FRIB

# phantasy Documentation

*Release 1.0.7*

Tong Zhang

Sep 14, 2018

# DOCUMENTATION

**phantasy Python package**

`phantasy`: Physics High-level Applications aNd Toolkit for Accelerator SYstem

**Author** Tong Zhang

**E-mail** zhangt@frib.msu.edu

`phantasy` is the name of a Python package, which is created for the high-level physics controls on the accelerator facilities. The users (typically are accelerator physicists, scientists) could build complex higher-level physics applications based on the high-level physics controls environment that provided by `phantasy`, rather than disturbing by the trivial lower-level controls system. Also, the following highlighted features are included in this package[1]:

- Device configuration management
- Device abstraction
- Online modeling
- Python interactive scripting environment for high-level controls
- Virtual accelerator based on EPICS control environment
- Web service integration

[1] The feature list may grow or change as the development moving forward.

CHAPTER

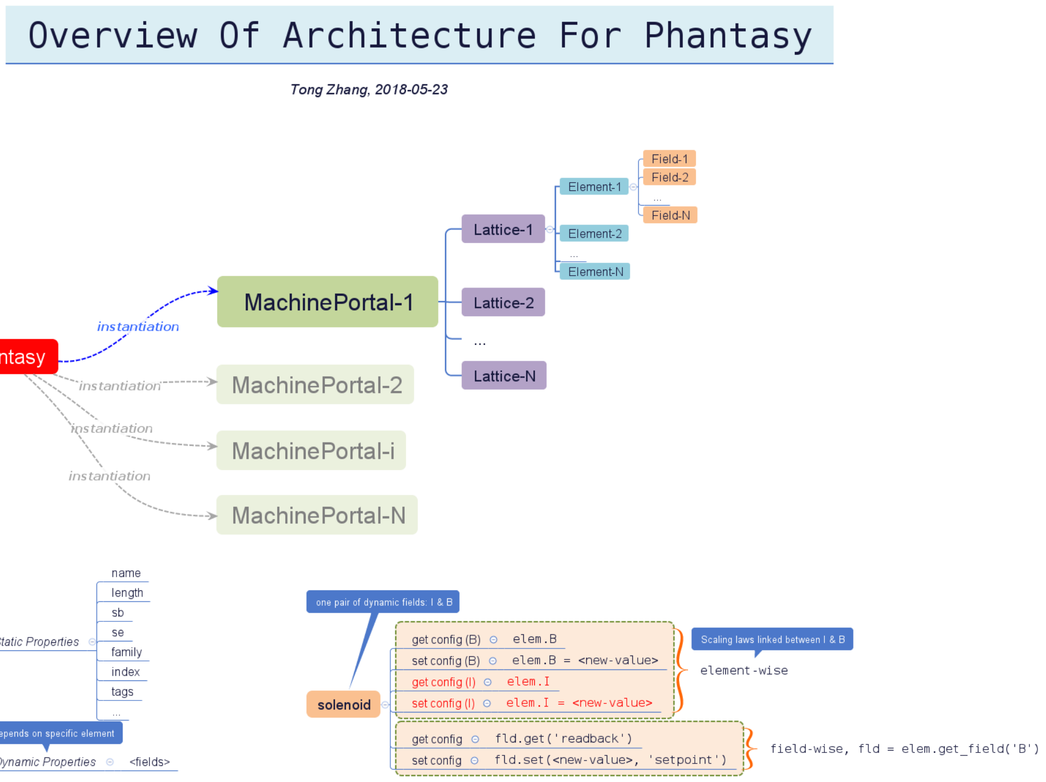
# ONE

# INTRODUCTION

## 1.1 Package Name

phantasy is brief for **P**hysics **H**igh-level **A**pplications a **n**d **T**oolkit for **A**ccelerator **Sy**stem.

## 1.2 Architecture



### 1.2.1 Lattice

### 1.2.2 Element

Element with Channel Access support is instantiated from `CaElement`, which in aggregate attaches relevant information with the target device.

From the user side, interested information of the element can be reached by referring the attributes, which are composed of static and dynamic fields, the static fields are to represent the device properties that do not change often, especially the attribute name itself in Python object, e.g. device name, type, geometry length and location, etc., while the dynamic fields are to represent the device properties regarding to Channel Access ability, that is:

1. The attribute name is not fixed, depends on the device configuration;
2. The value of the attribute usually is not fixed, depends on the runtime;
3. Each dynamic field is instantiated from `CaField`.

### 1.2.3 Field

CHAPTER

# TWO

# DEPLOYMENT

Deploy `phantasy` to different operating systems is quite simple, both online and offline approaches are provided. Before installation, there may be packages/libraries dependency issues to be resolved first.

## 2.1 Prerequisites

Required Python packages: `numpy`, `scipy`, `matplotlib`, `cothread`, `xlrd`, Optional Python packages: `tornado`, `motor`, `jinja2`, `humanize`, `jsonschema`, Optional Python packages: `pyCFClient`, `scanclient`, Suggested packages: `phantasy-machines`, `python-unicorn`, `unicorn-webapp`,

Other home-made packages: - Python: `flame`, `genopt` - C++: `flame`, `impact` (FRIB-version), `dakota-drivers`

## 2.2 Install via APT

This is the recommended way to deploy `phantasy`, however, the **APT** way is **FRIB intranet only**.

The target workstation is running Debian 8, add the following lines to `/etc/apt/sources.list` or save as a separated file to the directory `/etc/apt/sources.list.d`:

```
deb http://ci.frib.msu.edu/debian/ jessie unstable
deb-src http://ci.frib.msu.edu/debian/ jessie unstable
```

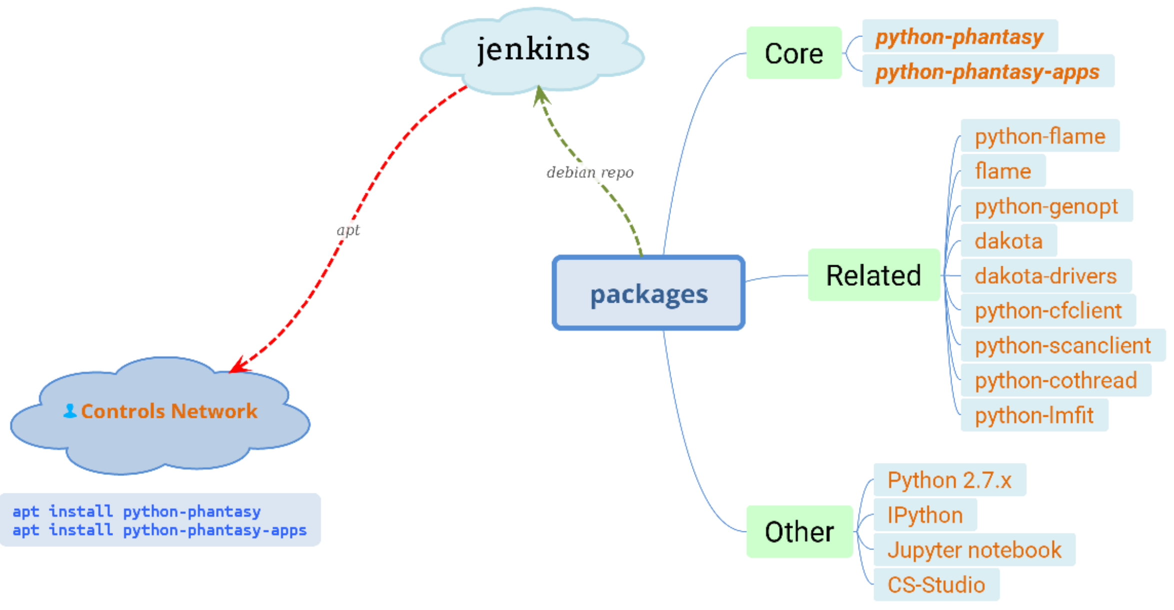
The public key can be imported by[1]:

```
wget http://ci.frib.msu.edu/debian/repo_key.gpg -O - | sudo apt-key add -
```

After that, in the terminal, issue `sudo apt-get update` and `sudo apt-get install python-phantasy` to install `phantasy`, `apt` will handle all the dependencies automatically.

[1] Details see: http://ci.frib.msu.edu/

### 2.2.1 FRIB controls network case



## 2.3 Install via PIP

Download `.whl` package of `phantasy` from HERE, select the newest version, and install it by:

```
pip install <phantasy.VERSION.whl>
```

Or upgrade from earlier version by:

```
pip install <phantasy.VERSION.whl> --upgrade --no-deps
```

Or simply install by:

```
pip install phantasy
```

## 2.4 Run Tests

After installation, commands `test_phantasy` (for Python 2.7) and `test_phantasy3` (for Python 3.x)[2] can be invoked to run tests distributed with `phantasy` package.

Alternative way to do in Python terminal:

```
>>> from phantasy.tests import main
>>> main()
```

[2] The name *test_phantasy3* is only valid when deployed by APT way.

CHAPTER

THREE

# USER GUIDE

Here goes some examples to show how to use `phantasy` package to fulfill various objectives, the development guide should see *API References*.

## 3.1 Modeling Accelerator with Lattice File

`FLAME`[1] is a new envelope tracking code developed by FRIB for the purpose of modeling (ion) accelerator efficiently, especially cases of multi-charge states could be right handled, now it is still under development.

`FLAME` itself has Python interface with the package name of `flame`. Another Python package so-called `flame_utils`[2], which used to be one of the modules of `phantasy` is under developing, not only to make the `flame` Python interface more friendly and clear, but also to better support the entire high-level physics applications, which is a part of high-level control system.

### 3.1.1 Getting started

Here is the simplest way to model an accelerator from FLAME lattice file.

```
1  from flame import Machine
2  import flame_utils
3
4  # create FLAME machine
5  latfile = "test.lat"
6  m = Machine(open(latfile, 'rb'))
7
8  # create BeamState object
9  bs = flame_utils.BeamState(machine=m)
10
11 # create ModelFlame object
12 fm = flame_utils.ModelFlame()
13 # setup machine and state
14 fm.bmstate, fm.machine = bs, m
15
16 # setup observers and run flame model
17 obs = fm.get_index_by_type(type='bpm')['bpm']
18 r,s = fm.run(monitor=obs)
19
20 # get data of intereset from running results
21 data = fm.collect_data(r, pos=True, x0_env=True, ref_IonEk=True)
```

The lattice file used here could be downloaded from `here`.

The 9th line is to create a general FLAME beam state object, which is a super class of FLAME interal state, one can use this object as the same API as the FLAME interal state (except `show()` method).

[1] https://github.com/frib-high-level-controls/FLAME
[2] https://github.com/frib-high-level-controls/flame-utils

The 12th line is to create a `ModelFlame` object (see `ModelFlame`), after that, `machine` and `bmstate` should be assigned to make it alive. The `machine` attribute is just the FLAME machine object, `bmstate` could accept both FLAME interal state or `BeamState` (see `BeamState`), for the possible user-customized states, the `BeamState` is recommanded to include all the operations upon the machine states object.

The advantage of re-invent the new `BeamState` is to improve the user experience in the Python CLI environment that support auto-completion, e.g. *ipython*, then all properties that `BeamState` has could be reached by double hitting `<TAB>`; moreover, additional attributes could be defined in `BeamState` to make the higher level interface more clear and clean, see details at *API References* and *General FLAME beam state*.

The `run()` method of `ModelFlame` is used to simulate the model, and `collect_data()` could be used to extract the data-of-interest from the simulation results, then other operations could be done, e.g. data plotting.

**Note:** Method `run()` does not change `BeamState` inplace, instead one can get the updated `BeamState` from the second element of the returned tuple, see 18th line.

### 3.1.2 General FLAME beam state

FLAME beam state is the most essential object in modeling an accelerator. The Python interface of `FLAME` represents the state as `_internal.State`, could be created by `allocState()` method, however, there are only two methods (`clone()` and `show()`) that are exposed explicitly, one of the reasons is that `_internal.State` is designed for not only `MomentMatrix` simulation type.

In order to make it more user-friendly, `BeamState` class is created specifically for the case of `MomentMatrix` simulation type, and exposing as many attributes as possible, since *Explicit is better than implicit*[3].

For a typical `BeamState` object, the following attributes could be reached at the moment:

- `pos`,
- `ref_beta`, `ref_bg`, `ref_gamma`, `ref_IonEk`, `ref_IonEs`, `ref_IonQ`, `ref_IonW`, `ref_IonZ`, `ref_phis`, `ref_SampleIonK`,
- `beta`, `bg`, `gamma`, `IonEk`, `IonEs`, `IonQ`, `IonW`, `IonZ`, `phis`, `SampleIonK`,
- `moment0`, `moment0_rms`, `moment0_env`, `moment1`
- `x0`, `xp0`, `y0`, `yp0`, `phi0`, `dEk0`
- `x0_env`, `xp0_env`, `y0_env`, `yp0_env`, `phi0_env`, `dEk0_env`
- `x0_rms`, `xp0_rms`, `y0_rms`, `yp0_rms`, `phi0_rms`, `dEk0_rms`

**Todo:** More attributes, that could be calculated from `_internal.State` could be added to `BeamState` class.

#### Create BeamState object

Basically, there are several ways to initialize the `BeamState`, slight differences should be aware of.

**Note:** `BeamState` needs FLAME machine information (got from `machine` or `latfile` keyword parameter) to do further initialization, especially, for the case of the beam state is composed of pure zeros.

[3] https://www.python.org/dev/peps/pep-0020/

### Approach 1: Initialize with pre-defined `flame._internal.State` object

Fist create machine and state object by FLAME Python package:

```
>>> import flame_utils
>>> from flame import Machine
>>>
>>> latfile = 'test.lat'
>>> m = Machine(open(latfile, 'rb'))
>>> s = m.allocState({})
>>> m.propagate(s, 0, 1)
```

Then, `BeamState` can be created by:

```
>>> bs = flame_utils.BeamState(s)
```

or:

```
>>> bs = flame_utils.BeamState()
>>> bs.state = s
```

Now, one can use `bs` object just the same way as `_internal.State`, e.g. `bs` can be passed as the first argument of `m` machine object's `propagate()` method; `bs` also can duplicated by `clone()` method; and even the represetation of `bs` itself is much similar as `s`,i.e. `print(bs)` would gives `BeamState: moment0 mean=[7](-0.0007886,1.08371e-05,0.0133734,6.67853e-06,-0.000184773,0.000309995,1)`

### Approach 2: Initialize with pre-defined FLAME machine object

This approach will initialize a `BeamState` object with the initial attributes' values from the pre-defined FLAME machine object, e.g.:

```
>>> bs = flame_utils.BeamState(machine=m)
>>> print(bs)
BeamState: moment0 mean=[7](-0.0007886,1.08371e-05,0.0133734,6.67853e-06,-0.000184773,0.000309995,1)
```

Also can do this by assigning `latfile` keyword parameter:

```
>>> bs = flame_utils.BeamState(latfile=latfile)
```

### Approach 3: Initialize with another `BeamState` object

For example:

```
>>> bs1 = flame_utils.BeamState(latfile=latfile)
>>> bs2 = flame_utils.BeamState(bmstate=bs1)
```

**Note:** `clone()` could be used to create a copy, e.g. `bs2 = bs1.clone()`.

### Configure BeamState object

To confiugre `BeamState` is to set new values to attributes, which could be done through properties setter methods, e.g. the initial kinetic energy of reference charge state can be adjusted by:

```
>>> bs.ref_IonEk = 100000
```

The same rule applies to the scalar properties, however different rule should be applied when updating array properties, e.g. `moment0`, whose value is a numpy array, if even only one element of that array needs to be changed, one should create a new array and assign to `moment0`, rather than updating inplace, here is the example:

```
>>> # before adjustment
>>> print(bs.moment0)
array([[ -7.88600000e-04],
       [  1.08371000e-05],
       [  1.33734000e-02],
       [  6.67853000e-06],
       [ -1.84773000e-04],
       [  3.09995000e-04],
       [  1.00000000e+00]])
>>> # right way to change the first element of moment0
>>> m0_val = bs.moment0
>>> m0_val[0] = 0
>>> bs.moment0 = b0_val
>>> print(bs.moment0)
array([[  0.00000000e+00],
       [  1.08371000e-05],
       [  1.33734000e-02],
       [  6.67853000e-06],
       [ -1.84773000e-04],
       [  3.09995000e-04],
       [  1.00000000e+00]])
```

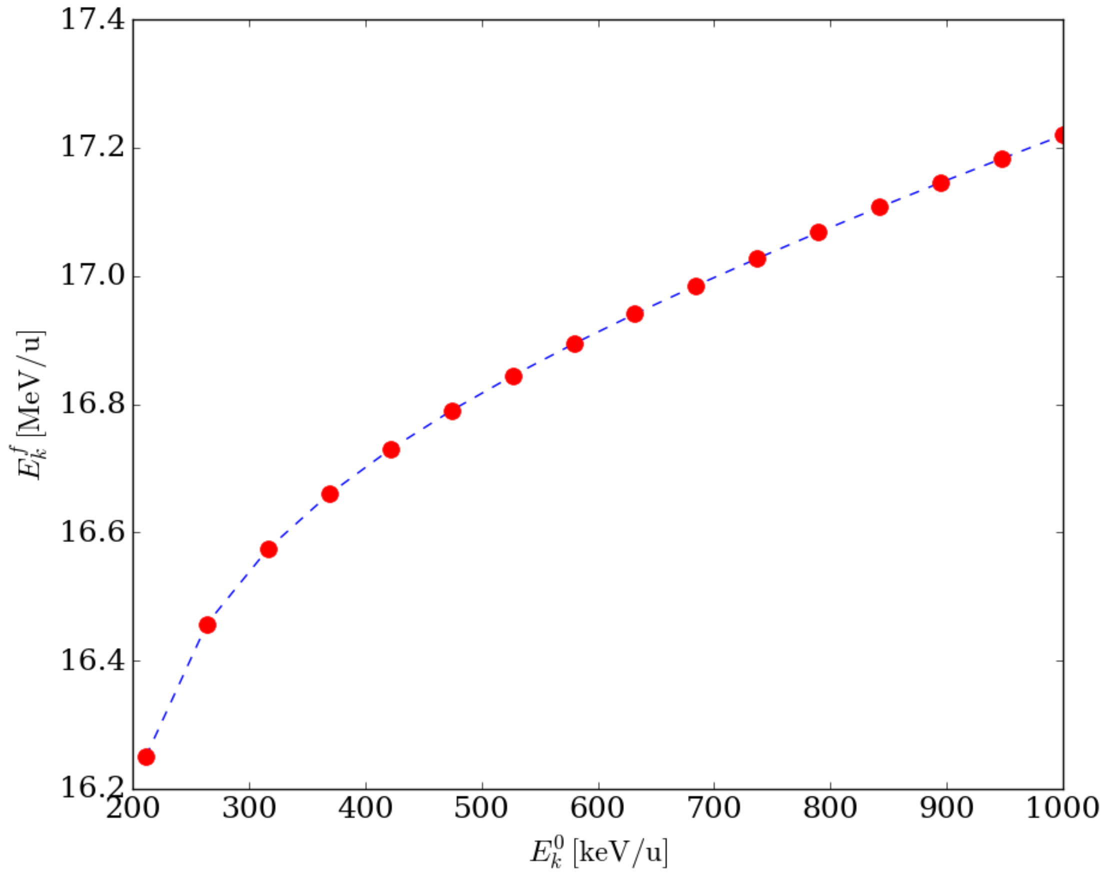
### Use BeamState object

Different `BeamState` represent different initial conditions for modeling processes, here are the possible cases:

#### Scan initial kinetic energy

```
import matplotlib.pyplot as plt
import numpy as np

from flame import Machine
import flame_utils


latfile = "test.lat"
m = Machine(open(latfile, 'rb'))

ek_out = []
ek0_arr = np.linspace(1, 1000, 20)
for ek0 in ek0_arr:
    bs = flame_utils.BeamState(machine=m)
    bs.ref_IonEk = ek0 * 1000

    fm = flame_utils.ModelFlame()
    fm.bmstate, fm.machine = bs, m

    obs = fm.get_index_by_type(type='bpm')['bpm']
    r,s = fm.run(monitor=obs)

    ek_out.append(s.ref_IonEk)
```

Final reference ion kinetic energy v.s. initial input values could be shown as the following figure:



**Tip:** To disable logging messages from `flame_utils`:

```
from flame_utils import disable_warnings
disable_warnings()
```

To disable logging messages from `flame`:

```
import logging
logging.getLogger('flame.machine').disabled = True
```

## 3.2 Modeling Accelerator Facility

One of the key features of `phantasy` is to abstract the accelerator devices into hierarchical structure, which opens the portal for the high-level object-oriented controls way, and based on which, modularized high-level controls software could be developed.

### 3.2.1 Configuration Files

Currently, `phantasy` uses (at most) seven (7) types of configuration files to describe the entire accelerator, including device abstraction, online-modeling, controls PV management, etc., among which, the file to integrate the EPICS channels or PV names with accelerator layout is very important. The full list of configuration file types are:

| Type Name | File name[1] | Description |
|---|---|---|
| Machine configuration | phantasy.ini | Global configuration for loading specific machine |
| Device/Model configuration | phantasy.cfg | Configuration for physics models and devices |
| Channels configuration | channels.csv | Includes EPICS PV names and device properties |
| Layout configuration | layout.csv | Devices geometrical layout configuration |
| Settings configuration | settings.csv | Initial settings for physics models |
| Unit scaling configuration | unicorn.xlsx | Scaling laws for physics/engineering units conversion |

#### Machine configuration

Machine configuration file (typical name: *phantasy.ini*) is the main file that used by `phantasy` `MachinePortal` class to instantiate the seleted accelerator and specific segment, which is `machine` and `segment` parameter of `MachinePortal` initialization method, respectively.

#### Channels configuration

Generally, each device can have one or more than one PV names, which depends on the number of fields to be controlled, please note for this high-level physics controls framework, the only interested fields are physics related, e.g. one solenoid usually only has one field to control (read and write), that is the current applied on the device, while the RF cavity should have both phase and amplitude of electric field to control. In this configuration file, all these information should be included, togther with other general meta-information as device position, length, type, etc.

### 3.2.2 Simple Example

Here is a simple example of *channels.csv* (`HERE`) to describe the PV names in a machine named *simple_machine*, of segment name *TEST*:

[1] Typical file names used by *phantasy-machines* package, see here for the details of each file.

Table 1: Table Title

| PV | elemField_eng | elemField_phy | elemHandle | elemIndex | elemLength | elemName | elemPosition | elemType | machine | physicsName | physicsType | pvPolicy | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FE_SCS1:PSOL_D0704:I_CSET | I | B | setpoint | 1 | 0.3998 | FE_SCS1:SOLR_D0704 | 0.3998 | SOL | LIVE | solenoid | SOL_S46 | DEFAULT | phantasy.sys.FE | phantasy.sub.SCS1 | LEBT | LINAC |
| FE_SCS1:PSOL_D0704:I_RSET | I | B | readset | 1 | 0.3998 | FE_SCS1:SOLR_D0704 | 0.3998 | SOL | LIVE | solenoid | SOL_S46 | DEFAULT | phantasy.sys.FE | phantasy.sub.SCS1 | LEBT | LINAC |
| FE_SCS1:PSOL_D0704:I_RD | I | B | readback | 1 | 0.3998 | FE_SCS1:SOLR_D0704 | 0.3998 | SOL | LIVE | solenoid | SOL_S46 | DEFAULT | phantasy.sys.FE | phantasy.sub.SCS1 | LEBT | LINAC |
| FE_SCS1:PSQ1_D0726:V_CSET | V | V | setpoint | 2 | 0.2052 | FE_SCS1:QHE_D0726 | 2.5252 | EQUAD | LIVE | ES_quad | QUAD | EQUAD | phantasy.sys.FE | phantasy.sub.SCS1 | LEBT | LINAC |
| FE_SCS1:PSQ1_D0726:V_RSET | V | V | readset | 2 | 0.2052 | FE_SCS1:QHE_D0726 | 2.5252 | EQUAD | LIVE | ES_quad | QUAD | EQUAD | phantasy.sys.FE | phantasy.sub.SCS1 | LEBT | LINAC |
| FE_SCS1:PSQ1_D0726:V_RD | V | V | readback | 2 | 0.2052 | FE_SCS1:QHE_D0726 | 2.5252 | EQUAD | LIVE | ES_quad | QUAD | EQUAD | phantasy.sys.FE | phantasy.sub.SCS1 | LEBT | LINAC |
| FE_SCS1:PSQ2_D0726:V_CSET | V | V | setpoint | 2 | 0.2052 | FE_SCS1:QHE_D0726 | 2.5252 | EQUAD | LIVE | ES_quad | QUAD | EQUAD | phantasy.sys.FE | phantasy.sub.SCS1 | LEBT | LINAC |
| FE_SCS1:PSQ2_D0726:V_RSET | V | V | readset | 2 | 0.2052 | FE_SCS1:QHE_D0726 | 2.5252 | EQUAD | LIVE | ES_quad | QUAD | EQUAD | phantasy.sys.FE | phantasy.sub.SCS1 | LEBT | LINAC |
| FE_SCS1:PSQ2_D0726:V_RD | V | V | readback | 2 | 0.2052 | FE_SCS1:QHE_D0726 | 2.5252 | EQUAD | LIVE | ES_quad | QUAD | EQUAD | phantasy.sys.FE | phantasy.sub.SCS1 | LEBT | LINAC |

While the machine configuration for this example is:

```
[COMMON]
segments: TEST
default_segment: TEST
root_data_dir: /tmp/phantasy_data

[TEST]
cfs_url: channels.csv
cfs_tag: LINAC
cfs_property_names: elem*, pvPolicy
#settings_file: settings.json
#layout_file: layout.csv
#config_file: phantasy.cfg
```

The typical snippet to abstract the whole segment could be:

```
>>> from phantasy import MachinePortal
>>> mp = MachinePortal(machine='simple_machine', segment='TEST')
>>> lat = mp.work_lattice_conf
>>> for e in lat:
>>>     print(e)
0001 | FE_SCS1:SOLR_D0704   SOL       0.00  [m] 0.399800 [m]
0002 | FE_SCS1:QHE_D0726    EQUAD     2.32  [m] 0.205200 [m]
```

All the devices could be reached via *mp.work_lattice_conf*.

CHAPTER

FOUR

# FEATURES

## 4.1 Work with MachinePortal

### 4.1.1 Set Up Testing Environment

#### Install Docker

Please ref to https://docs.docker.com/install/ for all platforms.

#### Start IOC container

Start the test IOC by:

```
docker run -d --name phantasy-ioc tonyzhang/phantasy-ioc:jessie
```

After that, bunch of PVs are alive and ready to control, among which the PV named `fac` could be used to control the random noise level, set it with `0` (`caput fac 0`) will totally disable noise, i.e. all PVs serve with the constant value (`0.1`).

### 4.1.2 Physics High-level Controls

Instantiate `MachinePortal` class with the machine configuration named as `FRIB` and segment named as `LEBT`:

```
# import packages and modules
In [1]: from phantasy import MachinePortal

# create MachinePortal instance
In [2]: mp = MachinePortal(machine='FRIB', segment='LEBT')
```

## 4.2 Work with Device

This note demonstrates how to get work with the device, the device usually can be reached by accessing the abstracted Python object, specifically, the instance of `CaElement`, in which fundamental APIs are created to make the device operation easy and functional.

Assuming the following `MachinePortal` instance *mp* has already been created, otherwise please see *Work with MachinePortal*.

The generic device operation procedure is:

1. Locate the interested device(s) or element(s);

2. Control the device(s) by calling methods or accessing attributes;
3. Post-processing the for other consumers.

### 4.2.1 Locate Device

Just as the name implies, *mp* is the entry to all the configurations of the loaded machine, the following snippet shows how can we locate the interested device(s). All valid device types can be known by `get_all_types()`:

```
In [1]: from phantasy import MachinePortal

In [2]: mp = MachinePortal(machine="FRIB", segment="LEBT")

In [3]: mp.get_all_types()
Out[3]: ['EQUAD', 'SOL', 'HCOR', 'PM', 'VCOR', 'BCM', 'BEND', 'CAV', 'EBEND', 'FC']
```

The method `get_elements()` is created for the general purpose of element searching, e.g. all the solenoid could be located by passing the `type` parameter with the value of `SOL` ( which is one member of the list returned from *get_all_types()* method), the returned result is a list, so if the first one is wanted, simply referring by `[0]`.

```
In [4]: all_sols = mp.get_elements(type='SOL')

In [5]: all_sols
Out[5]:
[FE_SCS1:SOLR_D0704 [SOL] @ sb=70.206995,
 FE_LEBT:SOLR_D0787 [SOL] @ sb=78.481995,
 FE_LEBT:SOLR_D0802 [SOL] @ sb=79.957145,
 FE_LEBT:SOLR_D0818 [SOL] @ sb=81.597295,
 FE_LEBT:SOLR_D0951 [SOL] @ sb=94.894800,
 FE_LEBT:SOLR_D0967 [SOL] @ sb=96.534950,
 FE_LEBT:SOLR_D0982 [SOL] @ sb=98.010100,
 FE_LEBT:SOLR_D0995 [SOL] @ sb=99.320100]

In [6]: first_sol = all_sols[0]

In [7]: first_sol
Out[7]: FE_SCS1:SOLR_D0704 [SOL] @ sb=70.206995
```

### 4.2.2 Inspect Device

Each one within the list returned from `get_elements()` is the instance of `CaElement`, which is bundled of various information, represented as attributes of the Python object[1], simply by hitting `<TAB>` button after dot (.), list of possible methods, attributes will be pop out, select any of them to get the execution results.

[1] The valid common attributes and element specific ones, as well as the valid methods attched to element is detailed in *this page*.

CHAPTER

FIVE

# API REFERENCES

## 5.1 API References (operation)

`class MachinePortal`(*machine=None*, *segment=None*, ***kws*)

Very first step to control the machine on physics high-level layer, create lattice object from segment of machine, upon which various operations could be preceded.

**Parameters**

- **machine** (`str`) – Name of the accelerator machine, typically, use one folder of the machine name to host all the related configuration files, also could be the path of the configuration folder, "FRIB" by default.
- **segment** (`str`) – All machine segments are defined in *segments* field in the configuration file "phantasy.ini", separated by space, e.g. `segments: LINAC LS1` defines two segments `LINAC` and `LS1`, the `default_segment` field in that file is used to define the default segment to use. If *segment* parameter is not defined, use the one defined by `default_segment`.

**Keyword Arguments** **verbose** (`int`) – If set nonzero, print out verbose message.

---

**Note:**

1. Lattice if created from segment of machine.
2. Directory searching rule for the machine configuration files: (list by searching priority)
   - User-defined directory;
   - Current working directory;
   - Environmental variable: `PHANTASY_CONFIG_DIR`;
   - Current user's home folder: `~/.phantasy`;

   If the found directory is `MPATH`, then the naming rule of *machine*: `MPATH` + machine name, e.g. `MPATH=/home/user/develop`, machine name is `FRIB`, then *machine* could be defined as: `/home/user/develop/FRIB`, all configuration files of `FRIB` should be in that directory.

---

### Examples

```
>>> # Use default configuration
>>> mp = MachinePortal()
>>> mp.last_machine_path # where I put FRIB machine
/home/tong1/work/FRIB/projects/machines/FRIB
>>> # with machine name, just the same as above
>>> mp = MachinePortal(machine="FRIB")
>>> # another machine name
>>> mp = MachinePortal(machine="FRIB1")
# FRIB1 is in the current working directory
>>> os.path.relpath(mp.last_machine_path)
'FRIB1'
```

`get_all_names`(*latname=None*, *virtual=False*, ***kws*)

Get names of all elements from given lattice.

**Parameters**

- **latname** (`str`) – Name of lattice to be investigated.
- **virtual** (`True or False`) – Return virtual elements or not, `False` by default.

**Returns** **ret** – List of element names.

**Return type** List(str)

**See also:**

`lattice_names()` Names of all loaded lattices.

`get_all_types()` Get all element types from given lattice.

**get_all_segment_names**()

Return all available segment names.

**get_all_types**(*latname=None*, *virtual=False*, ***kws*)

Get names of element types (groups/families) from given lattice.

**Parameters**

- **latname** (str) – Name of lattice to be investigated.
- **virtual** (`True or False`) – Return virtual group or not, `False` by default.

**Returns** **ret** – List of type names.

**Return type** List(str)

**See also:**

`lattice_names()` Names of all loaded lattices.

`get_all_names()` Get all element names from given lattice.

**get_elements**(*latname=None*, *name=None*, *type=None*, *srange=None*, ***kws*)

Get element(s) from working lattice.

**Parameters**

- **latname** (str) – Use the (valid) defined lattice name instead of current working lattice name, maybe useful to inspect non-working lattices.
- **name** (`str or list[str]`) – (List of) Element names or Unix shell style patterns.
- **type** (`str or list[str]`) – (List of) Element type/group/family, or Unix shell style patterns.
- **srange** (tuple) – Start and end points (tuple of float) of elements' longitudinal position.

**Keyword Arguments** **sort_key** (str) – Ascendingly sort key of the returned list, `name` or `pos`, `pos` by default, or other attributes valid for `CaElement`.

**Returns** **ret** – List of elements (`CaElement`), excluding virtual elements.

**Return type** List

**Note:**

1. The pattern here used is Unix shell style, slightly different from regex, e.g. pattern 'BP' matches 'BPM' in regex, but matches nothing in Unix shell style, 'BP*' works;
2. If more than one positional parameters (*name*, *type*, *srange*) are defined, return elements that meet all definitions;
3. By default, the returned elements are ascendingly sorted according to element position values.

## Examples

Create MachinePortal instance, e.g. mp 1. Define *name* with an invalid name:

```
>>> mp.get_elements(name='NOEXISTS')
[]
```

2. Define *name* with name or name patterns:

```
>>> mp.get_elements(name='FS1_BMS:DCV_D2662')
[FS1_BMS:DCV_D2662 [VCOR] @ sb=153.794690]
>>> mp.get_elements(name=['FS1_B?*D266?', 'LS1_B*DCV*'], latname='LINAC')
[LS1_BTS:DCV_D1937 [VCOR] @ sb=81.365954,
 LS1_BTS:DCV_D1964 [VCOR] @ sb=84.013954,
 LS1_BTS:DCV_D1997 [VCOR] @ sb=87.348954,
 LS1_BTS:DCV_D2024 [VCOR] @ sb=90.055166,
 LS1_BTS:DCV_D2061 [VCOR] @ sb=93.710487,
 LS1_BTS:DCV_D2114 [VCOR] @ sb=98.985556,
 FS1_BMS:DCV_D2662 [VCOR] @ sb=153.794690,
 FS1_BMS:DCH_D2662 [HCOR] @ sb=153.794690,
 FS1_BMS:BPM_D2664 [BPM] @ sb=153.963690,
 FS1_BMS:QH_D2666 [QUAD] @ sb=154.144690]
```

3. Filter BPMs from the above result:

```
>>> mp.get_elements(name=['FS1_B?*D266?', 'LS1_B*DCV*'], type='BPM',
>>>                 latname='LINAC')
[FS1_BMS:BPM_D2664 [BPM] @ sb=153.963690]
>>> # type='BPM' also could be be pattern
```

4. Filter hybrid types:

```
>>> mp.get_elements(name=['FS1_B?*D266?', 'LS1_B*DCV*'],
>>>                 type=['BPM', 'QUAD'], latname='LINAC')
[FS1_BMS:BPM_D2664 [BPM] @ sb=153.963690,
 FS1_BMS:QH_D2666 [QUAD] @ sb=154.144690]
```

5. Get subsection from lattice according to s-position range:

```
>>> mp.get_elements(srange=(10, 11))
[LS1_CB01:CAV1_D1229 [CAV] @ sb=10.366596,
 LS1_CB01:BPM_D1231 [BPM] @ sb=10.762191,
 LS1_CB01:SOL1_D1235 [SOL] @ sb=10.894207]
```

6. Continue filter with *srange* parameter

```
>>> mp.get_elements(name=['FS1_B?*D266?', 'LS1_B*DCV*'],
>>>                 type=['BPM', 'QUAD'], srange=(154, 155),
>>>                 latname='LINAC')
[FS1_BMS:QH_D2666 [QUAD] @ sb=154.144690]
```

**Note:** Select subsection by `srange` parameter is realized by new approach, other than `~phantasy.library.Lattice.getLine()`, e.g. the result of `getLine((10,11))` contains element before the start range: i.e. `LS1_WA03:PM_D1223:PM @ sb=9.929284`, which is beyond the range.

**See also:**

`get_virtual_elements()` Get virtual elements.

`next_elements()` Get neighborhood of reference element.

`CaElement` Element class.

**static get_pv_names**(*elem*, *field=None*, ***kws*)
Get PV names by given fields for defined elements.

**Parameters**

- **elem** – (List of) CaElement objects.
- **field** (`str or List(str)`) – (List of) Field name of PV, if list of names is defined, only names shared by all elements are valid; if not defined, all shared fields will be used.

**Keyword Arguments handle** (`str`) – Handle of pv, 'readback' (default), 'setpoint' or 'readset'.

**Returns ret** – dict of PV names, with keys of field names.

**Return type** dict

## Examples

1. Get all BPM and PM elements:

```
>>> elem = mp.get_elements(type='*PM')
```

2. Get all pv names with same field:

```
>>> pv1 = mp.get_pv_names(elem) # {'X':[...], 'Y':[...]}
```

3. Get define field(s):

```
>>> pv2 = mp.get_pv_names(elem, 'X')
>>> pv2 = mp.get_pv_names(elem, ['X'])
```

4. Get all PV names from one elements:

```
>>> pv3 = mp.get_pv_names(elem[0])
>>> # return value example:
{u'ENG': [u'V_1:LS1_CA01:BPM_D1129:ENG_RD'],
 u'PHA': [u'V_1:LS1_CA01:BPM_D1129:PHA_RD'],
 u'X': [u'V_1:LS1_CA01:BPM_D1129:X_RD'],
 u'Y': [u'V_1:LS1_CA01:BPM_D1129:Y_RD']}
```

**See also:**

`get_pv_values()` Get PV values.

`get_readback()` Get PV readbacks.

`CaElement` Element class.

**static get_pv_values**(*elem*, *field=None*, ***kws*)
Get PV readback values by given fields for defined elements.

**Parameters**

- **elem** – (List of) CaElement objects.
- **field** (str or List(str)) – (List of) Field name of PV, if list of names is defined, only names shared by all elements are valid; if not defined, all shared fields will be used.

**Returns** **ret** – dict of PV readback values, with keys of field names.

**Return type** dict

## Examples

```
>>> # get all BPM and PM elements
>>> elem = mp.get_elements(type='*PM')
>>> # get 'X' and 'Y' pv readback values
>>> data = mp.get_pv_values(elem, ['X','Y'])
>>> data.keys()
['Y', 'X']
```

**See also:**

`get_pv_names()` Get PV names.

`get_readback()` Get PV readbacks.

**get_virtual_elements**(***kws*)

Get all virtual elements from given lattice.

**Keyword Arguments** **latname** (str) – Name of lattice to be investigated.

**Returns** **ret** – List of virtual `CaElement` objects.

**Return type** List

**inspect_mconf**(*mconf=None*, *out=None*)

Inspect given machine configuration.

**Parameters**

- **mconf** – Machine configuration object, if not given, inspect the last loaded machine.
- **out** – Output stream, if not given, only return results, or besides returning results, also print into defined stream, could be `stdout`, `file``(valid file object)`, `or``StringIO`.

**Returns**

**ret** – Inspection results, retur a dict when *out* is None, or StringIO object when *out* is `sio`, or None; keys of dict:

- `path` : (str), phantasy.ini fullpath
- `lattices` : (list), all defined lattices
- `machine` : (str), defined machine name
- `config` : (dict), all configurations

**Return type** dict or StringIO or None

## Examples

```
>>> mconf = mp.inspect_mconf()
>>> # write inspection results into file
>>> with open('fileout.dat', 'w') as f:
>>>     mconf = mp.inspect_mconf(out=f)
>>> # out could be StringIO or sys.stdout or 'stdout'.
```

**See also:**

`Configuration`

**`static is_virtual`**(*elem*)
Test if input element is virtual element.

**Parameters** **`elem`** – `CaElement` object.

**Returns** **ret** – True for virtual element, else False.

**Return type** True or False

**`last_lattice_conf`**
*list* – Configuration of last loaded lattice, composed of caElements.

**See also:**

`CaElement`

**`last_lattice_name`**
*str* – Name of last loaded lattice.

**`last_machine_conf`**
*Configuration* – Last loaded configuration object.

**See also:**

`Configuration`

**`last_machine_name`**
*str* – Name of last loaded machine.

**`last_machine_path`**
*str* – Full path of the last loaded *phantasys.ini* file

**`lattice_names`**
*list* – Names of all loaded lattices.

**`lattices`**
*dict* – All loaded lattices.

**`load_lattice`**(*segment=None*, *machine=None*, ***kws*)
Load machine segment from *phantasy.ini* file.

**Parameters**

- **`segment`** (`str`) – Segment name.
- **`machine`** (`str`) – Machine name, or path of configuration files.

**Keyword Arguments**

- **`verbose`** (`int`) – If set nonzero, print out verbose message.
- **`re_load`** (`True or False`) – If set True, reload segment, `False` by default.

**Returns** **ret** – Configuration of loaded segment of machine, with the keys of `lat_name`, `lat_conf`, `mach_name`, `mach_path` and `mach_conf`.

**Return type** dict

### Examples

```
>>> mp.load_lattice('LS1')
>>> mp.work_lattice_name # 'LS1'
>>> mp.load_lattice('LINAC') # does not actually load, use cached
>>> mp.work_lattice_name # 'LINAC'
# Note working lattice is changed from 'LS1' to 'LINAC', although
# not actually loaded the lattice, see use_lattice().
```

The cached tricky could improve performance, e.g. in ipython:

```
>>> %%timeit
>>> mp.load_lattice('LS1', re_load=True)
10 loops, best of 3: 180 ms per loop
>>> %%timeit
>>> mp.load_lattice('LS1')
10000 loops, best of 3: 190 µs per loop
```

---

**Note:**

1. If *segment* of *machine* has already been loaded, will not load again, just switch working lattice to be the *segment*; keyword parameter *re_load* could be set to force reload;
2. *re_load* could be used if necessary, e.g. the configuration file for some segment is changed, etc.

---

**See also:**

`use_lattice()` Choose working lattice from loaded lattices.

`reload_lattice()` Reload machine/lattice.

**`machine_names`**
*list* – Names of all loaded machines or facilities.

**`machines`**
*dict* – All loaded machines.

**`next_elements`**(*ref_elem*, *count=1*, ***kws*)
Get elements w.r.t reference element, according to the defined confinement, from given lattice name, if not given *latname*, use the current working lattice.

**Parameters**

- **ref_elem** – `CaElement` object, reference element.
- **count** (`int`) – Skip element number after *ref_elem*, negative input means before, e.g. `count=1` will locate the next one of *ref_elem* in the investigated lattice, if keyword parameter *type* is given, will locate the next one element of the defined type; `count=-1` will locate in the opposite direction.

**Keyword Arguments**

- **type** (`str or list(str)`) – (List of) Element type/group/family, if *type* is a list of more than one element types, the *next* parameter will apply on each type.
- **range** (`str`) – String of format `start:stop:step`, to slicing the output list, e.g. return 50 BPMs after *ref_elem* (`count=50`), but only get every two elements, simply by setting `range=0::2`.

- **latname** (`str`) – Name of lattice to be investigated.
- **ref_include** (`True or False`) – Include *ref_elem* in the returned list or not, False by default.

**Returns** **ret** – List of next located elements, ascendingly sorted by position, by default, only return one element (for eath *type*) that meets the confinement, return more by assgining *range* keyword parameter.

**Return type** List

## Examples

Create MachinePortal instance, e.g. mp

1. Select an element as reference element:

```
>>> print(all_e)
[LS1_CA01:CAV1_D1127 [CAV] @ sb=0.207064,
 LS1_CA01:BPM_D1129 [BPM] @ sb=0.511327,
 LS1_CA01:SOL1_D1131 [SOL] @ sb=0.643330,
 LS1_CA01:DCV_D1131 [VCOR] @ sb=0.743330,
 LS1_CA01:DCH_D1131 [HCOR] @ sb=0.743330,
 LS1_CA01:CAV2_D1135 [CAV] @ sb=0.986724,
 LS1_CA01:CAV3_D1143 [CAV] @ sb=1.766370,
 LS1_CA01:BPM_D1144 [BPM] @ sb=2.070634,
 LS1_CA01:SOL2_D1147 [SOL] @ sb=2.202637,
 LS1_CA01:DCV_D1147 [VCOR] @ sb=2.302637,
 LS1_CA01:DCH_D1147 [HCOR] @ sb=2.302637,
 LS1_CA01:CAV4_D1150 [CAV] @ sb=2.546031,
 LS1_WA01:BPM_D1155 [BPM] @ sb=3.109095,
 LS1_CA02:CAV1_D1161 [CAV] @ sb=3.580158,
 LS1_CA02:BPM_D1163 [BPM] @ sb=3.884422,
 LS1_CA02:SOL1_D1165 [SOL] @ sb=4.016425,
 LS1_CA02:DCV_D1165 [VCOR] @ sb=4.116425,
 LS1_CA02:DCH_D1165 [HCOR] @ sb=4.116425,
 LS1_CA02:CAV2_D1169 [CAV] @ sb=4.359819,
 LS1_CA02:CAV3_D1176 [CAV] @ sb=5.139465,
 LS1_CA02:BPM_D1178 [BPM] @ sb=5.443728]
>>> ref_elem = all_e[5]
```

2. Get next element of *ref_elem*:

```
>>> mp.next_elements(ref_elem)
[LS1_CA01:CAV3_D1143 [CAV] @ sb=1.766370]
```

3. Get last of the next two element:

```
>>> mp.next_elements(ref_elem, count=2)
[LS1_CA01:BPM_D1144 [BPM] @ sb=2.070634]
```

4. Get all of the next two elements:

```
>>> mp.next_elements(ref_elem, count=2, range='0::1')
[LS1_CA01:CAV3_D1143 [CAV] @ sb=1.766370,
 LS1_CA01:BPM_D1144 [BPM] @ sb=2.070634]
```

5. Get all of the two elements before *ref_elem*:

```
>>> mp.next_elements(ref_elem, count=-2, range='0::1')
[LS1_CA01:DCV_D1131 [VCOR] @ sb=0.743330,
 LS1_CA01:DCH_D1131 [HCOR] @ sb=0.743330]
```

6. Get next two BPM elements after *ref_elem*, including itself:

```
>>> mp.next_elements(ref_elem, count=2, type=['BPM'],
>>>                  ref_include=True, range='0::1')
[LS1_CA01:CAV2_D1135 [CAV] @ sb=0.986724,
 LS1_CA01:BPM_D1144 [BPM] @ sb=2.070634,
 LS1_WA01:BPM_D1155 [BPM] @ sb=3.109095]
```

7. Get with hybrid types:

```
>>> mp.next_elements(ref_elem, count=2, type=['BPM', 'CAV'],
>>>                  range='0::1')
[LS1_CA01:CAV3_D1143 [CAV] @ sb=1.766370,
 LS1_CA01:BPM_D1144 [BPM] @ sb=2.070634,
 LS1_CA01:CAV4_D1150 [CAV] @ sb=2.546031,
 LS1_WA01:BPM_D1155 [BPM] @ sb=3.109095]
```

**print_all_properties**()
Print all properties, for debug only.

**reload_lattice**(*segment=None*, *machine=None*, ***kws*)
Reload machine lattice, if parameters *machine* and *segment* are not defined, reload last loaded one.

**Parameters**

- **segment** (str) – Name of segment of machine.
- **machine** (str) – Name of machine, or path of configuration files.

**See also:**

`load_lattice()` Load machine/lattice from configuration files.

**use_lattice**(*lattice_name=None*)
Choose name of one of the loaded lattices as the working lattice, if this method is not evoked, the working lattice is the last loaded lattice.

**Parameters** **lattice_name** (str) – Lattice name.

**Returns** **ret** – Selected working lattice name.

**Return type** str

**Note:** If `load_lattice()` or `reload_lattice()` is called, the working lattice name would be changed to the just loaded one, since, usually as the user loading a lattice, most likely he/she would like to switch onto that lattice, or explicitly invoking `use_lattice()` again to switch to another one.

**See also:**

`load_lattice()` Load machine/lattice from configuration files.

**work_lattice_conf**
*list* – Configuration of working lattice, composed of CaElements.

**See also:**

`CaElement` Element object for channel access.

`use_lattice()` Choose working lattice from loaded lattices.

**`work_lattice_name`**
*str* – Name of working lattice.

## 5.2 API References (lattice)

`class CaElement(**kws)`

Element with Channel Access support.

This class could be used to create an element with data from Channel Finder Service or input keyword parameters.

**Parameters** **`enable`** (`True or False`) – Element is enabled or not, `True` is controllable, default is True.

**Keyword Arguments**

- **`name`** (`str`) – Element name.
- **`family`** (`str`) – Element type.
- **`index`** (`int`) – Element index, default sort key, otherwise, `sb` is used as sort key.
- **`length`** (`float`) – Effective element length.
- **`sb`** (`float`) – Longitudinal position at the beginning point, unit: *m*.
- **`se`** (`float`) – Longitudinal position at the end point, unit: *m*.
- **`virtual`** (`bool`) – pass
- **`tags`** (`dict`) – Tags for each PV as key name and set of strings as tag names.
- **`fields`** (`dict`) – pass
- **`enable`** (`True or False`) – Element is enabled or not, `True` is controllable, default is True.
- **`pv_data`** (`list or dict`) – PV record data to build an element, should be of a list of: `string of PV name, dict of properties, list of tags`, or with dict of keys of: `pv_name`, `pv_props` and `pv_tags`.

**Note:** If *pv_data* is defined, element will be initialized with data from *pv_data*, if *pv_data* is CFS formatted, `simplify_data` should be used first to convert data structure.

**See also:**

**`simplify_data()`** Convert CFS formatted data into simple tuple.

**`phantasy.library.pv.datasource.DataSource`** PV data source.

**`design_settings`**

*dict* – Physics design setting(s) for all dynamic field(s).

**`fields`**

*list* – Valid Channel Access field names.

**Warning:** *fields* can only accept dict, with CA field name as key and `CaField` as value.

**`get_eng_fields()`**

Return list of all engineering fields.

**`get_field`**(*field*)

Get element field of CA support.

**Parameters** **`field`** (`str`) – Field name.

---

**Note:** All valid field names could be retrieved by `fields` attribute.

---

**Returns** CaField or None.

**Return type** ret

**`get_phy_fields`**()

Return list of all physics fields.

---

**Note:** If returned list is empty, but *get_eng_fields* is not, then physics fields should be the same as engineering fields, but only appeared as ENG field type.

---

**`last_settings`**

*dict* – Last setting(s) for all dynamic field(s).

**`process_pv`**(*pv_name*, *pv_props*, *pv_tags=None*, *u_policy=None*)

Process PV record to update element with properties and tags.

**Parameters**

- **`pv_name`** (`str`) – PV name.
- **`pv_props`** (`dict`) – PV properties, key-value pairs.
- **`pv_tags`** (`list`) – PV tag list.

**`pv`**(*field=None*, *handle=None*, ***kws*)

Get PV names with defined *field* and *handle*, if none of them is defined, return all PV names.

**Parameters**

- **`field`** (`str or list`) – Channel access field name, all available fields will be used if not defined, ignore invalid field.
- **`handle`** (`str`) – Channel access protocol type, could be one of *readback*, *readset* and *setpoint*.

---

**Note:**

1. All Valid field names could be retrieved by `fields` attribute.
2. If more than one field is defined with *field*, i.e. *field* is a list of string, return PV names binding with these fields, may apply *handle* as a filter, e.g. if *handle* is not defined, return all PVs.

---

**Returns** **ret** – List of valid PVs as request.

**Return type** list

**`set_field`**(*field*, *pv*, *handle=None*, ***kws*)

Set element field with CA support.

**Parameters**

- **field** (str) – Field name.
- **pv** (str) – Valid PV name.
- **handle** (str) – PV channel type, valid options: `readback`, `readset`, `setpoint`, default is `readback`.

**Keyword Arguments**

- **pv_policy** (dict) – Name of PV read/write policy, keys: 'read' and 'write', values: scaling law function object.
- **ftype** (str) – Field type, 'ENG' (default) or 'PHY'.

**tags**
*dict* – Tags that element PVs have been assigned.

**update_groups**(*props*, ***kws*)
Update new group with *family* name.

**Parameters** **props** (dict) – Element properties.

**See also:**

`update_properties()`

**update_properties**(*props*, ***kws*)
Update element properties.

**Parameters** **props** (dict) – Dictionary of properties, two categories:

- static properties: without CA features: *name*, *family*, *index*, *se*, *sb* (optional), *length*
- dynamic properties: with CA features: *handle*, *field*

**Keyword Arguments** **pv** (str) – Valid PV name.

**update_tags**(*tags*, ***kws*)
Update element tags.

**Parameters** **tags** (list) – List of tags.

**Keyword Arguments** **pv** (str) – Valid PV name.

**virtual**
*bool* – Virtual element or not.

**class CaField**(*name=''*, ***kws*)
Channel Access support for element field.

Usually, `CaField` instance has at most three types of PV names: *readback*, *readset* and *setpoint*, each PV should linked with valid PV connections.

There are two approaches to retrieve the PV values, one is through `value` attribute, another one is by explicitly calling `get()`;

The same rule applies to set values, i.e. by setting `value` attribute and by calling `put()` method.

**Parameters**

- **name** (str) – Name of CA field, usually represents the physics attribute linked with CA.
- **wait** (bool) – If True, wait until put operation completed, default is True.
- **timeout** (float) – Time out in second for put operation, default is 10 seconds.
- **ename** (str) – Name of element which the field attaches to.

**Keyword Arguments**

- **readback** (str, list(str)) – Readback PV name(s), if a single string is defined, append operation will be issued, if list or tuple of strings is defined, *readback* attribute will be overwritten with the new list, the same rule applies to *readset* and *setpoint*, as well as *readback_pv*, *readset_pv* and *setpoint_pv*.
- **readset** (str, list(str)) – Readset PV name(s).
- **setpoint** (str, list(str)) – Setpoint PV name(s).

**ename**

*str* – Name of element the field attaches to.

**get**(*handle='readback'*, *n_sample=1*, *timeout=10.0*, *with_timestamp=False*, *ts_format='raw'*, *keep_raw=False*, ***kws*)

Get value of PV with specified *handle*, if argument *n_sample* is larger than 1, statistical result with averaged value and standard deviation will be returned; the sample rate depends on the device scan rate. If *timeout* is defined, DAQ will be inactivated after *timeout* second.

**Warning:** For the case of devices that generate constant readings, if *n_sample* is larger than 1, set *timeout* parameter with a reasonable value is required, or this method will hang up your program.

**Parameters**

- **handle** (str) – PV handle, 'readback', 'readset' or 'setpoint'.
- **n_sample** (int) – Sample number, total DAQ count.
- **timeout** (float) – Timeout in second for the whole DAQ process, set *None* to wait forever.
- **with_timestamp** (bool) – If *True*, return timestamp, default value is *False*.
- **ts_format** (str) – Format for timestamp, valid options: *raw*, *epoch* and *human*.
- **keep_raw** (bool) – If *True*, return raw read data as well.

**Returns** **r** – Valid keys: 'mean', 'std', 'timestamp', 'data', the values are the average of all shots, the standard deviation of all shots, the timestamp of the first shot, and all the aquired data array.

**Return type** dict

## Examples

Get CA field instance from element:

```
>>> from phantasy import MachinePortal
>>> mp = MachinePortal(machine='<mach_name>')
>>> lat = mp.work_lattice_conf
>>> elem = lat[0]
>>> print(elem.fields)
>>> fld = elem.get_field('<field_name>')
```

Get readings from PVs with 'readback' handle

```
>>> print(fld.get('readback'))
```

Get readings from PVs with 'setpoint' handle

```
>>> print(fld.get('setpoint'))
```

Get readings from PVs with 'readset' handle

```
>>> print(fld.get('readset'))
```

Get readings with timestamp

```
>>> fld.get('readback', with_timestamp=True)
```

Define the style of timestamp

```
>>> fld.get('readback', with_timestamp=True, ts_format='human')
```

Get statistical readings, and keep all the raw data

```
>>> fld.get('readback', n_sample=1, with_timestamp=True,
            ts_format="epoch", keep_raw=True)
```

**init_pvs**(***kws*)
PV initialization.

**is_engineering_field**()
Test if *field* is engineering field.

**is_physics_field**()
Test if *field* is physics field.

**name**
*str* – Field name, empty string if not given.

**pvs**()
Return dict of valid pv type and names.

**read_policy**
Defined read policy, i.e. how to read value from field.

The defined policy is a function, with *readback_pv* attribute as argument, return a value.

**readback**
*list[str]* – Readback PV name, usually ends with *_RD*.

**readback_pv**
*PV* – Readback PV object.

**readset**
*list[str]* – Readset PV name, usually ends with *_RSET*.

**readset_pv**
*PV* – Readset PV object.

**reset_policy**(*policy=None*)
Reset policy, by policy name.

**set**(*value*, *handle='setpoint'*, ***kws*)
Set value(s) of PV(s) with specified *handle*.

**Parameters**

- **value** (`list` or `list(val)`) – New value(s) to be set.
- **handle** (`str`) – PV handle, 'readback', 'readset' or 'setpoint'.

### Examples

Get CA field instance, see *get()*, set one field of an element which has two 'setpoint' PVs, e.g. quad:

```
>>> fld.set([val1, val2], 'setpoint')
>>> # Check with get:
>>> fld.get('setpoint')
>>> # should return [val1, val2]
```

**Note:** `get()` and `set()` are a pair of methods that can read/write PV(s) bypass field defined read/write policies.

**`setpoint`**
*list[str]* – Setpoint PV name, usually ends with *_CSET*.

**`setpoint_pv`**
*PV* – Setpoint PV object.

**`timeout`**
**float* – Time out in second for put operation, default* – 10 [sec].

**`update`**(***kws*)
Update PV with defined handle.

**`value`**
Get value of PV, returned from CA request.

**`wait`**
*boolean* – Wait (True) for not (False) when issuing set command, default: True.

**`write_policy`**
Defined write policy, i.e. how to set value to field.

The defined policy is a function, with *setpoint_pv* attribute and new value as arguments.

## 5.3 API References (flame_utils)

**class ModelFlame**(*lat_file=None*, ***kws*)

General FLAME modeling class.

**Parameters** **lat_file** (str) – FLAME lattice file, if not set, None.

### Examples

```
>>> from flame import Machine
>>> from flame_utlis import ModelFlame
>>>
>>> latfile = "lattice/test.lat"
>>> fm1 = ModelFlame()
>>> # manually initialization
>>> fm1.latfile = latfile
>>> m = Machine(latfile)
>>> fm1.machine = m
>>> fm1.bmstate = m.allocState({})
>>> # or by explicitly calling:
>>> fm1.machine, fm1.bmstate = fm1.init_machine(latfile)
>>>
>>> # initialize with valid lattice file
>>> fm2 = ModelFlame(lat_file=latfile)
>>>
>>> # (Recommanded) initialize with BeamState
>>> fm = ModelFlame()
>>> bs = BeamState(machine=m)
>>> # now the attributes of ms could be arbitarily altered
>>> fm.bmstate = bs
>>> fm.machine = m
>>>
>>> # run fm
>>> obs = fm.get_index_by_type(type='bpm')['bpm']
>>> r, s = fm.run(monitor=obs)
>>>
>>> # get result, storing as a dict, e.g. data
>>> data = fm.collect_data(r, pos=True, x0=True, y0=True)
```

**See also:**

`BeamState` FLAME beam state class for `MomentMatrix` simulation type.

**bmstate**

*BeamState* – Could be initialized with FLAME internal state or BeamState object.

**See also:**

`BeamState` FLAME beam state class created for `MomentMatrix`.

**clone_machine**()

Clone FLAME Machine object.

**Returns** FLAME Machine object.

**Return type** ret

**static collect_data**(*result*, **args*, ***kws*)

Collect data of interest from propagation results.

**Parameters**

- **result** – Propagation results with `BeamState` object.
- **args** – Names of attribute, separated by comma.

**See also:**

`collect_data()` Get data of interest from results.

`configure`(*econf*)

Configure FLAME model.

**Parameters** **econf** (`(list of) dict`) – Element configuration(s), see `get_element()`.

**See also:**

`configure()` Configure FLAME machine.

`get_element()` Get FLAME lattice element configuration.

---

**Note:** Pass *econf* with a list of dict for applying batch configuring.

---

`static convert_results`(*res*, ***kws*)

Convert all beam states of results generated by `run()` method to be `BeamState` object.

**Parameters** **res** (`list of tuple`) – List of propagation results.

**Returns** Tuple of `(r, s)`, where `r` is list of results at each monitor points, `s` is `BeamState` object after the last monitor point.

**Return type** list of tuple

`get_all_names`()

Get all uniqe element names.

**Returns** **res** – List of element names.

**Return type** list of str

**See also:**

`get_all_names()` Get all uniqe names from a FLAME machine.

`get_all_types`()

Get all uniqe element types.

**Returns** **res** – List of element type names

**Return type** list of str

**See also:**

`get_all_types()` Get all unique types from a FLAME machine.

`get_element`(*name=None*, *index=None*, *type=None*, ***kws*)

Element inspection, get properties.

**Returns** **res** – List of dict of properties or empty list.

**Return type** list of dict

**See also:**

`get_element()` Get element from FLAME machine object.

`get_index_by_name`(*name=", rtype='dict'*)
Get index(s) by name(s).

**Parameters**

- **name** (`list or tuple of str`) – Single element name or list[tuple] of element names.
- **rtype** (`str`) – Return type, 'dict' (default) or 'list'.

**Returns** Dict of element indices, key is name, value is index, list of element indices list.

**Return type** dict or list

**See also:**

`get_index_by_name()` Get index(s) by element name(s).

`get_index_by_type`(*type=", rtype='dict'*)
Get element(s) index by type(s).

**Parameters**

- **type** (`str or list of str`) – Single element type name or list[tuple] of element type names.
- **rtype** (`str`) – Return type, 'dict' (default) or 'list'.

**Returns** **ind** – Dict, key is type name, value if indice list of each type name, list, of indices list, with the order of type.

**Return type** dict or list

**See also:**

`get_index_by_type()` Get element(s) index by type(s).

`static init_machine`(*latfile*)
Initialize FLAME machine.

**Parameters** **latfile** – FLAME lattice file.

**Returns** **res** – Tuple of `(m, s)`, where `m` is FLAME machine instance, and `s` is initial machine states.

**Return type** tuple

`insert_element`(*index=None, element=None, econf=None*)
Insert new element to the machine.

**Parameters**

- **econf** (`dict`) – Element configuration (see `get_element()`).
- **index** (`int or str`) – Insert element before the index (or element name).
- **element** (`dict`) – Lattice element dictionary.

**Note:** User must input 'econf' or 'index and element'. If econf is defined, insert econf['properties'] element before econf['index'].

**inspect_lattice**()
Inspect FLAME machine and print out information.

**See also:**

`inspect_lattice()` Inspect FLAME lattice file, print a brief report.

**latfile**
*str* – FLAME lattice file name.

**machine**
FLAME machine object.

**run**(*bmstate=None*, *from_element=None*, *to_element=None*, *monitor=None*)
Simulate model.

**Parameters**

- **bmstate** – FLAME beam state object, also could be `BeamState` object, if not set, will use the one from `ModelFlame` object itself, usually is created at the initialization stage, see `init_machine()`.
- **from_element** (`int`) – Element index of start point, if not set, will be the first element if not set, will be 0 for zero states, or 1.
- **to_element** (`int`) – Element index of end point, if not set, will be the last element.
- **monitor** (`list[int]`) – List of element indice selected as states monitors, if set -1, will be a list of only last element.

**Returns** Tuple of `(r, s)`, where `r` is list of results at each monitor points, `s` is `BeamState` object after the last monitor point.

**Return type** tuple

**Warning:** This method does not change the input *bmstate*, while `propagate` changes.

**See also:**

`BeamState()` FLAME BeamState class created for `MomentMatrix` type.

`propagate()` Propagate `BeamState` object for FLAME machine object.

**class BeamState**(*s=None*, ***kws*)
FLAME beam state, from which simulated results could be retrieved.

All attributes of states:

- `pos`,
- `ref_beta`, `ref_bg`, `ref_gamma`, `ref_IonEk`, `ref_IonEs`, `ref_IonQ`, `ref_IonW`, `ref_IonZ`, `ref_phis`, `ref_SampleIonK`,
- `beta`, `bg`, `gamma`, `IonEk`, `IonEs`, `IonQ`, `IonW`, `IonZ`, `phis`, `SampleIonK`,
- `moment0` (`cenvector_all`), `moment0_rms` (`rmsvector`), `moment0_env` (`cenvector`),
- `moment1` (`beammatrix_all`), `moment1_env` (`beammatrix`),
- `dEk0` (`dEkcen_all`), `dEk0_env` (`dEkcen`), `dEk0_rms` (`dEkrms`), `dEkrms_all`
- `phi0` (`phicen_all`), `phi0_env` (`phicen`), `phi0_rms` (`phirms`), `phirms_all`

- `x0` (`xcen_all`), `x0_env` (`xcen`), `x0_rms` (`xrms`), `xrms_all`,
- `xp0` (`xpcen_all`), `xp0_env` (`xpcen`), `xp0_rms` (`xprms`), `xprms_all`,
- `y0` (`ycen_all`), `y0_env` (`ycen`), `y0_rms` (`yrms`), `yrms_all`,
- `yp0` (`ypcen_all`), `yp0_env` (`ypcen`), `yp0_rms` (`yprms`), `yprms_all`,
- `last_caviphi0` (since version 1.1.1)

**Warning:**

1. These attributes are only valid for the case of `sim_type` being defined as `MomentMatrix`, which is de facto the exclusive option used at FRIB.
2. If the attribute is an array, new array value should be assigned instead of by element indexing way, e.g.

```
>>> bs = BeamState(s)
>>> print(bs.moment0)
array([[ -7.88600000e-04],
       [  1.08371000e-05],
       [  1.33734000e-02],
       [  6.67853000e-06],
       [ -1.84773000e-04],
       [  3.09995000e-04],
       [  1.00000000e+00]])
>>> # the right way to just change the first element of the array
>>> m_tmp = bs.moment0
>>> m_tmp[0] = 0
>>> bs.moment0 = m_tmp
>>> print(bs.moment0)
array([[  0.00000000e+00],
       [  1.08371000e-05],
       [  1.33734000e-02],
       [  6.67853000e-06],
       [ -1.84773000e-04],
       [  3.09995000e-04],
       [  1.00000000e+00]])
>>> # while this way does not work: ms.moment0[0] = 0
```

**Parameters** **s** – FLAME state object, created by *allocState()*.

**Keyword Arguments**

- **bmstate** – BeamState object, priority: high
- **machine** – FLAME machine object, priority: middle
- **latfile** – FLAME lattice file name, priority: low

**Note:** If more than one keyword parameters are provided, the selection policy follows the priority from high to low.

**Warning:** If only `s` is assigned with all-zeros states (usually created by `allocState({})` method), then please note that this state can only propagate from the first element, i.e. `SOURCE` (`from_element` parameter of `run()` or `propagate()` should be 0), or errors happen; the better initialization should be passing one of keyword parameters of `machine` and `latfile` to initialize the state to be significant for the `propagate()` method.

**IonEk**
*Array* – kinetic energy of all charge states, [eV/u]

**IonEs**
*Array* – rest energy of all charge states, [eV/u]

**IonQ**
*Array* – macro particle number of all charge states

**Note:** This is what `NCharge` means in the FLAME lattice file.

**IonW**
*Array* – total energy of all charge states, [eV/u], i.e. $W = E_s + E_k$

**IonZ**
*Array* – all charge states, measured by charge to mass ratio

**Note:** This is what `IonChargeStates` means in the FLAME lattice file.

**SampleIonK**
*Array* – wave-vector in cavities with different beta values of all charge states

**beammatrix**
*Array* – averaged correlation tensor of all charge states

**beammatrix_all**
*Array* – correlation tensor of all charge states, for each charge state, the correlation matrix could be written as:

$$\begin{array}{ccccccc}
\langle x \cdot x \rangle & \langle x \cdot x' \rangle & \langle x \cdot y \rangle & \langle x \cdot y' \rangle & \langle x \cdot \phi \rangle & \langle x \cdot \delta E_k \rangle & 0 \\
\langle x' \cdot x \rangle & \langle x' \cdot x' \rangle & \langle x' \cdot y \rangle & \langle x' \cdot y' \rangle & \langle x' \cdot \phi \rangle & \langle x' \cdot \delta E_k \rangle & 0 \\
\langle y \cdot x \rangle & \langle y \cdot x' \rangle & \langle y \cdot y \rangle & \langle y \cdot y' \rangle & \langle y \cdot \phi \rangle & \langle y \cdot \delta E_k \rangle & 0 \\
\langle y' \cdot x \rangle & \langle y' \cdot x' \rangle & \langle y' \cdot y \rangle & \langle y' \cdot y' \rangle & \langle y' \cdot \phi \rangle & \langle y' \cdot \delta E_k \rangle & 0 \\
\langle \phi \cdot x \rangle & \langle \phi \cdot x' \rangle & \langle \phi \cdot y \rangle & \langle \phi \cdot y' \rangle & \langle \phi \cdot \phi \rangle & \langle \phi \cdot \delta E_k \rangle & 0 \\
\langle \delta E_k \cdot x \rangle & \langle \delta E_k \cdot x' \rangle & \langle \delta E_k \cdot y \rangle & \langle \delta E_k \cdot y' \rangle & \langle \delta E_k \cdot \phi \rangle & \langle \delta E_k \cdot \delta E_k \rangle & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

**beta**
*Array* – speed in the unit of light velocity in vacuum of all charge states, Lorentz beta

**bg**
*Array* – multiplication of beta and gamma of all charge states

**cenvector**
*Array* – weight average of centroid for all charge states, array of `[x, x', y, y', phi, dEk, 1]`, with the units of `[mm, rad, mm, rad, rad, MeV/u, 1]`.

**Note:** The physics meanings for each column are:

- `x`: x position in transverse plane;
- `x'`: x divergence;
- `y`: y position in transverse plane;
- `y'`: y divergence;
- `phi`: longitudinal beam length, measured in RF frequency;

- `dEk`: kinetic energy deviation w.r.t. reference charge state;
- `1`: should be always 1, for the convenience of handling corrector (i.e. `orbtrim` element)

---

**`cenvector_all`**
*Array* – centroid for all charge states, array of `[x, x', y, y', phi, dEk, 1]`

**`clone`**`()`
Return a copy of Beamstate object.

**`dEk0`**
*Array* – kinetic energy deviation w.r.t. reference charge state, for all charge states, [MeV/u]

**`dEk0_env`**
*Array* – weight average of all charge states for $\delta E_k$, [MeV/u]

**`dEk0_rms`**
*float* – general rms beam envelope for $\delta E_k$, [MeV/u]

**`dEkcen`**
*Array* – weight average of all charge states for $\delta E_k$, [MeV/u]

**`dEkcen_all`**
*Array* – kinetic energy deviation w.r.t. reference charge state, for all charge states, [MeV/u]

**`dEkrms`**
*float* – general rms beam envelope for $\delta E_k$, [MeV/u]

**`dEkrms_all`**
*Array* – general rms beam envelope for $\delta E_k$ of all charge states, [MeV/u]

**`gamma`**
*Array* – relativistic energy of all charge states, Lorentz gamma

**`last_caviphi0`**
*float* – Last RF cavity's driven phase, [deg]

**`moment0`**
*Array* – centroid for all charge states, array of `[x, x', y, y', phi, dEk, 1]`

**`moment0_env`**
*Array* – weight average of centroid for all charge states, array of `[x, x', y, y', phi, dEk, 1]`, with the units of `[mm, rad, mm, rad, rad, MeV/u, 1]`.

---

**Note:** The physics meanings for each column are:

- `x`: x position in transverse plane;
- `x'`: x divergence;
- `y`: y position in transverse plane;
- `y'`: y divergence;
- `phi`: longitudinal beam length, measured in RF frequency;
- `dEk`: kinetic energy deviation w.r.t. reference charge state;
- `1`: should be always 1, for the convenience of handling corrector (i.e. `orbtrim` element)

---

**`moment0_rms`**
*Array* – rms beam envelope, part of statistical results from `moment1`.

**Note:** The square of `moment0_rms` should be equal to the diagonal elements of `moment1`.

**See also:**

`moment1` correlation tensor of all charge states

**moment1**
*Array* – correlation tensor of all charge states, for each charge state, the correlation matrix could be written as:

$$\begin{array}{ccccccc}
\langle x \cdot x\rangle & \langle x \cdot x'\rangle & \langle x \cdot y\rangle & \langle x \cdot y'\rangle & \langle x \cdot \phi\rangle & \langle x \cdot \delta E_k\rangle & 0 \\
\langle x' \cdot x\rangle & \langle x' \cdot x'\rangle & \langle x' \cdot y\rangle & \langle x' \cdot y'\rangle & \langle x' \cdot \phi\rangle & \langle x' \cdot \delta E_k\rangle & 0 \\
\langle y \cdot x\rangle & \langle y \cdot x'\rangle & \langle y \cdot y\rangle & \langle y \cdot y'\rangle & \langle y \cdot \phi\rangle & \langle y \cdot \delta E_k\rangle & 0 \\
\langle y' \cdot x\rangle & \langle y' \cdot x'\rangle & \langle y' \cdot y\rangle & \langle y' \cdot y'\rangle & \langle y' \cdot \phi\rangle & \langle y' \cdot \delta E_k\rangle & 0 \\
\langle \phi \cdot x\rangle & \langle \phi \cdot x'\rangle & \langle \phi \cdot y\rangle & \langle \phi \cdot y'\rangle & \langle \phi \cdot \phi\rangle & \langle \phi \cdot \delta E_k\rangle & 0 \\
\langle \delta E_k \cdot x\rangle & \langle \delta E_k \cdot x'\rangle & \langle \delta E_k \cdot y\rangle & \langle \delta E_k \cdot y'\rangle & \langle \delta E_k \cdot \phi\rangle & \langle \delta E_k \cdot \delta E_k\rangle & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

**moment1_env**
*Array* – averaged correlation tensor of all charge states

**phi0**
*Array* – longitudinal beam length, measured in RF frequency for all charge states, [rad]

**phi0_env**
*Array* – weight average of all charge states for $\phi$, [mm]

**phi0_rms**
*float* – general rms beam envelope for $\phi$, [mm]

**phicen**
*Array* – weight average of all charge states for $\phi$, [mm]

**phicen_all**
*Array* – longitudinal beam length, measured in RF frequency for all charge states, [rad]

**phirms**
*float* – general rms beam envelope for $\phi$, [mm]

**phirms_all**
*Array* – general rms beam envelope for $\phi$ of all charge states, [mm]

**phis**
*Array* – absolute synchrotron phase of all charge states, [rad]

**pos**
*float* – longitudinally propagating position, [m]

**ref_IonEk**
*float* – kinetic energy of reference charge state, [eV/u]

**ref_IonEs**
*float* – rest energy of reference charge state, [eV/u]

**ref_IonQ**
*int* – macro particle number of reference charge state

**ref_IonW**
*float* – total energy of reference charge state, [eV/u], i.e. $W = E_s + E_k$

**`ref_IonZ`**
*float* – reference charge state, measured by charge to mass ratio, e.g. ${}^{33+}_{238}U : Q[33]/A[238]$

**`ref_SampleIonK`**
*float* – wave-vector in cavities with different beta values of reference charge state

**`ref_beta`**
*float* – speed in the unit of light velocity in vacuum of reference charge state, Lorentz beta

**`ref_bg`**
*float* – multiplication of beta and gamma of reference charge state

**`ref_gamma`**
*float* – relativistic energy of reference charge state, Lorentz gamma

**`ref_phis`**
*float* – absolute synchrotron phase of reference charge state, [rad]

**`rmsvector`**
*Array* – rms beam envelope, part of statistical results from `moment1`.

**Note:** The square of `moment0_rms` should be equal to the diagonal elements of `moment1`.

**See also:**

**`moment1`** correlation tensor of all charge states

**`state`**
*flame._internal.State* – FLAME state object, also could be initialized with BeamState object

**`x0`**
*Array* – x centroid for all charge states, [mm]

**`x0_env`**
*Array* – weight average of all charge states for `x`, [mm]

**`x0_rms`**
*float* – general rms beam envelope for `x`, [mm]

**`xcen`**
*Array* – weight average of all charge states for `x`, [mm]

**`xcen_all`**
*Array* – x centroid for all charge states, [mm]

**`xp0`**
*Array* – x centroid divergence for all charge states, [rad]

**`xp0_env`**
*Array* – weight average of all charge states for `x'`, [rad]

**`xp0_rms`**
*float* – general rms beam envelope for `x'`, [rad]

**`xpcen`**
*Array* – weight average of all charge states for `x'`, [rad]

**`xpcen_all`**
*Array* – x centroid divergence for all charge states, [rad]

**xprms**
: *float* – general rms beam envelope for `x'`, [rad]

**xprms_all**
: *Array* – general rms beam envelope for `x'` of all charge states, [rad]

**xrms**
: *float* – general rms beam envelope for `x`, [mm]

**xrms_all**
: *Array* – general rms beam envelope for `x` of all charge states, [mm]

**y0**
: *Array* – y centroid for all charge states, [mm]

**y0_env**
: *Array* – weight average of all charge states for `y`, [mm]

**y0_rms**
: *float* – general rms beam envelope for `y`, [mm]

**ycen**
: *Array* – weight average of all charge states for `y`, [mm]

**ycen_all**
: *Array* – y centroid for all charge states, [mm]

**yp0**
: *Array* – y centroid divergence for all charge states, [rad]

**yp0_env**
: *Array* – weight average of all charge states for `y'`, [rad]

**yp0_rms**
: *float* – general rms beam envelope for `y'`, [rad]

**ypcen**
: *Array* – weight average of all charge states for `y'`, [rad]

**ypcen_all**
: *Array* – y centroid divergence for all charge states, [rad]

**yprms**
: *float* – general rms beam envelope for `y'`, [rad]

**yprms_all**
: *Array* – general rms beam envelope for `y'` of all charge states, [rad]

**yrms**
: *float* – general rms beam envelope for `y`, [mm]

**yrms_all**
: *Array* – general rms beam envelope for `y` of all charge states, [mm]

CHAPTER

SIX

# GLOSSARY

**machine** Accelerator facility, defined by a string name, should have configuration files (usually hosted in a folder with machine name as folder name) to describe how the machine is structured, including EPICS control related data source, lattice generation rules, etc.

**segment** Part of machine, a collection of sequential devices, could be represented as lattice file if written into file for simulations.

**lattice** Lattice file for simulation, e.g. FLAME lattice ,IMPACT lattice.

**layout** Geometry-related description of the machine/segment devices arrangement.

**settings** Device configurations, e.g. power supply setting value.

**high-level lattice** Instantiated from `phantasy.lattice.Lattice`, which will created a `Lattice` object to represent the real accelerator based on the machine configurations. Concreted devices will be established as high-level elements that allow users to manipulate interactively. High-level lattice is managed by `phantasy.MachinePortal` instance, model-based machine tunning and online tunning approaches are supported for specific high-level lattice.

**viewer element** Element type or family is one of the following: `BPM`, `PM`, etc., which could only be used as readonly devices for diagnostics.

# INDEX