pyOpt Quickguide

Release 1.2.0

Peter W. Jansen, Ruben E. Perez

July 31, 2014

Contents

1 pyOpt Quick Reference Guide

This is a quick guide to begin solving optimization problems with pyOpt.

1.1 Optimization Problem Definition

pyOpt is design to solve general constrained nonlinear optimization problems:

min f(x) w.r.t. x

s.t. g_j(x) = 0, j = 1, ..., m_e g_j(x) <= 0, j = m_e + 1, ..., m

x_i_L <= x_i <= x_i_U, i = 1, ..., n

where:

- x is the vector of design variables
- f(x) is a nonlinear function
- g(x) is a linear or nonlinear function
- n is the number of design variables
- m_e is the number of equality constraints
- m is the total number of constraints (number of equality constraints: m_i = m - m_e)

1.2 Optimization Class

Instantiating an Optimization Problem:

```
>>> opt_prob = Optimization('name',obj_fun)
```

Notes on Objective Functions:

General Objective Function Template:

```
def obj_fun(x, *args, **kwargs):

  fail = 0
  f = function(x,*args,**kwargs)
  g = function(x,*args,**kwargs)

  return f,g,fail
```

where:

f - objective value

g - array (or list) of constraint values

fail - 0 for successful function evaluation - 1 for unsuccessful function evaluation (test must be provided by user)

If the Optimization problem is unconstrained, g must be returned as an empty list or array: g = [ ]

Inequality constraints are handled as <=.

Assigning Objective:

```
>>> opt_prob.addObj('name', value=0.0, optimum=0.0)
```

Assigning Design Variables:

Single Design variable:

```
>>> opt_prob.addVar('name', type='c', value=0.0, lower=-inf, upper=inf,
choices=listochoices)
```

A Group of Design Variables:

```
>>> opt_prob.addVarGroup('name', numerinGroup, type='c', value=value,
 lower=lb, upper=up,choices=listochoices)
```

where:

value,lb,ub - (float or int or list or 1Darray).

Supported Types:

'c' - continuous design variable.

'i' - integer design variable.

'd' - discrete design variable (based on choices, e.g.: list/dict of materials).

Assigning Constraints:

Single Constraint:

```
>>> opt_prob.addCon('name', type='i', lower=-inf, upper=inf, equal=0.0)
```

A Group of Constraints:

```
>>> opt_prob.addConGroup('name', numberinGroup, type='i', lower=lb,
 upper=up, equal=eq)
```

where:

lb,ub,eq - (float or int or list or 1Darray).

Supported Types:

'i' - inequality constraint.

'e' - equality constraint.

1.3 Optimizer Class

Instantiating an Optimizer (e.g.: Snopt):

```
>>> opt = pySNOPT.SNOPT()
```

Setting Optimizer Options:

either during instanciation:

```
>>> opt = pySNOPT.SNOPT(options={'name':value,...})
```

or one by one:

```
>>> opt.setOption('name',value)
```

Getting Optimizer Options/Attributes:

```
>>> opt.getOption('name')
```

```
>>> opt.ListAttributes()
```

1.4 Optimizing

Solving the Optimization Problem:

```
>>> opt(opt_prob, sens_type='FD', disp_opts=False, sens_mode='',*args, **kwargs)
```

disp_opts - flag for displaying the options in the solution output.

sens_type - sensitivity type. - 'FD' = finite differences. - 'CS' = complex step. - grad_function = user provided function.

format: grad_function(x,f,g) returns g_obj,g_con,fail

sens_mode - parallel sensitivity flag (''-serial,'pgc'-parallel).

Additional arguments and keyword arguments (e.g.: parameters) can be passed to the objective function.

Output:

- Prompt output of the Optimization problem with initial values:

```
>>> print opt_prob
```

- Prompt output of specific solution of the Optimization problem:

```
>>> print opt_prob._solutions[key]
```

key - index in order of optimizer call.

- File output of the Optimization problem:

```
>>> opt_prob.write2file(outfile='', disp_sols=False, solutions=[])
```

where:

outfile - (filename, fileinstance, default=opt_prob name[0].txt).

disp_sols - True will display all the stored solutions.

solutions - list of indices of stored solutions to display.

Output as Input:

The solution can be used directly as an optimization problem for refinement by the same or a new optimizer:

```
>>> optimizer(opt_prob._solutions[key])
```

key - index in order of optimizer call.

The new solution will be stored as a sub-solution of the previous solution:

e.g.: print opt_prob._solutions[key]._solutions[nkey]

History and Hot Start:

The history flag stores all function evaluations from an optimizer in binary format in a .bin and .cue file:

```
>>> optimizer(opt_prob, store_hst=True)
```

True - uses default print name for the file names str - filename

The binary history file can be used to hot start the optimizer if the optimization was interrupted. The flag needs the filename of the history (True will use the default name).

```
>>> optimizer(opt_prob, store_hst=True, hot_start=True)
```

If the store history flag is set, the same as the hot start flag a temporary file will be created during the run, and the original file will be overwritten at the end.

For hot start to work properly, all options must be the same as when the history was created.

1.5 Parallel Processing

PyOpt takes advantage of parallel processing using mpi4py and MPI (via mpirun, srun, etc).

Different parallelization options are available depending on the optimizer being used as follows:

Gradient-based optimizers:

- Parallel Objective Analysis, were the optimizer is used to optimize functions for which a parallel analysis are being done. This option is defined at the optimizer instanciation and used at run time:

```
>>> snopt = SNOPT(pll_type='POA')
```

- Parallel Gradients, were gradient function evaluations are done in parallel. This option is defined at run time:

```
>>> snopt(opt_prob,sens_mode='pgc')
```

Population-based optimizers:

- Parallel Objective Analysis, as above.
- Static Process Management (SPM), were function evaluations are parallelized using fixed processors allocations. This option is defined at the optimizer instanciation and used at run time:

  ```
  >>> midaco = MIDACO(pll_type='SPM')
  ```

  Note: this option is currently implemented only for the ALPSO and MIDACO optimizers.
- Dynamic Process Management (DPM), were function evaluations are parallelized using dynamic processors allocations. This option is defined at the optimizer instanciation and used at run time:

  ```
  >>> alpso = ALPSO(pll_type='DPM')
  ```

  Note: this option is currently implemented only for the ALPSO optimizer.