The Journal of Open Source Software

pyOptSparse: A Python framework for large-scale constrained nonlinear optimization of sparse systems

Neil Wu[1], **Gaetan Kenway**[1], **Charles A. Mader**[1], **John Jasa**[1], **and Joaquim R. R. A. Martins**[1]

1 Department of Aerospace Engineering, University of Michigan

DOI: 10.21105/joss.02564

Software

- Review
- Repository
- Archive

Editor: Jack Poulson

Reviewers:

- @jgoldfar
- @vissarion
- @matbesancon

Submitted: 15 July 2020
Published: 24 October 2020

Summary

pyOptSparse is an optimization framework designed for constrained nonlinear optimization of large sparse problems and provides a unified interface for various gradient-free and gradient-based optimizers. By using an object-oriented approach, the software maintains independence between the optimization problem formulation and the implementation of the specific optimizers. The code is MPI-wrapped to enable execution of expensive parallel analyses and gradient evaluations, such as when using computational fluid dynamics (CFD) simulations, which can require hundreds of processors. The optimization history can be stored in a database file, which can then be used both for post-processing and restarting another optimization. A graphical user interface application is provided to visualize the optimization history interactively.

pyOptSparse considers optimization problems of the form

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{with respect to} \quad & x \\ \text{such that} \quad & l \leq \begin{pmatrix} x \\ Ax \\ g(x) \end{pmatrix} \leq u \end{aligned}$$

where x is the vector of design variables and $f(x)$ is a nonlinear objective function. A is the linear constraint Jacobian, and $g(x)$ is the set of nonlinear constraint functions. At time of writing, the latest released version of pyOptSparse is v2.2.0.

Features

Support for multiple optimizers

pyOptSparse provides built-in support for several popular proprietary and open-source optimizers. Each optimizer usually has its own way to specify the problem: It might require different constraint ordering, have different ways of specifying equality constraints, or use a sparse matrix format to represent the constraint Jacobian. pyOptSparse provides a common Python interface for the various optimizers that hides these differences from the user. By isolating the optimization problem definition from the optimizer, the user can easily switch between different optimizers applied to the same optimization problem. The optimizer can be switched by editing a single line of code.

Although pyOptSparse focuses primarily on large-scale gradient-based optimization, it provides support for gradient-free optimizers as well. Also, discrete variables, multi-objective, and population-based optimizers are all supported. Because of the object-oriented programming approach, it is also straightforward to extend pyOptSparse to support any additional optimizers that are not currently available. All of the features within pyOptSparse, including problem scaling and optimization hot-start, are automatically inherited when new optimizers are added.

String-based indexing

Unlike many other publicly available optimization frameworks, pyOptSparse is designed to handle large-scale optimizations, with a focus on engineering applications. With thousands of design variables and constraints, it is crucial to keep track of all values during optimization correctly. pyOptSparse employs string-based indexing to accomplish this. Instead of using a single flattened array, the related design variables and constraints can be grouped into separate arrays. These arrays are combined using an ordered dictionary, where each group is identified by a unique key. Similarly, the constraint Jacobian is represented by a nested dictionary approach. This representation has several advantages:

- The design variable and constraint values can be accessed without knowing their global indices, which reduces possible user error.
- The global indices are also often optimizer-dependent and this extra level of wrapping abstracts away potentially-confusing differences between optimizers.
- The constraint Jacobian can be computed and provided at the sub-block level, leaving pyOptSparse to assemble the whole Jacobian. This mimics the engineering workflow where different tools often compute different sub-blocks of the Jacobian. The user only has to ensure that the indices within each sub-block are correct, and the rest is handled automatically.

Support for sparse linear and nonlinear constraints

One prominent feature of pyOptSparse is the support for sparse constraints. When defining constraints, it is possible to provide the sparsity pattern of the Jacobian. This can be done at the global level by specifying which constraint groups are independent of which design variable groups, thereby letting pyOptSparse know that the corresponding sub-blocks of the Jacobian are always zero. For nonzero sub-blocks, it is also possible to supply the sparsity pattern of that sub-block, again using local indexing, such that the actual derivative computation can use sparse matrices as well.

pyOptSparse also provides explicit support for linear constraints since some optimizers provide special handling for these constraints. In these cases, only the Jacobian and the bounds of the constraint need to be supplied. The values and gradients of these constraints do not need to be evaluated every iteration, since the optimizer satisfies them internally.

Automatic computation of derivatives

If analytic derivatives for the objective and constraint functions are not available, pyOptSparse can automatically compute them internally using finite differences or the complex-step method (Martins, Sturdza, & Alonso, 2003). For finite differences, the user can use forward or central differences, with either an absolute or relative step size. Computing derivatives using finite differences can be expensive, requiring n extra evaluations for forward differences and $2n$ for centered differences. Finite differences are also inaccurate due to subtractive cancellation errors under finite precision arithmetic. The complex-step method, on the other hand, avoids subtractive cancellation errors. By using small enough steps, the complex-step derivatives can

be accurate to machine precision (Martins et al., 2003). The user must make sure that the objective and constraint functions can be evaluated correctly with complex design variable values when using this feature.

Optimizer-independent problem scaling

pyOptSparse offers optimizer-independent scaling for individual design variables, objective, and constraints. By separating the optimization problem definition from the particular optimizer, pyOptSparse can apply the scaling automatically and consistently with any supported optimizer. Since the optimization problem is always defined in the physical, user-defined space, the bounds on the design variables and constraints do not need to be modified when applying a different scaling. Furthermore, for gradient-based optimizers, all the derivatives are scaled automatically and consistently without any effort from the user. The user only needs to pass in a `scale` option when defining design variables, objective, and constraints. This is particularly useful in engineering applications, where the physical quantities can sometimes cause undesirable problem scaling, which leads to poor optimization convergence. pyOptSparse allows the user to adjust problem scaling for each design variable, constraint, and objective separately, without needing to change the bound specification or derivative computation.

Parallel execution

pyOptSparse can use MPI to execute function evaluations in parallel, in three distinct ways. Firstly and most commonly, it can perform parallel function evaluations when the functions themselves require multiple processors. This is usually the case when performing large-scale optimizations, where the objective and constraint functions are the result of a complex analysis, such as computational fluid dynamic simulations. In this scenario, pyOptSparse can be executed with multiple processors, where all processors perform the function evaluation, but only the root processor runs the optimizer itself. That way, we avoid the scenario where each processor runs an independent copy of the optimizer, potentially causing inconsistencies or processor locking.

Secondly, it is possible to perform parallel gradient evaluation when automatic finite-difference or complex-step derivatives are computed. If the function evaluation only requires a single processor, it is possible to call pyOptSparse with multiple processors so that each point in the finite-difference stencil is evaluated in parallel, reducing the wall time for derivative computations.

Lastly, some population-based optimizers may support parallel function evaluation for each optimizer iteration. In the case of a genetic algorithm or particle swarm optimization, multiple function evaluations are required at each optimizer iteration. These evaluations can be done in parallel if multiple processors are available and the functions only require a single processor to execute. However, the support and implementation of this mechanism is optimizer-dependent.

Leveraging the history file: visualization and restart

pyOptSparse can store an optimization history file using its own format based on SQLite. The history file contains the design variables and function values for each optimizer iteration, along with some metadata such as optimizer options. This file can then be visualized using OptView, a graphical user interface application provided by pyOptSparse. Alternatively, users can manually post-process results by using an API designed to query the history file and access the optimization history to generate plots.

The history file also enables two types of optimization restarts. A *cold start* merely sets the initial design variables to the previous optimization's final design variables. A *hot start*, on the

other hand, initializes the optimizer with the full state by replaying the previous optimization history. For a deterministic optimizer, the hot start generates the same sequence of iterates as long as the functions and gradients remain the same. For each iteration, pyOptSparse retrieves the previously-evaluated quantities and provides them to the optimizer without actually calling the objective and constraint functions, allowing us to exactly retrace the previous optimization and generate the same state within the optimizer in a non-intrusive fashion. This feature is particularly useful if the objective function is expensive to evaluate and the previous optimization was terminated due to problems such as reaching the maximum iteration limit. In this case, the full state within the optimizer can be regenerated through the hot start process so that the optimization can continue without performance penalties.

Simple optimization script

To highlight some of the features discussed above, we present the pyOptSparse script to solve a toy problem involving six design variables split into two groups, x and y. We also add two nonlinear constraints, one linear constraint, and design variable bounds. The optimization problem is as follows:

$$
\begin{aligned}
\text{minimize} \quad & x_0 + x_1^3 + y_0^2 + y_1^2 + y_2^2 + y_3^2 \\
\text{with respect to} \quad & x_0, x_1, y_0, y_1, y_2, y_3 \\
\text{such that} \quad & -10 \leq x_0 x_1 \\
& -10 \leq 3x_0 - \sin(x_1) \leq 10 \\
& y_0 - 2y_1 = 5 \\
& -10 < \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} < \begin{pmatrix} 10 \\ 100 \end{pmatrix} \\
& -10 < \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}
\end{aligned}
$$

The sparsity structure of the constraint Jacobian is shown below:

```
                 x (2)    y (4)
               +---------------+
       con (2) |   X   |       |
               ------------------
lin_con(L) (1) |       |   X   |
               +---------------+
```

This allows us to only specify derivatives for the two nonzero sub-blocks. For simplicity, we supply the linear Jacobian explicitly and use the complex-step method to compute the derivatives for the nonlinear constraints automatically.

We first define the imports and the objective function.

```
import numpy as np
from pyoptsparse import Optimization, OPT

def objfunc(xdict):
    x = xdict["x"]
    y = xdict["y"]
```

```
    funcs = {}
    funcs["obj"] = x[0] + x[1] ** 3 + np.sum(np.power(y, 2))
    funcs["con"] = np.zeros(2, np.complex)
    funcs["con"][0] = x[0] * x[1]
    funcs["con"][1] = 3 * x[0] - np.sin(x[1])
    fail = False
    return funcs, fail
```

Only the nonlinear constraints need to be evaluated here. Next, we set up the optimization problem, including design variables, objective, and constraints.

```
# Optimization Object
optProb = Optimization("Example Optimization", objfunc)

# Design Variables
nx = 2
lower = [-10, -10]
upper = [10, 100]
value = [-5, 6]
optProb.addVarGroup("x", nx, lower=lower, upper=upper, value=value)
ny = 4
optProb.addVarGroup("y", ny, lower=-10, upper=None, value=0)

# Nonlinear constraints
ncons = 2
lower = [-10, -10]
upper = [None, 10]
optProb.addConGroup("con", ncons, wrt="x", lower=lower, upper=upper)
# Linear constraint
jac = np.zeros((1, ny))
jac[0, 0] = 1
jac[0, 1] = -2
optProb.addConGroup(
    "lin_con", 1, lower=5, upper=5, wrt="y", jac={"y": jac}, linear=True
)
# Objective
optProb.addObj("obj")
```

By using the `wrt` argument when adding constraints, we tell pyOptSparse that only the specified sub-blocks of the Jacobian are nonzero.

The linear Jacobian for this problem is

$$\begin{pmatrix} 1 \\ -2 \\ 0 \\ 0 \end{pmatrix}$$

which we construct as `jac` and pass to pyOptSparse. For large optimization problems, the Jacobian can be constructed using sparse matrices.

Finally, we set up SLSQP (Kraft, 1988) as the optimizer and solve the optimization problem.

```
# Optimizer
opt = OPT("SLSQP", options={})
```

```
# Optimize
sol = opt(optProb, sens="CS")
print(sol)
```

For more extensive examples and API documentation, please refer to the documentation site for pyOptSparse

Statement of Need

pyOptSparse is a fork of pyOpt (Perez, Jansen, & Martins, 2012). As the name suggests, its primary motivation is to support sparse linear and nonlinear constraints in gradient-based optimization. This sets pyOptSparse apart from other optimization frameworks, such as SciPy (Virtanen et al., 2020) and NLopt (Johnson, 2020), which do not provide the same level of support for sparse constraints. By using string-based indexing, different sub-blocks of the constraint Jacobian can be computed by separate engineering tools, and assembled automatically by pyOptSparse in a sparse fashion. In addition, other frameworks do not offer convenience features, such as user-supplied optimization problem scaling, optimization hot-start, or post-processing utilities. Although pyOptSparse is a general optimization framework, it is tailored to gradient-based optimizations of large-scale problems with sparse constraints.

pyOptSparse has been used extensively in engineering applications, particularly in multidisciplinary design optimization. Researchers have used it to perform aerodynamic shape optimization of aircraft wings (Secco & Martins, 2019), wind turbines (Madsen, Zahle, Sørensen, & Martins, 2019), and aerostructural optimization of an entire aircraft (Brooks, Kenway, & Martins, 2018). pyOptSparse is also supported by OpenMDAO (Gray, Hwang, Martins, Moore, & Naylor, 2019), a popular Python framework for multidisciplinary analysis and optimization. Through OpenMDAO, pyOptSparse has been applied to problems such as low-fidelity aerostructural wing design (Chauhan & Martins, 2018) and aeropropulsive optimization of a boundary-layer ingestion propulsor (Gray & Martins, 2018).

Acknowledgements

We acknowledge the original pyOpt developers' efforts, notably Ruben E. Perez and Peter W. Jansen, who helped lay the code's foundation. We also acknowledge the numerous pyOptSparse users who have contributed to the code over the years.

References

Brooks, T. R., Kenway, G. K. W., & Martins, J. R. R. A. (2018). Benchmark aerostructural models for the study of transonic aircraft wings. *AIAA Journal*, *56*(7), 2840–2855. doi:10.2514/1.J056603

Chauhan, S. S., & Martins, J. R. R. A. (2018). Low-fidelity aerostructural optimization of aircraft wings with a simplified wingbox model using OpenAeroStruct. In *Proceedings of the 6th International Conference on Engineering Optimization, EngOpt 2018* (pp. 418–431). Lisbon, Portugal: Springer. doi:10.1007/978-3-319-97773-7_38

Gray, J. S., Hwang, J. T., Martins, J. R. R. A., Moore, K. T., & Naylor, B. A. (2019). OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization. *Structural and Multidisciplinary Optimization*, *59*(4), 1075–1104. doi:10.1007/s00158-019-02211-z

Gray, J. S., & Martins, J. R. R. A. (2018). Coupled aeropropulsive design optimization of a boundary-layer ingestion propulsor. *The Aeronautical Journal*, *123*(1259), 121–137. doi:10.1017/aer.2018.120

Johnson, S. G. (2020). The NLopt nonlinear-optimization package. Retrieved from http://github.com/stevengj/nlopt

Kraft, D. (1988). *A software package for sequential quadratic programming*. Tech. Rep. DFVLR-FB 88-28, DLR German Aerospace Center.

Madsen, M. H. A., Zahle, F., Sørensen, N. N., & Martins, J. R. R. A. (2019). Multipoint high-fidelity CFD-based aerodynamic shape optimization of a 10 MW wind turbine. *Wind Energy Science*, *4*, 163–192. doi:10.5194/wes-4-163-2019

Martins, J. R. R. A., Sturdza, P., & Alonso, J. J. (2003). The complex-step derivative approximation. *ACM Transactions on Mathematical Software*, *29*(3), 245–262. doi:10.1145/838250.838251

Perez, R. E., Jansen, P. W., & Martins, J. R. R. A. (2012). pyOpt: A Python-based object-oriented framework for nonlinear constrained optimization. *Structural and Multidisciplinary Optimization*, *45*(1), 101–118. doi:10.1007/s00158-011-0666-3

Secco, N. R., & Martins, J. R. R. A. (2019). RANS-based aerodynamic shape optimization of a strut-braced wing with overset meshes. *Journal of Aircraft*, *56*(1), 217–227. doi:10.2514/1.C034934

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., et al. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*, 261–272. doi:10.1038/s41592-019-0686-2