Search the docs ...

scipy
scipy.cluster
scipy.constants
scipy.datasets
scipy.fft
scipy.fftpack
scipy.integrate
scipy.interpolate
scipy.io
scipy.linalg
scipy.misc
scipy.ndimage
scipy.odr
scipy.optimize
scipy.signal
scipy.sparse
scipy.spatial
scipy.special
scipy.stats

scipy.optimize.linprog

scipy.optimize.linprog(*c, A_ub=None, b_ub=None, A_eq=None, b_eq=None, bounds=None, method='highs', callback=None, options=None, x0=None, integrality=None*) [source]

Linear programming: minimize a linear objective function subject to linear equality and inequality constraints.

Linear programming solves problems of the following form:

$$\begin{aligned} \min_x \ & c^T x \\ \text{such that } & A_{ub} x \leq b_{ub}, \\ & A_{eq} x = b_{eq}, \\ & l \leq x \leq u, \end{aligned}$$

where x is a vector of decision variables; c, b_{ub}, b_{eq}, l, and u are vectors; and A_{ub} and A_{eq} are matrices.

Alternatively, that's:

- minimize

```
c @ x
```

- such that

```
A_ub @ x <= b_ub
A_eq @ x == b_eq
lb <= x <= ub
```

Note that by default `lb = 0` and `ub = None`. Other bounds can be specified with `bounds`.

Parameters: **c : *1-D array***

The coefficients of the linear objective function to be minimized.

A_ub : *2-D array, optional*

The inequality constraint matrix. Each row of `A_ub` specifies the coefficients of a linear inequality constraint on `x`.

b_ub : *1-D array, optional*

The inequality constraint vector. Each element represents an upper bound on the corresponding value of `A_ub @ x`.

A_eq : *2-D array, optional*

The equality constraint matrix. Each row of `A_eq` specifies the coefficients of a linear equality constraint on `x`.

b_eq : *1-D array, optional*

The equality constraint vector. Each element of `A_eq @ x` must equal the corresponding element of `b_eq`.

bounds : *sequence, optional*

A sequence of `(min, max)` pairs for each element in `x`, defining the minimum and maximum values of that decision variable. If a single tuple `(min, max)` is provided, then `min` and `max` will serve as bounds for all decision variables. Use `None` to indicate that there is no bound. For instance, the default bound `(0, None)` means that all decision variables are non-negative, and the pair `(None, None)` means no bounds at all, i.e. all variables are allowed to be any real.

method : *str, optional*

The algorithm used to solve the standard form problem. 'highs' (default), 'highs-ds', 'highs-ipm', 'interior-point' (legacy), 'revised simplex' (legacy), and 'simplex' (legacy) are supported. The legacy methods are deprecated and will be removed in SciPy 1.11.0.

callback : *callable, optional*

If a callback function is provided, it will be called at least once per iteration of the algorithm. The callback function must accept a single `scipy.optimize.OptimizeResult` consisting of the following fields:

x : *1-D array*

The current solution vector.

fun : *float*

The current value of the objective function `c @ x`.

success : *bool*

`True` when the algorithm has completed successfully.

slack : *1-D array*

The (nominally positive) values of the slack, `b_ub - A_ub @ x`.

con : *1-D array*

The (nominally zero) residuals of the equality constraints, `b_eq - A_eq @ x`.

phase : *int*

The phase of the algorithm being executed.

status : *int*

An integer representing the status of the algorithm.

`0` : Optimization proceeding nominally.

`1` : Iteration limit reached.

`2` : Problem appears to be infeasible.

`3` : Problem appears to be unbounded.

`4` : Numerical difficulties encountered.

nit : *int*

The current iteration number.

message : *str*

A string descriptor of the algorithm status.

Callback functions are not currently supported by the HiGHS methods.

options : *dict, optional*

A dictionary of solver options. All methods accept the following options:

maxiter : *int*

Maximum number of iterations to perform. Default: see method-specific documentation.

disp : *bool*

Set to `True` to print convergence messages. Default: `False`.

presolve : *bool*

Set to `False` to disable automatic presolve. Default: `True`.

All methods except the HiGHS solvers also accept:

tol : *float*

A tolerance which determines when a residual is “close enough” to zero to be considered exactly zero.

autoscale : *bool*

Set to `True` to automatically perform equilibration. Consider using this option if the numerical values in the constraints are separated by several orders of magnitude. Default: `False`.

rr : *bool*

Set to `False` to disable automatic redundancy removal. Default: `True`.

rr_method : *string*

Method used to identify and remove redundant rows from the equality constraint matrix after presolve. For problems with dense input, the available methods for redundancy removal are:

“SVD”:

Repeatedly performs singular value decomposition on the matrix, detecting redundant rows based on nonzeros in the left singular vectors that correspond with zero singular values. May be fast when the matrix is nearly full rank.

“pivot”:

Uses the algorithm presented in [5] to identify redundant rows.

“ID”:

Uses a randomized interpolative decomposition. Identifies columns of the matrix transpose not used in a full-rank interpolative decomposition of the matrix.

None:

Uses “svd” if the matrix is nearly full rank, that is, the difference between the matrix rank and the number of rows is less than five. If not, uses “pivot”. The behavior of this default is subject to change without prior notice.

Default: None. For problems with sparse input, this option is ignored, and the pivot-based algorithm presented in [5] is used.

For method-specific options, see `show_options('linprog')`.

x0 : *1-D array, optional*

Guess values of the decision variables, which will be refined by the optimization algorithm. This argument is currently used only by the ‘revised simplex’ method, and can only be used if *x0* represents a basic feasible solution.

integrality : *1-D array or int, optional*

Indicates the type of integrality constraint on each decision variable.
`0` : Continuous variable; no integrality constraint.
`1` : Integer variable; decision variable must be an integer within *bounds*.
`2` : Semi-continuous variable; decision variable must be within *bounds* or take value `0`.
`3` : Semi-integer variable; decision variable must be an integer within *bounds* or take value `0`.
By default, all variables are continuous.
For mixed integrality constraints, supply an array of shape *c.shape*. To infer a constraint on each decision variable from shorter inputs, the argument will be broadcasted to *c.shape* using *np.broadcast_to*.
This argument is currently used only by the `'highs'` method and ignored otherwise.

Returns: **res : *OptimizeResult***

A `scipy.optimize.OptimizeResult` consisting of the fields below. Note that the return types of the fields may depend on whether the optimization was successful, therefore it is recommended to check *OptimizeResult.status* before relying on the other fields:

x : *1-D array*

The values of the decision variables that minimizes the objective function while satisfying the constraints.

fun : *float*

The optimal value of the objective function `c @ x`.

slack : *1-D array*

The (nominally positive) values of the slack variables, `b_ub - A_ub @ x`.

con : *1-D array*

The (nominally zero) residuals of the equality constraints, `b_eq - A_eq @ x`.

success : *bool*

`True` when the algorithm succeeds in finding an optimal solution.

status : *int*

An integer representing the exit status of the algorithm.
`0` : Optimization terminated successfully.
`1` : Iteration limit reached.
`2` : Problem appears to be infeasible.
`3` : Problem appears to be unbounded.
`4` : Numerical difficulties encountered.

nit : *int*

The total number of iterations performed in all phases.

message : *str*

A string descriptor of the exit status of the algorithm.

See also

`show_options`

Additional options accepted by the solvers.

Notes

This section describes the available solvers that can be selected by the 'method' parameter.

'highs-ds' and *'highs-ipm'* are interfaces to the HiGHS simplex and interior-point method solvers [13], respectively. *'highs'* (default) chooses between the two automatically. These are the fastest linear programming solvers in SciPy, especially for large, sparse problems; which of these two is faster is problem-dependent. The other solvers (*'interior-point'*, *'revised simplex'*, and *'simplex'*) are legacy methods and will be removed in SciPy 1.11.0.

Method *highs-ds* is a wrapper of the C++ high performance dual revised simplex implementation (HSOL) [13], [14]. Method *highs-ipm* is a wrapper of a C++ implementation of an **i**nterior-**p**oint **m**ethod [13]; it features a crossover routine, so it is as accurate as a simplex solver. Method *highs* chooses between the two automatically. For new code involving `linprog`, we recommend explicitly choosing one of these three method values.

New in version 1.6.0.

Method *interior-point* uses the primal-dual path following algorithm as outlined in [4]. This algorithm supports sparse constraint matrices and is typically faster than the simplex methods, especially for large, sparse problems. Note, however, that the solution returned may be slightly less accurate than those of the simplex methods and will not, in general, correspond with a vertex of the polytope defined by the constraints.

New in version 1.0.0.

Method *revised simplex* uses the revised simplex method as described in [9], except that a factorization [11] of the basis matrix, rather than its inverse, is efficiently maintained and used to solve the linear systems at each iteration of the algorithm.

New in version 1.3.0.

Method *simplex* uses a traditional, full-tableau implementation of Dantzig's simplex algorithm [1], [2] (*not* the Nelder-Mead simplex). This algorithm is included for backwards compatibility and educational purposes.

New in version 0.15.0.

Before applying *interior-point*, *revised simplex*, or *simplex*, a presolve procedure based on [8] attempts to identify trivial infeasibilities, trivial unboundedness, and potential problem simplifications. Specifically, it checks for:

- rows of zeros in `A_eq` or `A_ub`, representing trivial constraints;
- columns of zeros in `A_eq` *and* `A_ub`, representing unconstrained variables;
- column singletons in `A_eq`, representing fixed variables; and
- column singletons in `A_ub`, representing simple bounds.

If presolve reveals that the problem is unbounded (e.g. an unconstrained and unbounded variable has negative cost) or infeasible (e.g., a row of zeros in `A_eq` corresponds with a nonzero in `b_eq`), the solver terminates with the appropriate status code. Note that presolve terminates as soon as any sign of

unboundedness is detected; consequently, a problem may be reported as unbounded when in reality the problem is infeasible (but infeasibility has not been detected yet). Therefore, if it is important to know whether the problem is actually infeasible, solve the problem again with option `presolve=False`.

If neither infeasibility nor unboundedness are detected in a single pass of the presolve, bounds are tightened where possible and fixed variables are removed from the problem. Then, linearly dependent rows of the `A_eq` matrix are removed, (unless they represent an infeasibility) to avoid numerical difficulties in the primary solve routine. Note that rows that are nearly linearly dependent (within a prescribed tolerance) may also be removed, which can change the optimal solution in rare cases. If this is a concern, eliminate redundancy from your problem formulation and run with option `rr=False` or `presolve=False`.

Several potential improvements can be made here: additional presolve checks outlined in [8] should be implemented, the presolve routine should be run multiple times (until no further simplifications can be made), and more of the efficiency improvements from [5] should be implemented in the redundancy removal routines.

After presolve, the problem is transformed to standard form by converting the (tightened) simple bounds to upper bound constraints, introducing non-negative slack variables for inequality constraints, and expressing unbounded variables as the difference between two non-negative variables. Optionally, the problem is automatically scaled via equilibration [12]. The selected algorithm solves the standard form problem, and a postprocessing routine converts the result to a solution to the original problem.

References

[1] Dantzig, George B., Linear programming and extensions. Rand Corporation Research Study Princeton Univ. Press, Princeton, NJ, 1963

[2] Hillier, S.H. and Lieberman, G.J. (1995), “Introduction to Mathematical Programming”, McGraw-Hill, Chapter 4.

[3] Bland, Robert G. New finite pivoting rules for the simplex method. Mathematics of Operations Research (2), 1977: pp. 103-107.

[4] Andersen, Erling D., and Knud D. Andersen. “The MOSEK interior point optimizer for linear programming: an implementation of the homogeneous algorithm.” High performance optimization. Springer US, 2000. 197-232.

[5](1,2,3) Andersen, Erling D. “Finding all linearly dependent rows in large-scale linear programming.” Optimization Methods and Software 6.3 (1995): 219-227.

[6] Freund, Robert M. “Primal-Dual Interior-Point Methods for Linear Programming based on Newton’s Method.” Unpublished Course Notes, March 2004. Available 2/25/2017 at https://ocw.mit.edu/courses/sloan-school-of-management/15-084j-nonlinear-programming-spring-2004/lecture-notes/lec14_int_pt_mthd.pdf

[7] Fourer, Robert. “Solving Linear Programs by Interior-Point Methods.” Unpublished Course Notes, August 26, 2005. Available 2/25/2017 at http://www.4er.org/CourseNotes/Book%20B/B-III.pdf

[8](1,2) Andersen, Erling D., and Knud D. Andersen. “Presolving in linear programming.” Mathematical Programming 71.2 (1995): 221-245.

[9] Bertsimas, Dimitris, and J. Tsitsiklis. “Introduction to linear programming.” Athena Scientific 1 (1997): 997.

[10] Andersen, Erling D., et al. Implementation of interior point methods for large scale linear programming. HEC/Universite de Geneve, 1996.

[11] Bartels, Richard H. “A stabilization of the simplex method.” Journal in Numerische Mathematik 16.5 (1971): 414-434.

[12] Tomlin, J. A. “On scaling linear programming problems.” Mathematical Programming Study 4 (1975): 146-166.

[13](1,2,3) Huangfu, Q., Galabova, I., Feldmeier, M., and Hall, J. A. J. “HiGHS - high performance software for linear optimization.” https://highs.dev/

[14] Huangfu, Q. and Hall, J. A. J. “Parallelizing the dual revised simplex method.” Mathematical Programming Computation, 10 (1), 119-142, 2018. DOI: 10.1007/s12532-017-0130-5

Examples

Consider the following problem:

$$\begin{aligned} \min_{x_0, x_1} \ & -x_0 + 4x_1 \\ \text{such that} \ & -3x_0 + x_1 \leq 6, \\ & -x_0 - 2x_1 \geq -4, \\ & x_1 \geq -3. \end{aligned}$$

The problem is not presented in the form accepted by **linprog**. This is easily remedied by converting the "greater than" inequality constraint to a "less than" inequality constraint by multiplying both sides by a factor of -1. Note also that the last constraint is really the simple bound $-3 \leq x_1 \leq \infty$. Finally, since there are no bounds on x_0, we must explicitly specify the bounds $-\infty \leq x_0 \leq \infty$, as the default is for variables to be non-negative. After collecting coeffecients into arrays and tuples, the input for this problem is:

```
>>> from scipy.optimize import linprog
>>> c = [-1, 4]
>>> A = [[-3, 1], [1, 2]]
>>> b = [6, 4]
>>> x0_bounds = (None, None)
>>> x1_bounds = (-3, None)
>>> res = linprog(c, A_ub=A, b_ub=b, bounds=[x0_bounds, x1_bounds])
>>> res.fun
-22.0
>>> res.x
array([10., -3.])
>>> res.message
'Optimization terminated successfully. (HiGHS Status 7: Optimal)'
```

The marginals (AKA dual values / shadow prices / Lagrange multipliers) and residuals (slacks) are also available.

```
>>> res.ineqlin
  residual: [ 3.900e+01  0.000e+00]
 marginals: [-0.000e+00 -1.000e+00]
```

For example, because the marginal associated with the second inequality constraint is -1, we expect the optimal value of the objective function to decrease by `eps` if we add a small amount `eps` to the right hand side of the second inequality constraint:

```
>>> eps = 0.05
>>> b[1] += eps
>>> linprog(c, A_ub=A, b_ub=b, bounds=[x0_bounds, x1_bounds]).fun
-22.05
```

Also, because the residual on the first inequality constraint is 39, we can decrease the right hand side of the first constraint by 39 without affecting the optimal solution.

```
>>> b = [6, 4]  # reset to original values
>>> b[0] -= 39
>>> linprog(c, A_ub=A, b_ub=b, bounds=[x0_bounds, x1_bounds]).fun
-22.0
```

Previous
scipy.optimize.milp

Next
linprog(method='simplex')