pyOptSparse Documentation

Release dev

Gaetan Kenway

May 31, 2020

Contents

1 Introduction **3**

2 Changes to pyOptSparse **5**

3 Installation **7**
- 3.1 Requirements . . . 7
- 3.2 Building . . . 8

4 Tutorial **11**

5 Guide **15**
- 5.1 Design Variables . . . 16
- 5.2 Constraints . . . 16
- 5.3 Objectives . . . 18

6 Optimizers **19**
- 6.1 SNOPT . . . 19
- 6.2 IPOPT . . . 20
- 6.3 SLSQP . . . 22
- 6.4 NLPQLP . . . 23
- 6.5 FSQP . . . 23
- 6.6 NSGA2 . . . 24
- 6.7 PSQP . . . 25
- 6.8 ParOpt . . . 26
- 6.9 CONMIN . . . 27
- 6.10 ALPSO . . . 28

7 pyOptSparse API Documentation **29**
- 7.1 Optimziation . . . 29
- 7.2 Optimzer . . . 32
- 7.3 Constraint . . . 33
- 7.4 Variable . . . 33
- 7.5 Gradient . . . 33
- 7.6 History . . . 34

8 Postprocessing **39**
- 8.1 Requirements . . . 39

8.2 Usage 39
8.3 Features 40
8.4 Parsing SNOPT Printout files 41

Index **43**

PYthon OPTimization (Sparse) Framework

pyOptSparseSparse is a replacement for pyOpt.

CHAPTER 1

Introduction

`pyOptSparse` is fairly extensive modification to the original pyOpt framework originally written by Dr. Ruben E. Perez and Peter W. Jansen. It is not backwards compatible with pyOpt and thus optimization scripts will need to be modified to use `pyOptSparse`.

The original goal of pyOpt was to create an object-oriented framework for formulating and solving nonlinear constrained optimization problems. This goal remains the same with `pyOptSparse`.

From the pyOpt README some of the main pyOpt features are:

- Object-oriented development maintains independence between the optimization problem formulation and its solution by different optimizers
- Allows for easy integration of gradient-based, gradient-free, and population-based optimization algorithms
- Interfaces both open source as well as industrial optimizers
- Ease the work required to do nested optimization and provides automated solution refinement
- On parallel systems it enables the use of optimizers when running in a mpi parallel environment, allows for evaluation of gradients in parallel, and can distribute function evaluations for gradient-free optimizers
- Optimization solution histories can be stored during the optimization process. A partial history can also be used to warm-restart the optimization

CHAPTER 2

Changes to pyOptSparse

The following list summarizes some of the changes/improvements made to pyOpt to get `pyOptSparse`:

- Elimination of n^2 scaling behaviour when adding large numbers of design variables (>10,000)
- Proper handling of all optimizers when run in parallel environment (Only a single optimization instance is run, not nProc instances)
- More flexible return specification of constraints
- More flexible return specification of constraint *gradients*
- Complete elimination of gradient indexing errors using dictionary based returns
- Substantial improvement of optimization script robustness through indexing elimination
- Automatic assembly of sparse jacobians with both dense and sparse subblocks
- Automatic conversion of sparse jacobian to dense for optimizers that cannot handle sparse constraints
- User specification of design variable scaling
- User specificaiton of constraint scaling
- Design variable returns in dictionary format only
- Specification of linear constraints, dense or sparse
- Sparse non-linear jacobians
- New history file format. Uses naitve python shelve database format. Much easier to retrieve optimization data after optimization.
- Fixed hot start bug where first call to user functions is a gradient. It is now guaranteed, that the first call is to the function evlaution, not the gradient.
- Various bug fixes in SNOPT
- Constraints can be *any* order, independent of what the individual optimizer requires
- Python 3.x compatibility

CHAPTER 3

Installation

3.1 Requirements

pyOpt has the following dependencies:

- Python 2.7+ (Python 3.2+)
- Numpy 1.16+
- Scipy 1.3+
- six 1.13
- sqlitedict 1.6.0
- c/FORTRAN compiler (compatible with f2py)

Please make sure these are installed and available for use. In order to use NSGA2 and NOMAD, SWIG (v1.3+) is also required. If those optimizers are not needed, then you do not need to install SWIG. Simply comment out the corresponding lines in `pyoptsparse/pyoptsparse/setup.py` so that they are not compiled.

To facilitate the installation of Python dependencies, there is a file called `requirements.txt` which can be used together with `pip`. Simply type

```
pip install -r requirements.txt
```

In the future, we hope to make the package pip-installable so that dependencies can be managed more easily.

Note:

- In Windows MinGW is recommended if c/FORTRAN compilers are not available
- In Linux, the python header files (python-dev) are also required.
- Compatibility on Windows 64bit has not been tested

3.2 Building

The easiest and recommended way to install pyOptSparse is with `pip`. First clone the repository into a location which is not on the `$PYTHONPATH`, for example `~/packages`. Then in the root `pyoptsparse` folder type:

```
pip install -e .
```

For those not using `conda`, a user install is needed:

```
pip install -e . --user
```

Two other ways of installing `pyOptSparse` are possible, both require running `setup.py` manually. The first approach is to install the package inplace, similar to the `-e` flag used above. This means that no source code is actually copied to a `site-packages` folder, meaning modifying the source code would have an immediate effect, without the need to re-install the code. From the root directory run:

```
>>> python setup.py build_ext --inplace
```

To use pyOptSparse in this case, the user should add the path of the root directory to the user's `$PYTHONPATH` enviromental variable. For example, if the pyoptsparse directory is located at:

```
/home/<user>/packages/pyoptsparse
```

The required line in the `.bashrc` file would be:

```
export PYTHONPATH=$PYTHONPATH:/home/<user>/packages/pyoptsparse
```

To install the `pyOptSparse` package in a folder on the Python search path (usually in a python site-packages or dist-packages folder) run:

```
>>> python setup.py install --user
```

This will install the package to `~/.local` which is typically found automatically by Python. If a system wide install is desired the command to run would be (requiring root access):

```
>>> sudo python setup.py install
```

Warning: Remember to delete the `build` directory first when re-building from source.

Notes:

- You may want to uninstall any previous version of pyOpt before installing a new version, as there may be conflicts.
- Some optimizers are licensed and their sources are not included with this distribution. To use them, please request their sources from the authors as indicated in the optimizer LICENSE files, and place them in their respective source folders before installing the package. Refer to specific optimizer pages for additional information.
- In Windows, if MinGW is used make sure to install for it the C, C++, and Fortran compilers and run:

  ```
  >>> python setup.py install --compiler=mingw32
  ```

- By default pyOpt will attempt to use compilers available on the system. To get a list of available compilers and their corresponding flag on a specific system use:

```
>>> python setup.py build --help-fcompiler
```

- To see a list of all available `setup.py` options for building run

```
>>> python setup.py build --help
```

- In macOS, you may need to force an update of gcc using 'brew uninstall gcc' followed by a fresh installation of gcc using 'brew install gcc' as 'brew upgrade gcc' can be insufficient if you have recently updated your macOS version.

CHAPTER 4

Tutorial

The following shows how to get started with pyOptSparse by solving Schittkowski's TP37 constrained problem. First, we show the complete program listing and then go through each statement line by line:

```
import pyoptsparse
def objfunc(xdict):
    x = xdict['xvars']
    funcs = {}
    funcs['obj'] = -x[0]*x[1]*x[2]
    conval = [0]*2
    conval[0] = x[0] + 2.*x[1] + 2.*x[2] - 72.0
    conval[1] = -x[0] - 2.*x[1] - 2.*x[2]
    funcs['con'] = conval
    fail = False

    return funcs, fail

optProb = pyoptsparse.Optimization('TP037', objfunc)
optProb.addVarGroup('xvars',3, 'c',lower=[0,0,0], upper=[42,42,42], value=10)
optProb.addConGroup('con',2, lower=None, upper=0.0)
optProb.addObj('obj')
print optProb
opt = pyoptsparse.SLSQP()
sol = opt(optProb, sens='FD')
print sol
```

Start by importing the pyOptSparse package:

```
>>> import pyoptsparse
```

Next we define the objective function that takes in the design variable *dictionary* and returns a *dictionary* containing the constraints and objective, as well as a (boolean) flag indicating if the objective function evaluation was successful. For the TP37, the objective function is a simple analytic function:

```
def objfunc(xdict):
    x = xdict['xvars']
    funcs = {}
    funcs['obj'] = -x[0]*x[1]*x[2]
    conval = [0]*2
    conval[0] = x[0] + 2.*x[1] + 2.*x[2] - 72.0
    conval[1] = -x[0] - 2.*x[1] - 2.*x[2]
    funcs['con'] = conval
    fail = False

    return funcs, fail
```

Notes:

1. The `xdict` variable is a dictionary whose keys are the names from each `addVar()` and `addVarGroup()` call. The line:

   ```
   x = xdict['xvars']
   ```

   retrieves an array of length 3 which are all the variables for this optimization.

2. The line:

   ```
   conval = [0]*2
   ```

   creates a list of length 2, which stores the numerical values of the two constraints. The `funcs` dictionary return must contain keys that match the constraint names from `addCon` and `addConGroup` as well as the objectives from `addObj` calls. This is done in the following calls:

   ```
   funcs['obj'] = -x[0]*x[1]*x[2]
   funcs['con'] = conval
   ```

Now the optimization problem can be initialized:

```
>>> optProb = Optimization('TP037', objfunc)
```

This creates an instance of the optimization class with a name and a reference to the objective function. To complete the setup of the optimization problem, the design variables and constraints need to be defined.

Design variables and constraints can be added either one-by-one or as a group. Adding variables by group is generally recommended for related variables:

```
>>> optProb.addVarGroup('xvars', 3, 'c', lower=[0,0,0], upper=[42,42,42], value=10)
```

This calls adds a group of 3 variables with name 'xvars'. The variable bounds (side constraints) are 0 for the lower bounds, 42 for the upper bounds. The inital values for each variable is 10.0

Now, we must add the constraints. Like design variables, these may be added individually or by group. It is recommended that related constraints are added by group where possible:

```
>>> optProb.addConGroup('con',2, lower=None, upper=0.0)
```

This call adds two variables with name 'con'. There is no lower bound for the variables and the upper bound is 0.0.

We must also assign the the key value for the objective using the `addObj()` call:

```
>>> optProb.addObj('obj')
```

The optimization problem can be printed to verify that it is setup correctly:

```
>>> print optProb
```

To solve an optimization problem with `pyOptSparse` an optimizer must be initialized. The initialization of one or more optimizers is independent of the initialization of any number of optimization problems. To initialize `SLSQP`, which is an open-source, sequential least squares programming algorithm that comes as part of the pyOptSparse package, use:

```
>>> opt = pyoptsparse.SLSQP()
```

This initializes an instance of `SLSQP` with the default options. The setOption() method can be used to change any optimizer specific option, for example the internal output flag of `SLSQP`:

```
>>> opt.setOption('IPRINT', -1)
```

Now TP37 can be solved using `SLSQP` and for example, `pyOptSparse`'s automatic finite difference for the gradients:

```
>>> sol = opt(optProb, sensType='FD')
```

We can print the solution objection to view the result of the optimization:

```
>>> print sol

  TP037
================================================================================


        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                    0.0256
       User Objective Time :       0.0003
       User Sensitivity Time :     0.0021
       Interface Time :            0.0226
       Opt Solver Time:            0.0007
    Calls to Objective Function :      23
    Calls to Sens Function :            9

    Objectives:
        Name        Value        Optimum
           f                0             0

        Variables (c - continuous, i - integer, d - discrete):
           Name      Type       Value       Lower Bound  Upper Bound
          xvars_0     c          24.000000       0.00e+00     4.20e+01
          xvars_1     c          12.000000       0.00e+00     4.20e+01
          xvars_2     c          12.000000       0.00e+00     4.20e+01

        Constraints (i - inequality, e - equality):
        Name     Type                    Bounds
            con          i        1.00e-20 <= 0.000000 <= 0.00e+00
            con          i        1.00e-20 <= 0.000000 <= 0.00e+00

--------------------------------------------------------------------------------
```

CHAPTER 5

Guide

pyOptSparse is designed to solve general, constrained nonlinear optimization problems of the form:

$$\begin{aligned} \min \quad & f(x) \\ \text{with respect to} \quad & x \\ \text{such that} \quad & g_{j,\text{L}} \leq g_j(x) \leq g_{j,\text{U}}, \quad j = 1, ..., m \\ & x_{i,\text{L}} \leq x_i \leq x_{i,\text{U}}, \quad i = 1, ..., n \end{aligned}$$

where: x is the vector of n design variables, $f(x)$ is a nonlinear function, and $g(x)$ is a set of m nonlinear functions.

Equality constraints are specified using the same upper and lower bounds for the constraint. ie. $g_{j,\text{L}} = g_{j,\text{U}}$. The ordering of the constraints is arbitrary; pyOptSparse reorders the problem automatically depending on the requirements of each individual optimizer.

The optimization class is created using the following call:

```
>>> optProb = Optimization('name', objFun)
```

The general template of the objective function is as follows:

```
def obj_fun(xdict):
  funcs = {}
  funcs['obj'] = function(x)
  funcs['con_name'] = function(x)
  fail = False # Or True if an analysis failed

  return funcs, fail
```

where:

- funcs is the dictionary of constraints and objective value(s)
- fail can be a Boolean or an int. False (or 0) for successful evaluation and True (or 1) for unsuccessful. Can also be 2 when using SNOPT and requesting a clean termination of the run.

If the Optimization problem is unconstrained, funcs will contain only the objective key(s).

5.1 Design Variables

The simplest way to add a single continuous variable with no bounds (side constraints) and initial value of 0.0 is:

```
>>> optProb.addVar('var_name')
```

This will result in a scalar variable included in the `x` dictionary call to `obj_fun` which can be accessed by doing:

```
>>> x['var_name']
```

A more complex example will include lower bounds, upper bounds and a non-zero initial value:

```
>>> optProb.addVar('var_name',lower=-10, upper=5, value=-2)
```

The `lower` or `upper` keywords may be specified as `None` to signify there is no bound on the variable.

Finally, an additional keyword argument `scale` can be specified which will perform an internal design variable scaling. The `scale` keyword will result in the following:

```
x_optimizer = x_user * scale
```

The purpose of the scale factor is ensure that design variables of widely different magnitudes can be used in the same optimization. Is it desirable to have the magnitude of all variables within an order of magnitude or two of each other.

The `addVarGroup` call is similar to `addVar` except that it adds a group of 1 or more variables. These variables are then returned as a numpy array within the x-dictionary. For example, to add 10 variables with no lower bound, and a scale factor of 0.1:

```
>>> optProb.addVarGroup('con_group', 10, upper=2.5, scale=0.1)
```

5.2 Constraints

The simplest way to add a single constraint with no bounds (ie not a very useful constraint!) is:

```
>>> optProb.addCon('not_a_real_constraint')
```

To include bounds on the constraints, use the `lower` and `upper` keyword arguments. If `lower` and `upper` are the same, it will be treated as an equality constraint:

```
>>> optProb.addCon('inequality_constraint', upper=10)
>>> optProb.addCOn('equality_constraint', lower=5, upper=5)
```

Like design variables, it is often necessary to scale constraints such that all constraint values are approximately the same order of magnitude. This can be specified using the `scale` keyword:

```
>>> optProb.addCon('scaled_constraint', upper=10000, scale=1.0/10000)
```

Even if the `scale` keyword is given, the `lower` and `upper` bounds are given in their un-scaled form. Internally, `pyOptSparse` will use the scaling factor to produce the following constraint:

```
con_optimizer = con_user * scale
```

In the example above, the constraint values are divided by 10000, which results in a upper bound (that the optimizer sees) of 1.0.

Constraints may also be flagged as liner using the `linear=True` keyword option. Some optimizers can perform special treatment on linear constraint, often ensuring that they are always satisfied exactly on every function call (SNOPT for example). Linear constraints also require the use of the `wrt` and `jac` keyword arguments. These are explained below.

One of the major goals of `pyOptSparse` is to enable the use of sparse constraint jacobians. (Hence the 'Sparse' in the name!). Manually computing sparsity structure of the constraint Jacobian is tedious at best and become even more complicated as optimization scripts are modified by adding or deleting design variables and/or constraints. `pyOptSParse` is designed to greatly facilitate the assembly of sparse constraint jacobians, alleviating the user of thus burden. The idea is that instead of the user computing a dense matrix representing the constraint jacobian, a `dictionary of keys` approach is used which allows incrementally specifying parts of the constraint jacobain. Consider the optimization problem given below:

```
                varA (3)   varB (1)   varC (3)
              +--------------------------------+
conA (2)      |          |     X     |     X    |
              ----------------------------------
conB (2)      |     X    |           |     X    |
              ----------------------------------
conC (4)      |     X    |     X     |     X    |
              ----------------------------------
conD (3)      |          |           |     X    |
              +--------------------------------+
```

The `X`'s denote which parts of the jacobian have non-zero values. `pyOptSparse` does not determine the sparsity structure of the jacobian automatally, it must be specified by the user during calls to `addCon` and `addConGroup`. By way of example, the code that generates the hypothetical optimization problem is as follows:

```
optProb.addVarGroup('varA', 3)
optProb.addVarGroup('varB', 1)
optProb.addVarGroup('varC', 3)

optProb.addConGroup('conA', 2, upper=0.0, wrt=['varB', 'varC'])
optProb.addConGroup('conB', 2, upper=0.0, wrt=['varC', 'varA'])
optProb.addConGroup('conC', 4, upper=0.0)
optProb.addConGroup('conD', 3, upper=0.0, wrt=['varC'])
```

Note that the order of the `wrt` (which stands for with-respect-to) is not significant. Furthermore, if the `wrt` argument is omitted altogether, `pyOptSparse` assumes that the constraint is dense.

Using the `wrt` keyword allows the user to determine the overall sparsity structure of the constraint jacobian. However, we have currently assumed that each of the blocks with an `X` in is a dense sub-block. `pyOptSparse` allows each of the *sub-blocks* to itself be sparse. `pyOptSparse` requires that this sparsity structure to be specified when the constraint is added. This information is supplied through the `jac` keyword argument. Lets say, that the (conD, varC) block of the jacobian is actually a sparse and linear. By way of example, the call instead may be as follows:

```
jac = sparse.lil_matrix((3,3))
jac[0,0] = 1.0
jac[1,1] = 4.0
jac[2,2] = 5.0

optProb.addConGroup('conD', 3, upper=0.0, wrt=['varC'], linear=True, jac={'varC':jac})
```

We have created a linked list sparse matrix using `scipy.sparse`. Any scipy sparse matrix format can be accepted. We have then provided this constraint jacobian using the `jac=` keyword argument. This argument is a dictionary, and the keys must match the design variable sets given in the `wrt` to keyword. Essentially what we have done is specified the which blocks of the constraint rows are non-zero, and provided the sparsity structure of ones that are sparse.

For linear constraints the values in `jac` are meaningful: They must be the actual linear constraint jacobian values (which do not change). For non-linear constraints, on the sparsity structure (non-zero pattern) is significant. The values themselves will be determined by a call the sens() function.

Also note, that the `wrt` and `jac` keyword arguments are only supported when user-supplied sensitivity is used. If one used the automatic gradient in `pyOptSparse` the constraint jacobian will necessarily be dense.

5.3 Objectives

Each optimization will require at least one objective to be added. This is accomplished using a the call:

```
otpProb.addObj('obj')
```

What this does is tell `pyOptSparse` that the key `obj` in the function returns will be taken as the objective. For optimizers that can do multi-objective optimization, (NSGA2 for example) multiple objectives can be added. Optimizers that can only handle one objective enforce that only a single objective is added to the optimization description.

CHAPTER 6

Optimizers

6.1 SNOPT

SNOPT is a sparse nonlinear optimizer that is particularly useful for solving large-scale constrained problems with smooth objective functions and constraints. The algorithm consists of a sequential quadratic programming (SQP) algorithm that uses a smooth augmented Lagrangian merit function, while making explicit provision for infeasibility in the original problem and in the quadratic programming subproblems. The Hessian of the Lagrangian is approximated using the BFGS quasi-Newton update.

6.1.1 Installation

SNOPT is available for purchase here. Upon purchase, you should receive a zip file. Within the zip file, there is a folder called `src`. To use SNOPT with pyoptsparse, paste all files from `src` except snopth.f into `pyoptsparse/pySNOPT/source`.

From v2.0 onwards, only SNOPT v7.7 is officially supported. To use pyOptSparse with previous versions of SNOPT, please checkout release v1.2.

6.1.2 API

class `pyoptsparse.pySNOPT.pySNOPT.`**SNOPT**(**args*, ***kwargs*)

SNOPT Optimizer Class - Inherited from Optimizer Abstract Class

__call__(*self*, *optProb*, *sens=None*, *sensStep=None*, *sensMode=None*, *storeHistory=None*, *hotStart=None*, *storeSens=True*, *timeLimit=None*)

This is the main routine used to solve the optimization problem.

Parameters

optProb [Optimization or Solution class instance] This is the complete description of the optimization problem to be solved by the optimizer

sens [str or python Function.] Specifiy method to compute sensitivities. The default is None which will use SNOPT's own finite differences which are vastly superiour to the pyOptSparse implementation. To explictly use pyOptSparse gradient class to do the derivatives with finite differenes use 'FD'. 'sens' may also be 'CS' which will cause pyOptSpare to compute the derivatives using the complex step method. Finally, 'sens' may be a python function handle which is expected to compute the sensitivities directly. For expensive function evaluations and/or problems with large numbers of design variables this is the preferred method.

sensStep [float] Set the step size to use for design variables. Defaults to 1e-6 when sens is 'FD' and 1e-40j when sens is 'CS'.

sensMode [str] Use 'pgc' for parallel gradient computations. Only available with mpi4py and each objective evaluation is otherwise serial

storeHistory [str] File name of the history file into which the history of this optimization will be stored

hotStart [str] File name of the history file to "replay" for the optimziation. The optimization problem used to generate the history file specified in 'hotStart' must be **IDENTICAL** to the currently supplied 'optProb'. By identical we mean, **EVERY SINGLE PARAMETER MUST BE IDENTICAL**. As soon as he requested evaluation point from SNOPT does not match the history, function and gradient evaluations revert back to normal evaluations.

storeSens [bool] Flag sepcifying if sensitivities are to be stored in hist. This is necessay for hot-starting only.

timeLimit [float] Specify the maximum amount of time for optimizer to run. Must be in seconds. This can be useful on queue systems when you want an optimization to cleanly finish before the job runs out of time.

6.2 IPOPT

IPOPT (Interior Point OPTimizer) is an open source interior point optimizer, designed for large-scale nonlinear optimization. The source code can be found here. The latest version we support is 3.11.7.

6.2.1 Installation

Install instructions for `pyIPOPT`.

1. Download `IPopt-3.11.7.tar.gz` and put in the `/pyoptsparse/pyoptsparse/pyIPOPT` directory
2. Untar
3. Rename the directory from `Ipopt-x.x.x` to `Ipopt`
4. Obtain the MA27 linear solver from HSL. Rename the source file `ma27ad.f` and put it in the `Ipopt/ThirdParty/HSLold/` directory
5. Go to:

```
Ipopt/ThirdParty/Blas/
```

and run:

```
sh ./get.Blas
```

This will download a blas copy and `Ipopt` will use that.

6. Go to:

```
Ipopt/ThirdParty/Lapack/
```

and run:

```
sh ./get.Lapack
```

7. Run in the root directory:

```
$ ./configure --disable-linear-solver-loader
```

8. Now make:

```
$ make install
```

9. You must add the `lib` directory `Ipopt` to your `LD_LIBRARY_PATH` variable for things to work right:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/user/hg/pyoptsparse/pyoptsparse/
↪pyIPOPT/Ipopt/lib
```

10. Now the pyOptSparse builder (run from the root directory) should take care of the rest.

6.2.2 API

class pyoptsparse.pyIPOPT.pyIPOPT.**IPOPT**(**args*, ***kwargs*)

IPOPT Optimizer Class - Inherited from Optimizer Abstract Class

__call__(*self*, *optProb*, *sens=None*, *sensStep=None*, *sensMode=None*, *storeHistory=None*, *hotStart=None*, *storeSens=True*)

This is the main routine used to solve the optimization problem.

Parameters

optProb [Optimization or Solution class instance] This is the complete description of the optimization problem to be solved by the optimizer

sens [str or python Function.] Specifiy method to compute sensitivities. To explictly use pyOptSparse gradient class to do the derivatives with finite differenes use 'FD'. 'sens' may also be 'CS' which will cause pyOptSpare to compute the derivatives using the complex step method. Finally, 'sens' may be a python function handle which is expected to compute the sensitivities directly. For expensive function evaluations and/or problems with large numbers of design variables this is the preferred method.

sensStep [float] Set the step size to use for design variables. Defaults to 1e-6 when sens is 'FD' and 1e-40j when sens is 'CS'.

sensMode [str] Use 'pgc' for parallel gradient computations. Only available with mpi4py and each objective evaluation is otherwise serial

storeHistory [str] File name of the history file into which the history of this optimization will be stored

hotStart [str] File name of the history file to "replay" for the optimziation. The optimization problem used to generate the history file specified in 'hotStart' must be **IDENTICAL** to the currently supplied 'optProb'. By identical we mean, **EVERY SINGLE PARAMETER MUST BE IDENTICAL**. As soon as he requested evaluation point does not match the history, function and gradient evaluations revert back to normal evaluations.

storeSens [bool] Flag sepcifying if sensitivities are to be stored in hist. This is necessay for hot-starting only.

6.3 SLSQP

SLSQP optimizer is a sequential least squares programming algorithm which uses the Han–Powell quasi–Newton method with a BFGS update of the B–matrix and an L1–test function in the step–length algorithm. The optimizer uses a slightly modified version of Lawson and Hanson's NNLS nonlinear least-squares solver.

The version provided is the original source code from 1991 by Dieter Kraft.

6.3.1 API

class pyoptsparse.pySLSQP.pySLSQP.**SLSQP**(**args*, ***kwargs*)

SLSQP Optimizer Class - Inherited from Optimizer Abstract Class

__call__(*self*, *optProb*, *sens=None*, *sensStep=None*, *sensMode=None*, *storeHistory=None*, *hotStart=None*, *storeSens=True*)

This is the main routine used to solve the optimization problem.

Parameters

optProb [Optimization or Solution class instance] This is the complete description of the optimization problem to be solved by the optimizer

sens [str or python Function.] Specifiy method to compute sensitivities. To explictly use pyOptSparse gradient class to do the derivatives with finite differenes use 'FD'. 'sens' may also be 'CS' which will cause pyOptSpare to compute the derivatives using the complex step method. Finally, 'sens' may be a python function handle which is expected to compute the sensitivities directly. For expensive function evaluations and/or problems with large numbers of design variables this is the preferred method.

sensStep [float] Set the step size to use for design variables. Defaults to 1e-6 when sens is 'FD' and 1e-40j when sens is 'CS'.

sensMode [str] Use 'pgc' for parallel gradient computations. Only available with mpi4py and each objective evaluation is otherwise serial

storeHistory [str] File name of the history file into which the history of this optimization will be stored

hotStart [str] File name of the history file to "replay" for the optimziation. The optimization problem used to generate the history file specified in 'hotStart' must be **IDENTICAL** to the currently supplied 'optProb'. By identical we mean, **EVERY SINGLE PARAMETER MUST BE IDENTICAL**. As soon as he requested evaluation point from SLSQP does not match the history, function and gradient evaluations revert back to normal evaluations.

storeSens [bool] Flag sepcifying if sensitivities are to be stored in hist. This is necessay for hot-starting only.

6.4 NLPQLP

NLPQLP is a sequential quadratic programming (SQP) method which solves problems with smooth continuously differentiable objective function and constraints. The algorithm uses a quadratic approximation of the Lagrangian function and a linearization of the constraints. To generate a search direction a quadratic subproblem is formulated and solved. The line search can be performed with respect to two alternative merit functions, and the Hessian approximation is updated by a modified BFGS formula.

NLPQLP is a proprietary software, which can be obtained here. The latest version supported is v4.2.2.

6.4.1 API

class pyoptsparse.pyNLPQLP.pyNLPQLP.**NLPQLP**(**args*, ***kwargs*)

NLPQL Optimizer Class - Inherited from Optimizer Abstract Class

__call__(*self*, *optProb*, *sens=None*, *sensStep=None*, *sensMode=None*, *storeHistory=None*, *hotStart=None*, *storeSens=True*)

This is the main routine used to solve the optimization problem.

Parameters

optProb [Optimization or Solution class instance] This is the complete description of the optimization problem to be solved by the optimizer

sens [str or python Function.] Specifiy method to compute sensitivities. To explictly use pyOptSparse gradient class to do the derivatives with finite differenes use 'FD'. 'sens' may also be 'CS' which will cause pyOptSpare to compute the derivatives using the complex step method. Finally, 'sens' may be a python function handle which is expected to compute the sensitivities directly. For expensive function evaluations and/or problems with large numbers of design variables this is the preferred method.

sensStep [float] Set the step size to use for design variables. Defaults to 1e-6 when sens is 'FD' and 1e-40j when sens is 'CS'.

sensMode [str] Use 'pgc' for parallel gradient computations. Only available with mpi4py and each objective evaluation is otherwise serial

storeHistory [str] File name of the history file into which the history of this optimization will be stored

hotStart [str] File name of the history file to "replay" for the optimziation. The optimization problem used to generate the history file specified in 'hotStart' must be **IDENTICAL** to the currently supplied 'optProb'. By identical we mean, **EVERY SINGLE PARAMETER MUST BE IDENTICAL**. As soon as he requested evaluation point from NLPQL does not match the history, function and gradient evaluations revert back to normal evaluations.

storeSens [bool] Flag sepcifying if sensitivities are to be stored in hist. This is necessay for hot-starting only.

6.5 FSQP

This code implements an SQP approach that is modified to generate feasible iterates. In addition to handling general single objective constrained nonlinear optimization problems, the code is also capable of handling multiple competing linear and nonlinear objective functions (minimax), linear and nonlinear inequality constraints, as well as linear and nonlinear equality constraints

> **Warning:** FSQP build fails, and is therefore deprecated.

6.5.1 API

class pyoptsparse.pyFSQP.pyFSQP.**FSQP**(**args*, ***kwargs*)
FSQP Optimizer Class - Inherited from Optimizer Abstract Class

__call__(*self*, *optProb*, *sens=None*, *sensStep=None*, *sensMode='FD'*, *storeHistory=None*, *hotStart=None*, *storeSens=True*)
This is the main routine used to solve the optimization problem.

Parameters

optProb [Optimization or Solution class instance] This is the complete description of the optimization problem to be solved by the optimizer

sens [str or python Function.] Specifiy method to compute sensitivities. To explictly use pyOptSparse gradient class to do the derivatives with finite differenes use 'FD'. 'sens' may also be 'CS' which will cause pyOptSpare to compute the derivatives using the complex step method. Finally, 'sens' may be a python function handle which is expected to compute the sensitivities directly. For expensive function evaluations and/or problems with large numbers of design variables this is the preferred method.

sensStep [float] Set the step size to use for design variables. Defaults to 1e-6 when sens is 'FD' and 1e-40j when sens is 'CS'.

sensMode [str] Use 'pgc' for parallel gradient computations. Only available with mpi4py and each objective evaluation is otherwise serial

storeHistory [str] File name of the history file into which the history of this optimization will be stored

hotStart [str] File name of the history file to "replay" for the optimziation. The optimization problem used to generate the history file specified in 'hotStart' must be **IDENTICAL** to the currently supplied 'optProb'. By identical we mean, **EVERY SINGLE PARAMETER MUST BE IDENTICAL**. As soon as he requested evaluation point from SNOPT does not match the history, function and gradient evaluations revert back to normal evaluations.

storeSens [bool] Flag sepcifying if sensitivities are to be stored in hist. This is necessay for hot-starting only.

6.6 NSGA2

This optimizer is a non-dominating sorting genetic algorithm that solves non-convex and non-smooth single and multiobjective optimization problems. The algorithm attempts to perform global optimization, while enforcing constraints using a tournament selection-based strategy

> **Warning:** Currently, the Python wrapper does not catch exceptions. If there is **any** error in the user-supplied function, you will get a seg-fault and no idea where it happened. Please make sure the objective is without errors before trying to use nsga2.

6.6.1 API

class pyoptsparse.pyNSGA2.pyNSGA2.**NSGA2**(**args*, ***kwargs*)
NSGA2 Optimizer Class - Inherited from Optimizer Abstract Class

__call__(*self*, *optProb*, *storeHistory=None*, *hotStart=None*, ***kwargs*)
This is the main routine used to solve the optimization problem.

Parameters

optProb [Optimization or Solution class instance] This is the complete description of the optimization problem to be solved by the optimizer

storeHistory [str] File name of the history file into which the history of this optimization will be stored

hotStart [str] File name of the history file to "replay" for the optimziation. The optimization problem used to generate the history file specified in 'hotStart' must be **IDENTICAL** to the currently supplied 'optProb'. By identical we mean, **EVERY SINGLE PARAMETER MUST BE IDENTICAL**. As soon as he requested evaluation point from NSGA2 does not match the history, function and gradient evaluations revert back to normal evaluations.

Notes

The kwargs are there such that the sens= argument can be supplied (but ignored here in nsga2)

6.7 PSQP

This optimizer implements a sequential quadratic programming method with a BFGS variable metric update

6.7.1 API

class pyoptsparse.pyPSQP.pyPSQP.**PSQP**(**args*, ***kwargs*)
PSQP Optimizer Class - Inherited from Optimizer Abstract Class

__call__(*self*, *optProb*, *sens=None*, *sensStep=None*, *sensMode=None*, *storeHistory=None*, *hotStart=None*, *storeSens=True*)
This is the main routine used to solve the optimization problem.

Parameters

optProb [Optimization or Solution class instance] This is the complete description of the optimization problem to be solved by the optimizer

sens [str or python Function.] Specifiy method to compute sensitivities. To explictly use pyOptSparse gradient class to do the derivatives with finite differenes use 'FD'. 'sens' may also be 'CS' which will cause pyOptSpare to compute the derivatives using the complex step method. Finally, 'sens' may be a python function handle which is expected to compute the sensitivities directly. For expensive function evaluations and/or problems with large numbers of design variables this is the preferred method.

sensStep [float] Set the step size to use for design variables. Defaults to 1e-6 when sens is 'FD' and 1e-40j when sens is 'CS'.

sensMode [str] Use 'pgc' for parallel gradient computations. Only available with mpi4py and each objective evaluation is otherwise serial

storeHistory [str] File name of the history file into which the history of this optimization will be stored

hotStart [str] File name of the history file to "replay" for the optimziation. The optimization problem used to generate the history file specified in 'hotStart' must be **IDENTICAL** to the currently supplied 'optProb'. By identical we mean, **EVERY SINGLE PARAMETER MUST BE IDENTICAL**. As soon as he requested evaluation point from PSQP does not match the history, function and gradient evaluations revert back to normal evaluations.

storeSens [bool] Flag sepcifying if sensitivities are to be stored in hist. This is necessay for hot-starting only.

6.8 ParOpt

ParOpt is a nonlinear interior point optimizer that is designed for large parallel design optimization problems with structured sparse constraints. ParOpt is open source and can be downloaded at https://github.com/gjkennedy/paropt. Documentation and examples for ParOpt can be found at https://gjkennedy.github.io/paropt/. ParOpt does not provide version tagging, but the commit `f692160` from October 2019 has been verified to work.

6.8.1 Installation

Please follow the instructions here to install ParOpt as a separate Python package. Make sure that the package is named `paropt` and the installation location can be found by Python, so that `from paropt import ParOpt` works within the pyOptSparse folder. This typically requires installing it in a location which is already present under `$PYTHONPATH` environment variable, or you can modify the `.bashrc` file and manually append the path.

6.8.2 API

class `pyoptsparse.pyParOpt.ParOpt.`**`ParOpt`**(**args*, ***kwargs*)

ParOpt optimizer class

ParOpt has the capability to handle distributed design vectors. This is not replicated here since pyOptSparse does not have the capability to handle this type of design problem.

`__call__`(*self*, *optProb*, *sens=None*, *sensStep=None*, *sensMode=None*, *storeHistory=None*, *hotStart=None*, *storeSens=True*)

This is the main routine used to solve the optimization problem.

Parameters

optProb [Optimization or Solution class instance] This is the complete description of the optimization problem to be solved by the optimizer

sens [str or python Function.] Specifiy method to compute sensitivities. To explictly use pyOptSparse gradient class to do the derivatives with finite differenes use 'FD'. 'sens' may also be 'CS' which will cause pyOptSpare to compute the derivatives using the complex step method. Finally, 'sens' may be a python function handle which is expected to compute the sensitivities directly. For expensive function evaluations and/or problems with large numbers of design variables this is the preferred method.

sensStep [float] Set the step size to use for design variables. Defaults to 1e-6 when sens is 'FD' and 1e-40j when sens is 'CS'.

sensMode [str] Use 'pgc' for parallel gradient computations. Only available with mpi4py and each objective evaluation is otherwise serial

storeHistory [str] File name of the history file into which the history of this optimization will be stored

hotStart [str] File name of the history file to "replay" for the optimziation. The optimization problem used to generate the history file specified in 'hotStart' must be **IDENTICAL** to the currently supplied 'optProb'. By identical we mean, **EVERY SINGLE PARAMETER MUST BE IDENTICAL**. As soon as he requested evaluation point from ParOpt does not match the history, function and gradient evaluations revert back to normal evaluations.

storeSens [bool] Flag sepcifying if sensitivities are to be stored in hist. This is necessay for hot-starting only.

6.9 CONMIN

CONstrained function MINimization (CONMIN) is a gradient-based optimizer that uses the methods of feasible directions.

6.9.1 API

class pyoptsparse.pyCONMIN.pyCONMIN.**CONMIN**(**args*, ***kwargs*)

CONMIN Optimizer Class - Inherited from Optimizer Abstract Class

__call__(*self*, *optProb*, *sens=None*, *sensStep=None*, *sensMode=None*, *storeHistory=None*, *hotStart=None*, *storeSens=True*)

This is the main routine used to solve the optimization problem.

Parameters

optProb [Optimization or Solution class instance] This is the complete description of the optimization problem to be solved by the optimizer

sens [str or python Function.] Specifiy method to compute sensitivities. To explictly use pyOptSparse gradient class to do the derivatives with finite differenes use 'FD'. 'sens' may also be 'CS' which will cause pyOptSpare to compute the derivatives using the complex step method. Finally, 'sens' may be a python function handle which is expected to compute the sensitivities directly. For expensive function evaluations and/or problems with large numbers of design variables this is the preferred method.

sensStep [float] Set the step size to use for design variables. Defaults to 1e-6 when sens is 'FD' and 1e-40j when sens is 'CS'.

sensMode [str] Use 'pgc' for parallel gradient computations. Only available with mpi4py and each objective evaluation is otherwise serial

storeHistory [str] File name of the history file into which the history of this optimization will be stored

hotStart [str] File name of the history file to "replay" for the optimziation. The optimization problem used to generate the history file specified in 'hotStart' must be **IDENTICAL** to the currently supplied 'optProb'. By identical we mean, **EVERY SINGLE PARAMETER MUST BE IDENTICAL**. As soon as he requested evaluation point from CONMIN does not match the history, function and gradient evaluations revert back to normal evaluations.

storeSens [bool] Flag sepcifying if sensitivities are to be stored in hist. This is necessay for hot-starting only.

6.10 ALPSO

Augmented Lagrangian Particle Swarm Optimizer (ALPSO) is a PSO method that uses the augmented Lagrangian approach to handle constraints.

6.10.1 API

class pyoptsparse.pyALPSO.pyALPSO.**ALPSO**(**args*, ***kwargs*)

ALPSO Optimizer Class - Inherited from Optimizer Abstract Class

*Keyword arguments:**

- pll_type -> STR: ALPSO Parallel Implementation (None, SPM- Static, DPM- Dynamic, POA-Parallel Analysis), *Default* = None

__call__(*self*, *optProb*, *storeHistory=None*, ***kwargs*)

This is the main routine used to solve the optimization problem.

Parameters

optProb [Optimization or Solution class instance] This is the complete description of the optimization problem to be solved by the optimizer

storeHistory [str] File name of the history file into which the history of this optimization will be stored

Notes

The kwargs are there such that the sens= argument can be supplied (but ignored here in alpso)

CHAPTER 7

pyOptSparse API Documentation

7.1 Optimziation

class pyoptsparse.pyOpt_optimization.**Optimization**(*name*, *objFun*, *comm=None*)
Create a description of an optimization problem.

Parameters

name [str] Name given to optimization problem. This is name is currently not used for anything, but may be in the future.

objFun [python function] Python function handle of function used to evaluate the objective function.

comm [MPI intra communication] The communicator this problem will be solved on. This is required for both analysis when the objective is computed in parallel as well as to use the internal parallel gradient computations. Defaults to MPI.COMM_WORLD if not given.

__init__(*self*, *name*, *objFun*, *comm=None*)
Initialize self. See help(type(self)) for accurate signature.

addCon(*self*, *name*, **args*, ***kwargs*)
Convenience function. See addConGroup() for more information

addConGroup(*self*, *name*, *nCon*, *lower=None*, *upper=None*, *scale=1.0*, *linear=False*, *wrt=None*, *jac=None*)
Add a group of variables into a variable set. This is the main function used for adding variables to pyOptSparse.

Parameters

name [str] Constraint name. All names given to constraints must be unique

nCon [int] The number of constraints in this group

lower [scalar or array] The lower bound(s) for the constraint. If it is a scalar, it is applied to all nCon constraints. If it is an array, the array must be the same length as nCon.

upper [scalar or array] The upper bound(s) for the constraint. If it is a scalar, it is applied to all nCon constraints. If it is an array, the array must be the same length as nCon.

scale [scalar or array] A scaling factor for the constraint. It is generally advisable to have most optimization constraint around the same order of magnitude.

linear [bool] Flag to specify if this constraint is linear. If the constraint is linear, both the 'wrt' and 'jac' keyword arguments must be given to specify the constant portion of the constraint jacobian.

wrt [iterable (list, set, OrderedDict, array etc)] 'wrt' stand for stands for 'With Respect To'. This specifies for what dvs have non-zero jacobian values for this set of constraints. The order is not important.

jac [dictionary] For linear and sparse non-linear constraints, the constraint jacobian must be passed in. The structure is jac dictionary is as follows:

{'dvName1':<matrix1>, 'dvName2', <matrix1>, . . . }

They keys of the jacobian must correspond to the dvGroups given in the wrt keyword argument. The dimensions of each "chunk" of the constraint jacobian must be consistent. For example, <matrix1> must have a shape of (nCon, nDvs) where nDVs is the total number of design variables in dvName1. <matrix1> may be a dense numpy array or it may be scipy sparse matrix. However, it is *HIGHLY* recommended that sparse constraints are supplied to pyOptSparse using the pyOptSparse's simplified sparse matrix format. The reason for this is that it is *impossible* for force scipy sparse matrices to keep a fixed sparsity pattern; if the sparsity pattern changes during an optimization, *IT WILL FAIL*.

The three simplified pyOptSparse sparse matrix formats are summarized below:

mat = {'coo':[row, col, data], 'shape':[nrow, ncols]} # A coo matrix mat = {'csr':[rowp, colind, data], 'shape':[nrow, ncols]} # A csr matrix mat = {'coo':[colp, rowind, data], 'shape':[nrow, ncols]} # A csc matrix

Note that for nonlinear constraints (linear=False), the values themselves in the matrices in jac do not matter, but the sparsity structure **does** matter. It is imperative that entries that will at some point have non-zero entries have non-zero entries in jac argument. That is, we do not let the sparsity structure of the jacobian change throughout the optimization. This stipulation is automatically checked internally.

`addVar`(*self*, *name*, **args*, ***kwargs*)

This is a convenience function. It simply calls addVarGroup() with nVars=1. Variables added with addVar() are returned as *scalars*.

`addVarGroup`(*self*, *name*, *nVars*, *type='c'*, *value=0.0*, *lower=None*, *upper=None*, *scale=1.0*, *offset=0.0*, *choices=None*, ***kwargs*)

Add a group of variables into a variable set. This is the main function used for adding variables to pyOptSparse.

Parameters

name [str] Name of variable group. This name should be unique across all the design variable groups

nVars [int] Number of design variables in this group.

type [str.] String representing the type of variable. Suitable values for type are: 'c' for continuous variables, 'i' for integer values and 'd' for discrete selection.

value [scalar or array.] Starting value for design variables. If it is a a scalar, the same value is applied to all 'nVars' variables. Otherwise, it must be iterable object with length equal to 'nVars'.

lower [scalar or array.] Lower bound of variables. Scalar/array usage is the same as value keyword

upper [scalar or array.] Upper bound of variables. Scalar/array usage is the same as value keyword

scale [scalar or array. Define a user supplied scaling] variable for the design variable group. This is often necessary when design variables of widely varying magnitudes are used within the same optimization. Scalar/array usage is the same as value keyword.

offset [scalar or array. Define a user supplied offset] variable for the design variable group. This is often necessary when design variable has a large magnitude, but only changes a little about this value.

choices [list] Specify a list of choices for discrete design variables

Notes

Calling addVar() and addVarGroup(. . . , nVars=1, . . .) are **NOT** equivalent! The variable added with addVar() will be returned as scalar, while variable returned from addVarGroup will be an array of length 1.

It is recommended that the addVar() and addVarGroup() calls follow the examples above by including all the keyword arguments. This make it very clear the intent of the script's author. The type, value, lower, upper and scale should be given for all variables even if the default value is used.

Examples

```
>>> # Add a single design variable 'alpha'
>>> optProb.addVar('alpha', type='c', value=2.0, lower=0.0, upper=10.0,
↪ scale=0.1)
>>> # Add 10 unscaled variables of 0.5 between 0 and 1 with name 'y'
>>> optProb.addVarGroup('y', type='c', value=0.5, lower=0.0, upper=1.0,
↪ scale=1.0)
```

delVar (*self*, *name*)

Delete a variable or variable group

Parameters

name [str] Name of variable or variable group to remove

getDVs (*self*)

Return a dictionary of the design variables. In most common usage, this function is not required.

Returns

outDVs [dict] The dictionary of variables. This is the same as 'x' that would be used to call the user objective function.

printSparsity (*self*, *verticalPrint=False*)

This function prints an (ascii) visualization of the jacobian sparsity structure. This helps the user visualize what pyOptSparse has been given and helps ensure it is what the user expected. It is highly recommended this function be called before the start of every optimization to verify the optimization problem setup.

Parameters

verticalPrint [bool] True if the design variable names in the header should be printed vertically instead of horizontally. If true, this will make the constraint Jacobian print out more narrow and taller.

> **Warning:** This function is **collective** on the optProb comm. It is therefore necessary to call this function on **all** processors of the optProb comm.

setDVs (*self*, *inDVs*)
set the problem design variables from a dictionary. In most common usage, this function is not required.

Parameters

inDVs [dict] The dictionary of variables. This dictionary is like the 'x' that would be used to call the user objective function.

setDVsFromHistory (*self*, *histFile*, *key=None*)
Set optimization variables from a previous optimization. This is like a cold start, but some variables may have been added or removed from the previous optimization. This will try to set all variables it can.

Parameters

histFile [str] Filename of the history file to read

key [str] Key of the history file to use for the x values. The default is None which will use the last x-value stored in the dictionary.

7.2 Optimzer

class pyoptsparse.pyOpt_optimizer.**Optimizer** (*name=None*, *category=None*, *defOptions=None*, *informs=None*, ***kwargs*)
Base optimizer class

Parameters

name [str] Optimizer name

category [str] Typically local or global

defOptions [dictionary] A dictionary containing the default options

informs [dict] Dictionary of the inform codes

getInform (*self*, *infocode=None*)
Get optimizer result inform code at exit

Parameters

infocode [int] Integer information code

getOption (*self*, *name*)
Return the optimizer option value for name

Parameters

name [str] name of option for which to retrieve value

Returns

value [varies] value of option for 'name'

setOption (*self*, *name*, *value=None*)
Generic routine for all option setting. The routine does error checking on the type of the value.

Parameters

name [str] Name of the option to set

value [varies] Variable value to set.

`pyoptsparse.pyOpt_optimizer.OPT(optName, *args, **kwargs)`

This is a simple utility function that enables creating an optimizer based on the 'optName' string. This can be useful for doing optimization studies with respect to optimizer since you don't need massive if-statements.

Parameters

optName [str] String identifying the optimizer to create

***args, **kwargs** [varies] Passed to optimizer creation.

Returns

opt [pyOpt_optimizer inherited optimizer] The desired optimizer

7.3 Constraint

`class pyoptsparse.pyOpt_constraint.Constraint(name, nCon, linear, wrt, jac, lower, upper, scale)`

Constraint Class Initialization

`finalize(self, variables, dvOffset, index)`

This function should not need to be called by the user

After the design variables have been finalized and the order is known we can check the constraint for consistency.

Parameters

variables [Ordered Dict] The pyOpt variable list after they have been finalized.

dvOffset [dict] Design variable offsets from pyOpt_optimization

index [int] The starting index of this constraint in natural order

7.4 Variable

`class pyoptsparse.pyOpt_variable.Variable(name, type, value, lower, upper, scale, offset, scalar=False, choices=None)`

Variable Class Initialization

7.5 Gradient

`class pyoptsparse.pyOpt_gradient.Gradient(optProb, sensType, sensStep=None, sensMode='', comm=None)`

Gradient class for automatically computing gradients with finite difference or complex step.

Parameters

optProb [Optimization instance] This is the complete description of the optimization problem.

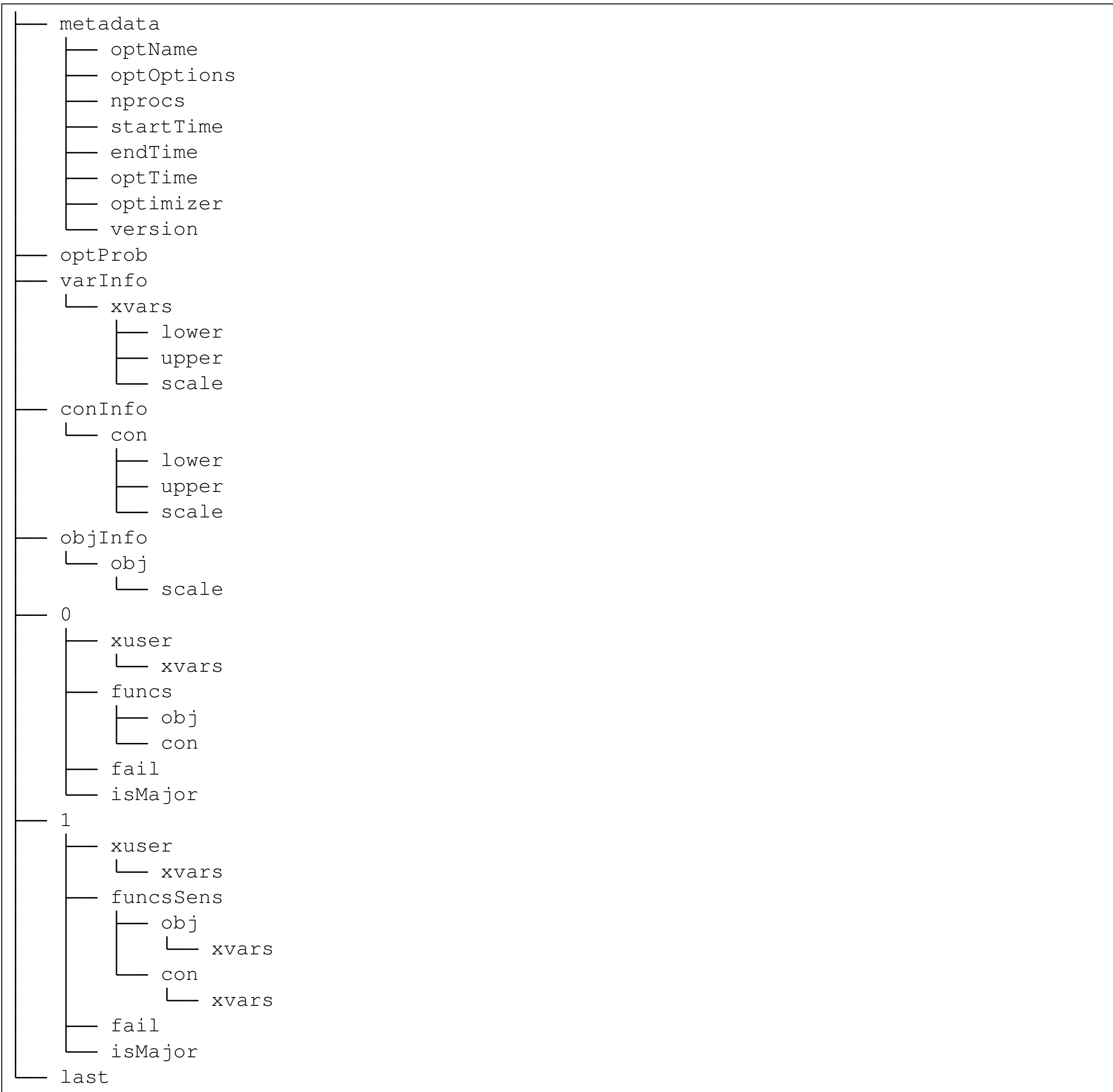
sensType [str] 'FD' for forward difference, 'CD' for central difference, 'FDR' for forward difference with relative step size, 'CDR' for central difference with relative step size, and 'CS' for complex step

sensStep [number] Step size to use for differencing

sensMode [str] Flag to compute gradients in parallel.

7.6 History

Suppose we have an optimization problem with one DV group `xvars`, one constraint `con`, and the objective is called `obj`. In this case, the history file would have the following layout:



```
├── metadata
│   ├── optName
│   ├── optOptions
│   ├── nprocs
│   ├── startTime
│   ├── endTime
│   ├── optTime
│   ├── optimizer
│   └── version
├── optProb
├── varInfo
│   └── xvars
│       ├── lower
│       ├── upper
│       └── scale
├── conInfo
│   └── con
│       ├── lower
│       ├── upper
│       └── scale
├── objInfo
│   └── obj
│       └── scale
├── 0
│   ├── xuser
│   │   └── xvars
│   ├── funcs
│   │   ├── obj
│   │   └── con
│   ├── fail
│   └── isMajor
├── 1
│   ├── xuser
│   │   └── xvars
│   ├── funcsSens
│   │   ├── obj
│   │   │   └── xvars
│   │   └── con
│   │       └── xvars
│   ├── fail
│   └── isMajor
└── last
```

The main optimization history is indexed via call counters, in this example `0` and `1`. Note that they do not match the major/minor iterations of a given optimizer, since gradient evaluations are stored separate from the function evaluation.

For SNOPT, a number of other values can be requested and stored in each major iteration, such as the feasibility and optimality from the SNOPT print out file.

7.6.1 API

class pyoptsparse.pyOpt_history.**History**(*fileName*, *optProb=None*, *temp=False*, *flag='r'*)
Optimizer History Class Initialization. This is essentially a thin wrapper around a SqliteDict dictionary to facilitate operations with pyOptSparse

Parameters

fileName [str] File name for history file

optProb [pyOpt_Optimization] the optimization object

temp [bool] Flag to signify that the file should be deleted after it is closed

flag [str] String specifying the mode. Similar to what was used in shelve. 'n' for a new database and 'r' to read an existing one.

close(*self*)
Close the underlying database. This should only be used in write mode. In read mode, we close the db during initialization.

getCallCounters(*self*)
Returns a list of all call counters stored in the history file.

Returns

list a list of strings, each entry being a call counter.

getConInfo(*self*, *key=None*)
Returns the *ConInfo*, for all keys by default. A *key* parameter can also be supplied, to retrieve *ConInfo* corresponding to specific constraints.

Parameters

key [str or list of str, optional] Specifies for which constraint to extract *ConInfo*.

Returns

dict A dictionary containing ConInfo. For a single key, the return is one level deeper.

getConNames(*self*)
Returns the names of constraints.

Returns

list of str A list containing the names of constraints.

getDVInfo(*self*, *key=None*)
Returns the *DVInfo*, for all keys by default. A *key* parameter can also be supplied, to retrieve *DVInfo* corresponding to specific DVs.

Parameters

key [str or list of str, optional] Specifies for which DV to extract *DVInfo*.

Returns

dict A dictionary containing DVInfo. For a single key, the return is one level deeper.

getDVNames(*self*)
Returns the names of the DVs.

Returns

list of str A list containing the names of DVs.

getIterKeys(*self*)
Returns the keys available at each optimization iteration. This function is useful for inspecting the history file, to determine what information is saved at each iteration.

Returns

list of str A list containing the names of keys stored at each optimization iteration.

getMetadata(*self*)
Returns a copy of the metadata stored in the history file.

Returns

dict A dictionary containing the metadata.

getObjInfo(*self*, *key=None*)
Returns the *ObjInfo*, for all keys by default. A *key* parameter can also be supplied, to retrieve *ObjInfo* corresponding to specific keys.

Parameters

key [str or list of str, optional] Specifies for which obj to extract *ObjInfo*.

Returns

dict A dictionary containing ObjInfo. For a single key, the return is one level deeper.

Notes

Recall that for the sake of generality, pyOptSparse allows for multiple objectives to be added. This feature is not used currently, but does make *ObjInfo* follow the same structure as *ConInfo* and *DVInfo*. Because of this, it is recommended that this function be accessed using the optional *key* argument.

getObjNames(*self*)
Returns the names of the objectives.

Returns

list of str A list containing the names of objectives.

Notes

Recall that for the sake of generality, pyOptSparse allows for multiple objectives to be added. This feature is not used currently, but does make *ObjNames* follow the same structure as *ConNames* and *DVNames*.

getOptProb(*self*)
Returns a copy of the optProb associated with the optimization.

Returns

optProb [pyOpt_optimization object] The optProb associated with the optimization. This is taken from the history file, and therefore has the `comm` and `objFun` fields removed.

getValues(*self*, *names=None*, *callCounters=None*, *major=True*, *scale=False*, *stack=False*)
Parses an existing history file and returns a data dictionary used to post-process optimization results, containing the requested optimization iteration history.

Parameters

names [list or str] the values of interest, can be the name of any DV, objective or constraint, or a list of them. If None, all values are returned. This includes the DVs, funcs, and any values stored by the optimizer.

callCounters [list of ints, can also contain 'last'] a list of callCounters to extract information from. If the callCounter is invalid, i.e. outside the range or is a funcsSens evaluation, then it is skipped. 'last' represents the last major iteration. If None, values from all callCounters are returned.

major [bool] flag to specify whether to include only major iterations.

scale [bool] flag to specify whether to apply scaling for the values. True means to return values that are scaled the same way as the actual optimization.

stack [bool] flag to specify whether the DV should be stacked into a single numpy array with the key *xuser*, or retain their separate DVGroups.

Returns

dict a dictionary containing the information requested. The keys of the dictionary correspond to the *names* requested. Each value is a numpy array with the first dimension equal to the number of callCounters requested.

Notes

Regardless of the major flag, failed function evaluations are not returned.

Examples

First we can request DV history over all major iterations:

```
>>> hist.getValues(names='xvars', major=True)
{'xvars': array([[-2.00000000e+00,  1.00000000e+00],
    [-1.00000000e+00,  9.00000000e-01],
    [-5.00305827e-17,  4.21052632e-01],
    [ 1.73666171e-06,  4.21049838e-01],
    [ 9.08477459e-06,  4.21045261e-01],
    [ 5.00000000e-01,  2.84786405e-01],
    [ 5.00000000e-01,  5.57279939e-01],
    [ 5.00000000e-01,  2.00000000e+00]])}
```

Next we can look at DV and optimality for the first and last iteration only:

```
>>> hist.getValues(names=['xvars','optimality'],callCounters=[0,'last'])
{'optimality': array([1.27895528, 0. ]),
'xvars': array([[-2. , 1. ],
                [ 0.5, 2. ]])}
```

pointExists (*self*, *callCounter*)

Determine if callCounter is in the database

Parameters

callCounter [int or str of int]

Returns

bool True if the callCounter exists in the history file. False otherwise.

read (*self*, *key*)

Read data for an arbitrary key. Returns None if key is not found. If passing in a callCounter, it should be verified by calling pointExists() first.

Parameters

key [str or int] generic key[str] or callCounter[int]

Returns

dict The value corresponding to *key* is returned. If the key is not found, *None* is returned instead.

write(*self*, *callCounter*, *data*)

This is the main to write data. Basically, we just pass in the callCounter, the integer forming the key, and a dictionary which will be written to the key

Parameters

callCounter [int] the callCounter to write to

data [dict] the dictionary corresponding to the callCounter

writeData(*self*, *key*, *data*)

Write arbitrary *key:data* value to db.

Parameters

key [str] The key to be added to the history file

data The data corresponding to the key. It can be anything as long as it is serializable in *sqlitedict*.

CHAPTER 8

Postprocessing

OptView is designed to quickly and interactively visualize optimization histories.

8.1 Requirements

OptView has the following dependency tree:

```
matplotlib (OptView)
  \-backends
  | \-backend_tkagg
  |   \-FigureCanvasTkAgg (OptView)
  |   \-NavigationToolbar2TkAgg (OptView)
  \-pyplot (OptView)
mpl_toolkits
  \-axes_grid1
  | \-host_subplot (OptView)
  \-axisartist (OptView)
numpy (OptView)
```

If you are successfully running pyOptSparse, these packages are most likely already installed.

Although not necessary for most usage, the dill package is needed if you wish to save an editable version of the graph produced in OptView. dill can be installed via pip in a terminal using:

```
pip install dill
```

view_saved_figure.py can be used to reformat and view the saved figure.

8.2 Usage

OptView can be run via terminal as:

```
python OptView.py --histFile --outputDirectory
```

Here, `histFile` is the name of the history file to be examined (default is 'opt_hist.hst'). `outputDirectory` is the name of the desired output directory for saved images (default is within the same folder as `OptView.py`.)

`OptView` can also be ran from any directory by adding an alias line to your `.bashrc` file such as:

```
alias OptView='python ~/hg/pyoptsparse/postprocessing/OptView.py
```

Through this usage, `OptView` can be called from any directory as:

```
OptView histFile --outputDirectory
```

Additionally, you can open multiple history files in the same `OptView` instance by calling them via the command line:

```
OptView histFile1 histFile2 histFile3 --outputDirectory
```

Each file's contents will be loaded into `OptView` with a flag appended to the end of each variable or function name corresponding to the history file. The first one listed will have '_A' added to the name, the second will have '_B' added, etc. There is currently no limit to the number of history files than can be loaded.

8.3 Features

`OptView` has many options and features, including:

- plotting multiple variables on a single plot
- producing stacked plots
- live searchable variable names
- hovering plot labels
- saving the figure to an image or pickling it for later formatting
- refreshing the optimization history on the fly

Although some of these are self-explanatory, the layout and usage of `OptView` will be explained below.

8.3.1 GUI Layout

The window is divided into two sections. The top is the canvas where the figure and graphs will be produced, while the bottom grayed section contains user-selectable options. Here, we will focus on the user options.

The selectable variables are contained on the lefthand side of the options panel in scrollable listboxes. You can select multiple items from the listboxes using the normal selection operators such as control and shift. If a selected variable is an array, a third listbox should appear on the righthand side of the options panel, allowing you to select specific subvariables within the single array variable.

There are three main options when selecting how to produce the graph(s):

- Shared axes - all selected variables are plotted on a single pair of axes
- Multiple axes - each selected variable gets its own y-axis while all selected data shares an x-axis
- Stacked plots - each variable gets its own individual plot and the set is stacked vertically

Most checkbox options should play well with any of these three main options, though there are known issues with using the ‘multiple axes’ option and delta values or for displaying arrays.

There are seven checkbox options:

- Absolute delta values - displays the absolute difference between one iteration’s value and the previous
- Log scale - sets the y-axis as a log scale
- Min/max for arrays - only shows the minimum and maximum value of a variable for each iteration
- Show all for arrays - plots all variables within an array
- Show legend - reveals the legend for the plotted data
- Show bounds - shows the variable bounds as dashed lines
- Show ‘major’ iterations - a heuristic filter to remove the line search iterations from the plotting results; especially useful for SNOPT output

Additionally, four buttons allow control of the plot:

- Refresh history - reloads the history file; used if checking on an optimization run on the fly
- Save all figures - saves .png versions of a basic plot for each variable in the history file
- Save figure - saves a .png and .pickle version of the current plot (the .pickle version can be reformatted afterwards)
- Quit - exits the program

Lastly, there some miscellaneous features:

- A search box to cull the selectable variables
- A font size slider to control the text size on the plot
- Hoverable tooltips when the cursor is on a plot line
- A variable called *actual_iteration_number* that gives a translation between history file iteration number and run file iteration number. This is especially useful for debugging specific steps of an optimization or comparing values across different histories.

More features are being developed on an as-needed basis. Feel free to edit the code as you see fit and submit a pull request if you would like to see a feature added. Alternatively, you can submit an issue ticket to discuss possible features.

8.4 Parsing SNOPT Printout files

The script `SNOPT_parse.py` has been included in the `postprocessing` folder for extracting the optimality, feasibility and meric function values for each major iteration. It then generates a `.dat` file for use with Tecplot.

The file can be run via terminal as:

```
python SNOPT_parse.py filename
```

Here, `filename` is the name of the SNOPT printout file to be examined. If no filename is provided the default name `SNOPT_print.out` will be assumed.

Index

Symbols

__call__() (*pyoptsparse.pyALPSO.pyALPSO.ALPSO method*), 28
__call__() (*pyoptsparse.pyCONMIN.pyCONMIN.CONMIN method*), 27
__call__() (*pyoptsparse.pyFSQP.pyFSQP.FSQP method*), 24
__call__() (*pyoptsparse.pyIPOPT.pyIPOPT.IPOPT method*), 21
__call__() (*pyoptsparse.pyNLPQLP.pyNLPQLP.NLPQLP method*), 23
__call__() (*pyoptsparse.pyNSGA2.pyNSGA2.NSGA2 method*), 25
__call__() (*pyoptsparse.pyPSQP.pyPSQP.PSQP method*), 25
__call__() (*pyoptsparse.pyParOpt.ParOpt.ParOpt method*), 26
__call__() (*pyoptsparse.pySLSQP.pySLSQP.SLSQP method*), 22
__call__() (*pyoptsparse.pySNOPT.pySNOPT.SNOPT method*), 19
__init__() (*pyoptsparse.pyOpt_optimization.Optimization method*), 29

A

addCon() (*pyoptsparse.pyOpt_optimization.Optimization method*), 29
addConGroup() (*pyoptsparse.pyOpt_optimization.Optimization method*), 29
addVar() (*pyoptsparse.pyOpt_optimization.Optimization method*), 30
addVarGroup() (*pyoptsparse.pyOpt_optimization.Optimization method*), 30
ALPSO (*class in pyoptsparse.pyALPSO.pyALPSO*), 28

C

close() (*pyoptsparse.pyOpt_history.History method*), 35
CONMIN (*class in pyoptsparse.pyCONMIN.pyCONMIN*), 27
Constraint (*class in pyoptsparse.pyOpt_constraint*), 33

D

delVar() (*pyoptsparse.pyOpt_optimization.Optimization method*), 31

F

finalize() (*pyoptsparse.pyOpt_constraint.Constraint method*), 33
FSQP (*class in pyoptsparse.pyFSQP.pyFSQP*), 24

G

getCallCounters() (*pyoptsparse.pyOpt_history.History method*), 35
getConInfo() (*pyoptsparse.pyOpt_history.History method*), 35
getConNames() (*pyoptsparse.pyOpt_history.History method*), 35
getDVInfo() (*pyoptsparse.pyOpt_history.History method*), 35
getDVNames() (*pyoptsparse.pyOpt_history.History method*), 35
getDVs() (*pyoptsparse.pyOpt_optimization.Optimization method*), 31
getInform() (*pyoptsparse.pyOpt_optimizer.Optimizer method*), 32
getIterKeys() (*pyoptsparse.pyOpt_history.History method*), 35
getMetadata() (*pyoptsparse.pyOpt_history.History method*), 36
getObjInfo() (*pyoptsparse.pyOpt_history.History method*), 36
getObjNames() (*pyoptsparse.pyOpt_history.History method*), 36

getOption() (*pyoptsparse.pyOpt_optimizer.Optimizer method*), 32
getOptProb() (*pyoptsparse.pyOpt_history.History method*), 36
getValues() (*pyoptsparse.pyOpt_history.History method*), 36
Gradient (*class in pyoptsparse.pyOpt_gradient*), 33

H

History (*class in pyoptsparse.pyOpt_history*), 35

I

IPOPT (*class in pyoptsparse.pyIPOPT.pyIPOPT*), 21

N

NLPQLP (*class in pyoptsparse.pyNLPQLP.pyNLPQLP*), 23
NSGA2 (*class in pyoptsparse.pyNSGA2.pyNSGA2*), 25

O

OPT() (*in module pyoptsparse.pyOpt_optimizer*), 33
Optimization (*class in pyoptsparse.pyOpt_optimization*), 29
Optimizer (*class in pyoptsparse.pyOpt_optimizer*), 32

P

ParOpt (*class in pyoptsparse.pyParOpt.ParOpt*), 26
pointExists() (*pyoptsparse.pyOpt_history.History method*), 37
printSparsity() (*pyoptsparse.pyOpt_optimization.Optimization method*), 31
PSQP (*class in pyoptsparse.pyPSQP.pyPSQP*), 25

R

read() (*pyoptsparse.pyOpt_history.History method*), 37

S

setDVs() (*pyoptsparse.pyOpt_optimization.Optimization method*), 32
setDVsFromHistory() (*pyoptsparse.pyOpt_optimization.Optimization method*), 32
setOption() (*pyoptsparse.pyOpt_optimizer.Optimizer method*), 32
SLSQP (*class in pyoptsparse.pySLSQP.pySLSQP*), 22
SNOPT (*class in pyoptsparse.pySNOPT.pySNOPT*), 19

V

Variable (*class in pyoptsparse.pyOpt_variable*), 33

W

write() (*pyoptsparse.pyOpt_history.History method*), 38
writeData() (*pyoptsparse.pyOpt_history.History method*), 38