FRIB

genopt Documentation

Release 0.1.0

Tong Zhang

Jul 31, 2018

Contents

genopt Python package

genopt: general multi-dimensional optimization

Author Tong Zhang

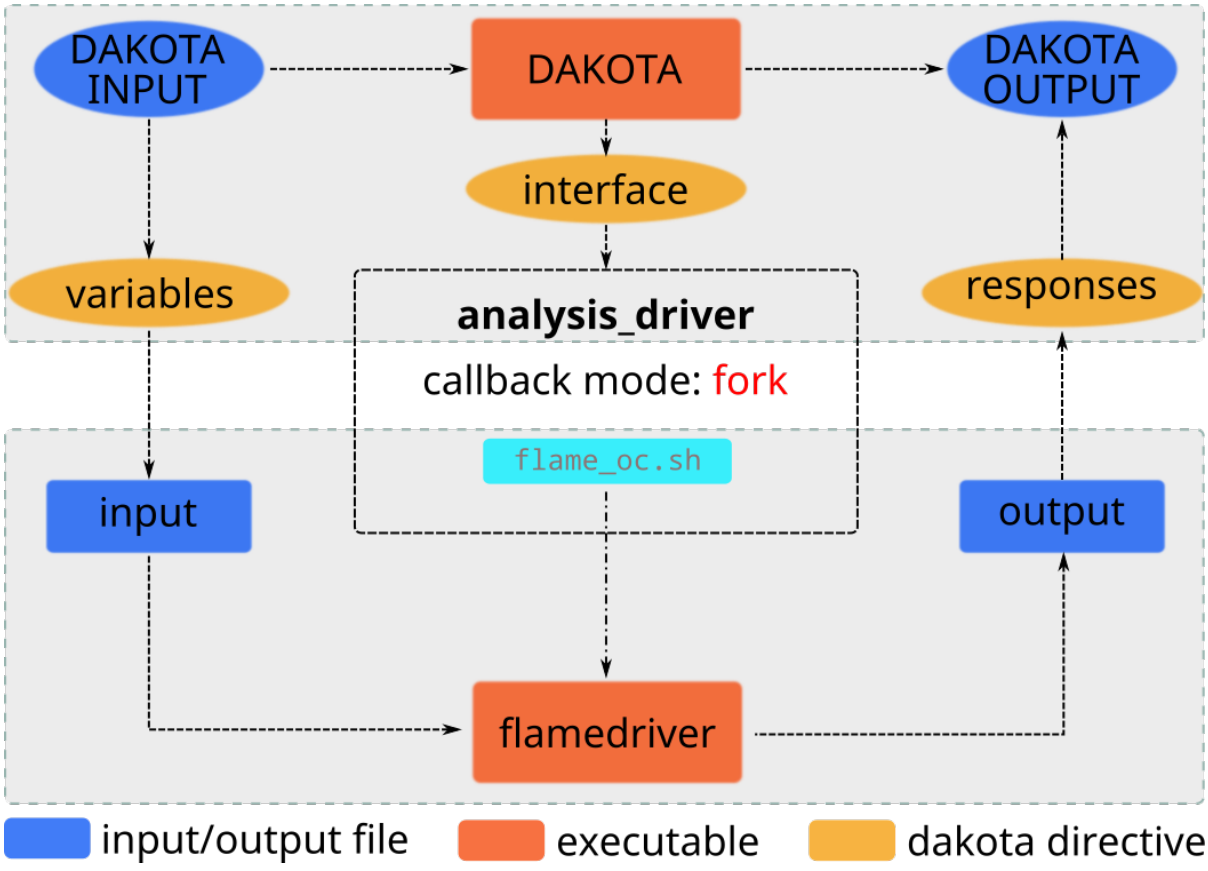
E-mail zhangt@frib.msu.edu

CHAPTER 1

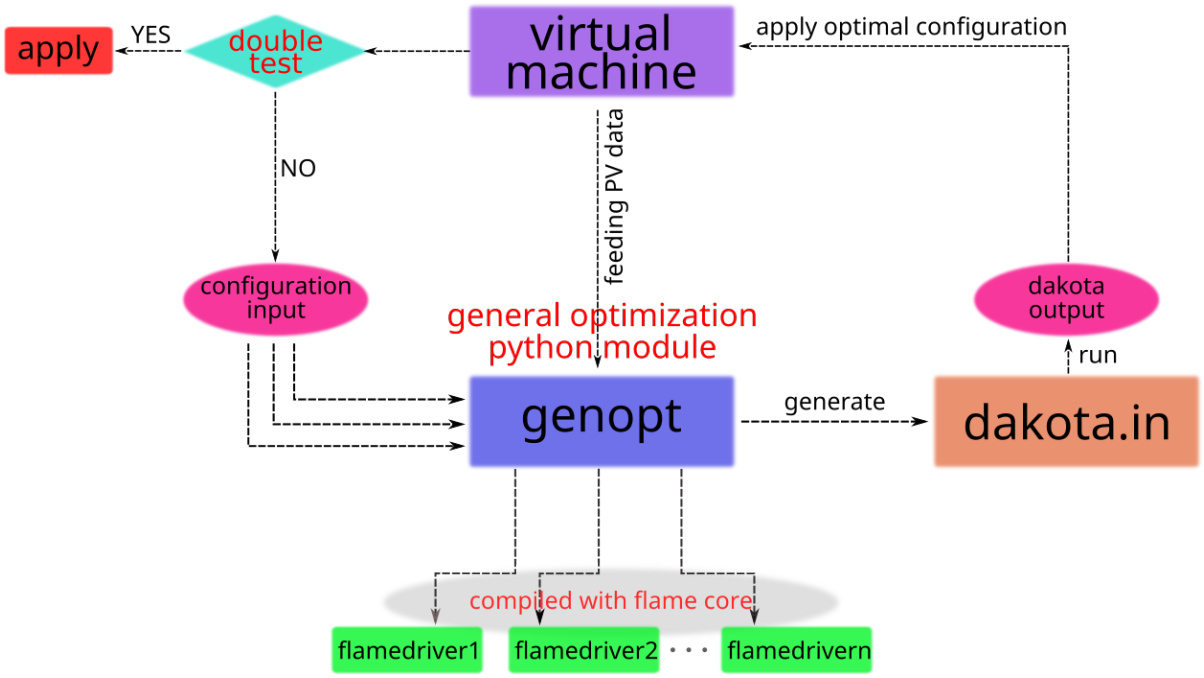
Introduction

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 2

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.

2.1 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;
3. `echo`: if False, will not generate output when optimizing, the same for `run()`;

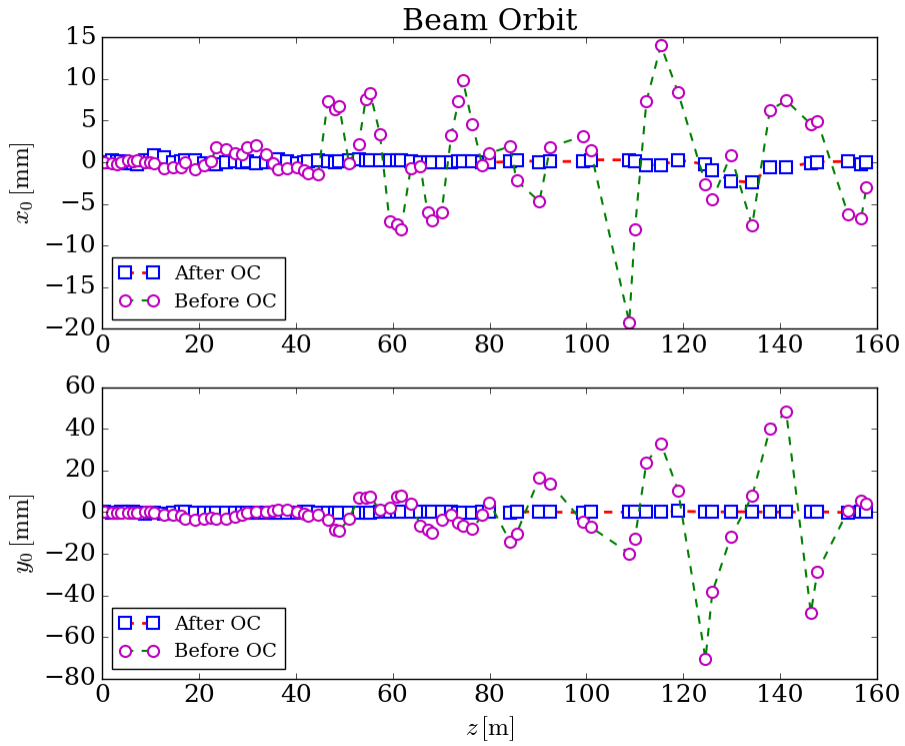
- **gradient descent, i.e. `method=cg`:**
  1. `iternum`: max iteration number, 20 by default;
  2. `step`: forward gradient step size, 1e-6 by default;
- **pattern search, i.e. `method=ps`:**
  1. `iternum`: max iteration number, 20 by default;
  2. `evalnum`: max function evaulation number, 1000 by 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:



2.2 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.

Warning: All elements could be treated as *BPMs*, see `set_bpms()`, set `pseudo_all=True` option will use all elements as monitors.

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

2.3 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.

2.4 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.

2.4.1 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)
```

2.4.2 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',
```

(continues on next page)

(continued from previous page)

```
                    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")

2.4.3 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)
```

2.4.4 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.

2.4.5 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)
```

2.5 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.

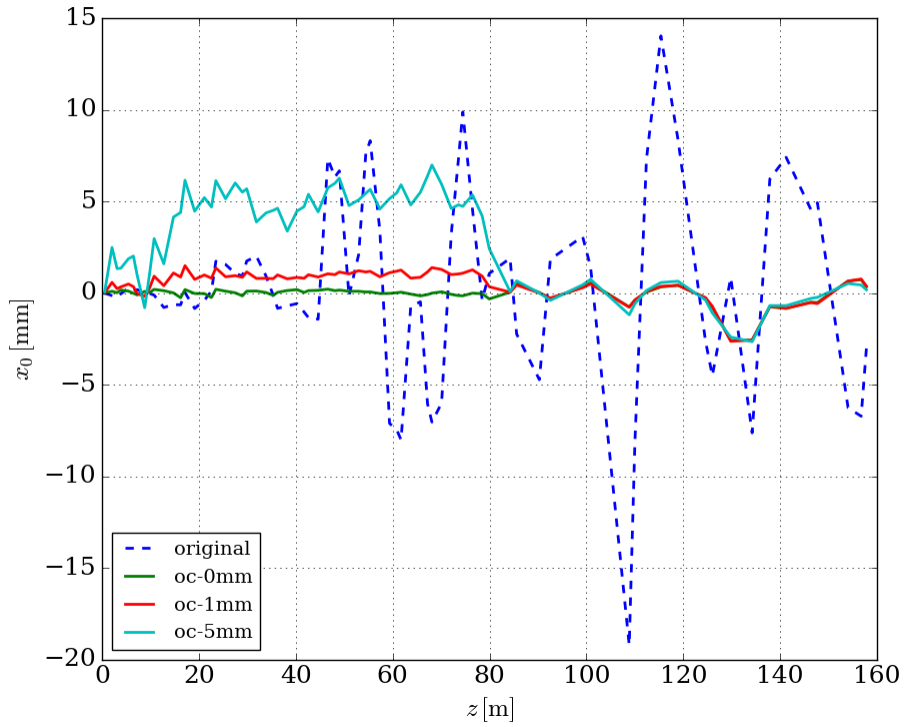
```
import os
import genopt

""" orbit correction demo
```

(continues on next page)

(continued from previous page)

```
"""
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 = genopt.DakotaMethod(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.hcors, oc_ins.vcors), oc_ins.get_opt_results(),
                 outfile='orbit.dat')

# plot
#oc_ins.plot()
```

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



2.6 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)
```

2.6.1 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,
```

(continues on next page)

(continued from previous page)

```
 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'`.

2.6.2 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.

2.6.3 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.

Note: `generate_latfile()` in module `genopt.dakutils` could be used to generate lattice file from `flame.Machine` object.

CHAPTER 3

API

3.1 genopt package

3.1.1 Submodules

genopt.dakopt module

genopt.dakutils module

CHAPTER 4

Indices and tables

- genindex
- modindex
- search