genopt Documentation

Release 0.0.2

Tong Zhang

Nov 15, 2016

CONTENTS

genopt Python package

`genopt`: general multi-dimensional optimization

PDF documentation: `Download`

Github repo: https://github.com/archman/genopt

Author Tong Zhang

E-mail zhangt@frib.msu.edu

Date 2016

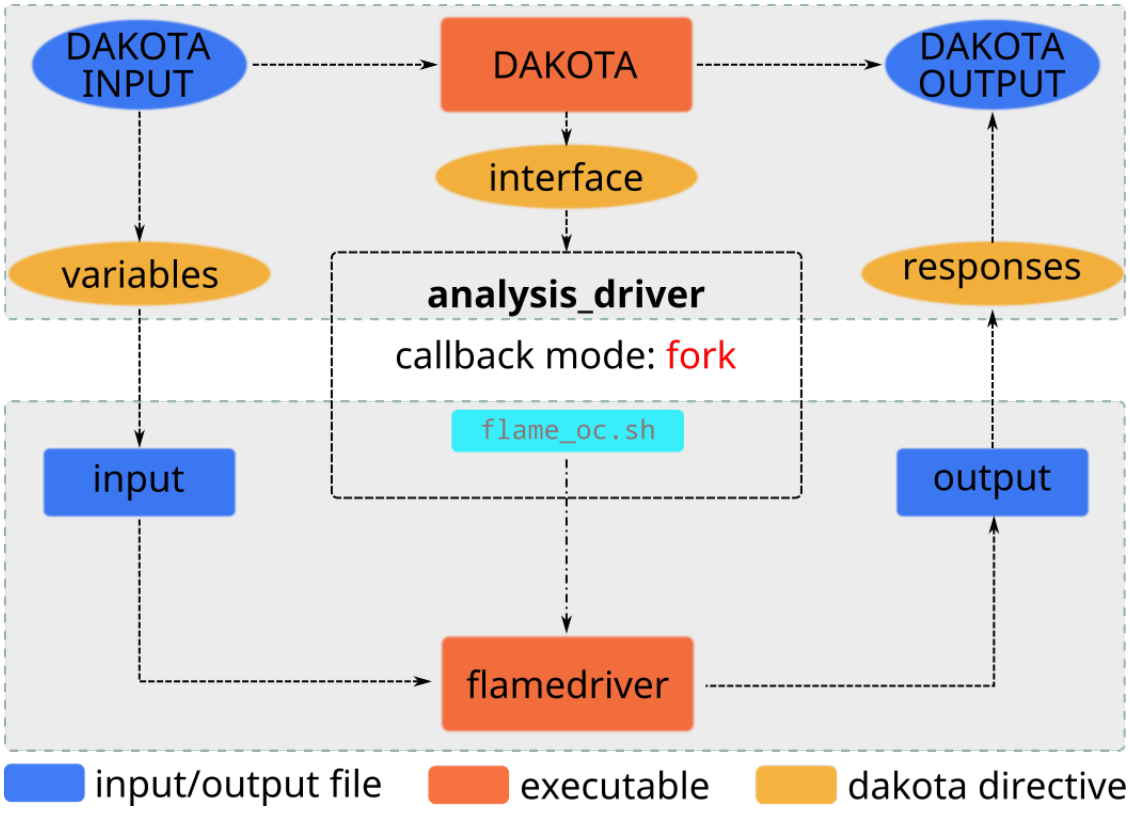
CHAPTER

ONE

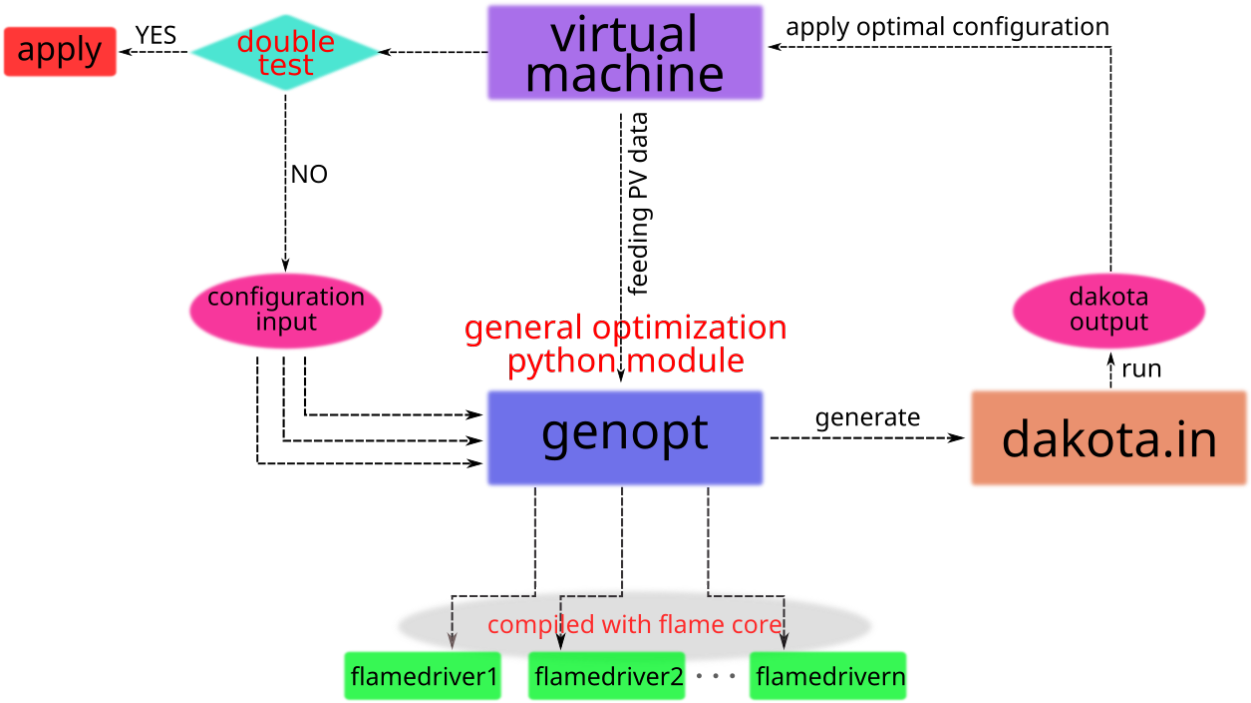
INTRODUCTION

genopt is a python package, trying to serve as a solution of general multi-dimensional optimization. The core optimization algorithms employed inside are mainly provided by DAKOTA, which is the brief for *Design Analysis Kit for Optimization and Terascale Applications*, another tool written in C++.

The following image illustrates the general optimization framework by properly utilizing DAKOTA.



To apply this optimization framework, specific `analysis drivers` should be created first, e.g. `flamedriver1`, `flamedriver2`... indicate the dedicated executable drivers built from C++, for the application in accelerator commissioning, e.g. FRIB.



Note: `flame` is an particle envolope tracking code developed by C++, with the capbility of multi-charge particle states momentum space tracking, it is developed by FRIB; `flamedriver(s)` are user-customized executables by linking the flame core library (`libflame_core.so`) to accomplish various different requirements.

The intention of `genopt` is to provide a uniform interface to do the multi-dimensional optimization tasks. It provides interfaces to let the users to customize the optimization drivers, optimization methods, variables, etc. The optimized results are returned by clean interface. Dedicated analysis drivers should be created and tell the package to use. `DakotaOC` is a dedicated class designed for orbit correction for accelerator, which uses `flame` as the modeling tool.

CHAPTER

TWO

DEMONSTRATIONS

Here goes some examples to use `genopt` package to do orbit correction, it should be noted that the more complicated the script is, the more options could be adjusted to fulfill specific goals.

Getting started

This approach requires fewest input of code to complete the orbit correction optimization task, which also means you only has very few options to adjust to the optimization model. Hopefully, this approach could be used as an ordinary template to fulfill most of the orbit correction tasks. Below is the demo code:

```
import genopt

latfile = 'test_392.lat'
oc_ins = genopt.DakotaOC(lat_file=latfile)

oc_ins.simple_run(method='cg', mpi=True, np=4, iternum=20)

# get output
oc_ins.get_orbit(outfile='orbit.dat')

# plot
oc_ins.plot()
```

The lattice file used here could be found from `here`, or from https://github.com/archman/genopt/blob/master/lattice/test_392.lat.

For this approach, the following default configuration is applied:

1. Selected all BPMs and correctors (both horizontal and vertical types);
2. Set the reference orbit with all BPMs' readings of `x=0` and `y=0`;
3. Set the objective function with the sum of all the square of orbit deviations w.r.t. reference orbit.

By default, `conmin_frcg` optimization method is used, possible options for `simple_run()` could be:

- **common options:**
    1. `mpi`: if True, run in parallel mode; if False, run in serial mode;
    2. `np`: number of cores to use if `mpi` is True;
- **gradient descent, i.e. `method=cg`:**
    1. `iternum`: max iteration number, 20 by default;
    2. `step`: forward gradient step size, 1e-6 by default;
- **pattern search, i.e. `method=ps`:**
    1. `iternum`: max iteration number, 20 by default;

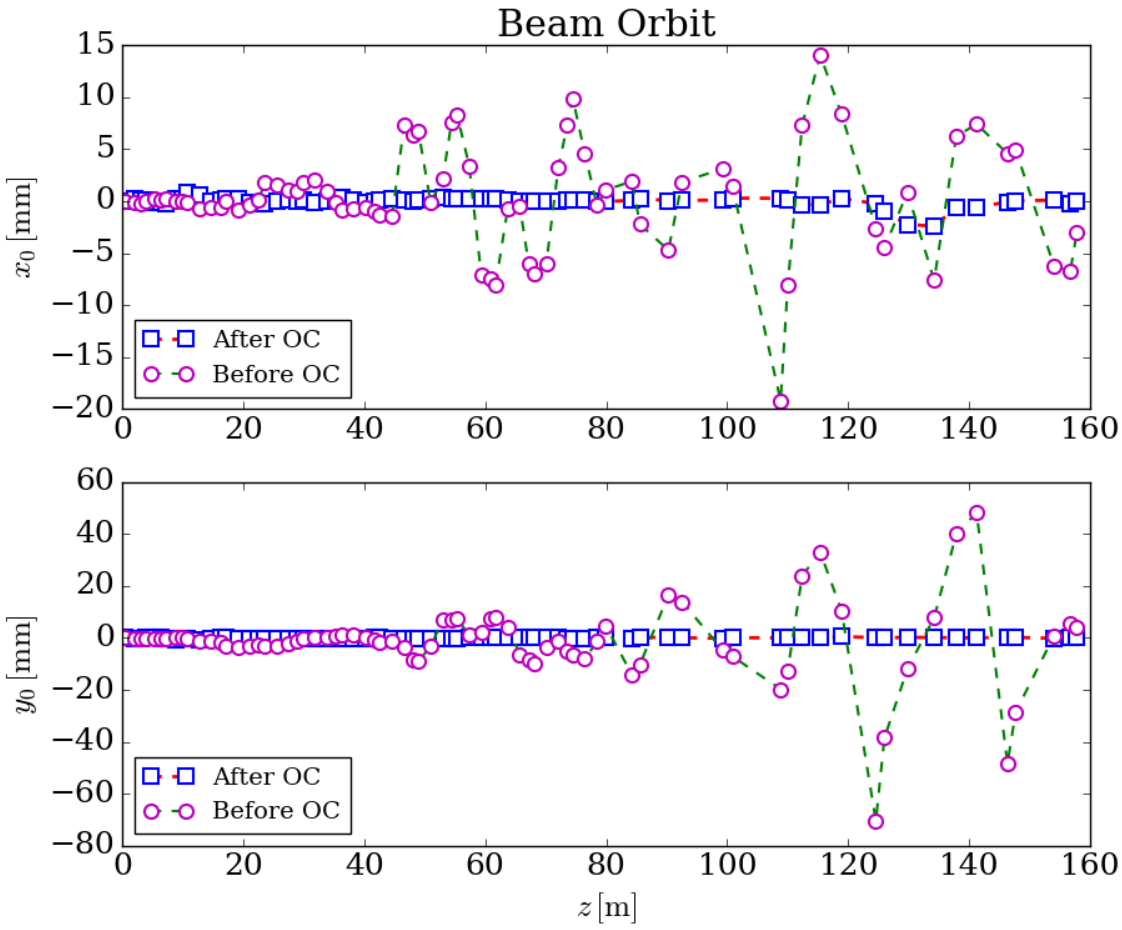
2. `evalnum`: max function evaulation number, 1000 by default;

There are two options for `DakotaOC` maybe useful sometimes:

1. `workdir`: root directory for dakota input and output files
2. `keep`: if keep working files, True or False

After run this script, beam orbit data could be saved into file, e.g. `orbit.dat`:

which could be used to generate figures, the following figure is a typical one could be generated from the optimized results:



Setup BPMs, correctors and reference orbit

For more general cases, genopt provides interfaces to setup BPMs, correctors, reference orbit and objective function type, etc., leaving more controls to the user side, to fulfill specific task.

Here is an exmaple to show how to use these capabilities.

```
import genopt

# lattice file
latfile = 'test_392.lat'
oc_ins = genopt.DakotaOC(lat_file=latfile)
```

```
# select BPMs
bpms = oc_ins.get_elem_by_type('bpm')
oc_ins.set_bpms(bpm=bpms)

# select correctors
hcors = oc_ins.get_all_cors(type='h')[0:40]
vcors = oc_ins.get_all_cors(type='v')[0:40]
oc_ins.set_cors(hcor=hcors, vcor=vcors)

# setup objective function type
oc_ins.ref_flag = "xy"

# setup reference orbit in x and y
bpms_size = len(oc_ins.bpms)
oc_ins.set_ref_x0(np.ones(bpms_size)*0.0)
oc_ins.set_ref_y0(np.ones(bpms_size)*0.0)

# run optimizaiton
oc_ins.simple_run(method='cg', mpi=True, np=4, iternum=30)

# get output
oc_ins.get_orbit(outfile='orbit.dat')

# plot
oc_ins.plot()
```

The highlighted code block is added for controlling all these abovementioned properties.

Warning:

1. BPMs and correctos are distinguished by the element index, which could be get by proper method, e.g. `get_all_cors()`;
2. The array size of selected BPMs and reference orbit must be the same;
3. `bpms`, `hcors`, `vcors` are properties of `DakotaOC` instance.

Note: Objective functions could be chosen from three types according to the value of `ref_flag`:

1. `ref_flag="xy"`: $\sum \Delta x^2 + \sum \Delta y^2$
2. `ref_flag="x"`: $\sum \Delta x^2$
3. `ref_flag="y"`: $\sum \Delta y^2$

where $\Delta x = x - x_0$, $\Delta y = y - y_0$.

Setup variables

By default the variables to be optimized is setup with the following parameters:

initial value	lower bound	upper bound
1e-4	-0.01	0.01

However, subtle configuration could be achieved by using `set_variables()` method of `DakotaOc` class, here is how to do it:

Parameter could be created by using `DakotaParam` class, here is the code:

```
# set x correctors
hcors = oc_ins.get_all_cors(type='h')[0:40]

# set initial, lower, upper values for each variables
n_h = len(hcors)
xinit_vals = (np.random.random(size=n_h) - 0.5) * 1.0e-4
xlower_vals = np.ones(n_h) * (-0.01)
xupper_vals = np.ones(n_h) * 0.01
xlbls = ['X{0:03d}'.format(i) for i in range(1, n_h+1)]

# create parameters
plist_x = [genopt.DakotaParam(lbl, val_i, val_l, val_u)
           for (lbl, val_i, val_l, val_u) in
               zip(xlbls, xinit_vals, xlower_vals, xupper_vals)]
```

`plist_y` could be created in the same way, then issue `set_variables()` with `set_variables(plist=plist_x+plist_y)`.

Note: The emphasized line is to setup the variable labels, it is recommended that all parameters' label with the format like `x001`, `x002`, etc.

Setup optimization engine

The simplest approach, (see *Getting started*), just covers detail of the more specific configurations, especially for the optimization engine itself, however `genopt` provides different interfaces to make customized adjustment.

Method

`DakotaMethod` is designed to handle `method` block, which is essential to define the optimization method, e.g.

```
oc_method = genopt.DakotaMethod(method='ps', max_iterations=200,
            contraction_factor=0.8)
# other options could be added, like max_function_evaluations=2000
oc_ins.set_method(oc_method)
```

Interface

`DakotaInterface` is designed to handle `interface` block, for the general optimization regime, `fork` mode is the common case, only if the analysis driver is compile into dakota, `direct` could be used.

Here is an example of user-defined interface:

```
bpms = [10,20,30]
hcors, vcors = [5, 10, 20], [7, 12, 30]
latfile = 'test.lat'
oc_inter = genopt.DakotaInterface(mode='fork',
                    driver='flamedriver_oc',
                    latfile=latfile,
                    bpms=bpms, hcors=hcors, vcors=vcors,)
# set interface
oc_ins.set_interface(oc_inter)
```

Note: Extra parameters could be added by this way: oc_inter.set_extra(deactivate="active_set_vector")

Responses

Objective function(s) and gradients/hessians could be set in `responses` block, which is handled by `DakotaResponses` class.

Typical example:

```
oc_responses = DakotaResponses(gradient='numerical', step=2.0e-7)
oc_ins.set_responses(oc_responses)
```

Environment

Dakota environment block could be adjusted by instantiating class `DakotaEnviron`, e.g.

```
datfile = 'dakota1.dat'
e = genopt.DakotaEnviron(tabfile=datfile)
oc_ins.set_environ(e)
```

`tabfile` option could be used to define where the dakota tabular data should go, will not generate tabular file if not set.

Model

`DakotaModel` is designed to handle `model` block, recently, just use the default configuration, i.e:

```
oc_ins.set_model()
# or:
m = genopt.DakotaModel()
oc_ins.set_model(m)
```

Run optimization

If running optimization not by `simple_run()` method, another approach should be utilized.

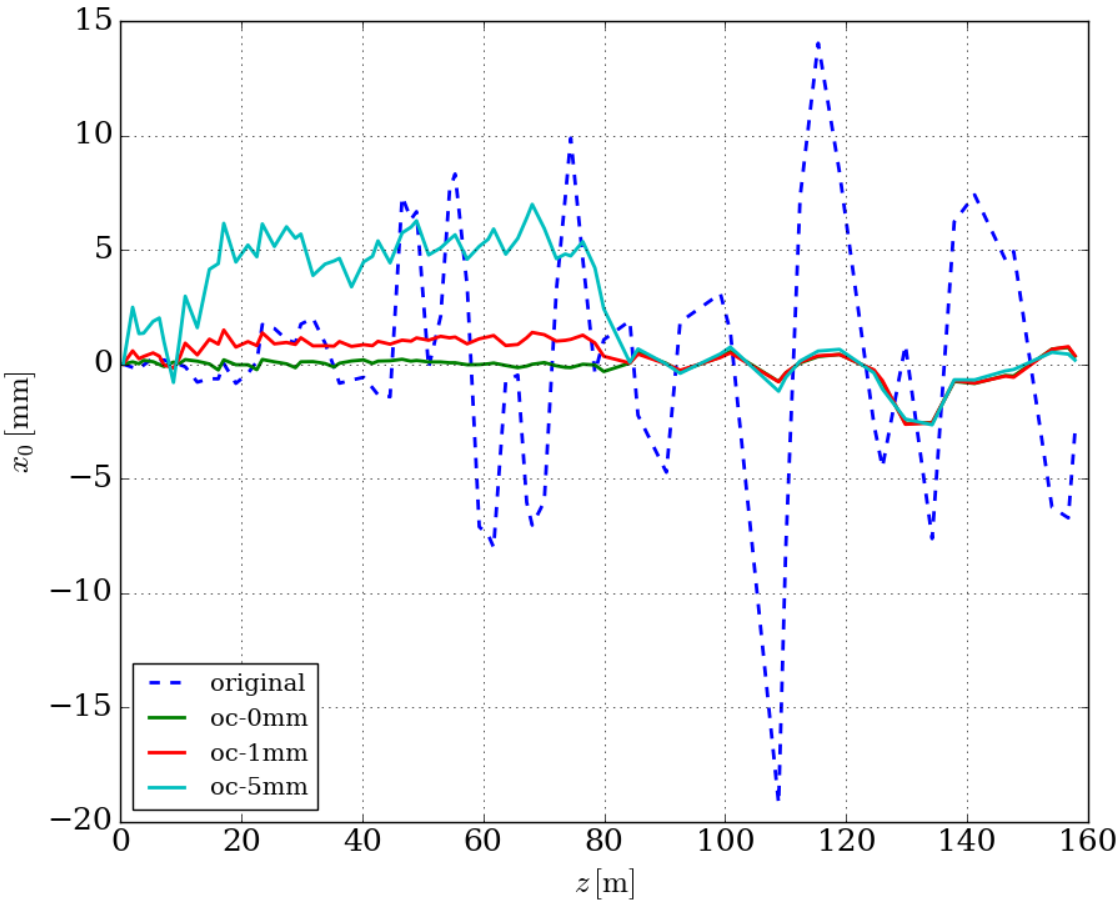
```
# generate input file for optimization
oc_ins.gen_dakota_input()

# run optimization
oc_ins.run(mpi=True, np=4)
```

Below is a typical user customized script to find the optimized correctors configurations.

```
import os
import genopt

""" orbit correction demo
"""
latfile = 'test_392.lat'
oc_ins = genopt.DakotaOC(lat_file=latfile,
                         workdir='./oc_tmp4',
                         keep=True)

# set BPMs and correctors
bpms = oc_ins.get_elem_by_type('bpm')
hcors = oc_ins.get_all_cors(type='h')[0:40]
vcors = oc_ins.get_all_cors(type='v')[0:40]
oc_ins.set_bpms(bpm=bpms)
oc_ins.set_cors(hcor=hcors, vcor=vcors)

# set parameters
oc_ins.set_variables()
```

```
# set interface
oc_ins.set_interface()

# set responses
r = genopt.DakotaResponses(gradient='numerical',step=2.0e-5)
oc_ins.set_responses(r)

# set model
m = genopt.DakotaModel()
oc_ins.set_model(m)

# set method
md = genoptDakotaMethod(method='ps',
        max_function_evaluations=1000)
oc_ins.set_method(method=md)

# set environment
tabfile = os.path.abspath('./oc_tmp4/dakota1.dat')
e = genopt.dakutils.DakotaEnviron(tabfile=tabfile)
oc_ins.set_environ(e)

# set reference orbit
bpms_size = len(oc_ins.bpms)
ref_x0 = np.ones(bpms_size)*0.0
ref_y0 = np.ones(bpms_size)*0.0
oc_ins.set_ref_x0(ref_x0)
oc_ins.set_ref_y0(ref_y0)

# set objective function
oc_ins.ref_flag = "xy"

# generate input
oc_ins.gen_dakota_input()

# run
oc_ins.run(mpi=True, np=4)
#print oc_ins.get_opt_results()

# get output
oc_ins.get_orbit((oc_ins.hcor, oc_ins.vcor), oc_ins.get_opt_results(),
                 outfile='orbit.dat')

# plot
#oc_ins.plot()
```

The following figure shows correct the orbit to different reference orbits.



After optimization

Suppose all the optimized results have been generated, here are the possible post-operations:

1. Operations on the optimized `Machine` object;
2. Generate new lattice file with optimized results for other programs.

Optimization snippet:

```
latfile = 'test_392.lat'
oc = genopt.DakotaOC(lat_file=latfile)
oc.simple_run(iternum=20)
```

Get optimized results

Optimized results could be retrieved by `get_opt_results()` method of `DakotaOC` class:

- return type: list

```
>>> r = oc.get_opt_results(rtype='list')
>>> print(r)
```

```
[0.00013981587907,
 7.5578423135e-05,
 -5.3982438406e-05,
 -1.9620020032e-06,
 0.00017942079806,
 ...
 2.0182502319e-05,
 0.0001173634281,
 8.685656753e-05,
 7.3950720611e-05,
 8.2924283647e-05]
```

The returned list is alphabetically sorted according to the variables' names.

- return type: dictionary, label format: `plain`

```
>>> r = oc.get_opt_results()
>>> # or
>>> r = oc.get_opt_results(rtype='dict', label='plain')
>>> print(r)
{'x001': 0.00013981587907,
 'x002': 7.5578423135e-05,
 'x003': -5.3982438406e-05,
 'x004': -1.9620020032e-06,
 'x005': 0.00017942079806,
 ...
 'y056': 2.0182502319e-05,
 'y057': 0.0001173634281,
 'y058': 8.685656753e-05,
 'y059': 7.3950720611e-05,
 'y060': 8.2924283647e-05}
```

- return type: dictionary, label format: `fancy`

```
>>> r = oc.get_opt_results(label='fancy')
>>> print(r)
{'FS1_BBS:DCH_D2412': {'config': {'theta_x': 0.00021066533055}, 'id': 1048},
 'FS1_BBS:DCH_D2476': {'config': {'theta_x': 0.00025833402592}, 'id': 1098},
 ...
```

This is the more comprehensive way to represent the results, one of the advantages is that results with this format could be easily to apply on to `reconfigure` method of `Machine` object, for instance:

```
>>> for k,v in r.items():
>>>     m.reconfigure(v['id'], v['config'])
```

Note: `get_opt_results` has `outfile` optional parameter, if not defined, output file that generated by current optimization instance would be used, or the defined dakota output file would be used, but only valid for cases of `label='plain'`; `label='fancy'` is only valid for the case of `rtype='dict'`.

Get orbit data

`get_orbit()` could be used to apply all the optimized results, then new `Machine` could be get in the following way:

```
>>> z,x,y,m = oc.get_orbit()
>>> print(m.conf(1224)['theta_x'])
8.5216269467e-05
```

Or in another way:

```
>>> oc.get_orbit()
>>> m = oc.get_machine()
```

New machine `m` could be used for the next operations.

Note: `get_orbit()` could be assigned a optional parameter: `outfile`, into which the plain ASCII data of `zpos`, `x`, and `y` would be saved.

Get new optimized lattice file

`get_opt_latfile()` is created to generate new lattice file with optimized results, for the sake of possible next usage of asking for lattice file, this is kind of more general interface.

```
>>> oc.get_opt_latfile(outfile='opt1.lat')
```

Here is the links to the lattice files of `original` and `optimized` ones, both could be used as the input lattice file of `flame` program.

CHAPTER

THREE

API

genopt package

General multi-dimensional optimization package built by Python, incorporating optimization algorithms provided by DAKOTA.

version 0.0.2

author Tong Zhang <zhangt@frib.msu.edu>

Example

```
>>> # This is a ordinary example to do orbit correction by
>>> # multi-dimensional optimization approach.
>>>
>>> # import package
>>> import genopt
>>>
>>> # lattice file name
>>> latfile = './contrib/test_392.lat'
>>>
>>> # create optimization object
>>> oc_ins = genopt.DakotaOC(lat_file=latfile)
>>>
>>> # get indices of BPMs and correctors
>>> bpms = oc_ins.get_elem_by_type('bpm')
>>> cors = oc_ins.get_all_cors()[45:61]
>>>
>>> # set BPMs and correctors
>>> oc_ins.set_bpms(bpm=bpms)
>>> oc_ins.set_cors(cor=cors)
>>>
>>> # run optimization, enable MPI,
>>> # with optimization of CG, 20 iterations
>>> oc_ins.simple_run(method='cg', mpi=True, np=4, iternum=20)
>>>
>>> # get optimized results:
>>> opt_vars = oc_ins.get_opt_results()
>>>
>>> # or show orbit after correction
>>> oc_ins.plot()
>>>
>>> # or save the orbit data (to file)
>>> oc_ins.get_orbit(outfile='orbit.dat')
>>>
```

class `DakotaInput`(***kws*)

Bases: `object`

template of dakota input file, field could be overriden by providing additional keyword arguments,

Parameters **kws** – keyword arguments, valid keys are dakota directives

Example

```
>>> dak_inp = DakotaInput(method=["max_iterations = 500",
                                  "convergence_tolerance = 1e-7",
                                  "conmin_frcg",])
>>>
```

set_template(*name='oc'*)

write(*infile=None*)
write all the input into file, as dakota input file

Parameters **infile** – fullname of input file, if not defined, infile will be assigned as 'dakota.in' in current working directory

class **DakotaParam**(*label*, *initial=0.0*, *lower=-10000000000.0*, *upper=10000000000.0*)
Bases: `object`

create dakota variable for `variables` block

Parameters

- **label** – string to represent itself, e.g. `x001`, it is recommended to annotate the number with the format of `%03d`, i.e. `1 --> 001`, `10 --> 010`, `100 --> 100`, etc.
- **initial** – initial value, 0.0 by default
- **lower** – lower bound, -1.0e10 by default
- **upper** – upper bound, 1.0e10 by default

initial

label

lower

upper

class **DakotaBase**(***kws*)
Bases: `object`

Base class for general optimization, initialized parameters: valid keyword parameters:

- workdir: root dir for dakota input/output files, the defualt one should be created in /tmp, or define some dir path
- dakexec: full path of dakota executable, the default one should be *dakota*, or define the full path
- dakhead: prefixed name for input/output files of *dakota*, the default one is *dakota*
- keep: if keep the working directory (i.e. defined by *workdir*), default is False

dakexec

dakhead

keep

workdir

class **DakotaOC**(*lat_file=None*, *elem_bpm=None*, *elem_cor=None*, *elem_hcor=None*, *elem_vcor=None*, *ref_x0=None*, *ref_y0=None*, *ref_flag=None*, *model=None*, *optdriver=None*, ***kws*)
Bases: `genopt.dakopt.DakotaBase`

Dakota optimization class with orbit correction driver

Parameters

- **lat_file** – lattice file

- **elem_bpm** – list of element indice of BPMs
- **elem_cor** – list of element indice of correctors, always folders of 2
- **elem_hcor** – list of element indice of horizontal correctors
- **elem_vcor** – list of element indice of vertical correctors
- **ref_x0** – reference orbit in x, list of BPM readings
- **ref_y0** – reference orbit in y, list of BPM readings
- **ref_flag** – string flag for objective functions:
  1. "x": $\sum \Delta x^2$, $\Delta x = x - x_0$;
  2. "y": $\sum \Delta y^2$, $\Delta y = y - y_0$;
  3. "xy": $\sum \Delta x^2 + \sum \Delta y^2$.
- **model** – simulation model, 'flame' or 'impact'
- **optdriver** – analysis driver for optimization, 'flamedriver_oc' by default
- **kws** –

  keywords parameters for additional usage, defined in DakotaBase class valid keys:

  - *workdir*: root dir for dakota input/output files, the defualt one should be created in /tmp, or define some dir path
  - *dakexec*: full path of dakota executable, the default one should be *dakota*, or define the full path
  - *dakhead*: prefixed name for input/output files of *dakota*, the default one is *dakota*
  - *keep*: if keep the working directory (i.e. defined by *workdir*), default is False

bpms

create_machine(*lat_file*)
create machine instance with model configuration

- setup _machine
- setup _elem_bpm, _elem_cor or (_elem_hcor and _elem_vcor)

gen_dakota_input(*infile=None*, *debug=False*)
generate dakota input file

Parameters

- **infile** – dakota input filename
- **debug** – if True, generate a simple test input file

get_all_bpms()
get list of all valid bpms indices

Returns a list of bpm indices

Example

```
>>> dakoc = DakotaOC('test/test.lat')
>>> print(dakoc.get_all_bpms())
```

get_all_cors(*type=None*)
get list of all valid correctors indices

Parameters **type** – define corrector type, 'h': horizontal, 'v': vertical, if not defined, return all correctors

Returns a list of corrector indices

Example

```
>>> dakoc = DakotaOC('test/test.lat')
>>> print(dakoc.get_all_cors())
```

get_elem_by_name(*name*)
get list of element(s) by name(s)

Parameters **name** – tuple or list of name(s)

Returns list of element indices

Example

```
>>> dakoc = DakotaOC('test/test.lat')
>>> names = ('LS1_CA01:BPM_D1144', 'LS1_WA01:BPM_D1155')
>>> idx = dakoc.get_elem_by_name(names)
>>> print(idx)
[18, 31]
```

get_elem_by_type(*type*)
get list of element(s) by type

Parameters **type** – string name of element type

Returns list of element indices

Example

```
>>> dakoc = DakotaOC('test/test.lat')
>>> type = 'bpm'
>>> idx = dakoc.get_elem_by_type(type)
>>> print(idx)
```

get_machine()
get flame machine object for potential usage

Returns flame machine object or None

get_opt_latfile(*outfile='out.lat'*)
get optimized lattice file for potential next usage, `run()` or `simple_run()` should be evoked first to get the optimized results.

Parameters **outfile** – file name for generated lattice file

Returns lattice file name or None if failed

get_opt_results(*outfile=None*, *rtype='dict'*, *label='plain'*)
extract optimized results from dakota output

Parameters

- **outfile** – file name of dakota output file, 'dakota.out' by default
- **rtype** – type of returned results, 'dict' or 'list', 'dict' by default
- **label** – label types for returned variables, only valid when rtype 'dict', 'plain' by default:

- *'plain'*: variable labels with format of x1, x2, y1, y2, etc. e.g. `{'x1': v1,'y1': v2}`
- *'fancy'*: variable labels with the name defined in lattice file, e.g. `'LS1_CA01:DCH_D1131'`, dict returned sample: `{'LS1_CA01:DCH_D1131': {'id':9,'config':{'theta_x':v1}}}`

Note: The `fancy` option will make re-configuring flame machine in a more convenient way, such as:

```
>>> opt_cors = get_opt_results(label='fancy')
>>> for k,v in opt_cors.items():
>>>     m.reconfigure(v['id'], v['config'])
>>> # here m is an instance of flame.Machine class
>>>
```

Returns by default return a dict of optimized results with each item of the format like "x1":0.1 or more fancy format by set label with 'fancy', etc., if rtype='list', return a list of values, when the keys are ascend sorted.

Example

```
>>> opt_vars = get_optresults(outfile='flame_oc.out', rtype='dict'):
>>> print(opt_vars)
{'x2': 0.0020782814353, 'x1': -0.0017913264033}
>>> opt_vars = get_optresults(outfile='flame_oc.out', rtype='list'):
>>> print(opt_vars)
[-0.0017913264033, 0.0020782814353]
```

get_orbit(*idx=None*, *val=None*, *outfile=None*)
calculate the orbit with given configurations

Parameters

- **idx** – (idx_hcor, idx_vcor), tuple of list of indices of h/v cors
- **val** – values for each correctos, h/v
- **outfile** – filename to save the data

Returns tuple of zpos, env_x, env_y, machine

hcors

latfile

optdriver

plot(*outfile=None*, *figsize=(10, 8)*, *dpi=120*, ***kws*)
show orbit

Parameters

- **outfile** – output file of dakota
- **figsize** – figure size, (h, w)
- **dpi** – figure dpi

ref_flag

ref_x0

ref_y0

run(*mpi=False*, *np=None*)
run optimization

Parameters

- **mpi** – if True, run DAKOTA in parallel mode, False by default
- **np** – number of processes to use, only valid when `mpi` is True

set_bpms(*bpm=None*)
set BPMs

Parameters **bpm** – list of bpm indices, if None, use all BPMs

set_cors(*cor=None*, *hcor=None*, *vcor=None*)
set correctors, if cor, hcor and vcor are None, use all correctors if cor is not None, use cor, ignore hcor and vcor

Parameters

- **cor** – list of corrector indices, hcor, vcor,...
- **hcor** – list of horizontal corrector indices
- **vcor** – list of vertical corrector indices

set_environ(*environ=None*)
setup environment block, that is setup `oc_environ`

Parameters **environ** – `DakotaEnviron` object, automatically setup if not defined

set_interface(*interface=None*, ***kws*)
setup interface block, that is setup `oc_interface` should be ready to invoke after `set_cors` and `set_bpms`

Parameters **interface** – `DakotaInterface` object, automatically setup if not defined

set_method(*method=None*)
setup method block, that is setup `oc_method`

Parameters **method** – `DakotaMethod` object, automatically setup if not defined

set_model(*model=None*, ***kws*)
setup model block, that is setup `oc_model`

Parameters **model** – `DakotaModel` object, automatically setup if not defined

set_ref_x0(*ref_arr=None*)
set reference orbit in x, if not set, use 0s

Parameters **ref_arr** – array of reference orbit values size should be the same number as selected BPMs

set_ref_y0(*ref_arr=None*)
set reference orbit in y, if not set, use 0s

Parameters **ref_arr** – array of reference orbit values size should be the same number as selected BPMs

set_responses(*responses=None*, ***kws*)
setup responses block, that is setup `oc_responses`

Parameters **responses** – `DakotaResponses` object, automatically setup if not defined

set_variables(*plist=None*, *initial=0.0001*, *lower=-0.01*, *upper=0.01*)
setup variables block, that is setup `oc_variables` should be ready to invoke after `set_cors()`

Parameters

- **plist** – list of defined parameters (DakotaParam object), automatically setup if not defined
- **initial** – initial values for all variables, only valid when plist is None
- **lower** – lower bound for all variables, only valid when plist is None
- **upper** – upper bound for all variables, only valid when plist is None

simple_run(*method='cg'*, *mpi=None*, *np=None*, ***kws*)

run optimization after set_bpms() and set_cors(), by using default configuration and make full use of computing resources.

Parameters

- **method** – optimization method, 'cg', 'ps', 'cg' by default
- **mpi** – if True, run DAKOTA in parallel mode, False by default
- **np** – number of processes to use, only valid when mpi is True
- **kws** – keyword parameters valid keys:
  - step: gradient step, 1e-6 by default
  - iternum: max iteration number, 20 by default
  - evalnum: max function evaulation number, 1000 by default

vcors

class DakotaEnviron(*tabfile=None*, ***kws*)

Bases: object

create datako environment for environment block

Parameters

- **tabfile** – tabular file name, by default not save tabular data
- **kws** – other keyword parameters

Example

```
>>> # default
>>> oc_environ = DakotaEnviron()
>>> print oc_environ.get_config()
[]
>>> # define name of tabular file
>>> oc_environ = DakotaEnviron(tabfile='tmp.dat')
>>> print oc_environ.get_config()
['tabular_data', " tabular_data_file 'tmp.dat'"]
>>>
```

get_config(*rtype='list'*)

get responses configuration for dakota input block

Parameters **rtype** – 'list' or 'string'

Returns dakota responses input

class DakotaInterface(*mode='fork'*, *driver='flamedriver_oc'*, *latfile=None*, *bpms=None*, *hcors=None*, *vcors=None*, *ref_x0=None*, *ref_y0=None*, *ref_flag=None*, ***kws*)

Bases: object

create dakota interface for interface block

Parameters

- **mode** – 'fork' or 'direct' (future usage)
- **driver** – analysis driver, external ('fork') executable file internal ('direct') executable file
- **latfile** – file name of (flame) lattice file
- **bpms** – array of selected BPMs' id
- **hcors** – array of selected horizontal (x) correctors' id
- **vcors** – array of selected vertical (y) correctors' id
- **ref_x0** – array of BPM readings for reference orbit in x, if not defined, use 0s
- **ref_y0** – array of BPM readings for reference orbit in y, if not defined, use 0s
- **ref_flag** – string flag for objective functions:
  1. "x": $\sum \Delta x^2$, $\Delta x = x - x_0$;
  2. "y": $\sum \Delta y^2$, $\Delta y = y - y_0$;
  3. "xy": $\sum \Delta x^2 + \sum \Delta y^2$.
- **kws** – keyword parameters, valid keys: e.g.: * deactivate, possible value: 'active_set_vector'

Note: mode should be set to be 'direct' when the analysis drivers are built with dakota library, presently, 'fork' is used.

Example

```
>>> # for orbit correction
>>> bpms = [1,2,3] # just for demonstration
>>> hcors, vcors = [1,3,5], [2,4,6]
>>> latfile = 'test.lat'
>>> oc_interface = DakotaInterface(mode='fork',
>>>                                driver='flamedriver_oc',
>>>                                latfile=latfile
>>>                                bpms=bpms, hcors=hcors, vcors=vcors,
>>>                                ref_x0=None, ref_y0=None,
>>>                                ref_flag=None,
>>>                                deactivate='active_set_vector')
>>> # add extra configurations
>>> oc_interface.set_extra(p1='v1', p2='v2')
>>> # get configuration
>>> config_str = oc_interface.get_config()
>>>
```

bpms

driver

get_config(*rtype='list'*)

get interface configuration for dakota input block

Parameters **rtype** – 'list' or 'string'

Returns dakota interface input

hcors

latfile

mode

ref_flag

ref_x0

ref_y0

set_extra(***kws*)
add extra configurations

vcors

class DakotaMethod(*method='cg'*, *iternum=20*, *tolerance=0.0001*, ***kws*)
Bases: `object`

create dakota method for `method` block

Parameters

- **method** – method name, 'cg' by default, all possible choices: 'cg', 'ps'
- **iternum** – max iteration number, 20 by default
- **tolerance** – convergence tolerance, 1e-4 by default
- **kws** – other keyword parameters

Example

```
>>> # default
>>> oc_method = DakotaMethod()
>>> print oc_method.get_config()
['conmin_frcg', ' convergence_tolerance 0.0001', ' max_iterations 20']
>>> # define method with pattern search
>>> oc_method = DakotaMethod(method='ps')
>>> print oc_method.get_config()
['coliny_pattern_search', ' contraction_factor 0.75', ' max_function_evaluations 500',
 ' solution_accuracy 0.0001', ' exploratory_moves basic_pattern',
 ' threshold_delta 0.0001', ' initial_delta 0.5', ' max_iterations 100']
>>> # modify options of pattern search method
>>> oc_method = DakotaMethod(method='ps', max_iterations=200, contraction_factor=0.8)
>>> print oc_method.get_config()
['coliny_pattern_search', ' contraction_factor 0.8', ' max_function_evaluations 500',
 ' solution_accuracy 0.0001', ' exploratory_moves basic_pattern',
 ' threshold_delta 0.0001', ' initial_delta 0.5', ' max_iterations 200']
>>> # conmin_frcg method
>>> oc_method = DakotaMethod(method='cg')
>>> print oc_method.get_config()
['conmin_frcg', ' convergence_tolerance 0.0001', ' max_iterations 20']
>>> # modify options
>>> oc_method = DakotaMethod(method='cg', max_iterations=100)
>>> print oc_method.get_config()
['conmin_frcg', ' convergence_tolerance 0.0001', ' max_iterations 100']
>>>
```

get_config(*rtype='list'*)
get method configuration for dakota input block

Parameters **rtype** – 'list' or 'string'

Returns dakota method input

get_default_method(*method*)
get default configuration of some method

Parameters **method** – method name, 'cg' or 'ps'

Returns dict of configuration

method(*method*)
return method configuration

Parameters **method** – method stirng name, 'cg' or 'ps'

Returns list of method configuration

class DakotaModel(***kws*)

Bases: object

create dakota model for model block

get_config(*rtype='list'*)

get model configuration for dakota input block

Parameters **rtype** – 'list' or 'string'

Returns dakota model input

class DakotaResponses(*nfunc=1*, *gradient=None*, *hessian=None*, ***kws*)

Bases: object

create dakota responses for responses block

Parameters

- **nfunc** – num of objective functions
- **gradient** – gradient type: 'analytic' or 'numerical'
- **hessian** – hessian configuration
- **kws** – keyword parameters for gradients and hessians valid keys: any available for responses among which key name of 'grad' is for gradients configuration, the value should be a dict (future)

Example

```
>>> # default responses:
>>> response = DakotaResponses()
>>> print response.get_config()
['num_objective_functions = 1', 'no_gradients', 'no_hessians']
>>>
>>> # responses with analytic gradients:
>>> response = DakotaResponses(gradient='analytic')
>>> print response.get_config()
['num_objective_functions = 1', 'analytic_gradients', 'no_hessians']
>>>
>>> # responses with numerical gradients, default configuration:
>>> oc_responses = DakotaResponses(gradient='numerical')
>>> print oc_responses.get_config()
['num_objective_functions = 1', 'numerical_gradients', ' method_source dakota',
 ' interval_type forward', ' fd_gradient_step_size 1e-06', 'no_hessians']
>>>
>>> # responses with numerical gradients, define step:
>>> oc_responses = DakotaResponses(gradient='numerical', step=2.0e-7)
>>> print oc_responses.get_config()
['num_objective_functions = 1', 'numerical_gradients', ' method_source dakota',
 ' interval_type forward', ' fd_gradient_step_size 2e-07', 'no_hessians']
>>>
>>> # given other keyword parameters:
>>> oc_responses = DakotaResponses(gradient='numerical', step=2.0e-7, k1='v1', k2='v2')
>>> print oc_responses.get_config()
['num_objective_functions = 1', 'numerical_gradients', ' method_source dakota',
 ' interval_type forward', ' fd_gradient_step_size 2e-07', 'no_hessians', 'k2 = v2', 'k1 = v1']
>>>
```

get_config(*rtype='list'*)

get responses configuration for dakota input block

Parameters **rtype** – 'list' or 'string'

Returns dakota responses input

gradients(*type=None*, *step=1e-06*, ***kws*)

generate gradients configuration

Parameters

- **type** – 'numerical' or 'analytic' (default)
- **step** – gradient step size, only valid when type is numerical
- **kws** – other keyword parameters

Returns list of configuration

get_opt_results(*outfile='dakota.out'*, *rtype='dict'*)
extract optimized results from dakota output

Parameters

- **outfile** – file name of dakota output file, 'dakota.out' by default
- **rtype** – type of returned results, 'dict' or 'list', 'dict' by default

Returns by default return a dict of optimized results with each item of the format like "x1":0.1, etc., or if rtype='list', return a list of values, when the keys are ascend sorted.

Example

```
>>> opt_vars = get_opt_results(outfile='flame_oc.out', rtype='dict'):
>>> print(opt_vars)
{'x2': 0.0020782814353, 'x1': -0.0017913264033}
>>> opt_vars = get_opt_results(outfile='flame_oc.out', rtype='list'):
>>> print(opt_vars)
[-0.0017913264033, 0.0020782814353]
```

Submodules

genopt.dakopt module

General optimization module by utilizing DAKOTA

- orbit correction: `DakotaOC`

Tong Zhang <zhangt@frib.msu.edu>

2016-10-23 14:26:13 PM EDT

class DakotaBase(***kws*)
Bases: `object`

Base class for general optimization, initialized parameters: valid keyword parameters:

- workdir: root dir for dakota input/output files, the defualt one should be created in /tmp, or define some dir path
- dakexec: full path of dakota executable, the default one should be *dakota*, or define the full path
- dakhead: prefixed name for input/output files of *dakota*, the default one is *dakota*
- keep: if keep the working directory (i.e. defined by *workdir*), default is False

dakexec

dakhead

keep

workdir

class `DakotaOC`(*lat_file=None*, *elem_bpm=None*, *elem_cor=None*, *elem_hcor=None*, *elem_vcor=None*, *ref_x0=None*, *ref_y0=None*, *ref_flag=None*, *model=None*, *optdriver=None*, ***kws*)

Bases: `genopt.dakopt.DakotaBase`

Dakota optimization class with orbit correction driver

Parameters

- **`lat_file`** – lattice file
- **`elem_bpm`** – list of element indice of BPMs
- **`elem_cor`** – list of element indice of correctors, always folders of 2
- **`elem_hcor`** – list of element indice of horizontal correctors
- **`elem_vcor`** – list of element indice of vertical correctors
- **`ref_x0`** – reference orbit in x, list of BPM readings
- **`ref_y0`** – reference orbit in y, list of BPM readings
- **`ref_flag`** – string flag for objective functions:
  1. "x": $\sum \Delta x^2$, $\Delta x = x - x_0$;
  2. "y": $\sum \Delta y^2$, $\Delta y = y - y_0$;
  3. "xy": $\sum \Delta x^2 + \sum \Delta y^2$.
- **`model`** – simulation model, 'flame' or 'impact'
- **`optdriver`** – analysis driver for optimization, 'flamedriver_oc' by default
- **`kws`** –

  keywords parameters for additional usage, defined in `DakotaBase` class valid keys:

  - *workdir*: root dir for dakota input/output files, the defualt one should be created in /tmp, or define some dir path
  - *dakexec*: full path of dakota executable, the default one should be *dakota*, or define the full path
  - *dakhead*: prefixed name for input/output files of *dakota*, the default one is *dakota*
  - *keep*: if keep the working directory (i.e. defined by *workdir*), default is False

`bpms`

`create_machine`(*lat_file*)

create machine instance with model configuration

- setup _machine
- setup _elem_bpm, _elem_cor or (_elem_hcor and _elem_vcor)

`gen_dakota_input`(*infile=None*, *debug=False*)

generate dakota input file

Parameters

- **`infile`** – dakota input filename
- **`debug`** – if True, generate a simple test input file

get_all_bpms()
get list of all valid bpms indices

Returns a list of bpm indices

Example

```
>>> dakoc = DakotaOC('test/test.lat')
>>> print(dakoc.get_all_bpms())
```

get_all_cors(*type=None*)
get list of all valid correctors indices

Parameters **type** – define corrector type, 'h': horizontal, 'v': vertical, if not defined, return all correctors

Returns a list of corrector indices

Example

```
>>> dakoc = DakotaOC('test/test.lat')
>>> print(dakoc.get_all_cors())
```

get_elem_by_name(*name*)
get list of element(s) by name(s)

Parameters **name** – tuple or list of name(s)

Returns list of element indices

Example

```
>>> dakoc = DakotaOC('test/test.lat')
>>> names = ('LS1_CA01:BPM_D1144', 'LS1_WA01:BPM_D1155')
>>> idx = dakoc.get_elem_by_name(names)
>>> print(idx)
[18, 31]
```

get_elem_by_type(*type*)
get list of element(s) by type

Parameters **type** – string name of element type

Returns list of element indices

Example

```
>>> dakoc = DakotaOC('test/test.lat')
>>> type = 'bpm'
>>> idx = dakoc.get_elem_by_type(type)
>>> print(idx)
```

get_machine()
get flame machine object for potential usage

Returns flame machine object or None

get_opt_latfile(*outfile='out.lat'*)
get optimized lattice file for potential next usage, `run()` or `simple_run()` should be evoked first to get the optimized results.

Parameters **outfile** – file name for generated lattice file

Returns lattice file name or None if failed

get_opt_results(*outfile=None*, *rtype='dict'*, *label='plain'*)
extract optimized results from dakota output

Parameters

- **outfile** – file name of dakota output file, 'dakota.out' by default
- **rtype** – type of returned results, 'dict' or 'list', 'dict' by default
- **label** – label types for returned variables, only valid when rtype 'dict', 'plain' by default:
  - *'plain'*: variable labels with format of x1, x2, y1, y2, etc. e.g. `{'x1': v1,'y1': v2}`
  - *'fancy'*: variable labels with the name defined in lattice file, e.g. `'LS1_CA01:DCH_D1131'`, dict returned sample: `{'LS1_CA01:DCH_D1131': {'id':9,'config':{'theta_x':v1}}}`

Note: The `fancy` option will make re-configuring flame machine in a more convenient way, such as:

```
>>> opt_cors = get_opt_results(label='fancy')
>>> for k,v in opt_cors.items():
>>>     m.reconfigure(v['id'], v['config'])
>>> # here m is an instance of flame.Machine class
>>>
```

Returns by default return a dict of optimized results with each item of the format like "x1":0.1 or more fancy format by set label with 'fancy', etc., if rtype='list', return a list of values, when the keys are ascend sorted.

Example

```
>>> opt_vars = get_optresults(outfile='flame_oc.out', rtype='dict'):
>>> print(opt_vars)
{'x2': 0.0020782814353, 'x1': -0.0017913264033}
>>> opt_vars = get_optresults(outfile='flame_oc.out', rtype='list'):
>>> print(opt_vars)
[-0.0017913264033, 0.0020782814353]
```

get_orbit(*idx=None*, *val=None*, *outfile=None*)
calculate the orbit with given configurations

Parameters

- **idx** – (idx_hcor, idx_vcor), tuple of list of indices of h/v cors
- **val** – values for each correctos, h/v
- **outfile** – filename to save the data

Returns tuple of zpos, env_x, env_y, machine

hcors

latfile

optdriver

plot(*outfile=None*, *figsize=(10, 8)*, *dpi=120*, ***kws*)
show orbit

Parameters

- **outfile** – output file of dakota

- **figsize** – figure size, (h, w)
- **dpi** – figure dpi

ref_flag

ref_x0

ref_y0

run(*mpi=False*, *np=None*)
run optimization

Parameters

- **mpi** – if True, run DAKOTA in parallel mode, False by default
- **np** – number of processes to use, only valid when `mpi` is True

set_bpms(*bpm=None*)
set BPMs

Parameters **bpm** – list of bpm indices, if None, use all BPMs

set_cors(*cor=None*, *hcor=None*, *vcor=None*)
set correctors, if cor, hcor and vcor are None, use all correctors if cor is not None, use cor, ignore hcor and vcor

Parameters

- **cor** – list of corrector indices, hcor, vcor,...
- **hcor** – list of horizontal corrector indices
- **vcor** – list of vertical corrector indices

set_environ(*environ=None*)
setup environment block, that is setup `oc_environ`

Parameters **environ** – `DakotaEnviron` object, automatically setup if not defined

set_interface(*interface=None*, ***kws*)
setup interface block, that is setup `oc_interface` should be ready to invoke after `set_cors` and `set_bpms`

Parameters **interface** – `DakotaInterface` object, automatically setup if not defined

set_method(*method=None*)
setup method block, that is setup `oc_method`

Parameters **method** – `DakotaMethod` object, automatically setup if not defined

set_model(*model=None*, ***kws*)
setup model block, that is setup `oc_model`

Parameters **model** – `DakotaModel` object, automatically setup if not defined

set_ref_x0(*ref_arr=None*)
set reference orbit in x, if not set, use 0s

Parameters **ref_arr** – array of reference orbit values size should be the same number as selected BPMs

set_ref_y0(*ref_arr=None*)
set reference orbit in y, if not set, use 0s

Parameters **ref_arr** – array of reference orbit values size should be the same number as selected BPMs

set_responses(*responses=None*, ***kws*)
setup responses block, that is setup `oc_responses`

Parameters **responses** – `DakotaResponses` object, automatically setup if not defined

set_variables(*plist=None*, *initial=0.0001*, *lower=-0.01*, *upper=0.01*)
setup variables block, that is setup `oc_variables` should be ready to invoke after `set_cors()`

Parameters

- **plist** – list of defined parameters (`DakotaParam` object), automatically setup if not defined
- **initial** – initial values for all variables, only valid when plist is None
- **lower** – lower bound for all variables, only valid when plist is None
- **upper** – upper bound for all variables, only valid when plist is None

simple_run(*method='cg'*, *mpi=None*, *np=None*, ***kws*)
run optimization after `set_bpms()` and `set_cors()`, by using default configuration and make full use of computing resources.

Parameters

- **method** – optimization method, 'cg', 'ps', 'cg' by default
- **mpi** – if True, run DAKOTA in parallel mode, False by default
- **np** – number of processes to use, only valid when `mpi` is True
- **kws** – keyword parameters valid keys:
  - step: gradient step, 1e-6 by default
  - iternum: max iteration number, 20 by default
  - evalnum: max function evaulation number, 1000 by default

vcors

test_dakotaoc1()

test_dakotaoc2()

genopt.dakutils module

module contains utilities:

- generate dakota input files
- extract data from output files

Tong Zhang <zhangt@frib.msu.edu>

2016-10-17 09:19:25 AM EDT

class DakotaEnviron(*tabfile=None*, ***kws*)
Bases: `object`

create datako environment for `environment` block

Parameters

- **tabfile** – tabular file name, by default not save tabular data
- **kws** – other keyword parameters

Example

```
>>> # default
>>> oc_environ = DakotaEnviron()
>>> print oc_environ.get_config()
[]
>>> # define name of tabular file
>>> oc_environ = DakotaEnviron(tabfile='tmp.dat')
>>> print oc_environ.get_config()
['tabular_data', " tabular_data_file 'tmp.dat'"]
>>>
```

get_config(*rtype='list'*)
get responses configuration for dakota input block

Parameters **rtype** – 'list' or 'string'

Returns dakota responses input

class DakotaInput(***kws*)
Bases: object

template of dakota input file, field could be overriden by providing additional keyword arguments,

Parameters **kws** – keyword arguments, valid keys are dakota directives

Example

```
>>> dak_inp = DakotaInput(method=["max_iterations = 500",
                                  "convergence_tolerance = 1e-7",
                                  "conmin_frcg",])
>>>
```

set_template(*name='oc'*)

write(*infile=None*)
write all the input into file, as dakota input file

Parameters **infile** – fullname of input file, if not defined, infile will be assigned as 'dakota.in' in current working directory

class DakotaInterface(*mode='fork'*, *driver='flamedriver_oc'*, *latfile=None*, *bpms=None*, *hcors=None*, *vcors=None*, *ref_x0=None*, *ref_y0=None*, *ref_flag=None*, ***kws*)
Bases: object

create dakota interface for `interface` block

Parameters

- **mode** – 'fork' or 'direct' (future usage)
- **driver** – analysis driver, external ('fork') executable file internal ('direct') executable file
- **latfile** – file name of (flame) lattice file
- **bpms** – array of selected BPMs' id
- **hcors** – array of selected horizontal (x) correctors' id
- **vcors** – array of selected vertical (y) correctors' id
- **ref_x0** – array of BPM readings for reference orbit in x, if not defined, use 0s
- **ref_y0** – array of BPM readings for reference orbit in y, if not defined, use 0s

- **ref_flag** – string flag for objective functions:
  1. "x": $\sum \Delta x^2$, $\Delta x = x - x_0$;
  2. "y": $\sum \Delta y^2$, $\Delta y = y - y_0$;
  3. "xy": $\sum \Delta x^2 + \sum \Delta y^2$.
- **kws** – keyword parameters, valid keys: e.g.: * deactivate, possible value: 'active_set_vector'

Note: mode should be set to be 'direct' when the analysis drivers are built with dakota library, presently, 'fork' is used.

Example

```
>>> # for orbit correction
>>> bpms = [1,2,3] # just for demonstration
>>> hcors, vcors = [1,3,5], [2,4,6]
>>> latfile = 'test.lat'
>>> oc_interface = DakotaInterface(mode='fork',
>>>                                driver='flamedriver_oc',
>>>                                latfile=latfile
>>>                                bpms=bpms, hcors=hcors, vcors=vcors,
>>>                                ref_x0=None, ref_y0=None,
>>>                                ref_flag=None,
>>>                                deactivate='active_set_vector')
>>> # add extra configurations
>>> oc_interface.set_extra(p1='v1', p2='v2')
>>> # get configuration
>>> config_str = oc_interface.get_config()
>>>
```

bpms

driver

get_config(*rtype='list'*)

get interface configuration for dakota input block

Parameters **rtype** – 'list' or 'string'

Returns dakota interface input

hcors

latfile

mode

ref_flag

ref_x0

ref_y0

set_extra(***kws*)

add extra configurations

vcors

class DakotaMethod(*method='cg'*, *iternum=20*, *tolerance=0.0001*, ***kws*)

Bases: object

create dakota method for method block

Parameters

- **method** – method name, ‘cg’ by default, all possible choices: ‘cg’, ‘ps’
- **iternum** – max iteration number, 20 by default
- **tolerance** – convergence tolerance, 1e-4 by default
- **kws** – other keyword parameters

Example

```
>>> # default
>>> oc_method = DakotaMethod()
>>> print oc_method.get_config()
['conmin_frcg', ' convergence_tolerance 0.0001', ' max_iterations 20']
>>> # define method with pattern search
>>> oc_method = DakotaMethod(method='ps')
>>> print oc_method.get_config()
['coliny_pattern_search', ' contraction_factor 0.75', ' max_function_evaluations 500',
 ' solution_accuracy 0.0001', ' exploratory_moves basic_pattern',
 ' threshold_delta 0.0001', ' initial_delta 0.5', ' max_iterations 100']
>>> # modify options of pattern search method
>>> oc_method = DakotaMethod(method='ps', max_iterations=200, contraction_factor=0.8)
>>> print oc_method.get_config()
['coliny_pattern_search', ' contraction_factor 0.8', ' max_function_evaluations 500',
 ' solution_accuracy 0.0001', ' exploratory_moves basic_pattern',
 ' threshold_delta 0.0001', ' initial_delta 0.5', ' max_iterations 200']
>>> # conmin_frcg method
>>> oc_method = DakotaMethod(method='cg')
>>> print oc_method.get_config()
['conmin_frcg', ' convergence_tolerance 0.0001', ' max_iterations 20']
>>> # modify options
>>> oc_method = DakotaMethod(method='cg', max_iterations=100)
>>> print oc_method.get_config()
['conmin_frcg', ' convergence_tolerance 0.0001', ' max_iterations 100']
>>>
```

get_config(*rtype='list'*)
get method configuration for dakota input block

Parameters **rtype** – ‘list’ or ‘string’

Returns dakota method input

get_default_method(*method*)
get default configuration of some method

Parameters **method** – method name, ‘cg’ or ‘ps’

Returns dict of configuration

method(*method*)
return method configuration

Parameters **method** – method stirng name, ‘cg’ or ‘ps’

Returns list of method configuration

class DakotaModel(***kws*)
Bases: `object`

create dakota model for `model` block

get_config(*rtype='list'*)
get model configuration for dakota input block

Parameters **rtype** – ‘list’ or ‘string’

Returns dakota model input

class DakotaParam(*label*, *initial=0.0*, *lower=-10000000000.0*, *upper=10000000000.0*)
Bases: `object`

create dakota variable for `variables` block

Parameters

- **label** – string to represent itself, e.g. `x001`, it is recommended to annotate the number with the format of `%03d`, i.e. `1 --> 001`, `10 --> 010`, `100 --> 100`, etc.
- **initial** – initial value, 0.0 by default
- **lower** – lower bound, -1.0e10 by default
- **upper** – upper bound, 1.0e10 by default

initial

label

lower

upper

class DakotaResponses(*nfunc=1*, *gradient=None*, *hessian=None*, ***kws*)
Bases: `object`

create dakota responses for `responses` block

Parameters

- **nfunc** – num of objective functions
- **gradient** – gradient type: 'analytic' or 'numerical'
- **hessian** – hessian configuration
- **kws** – keyword parameters for gradients and hessians valid keys: any available for responses among which key name of 'grad' is for gradients configuration, the value should be a dict (future)

Example

```
>>> # default responses:
>>> response = DakotaResponses()
>>> print response.get_config()
['num_objective_functions = 1', 'no_gradients', 'no_hessians']
>>>
>>> # responses with analytic gradients:
>>> response = DakotaResponses(gradient='analytic')
>>> print response.get_config()
['num_objective_functions = 1', 'analytic_gradients', 'no_hessians']
>>>
>>> # responses with numerical gradients, default configuration:
>>> oc_responses = DakotaResponses(gradient='numerical')
>>> print oc_responses.get_config()
['num_objective_functions = 1', 'numerical_gradients', ' method_source dakota',
 ' interval_type forward', ' fd_gradient_step_size 1e-06', 'no_hessians']
>>>
>>> # responses with numerical gradients, define step:
>>> oc_responses = DakotaResponses(gradient='numerical', step=2.0e-7)
>>> print oc_responses.get_config()
['num_objective_functions = 1', 'numerical_gradients', ' method_source dakota',
 ' interval_type forward', ' fd_gradient_step_size 2e-07', 'no_hessians']
>>>
>>> # given other keyword parameters:
>>> oc_responses = DakotaResponses(gradient='numerical', step=2.0e-7, k1='v1', k2='v2')
>>> print oc_responses.get_config()
['num_objective_functions = 1', 'numerical_gradients', ' method_source dakota',
 ' interval_type forward', ' fd_gradient_step_size 2e-07', 'no_hessians', 'k2 = v2', 'k1 = v1']
>>>
```

get_config(*rtype='list'*)
get responses configuration for dakota input block

Parameters **rtype** – 'list' or 'string'

Returns dakota responses input

gradients(*type=None*, *step=1e-06*, ***kws*)
generate gradients configuration

Parameters

- **type** – 'numerical' or 'analytic' (default)
- **step** – gradient step size, only valid when type is numerical
- **kws** – other keyword parameters

Returns list of configuration

generate_latfile(*machine*, *latfile='out.lat'*)
Generate lattice file for the usage of FLAME code

Parameters

- **machine** – flame machine object
- **latfile** – file name for generated lattice file, 'out.lat' by default

Returns None if failed to generate lattice file, or the out file name

Example

```
>>> from flame import Machine
>>> latfile = 'test.lat'
>>> m = Machine(open(latfile))
>>> outfile1 = generate_latfile(m, 'out1.lat')
>>> m.reconfigure(80, {'theta_x': 0.1})
>>> outfile2 = generate_latfile(m, 'out2.lat')
>>>
```

Warning: To get element configuration only by `m.conf(i)` method, where `m` is `flame.Machine` object, `i` is element index, when some re-configuring operation is done, `m.conf(i)` will be update, but `m.conf()["elements"]` remains with the initial value.

get_opt_results(*outfile='dakota.out'*, *rtype='dict'*)
extract optimized results from dakota output

Parameters

- **outfile** – file name of dakota output file, 'dakota.out' by default
- **rtype** – type of returned results, 'dict' or 'list', 'dict' by default

Returns by default return a dict of optimized results with each item of the format like "x1":0.1, etc., or if rtype='list', return a list of values, when the keys are ascend sorted.

Example

```
>>> opt_vars = get_opt_results(outfile='flame_oc.out', rtype='dict'):
>>> print(opt_vars)
{'x2': 0.0020782814353, 'x1': -0.0017913264033}
>>> opt_vars = get_opt_results(outfile='flame_oc.out', rtype='list'):
>>> print(opt_vars)
[-0.0017913264033, 0.0020782814353]
```

random_string(*length=8*)
generate random string with given length

Parameters **length** – string length, 8 by default

Returns random strings with defined length

test_dakotaenviron()

test_dakotainput()

test_dakotainterface()

test_dakotamethod()

test_dakotamodel()

test_dakotaparam()

test_dakotaresponses()

test_get_opt_results()

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

g

S

T

U

V

W