6.18.4 Optimization

Scalar Functions Optimization

`minimize_scalar`(fun[, bracket, bounds, …])	Minimization of scalar function of one variable.

scipy.optimize.minimize_scalar

`scipy.optimize.`**`minimize_scalar`**(*fun*, *bracket=None*, *bounds=None*, *args=()*, *method='brent'*, *tol=None*, *options=None*)

Minimization of scalar function of one variable.

Parameters

fun [callable] Objective function. Scalar function, must return a scalar.

bracket [sequence, optional] For methods 'brent' and 'golden', `bracket` defines the bracketing interval and can either have three items `(a, b, c)` so that `a < b < c` and `fun(b) < fun(a), fun(c)` or two items `a` and `c` which are assumed to be a starting interval for a downhill bracket search (see `bracket`); it doesn't always mean that the obtained solution will satisfy `a <= x <= c`.

bounds [sequence, optional] For method 'bounded', *bounds* is mandatory and must have two items corresponding to the optimization bounds.

args [tuple, optional] Extra arguments passed to the objective function.

method [str or callable, optional] Type of solver. Should be one of:

- 'Brent' *(see here)*
- 'Bounded' *(see here)*
- 'Golden' *(see here)*
- custom - a callable object (added in version 0.14.0), see below

tol [float, optional] Tolerance for termination. For detailed control, use solver-specific options.

options [dict, optional] A dictionary of solver options.

  maxiter [int] Maximum number of iterations to perform.

  disp [bool] Set to True to print convergence messages.

See `show_options` for solver-specific options.

Returns

res [OptimizeResult] The optimization result represented as a `OptimizeResult` object. Important attributes are: `x` the solution array, `success` a Boolean flag indicating if the optimizer exited successfully and `message` which describes the cause of the termination. See `OptimizeResult` for a description of other attributes.

See also:

`minimize`

Interface to minimization algorithms for scalar multivariate functions

`show_options`

Additional options accepted by the solvers

Notes

This section describes the available solvers that can be selected by the 'method' parameter. The default method is *Brent*.

Method *Brent* uses Brent's algorithm to find a local minimum. The algorithm uses inverse parabolic interpolation when possible to speed up convergence of the golden section method.

Method *Golden* uses the golden section search technique. It uses analog of the bisection method to decrease the bracketed interval. It is usually preferable to use the *Brent* method.

Method *Bounded* can perform bounded minimization. It uses the Brent method to find a local minimum in the interval x1 < xopt < x2.

Custom minimizers

It may be useful to pass a custom minimization method, for example when using some library frontend to minimize_scalar. You can simply pass a callable as the `method` parameter.

The callable is called as `method(fun, args, **kwargs, **options)` where `kwargs` corresponds to any other parameters passed to `minimize` (such as `bracket`, *tol*, etc.), except the *options* dict, which has its contents also passed as *method* parameters pair by pair. The method shall return an `OptimizeResult` object.

The provided *method* callable must be able to accept (and possibly ignore) arbitrary parameters; the set of parameters accepted by `minimize` may expand in future versions and then these parameters will be passed to the method. You can find an example in the scipy.optimize tutorial.

New in version 0.11.0.

Examples

Consider the problem of minimizing the following function.

```
>>> def f(x):
...     return (x - 2) * x * (x + 2)**2
```

Using the *Brent* method, we find the local minimum as:

```
>>> from scipy.optimize import minimize_scalar
>>> res = minimize_scalar(f)
>>> res.x
1.28077640403
```

Using the *Bounded* method, we find a local minimum with specified bounds as:

```
>>> res = minimize_scalar(f, bounds=(-3, -1), method='bounded')
>>> res.x
-2.0000002026
```

The `minimize_scalar` function supports the following methods:

minimize_scalar(method='brent')

`scipy.optimize.`**`minimize_scalar`**(*fun*, *args=()*, *method='brent'*, *tol=None*, *options={'func': None, 'brack': None, 'xtol': 1.48e-08, 'maxiter': 500}*)

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize_scalar`

Options

maxiter	[int] Maximum number of iterations to perform.
xtol	[float] Relative error in solution *xopt* acceptable for convergence.

Notes

Uses inverse parabolic interpolation when possible to speed up convergence of golden section method.

minimize_scalar(method='bounded')

`scipy.optimize.`**`minimize_scalar`**(*fun*, *bounds=None*, *args=()*, *method='bounded'*, *tol=None*, *options={'func': None, 'xatol': 1e-05, 'maxiter': 500, 'disp': 0}*)

See also:

For documentation for the rest of the parameters, see *`scipy.optimize.minimize_scalar`*

Options

maxiter [int] Maximum number of iterations to perform.

disp: int, optional

If non-zero, print messages.
0 : no message printing. 1 : non-convergence notification messages only. 2 : print a message on convergence too. 3 : print iteration results.

xatol [float] Absolute error in solution *xopt* acceptable for convergence.

minimize_scalar(method='golden')

`scipy.optimize.`**`minimize_scalar`**(*fun*, *args=()*, *method='golden'*, *tol=None*, *options={'func': None, 'brack': None, 'xtol': 1.4901161193847656e-08, 'maxiter': 5000}*)

See also:

For documentation for the rest of the parameters, see *`scipy.optimize.minimize_scalar`*

Options

maxiter [int] Maximum number of iterations to perform.

xtol [float] Relative error in solution *xopt* acceptable for convergence.

Local (Multivariate) Optimization

`minimize`(fun, x0[, args, method, jac, hess, …])	Minimization of scalar function of one or more variables.

scipy.optimize.minimize

`scipy.optimize.`**`minimize`**(*fun*, *x0*, *args=()*, *method=None*, *jac=None*, *hess=None*, *hessp=None*, *bounds=None*, *constraints=()*, *tol=None*, *callback=None*, *options=None*)

Minimization of scalar function of one or more variables.

Parameters

fun [callable] The objective function to be minimized.

```
fun(x, *args) -> float
```

where x is an 1-D array with shape (n,) and *args* is a tuple of the fixed parameters needed to completely specify the function.

x0 [ndarray, shape (n,)] Initial guess. Array of real elements of size (n,), where 'n' is the number of independent variables.

args [tuple, optional] Extra arguments passed to the objective function and its derivatives (*fun*, *jac* and *hess* functions).

method [str or callable, optional] Type of solver. Should be one of

- 'Nelder-Mead' *(see here)*
- 'Powell' *(see here)*
- 'CG' *(see here)*
- 'BFGS' *(see here)*
- 'Newton-CG' *(see here)*
- 'L-BFGS-B' *(see here)*
- 'TNC' *(see here)*
- 'COBYLA' *(see here)*
- 'SLSQP' *(see here)*
- 'trust-constr'*(see here)*
- 'dogleg' *(see here)*
- 'trust-ncg' *(see here)*
- 'trust-exact' *(see here)*
- 'trust-krylov' *(see here)*
- custom - a callable object (added in version 0.14.0), see below for description.

If not given, chosen to be one of `BFGS`, `L-BFGS-B`, `SLSQP`, depending if the problem has constraints or bounds.

jac [{callable, '2-point', '3-point', 'cs', bool}, optional] Method for computing the gradient vector. Only for CG, BFGS, Newton-CG, L-BFGS-B, TNC, SLSQP, dogleg, trust-ncg, trust-krylov, trust-exact and trust-constr. If it is a callable, it should be a function that returns the gradient vector:

```
jac(x, *args) -> array_like, shape (n,)
```

where x is an array with shape (n,) and *args* is a tuple with the fixed parameters. Alternatively, the keywords {'2-point', '3-point', 'cs'} select a finite difference scheme for numerical estimation of the gradient. Options '3-point' and 'cs' are available only to 'trust-constr'. If *jac* is a Boolean and is True, *fun* is assumed to return the gradient along with the objective function. If False, the gradient will be estimated using '2-point' finite difference estimation.

hess [{callable, '2-point', '3-point', 'cs', HessianUpdateStrategy}, optional] Method for computing the Hessian matrix. Only for Newton-CG, dogleg, trust-ncg, trust-krylov, trust-exact and trust-constr. If it is callable, it should return the Hessian matrix:

```
hess(x, *args) -> {LinearOperator, spmatrix, array}, (n,
n)
```

where x is a (n,) ndarray and *args* is a tuple with the fixed parameters. LinearOperator and sparse matrix returns are allowed only for 'trust-constr' method. Alternatively, the keywords {'2-point', '3-point', 'cs'} select a finite difference scheme for numerical estimation. Or, objects implementing *`HessianUpdateStrategy`* interface can be used to approximate the Hessian. Available quasi-Newton methods implementing this interface are:

- *`BFGS`*;
- *`SR1`*.

Whenever the gradient is estimated via finite-differences, the Hessian cannot be estimated with options {'2-point', '3-point', 'cs'} and needs to be estimated using one of the quasi-Newton strategies. Finite-difference options {'2-point', '3-point', 'cs'} and *`HessianUpdateStrategy`* are available only for 'trust-constr' method.

hessp [callable, optional] Hessian of objective function times an arbitrary vector p. Only for Newton-CG, trust-ncg, trust-krylov, trust-constr. Only one of *hessp* or *hess* needs to be given. If *hess* is provided, then *hessp* will be ignored. *hessp* must compute the Hessian times an arbitrary vector:

```
hessp(x, p, *args) -> ndarray shape (n,)
```

where x is a (n,) ndarray, p is an arbitrary vector with dimension (n,) and *args* is a tuple with the fixed parameters.

bounds [sequence or *`Bounds`*, optional] Bounds on variables for L-BFGS-B, TNC, SLSQP and trust-constr methods. There are two ways to specify the bounds:

1. Instance of *`Bounds`* class.

2. Sequence of `(min, max)` pairs for each element in *x*. None is used to specify no bound.

constraints
[{Constraint, dict} or List of {Constraint, dict}, optional] Constraints definition (only for COBYLA, SLSQP and trust-constr). Constraints for 'trust-constr' are defined as a single object or a list of objects specifying constraints to the optimization problem. Available constraints are:

- *`LinearConstraint`*
- *`NonlinearConstraint`*

Constraints for COBYLA, SLSQP are defined as a list of dictionaries. Each dictionary with fields:

type [str] Constraint type: 'eq' for equality, 'ineq' for inequality.
fun [callable] The function defining the constraint.
jac [callable, optional] The Jacobian of *fun* (only for SLSQP).
args [sequence, optional] Extra arguments to be passed to the function and Jacobian.

Equality constraint means that the constraint function result is to be zero whereas inequality means that it is to be non-negative. Note that COBYLA only supports inequality constraints.

tol [float, optional] Tolerance for termination. For detailed control, use solver-specific options.

options [dict, optional] A dictionary of solver options. All methods accept the following generic options:

maxiter [int] Maximum number of iterations to perform.
disp [bool] Set to True to print convergence messages.

For method-specific options, see *`show_options`*.

callback [callable, optional] Called after each iteration. For 'trust-constr' it is a callable with the signature:

```
callback(xk, OptimizeResult state) -> bool
```

where `xk` is the current parameter vector. and `state` is an *`OptimizeResult`* object, with the same fields as the ones from the return. If callback returns True the algorithm execution is terminated. For all the other methods, the signature is:

```
callback(xk)
```

where `xk` is the current parameter vector.

Returns

res [OptimizeResult] The optimization result represented as a `OptimizeResult` object. Important attributes are: `x` the solution array, `success` a Boolean flag indicating if the optimizer exited successfully and `message` which describes the cause of the termination. See *`OptimizeResult`* for a description of other attributes.

See also:

`minimize_scalar`

Interface to minimization algorithms for scalar univariate functions

`show_options`

Additional options accepted by the solvers

Notes

This section describes the available solvers that can be selected by the 'method' parameter. The default method is *BFGS*.

Unconstrained minimization

Method *Nelder-Mead* uses the Simplex algorithm [1], [2]. This algorithm is robust in many applications. However, if numerical computation of derivative can be trusted, other algorithms using the first and/or second derivatives information might be preferred for their better performance in general.

Method *Powell* is a modification of Powell's method [3], [4] which is a conjugate direction method. It performs sequential one-dimensional minimizations along each vector of the directions set (*direc* field in *options* and *info*), which is updated at each iteration of the main minimization loop. The function need not be differentiable, and no derivatives are taken.

Method *CG* uses a nonlinear conjugate gradient algorithm by Polak and Ribiere, a variant of the Fletcher-Reeves method described in [5] pp. 120-122. Only the first derivatives are used.

Method *BFGS* uses the quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shanno (BFGS) [5] pp. 136. It uses the first derivatives only. BFGS has proven good performance even for non-smooth optimizations. This method also returns an approximation of the Hessian inverse, stored as *hess_inv* in the OptimizeResult object.

Method *Newton-CG* uses a Newton-CG algorithm [5] pp. 168 (also known as the truncated Newton method). It uses a CG method to the compute the search direction. See also *TNC* method for a box-constrained minimization with a similar algorithm. Suitable for large-scale problems.

Method *dogleg* uses the dog-leg trust-region algorithm [5] for unconstrained minimization. This algorithm requires the gradient and Hessian; furthermore the Hessian is required to be positive definite.

Method *trust-ncg* uses the Newton conjugate gradient trust-region algorithm [5] for unconstrained minimization. This algorithm requires the gradient and either the Hessian or a function that computes the product of the Hessian with a given vector. Suitable for large-scale problems.

Method *trust-krylov* uses the Newton GLTR trust-region algorithm [14], [15] for unconstrained minimization. This algorithm requires the gradient and either the Hessian or a function that computes the product of the Hessian with a given vector. Suitable for large-scale problems. On indefinite problems it requires usually less iterations than the *trust-ncg* method and is recommended for medium and large-scale problems.

Method *trust-exact* is a trust-region method for unconstrained minimization in which quadratic subproblems are solved almost exactly [13]. This algorithm requires the gradient and the Hessian (which is *not* required to be positive definite). It is, in many situations, the Newton method to converge in fewer iteraction and the most recommended for small and medium-size problems.

Bound-Constrained minimization

Method *L-BFGS-B* uses the L-BFGS-B algorithm [6], [7] for bound constrained minimization.

Method *TNC* uses a truncated Newton algorithm [5], [8] to minimize a function with variables subject to bounds. This algorithm uses gradient information; it is also called Newton Conjugate-Gradient. It differs from the *Newton-CG* method described above as it wraps a C implementation and allows each variable to be given upper and lower bounds.

Constrained Minimization

Method *COBYLA* uses the Constrained Optimization BY Linear Approximation (COBYLA) method [9], [10], [11]. The algorithm is based on linear approximations to the objective function and each constraint. The method wraps a FORTRAN implementation of the algorithm. The constraints functions 'fun' may return either a single number or an array or list of numbers.

Method *SLSQP* uses Sequential Least SQuares Programming to minimize a function of several variables with any combination of bounds, equality and inequality constraints. The method wraps the SLSQP Optimization subroutine originally implemented by Dieter Kraft [12]. Note that the wrapper handles infinite values in bounds by converting them into large floating values.

Method *trust-constr* is a trust-region algorithm for constrained optimization. It swiches between two implementations depending on the problem definition. It is the most versatile constrained minimization algorithm implemented in SciPy and the most appropriate for large-scale problems. For equality constrained problems it is an implementation of Byrd-Omojokun Trust-Region SQP method described in [17] and in [5], p. 549. When inequality constraints are imposed as well, it swiches to the trust-region interior point method described in [16]. This interior point algorithm, in turn, solves inequality constraints by introducing slack variables and solving a sequence of equality-constrained barrier problems for progressively smaller values of the barrier parameter. The previously described equality constrained SQP method is used to solve the subproblems with increasing levels of accuracy as the iterate gets closer to a solution.

Finite-Difference Options

For Method *trust-constr* the gradient and the Hessian may be approximated using three finite-difference schemes: {'2-point', '3-point', 'cs'}. The scheme 'cs' is, potentially, the most accurate but it requires the function to correctly handles complex inputs and to be differentiable in the complex plane. The scheme '3-point' is more accurate than '2-point' but requires twice as much operations.

Custom minimizers

It may be useful to pass a custom minimization method, for example when using a frontend to this method such as `scipy.optimize.basinhopping` or a different library. You can simply pass a callable as the `method` parameter.

The callable is called as `method(fun, x0, args, **kwargs, **options)` where `kwargs` corresponds to any other parameters passed to `minimize` (such as *callback*, *hess*, etc.), except the *options* dict, which has its contents also passed as *method* parameters pair by pair. Also, if *jac* has been passed as a bool type, *jac* and *fun* are mangled so that *fun* returns just the function values and *jac* is converted to a function returning the Jacobian. The method shall return an `OptimizeResult` object.

The provided *method* callable must be able to accept (and possibly ignore) arbitrary parameters; the set of parameters accepted by `minimize` may expand in future versions and then these parameters will be passed to the method. You can find an example in the scipy.optimize tutorial.

New in version 0.11.0.

References

[1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]

Examples

Let us consider the problem of minimizing the Rosenbrock function. This function (and its respective derivatives) is implemented in `rosen` (resp. `rosen_der`, `rosen_hess`) in the `scipy.optimize`.

```
>>> from scipy.optimize import minimize, rosen, rosen_der
```

A simple application of the *Nelder-Mead* method is:

```
>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> res = minimize(rosen, x0, method='Nelder-Mead', tol=1e-6)
>>> res.x
array([ 1.,  1.,  1.,  1.,  1.])
```

Now using the *BFGS* algorithm, using the first derivative and a few options:

```
>>> res = minimize(rosen, x0, method='BFGS', jac=rosen_der,
...                options={'gtol': 1e-6, 'disp': True})
Optimization terminated successfully.
         Current function value: 0.000000
```

(continues on next page)

(continued from previous page)

```
         Iterations: 26
         Function evaluations: 31
         Gradient evaluations: 31
>>> res.x
array([ 1.,  1.,  1.,  1.,  1.])
>>> print(res.message)
Optimization terminated successfully.
>>> res.hess_inv
array([[ 0.00749589,  0.01255155,  0.02396251,  0.04750988,  0.09495377],␣
↪ # may vary
       [ 0.01255155,  0.02510441,  0.04794055,  0.09502834,  0.18996269],
       [ 0.02396251,  0.04794055,  0.09631614,  0.19092151,  0.38165151],
       [ 0.04750988,  0.09502834,  0.19092151,  0.38341252,  0.7664427 ],
       [ 0.09495377,  0.18996269,  0.38165151,  0.7664427,   1.53713523]])
```

Next, consider a minimization problem with several constraints (namely Example 16.4 from [5]). The objective function is:

```
>>> fun = lambda x: (x[0] - 1)**2 + (x[1] - 2.5)**2
```

There are three constraints defined as:

```
>>> cons = ({'type': 'ineq', 'fun': lambda x:  x[0] - 2 * x[1] + 2},
...         {'type': 'ineq', 'fun': lambda x: -x[0] - 2 * x[1] + 6},
...         {'type': 'ineq', 'fun': lambda x: -x[0] + 2 * x[1] + 2})
```

And variables must be positive, hence the following bounds:

```
>>> bnds = ((0, None), (0, None))
```

The optimization problem is solved using the SLSQP method as:

```
>>> res = minimize(fun, (2, 0), method='SLSQP', bounds=bnds,
...                constraints=cons)
```

It should converge to the theoretical solution (1.4 ,1.7).

The `minimize` function supports the following methods:

minimize(method='Nelder-Mead')

`scipy.optimize.minimize`(*fun*, *x0*, *args=()*, *method='Nelder-Mead'*, *tol=None*, *callback=None*, *options={'func': None, 'maxiter': None, 'maxfev': None, 'disp': False, 'return_all': False, 'initial_simplex': None, 'xatol': 0.0001, 'fatol': 0.0001, 'adaptive': False}*)

Minimization of scalar function of one or more variables using the Nelder-Mead algorithm.

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options

disp [bool] Set to True to print convergence messages.

maxiter, maxfev

[int] Maximum allowed number of iterations and function evaluations. Will default to `N*200`, where `N` is the number of variables, if neither *maxiter* or *maxfev* is set. If both *maxiter* and *maxfev* are set, minimization will stop at the first reached.

initial_simplex	[array_like of shape (N + 1, N)] Initial simplex. If given, overrides *x0*. `initial_simplex[j,:]` should contain the coordinates of the j-th vertex of the `N+1` vertices in the simplex, where `N` is the dimension.
xatol	[float, optional] Absolute error in xopt between iterations that is acceptable for convergence.
fatol	[number, optional] Absolute error in func(xopt) between iterations that is acceptable for convergence.
adaptive	[bool, optional] Adapt algorithm parameters to dimensionality of problem. Useful for high-dimensional minimization [1].

References

[1]

minimize(method='Powell')

`scipy.optimize.`**`minimize`**(*fun*, *x0*, *args=()*, *method='Powell'*, *tol=None*, *callback=None*, *options={'func': None, 'xtol': 0.0001, 'ftol': 0.0001, 'maxiter': None, 'maxfev': None, 'disp': False, 'direc': None, 'return_all': False}*)

Minimization of scalar function of one or more variables using the modified Powell algorithm.

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options

disp	[bool] Set to True to print convergence messages.
xtol	[float] Relative error in solution *xopt* acceptable for convergence.
ftol	[float] Relative error in `fun(xopt)` acceptable for convergence.
maxiter, maxfev	[int] Maximum allowed number of iterations and function evaluations. Will default to `N*1000`, where `N` is the number of variables, if neither *maxiter* or *maxfev* is set. If both *maxiter* and *maxfev* are set, minimization will stop at the first reached.
direc	[ndarray] Initial set of direction vectors for the Powell method.

minimize(method='CG')

`scipy.optimize.`**`minimize`**(*fun*, *x0*, *args=()*, *method='CG'*, *jac=None*, *tol=None*, *callback=None*, *options={'gtol': 1e-05, 'norm': inf, 'eps': 1.4901161193847656e-08, 'maxiter': None, 'disp': False, 'return_all': False}*)

Minimization of scalar function of one or more variables using the conjugate gradient algorithm.

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options

disp	[bool] Set to True to print convergence messages.
maxiter	[int] Maximum number of iterations to perform.
gtol	[float] Gradient norm must be less than *gtol* before successful termination.
norm	[float] Order of norm (Inf is max, -Inf is min).
eps	[float or ndarray] If *jac* is approximated, use this value for the step size.

minimize(method='BFGS')

`scipy.optimize.minimize`(*fun*, *x0*, *args=()*, *method='BFGS'*, *jac=None*, *tol=None*, *callback=None*, *options={'gtol': 1e-05, 'norm': inf, 'eps': 1.4901161193847656e-08, 'maxiter': None, 'disp': False, 'return_all': False}*)

Minimization of scalar function of one or more variables using the BFGS algorithm.

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options

disp	[bool] Set to True to print convergence messages.
maxiter	[int] Maximum number of iterations to perform.
gtol	[float] Gradient norm must be less than *gtol* before successful termination.
norm	[float] Order of norm (Inf is max, -Inf is min).
eps	[float or ndarray] If *jac* is approximated, use this value for the step size.

minimize(method='Newton-CG')

`scipy.optimize.minimize`(*fun*, *x0*, *args=()*, *method='Newton-CG'*, *jac=None*, *hess=None*, *hessp=None*, *tol=None*, *callback=None*, *options={'xtol': 1e-05, 'eps': 1.4901161193847656e-08, 'maxiter': None, 'disp': False, 'return_all': False}*)

Minimization of scalar function of one or more variables using the Newton-CG algorithm.

Note that the *jac* parameter (Jacobian) is required.

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options

disp	[bool] Set to True to print convergence messages.
xtol	[float] Average relative error in solution *xopt* acceptable for convergence.
maxiter	[int] Maximum number of iterations to perform.
eps	[float or ndarray] If *jac* is approximated, use this value for the step size.

minimize(method='L-BFGS-B')

`scipy.optimize.minimize`(*fun*, *x0*, *args=()*, *method='L-BFGS-B'*, *jac=None*, *bounds=None*, *tol=None*, *callback=None*, *options={'disp': None, 'maxcor': 10, 'ftol': 2.220446049250313e-09, 'gtol': 1e-05, 'eps': 1e-08, 'maxfun': 15000, 'maxiter': 15000, 'iprint': -1, 'maxls': 20}*)

Minimize a scalar function of one or more variables using the L-BFGS-B algorithm.

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options

disp	[None or int] If *disp is None* (the default), then the supplied version of *iprint* is used. If *disp is not None*, then it overrides the supplied version of *iprint* with the behaviour you outlined.
maxcor	[int] The maximum number of variable metric corrections used to define the limited memory matrix. (The limited memory BFGS method does not store the full hessian but uses this many terms in an approximation to it.)
ftol	[float] The iteration stops when `(f^k - f^{k+1})/max{|f^k|,|f^{k+1}|,1} <= ftol`.

gtol	[float] The iteration will stop when `max{|proj g_i	i = 1, ..., n} <= gtol` where `pg_i` is the i-th component of the projected gradient.
eps	[float] Step size used for numerical approximation of the jacobian.
maxfun	[int] Maximum number of function evaluations.
maxiter	[int] Maximum number of iterations.
maxls	[int, optional] Maximum number of line search steps (per iteration). Default is 20.

Notes

The option *ftol* is exposed via the `scipy.optimize.minimize` interface, but calling `scipy.optimize.fmin_l_bfgs_b` directly exposes *factr*. The relationship between the two is `ftol = factr * numpy.finfo(float).eps`. I.e., *factr* multiplies the default machine floating-point precision to arrive at *ftol*.

minimize(method='TNC')

`scipy.optimize.`**`minimize`**(*fun*, *x0*, *args=()*, *method='TNC'*, *jac=None*, *bounds=None*, *tol=None*, *callback=None*, *options={'eps': 1e-08*, *'scale': None*, *'offset': None*, *'mesg_num': None*, *'maxCGit': -1*, *'maxiter': None*, *'eta': -1*, *'stepmx': 0*, *'accuracy': 0*, *'minfev': 0*, *'ftol': -1*, *'xtol': -1*, *'gtol': -1*, *'rescale': -1*, *'disp': False}*)

Minimize a scalar function of one or more variables using a truncated Newton (TNC) algorithm.

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options

eps	[float] Step size used for numerical approximation of the jacobian.
scale	[list of floats] Scaling factors to apply to each variable. If None, the factors are up-low for interval bounded variables and 1+|x] fo the others. Defaults to None
offset	[float] Value to subtract from each variable. If None, the offsets are (up+low)/2 for interval bounded variables and x for the others.
disp	[bool] Set to True to print convergence messages.
maxCGit	[int] Maximum number of hessian*vector evaluations per main iteration. If maxCGit == 0, the direction chosen is -gradient if maxCGit < 0, maxCGit is set to max(1,min(50,n/2)). Defaults to -1.
maxiter	[int] Maximum number of function evaluation. if None, *maxiter* is set to max(100, 10*len(x0)). Defaults to None.
eta	[float] Severity of the line search. if < 0 or > 1, set to 0.25. Defaults to -1.
stepmx	[float] Maximum step for the line search. May be increased during call. If too small, it will be set to 10.0. Defaults to 0.
accuracy	[float] Relative precision for finite difference calculations. If <= machine_precision, set to sqrt(machine_precision). Defaults to 0.
minfev	[float] Minimum function value estimate. Defaults to 0.
ftol	[float] Precision goal for the value of f in the stopping criterion. If ftol < 0.0, ftol is set to 0.0 defaults to -1.
xtol	[float] Precision goal for the value of x in the stopping criterion (after applying x scaling factors). If xtol < 0.0, xtol is set to sqrt(machine_precision). Defaults to -1.
gtol	[float] Precision goal for the value of the projected gradient in the stopping criterion (after applying x scaling factors). If gtol < 0.0, gtol is set to 1e-2 * sqrt(accuracy). Setting it to 0.0 is not recommended. Defaults to -1.
rescale	[float] Scaling factor (in log10) used to trigger f value rescaling. If 0, rescale at each iteration. If a large value, never rescale. If < 0, rescale is set to 1.3.

minimize(method='COBYLA')

`scipy.optimize.minimize`(*fun*, *x0*, *args=()*, *method='COBYLA'*, *constraints=()*, *tol=None*, *callback=None*, *options={'rhobeg': 1.0*, *'maxiter': 1000*, *'disp': False*, *'catol': 0.0002}*)

Minimize a scalar function of one or more variables using the Constrained Optimization BY Linear Approximation (COBYLA) algorithm.

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options

rhobeg	[float] Reasonable initial changes to the variables.
tol	[float] Final accuracy in the optimization (not precisely guaranteed). This is a lower bound on the size of the trust region.
disp	[bool] Set to True to print convergence messages. If False, *verbosity* is ignored as set to 0.
maxiter	[int] Maximum number of function evaluations.
catol	[float] Tolerance (absolute) for constraint violations

minimize(method='SLSQP')

`scipy.optimize.minimize`(*fun*, *x0*, *args=()*, *method='SLSQP'*, *jac=None*, *bounds=None*, *constraints=()*, *tol=None*, *callback=None*, *options={'func': None*, *'maxiter': 100*, *'ftol': 1e-06*, *'iprint': 1*, *'disp': False*, *'eps': 1.4901161193847656e-08}*)

Minimize a scalar function of one or more variables using Sequential Least SQuares Programming (SLSQP).

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options

ftol	[float] Precision goal for the value of f in the stopping criterion.
eps	[float] Step size used for numerical approximation of the Jacobian.
disp	[bool] Set to True to print convergence messages. If False, *verbosity* is ignored and set to 0.
maxiter	[int] Maximum number of iterations.

minimize(method='trust-constr')

`scipy.optimize.minimize`(*fun*, *x0*, *args=()*, *method='trust-constr'*, *hess=None*, *hessp=None*, *bounds=None*, *constraints=()*, *tol=None*, *callback=None*, *options={'grad': None*, *'xtol': 1e-08*, *'gtol': 1e-08*, *'barrier_tol': 1e-08*, *'sparse_jacobian': None*, *'maxiter': 1000*, *'verbose': 0*, *'finite_diff_rel_step': None*, *'initial_constr_penalty': 1.0*, *'initial_tr_radius': 1.0*, *'initial_barrier_parameter': 0.1*, *'initial_barrier_tolerance': 0.1*, *'factorization_method': None*, *'disp': False}*)

Minimize a scalar function subject to constraints.

Parameters

gtol	[float, optional] Tolerance for termination by the norm of the Lagrangian gradient. The algorithm will terminate when both the infinity norm (i.e. max abs value) of the Lagrangian gradient and the constraint violation are smaller than `gtol`. Default is 1e-8.
xtol	[float, optional] Tolerance for termination by the change of the independent variable. The algorithm will terminate when `tr_radius < xtol`, where `tr_radius` is the radius of the trust region used in the algorithm. Default is 1e-8.

barrier_tol
[float, optional] Threshold on the barrier parameter for the algorithm termination. When inequality constraints are present the algorithm will terminate only when the barrier parameter is less than *barrier_tol*. Default is 1e-8.

sparse_jacobian
[{bool, None}, optional] Determines how to represent Jacobians of the constraints. If bool, then Jacobians of all the constraints will be converted to the corresponding format. If None (default), then Jacobians won't be converted, but the algorithm can proceed only if they all have the same format.

initial_tr_radius: float, optional
Initial trust radius. The trust radius gives the maximum distance between solution points in consecutive iterations. It reflects the trust the algorithm puts in the local approximation of the optimization problem. For an accurate local approximation the trust-region should be large and for an approximation valid only close to the current point it should be a small one. The trust radius is automatically updated throughout the optimization process, with `initial_tr_radius` being its initial value. Default is 1 (recommended in [1], p. 19).

initial_constr_penalty
[float, optional] Initial constraints penalty parameter. The penalty parameter is used for balancing the requirements of decreasing the objective function and satisfying the constraints. It is used for defining the merit function: `merit_function(x) = fun(x) + constr_penalty * constr_norm_l2(x)`, where `constr_norm_l2(x)` is the l2 norm of a vector containing all the constraints. The merit function is used for accepting or rejecting trial points and `constr_penalty` weights the two conflicting goals of reducing objective function and constraints. The penalty is automatically updated throughout the optimization process, with `initial_constr_penalty` being its initial value. Default is 1 (recommended in [1], p 19).

initial_barrier_parameter, initial_barrier_tolerance: float, optional
Initial barrier parameter and initial tolerance for the barrier subproblem. Both are used only when inequality constraints are present. For dealing with optimization problems `min_x f(x)` subject to inequality constraints `c(x) <= 0` the algorithm introduces slack variables, solving the problem `min_(x,s) f(x) + barrier_parameter*sum(ln(s))` subject to the equality constraints `c(x) + s = 0` instead of the original problem. This subproblem is solved for increasing values of `barrier_parameter` and with decreasing tolerances for the termination, starting with `initial_barrier_parameter` for the barrier parameter and `initial_barrier_tolerance` for the barrier subproblem barrier. Default is 0.1 for both values (recommended in [1] p. 19).

factorization_method
[string or None, optional] Method to factorize the Jacobian of the constraints. Use None (default) for the auto selection or one of:

- 'NormalEquation' (requires scikit-sparse)
- 'AugmentedSystem'
- 'QRFactorization'
- 'SVDFactorization'

The methods 'NormalEquation' and 'AugmentedSystem' can be used only with sparse constraints. The projections required by the algorithm will be computed using, respectively, the the normal equation and the augmented system approaches explained in [1]. 'NormalEquation' computes the Cholesky factorization of `A A.T` and 'AugmentedSystem' performs the LU factorization of an augmented system. They usually provide similar results. 'AugmentedSystem' is used by default for sparse matrices.

The methods 'QRFactorization' and 'SVDFactorization' can be used only with dense constraints. They compute the required projections using, respectively, QR and SVD factorizations. The 'SVDFactorization' method can cope with Jacobian matrices with deficient row rank and will be used whenever other factorization methods fail (which may imply the conversion of sparse matrices to a dense format when required). By default 'QRFactorization' is used for dense matrices.

finite_diff_rel_step
[None or array_like, optional] Relative step size for the finite difference approximation.

maxiter [int, optional] Maximum number of algorithm iterations. Default is 1000.

verbose [{0, 1, 2}, optional] Level of algorithm's verbosity:

- 0 (default) : work silently.
- 1 : display a termination report.
- 2 : display progress during iterations.
- 3 : display progress during iterations (more complete report).

disp [bool, optional] If True (default) then *verbose* will be set to 1 if it was 0.

Returns

'OptimizeResult' with the fields documented below. Note the following:

1. All values corresponding to the constraints are ordered as they were passed to the solver. And values corresponding to *bounds* constraints are put *after* other constraints.
2. All numbers of function, Jacobian or Hessian evaluations correspond to numbers of actual Python function calls. It means, for example, that if a Jacobian is estimated by finite differences then the number of Jacobian evaluations will be zero and the number of function evaluations will be incremented by all calls during the finite difference estimation.

x [ndarray, shape (n,)] Solution found.

optimality [float] Infinity norm of the Lagrangian gradient at the solution.

constr_violation
[float] Maximum constraint violation at the solution.

fun [float] Objective function at the solution.

grad [ndarray, shape (n,)] Gradient of the objective function at the solution.

lagrangian_grad
[ndarray, shape (n,)] Gradient of the Lagrangian function at the solution.

nit [int] Total number of iterations.

nfev [integer] Number of the objective function evaluations.

ngev [integer] Number of the objective function gradient evaluations.

nhev [integer] Number of the objective function Hessian evaluations.

cg_niter [int] Total number of the conjugate gradient method iterations.

method [{'equality_constrained_sqp', 'tr_interior_point'}] Optimization method used.

constr [list of ndarray] List of constraint values at the solution.

jac [list of {ndarray, sparse matrix}] List of the Jacobian matrices of the constraints at the solution.

v [list of ndarray] List of the Lagrange multipliers for the constraints at the solution. For an inequality constraint a positive multiplier means that the upper bound is active, a negative multiplier means that the lower bound is active and if a multiplier is zero it means the constraint is not active.

constr_nfev
[list of int] Number of constraint evaluations for each of the constraints.

constr_njev
[list of int] Number of Jacobian matrix evaluations for each of the constraints.

constr_nhev
[list of int] Number of Hessian evaluations for each of the constraints.

tr_radius [float] Radius of the trust region at the last iteration.

constr_penalty
[float] Penalty parameter at the last iteration, see *initial_constr_penalty*.

barrier_tolerance
[float] Tolerance for the barrier subproblem at the last iteration. Only for problems with inequality constraints.

barrier_parameter
[float] Barrier parameter at the last iteration. Only for problems with inequality constraints.

execution_time
[float] Total execution time.

message [str] Termination message.

status [{0, 1, 2, 3}] Termination status:

- 0 : The maximum number of function evaluations is exceeded.
- 1 : *gtol* termination condition is satisfied.
- 2 : *xtol* termination condition is satisfied.
- 3 : *callback* function requested termination.

cg_stop_cond
[int] Reason for CG subproblem termination at the last iteration:

- 0 : CG subproblem not evaluated.
- 1 : Iteration limit was reached.
- 2 : Reached the trust-region boundary.
- 3 : Negative curvature detected.
- 4 : Tolerance was satisfied.

References

[1]

minimize(method='dogleg')

`scipy.optimize.minimize`(*fun*, *x0*, *args=()*, *method='dogleg'*, *jac=None*, *hess=None*, *tol=None*, *callback=None*, *options={}*)

Minimization of scalar function of one or more variables using the dog-leg trust-region algorithm.

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options

initial_trust_radius
[float] Initial trust-region radius.

max_trust_radius
[float] Maximum value of the trust-region radius. No steps that are longer than this value will be proposed.

eta [float] Trust region related acceptance stringency for proposed steps.

gtol [float] Gradient norm must be less than *gtol* before successful termination.

minimize(method='trust-ncg')

`scipy.optimize.minimize`(*fun*, *x0*, *args=()*, *method='trust-ncg'*, *jac=None*, *hess=None*, *hessp=None*, *tol=None*, *callback=None*, *options={}*)

Minimization of scalar function of one or more variables using the Newton conjugate gradient trust-region algorithm.

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options

initial_trust_radius
[float] Initial trust-region radius.

max_trust_radius
[float] Maximum value of the trust-region radius. No steps that are longer than this value will be proposed.

eta [float] Trust region related acceptance stringency for proposed steps.

gtol [float] Gradient norm must be less than *gtol* before successful termination.

minimize(method='trust-krylov')

`scipy.optimize.minimize` (*fun*, *x0*, *args=()*, *method='trust-krylov'*, *jac=None*, *hess=None*, *hessp=None*, *tol=None*, *callback=None*, *options={'inexact': True}*)

Minimization of a scalar function of one or more variables using a nearly exact trust-region algorithm that only requires matrix vector products with the hessian matrix.

New in version 1.0.0.

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options

inexact [bool, optional] Accuracy to solve subproblems. If True requires less nonlinear iterations, but more vector products.

minimize(method='trust-exact')

`scipy.optimize.minimize` (*fun*, *x0*, *args=()*, *method='trust-exact'*, *jac=None*, *hess=None*, *tol=None*, *callback=None*, *options={}*)

Minimization of scalar function of one or more variables using a nearly exact trust-region algorithm.

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options

initial_tr_radius
[float] Initial trust-region radius.

max_tr_radius
[float] Maximum value of the trust-region radius. No steps that are longer than this value will be proposed.

eta [float] Trust region related acceptance stringency for proposed steps.

gtol [float] Gradient norm must be less than `gtol` before successful termination.

Constraints are passed to `minimize` function as a single object or as a list of objects from the following classes:

`NonlinearConstraint`(fun, lb, ub[, jac, ...])	Nonlinear constraint on the variables.
`LinearConstraint`(A, lb, ub[, keep_feasible])	Linear constraint on the variables.

scipy.optimize.NonlinearConstraint

class `scipy.optimize.NonlinearConstraint` (*fun*, *lb*, *ub*, *jac='2-point'*, *hess=<scipy.optimize._hessian_update_strategy.BFGS object>*, *keep_feasible=False*, *finite_diff_rel_step=None*, *finite_diff_jac_sparsity=None*)

Nonlinear constraint on the variables.