

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

Anugrah Jo Joshy<sup>1</sup> and John T. Hwang<sup>2</sup>

<sup>1</sup> PhD Candidate, Department of Mechanical and Aerospace Engineering, University of California San Diego  
<sup>2</sup> Associate Professor, Department of Mechanical and Aerospace Engineering, University of California San Diego

DOI: [10.xxxxxx/draft](https://doi.org/10.xxxxxx/draft)

## Software

- Review [↗](#)
- Repository [↗](#)
- Archive [↗](#)

Editor: [Open Journals](#) [↗](#)

## Reviewers:

- [@openjournals](#)

Submitted: 01 January 1970

Published: unpublished

## License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](#)).

## 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 purposes 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:

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

where  $x \in \mathbb{R}^n$  is the vector of optimization variables,  $f: \mathbb{R}^n \rightarrow \mathbb{R}$  is the objective function,  $c: \mathbb{R}^n \rightarrow \mathbb{R}^m$  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.

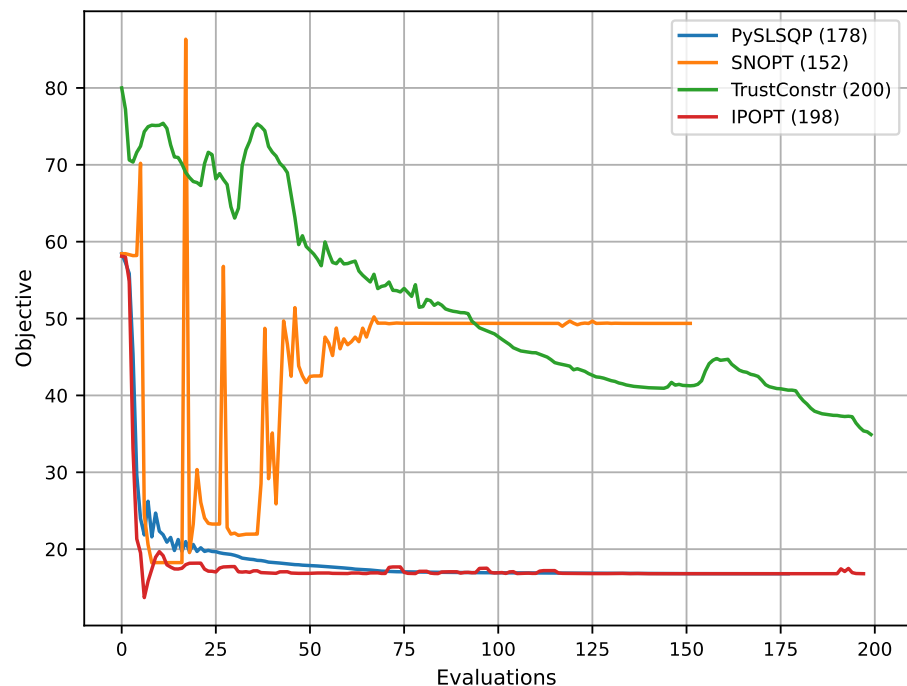


Figure 1: Performance comparison for an optimal control problem.

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 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 (Josh, 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 interfaced with the modOpt (Josh, 2024) library of optimizers, through which it has successfully solved problems in aircraft design, as well as aircraft and spacecraft optimal control.

## 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.

## 107 Problem scaling

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

## 115 Live visualization

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

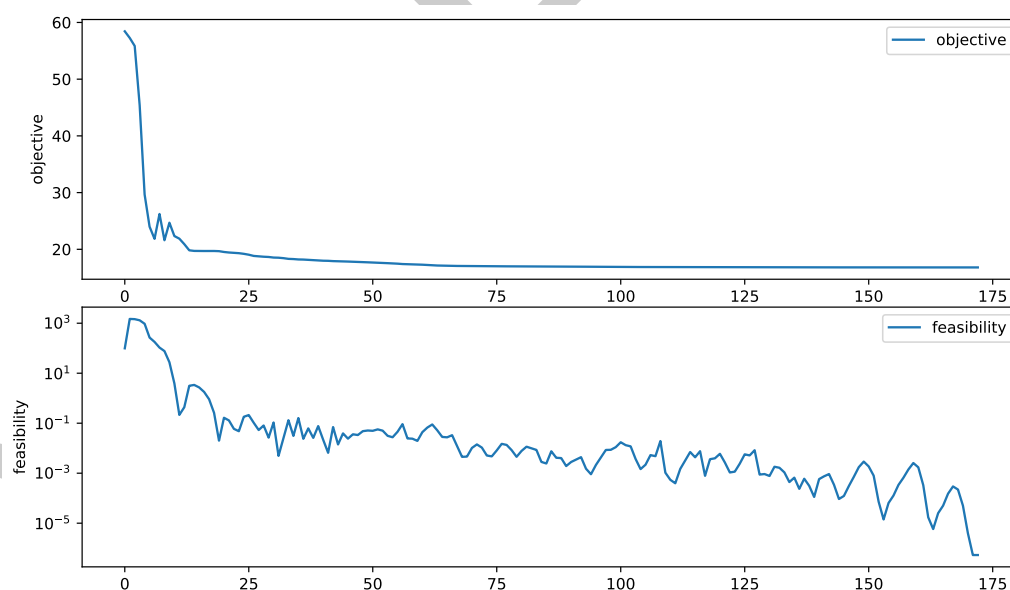


Figure 2: Live visualization of the objective and feasibility.

## 124 Access to internal optimization data

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

130 Users can specify which variables to save in the save file and whether they should be saved for  
131 every iteration or only for major iterations. The save file is valuable for analyzing optimization  
132 progress, post-processing, or performing warm/hot restarts. It can store all internal optimization  
133 variables - including optimization variables, objective, constraints, objective gradient, constraint  
134 Jacobian, optimality, feasibility, Lagrange multipliers, and line search step sizes - facilitating

135 advanced analysis of the optimization problem. PySLSQP provides various utilities for working  
136 with data from save files, including functions for loading and visualizing variables.

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

## 139 Warm/Hot starting

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

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

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

## 160 Ease of extension

161 PySLSQP is implemented in Python and relies on NumPy's *f2py* and the *Meson* build system for  
162 compiling and interfacing the underlying Fortran code with Python. The Python setup script  
163 automates the build process, making it straightforward for developers to build, install, and use  
164 PySLSQP after making modifications to the Fortran code. The package also includes GitHub  
165 workflows for automatically generating precompiled binaries in the cloud for different system  
166 architectures using PyPA's *cibuildwheel* tool. These automated workflows ensure that PySLSQP  
167 remains accessible to a broad range of users by providing consistent and reliable installation  
168 across different platforms. Additionally, this approach allows developers to focus on enhancing  
169 the core algorithm and features without the overhead of managing complex build environments,  
170 thus fostering an open-source community that can contribute effectively to the development  
171 of the SLSQP algorithm.

## 172 A simple example

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

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

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

```

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])

```

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

```

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]'])

```

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

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

190 For additional usage guidelines, API reference, and installation instructions, please consult the  
191 [documentation](#).

## 192 Acknowledgements

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

## 194 References

- 195 Gill, P. E., Murray, W., & Saunders, M. A. (2005). SNOPT: An SQP algorithm for large-  
196 scale constrained optimization. *SIAM Review*, 47(1), 99–131. [https://doi.org/10.1137/](https://doi.org/10.1137/S0036144504446096)  
197 [S0036144504446096](https://doi.org/10.1137/S0036144504446096)
- 198 Johnson, S. G. (2024). *The NLOpt nonlinear-optimization package*. [http://github.com/](http://github.com/stevengj/nlopt)  
199 [stevengj/nlopt](http://github.com/stevengj/nlopt)
- 200 Joshy, A. J. (2024). *A MODular development environment and library for OPTimization*  
201 *algorithms*. <https://github.com/LSDOlab/modopt>
- 202 Kraft, D. (1988). A software package for sequential quadratic programming. *Forschungsbericht-*  
203 *Deutsche Forschungs- Und Versuchsanstalt Fur Luft- Und Raumfahrt*.
- 204 Kraft, D. (1994). Algorithm 733: TOMP—fortran modules for optimal control calculations.  
205 *ACM Transactions on Mathematical Software (TOMS)*, 20(3), 262–281. [https://doi.org/](https://doi.org/10.1145/192115.192124)  
206 [10.1145/192115.192124](https://doi.org/10.1145/192115.192124)
- 207 Perez, R. E., Jansen, P. W., & Martins, J. R. (2012). pyOpt: A python-based object-  
208 oriented framework for nonlinear constrained optimization. *Structural and Multidisciplinary*  
209 *Optimization*, 45, 101–118. <https://doi.org/10.1007/s00158-011-0666-3>
- 210 Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D.,  
211 Burovski, E., Peterson, P., Weckesser, W., Bright, J., & others. (2020). SciPy 1.0:  
212 Fundamental algorithms for scientific computing in python. *Nature Methods*, 17(3),  
213 261–272. <https://doi.org/10.1038/s41592-019-0686-2>
- 214 Wu, N., Kenway, G., Mader, C. A., Jasa, J., & Martins, J. R. (2020). pyOptSparse: A python  
215 framework for large-scale constrained nonlinear optimization of sparse systems. *Journal of*  
216 *Open Source Software*, 5(54), 2564. <https://doi.org/10.21105/joss.02564>