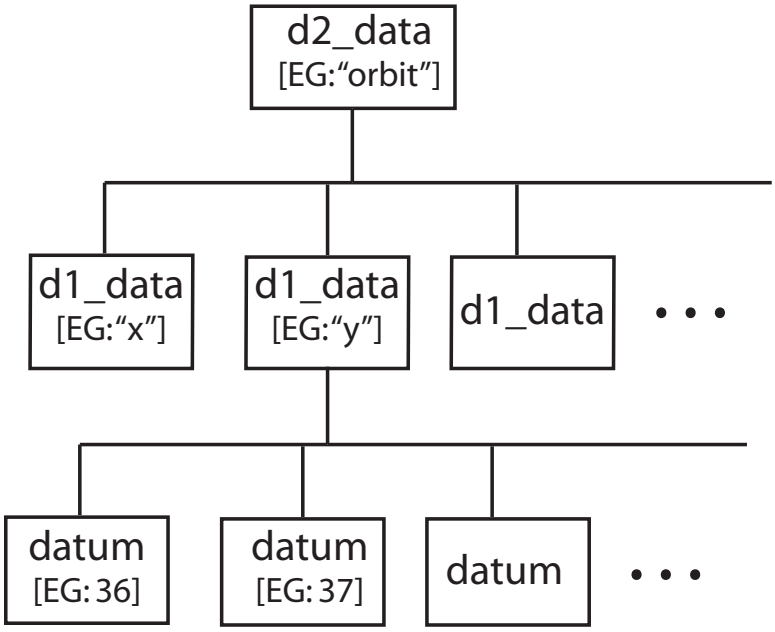



Figure 3.1: A `d2_data` structure holds a set of `d1_data` structures. A `d1_data` structure holds an array of datums.

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 §1.3, 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 viewed universe.
-1@orbit.x      ! Same as "orbit.x".
```

As explained in Section §3.2, 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 currently viewed 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.

## 3.2 Anatomy of a Datum

Each datum has a number of quantities 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. "lattice", or "beam".
ix_ele           ! Integer: Index of "ele" in the lattice element list.
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.
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: The model at some previous time.
base             ! Real: The value as calculated from the base model.
fit              ! Real: The value as calculated from a fitting procedure.
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.
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` (§7.24) command can be used to view the components of a datum. The `set` command (§7.23) can be used to set some of these components.

## 3.3 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 (4).

**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.

**model**
The value of the datum as calculated from the `model` lattice (§1.4).

**design**
The value of the datum as calculated from the `design` lattice (§1.4).

**base**
The datum value as calculated from the `base` lattice (§1.4).

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

## 3.4 Datums in Optimization

When using optimization for lattice correction or lattice design (§4), Individual datums can be excluded from the process using the `veto` (§7.28), `restore` (§7.19), and `use` (§7.27) 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 (§4.3). 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 (§6.9) or is set using the `set` command (§7.23). 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` (§6.5).

Finally good_opt is meant for use in custom versions of *Tao* (§9) and is always left True by the standard *Tao* code.

Example of using a show data (§7.24) 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.

## 3.5 Tao Data Types

Data can be classified by how it is calculated. This is set by the data_source parameter associated with an attribute. If the data comes from particle beam tracking, data_source must be set to "beam". For everything else, data_source must be set to "lattice". Some datum types, like the floor position of an element, only make sense with a "lattice" data_source. Other types, like dpx_dx, only make sense with a "beam" data_source. There is a third class of data types, like emit.a that can take a "lattice" or "beam" data_source. It is important to keep in mind that with this third case, such datums will have a different value depending upon what data_source is used. Table 3.5 lists the predefined data types in *Tao* that are valid for a "beam" data_source (§6.9) and Table 3.5 lists the predefined data types in *Tao* that are valid for a "lattice" data_source.

It is important to note the difference between the d2.d1 name that is used to refer to a given set of data and the actual type of the data of the datums in the set. The d2.d1 name is arbitrary and is specified in the *Tao* initialization file (§6.9). 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.

Associated with a datum are three lattice elements called the evaluation element, the reference element, and the start element. The corresponding components in a datum are shown in Table 3.1. 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.

Some datums, like the emittance in a circular ring, do not have associated lattice elements and their corresponding element names will be blank. All other datums must have an associated evaluation element

| | *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 3.1: 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.

that is specifed by setting the `ele_name` or `ix_ele` datum 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"
```

then 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.

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.

The individual data types are described below:

**beta.**

`beta.a` and `beta.b` are the lattice beta functions. `beta.x` and `beta.y` are beam projected beta functions defined by

$$\beta.x = \frac{< x^2 >}{\sqrt{< x^2 >< x'^2 > - < xx' >^2}}. \tag{3.1}$$

where the average `<>` is over all the particles in the beam.

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

**emit.**

Emittance can be calculated in one of two ways. One way is to calculate it from a tracked beam. The other is from the lattice using the standard radiation integrals (see the Bmad manual). For a linear lattice, the emittance varies along the length of the line while for a circular lattice there is a single emittance number. `emit.a` and `emit.b` are the standard normal mode emittances. With beam tracking, there are also `projected emittances` which give an indication of what the beam looks like if projected onto the $x$ or $y$ axes:

$$\text{emit.x} \equiv \sqrt{< x^2 >< p_x^2 > - < x p_x >^2} \tag{3.2}$$

Notice that this does *n*ot correspond to the standard emittance definition in one dimension:

$$\epsilon_x = \sqrt{< (x - \eta_x\, p_z)^2 >< (p_x - \eta'_x\, p_z)^2 > - < (x - \eta_x\, p_z)\,(p_x - \eta'_x\, p_z) >^2} \tag{3.3}$$

`emit.x` and `emit.y` are the projected emittances. Also notice 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.

**expression:**

The `expression:` The data type can be an expression (§1.7). In this case the `data_type` must begin with the string "`expression:`". For example:

```
data_type = "expression: 1@dat::beta.a[12] - 2@dat::beta.a[12]"
```

With this, the value of the datum will be the difference between datums `beta.a[12]` for universe 1 and universe 2. In this example, the source of both terms in the expression is explicitly given as `dat`. However, if an explicit source is not specified for a given term, the `datum%data_source` component in the initialization file (§6.9 will be used. An expression can also be used as the `default_data_type` in the `tao_d1_data` structure in the initialization file. In this case, the evaluation point is implicit. For example:

```
default_data_type = "expression: 1@dat::beta.a - 2@dat::beta.a"
```

And the evaluation point for the datums created will be, just like other data types, by the specified `datum%ele_name`.

To be valid, if an expression has a term with a `dat` source, the expression must be evaluated after the `dat` 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 example above, the expression terms are `1@dat::beta.a[12]|model` and `2@dat::beta.a[12]|model`. 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.

**floor.** This is the global floor position at the exit end of evaluation element. See the *Bmad* manual for details on the global coordinate system. See also `rel_floor.`.

**n_particle_loss**

If the start 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.

**periodic.tt.**

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

$$T_p \equiv (T_0 - I_4)^{-1} \tag{3.4}$$

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

$$I_4 \equiv \begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & 0 & \\ & & & & & 0 \end{pmatrix} \tag{3.5}$$

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 circular lattice.

**phase.** This is the 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. (4.7)) 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 evalustion.

**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.** 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.`.

**`t. tt.`**

The `t` and `tt` data types give terms of the Taylor map between two points. The difference between `t.` and `tt.` is that `t.` is restricted to exactly three indices and `tt.` is not. `t.` is superfluous but is keep for backwards compatibility.

Calculation of `t.` and `tt.` 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 (§6.10.2) is plotted for `t` and `tt` 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.

**unstable_orbit**

The `unstable_orbit` datum is used for linear lattices. 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

$$1 + i_{\text{ele}} - i_{\text{lost}} \tag{3.6}$$

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. When tracking beams, the value of `unstable_orbit` is the averaged value over all particles in the bunch. The default for the evaluation element, if `ele_name` nor `ix_ele` is not specified, is to use the last element in the lattice.

`unstable_orbit` is used in an optimization to avoid unstable solutions (§4.3).

**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 (§4.3).

**wire**

`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.

| *Data_ Type* | *Description* |
|---|---|
| beta.x, beta.y beta.z | Projected beta function |
| beta.a, beta.b | Normal-mode beta function |
| dpx_dx, dpx_dy, etc. | Bunch <x px> / <$x^2$> & Etc... |
| emit.x, emit.y, emit.z<br>emit.a, emit.b, emit.c | Emittance (global) |
| eta.x, eta.y | Projected dispersion |
| eta.a, eta.b | Normal-Mode dispersion |
| etap.x, etap.y | Projected dispersion derivative |
| etap.a, etap.b | Normal-Mode dispersion derivative |
| n_particle_loss | Number of particles lost |
| norm_emit.x, norm_emit.x, norm_emit.z<br>norm_emit.a, norm_emit.b, norm_emit.c | Normalized beam emittance |
| ref_time | Reference time |
| sigma.x, sigma.y, sigma.z<br>sigma.px, sigma.py, sigma.pz | Bunch size |
| spin.polarization, spin.theta, spin.phi | Particle spin |
| wire.<angle> | Wire scanner at wire angle <angle> |

Table 3.2: Predefined Data Types for "beam" data_source

| *Data_ Type* | *Description* |
|---|---|
| alpha.a, alpha.b | Normal-Mode alpha function |
| beta.a, beta.b | Normal-mode beta function |
| bpm_orbit.x, bpm_orbit.y | Measured orbit |
| bpm_phase.a, bpm_phase.b | Measured betatron phase |
| bpm_eta.x, bpm_eta.y | Measured dispersion |
| bpm_k.22a, bpm_k.12a,<br>bpm_k.11b, bpm_k.12b | Measured coupling |
| bpm_cbar.22a, bpm_cbar.12a,<br>bpm_cbar.11b, bpm_cbar.12b | Measured coupling |
| cbar.11, cbar.12, cbar.21, cbar.22 | Coupling |
| chrom.a, chrom.b | Chromaticity for a ring (global) |
| e_tot | Beam total energy |
| element_param.<param_name> | Lattice element parameter |
| emit.a, emit.b, emit.c | Emittance (global) |
| eta.x, eta.y | Lab Frame dispersion |
| eta.a, eta.b | Normal-mode dispersion |
| etap.x, etap.y | Lab Frame dispersion derivative |
| etap.a, etap.b | Normal-mode dispersion derivative |
| expression: <arithmetic expression> | See the text |
| face1.offset, face2.offset | Element offset with multipass |
| floor.x, floor.y, floor.z, floor.theta | Global (“floor”) position |
| gamma.a, gamma.b | Normal-mode gamma function |
| rad_int.i5a, rad_int.i5b | I5 radiation integral |
| rad_int.i5a_e6, rad_int.i5b_e6 | Energy normalized I5 radiation integral |
| k.11b, k.12a, k.12b, k.22a | Coupling |
| momentum_compaction | Momentum compaction factor |
| multi_turn_orbit.x, multi_turn_orbit.px<br>multi_turn_orbit.y, multi_turn_orbit.py<br>multi_turn_orbit.z, multi_turn_orbit.pz | Bunch size |
| norm_emit.a, norm_emit.b, norm_emit.c | Normalized beam emittance |
| orbit.x, orbit.y | Transverse orbit |
| orbit.px, orbit.py | Transverse momenta |
| orbit.z, orbit.pz | Longitudinal orbit and momenta |
| orbit.amp_a, orbit.amp_b | “Invariant” amplitude |
| orbit.norm_amp_a, orbit.norm_amp_b | Energy normalized “invariant” amplitude |
| periodic.tt.$ijklm\ldots$ $1 \le i,j,k,\ldots \le 6$ | Taylor term of the periodic map. |
| phase.a, phase.b | Betatron phase |
| phase_frac.a, phase_frac.b | Fractional betatron phase<br>$-\pi < \phi_{\texttt{frac}} < \pi$ |
| phase_frac_diff | Fractional betatron phase<br>difference (a-b) $-\pi < d\phi_{\texttt{frac}} < \pi$ |
| r.$ij$ $1 \le i,j \le 6$ | Term in linear transfer map |
| ref_time | Reference time |
| rel_floor.x, rel_floor.y,<br>rel_floor.z, rel_floor.theta | Relative global (“floor”) position |
| s_position | longitudinal length constraint |
| spin.polarization, spin.theta, spin.phi | Particle spin |
| t.$ijk$ $1 \le i,j,k \le 6$ | Term in $2^{nd}$ order transfer map |
| tt.$ijklm\ldots$ $1 \le i,j,k,\ldots \le 6$ | Term in n$^{th}$ order transfer map |
| tune.a, tune.b | Tune |
| unstable_orbit | Nonzero if particles are lost in tracking |
| unstable_ring | Nonzero if a ring is unstable (global) |

Table 3.3: Predefined Data Types for "lattice" data_source

# Chapter 4

# Lattice Correction and Design

This chapter covers the process of `optimization` which involves minimization of a `Merit Function` to correct and 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. Section §2.5 gives a tutorial on how to setup the merit function and run the `optimizer` in *Tao*. In this chapter, Section §4.1 presents the merit function in the context of lattice corrections while Section §4.2 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 §4.3.

## 4.1 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 as discussed in Chapter §10. *Tao* does, however, 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 (§1.3) with the aim to make the measured `data` (position data for an orbit correction) correspond to the values as calculated from the `model` lattice.

The merit function `M` that is a measure of how well `data_model`, the data as calculated from the `model`, fits `data_meas`, the measured data, is

$$\mathcal{M} \equiv \sum_i w_i \left[\text{data\_model}(i) - \text{data\_meas}(i)\right]^2 + \sum_j w_j \left[\text{var\_model}(j) - \text{var\_meas}(j)\right]^2 \qquad (4.1)$$

`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 merit function is

$$\mathcal{M} = \sum_i w_i \Big[\big(\text{data\_model}(i) - \text{data\_design}(i)\big) - \big(\text{data\_meas}(i) - \text{data\_ref}(i)\big)\Big]^2 + \sum_j w_j \Big[\big(\text{var\_model}(j) - \text{var\_design}(j)\big) - \big(\text{var\_meas}(i) - \text{var\_ref}(i)\big)\Big]^2 \tag{4.2}$$

where `data_ref` and `var_ref` refer to the reference measurement. This merit function is 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 merit function of Eq. (4.1). The `base` lattice is then set equal to the `model` lattice. The second step is to fit the model using the merit function

$$\mathcal{M} = \sum_i w_i \Big[\big(\text{data\_model}(i) - \text{data\_base}(i)\big) - \big(\text{data\_meas}(i) - \text{data\_ref}(i)\big)\Big]^2 + \sum_j w_j \Big[\big(\text{var\_model}(j) - \text{var\_base}(j)\big) - \big(\text{var\_meas}(i) - \text{var\_ref}(i)\big)\Big]^2 \tag{4.3}$$

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.

## 4.2 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 define a merit function to be minimized:

$$\mathcal{M} = \sum_i w_i C_i^2 \tag{4.4}$$

Typically constraints are either to limit values to some range so the constraint would be of the form

$$C = \begin{cases} \text{model} - \text{limit} & \text{model} > \text{Limit} \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

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

$$C = \text{model} - \text{target} \tag{4.6}$$

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`.

## 4.3 Generalized Design

Since lattice correction and design are similar, *Tao* combines the two into one generalized process. In this generalized process, the merit function becomes

$$\mathcal{M} = \sum_i w_i \, D_i^2 + \sum_j w_j \, V_j^2 \tag{4.7}$$

where $D_i$ are the data terms and $V_j$ are the variable terms. The general form of the data merit terms $D_i$ is

$$D = \begin{cases} \text{delta} & \text{Non-Zero-Condition} \\ 0 & \text{Otherwise} \end{cases} \tag{4.8}$$

The equation used to calculate `delta` is determined by two global logicals called `opt_with_ref` and `opt_with_base` (§6.5) as shown in Table 4.1. An exception occurs when using a `common base lattice`

| 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 4.1: The form of `delta`

(§4.5). 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`.

Another exception occurs with data when the datum value cannot be computed (§3.4). 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 (§3.2). There are five `merit_type` constraint types as given in Table 4.2.

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

```
target
limit
```

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

Table 4.2: Constraint Type List.

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

$$V = \begin{cases} \text{model} - \text{high\_lim} & \text{model} > \text{high\_lim} \\ \text{model} - \text{low\_lim} & \text{model} < \text{low\_lim} \\ 0 & \text{Otherwise} \end{cases} \tag{4.9}$$

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 (§3.5) must be set.

To see a list of constraints when running *Tao* use the `show constraints` command (§7.24). To see how a particular variable or datum is doing use the `show data` or `show variable` commands. See §3.4 for details on how datums are chosen to be included in an optimization.

## 4.4 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. 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 (§6.5).

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` (§6.5). 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.

There are currently three optimizers that can be used: `lm` is an optimizer based upon the Levenburg-Marquardt algorithm as implemented in `Numerical Recipes`[2]. 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 (§6.8). 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 (§6.5). 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.

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.

The third optimizer is called `de` which stands for `differential evolution`[3]. 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 (§7.23).

## 4.5 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[5]. 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 (§6.4). 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 (§6.8), 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 (§7.3) is thus restricted to always vary parameters in the common `model` lattice.

With `CBL`, the `opt_with_base` and `opt_with_ref` (§4.3) 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 viewed universe.

# Chapter 5

# Wave Analysis

## 5.1 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 5.1.

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 5.1: 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[6]. 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 5.1. Figure 5.1a 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

$$\delta phi(s) = D \sin(2\,\phi(s) + phi_0) + C \tag{5.1}$$

where $\phi$ is the phase advance and the quantities $C$, $D$ and $\phi_0$ are varied to give the best fit. Once $C$, $D$, and $\phi_0$ are fixed, Eq. (5.1) can be evaluated at any point. Figure 5.1b shows the orbit of 5.1a with the

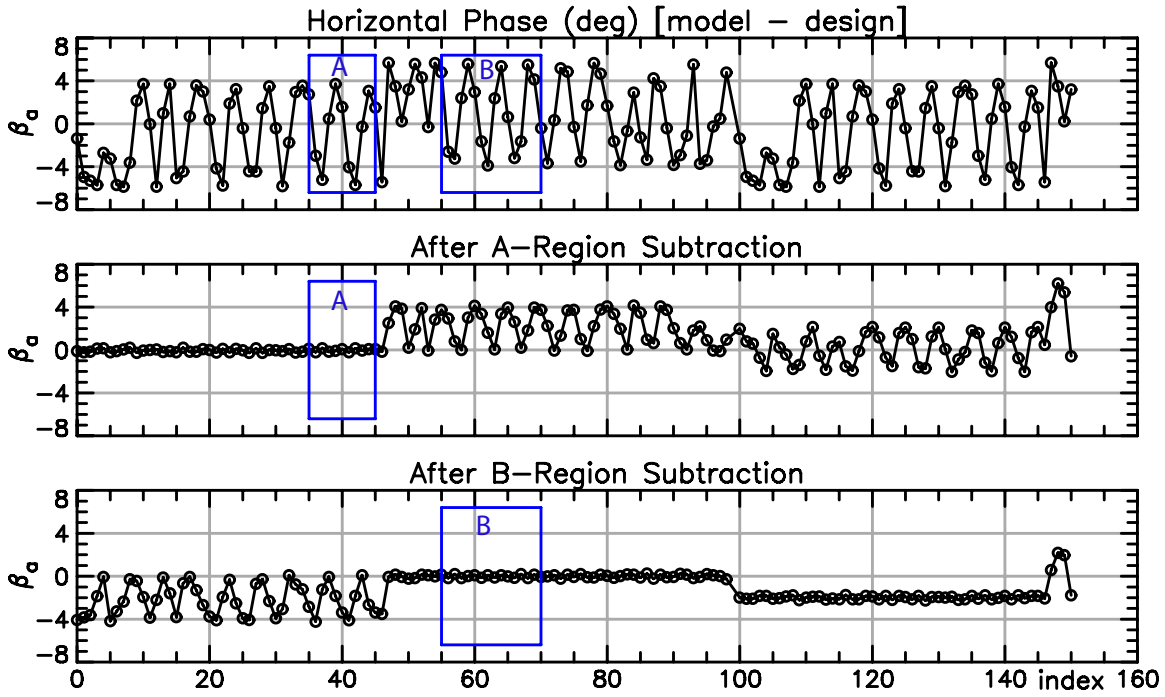



Figure 5.1: Example wave analysis for betatron phase data.

fit to the $A$ region subtracted off. Similarly, Figure 5.1c shows the orbit of Figure 5.1a with the fit to the $B$ region subtracted off. Concentrating on Figure 5.1b, 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 5.1b, 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 5.1c 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 $\phi_0$ 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 data5.1. 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 5.1. 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.

## 5.2 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.

### 5.2.1 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 (§3). That is, the plotted curve to be analyzed must have its `data_type` parameter set to `'dat'` (§6.9). 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.12
```

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 (§6.9):

```
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
```

### 5.2.2 Wave Analysis Commands and Output

The `wave` command (§7.30) sets which plotted data curve is used for the wave analysis. The `set wave` command (§7.23) is used for setting the $A$ and $B$ region locations. Finally the `show wave` command (§7.24) 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 $\sigma_a/A_a$ and $\sigma_b/A_b$ where $\sigma_a$ and $sigma_b$ are given by Eq. (42) of Sagan[6] and

$$A_a \equiv \sqrt{\xi_a^2 \eta_a^2} \tag{5.2}$$

with a similar equation for $A_b$. $\sigma_a/A_a$ and $\sigma_b/A_b$ 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 $\sigma_{\delta k}/\delta k$, $\sigma_\phi$, and $\xi_C$, and are given by Eq. (39), (43), and (44) respectively of [6]. 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 6

# 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.
```

## 6.1 Format

*Tao* first looks for input files in the current directory and then looks in a directory pointed to by the environmental variable `TAO_INIT_DIR`.

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. Quotes are not needed if the
                                     ! strings do not have any special characters.
    string_arr(1:3) = 1st,2nd,3rd    ! Same as above!
    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.

Special characters where quotes are needed for string input are:

```
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.

## 6.2 Initialization from the Command Line

The syntax of the command line is:

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

The `-init` optional argument can be used to replace the default initialization file name (`tao.init`) with `<init_file_name>`. An initialization file is actually not needed. In this case, a `-lat` switch is mandatory and *Tao* will use a set of default plot templates for plotting.

The `-beam_all` optional argument reads in a beam data file for the entire lattice and this data is used in place of beam tracking. This overrides the `beam_all_file` variable (§6.7).

The `-beam0` optional argument reads in a beam data file which is used to initialize the beam at the beginning of the lattice. This overrides the `beam0_file` variable (§6.7). The centroid of the particle distribution can be shifted using the `change beam_start` command (§7.3).

The `-noplot` optional argument suppresses the opening of the plot window.

The `-lat` switch is used to override the `design_lattice` lattice file specified in the initialization file (§6.4). Example:

```
$ACC_EXE/tao -init my.init -lat xsif::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:

```
$ACC_EXE/tao -lat xsif::slac.xsif|cesr.bmad
```

## 6.3 Beginning Initialization

The initialization starts with an initialization file. The default name for this file is `tao.init` (See §6.2). 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
  lattice_file      = "<file_name>"  ! Default = Initialization file.
  data_file         = "<file_name>"  ! Default = Initialization file.
  var_file          = "<file_name>"  ! Default = Initialization file.
  plot_file         = "<file_name1> {<file_name2>} ..."
                                     ! Default = Initialization file.
  single_mode_file  = "<file_name>"  ! Default = Initialization file.
  startup_file      = "<file_name>"  ! Default = "tao.startup"
  init_name         = "<init_name>"  ! Default = "Tao"
/
```

`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 6.1. Note: If `plot_file` specifies multiple files, the `tao_plot_page`, `element_shapes_lat_layout` and `element_shapes_floor_plan` 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* |
|---|---|---|---|
| tao_design_lattice | lattice_file | lattice files | §6.4 |
| tao_params | "tao.init" | Global Variables | §6.5 |
| tao_connected_uni_init | "tao.init" | Connected Universes | §6.6 |
| tao_beam_init | "tao.init" | Particle beam | §6.7 |
| tao_var | var_file | Variables | §6.8 |
| tao_d2_data | data_file | Data | §6.9 |
| tao_d1_data | data_file | Data | §6.9 |
| tao_plot_page | plot_file* | Plotting | §6.10 |
| element_shapes_lat_layout | plot_file* | Plotting | §6.10.7 |
| element_shapes_floor_plan | plot_file* | Plotting | §6.10.7 |
| tao_template_plot | plot_file | Plotting | §6.10 |
| tao_template_graph | plot_file | Plotting | §6.10 |

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

Table 6.1: Table of `tao` Initialization Namelists.

## 6.4 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.
  taylor_order      = <num>
```

```
    aperture_limit_on  = <logical>
    unique_name_suffix = "<string>"
    combine_consecutive_elements_of_like_name = <logical>
    common_lattice = <logical>        ! Default = False
    design_lattice(i) = "{<parser>::}<lattice_file>{@<use_line>}", {"<lattice2_file>"}
  /
```

`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`. Example:

```
  &tao_design_lattice
    n_universe = 4
    design_lattice(1) = "this.lat"      ! 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.
  /
```

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. 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.
```

`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. A secondary lattice file can be used to modify the primary lattice file. This 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.

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".

`taylor_order` is the order of the Taylor maps. This will override the Taylor order set in the lattice files.

`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 (§1.6). The class is used to restrict which elements can have their names changed. The `name` part is used as a suffix. This suffix must have a single `"?"` character. When this suffix is applied to an element's name, a unique integer is inserted in place of the `"?"`. For example, if `unique_name_suffix` is `"quad:_?"`, and if the following quadrupoles are in the lattice:

```
        QA    QB    QX    QA    QB    QB
```

then after initialization, the names will be:

```
        QA_1  QB_1  QX    QA_2  QB_2  QB_3
```

Setting `aperture_limit_on` to `False` will turn off the aperture limits set in all lattices. This overrides the setting of `parameter[aperture_limit_on]` in a lattice file.

The `common_lattice` switch can be used when there is a baseline lattice that is common to all universes. See §4.5 for more details.

## 6.5 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
/
```

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   ! Reinit derivative matrix cutoff
  real(rp) de_lm_step_ratio = 1       ! Step sizes between DE and LM optimizers.
  real(rp) floor_plan_rotation = 0    ! Rotation of floor plan plot: 1.0 -> 360^deg
  integer u_view = 1                  ! Which universe we are viewing.
  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
  real(rp) random_sigma_cutoff = 4.0             ! Cut-off in sigmas.
  character(16) track_type = "single"            ! or "beam"
  character(16) prompt_string = "Tao"
  character(16) optimizer     = "de"             ! optimizer to use.
  character(16) default_key_merit_type
  character(40) print_command = "lpr"
  character(80) var_out_file  = "var#.out"
  logical var_limits_on = T             ! Respect the variable limits?
  logical opt_with_ref = F              ! use reference data in optimization?
  logical opt_with_base = F             ! use base data in optimization?
  logical init_opt_wrapper = T
  logical label_lattice_elements = T    ! For lat_layout plots
  logical label_keys = T                ! For lat_layout plots
  logical derivative_recalc = T         ! Recalc before each optimizer run?
  logical init_plot_needed = T          ! reinitialize plotting?
```

```
  logical plot_on = T                    ! Do plotting?
  logical lattice_calc_on = T            ! Master switch.
  logical command_file_print_on = T      ! Toggle printing when using a command file.
  logical beam_timer_on = F              ! For timing the beam tracking calculation.
  logical optimizer_var_limit_warn = T   ! Warn when variables reach a limit when
                                         !   optimizing.
end type
```

All global parameters can be changed from their initial value using the `set` command (§7.23).

Random numbers are used by *Tao* in various algorithms. For example, random numbers are used in generating the initial coordinates of the particles in a beam and for misaligning elements in a lattice. As explained below, there are four parameters that govern how random numbers are generated. These can be set in the `tao_global_struct`. For particle beam generation, these parameters may be overridden by setting the corresponding parameters in the `beam_init` struct (§6.7). That is, separate parameters may be setup for beam particle generation verses everything else.

`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%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` selects the alogithm 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%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 (§7.23)), 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.

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

The `bmad_com_struct` holds bmad global variables.

```
type bmad_com_struct
  real(rp) d_orb(6) = 1e-5  ! for the make_mat6_tracking routine
  real(rp) max_aperture_limit = 1e3
  real(rp) rel_tolerance = 1e-5
  real(rp) abs_tolerance = 1e-8
  integer taylor_order = 3               ! 3rd order is default
  logical use_liar_lcavity = F           ! Liar like tracking?
  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 radiation_damping_on = F       ! Damping toggle.
  logical radiation_fluctuations_on = F  ! Fluctuations toggle.
  logical compute_ref_energy = T         ! Enable recomputation?
  logical trans_space_charge_on = F      ! Space charge switch
```

```
    logical coherent_synch_rad_on = F      ! Longitudinal csr
    logical spin_tracking_on = T           ! Do particle spin tracking
  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
```

the `global%track_type` must be set to `"beam"` for the CSR computation. 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 §4.1 for more details on `global%n_opti_cycles` and `global%n_opti_loops`.

## 6.6 Initializing Connected Universes

Universes can be connected together. This can be useful, for example, to attach a damping ring a pre-accelerator to a linac. Only one connection between two given universes is allowed. A universe can only inject into a universe with a greater universe index, so for example, universe 3 can inject into 4 or 5 but not 1 or 2. Each connection between two universes needs a separate `tao_connected_uni_init` namelist. The syntax of this namelist is:

```
  &tao_connected_uni_init
    to_universe           = <Integer>     ! Injecting into this Universe.
    connect%from_universe = <Integer>     ! connect from this universe
    connect%at_element    = <ele_name>    ! connect at end of element
    connect%at_ele_index  = <ele_index>   ! connect at end of ele with this index
    connect%at_s          = <number>      ! connect at position s
    connect%match_to_design = <logical>   ! match optics to design parameters
  /
```

`to_universe` refers to the universe which is to be injected into. Any of `at_element`, `at_ele_index` or `at_s` must be specified but not more than one. These refer to the location in the "from" lattice where the beam/particle is extracted. The injection is always at the beginning of the "to" lattice. Note: Setting a lattice to inject back into itself will not work to make a circular lattice. This must be set in the lattice

file. If there are more than one element named `<ele_name>` then the last element named as such will be used. If `ele_name = "end"` then the end of the injection lattice will be used as the connection point.

Setting `match_to_design` to True will set up a `match` element (see the *Bmad* manual for details on match elements) between the two universes that will match the `design` Twiss parameters from the injecting lattice to the `design` Twiss parameters as specified by the lattice injected into. This is, the input Twiss parameters of the match element will be the design Twiss parameters at the extraction point for the first lattice and the output Twiss parameters of the match element are the design Twiss parameters at the beginning of the second lattice. These input parameters to the match element are not affected by changes in the `model` lattice so optics errors can be simulated.

The match element is not inserted into either lattice. Instead, it resides in the *Tao* connection structure and is tracked through separately. Note that the match element is not an extraction kicker element. If an extraction kicker element is needed then it should be added to either the extraction point of the first lattice or the beginning of the second lattice.

Setting `match_to_design` to False (the default) will cause *Tao* to continually maintain the `model` Twiss parameters and reference energy at the start of the injected lattice equal to the `model` Twiss parameters and reference energy at the extraction point in the injecting lattice independent of any Twiss parameter or reference energy settings in the lattice file for the injected lattice.

Initialization due to a connection between lattices takes precedence over other types of initialization. For example, any particle beam initialization settings (§6.7) will be ignored for a lattice that is being injected into from another lattice.

Even if the connection point is not the end of the injection lattice, the standard lattice calculations will still be performed through to the end of that lattice.

Example:

```
&tao_connected_uni_init
  to_universe = 1
  coupled%from_universe = -1      ! no injection into this universe
  coupled%at_element    = "none"
/
&tao_connected_uni_init
  to_universe = 2
  coupled%from_universe = 1      ! inject beam/particle form universe 1
  coupled%at_element    = "end"  ! inject from the end of universe 1
  coupled%match_to_design = T    ! match the design lattice optics
/
```

## 6.7 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>
  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.
  ix_track_start          = <integer>        ! Lattice element start tracking index.
  ix_track_end            = <integer>        ! Lattice element end tracking index.
  beam_init%a_norm_emitt  = <number>         ! A-mode emittance
```

```
  beam_init%b_norm_emitt  = <number>         ! B-mode emittance
  beam_init%dPz_dZ        = <number>         ! Energy-Z correlation
  beam_init%center        = <number>*6       ! Bunch center offset relative to
                                             ! reference particle (BMAD coords)
  beam_init%sig_e         = <number>         ! e_sigma in dE/E0
  beam_init%sig_z         = <number>         ! Z sigma in m
  beam_init%n_bunch       = <integer>        ! Number of bunches
  beam_init%ds_bunch      = <number>         ! distance between bunches (meters)
  beam_init%n_particle    = <number>         ! Number of particles per bunch
  beam_init%bunch_charge  = <number>         ! charge per bunch (Coulombs)
  beam_init%renorm_center = <logical>        ! Default is T
  beam_init%renorm_sigma  = <logical>        ! Default is F
  beam_init%center_jitter = <number>*6       ! Bunch center rms jitter (meters)
  beam_init%emitt_jitter  = <bumber>*2       ! Emittance rms jitter (dε/ε)
  beam_init%sig_z_jitter  = <number>         ! bunch length rms jitter (dz/z)
  beam_init%sig_e_jitter  = <number>         ! bunch energy spread rms jitter (dE/E)
  beam_init%spin%polarization = <number>     ! spin polarization (1.0 = 100%)
  beam_init%spin%theta    = <number>         ! spin orientation  (polar coordinate)
  beam_init%spin%phi      = <number>         ! 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.
/
```

`ix_universe` refers to the universe index. See *Bmad* documentation on 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.

`%a_norm_emitt` and `%b_norm_emitt` are the normalized emittances used to construct the beam's particle distribution. If not set then 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 `output 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 (§6.2). The file can either be in binary format (binary files can be created by the `output 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.
  <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> <particle_charge> <ix_lost> <spin1> ... <spin4>
  [end particle loop]
  END_BUNCH       ! Marker to mark the end of the bunch specification block
[end bunch loop]
```

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 Reading. 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 three lines give the bunch_charge, z_center, and t_center values followed by a set of lines, one for each particle. Only 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. <ix_lost> is the index at which the particle has been lost at. A value of -1 for <ix_lost> indicates that the particle is still alive.

The particle spin is specified by 4 numbers <spin1> through <spin4> using complex spinor notation.

The number of particles specified 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.

The beam_all_file component specifies a beam data file (which can be created with the output 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 (§6.2). 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.

ix_track_start and ix_track_end are used when it is desired to only track the beam through part of the lattice. ix_track_start gives the starting element 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 index ix_track_end. The default, if ix_track_start and ix_track_end are not present, is to track through the entire lattice. In this case, ix_track_start and ix_track_end will be given values of -1 to indicate that they have not been set.

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 (§7.23) 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 (§7.20).

The default is single particle tracking. To turn on particle tracking the global%track_type perameter 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 §1.6. 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` (§6.5). 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 §6.5.

## 6.8 Initializing Variables

`Variables` 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 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       = <number>            ! Merit_function weight.
                                             ! default = 0.0
  default_step         = <number>            ! Small step value.
                                             ! default = 0.0
  default_merit_type   = "<merit_type>"      ! Sets how the merit is calculated.
                                             ! default = "limit"
  default_low_lim      = <number>            ! Lower variable value limit.
                                             ! default = -1e30
  default_high_lim     = <number>            ! Upper variable value limit.
                                             ! default =  1e30
  default_key_bound    = <logical>           ! Variables to be bound?
  default_key_delta    = <number>            ! 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      = "<integer>"         ! Universe containing variable to
                                             !    be controlled. "*" => All.
  var(i)%weight        = <number>            ! Merit function weight.
  var(i)%step          = <number>            ! Small step size.
  var(i)%low_lim       = <number>            ! Lower variable value limit
  var(i)%high_lim      = <number>            ! 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     = <number>            ! 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. See §8.1 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* (§1.5). 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 4 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 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 (§1.6). 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.

## 6.9 Initializing Data and Constraints

A set of data (§3) 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 4 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         = <number>            ! Merit function weight.
                                               !   Default = 0.0
  default_data_source    = "<source>"          ! "lattice" (default), or "beam".
  ix_min_data            = <integer>           ! Minimum array index.
  ix_max_data            = <integer>           ! Maximum array index.
  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           = "<number>"         ! Datum "measured" value
  datum(j)%weight         = "<weight>"         ! Merit function weight.
  datum(j)%good_user      = <logical>          ! Use for optimization and plotting?
  datum(j)%data_source    = "<source>"         ! "lattice", or "beam"
  datum(j)%ix_bunch       = <integer>          ! Bunch index.
                                               !   0 (default) = all bunches.
/
```

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:12.3"   ""        "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. ...
/
```

`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* (§3.2). 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>`.

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
  !           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...
```

Certain data types like the `emittance` along a Linac can be calculated from the lattice or can be calculated from the distribution of particles when `"beam"` is performed. Which is calculation is performed is determined by the `datum(:)%data_source` component. Possibilities are:

```
"beam"      ! Calculation using the beam distribution.
"dat"       ! Use data from a d1_data array.
"lat"       ! (Default) Calculation using the lattice layout.
```

`"lat"` is the default. With `"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 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 (§1.6). The `-no_lords` and `-no_slaves` switchs, 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`. Certain types are recognized by *Tao*. These are given by Table 4.2. Custom data types not specified in this table must have a corresponding definition in `tao_hook_load_data_array.f90`. See Chapter 10 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.

## 6.10 Initializing Plotting

### 6.10.1 Plot Window

Plotting is defined by an initialization file whose name is defined by the `tao_start` namelist (§6.3). 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                 = <b_x1>, <b_x2>, <b_y1>, <b_y2>, "<units>"
```

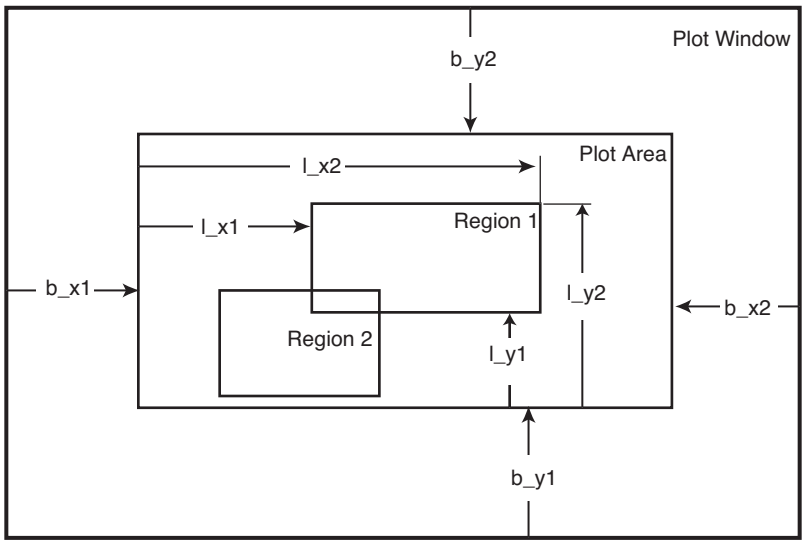



Figure 6.1: Regions define where on the plot page plots are placed.

```
    plot_page%text_height            = <num> ! height in POINTS. Def = 12
    plot_page%main_title_text_scale  = <num> ! Relative to text_height. Def = 1.3
    plot_page%graph_title_text_scale = <num> ! Relative to text_height. Def = 1.1
    plot_page%axis_number_text_scale = <num> ! Relative to text_height. Def = 0.9
    plot_page%axis_label_text_scale  = <num> ! Relative to text_height. Def = 1.0
    plot_page%legend_text_scale      = <num> ! Relative to text_height. Def = 0.8
    plot_page%key_table_text_scale   = <num> ! Relative to text_height. Def = 0.9
    plot_page%title(i)               = <string>, <x>, <y>, "<units>", "<justify>"
    plot_page%shape_height_max       = <num> ! Max half height for drawing elements.
                                             !   Def = 40
    plot_page%n_curve_pts            = <num> ! Num points used to construct a
                                             !   smooth curve. Def = 401
    plot_page%box_plots              = <T/F> ! For debugging. Def = F.
    region(i) = "<region_name>" <l_x1>, <l_x2>, <l_y1>, <l_y2>  ! % plot area
    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:

```
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 (§6.10.5).
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
```

`plot_page%border` sets a border around the edges of the window. As shown in Figure 6.1 `b_x1`, `b_x2` are the right and left border widths and `b_y1` and `b_y2` 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` = "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)` is an array of five elements that defines the $i^{th}$ region. The first element of this array is the name of the region. This name may not contain a dot ".". The second and third elements of the array, `l_x1`, and `l_x2`, define the location of the left and right edges of the region as a fraction of the plot area width starting from the left edge of the plot area. The final elements of the `region(i)` array, `l_y1` and `l_y2`. define the location of the bottom and top edges of the region as a fraction of the height of the plot area with respect to the plot area's bottom edge. Thus, in the above example, region 1 extends from the left border of the plot area (`region(1)%l_x1` = 0) to the right border (`region(1)%l_x2` = 0) and vertically from the center (`region(1)%l_y1` = 0.5) to the top edge (`region(1)%l_x2` = 1.0). Regions may overlap any one can define as many regions as one likes.

`shape_height_max` sets the maximum size for drawing shapes in `floor_plan`, and `lat_layout` drawings. See Section §6.10.4 for more details.

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

`default_plot` sets the defaults for any `plots` defined in the `tao_template_plot` namelists (§6.10.2). Similarly, `default_graph` sets defaults for the `graph` structure defined in the `tao_template_graph` namelist (§6.10.2). In the example above, the default x-axis min and max are set to 100 and 200 respectively.

### 6.10.2 Plot Templates

As shown in Figure 1.1, 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 transfered 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" or "s". Default is "index".
  plot%ix_universe = <number> ! used for lat_layout plots
  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.
  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 (§6.10.2). This overrides `default_graph` settings made in the `tao_template_plot` namelist (§6.10) 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` (§7.32) and `scale` (§7.22) 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
```

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            = <number>
  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%x                = <qp_axis_struct> ! Horizontal axis.
  graph%y                = <qp_axis_struct> ! Left axis.
  graph%y2               = <qp_axis_struct> ! Right axis.
  graph%n_curve          = <number>         ! number of curves
  graph%clip             = <logical>        ! Clip curves at boundary? Default = T
  graph%draw_axes        = <logical>        ! Default = T
  graph%component        = "<string>"       ! Eg: "model - design"
  graph%correct_xy_distortion  = <logical>  ! For Floor Plan plots: Default = F
  graph%x_axis_scale_factor = <factor>      ! Scale the x-axis by this.
  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".
  curve(i)%data_type           = "<string>" ! Default = plot%name.graph%name
  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>  ! Draw data symbols?
  curve(i)%ix_universe         = <number>   ! Default = -1 => Use viewed universe
  curve(i)%ix_branch           = <number>   ! 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       = <num>         ! Index number of reference element.
    curve(i)%smooth_line_calc = <Logical>     ! Calc data between symbol points?
  /
```

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      = "dat"
    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%component` sets what variable or data components are to be plotted. In the above example, what will be plotted is `model - design`. Possible components are:

```
  "model"     ! model values.
  "design"    ! design values.
  "base"      ! Base values
  "meas"      ! data values.
  "ref"       ! reference data values.
```

The default, if `graph%component` is not specified, is for the graph will show `model` values.

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

```
  "data"         ! Data plots (default)
  "floor_plan"   ! A 2-dimensional birds-eye view of the machine (§6.10.6).
  "key_table"    ! Key binding table for single mode (§6.10.5).
  "lat_layout"   ! Schematic showing placement of the lattice elements (§6.10.4).
  "phase_space"  ! Phase space plots (§6.10.8).
```

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

With graph%type set to floor_plan, the layout of the machine is drawn. The *Bmad* global reference system is covered in Chapter 1 of the *Bmad* reference manual. With the *Bmad* global reference system the $X - Z$ axes define the horizontal plane The conversion between *Bmad* global axes and *Tao* graph axes is:

```
Bmad      Tao
X    ->   -y
Z    ->   -x
```

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 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. To prevent the drawing of the axes set graph%draw_axes to F

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.

The curve structure is used to define the curves that are 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:

```
"dat"               ! 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 dat, 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 (§7.24). curve%draw_symbol_index determines whether the index number is printed besides the symbol. Use the set curve command (§7.23) to toggle the drawing of symbols.

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 (§7.23) to toggle the drawing of lines.

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.

### 6.10.3 Graphing a Data Slice

The standard data graph, as presented in the previous subsection, plots data from a given d1_data arrays. 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 then an example of a data slice is the collection of points (x, y) where:

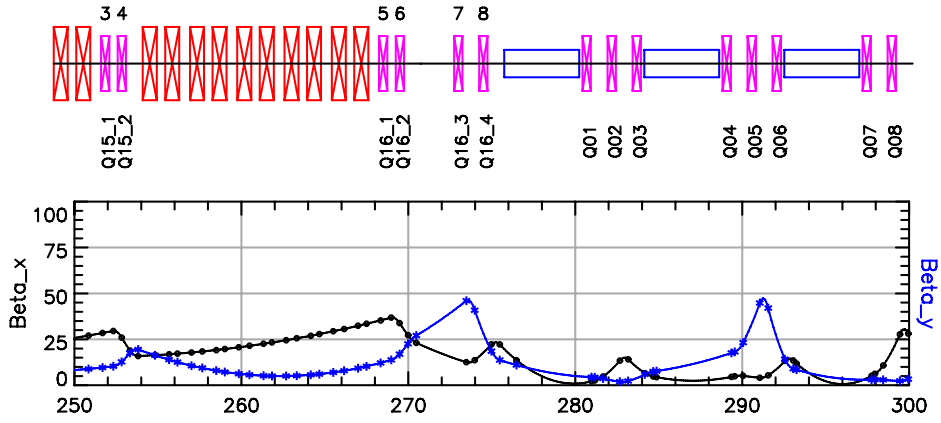
```
(x, y) = (<n>@orbit.x[23], <n>@orbit.y[23]),   <n> = 1, ..., n_universe
```

When defining a template for graphing a data slice, the plotbe set to "data", the curve(:)%data_source must be set to "dat" 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 = "dat"
  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,




| | 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 6.2: 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.

`orbit.y[23]` is used for the y-axis. The `set` command (§7.23) can be used to change `curve%ele_ref_name` and `graph%component` strings.

`curve%data_index` sets the index number for the symbol points (§6.10.2). 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` (§7.24) will print the symbol index number along with the $(x, y)$ coordinates of the symbols.

Arithmetic expressions (§1.7) 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.

### 6.10.4 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 6.2. The `tao_template_plot` needed to define a lattice layout looks like:

```
&tao_template_plot
  plot%name        = "<plot_name>"
  plot%x%min       = <number>
  plot%x%max       = <number>
  plot%n_graph     = <number>
  plot%x_axis_type = "s"
/
&tao_template_graph
  graph_index       = <number>
  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 currently viewed universe
  graph%ix_branch   = <integer> !  0 => use main lattice.
  graph%margin      = <ix1>, <ix2>, <iy1>, <iy2>, "<Units>"
  graph%n_curve     = 0
/
```

Example:

```
&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, 0.12, 0.12, "%BOX"
  graph%n_curve     = 0
/
```

Which elements are drawn is under user control and is defined using an `element_shapes_lat_layout` namelist. See Section §6.10.7 for more details.

### 6.10.5 Drawing a Key Table

The `key table` is explained more fully in Section §8.1. An example is shown in Figure 6.2. A template to create a key table looks like:

```
&tao_template_plot
```

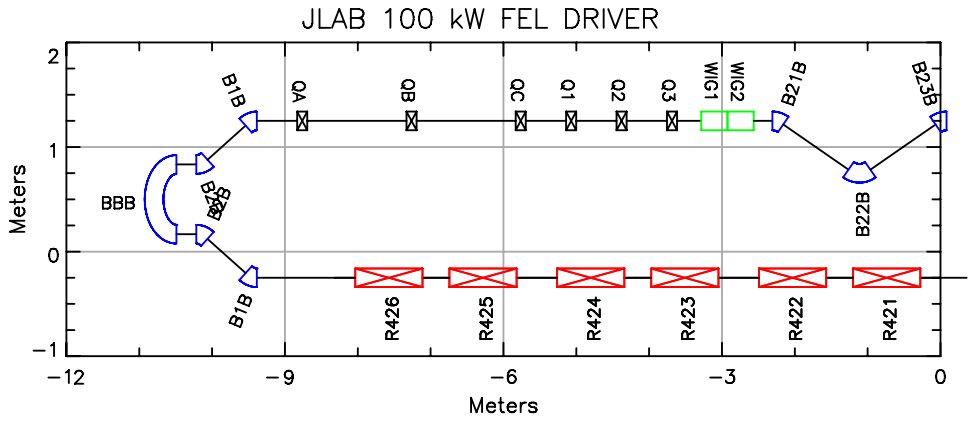



Figure 6.3: Example Floor Plan drawing.

```
  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. 6.2) 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

### 6.10.6 Floor Plan Drawing

A `Floor Plan` drawing gives a display of the machine projected onto the horizontal plane. An example is shown in Figure 6.3. Like a `Lattice Layout` (§6.10.4), Elements are represented by colored rectangles and which elements are drawn is determined by an `element_shapes_floor_plan` namelist (see §6.10.7).

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 $(X, Z)$ plane is horizontal plane. The conversion between the $(X, Z)$ coordinates of the `global reference system` and the $(x, y)$ coordinates of the Floor Plan plot are:

```
Global        Plot
```

```
Z    <--->   -x
X    <--->   -y
```

The `show ele` command (§7.24) can be used to view an element's global coordinates. Additionally, the global coordinates at the start of the lattice can be defined in the lattice file. See the Bmad manual for more details.

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
/
```

If `graph%correct_xy_distortion` is set to `True` (default is `False`), then 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.

### 6.10.7 Element Shapes

`Floor plan` (§6.10.6) 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: `element_shapes_lat_layout` for the lattice layout and `element_shapes_floor_plan` for floor plan drawings. Two different namelists are used since it is sometimes artistically pleasing to use, say, a different size for floor plans and lattice layouts.

The namelist syntax is the same for both:

```
&element_shapes_lat_layout
  ele_shape(i) = "<element_list>" "<shape>" "<color>" "<v_size>" "<Label_type>"
/
&element_shapes_floor_plan
  ele_shape(i) = "<element_list>" "<shape>" "<color>" "<v_size>" "<Label_type>"
/
```

For Example:

```
&element_shapes_floor_plan
```

```
  !               element_list       Shape     Color     Size   Label_Type
  ele_shape(1) = "quadrupole::q*"   "box"     "red"     30     name
  ele_shape(2) = "quadrupole::*"    "xbox"    "red"     30     none
  ele_shape(3) = "sbend::*"         "box"     "blue"    15     none
  ele_shape(4) = "wiggler::*"       "xbox"    "green"   20     name
/
```

A figure is drawn for each lattice element in the lattice that matches the `<element_list>` specification (§1.6) 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 the last shape matched will be used. For a floor plan, for `wiggler`s that have an `x_ray_line_len` attribute, The X-ray line will be drawn if an `ele_shape` for a `photon_branch` is present.

The `<shape>` parameter is the shape of the figure drawn. Valid Shapes are:

```
"box"             -- Rectangular box
"xbox"            -- Rectangular box with an x through it.
"var_box"         -- Rectangular box with variable height.
                      The box is symmetric about the center line.
"asym_var_box"    -- Like var_box but is not symmetric about the center line.
"diamond"         -- Diamond shape.
"bow_tie"         -- Bow-tie shape.
"circle"          -- Circle centered at center of element.
```

The height of a `var_height_box` is proportional to the element strength. For example, for a quadrupole the height is proportional to the `K1` focusing strength. Not all lattice elements can be used with a `var_height_box`.

`<color>` is the color of the shape. Good colors to use are:

```
"black"
"red"
"orange"
"magenta"
"yellow"
"green"
"cyan"
"blue"
"purple"
```

`<v_size>` is the vertical size of the shape in points (72 points = 1 inch). The measurement is from the centerline so, for example, a `box` element will have a total height of twice `<v_size>`. Since a `var_box` or `asym_var_box` can get arbitrarily large, the parameter `plot_page%shape_height_max` (§6.10.1). sets the maximum drawn size.

Finally `<Label_Type>` indicates what type of label to print next to the corresponding element glyph. Possibilities are:

```
name            -- The element name
none            -- No label is drawn.
s               -- Draw longitudinal s position.
```

The default is `"name"`

Note: There is an old, deprecated syntax where both the lattice layout and floor plan drawings are specified via one `element_shapes` namelist.

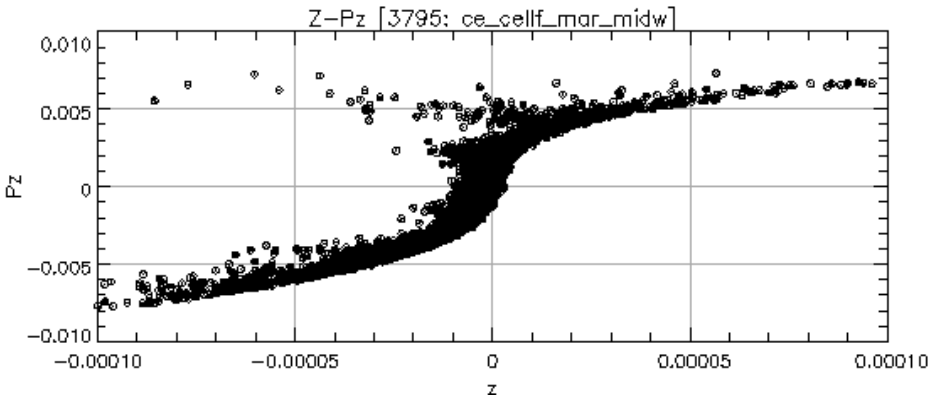



Figure 6.4: Example Phase Space plot.

### 6.10.8 Phase Space Plotting

A `Phase Space` plot displays a particle or particles phase space coordinates at a given location. An example is show in Figure 6.4. Example Phase Space template:

```
&tao_template_plot
  plot%name = "zphase"
  plot%x%min =   -10e-3
  plot%x%max = 10e-3
  plot%x%major_div_nominal = 4
  plot%x%label = "z"
  plot%n_graph = 1
/

&tao_template_graph
  graph_index = 1
  graph%name = "z"
  graph%type = "phase_space"
  graph%box = 1, 1, 1, 1
  graph%title = "Z-Pz"
  graph%margin =  0.12, 0.12, 0.12, 0.12, "%BOX"
  graph%y%label = "Pz"
  graph%y%max = 3
  graph%y%min = -3
  graph%y%major_div_nominal = 4
  graph%n_curve = 1
  curve(1)%data_type_x = "z"
  curve(1)%data_type = "pz"
  curve(1)%data_source = "beam"
  curve(1)%ele_ref_name = "BEGINNING"
/
```

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"
```

In this example above, the $x$-axis of the plot will correspond to the $x$ phase space coordinate and the $y$-axis will correspond to the $px$ coordinate.

To change the place at which the `phase_space` curve is drawn use the `set curve ele_ref_name` or `set curve ix_ele_ref` commands.

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 (§6.9). 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 7

# 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 §8, 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 (§7.25).

Commands are case sensitive. The list of commands is shown in Table 7.1. 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` (§7.1) 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:

```
view 2; show global  ! Two commands and a comment
```

| *Command* | *Section* | *Command* | *Section* |
|---|---|---|---|
| alias | §7.1 | quit | §7.17 |
| call | §7.2 | read | §7.18 |
| change | §7.3 | restore | §7.19 |
| clip | §7.4 | reinitialize | §7.20 |
| continue | §7.5 | run | §7.21 |
| derivative | §7.8 | scale | §7.22 |
| end-file | §7.6 | set | §7.23 |
| exit | §7.7 | show | §7.24 |
| flatten | §7.9 | single-mode | §7.25 |
| help | §7.10 | spawn | §7.26 |
| history | §7.11 | use | §7.27 |
| misalign | §7.12 | veto | §7.28 |
| output | §7.13 | view | §7.29 |
| pause | §7.14 | wave | §7.30 |
| place | §7.15 | x-axis | §7.31 |
| plot | §7.16 | x-scale | §7.32 |
| | | xy-scale | §7.33 |

Table 7.1: Table of *Tao* commands.

## 7.1 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 $i^{th}$ 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.

## 7.2 call

The `call` command opens a command file (§1.10) and executes the commands in it. Format:

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

*Tao* first looks in the current directory for the file. If not found *Tao* will look in the directory pointed to by the `TAO_COMMAND_DIR` directory. Up to 9 arguments may be passed to the command file. The $i^{th}$ 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.

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]]".

## 7.3 change

The `change` command changes element attribute values or variable values in the `model` lattice. Format:

```
change element {n@}<name_or_number> <attribute> <number>
change {-silent} variable <name> <locations> <number>
change  {n@}beam_start <coordinate> <number>
```

Generally `<number>` is added to the existing value of the attribute or variable. That is:

```
final_model_value = initial_model_value + <number>
```

If "@" is prepended to <number> then just the value of <number> is used to set the value
```
final_model_value = <number>
```

If "d" is prepended to <number> then the final value will be the design value plus <number>:
```
final_model_value = design_value + <number>
```

If "%" is prepended to <number> then the final value will be
```
final_model_value = initial_model_value * (1 + <number> / 100)
```

For change element, the <name_or_number> may be a name, an element number or list of numbers, or an element class (EG: "quadrupole"). Element names may contain the wild cards "*" which represents any number of characters or can be "%" which represents any single character.

For change element, and change beam_start, The optional n@ universe specification (§1.3) may be used to specify the universe or universes to apply the change command to.

For linear lattices, change beam_start <coordinate> <number> can be used to vary the starting coordinates where <coordinate> is one of:
```
(x, px, y, py, z, pz)
```

For circular lattices only the pz component is applicable. Also for linear 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.

Note: To set, say, datum values, etc. use the set command.

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.
```

## 7.4 clip

The clip command vetoes data points for plotting and optimizing. That is, the good_user logical of the datums 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
```

## 7.5 continue

The `continue` command is used to continue reading of a suspended command file (§1.10) after a `pause` command (`s:pause`). Format:

```
continue
```

## 7.6 end-file

The `end-file` command is used in command files (§1.10) 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
```

## 7.7 exit

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

```
exit
```

## 7.8 derivative

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

```
derivative
```

## 7.9 flatten

The `Flatten` command runs the optimizer to minimize the merit function. This is the same as `run`. See the `run` command for more details. Format:

```
flatten {<optimizer>}
```

## 7.10 help

The `help` command gives help on *Tao* commands. Format:
```
help {<command> {<subcommand>}}
```

The environmental variable `TAO_DIR` must be defined so *Tao* can find any help files.

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 command.
help show       ! Help on the show command.
help show alias ! Help on the show alias command.
```

## 7.11 history

The `history` command shows or reruns prior commands. Format:
```
history           ! Print the command history.
history <number>  ! Re-execute a command by number.
history <string>  ! Re-execute last command that begins with <string>.
```

Every *Tao* command entered is recorded in a "history stack" and these commands can be viewed and reinvoked as needed.

Examples
```
history 34   ! Re-execute command number 34.
history set  ! Re-execute last set command.
```

## 7.12 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 viewed universe 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 viewed
   ! 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
```

## 7.13 output

The `output` command creates various files. Format:

```
  output bmad_lattice {<file_name>}       ! Write a Bmad lattice file of the model
  output beam {-ascii} {-at <element_name_or_index>} {<file_name>}
                                          ! Write beam distribution data.
  output covariance_matrix {file_name}    ! Write the covariance and alpha matrices
                                          !   from the Levenburg (lm) optimization.
  output curve <curve_name> {<file_name>} ! Write the curve data
  output derivative_matrix {file_name}    ! Write the dModel_Data/dVar matrix.
  output digested {<file_name>}     ! Write a digested Bmad lattice file of the model.
  output gif {<file_name>}          ! create a gif file of the plot window.
  output hard                       ! Print the plot window to a printer.
  output hard-l                     ! Like "hard" except use landscape orientation.
  output mad8_lattice {<file_name>} ! Write a MAD-8 lattice file of the model
  output madx_lattice {<file_name>} ! Write a MAD-X lattice file of the model
  output ps {-scale <scale>} {<file_name>}
                                    ! Create a PS file of the plot window.
  output ps-l {-scale <scale>} {<file_name>}
                                    ! Create a PS file with landscape orientation.
  output 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
  ------------------            ------------------
  output bmad_lattice           lat_#.bmad
  output beam                   beam_#.dat
  output curve                  curve
  output derivative_mat         derivative_matrix.dat
  output digested               digested8_lat_universe_#.bmad
  output gif                    tao.gif
  output mad8_lattice           lat_#.mad8
  output madx_lattice           lat_#.madx
  output ps                     tao.ps
  output variable               global%var_out_file
```

where `#` is replaced by the universe number. `output 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 `output 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.

`output beam` will create a file of the particle positions when beam tracking is being used. If the switch `-at` is present then only the particle positions at the given element are written. Otherwise the positions at all elements will be written. 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* (§6.2).

Note: PGPLOT does a poor job producing gif files so consider making a postscript file instead and using a ps to gif converter.

## 7.14 pause

The `pause` command is used to pause *Tao* when executing a command file (§1.10). 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 (§7.5) 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.
```

## 7.15 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
```

## 7.16 plot

The `plot` command is used to determine what components are plotted in the graphs of a given region. Format:

```
plot <region> <component>
```

Use a "-" for baselines.

Examples:

```
plot bottom model - design       ! Plot model - design in the bottom region
plot top meas - model + design - ref
```

## 7.17 quit

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

```
quit
```

## 7.18 read

The `read` command is used to modify the currently viewed `model` lattice. Format:

```
read lattice <file_name>
```

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.

## 7.19 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 datums between 34
                                 !                                          and 56
restore data *@orbit[34]         ! un-veto orbit data in all universes.
restore var quad_k1[67]          ! un-veto variable
```

## 7.20 reinitialize

The `reinitialize` command reinitializes various things. Format:

```
reinitialize beam
reinitialize data
reinitialize tao {-init <tao_input_file>} {-beam_all <beam_file>}
         {-beam0 <beam_file>} {-lat <lattice_file>} {-noplot}
```

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 §6.2 for details on the arguments. If there are no arguments, the `reinitialize` command uses the same arguments that were used in the last `reinitialize` command, or, if this is the first reinitialization, what was used to start *Tao*. If there are arguments, The defaults will be used. The default `<init_file>` is what was used with the last reinitialization or, if this is the first reinitialization, what was used at startup.

The `-beam0` optional argument reads in a beam data file which is used to initialize the beam at the beginning of the lattice. This overrides the `beam0_file` variable (§6.7).

The `-noplot` optional argument surpresses the opening of the plot window.

Examples:

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

## 7.21 run

The `run` command runs an optimizer. Format:

```
run {<optimizer>}
```

If `<optimizer>` is not given then the default optimizer is used. 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:

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

See §4.1 for more details on the different optimizers.

Examples:

```
run
run de
```

## 7.22 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` (§6.10.2) 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
```

## 7.23 set

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

```
set beam_init {n@}<component> = <value>
set bmad_com <component> = <value>
set curve <curve> <component> = <value>
set data <data_name>|<component> = <value>
set global <component> = <value>
set graph <graph> <component> = <value>
set lattice {n@}<destination_lat> = <source_lat>
set plot <plot> <component> = <value>
set plot_page <component> = <value1> {<value2>}
set universe <what_universe> on/off
set universe <what_universe> recalculate
set universe <what_universe> mat6_recalc on/off
set var <var_name>|<component> = <value>
set wave <component> = <value>
```

Note: For setting element attributes in the `model` lattice use the `change` command.

To apply a set to all data or variable classes use "*" in place of `<data_name>` or `var_name`.

**set beam_init {n@}<component> = <value>**

For `set beam_init`, the `<component>`s that can be set can be found in section §6.7. The optional `n@` allows the specification of the universe or universes the set is applied to. The default is to set the viewed universe. Use the `show beam` command (§7.24) 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.
```

**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.
```

**set curve <curve> <component> = <value>**

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

```
ele_ref_name      = <string>  ! Name of reference element
ix_ele_ref        = <number>  ! Index of reference element
ix_universe       = <number>  ! Universe index.
symbol_every      = <number>  ! Symbol skip number.
draw_line         = <logical>
draw_symbols      = <logical>
draw_symbol_index = <logical>
```

See Section §6.10.2 for a description of these components. Use the `show curve` (§7.24) to view the settings of the components.

Examples:

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

**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 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 graph <graph> <component> = <value>**

For `set graph`, the `component`s that can be set are:

```
component   = <string>
clip        = <logical>
ix_universe = <number>
margin%x1   = <number>
margin%x2   = <number>
margin%y1   = <number>
margin%y2   = <number>
```

Example:

```
set graph orbit.x component = model - design
                          ! Plot model orbit - design orbit in the graph
```

**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 (§1.3) 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
```

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 currently viewed universe.
```

**set plot <plot> <component> = <value>**

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

```
autoscale_x = <logical>
autoscale_y = <logical>
```

Example:

```
set plot orbit.x component = model - design
                          ! Plot model orbit - design orbit in the graph
```

**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 title = 'XYZ'  ! Set plot page title string
```

**set universe <what_universe> on/off**
**set universe <what_universe> recalculate**
**set universe <what_universe> mat6_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 viewed universe, 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> mat6_recalc` command will set whether the 6x6 transfer matrices are calculated for a given universe. Turning this off is useful in speeding up calculations in the case where the transfer matrices are not being used (Warning: The transfer matrices are needed to compute the Twiss parameters). Use the `show universe` command to see the state of the `mat6_recalc_on` switch.

Example:

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

**set var <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.
```

Example:

```
set var quad_k1|weight = 0.1         ! Set quad_k1 weights.
```

**set wave <component> = <value>**

The `set wave` command sets the boundries of the $A$ and $B$ regions for the wave analysis (§5). The components are

```
ix_a = <ix_a1> <ix_a2>  ! A-region left and right boundries.
ix_b = <ix_b1> <ix_b2>  ! B-region left and right boundries.
```

Example:

```
set wave ix_a = 15 27    ! Set A-region to span from datum #15 to #27
```

## 7.24 show

The `show` command is used to display various about the state of *Tao*. Format:

```
show -append <file_name> ...
show -write <file_name> ...
show alias
show beam {<element_name_or_index>}
show constraints
show curve {-symbol} {-line} <curve_name>
show data {<data_name>}
show derivative <data_name(s)> <var_name(s)>
show element {-taylor} {-wig_terms} {-data} {-all_attributes} <ele_name>
show global
show graph <graph_name>
show hom
show key_bindings
show lattice {-all_tracking} {-0undef} {-branch <branch>} {-custom <file_name>}
       {-lords} {-middle} {-no_label_lines} {-blank_replacement <string>}
       {-no_tail_lines} {-s <s1>:<s2>} {<elements>}
show optimizer
show particle {<bunch_index>.}<particle_index> {<element_index>}
show particle -lost {<bunch_index>}
show plot
show plot {<template_plot_name>}
show plot {<plot_region_name>}
show plot -shapes
show top10 {-derivative}
show taylor_map {-order <n_order>} {loc1 {loc2}}
show universe {universe_number}
show universe -connections
show use
show value <expression>
show variable {<var_name>}
show variable <universe_number>@
show variable -bmad_format {-good_opt_only}
show wave
```

The `show` command has an optional argument to write the results to a file. as well as printing the information at the terminal. The format for this is:

```
show -write <file_name> <rest_of_the_command>
show -append <file_name> <rest_of_the_command>
```

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".
```

**show alias**

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

**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.

**show constraints**

Lists data and variable constraints.

**show curve {-symbol} {-line} <curve_name>**

Show information on a particular curve of a particular plot. See §1.8 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.

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".
```

**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 the currently viewed universe.
show data -1@               ! Same as "show data"
show data *@                ! Shows all d2_data in all universes.
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
```

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

Shows the derivative dModel_Value/dVariable. This derivative is used by the optimizer lm.

Example:

```
show deriv orbit.x[23] k1[34] ! Show dModel_Value/dVariable Derivative.
```

**show element {-taylor} {-wig_terms} {-data} {-all_attributes} <ele_name>**

This shows information on lattice elements. The syntax for <ele_name> is explained in section §1.6. 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 <ele_name> is a number $n$, then the $n^{th}$ element in the lattice list will be shown.

If the -taylor switch is present then the Taylor map associated with an element, if there is one, is also displayed. Similarly, if the -wig_terms switch is present, then any associated wiggler terms for a map_type wiggler element is printed.

If the -all_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.

If the -data switch is present, then information about the all the datums associated with the element will be listed.

Example:
```
show ele q*                 ! list all elements with names beginning with "q".
show ele q10w               ! Show a particular lattice element.
show ele -all_att 105       ! Show element #105 in the lattice.
```

**show global**
Shows information on the global parameter structure (§6.5).

**show graph <graph_name>**
Show information on a particular graph of a particular plot. See §1.8 for the syntax on plot, graph, and curve names. Use show plot to get a list of plot names.

Example:
```
show curve r2.g1          ! Show the attributes of graph "g1" which is
                          !   plotted in region "r2".
```

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

**show key_bindings**
Shows all key bindings.

**show lattice {-all_tracking} {-0undef} {-branch <branch>} {-custom <file_name>} {-lords} {-middle} {-no_label_lines} {-blank_replacement <string>} {-no_tail_lines} {-s <s1>:<s2>} {<elements>}**
Shows Twiss and orbit data, etc. for the model lattice 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.

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 supress the header lines at the bottom of the table.

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

The -branch can be used to specify the branch of the lattice. The default is the main branch (# 0).

The locations to show can either be specified using the -elements switch, or by specifying a longitudinal position range with -s, or (the default) by specifying a range of element indices. The syntax used for specifying which elements to show when the -elements switch is used is given in section §1.6. For example:
```
show lat -ele marker:bpm*   !
```
This will show the parameters at all marker elements whose name begin with "bpm".

Alternatively, a range of elements can be specified using the element index, name, or the element's longitudinal position with a ":" being used to separate the index. For example
```
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 -s  23.9:55.3       ! Show elements whose position is between
                             !   23.9 meters and 55.3 meters.
```
The -all_tracking switch can be used to show all the elements in the tracking part of the lattice. The element_list is optional but if it is present it must be at the end of the command line.

If neither -elements, -all, 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::#[beta_a]",       "f7.2",   7
  column(7) = "1e3 * ele::#[orbit_x]  "f8.3",   8, "Orbit_x| (mm)"
/
```

each `column(1)` has four components. The first component is what is to be displayed in that column. Algebraic expressions are permitted (§1.7). Note: Use of `lat::` 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.

To encode the element index, use a `#`. 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, `quadrupole`s 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)"` is:

```
show lattice 50:100         ! Show lattice elements with index 50 through 100
```

**show optimizer**

Shows information pertinent to optimization: Data and variables used, etc.

**show opt_vars**

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

**show particle {<bunch_index>.}<particle_index> {<element_name_or_index>}**
**show particle -lost {bunch_index}**

The command `show particle {<bunch_index>.}<particle_index>` shows information on a particle at a given element. The default for the optional `{bunch_index}` index is set by the global variable `global%bunch_to_plot`. The default `<element_name_or_index>` is 0 (the starting position). Also see `show beam`.

`show particle -lost` shows which particles are lost during beam tracking. The default for the optional `{bunch_index}` index is set by `global%bunch_to_plot`. 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.

Examples:

```
show part 3.47 8   ! Shows information on particle #47 of bunch #3 at
                   !   lattice element #8.
show part 47 8     ! Same as above except the default bunch is used.
show part -lost 3  ! Show lost particle positions for bunch #3
```

**show plot**
**show plot {<template_plot_name>}**
**show plot {<plot_region_name>}**
**show plot -shapes**

A simple `show plot` displays which templates are being plotted and in which regions and also all available templates. See §1.8 for the syntax on plot, graph, and curve names. A `show plot <plot_name>` will display information on a particular plot.

If the `-shapes` switch is present, the shapes used in drawing `floor_plan` or `lat_layout` plots are printed.

**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.

**show taylor_map {-order <n_order>} {loc1 {loc2}}**

Shows the Taylor transfer map for the `model` lattice of the currently `viewed` universe from the exit end of the element given by `loc1` to the exit end of the element given by `loc2` where `loc1` and `loc2` are element names or indexes. The exception is that if `loc1` is present but `loc2` is not then the map is from the beginning of the lattice to the exit end of `loc2`. If neither `loc1` nor `loc2` are present, the transfer map is computed for the entire lattice.

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 q10e q10w  ! 0th and 1st order maps from q10e to q10w
show taylor 45                  ! Transfer map from element #0 to #45
```

**show universe {universe_number}**
**show universe -connections**

If the `-connections` switch is present, show information on the connections between various universes. Otherwise, shows various parameters associated with a given universe. If no universe is specified then the current viewed universe is used. Parameters displayed include tune, chromaticity, radiation integrals, etc.

**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.

**show value <expression>** Shows the value of an expression. Examples:

```
show value 3@lat:orbit.x[34]  ! orbit at lattice element #34
show value sin(0.35)
```

**show variable {<var_name> <locations>}**
**show variable <universe_number>@**
**show variable -bmad_format {-good_opt_only}**

Shows variable information. If `<var_name>` is not present, a list of all `v1_var` classes is printed. To show variables associated with the `n`th universe use the syntax `show var n@`.

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.

Examples:

```
show var             ! List all 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 2@          ! List all variables that control attributes in universe 2.
show var -bmad       ! List variables in Bmad Lattice format.
```

**show wave** The `show wave` command shows the results of the current wave analysis (§5).

## 7.25 single-mode

The `sing-mode` command puts *Tao* into `single mode` (§8). Format:

```
single-mode
```

## 7.26 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)
```

## 7.27 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 viewed 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 viewed universe.
use data *@*                 ! use all data in all universes.
```

## 7.28 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.

## 7.29 view

The `view` command changes which universe data is taken from for plotting. Format:

```
view <number>
```

This also sets the default universe that commands are applied to in the absence of a universe prefix (§1.3).

Examples:

```
view 2   ! Make universe #2 the default.
```

## 7.30 wave

The wave command sets what data is to be used for the wave analysis (§5). 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. 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.

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.
```

## 7.31 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
```

## 7.32 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 (§6.10.2) 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 plotnice looking axis. The result can be that if two plots have the same range of data but differing major division settings, the `x-scale` command can produce differing results.

Example:

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

## 7.33 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 8

# 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 §7, *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.

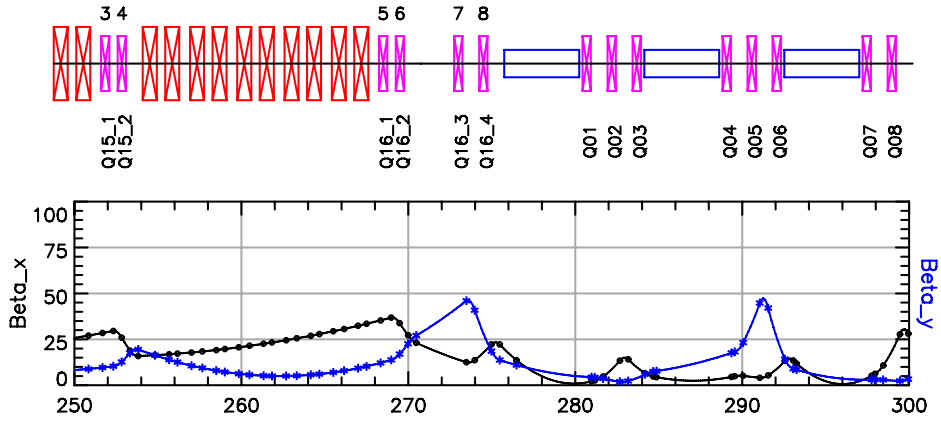
From `line mode` use the `single-mode` command (§7.25) to get into `single mode`. To go back to `line mode` type `"Z"`.

## 8.1 Key Bindings

The main purpose of Single Mode is to associate certain keys with certain variables so that the pressing of these keys will change their associated variable. This is called a `key binding`. Key bindings are established via the `var(i)%key_bound` logical (See Section §6.8). The variables are divided into banks of 10. The $0^{th}$ bank uses the first ten variables that have their `key_bound` attribute (§6.8) 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 (§6.10.5)can be setup as shown in Figure 8.1. 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   ("(")
  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` (§6.8. value for a variable. Note: In `line mode`, the command `show key_bindings` (§7.24) may




| | 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 8.1: 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.

be used to show the entire set of bound keys.

Initially the 0Th bank is active. The left arrow and right arrow are used to decrease or increase the bank number. Additionally the "<" and ">" keys can be used to change the deltas for the variables.

For example, looking at Figure 8.1, 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.

## 8.2 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.

**?** 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. 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 (§7) command.

/**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 viewed universe.

/**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.

# Part III

# Programmer's Guide

# Chapter 9

# Customizing Tao

## 9.1 It's all a matter of Hooks

The golden rule when extending *Tao* is that you are only allowed to replace routines or redefine structures that have the name "hook" in them. If you have the source code then it's within your power to modify any routine in *Tao* as much as you like. However, as time goes by, and revisions are made to the *Tao* routines to extend the usefulness of *Tao* and to eliminate bugs, only modifying the "hook" routines will ensure that custom changes will have a minimum impact on the specialized routines that will be written by various people.

*Tao* is written in Fortran 95 and a knowledge of Fortran is required. However, if you know C then Fortran can be learned in a couple of days. Because of the interoperability between C and Fortran once the wrapper routines are written to interface with *Tao* the rest of your coding can, in principle all be done in C.

*Tao* relies on extensive use of pointers and logical flags. However, all of the structures you will need to use are contained in the `tao_struct.f90` module. This module is heavily documented and provides all the information needed to use the intrinsic *Tao* structures on your customizations. It is also a very good idea to have a copy of `bmad_struct.f90` handy as this contains most of the structures used by *Bmad*.

## 9.2 Compiling your custom Tao

The *Tao* libraries can be compiled without compiling an executable. Here is where this comes in handy. Since the standard *Tao* subroutines have already been made into libraries, all you need to do is compile and link your custom routines with the standard *Tao* subroutines into an executable.

There are 11 "hook" files located in the `ROOT/tao/hook` directory. These are the files you can customize. There are two options here.

1. Change the files directly in `ROOT/tao/hook`, adding any extra files you may need, then recompile from the `ROOT/tao` directory with `gmake -f M.tao`.

2. Copy the hook files to a separate directory say `ROOT/my_tao`, adding any extra files you may need, then write a Makefile to compile and link these routines to the main *Tao* library.

Option 2 is HIGHLY recommended because it keeps the *Tao* distribution tree undisturbed and reserves the possibility to create multiple custom *Tao* programs using the same vanilla *Tao* library. This option is used in the following example.

## 9.3 An Example

As an example 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...). First, we should copy all the hook files to a separate directory, call it `ROOT/my_tao`. Also include the main program file from the `ROOT/tao/program` directory. (replace `ROOT` with whatever top directory you placed the `tao` directory.)

```
mkdir ROOT/my_tao
cp ROOT/tao/hook/*.f90 ROOT/my_tao
cp ROOT/tao/program/tao_cl.f90 ROOT/my_tao/my_tao_cl.f90
```

Next we need a Makefile. The `ROOT/tao/M.tao` Makefile is a great starting point.

```
cp ROOT/tao/M.tao ROOT/my_tao/Makefile
```

Now change the following lines in your Makefile

```
LIB_SRC_DIRS := ./code ./hook
OBJ_SRC_DIRS := ./program
```

to

```
LIB_SRC_DIRS :=
OBJ_SRC_DIRS := ./
```

This Makefile will tell gmake to use the tao library that has already been created (from `../tao/code` but the actual library is located at `../lib/libtao.a`) and then to compile all of the hook files, including the main program file (`my_tao_cl.f90`) into object files (everything in `./`, the current directory). Routines and declarations in object files always override similarly named code in the *Tao* libraries so this allows for your local hook files to override the dummy hook files in the *Tao* library. The only downside to this method is it clutters your `my_tao` directory with object files. You can always remove these object files with `gmake clean`.

There are two more lines to alter. change

```
MAIN_FILE :=
```

to

```
MAIN_FILE := ./my_tao_cl.f90
```

and finally,

```
MAKEFILE := M.tao
```

to

```
#MAKEFILE := M.tao !using default name for Makefile
```

Notice that this line is just being commented out with a '#'. You are now ready to make your customizations to the hook routines.

This example will only require the modification of one file: `tao_hook_evaluate_a_datum.f90`. The formula for single particle emittance is

$$\epsilon = \gamma x^2 + 2\alpha x x' + \beta x'^2 \tag{9.1}$$

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

```
case ('particle_emittance.x')

  datum_value =  ( ele%x%gamma * tao_lat%orb(ix1)%vec(1)**2 + &
   2 * ele%x%alpha * tao_lat%orb(ix1)%vec(1) * tao_lat%orb(ix1)%vec(2) + &
   ele%x%beta * tao_lat%orb(ix1)%vec(2)**2)

case ('particle_emittance.y')

  datum_value = ( ele%y%gamma * tao_lat%orb(ix1)%vec(3)**2 + &
   2 * ele%y%alpha * tao_lat%orb(ix1)%vec(3) * tao_lat%orb(ix1)%vec(4) + &
   ele%y%beta * tao_lat%orb(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 initialization files used for this tutorial.

```
cp ROOT/tao/program/*.init ROOT/my_tao
cp ROOT/tao/program/*.lat ROOT/my_tao
```

In `ROOT/my_tao/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_tao/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 = 'dat'
  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 = 'dat'
  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 6.

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/my_tao
gmake
../bin/my_tao
```

Notice that the name of the custom *Tao* program is `my_tao`. If you run '`../bin/tao`' then you will run "vanilla" *Tao*.

After your custom *Tao* initializes type

```
place bottom particle_emittance
scale
```

Your plot should look like Figure 9.1.

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`.

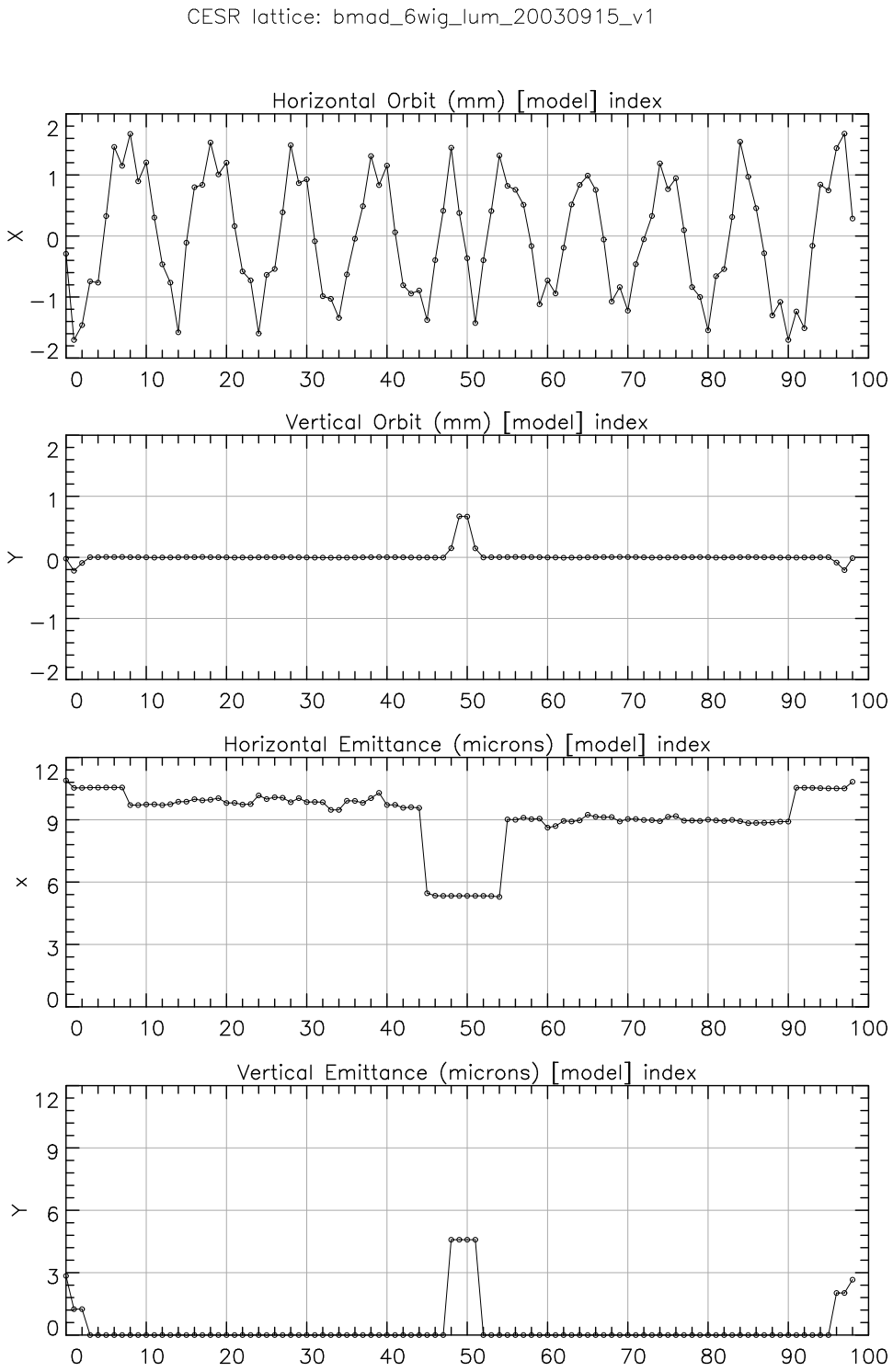



Figure 9.1: Custom data type: non-normalized emittance

## 9.4 Other Customizations

The above example just illustrates one of the customizations you can perform on *Tao*. The next chapter lays out all of the hook files and provides pointers for various customizations.

# Chapter 10

# Creating a Custom Version of Tao

*Tao* has been designed to be readily extensible with a minimum of effort. The tutorial provides a simple example of a custom data type. Here each the hook routines is explained and pointers are given on writing code.

The process for customizing is summarized as follows. For the purposes of this discussion assume that the directory that you are developing *Tao* in is called `ROOT`.

1. To extend *Tao* you will want to make a new directory, say, called `ROOT/my_tao`. In this directory you write the necessary routines to extend *Tao*. You will also need a standard `Makefile` for building programs.

2. You can compile and link your routines with the *Tao* routines using `gmake` in `ROOT/my_tao`.

The tutorial in Part I of this manual gave an example of how to carry out the above steps.

## 10.1 Creating the Tao Library and a Custom Tao Directory

After obtaining the *Tao* distribution the *Tao* library is created by typing `gmake` in the `ROOT/tao` directory. This will create two libraries called `libtao.a` and `libtao_g.a` in the directory `ROOT/lib` where the second is a debug version. If you then type `gmake -f M.tao "vanilla"` *Tao* will be compiled into the executables called `tao` and `tao_g` and placed in `ROOT/bin`

## 10.2 Modifying the Hook Routines and Structures

The golden rule when writing routines to extend *Tao* is that you are only allowed to replace routines or redefine structures that have the name "hook" in them. The reason for this 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 that will be written by various people. What happens if you need to replace or modify a non–hook routine or structure? 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.

Before one can begin writing code one must understand the structures that *Tao* uses. The structures are defined in a file `tao_struct.f90`. It is a good idea to have a copy of `tao_struct.f90` easily accessible. In particular, examine the `tao_super_universe_struct` and `tao_universe_struct` structures. They reference all other structures, either directly or indirectly, in `tao_struct`. *Tao* is based upon the *Bmad* software package for the simulation of relativistic charged particles and the *Tao* structures have components that are defined in *Bmad*. For information on these structures see the *Bmad* Reference Manual. Any hook structures can be defined in the file `tao_hook_mod.f90` but see the entry below on this file before adding any structures.

Also, 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. The routines to calculate the data structures are also called from within this module.

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.

### 10.2.1 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 §1.12.

### 10.2.2 tao_hook_does_data_exist

This routine is for use with custom data types. This routine sets `datum%exists` appropriately for datums where the rules built into the `tao_init_data` routine incorrectly flag the datum.

### 10.2.3 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.

*Tao* evaluates data at each element while each lattice is being calculated. At initialization time *Tao* determines which datums are to be evaluated at each element then calls `tao_evaluate_a_datum` at each element for each datum that needs to be evaluated. If a range of elements are specified for a datum then `tao_evaluate_a_datum` is called for this datum at the last element in the range. `tao_evaluate_a_datum` starts by calling `tao_hook_evaluate_a_datum` to evaluate the custom data types.

As explained in the dummy file, the datum merit type affects how a datum's value should be calculated. There is a helper subroutine in the dummy hook routine called `load_it` to aid in modifying the datum's value based on the merit type. If there is a range of elements associated with datum then a merit type other than `target` requires that the entire range of elements associated with the datum be searched for the appropriate value to be returned. See Chapter 4 for details on the merit type.

For example, if the data type is `orbit.x` (yes, this is already defined in *Tao*) then the appropriate case item in `tao_hook_load_data_array` would be:

```
select case (datum%data_type)

case ('orbit.x')
  call load_it (orb(:)%vec(1))
```

`load_it` will then look at each datum. If the merit type is other than `target` then `load_it` will search the appropriate range of elements for either the minimum or maximum horizontal orbit value and this will be the datum's value. Because, a datum may refer to a range of elements, the entire orbit array (from 0 to `n_ele_max`) is passed to `load_it` for each `orbit.x` datum.

If the only merit type that is going to be used is `target` then `load_it` can be ignored.

### 10.2.4 tao_hook_graph_data_setup

Use this to setup custom graph data for a plot.

### 10.2.5 tao_hook_init

After the lattice and all global and universe structures are initialized then `tao_hook_init` is called. 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.

### 10.2.6 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.

### 10.2.7 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.

### 10.2.8 tao_hook_merit_data

A custom data merit type can be defined here. Table 4.2 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.

### 10.2.9 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.

### 10.2.10 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.

### 10.2.11 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 DCSLIB's `quick_plot` works.

### 10.2.12 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.

### 10.2.13 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.

### 10.2.14 tao_hook_mod

Here any custom structures are defined. In the dummy hook routine there are already a number of structures defined. These are used in `tao_struct.f90` so that even if they are not used there is a dummy structure defined so that the compiler doesn't complain.

**This module is different from all the other hook routines in that the *Tao* library must be compiled after this file is modified so that `tao_struct` knows about the hook structure**. If hook structures are needed but do not need to be accessed from any `tao_struct.f90` structures then it is recommended that these be placed in a separate file so that it can be compiled when the custom *Tao*

program is compiled. Because of the special status of this hook module, it is not placed in the standard hook directory but in the `tao/code/` directory.

# Bibliography


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

[2] 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)

[3] 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).

[4] 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

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

[6] D. Sagan, "Betatron phase and coupling correction at the Cornell Electron/Positron Storage Ring", Phys. Rev. ST Accel. Beams 3, 102801 (2000).

# Index