The Journal of Open Source Software

PySLSQP: A transparent Python package for the SLSQP optimization algorithm modernized with utilities for visualization and post-processing

Anugrah Jo Joshy [1] and **John T. Hwang** [2]

1 PhD Candidate, Department of Mechanical and Aerospace Engineering, University of California San Diego **2** Associate Professor, Department of Mechanical and Aerospace Engineering, University of California San Diego

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

Nonlinear programming (NLP) addresses optimization problems involving nonlinear objective and/or constraint functions defined over continuous optimization variables. These functions are assumed to be smooth, i.e., continuously differentiable. Nonlinear programming has applications ranging from aircraft design in engineering to optimizing portfolios in finance and training models in machine learning. Sequential Quadratic Programming (SQP) is one of the most successful classes of algorithms for solving NLP problems. It solves an NLP problem by iteratively formulating and solving a sequence of Quadratic Programming (QP) subproblems. The Sequential Least SQuares Programming algorithm, or SLSQP, has been one of the most widely used SQP algorithms since the 1980s.

We present PySLSQP, a seamless interface for using the SLSQP algorithm from Python, that wraps the original Fortran code sourced from the SciPy repository and provides a host of new features to improve the research utility of the original algorithm. PySLSQP uses a simple yet modern workflow for compiling and using Fortran code from Python. This allows even beginner developers to easily modify the algorithm in Fortran for their specific needs and use in Python the wrapper auto-generated by the workflow.

Some of the additional features offered by PySLSQP include auto-generation of unavailable derivatives using finite differences, independent scaling of the problem variables and functions, access to internal optimization data, live-visualization, saving optimization data from each iteration, warm/hot restarting of optimization, and various other utilities for post-processing.

PySLSQP solves the general nonlinear programming problem:

$$
\begin{array}{rll}
\underset{x \in \mathbb{R}^n}{\text{minimize}} & f(x) & \\
& c_i(x) = 0, & i = 1, ..., m_{eq} \\
\text{subject to} & c_i(x) \geq 0, & i = m_{eq}+1, ..., m \\
& l_i \leq x_i \leq u_i, & i = 1, ..., n
\end{array}
$$

where $x \in \mathbb{R}^n$ is the vector of optimization variables, $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function, $c : \mathbb{R}^n \to \mathbb{R}^m$ is the vector-valued constraint function, and l and u are the vectors containing the lower and upper bounds for the optimization variables, respectively. The first m_{eq} constraints are equalities while the remaining $(m - m_{eq})$ constraints are inequalities.

Statement of need

The original SLSQP algorithm (Kraft, 1988, 1994), implemented in Fortran by Dieter Kraft, has been incorporated into several software packages for optimization across different programming languages. However, the algorithm itself has undergone only minimal improvements and has not kept pace with advancements in programming languages that could enhance its utility. In contrast, other SQP algorithms, such as SNOPT (Gill et al., 2005), which also began development around the same time as SLSQP, have seen continuous improvements. SNOPT has evolved significantly through both algorithmic enhancements and feature additions, becoming one of the leading algorithms for nonlinear programming.

The SLSQP algorithm available in most modern packages is implemented as a black-box function that takes the optimization functions and their derivatives and then outputs the optimized results. These packages do not provide users with any options for tuning the original algorithm or for assessing the progress of an ongoing optimization. This lack of transparency becomes a significant disadvantage for problems with expensive optimization functions or derivatives. Users might have to wait for hours, only to be informed at the end of the optimization procedure that the algorithm could not converge. Several such experiences with multiple research applications in the authors' lab were the primary motivation behind developing the new `PySLSQP` package.

Despite the lack of timely updates to the core algorithm and usability improvements, SLSQP continues to be widely used in research primarily due to its open-source nature and the availability of convenient installation options through packages such as SciPy (Virtanen et al., 2020). Many optimization practitioners use SLSQP for solving medium-sized optimization problems with up to a hundred optimization variables and constraints. Additionally, SLSQP is more successful compared to some of the most advanced algorithms in solving certain classes of optimization problems, such as optimal control problems with a coarse discretization in time.

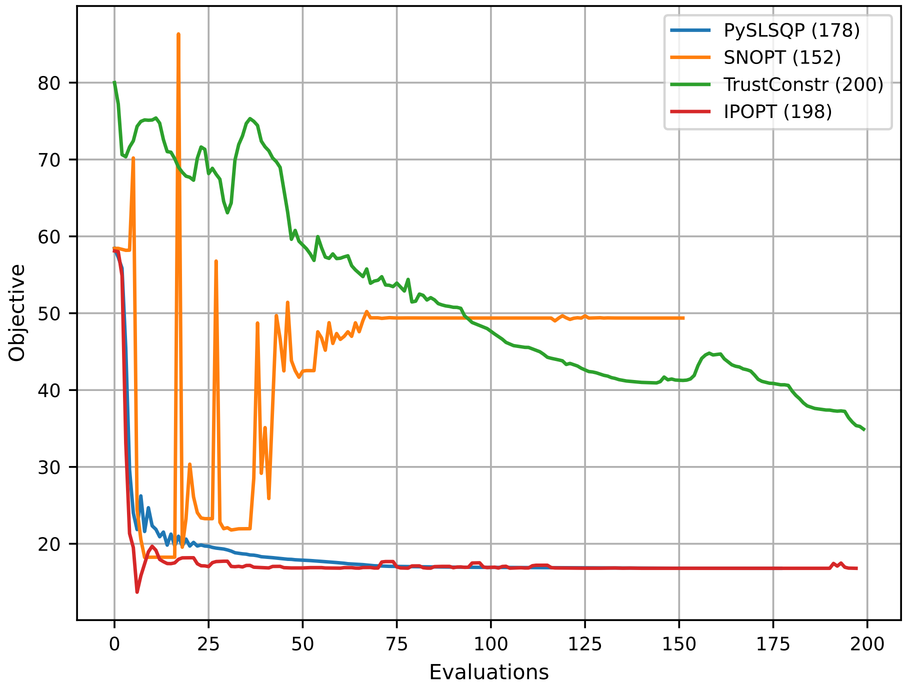

Figure 1: Performance comparison for an optimal control problem.

Figure 1 above compares the convergence behaviors of `PySLSQP` and some of the most advanced algorithms in nonlinear programming on a coarsely discretized optimal control problem. The problem aims to compute the optimal control parameters for a spacecraft landing scenario. The total number of function evaluations is indicated within parentheses in the legend. We see that `PySLSQP` is the only algorithm that solves the problem within the 200 function evaluation limit. Among the algorithms that failed to converge are SNOPT, TrustConstr, and IPOPT. SNOPT is a commercial SQP algorithm, while TrustConstr and IPOPT are open-source Interior Point (IP) algorithms. Although IPOPT appears to have converged in the plot, the solution returned by IPOPT does not satisfy the feasibility criteria. This underscores the relevance of SLSQP even today among state-of-the-art optimization algorithms. This problem is taken from the suite of examples in the modOpt (Joshy & Hwang, 2024) optimization library.

There are several optimization libraries in Python that include the SLSQP algorithm, such as SciPy (Virtanen et al., 2020), NLopt (Johnson, 2024), and pyOpt (Perez et al., 2012). NLopt and pyOpt require users to compile the Fortran code, which greatly deters the majority of users from utilizing SLSQP from these libraries. pyOptSparse (Wu et al., 2020) is a fork of the pyOpt package that supports sparse constraint Jacobians and includes additional optimization utilities for scaling, visualization, and storing optimization history. Most SLSQP users access it through SciPy, which offers precompiled libraries that can be easily installed from PyPI by running `pip install scipy`. However, like other libraries, the SLSQP implementation in SciPy also operates as a black-box providing limited visibility into the progress of optimization or access to internal variables during optimization iterations. This lack of transparency can be a drawback, particularly for users needing more insight into the optimization process.

`PySLSQP` is developed to overcome these limitations by:

- providing a precompiled package through PyPI that can be simply installed with `pip install pyslsqp`,
- offering access to internal optimization variables at each iteration through a save file, and
- informing users about the progress of optimization through a live-updated summary file and visualization.

The Python wrapper for `PySLSQP` is generated by a simple workflow automated on GitHub, which allows even beginner developers to tune the Fortran code for their specific application and extend the current codebase. Offering Python-level access to internal optimization variables such as optimality and feasibility measures, Lagrange multipliers, etc. enables further analysis of an ongoing or completed optimization. `PySLSQP` also features additional utilities for numerical differentiation, scaling, warm/hot restarting, and post-processing.

By addressing the current limitations and providing new capabilities, `PySLSQP` enhances the transparency and usability of the SLSQP algorithm, making it a more powerful and user-friendly tool for solving nonlinear programming problems. `PySLSQP` is now integrated with the modOpt (Joshy & Hwang, 2024) library of optimizers, through which it has successfully solved problems in aircraft design, spacecraft optimal control, and swimming robot design.

Software features

Numerical differentiation

In the absence of user-supplied first-order derivatives of the objective or constraint functions, `PySLSQP` estimates them using first-order finite differencing. Users have the option to set the absolute or relative step size for the finite differences. However, it is generally more efficient for users to provide the exact gradients, if possible, since each finite difference estimation requires $\mathcal{O}(n)$ objective or constraint evaluations. Moreover, finite difference approximations are susceptible to subtractive cancellation errors.

The Journal of Open Source Software

Problem scaling

Scaling of the variables and functions is crucial for the convergence of optimization algorithms. Poor scaling often leads to unsuccessful or extremely slow optimization. PySLSQP enables users to scale the optimization variables, objective, and constraints individually, independent of the user-defined optimization functions. PySLSQP automatically scales the variable bounds and derivatives according to the user-specified scaling for the variables and functions. This allows the user-defined initial guess, bounds, functions, and derivatives to remain the same each time an optimization is run with a different scaling.

Live visualization

Optimization becomes slow for problems with functions or derivatives that are costly to evaluate. In such scenarios, it is important for users to be able to monitor the optimization process to ensure that it is proceeding smoothly. PySLSQP offers the capability to visualize the optimization progress in real-time. This feature allows users to track convergence through optimality and feasibility measures, and to understand how the optimization variables, objective, constraints, Lagrange multipliers, and derivatives evolve during the optimization. An example of a visualization generated by PySLSQP, corresponding to the optimal control problem discussed earlier, is shown in Figure 2 below.

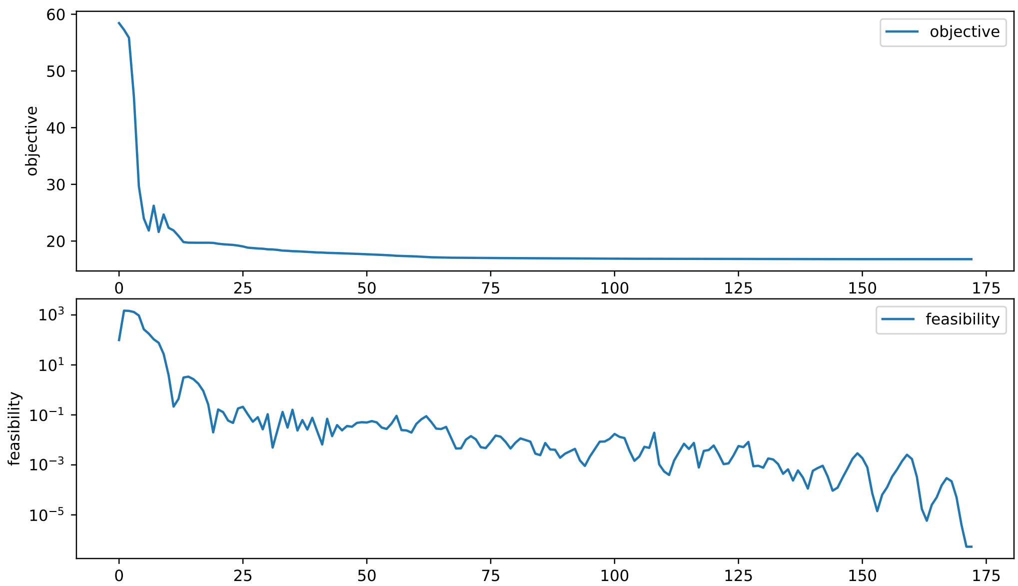

Figure 2: Live visualization of the objective and feasibility.

Access to internal optimization data

In addition to live visualization, PySLSQP provides real-time access to optimization data through dynamically updated summary and save files. PySLSQP generates a summary file that contains a table that is updated at the end of every major iteration. This summary table lists the values of different scalar variables in the algorithm to keep users informed about the current state of optimization.

Users can specify which variables to save in the save file and whether they should be saved for every iteration or only for major iterations. The save file is valuable for analyzing optimization progress, post-processing, or performing warm/hot restarts. It can store all internal optimization variables - including optimization variables, objective, constraints, objective gradient, constraint Jacobian, optimality, feasibility, Lagrange multipliers, and line search step sizes - facilitating advanced analysis of the optimization problem. PySLSQP provides various utilities for working with data from save files, including functions for loading and visualizing variables.

To the best of our knowledge, PySLSQP is the only Python interface to the SLSQP algorithm that provides this level of access to internal optimization information.

Warm/Hot starting

Re-running an optimization that was terminated prematurely can be inefficient and wasteful. For example, if a user desires higher accuracy than was achieved in a previous run, they would need to re-execute the optimization with a smaller accuracy parameter. Similarly, if an optimization terminates upon reaching the iteration limit before achieving the required accuracy, a rerun with a higher limit is necessary to complete the process. Such repeated runs not only consume additional computational resources but also extend the overall time required to achieve the desired results.

To address these scenarios, PySLSQP offers two options for users to efficiently restart an optimization using data from saved files: warm starting and hot starting. In PySLSQP, *warm starting* refers to restarting a previously run optimization using the most recent value of the optimization variables x from a saved file. During a warm start, the initial guess x_0 provided by the user is replaced with the last optimization variable iterate available in the saved file.

Hot starting in PySLSQP involves re-running a previously completed optimization by reusing the function (objective and constraints) and derivative values from a saved file. This method is particularly advantageous when the functions and/or their derivatives are costly to evaluate. A significant benefit of hot starting over warm starting is that the BFGS Hessians approximated by the SLSQP algorithm in a hot-start will follow the same path as in the previous optimization, while also saving the cost of function and derivative evaluations. In contrast, during a warm start, although the algorithm starts from the previous solution x^*, the Hessian is initialized as the identity matrix, which may necessitate more iterations to achieve convergence.

Ease of extension

PySLSQP is implemented in Python and leverages NumPy's *f2py* and the *Meson* build system for compiling and interfacing the underlying Fortran code with Python. The Python setup script automates the build process, making it straightforward for developers to build, install, test, and use PySLSQP after modifying the Fortran code. The package includes GitHub workflows to automatically generate precompiled binaries in the cloud for different system architectures using PyPA's *cibuildwheel* tool. These automated workflows ensure that PySLSQP remains accessible to a broad range of users by providing consistent and reliable installation across various platforms using Linux, macOS, and Windows operating systems. Additionally, this approach allows developers to focus on enhancing the core algorithm and features without the overhead of managing complex build environments, thus fostering an open-source community that can contribute effectively to the development and improvement of the SLSQP algorithm.

A simple example

In this section, we solve a simple optimization problem to illustrate some of the features explained above. In the standard SLSQP problem format presented in the *Summary*, the problem is

$$
\begin{aligned}
\underset{x \in \mathbb{R}^2}{\text{minimize}} \quad & x_1^2 + x_2^2 \\
\text{subject to} \quad & x_1 + x_2 - 1 = 0, \\
& 3x_1 + 2x_2 - 1 \geq 0, \\
\text{with} \quad & m_{eq} = 1,\ l = [0.4, -\infty]^T,\ \text{and}\ u = [+\infty, 0.6]^T.
\end{aligned}
$$

We begin by importing numpy and defining the optimization functions. We will only define the derivatives for the constraints and let PySLSQP approximate the derivatives for the objective function. We then define the constants for the optimization, which include the variable bounds, number of equality constraints, initial guess, and scaling factors.

```python
import numpy as np

def objective(x):
    return x[0]**2 + x[1]**2

def constraints(x):
    return  np.array([x[0] + x[1] - 1, 3*x[0] + 2*x[1] - 1])

def jacobian(x):
    return np.array([[1, 1], [3, 2]])

# Variable bounds
x_lower = np.array([0.4, -np.inf])
x_upper = np.array([np.inf, 0.6])

# Number of equality constraints
m_eq = 1

# Initial guess
x0 = np.array([2,3])

# Scaling factors
x_s = 10.0
o_s =  2.0
c_s = np.array([1., 0.5])
```


Most of the features in PySLSQP are accessed through the optimize function. We now import optimize and solve the problem by calling it with the functions and constants defined above. When calling optimize, we will define the absolute step size for the finite difference approximation of the objective gradient. Additionally, we instruct PySLSQP to save the optimization variables x and the objective value f from each major iteration to a file named save_file.hdf5. Lastly, we configure the arguments to live-visualize the objective f and the variable x_1 during the optimization.

```python
from pyslsqp import optimize

results = optimize(x0, obj=objective, con=constraints, jac=jacobian,
                   meq=m_eq, xl=x_lower, xu=x_upper, finite_diff_abs_step=1e-6,
                   x_scaler=x_s, obj_scaler=o_s, con_scaler=c_s,
                   save_itr='major', save_vars=['majiter', 'x', 'objective'],
                   save_filename="save_file.hdf5",
                   visualize=True, visualize_vars=['objective', 'x[0]'])
```

```python
# Print the returned results dictionary
print(results)
```

Once `optimize` is executed, a summary of the optimization will be printed to the console. The function also returns a dictionary that contains the results of the optimization. By default, PySLSQP writes the summary of major iterations to a file named `slsqp_summary.out`.

For additional usage guidelines, API reference, and installation instructions, please consult the documentation.

Acknowledgements

This work was supported by NASA under award No. 80NSSC23M0217.

References

Gill, P. E., Murray, W., & Saunders, M. A. (2005). SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Review*, *47*(1), 99–131. https://doi.org/10.1137/S0036144504446096

Johnson, S. G. (2024). *The NLopt nonlinear-optimization package*. http://github.com/stevengj/nlopt

Joshy, A. J., & Hwang, J. T. (2024). modOpt: A modular development environment and library for optimization algorithms. *arXiv Preprint arXiv:2410.12942*. https://doi.org/10.48550/arXiv.2410.12942

Kraft, D. (1988). A software package for sequential quadratic programming. *Forschungsbericht-Deutsche Forschungs- Und Versuchsanstalt Fur Luft- Und Raumfahrt*.

Kraft, D. (1994). Algorithm 733: TOMP–fortran modules for optimal control calculations. *ACM Transactions on Mathematical Software (TOMS)*, *20*(3), 262–281. https://doi.org/10.1145/192115.192124

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

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., & others. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in python. *Nature Methods*, *17*(3), 261–272. https://doi.org/10.1038/s41592-019-0686-2

Wu, N., Kenway, G., Mader, C. A., Jasa, J., & Martins, J. R. (2020). pyOptSparse: A python framework for large-scale constrained nonlinear optimization of sparse systems. *Journal of Open Source Software*, *5*(54), 2564. https://doi.org/10.21105/joss.02564