COBYQA Manual

Release 1.0.0

Tom M. Ragonneau and Zaikun Zhang

January 10, 2024

CONTENTS

CHAPTER

ONE

USER GUIDE

This guide explains how to *install* and *use* COBYQA. It also briefly introduce the *framework* of the method. For more details on the Python API of COBYQA, see the *API reference*.

1.1 Installation

1.1.1 Recommended installation

We highly recommend installing COBYQA via PyPI. This does not need you to download the source code. Install pip in your system, then execute

```
pip install cobyqa
```

in a command shell (e.g., the terminal in Linux or Mac, or the Command Shell for Windows). If your pip launcher is not `pip`, adapt the command (it may be `pip3` for example). If this command runs successfully, COBYQA is installed. You may verify whether COBYQA is successfully installed by executing

```
python -c "import cobyqa; cobyqa.show_versions()"
```

If your Python launcher is not `python`, adapt the command (it may be `python3` for example).

1.1.2 Alternative installation (using source distribution)

Alternatively, although discouraged, COBYQA can be installed from the source code. Download and decompress the source code package. You will obtain a folder containing `pyproject.toml`. In a command shell, change your directory to this folder, and then run

```
pip install .
```

1.2 Usage

We provide below basic usage information on how to use COBYQA. For more details on the signature of the `minimize` function, please refer to the *API documentation*.

1.2.1 How to use COBYQA

COBYQA provides a *minimize* function. This is the entry point to the solver. It solves unconstrained, bound-constrained, linearly constrained, and nonlinearly constrained optimization problems.

We provide below simple examples on how to use COBYQA.

1.2.2 Examples

Example of unconstrained optimization

Let us first minimize the Rosenbrock function implemented in `scipy.optimize`, defined as

$$f(x) = \sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2$$

for $x \in \mathbb{R}^n$. To solve the problem using COBYQA, run:

```
from cobyqa import minimize
from scipy.optimize import rosen

x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
res = minimize(rosen, x0)
print(res.x)
```

This should display the desired output `[1. 1. 1. 1. 1.]`.

Example of linearly constrained optimization

To see how bound and linear constraints are handled using *minimize*, let us solve Example 16.4 of [UU1], defined as

$$\begin{aligned}
\min_{x \in \mathbb{R}^2} \quad & (x_1 - 1)^2 + (x_2 - 2.5)^2\\
\text{s.t.} \quad & -x_1 + 2x_2 \le 2,\\
& x_1 + 2x_2 \le 6,\\
& x_1 - 2x_2 \le 2,\\
& x_1 \ge 0,\\
& x_2 \ge 0.
\end{aligned}$$

To solve the problem using COBYQA, run:

```
import numpy as np
from cobyqa import minimize
from scipy.optimize import Bounds, LinearConstraint

def fun(x):
    return (x[0] - 1.0) ** 2.0 + (x[1] - 2.5) ** 2.0

x0 = [2.0, 0.0]
bounds = Bounds([0.0, 0.0], np.inf)
constraints = LinearConstraint([[-1.0, 2.0], [1.0, 2.0], [1.0, -2.0]], -np.inf, [2.0,␣
↪6.0, 2.0])
res = minimize(fun, x0, bounds=bounds, constraints=constraints)
print(res.x)
```

This should display the desired output `[1.4 1.7]`.

Example of nonlinearly constrained optimization

To see how nonlinear constraints are handled, we solve Problem (F) of [UU2], defined as

$$\begin{aligned} \min_{x \in \mathbb{R}^2} \quad & -x_1 - x_2 \\ \text{s.t.} \quad & x_1^2 - x_2 \le 0, \\ & x_1^2 + x_2^2 \le 1. \end{aligned}$$

To solve the problem using COBYQA, run:

```
import numpy as np
from cobyqa import minimize
from scipy.optimize import NonlinearConstraint

def fun(x):
    return -x[0] - x[1]

x0 = [1.0, 1.0]
constraints = NonlinearConstraint(lambda x: [
    x[0] ** 2.0 - x[1],
    x[0] ** 2.0 + x[1] ** 2.0,
], -np.inf, [0.0, 1.0])
res = minimize(fun, x0, constraints=constraints)
print(res.x)
```

This should display the desired output `[0.7071 0.7071]`.

1.3 Framework

Work in progress. In the meantime, see Chapter 5 of [UF1].

1.4 Subproblem solvers

Work in progress. In the meantime, see Chapter 6 of [US1].

1.5 COBYQA license

```
BSD 3-Clause License

Copyright (c) 2021-2024, Tom M. Ragonneau and Zaikun Zhang

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
```

(continues on next page)

(continued from previous page)

CHAPTER

TWO

API DOCUMENTATION

Release
: 1.0.0

Date
: January 10, 2024

This section references a manual for using COBYQA in Python. Most users will only need to use the *minimize* function. An *installation* guide and *usage* examples are provided in the *user guide*.

2.1 Optimization solver

This module is the entry point of the COBYQA solver. Most users will only need to use the *minimize* function.

minimize(fun, x0[, args, bounds, ...])	Minimize a scalar function using the COBYQA method.
show_versions()	Display useful system and dependencies information.

2.1.1 cobyqa.minimize

`cobyqa.minimize`(*fun*, *x0*, *args=()*, *bounds=None*, *constraints=()*, *callback=None*, *options=None*)

Minimize a scalar function using the COBYQA method.

The COBYQA method is a derivative-free optimization method designed to solve general nonlinear optimization problems. A complete description of the method is given in [3].

Parameters

fun
: [{callable, None}] Objective function to be minimized.

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

where `x` is an array with shape (n,) and *args* is a tuple. If None, the objective function is assumed to be the zero function.

x0
: [array_like, shape (n,)] Initial guess.

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

bounds
: [{`scipy.optimize.Bounds`, array_like, shape (n, 2)}, optional] Bound constraints of the problem. It can be one of the cases below.

1. An instance of `scipy.optimize.Bounds`. For the time being, the argument `keep_feasible` is disregarded.

2. An array with shape (n, 2). The bound constraints for `x[i]` are `bounds[i][0] <= x[i] <= bounds[i][1]`. Set `bounds[i][0]` to $-\infty$ if there is no lower bound, and set `bounds[i][1]` to ∞ if there is no upper bound.

constraints
[{`scipy.optimize.LinearConstraint`, `scipy.optimize.NonlinearConstraint`, dict, list}, optional] General constraints of the problem. It can be one of the cases below.

1. An instance of `scipy.optimize.LinearConstraint`. The argument `keep_feasible` is disregarded.
2. An instance of `scipy.optimize.NonlinearConstraint`. The arguments `jac`, `hess`, `keep_feasible`, `finite_diff_rel_step`, and `finite_diff_jac_sparsity` are disregarded.
3. A dictionary with fields:

   type
   [{'eq', 'ineq'}] Whether the constraint is an equality `fun(x, *args) = 0` or an inequality `fun(x, *args) >= 0`.

   fun
   [callable] Constraint function.

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

4. A list, each of whose elements are described in the cases above.

callback
[callable, optional] A callback executed at each objective function evaluation. The method terminates if a `StopIteration` exception is raised by the callback function. Its signature can be one of the following:

```
callback(intermediate_result)
```

where `intermediate_result` is a keyword parameter that contains an instance of `scipy.optimize.OptimizeResult`, with attributes `x` and `fun`, being the point at which the objective function is evaluated and the value of the objective function, respectively. The name of the parameter must be `intermediate_result` for the callback to be passed an instance of `scipy.optimize.OptimizeResult`.

Alternatively, the callback function can have the signature:

```
callback(xk)
```

where `xk` is the point at which the objective function is evaluated. Introspection is used to determine which of the signatures to invoke.

options
[dict, optional] Options passed to the solver. Accepted keys are:

disp
[bool, optional] Whether to print information about the optimization procedure.

maxfev
[int, optional] Maximum number of function evaluations.

maxiter
[int, optional] Maximum number of iterations.

target
[float, optional] Target on the objective function value. The optimization procedure is terminated when the objective function value of a nearly feasible point is less than or equal to this target.

feasibility_tol
[float, optional] Tolerance on the constraint violation.

radius_init
[float, optional] Initial trust-region radius. Typically, this value should be in the order of one tenth of the greatest expected change to the variables.

radius_final
[float, optional] Final trust-region radius. It should indicate the accuracy required in the final values of the variables.

nb_points
[int, optional] Number of interpolation points used to build the quadratic models of the objective and constraint functions.

scale
[bool, optional] Whether to scale the variables according to the bounds.

filter_size
[int, optional] Maximum number of points in the filter. The filter is used to select the best point returned by the optimization procedure.

store_history
[bool, optional] Whether to store the history of the function evaluations.

history_size
[int, optional] Maximum number of function evaluations to store in the history.

debug
[bool, optional] Whether to perform additional checks. This option should be used only for debugging purposes and is highly discouraged to general users.

Returns

`scipy.optimize.OptimizeResult`
Result of the optimization procedure, with the following fields:

message
[str] Description of the cause of the termination.

success
[bool] Whether the optimization procedure terminated successfully.

status
[int] Termination status of the optimization procedure.

x
[`numpy.ndarray`, shape (n,)] Solution point.

fun
[float] Objective function value at the solution point.

maxcv
[float] Maximum constraint violation at the solution point.

nfev
[int] Number of function evaluations.

nit
[int] Number of iterations.

If `store_history` is True, the result also has the following fields:

fun_history
[`numpy.ndarray`, shape (nfev,)] History of the objective function values.

maxcv_history
[numpy.ndarray, shape (nfev,)] History of the maximum constraint violations.

A description of the termination statuses is given below.

Exit status	Description
0	The lower bound for the trust-region radius has been reached.
1	The target objective function value has been reached.
2	All variables are fixed by the bound constraints.
3	The callback requested to stop the optimization procedure.
4	The feasibility problem received has been solved successfully.
5	The maximum number of function evaluations has been exceeded.
6	The maximum number of iterations has been exceeded.
-1	The bound constraints are infeasible.
-2	A linear algebra error occurred.

References

[1], [2], [3]

Examples

To demonstrate how to use *minimize*, we first minimize the Rosenbrock function implemented in scipy.optimize in an unconstrained setting.

```
>>> from cobyqa import minimize
>>> from scipy.optimize import rosen
```

To solve the problem using COBYQA, run:

```
>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> res = minimize(rosen, x0)
>>> res.x
array([1., 1., 1., 1., 1.])
```

To see how bound and constraints are handled using *minimize*, we solve Example 16.4 of [1], defined as

$$\begin{aligned} \min_{x \in \mathbb{R}^2} \quad & (x_1 - 1)^2 + (x_2 - 2.5)^2 \\ \text{s.t.} \quad & -x_1 + 2x_2 \le 2, \\ & x_1 + 2x_2 \le 6, \\ & x_1 - 2x_2 \le 2, \\ & x_1 \ge 0, \\ & x_2 \ge 0. \end{aligned}$$

```
>>> import numpy as np
>>> from scipy.optimize import Bounds, LinearConstraint
```

Its objective function can be implemented as:

```
>>> def fun(x):
...     return (x[0] - 1.0) ** 2.0 + (x[1] - 2.5) ** 2.0
```

This problem can be solved using *minimize* as:

```
>>> x0 = [2.0, 0.0]
>>> bounds = Bounds([0.0, 0.0], np.inf)
>>> constraints = LinearConstraint([[-1.0, 2.0], [1.0, 2.0], [1.0, -2.0]], -np.
↪inf, [2.0, 6.0, 2.0])
>>> res = minimize(fun, x0, bounds=bounds, constraints=constraints)
>>> res.x
array([1.4, 1.7])
```

Finally, to see how nonlinear constraints are handled, we solve Problem (F) of [2], defined as

$$\begin{aligned}\min_{x \in \mathbb{R}^2} \quad & -x_1 - x_2\\ \text{s.t.} \quad & x_1^2 - x_2 \leq 0,\\ & x_1^2 + x_2^2 \leq 1.\end{aligned}$$

```
>>> from scipy.optimize import NonlinearConstraint
```

Its objective and constraint functions can be implemented as:

```
>>> def fun(x):
...     return -x[0] - x[1]
>>>
>>> def cub(x):
...     return [x[0] ** 2.0 - x[1], x[0] ** 2.0 + x[1] ** 2.0]
```

This problem can be solved using *minimize* as:

```
>>> x0 = [1.0, 1.0]
>>> constraints = NonlinearConstraint(cub, -np.inf, [0.0, 1.0])
>>> res = minimize(fun, x0, constraints=constraints)
>>> res.x
array([0.707, 0.707])
```

2.1.2 cobyqa.show_versions

`cobyqa.`**`show_versions`**`()`

Display useful system and dependencies information.

When reporting issues, please include this information.

The implementation of COBYQA is documented in the *developer guide*. It is completely unnecessary to read the *developer guide* to use COBYQA, and it only provides insights for people interested in the development of derivative-free optimization methods such as COBYQA.

CHAPTER

THREE

DEVELOPER GUIDE

This guide does not cover the usage of COBYQA. If you want to use COBYQA in your project, please refer to the *API documentation*. This guide is intended for developers who want to contribute to the COBYQA solver and to derivative-free optimization solvers in general.

The `cobyqa` module has four submodules, detailed below. Users do not need to import these submodules when using COBYQA.

The `problem` module implements classes for representing optimization problems.

3.1 Optimization problem

This module implements classes for representing optimization problems for COBYQA.

`ObjectiveFunction`(fun, verbose, debug, *args)	Real-valued objective function.
`BoundConstraints`(bounds)	Bound constraints `xl <= x <= xu`.
`LinearConstraints`(constraints, n, debug)	Linear constraints `a_ub @ x <= b_ub` and `a_eq @ x == b_eq`.
`NonlinearConstraints`(constraints, verbose, debug)	Nonlinear constraints `c_ub(x) <= 0` and `c_eq(x) == b_eq`.
`Problem`(obj, x0, bounds, linear, nonlinear, ...)	Optimization problem.

3.1.1 cobyqa.problem.ObjectiveFunction

class cobyqa.problem.**ObjectiveFunction**(*fun*, *verbose*, *debug*, **args*)

Real-valued objective function.

Attributes

n_eval
Number of function evaluations.

name
Name of the objective function.

Methods

`__call__`(x)	Evaluate the objective function.

cobyqa.problem.ObjectiveFunction.__call__

`ObjectiveFunction.__call__`(*x*)

Evaluate the objective function.

Parameters

x
[array_like, shape (n,)] Point at which the objective function is evaluated.

Returns

float
Function value at *x*.

3.1.2 cobyqa.problem.BoundConstraints

`class cobyqa.problem.BoundConstraints`(*bounds*)

Bound constraints `xl <= x <= xu`.

Attributes

`is_feasible`
Whether the bound constraints are feasible.

`m`
Number of bound constraints.

`xl`
Lower bound.

`xu`
Upper bound.

Methods

`maxcv`(x)	Evaluate the maximum constraint violation.
`project`(x)	Project a point onto the feasible set.

cobyqa.problem.BoundConstraints.maxcv

`BoundConstraints.maxcv`(*x*)

Evaluate the maximum constraint violation.

Parameters

x
[array_like, shape (n,)] Point at which the maximum constraint violation is evaluated.

Returns

float
Maximum constraint violation at *x*.

cobyqa.problem.BoundConstraints.project

`BoundConstraints.project(x)`

Project a point onto the feasible set.

Parameters

x
[array_like, shape (n,)] Point to be projected.

Returns

numpy.ndarray, shape (n,)
Projection of *x* onto the feasible set.

3.1.3 cobyqa.problem.LinearConstraints

class `cobyqa.problem.LinearConstraints(constraints, n, debug)`

Linear constraints `a_ub @ x <= b_ub` and `a_eq @ x == b_eq`.

Attributes

a_eq
Left-hand side matrix of the linear equality constraints.

a_ub
Left-hand side matrix of the linear inequality constraints.

b_eq
Right-hand side vector of the linear equality constraints.

b_ub
Right-hand side vector of the linear inequality constraints.

m_eq
Number of linear equality constraints.

m_ub
Number of linear inequality constraints.

Methods

maxcv(x)	Evaluate the maximum constraint violation.

cobyqa.problem.LinearConstraints.maxcv

`LinearConstraints.maxcv(x)`

Evaluate the maximum constraint violation.

Parameters

x
[array_like, shape (n,)] Point at which the maximum constraint violation is evaluated.

Returns

float
Maximum constraint violation at *x*.

3.1.4 cobyqa.problem.NonlinearConstraints

`class cobyqa.problem.NonlinearConstraints(`*constraints*, *verbose*, *debug*`)`

Nonlinear constraints `c_ub(x) <= 0` and `c_eq(x) == b_eq`.

Attributes

m_eq
Number of nonlinear equality constraints.

m_ub
Number of nonlinear inequality constraints.

n_eval
Number of function evaluations.

Methods

__call__(x)	Evaluate the constraints.
maxcv(x[, cub_val, ceq_val])	Evaluate the maximum constraint violation.

cobyqa.problem.NonlinearConstraints.__call__

`NonlinearConstraints.__call__(`*x*`)`

Evaluate the constraints.

Parameters

x
[array_like, shape (n,)] Point at which the constraints are evaluated.

Returns

numpy.ndarray, shape (m_nonlinear_ub,)
Nonlinear inequality constraint function values.

numpy.ndarray, shape (m_nonlinear_eq,)
Nonlinear equality constraint function values.

cobyqa.problem.NonlinearConstraints.maxcv

`NonlinearConstraints.maxcv(`*x*, *cub_val=None*, *ceq_val=None*`)`

Evaluate the maximum constraint violation.

Parameters

x
[array_like, shape (n,)] Point at which the maximum constraint violation is evaluated.

cub_val
[array_like, shape (m_nonlinear_ub,), optional] Values of the nonlinear inequality constraints. If not provided, the nonlinear inequality constraints are evaluated at *x*.

ceq_val
[array_like, shape (m_nonlinear_eq,), optional] Values of the nonlinear equality constraints. If not provided, the nonlinear equality constraints are evaluated at *x*.

Returns

float
Maximum constraint violation at *x*.

3.1.5 cobyqa.problem.Problem

class cobyqa.problem.**Problem**(*obj*, *x0*, *bounds*, *linear*, *nonlinear*, *callback*, *feasibility_tol*, *scale*, *store_history*, *history_size*, *filter_size*, *debug*)

Optimization problem.

Attributes

bounds
Bound constraints.

fun_history
History of objective function evaluations.

fun_name
Name of the objective function.

is_feasibility
Whether the problem is a feasibility problem.

linear
Linear constraints.

m_bounds
Number of bound constraints.

m_linear_eq
Number of linear equality constraints.

m_linear_ub
Number of linear inequality constraints.

m_nonlinear_eq
Number of nonlinear equality constraints.

m_nonlinear_ub
Number of nonlinear inequality constraints.

maxcv_history
History of maximum constraint violations.

n
Number of variables.

n_eval
Number of function evaluations.

n_orig
Number of variables in the original problem (with fixed variables).

type
Type of the problem.

x0
Initial guess.

Methods

`__call__`(x)	Evaluate the objective and nonlinear constraint functions.
`best_eval`(penalty)	Return the best point in the filter and the corresponding objective and nonlinear constraint function evaluations.
`build_x`(x)	Build the full vector of variables from the reduced vector.
`maxcv`(x[, cub_val, ceq_val])	Evaluate the maximum constraint violation.

cobyqa.problem.Problem.__call__

`Problem.__call__(x)`

Evaluate the objective and nonlinear constraint functions.

Parameters

x
[array_like, shape (n,)] Point at which the functions are evaluated.

Returns

float
Objective function value.

`numpy.ndarray`, shape (m_nonlinear_ub,)
Nonlinear inequality constraint function values.

`numpy.ndarray`, shape (m_nonlinear_eq,)
Nonlinear equality constraint function values.

cobyqa.problem.Problem.best_eval

`Problem.best_eval(penalty)`

Return the best point in the filter and the corresponding objective and nonlinear constraint function evaluations.

Parameters

penalty
[float] Penalty parameter

Returns

`numpy.ndarray`, shape (n,)
Best point.

float
Corresponding objective function value.

float
Corresponding maximum constraint violation.

cobyqa.problem.Problem.build_x

`Problem.build_x(x)`

Build the full vector of variables from the reduced vector.

Parameters

x
[array_like, shape (n,)] Reduced vector of variables.

Returns

numpy.ndarray, shape (n_orig,)
Full vector of variables.

cobyqa.problem.Problem.maxcv

`Problem.maxcv(x, cub_val=None, ceq_val=None)`

Evaluate the maximum constraint violation.

Parameters

x
[array_like, shape (n,)] Point at which the maximum constraint violation is evaluated.

cub_val
[array_like, shape (m_nonlinear_ub,), optional] Values of the nonlinear inequality constraints. If not provided, the nonlinear inequality constraints are evaluated at *x*.

ceq_val
[array_like, shape (m_nonlinear_eq,), optional] Values of the nonlinear equality constraints. If not provided, the nonlinear equality constraints are evaluated at *x*.

Returns

float
Maximum constraint violation at *x*.

The `models` module implements the models used by COBYQA.

3.2 Quadratic models

This module implements classes for representing the quadratic models used by COBYQA.

`Interpolation`(pb, options)	Interpolation set.
`Quadratic`(interpolation, values, debug)	Quadratic model.
`Models`(pb, options)	Models for a nonlinear optimization problem.

3.2.1 cobyqa.models.Interpolation

`class cobyqa.models.Interpolation(pb, options)`

Interpolation set.

This class stores a base point around which the models are expanded and the interpolation points. The coordinates of the interpolation points are relative to the base point.

Attributes

n
Number of variables.

npt
Number of interpolation points.

x_base
Base point around which the models are expanded.

xpt
Interpolation points.

Methods

point(k)	Get the k-th interpolation point.

cobyqa.models.Interpolation.point

`Interpolation.point(k)`

Get the k-th interpolation point.

The return point is relative to the origin.

Parameters

k
[int] Index of the interpolation point.

Returns

numpy.ndarray, shape (n,)
k-th interpolation point.

3.2.2 cobyqa.models.Quadratic

`class cobyqa.models.Quadratic(interpolation, values, debug)`

Quadratic model.

This class stores the Hessian matrix of the quadratic model using the implicit/explicit representation designed by Powell for NEWUOA [1].

References

[1]

Attributes

n
Number of variables.

npt
Number of interpolation points used to define the quadratic model.

Methods

`__call__`(x, interpolation)	Evaluate the quadratic model at a given point.
`build_system`(xpt)	Build the left-hand side matrix of the interpolation system.
`curv`(v, interpolation)	Evaluate the curvature of the quadratic model along a given direction.
`grad`(x, interpolation)	Evaluate the gradient of the quadratic model at a given point.
`hess`(interpolation)	Evaluate the Hessian matrix of the quadratic model.
`hess_prod`(v, interpolation)	Evaluate the right product of the Hessian matrix of the quadratic model with a given vector.
`shift_x_base`(interpolation, new_x_base)	Shift the point around which the quadratic model is defined.
`solve_system`(interpolation, rhs)	Solve the interpolation system.
`update`(interpolation, k_new, dir_old, ...)	Update the quadratic model.

cobyqa.models.Quadratic.__call__

`Quadratic.`**`__call__`**(*x*, *interpolation*)

Evaluate the quadratic model at a given point.

Parameters

x
[`numpy.ndarray`, shape (n,)] Point at which the quadratic model is evaluated.

interpolation
[`cobyqa.models.Interpolation`] Interpolation set.

Returns

float
Value of the quadratic model at *x*.

cobyqa.models.Quadratic.build_system

`static Quadratic.`**`build_system`**(*xpt*)

Build the left-hand side matrix of the interpolation system.

Parameters

xpt
[`numpy.ndarray`, shape (n, npt)] Interpolation points.

Returns

numpy.ndarray, shape (npt + n + 1, npt + n + 1)
Left-hand side matrix of the interpolation system.

cobyqa.models.Quadratic.curv

Quadratic.**curv**(*v*, *interpolation*)

Evaluate the curvature of the quadratic model along a given direction.

Parameters

v
[numpy.ndarray, shape (n,)] Direction along which the curvature of the quadratic model is evaluated.

interpolation
[*cobyqa.models.Interpolation*] Interpolation set.

Returns

float
Curvature of the quadratic model along *v*.

cobyqa.models.Quadratic.grad

Quadratic.**grad**(*x*, *interpolation*)

Evaluate the gradient of the quadratic model at a given point.

Parameters

x
[numpy.ndarray, shape (n,)] Point at which the gradient of the quadratic model is evaluated.

interpolation
[*cobyqa.models.Interpolation*] Interpolation set.

Returns

numpy.ndarray, shape (n,)
Gradient of the quadratic model at *x*.

cobyqa.models.Quadratic.hess

Quadratic.**hess**(*interpolation*)

Evaluate the Hessian matrix of the quadratic model.

Parameters

interpolation
[*cobyqa.models.Interpolation*] Interpolation set.

Returns

numpy.ndarray, shape (n, n)
Hessian matrix of the quadratic model.

cobyqa.models.Quadratic.hess_prod

Quadratic.**hess_prod**(*v*, *interpolation*)

Evaluate the right product of the Hessian matrix of the quadratic model with a given vector.

Parameters

v
[numpy.ndarray, shape (n,)] Vector with which the Hessian matrix of the quadratic model is multiplied from the right.

interpolation
[*cobyqa.models.Interpolation*] Interpolation set.

Returns

numpy.ndarray, shape (n,)
Right product of the Hessian matrix of the quadratic model with *v*.

cobyqa.models.Quadratic.shift_x_base

Quadratic.**shift_x_base**(*interpolation*, *new_x_base*)

Shift the point around which the quadratic model is defined.

Parameters

interpolation
[*cobyqa.models.Interpolation*] Previous interpolation set.

new_x_base
[numpy.ndarray, shape (n,)] Point that will replace `interpolation.x_base`.

cobyqa.models.Quadratic.solve_system

static Quadratic.**solve_system**(*interpolation*, *rhs*)

Solve the interpolation system.

Parameters

interpolation
[*cobyqa.models.Interpolation*] Interpolation set.

rhs
[numpy.ndarray, shape (npt + n + 1,)] Right-hand side vector of the interpolation system.

Returns

numpy.ndarray, shape (npt + n + 1,)
Solution of the interpolation system.

bool
Whether the interpolation system is ill-conditioned.

Raises

numpy.linalg.LinAlgError
If the interpolation system is ill-defined.

cobyqa.models.Quadratic.update

`Quadratic.update(interpolation, k_new, dir_old, values_diff)`

Update the quadratic model.

This method applies the derivative-free symmetric Broyden update to the quadratic model. The *knew*-th interpolation point must be updated before calling this method.

Parameters

interpolation
[`cobyqa.models.Interpolation`] Updated interpolation set.

k_new
[int] Index of the updated interpolation point.

dir_old
[`numpy.ndarray`, shape (n,)] Value of `interpolation.xpt[:, k_new]` before the update.

values_diff
[`numpy.ndarray`, shape (npt,)] Differences between the values of the interpolated nonlinear function and the previous quadratic model at the updated interpolation points.

Raises

`numpy.linalg.LinAlgError`
If the interpolation system is ill-defined.

3.2.3 cobyqa.models.Models

`class cobyqa.models.Models(pb, options)`

Models for a nonlinear optimization problem.

Attributes

`ceq_val`
Values of the nonlinear equality constraint functions at the interpolation points.

`cub_val`
Values of the nonlinear inequality constraint functions at the interpolation points.

`fun_val`
Values of the objective function at the interpolation points.

`interpolation`
Interpolation set.

`m_nonlinear_eq`
Number of nonlinear equality constraints.

`m_nonlinear_ub`
Number of nonlinear inequality constraints.

`n`
Dimension of the problem.

`npt`
Number of interpolation points.

Methods

ceq(x[, mask])	Evaluate the quadratic models of the nonlinear equality functions at a given point.
ceq_curv(v[, mask])	Evaluate the curvature of the quadratic models of the nonlinear equality functions along a given direction.
ceq_grad(x[, mask])	Evaluate the gradients of the quadratic models of the nonlinear equality functions at a given point.
ceq_hess([mask])	Evaluate the Hessian matrices of the quadratic models of the nonlinear equality functions.
ceq_hess_prod(v[, mask])	Evaluate the right product of the Hessian matrices of the quadratic models of the nonlinear equality functions with a given vector.
cub(x[, mask])	Evaluate the quadratic models of the nonlinear inequality functions at a given point.
cub_curv(v[, mask])	Evaluate the curvature of the quadratic models of the nonlinear inequality functions along a given direction.
cub_grad(x[, mask])	Evaluate the gradients of the quadratic models of the nonlinear inequality functions at a given point.
cub_hess([mask])	Evaluate the Hessian matrices of the quadratic models of the nonlinear inequality functions.
cub_hess_prod(v[, mask])	Evaluate the right product of the Hessian matrices of the quadratic models of the nonlinear inequality functions with a given vector.
determinants(x_new[, k_new])	Compute the normalized determinants of the new interpolation systems.
fun(x)	Evaluate the quadratic model of the objective function at a given point.
fun_alt_grad(x)	Evaluate the gradient of the alternative quadratic model of the objective function at a given point.
fun_curv(v)	Evaluate the curvature of the quadratic model of the objective function along a given direction.
fun_grad(x)	Evaluate the gradient of the quadratic model of the objective function at a given point.
fun_hess()	Evaluate the Hessian matrix of the quadratic model of the objective function.
fun_hess_prod(v)	Evaluate the right product of the Hessian matrix of the quadratic model of the objective function with a given vector.
reset_models()	Set the quadratic models of the objective function, nonlinear inequality constraints, and nonlinear equality constraints to the alternative quadratic models.
shift_x_base(new_x_base, options)	Shift the base point without changing the interpolation set.
update_interpolation(k_new, x_new, fun_val, ...)	Update the interpolation set.

cobyqa.models.Models.ceq

`Models.ceq(x, mask=None)`

Evaluate the quadratic models of the nonlinear equality functions at a given point.

Parameters

x
[`numpy.ndarray`, shape (n,)] Point at which to evaluate the quadratic models of the nonlinear equality functions.

mask
[`numpy.ndarray`, shape (m_nonlinear_eq,), optional] Mask of the quadratic models to consider.

Returns

`numpy.ndarray`
Values of the quadratic model of the nonlinear equality functions.

cobyqa.models.Models.ceq_curv

`Models.ceq_curv(v, mask=None)`

Evaluate the curvature of the quadratic models of the nonlinear equality functions along a given direction.

Parameters

v
[`numpy.ndarray`, shape (n,)] Direction along which the curvature of the quadratic models of the nonlinear equality functions is evaluated.

mask
[`numpy.ndarray`, shape (m_nonlinear_eq,), optional] Mask of the quadratic models to consider.

Returns

`numpy.ndarray`
Curvature of the quadratic models of the nonlinear equality functions along v.

cobyqa.models.Models.ceq_grad

`Models.ceq_grad(x, mask=None)`

Evaluate the gradients of the quadratic models of the nonlinear equality functions at a given point.

Parameters

x
[`numpy.ndarray`, shape (n,)] Point at which to evaluate the gradients of the quadratic models of the nonlinear equality functions.

mask
[`numpy.ndarray`, shape (m_nonlinear_eq,), optional] Mask of the quadratic models to consider.

Returns

`numpy.ndarray`
Gradients of the quadratic model of the nonlinear equality functions.

cobyqa.models.Models.ceq_hess

`Models.ceq_hess(mask=None)`

Evaluate the Hessian matrices of the quadratic models of the nonlinear equality functions.

Parameters

mask
[`numpy.ndarray`, shape (m_nonlinear_eq,), optional] Mask of the quadratic models to consider.

Returns

`numpy.ndarray`
Hessian matrices of the quadratic models of the nonlinear equality functions.

cobyqa.models.Models.ceq_hess_prod

`Models.ceq_hess_prod(v, mask=None)`

Evaluate the right product of the Hessian matrices of the quadratic models of the nonlinear equality functions with a given vector.

Parameters

v
[`numpy.ndarray`, shape (n,)] Vector with which the Hessian matrices of the quadratic models of the nonlinear equality functions are multiplied from the right.

mask
[`numpy.ndarray`, shape (m_nonlinear_eq,), optional] Mask of the quadratic models to consider.

Returns

`numpy.ndarray`
Right products of the Hessian matrices of the quadratic models of the nonlinear equality functions with *v*.

cobyqa.models.Models.cub

`Models.cub(x, mask=None)`

Evaluate the quadratic models of the nonlinear inequality functions at a given point.

Parameters

x
[`numpy.ndarray`, shape (n,)] Point at which to evaluate the quadratic models of the nonlinear inequality functions.

mask
[`numpy.ndarray`, shape (m_nonlinear_ub,), optional] Mask of the quadratic models to consider.

Returns

`numpy.ndarray`
Values of the quadratic model of the nonlinear inequality functions.

cobyqa.models.Models.cub_curv

`Models.cub_curv(v, mask=None)`

Evaluate the curvature of the quadratic models of the nonlinear inequality functions along a given direction.

Parameters

v
[`numpy.ndarray`, shape (n,)] Direction along which the curvature of the quadratic models of the nonlinear inequality functions is evaluated.

mask
[`numpy.ndarray`, shape (m_nonlinear_ub,), optional] Mask of the quadratic models to consider.

Returns

`numpy.ndarray`
Curvature of the quadratic models of the nonlinear inequality functions along *v*.

cobyqa.models.Models.cub_grad

`Models.cub_grad(x, mask=None)`

Evaluate the gradients of the quadratic models of the nonlinear inequality functions at a given point.

Parameters

x
[`numpy.ndarray`, shape (n,)] Point at which to evaluate the gradients of the quadratic models of the nonlinear inequality functions.

mask
[`numpy.ndarray`, shape (m_nonlinear_eq,), optional] Mask of the quadratic models to consider.

Returns

`numpy.ndarray`
Gradients of the quadratic model of the nonlinear inequality functions.

cobyqa.models.Models.cub_hess

`Models.cub_hess(mask=None)`

Evaluate the Hessian matrices of the quadratic models of the nonlinear inequality functions.

Parameters

mask
[`numpy.ndarray`, shape (m_nonlinear_ub,), optional] Mask of the quadratic models to consider.

Returns

`numpy.ndarray`
Hessian matrices of the quadratic models of the nonlinear inequality functions.

cobyqa.models.Models.cub_hess_prod

`Models.cub_hess_prod(v, mask=None)`

Evaluate the right product of the Hessian matrices of the quadratic models of the nonlinear inequality functions with a given vector.

Parameters

v
[numpy.ndarray, shape (n,)] Vector with which the Hessian matrices of the quadratic models of the nonlinear inequality functions are multiplied from the right.

mask
[numpy.ndarray, shape (m_nonlinear_ub,), optional] Mask of the quadratic models to consider.

Returns

numpy.ndarray
Right products of the Hessian matrices of the quadratic models of the nonlinear inequality functions with *v*.

cobyqa.models.Models.determinants

`Models.determinants(x_new, k_new=None)`

Compute the normalized determinants of the new interpolation systems.

Parameters

x_new
[numpy.ndarray, shape (n,)] New interpolation point. Its value is interpreted as relative to the origin, not the base point.

k_new
[int, optional] Index of the updated interpolation point. If *k_new* is not specified, all the possible determinants are computed.

Returns

{float, numpy.ndarray, shape (npt,)}
Determinant(s) of the new interpolation system.

Raises

numpy.linalg.LinAlgError
If the interpolation system is ill-defined.

Notes

The determinants are normalized by the determinant of the current interpolation system. For stability reasons, the calculations are done using the formula (2.12) in [1].

References

[1]

cobyqa.models.Models.fun

Models.**fun**(*x*)

Evaluate the quadratic model of the objective function at a given point.

Parameters

x
[numpy.ndarray, shape (n,)] Point at which to evaluate the quadratic model of the objective function.

Returns

float
Value of the quadratic model of the objective function at *x*.

cobyqa.models.Models.fun_alt_grad

Models.**fun_alt_grad**(*x*)

Evaluate the gradient of the alternative quadratic model of the objective function at a given point.

Parameters

x
[numpy.ndarray, shape (n,)] Point at which to evaluate the gradient of the alternative quadratic model of the objective function.

Returns

numpy.ndarray, shape (n,)
Gradient of the alternative quadratic model of the objective function at *x*.

Raises

numpy.linalg.LinAlgError
If the interpolation system is ill-defined.

cobyqa.models.Models.fun_curv

Models.**fun_curv**(*v*)

Evaluate the curvature of the quadratic model of the objective function along a given direction.

Parameters

v
[numpy.ndarray, shape (n,)] Direction along which the curvature of the quadratic model of the objective function is evaluated.

Returns

float
Curvature of the quadratic model of the objective function along *v*.

cobyqa.models.Models.fun_grad

`Models.fun_grad(x)`

Evaluate the gradient of the quadratic model of the objective function at a given point.

Parameters

x
[numpy.ndarray, shape (n,)] Point at which to evaluate the gradient of the quadratic model of the objective function.

Returns

numpy.ndarray, shape (n,)
Gradient of the quadratic model of the objective function at *x*.

cobyqa.models.Models.fun_hess

`Models.fun_hess()`

Evaluate the Hessian matrix of the quadratic model of the objective function.

Returns

numpy.ndarray, shape (n, n)
Hessian matrix of the quadratic model of the objective function.

cobyqa.models.Models.fun_hess_prod

`Models.fun_hess_prod(v)`

Evaluate the right product of the Hessian matrix of the quadratic model of the objective function with a given vector.

Parameters

v
[numpy.ndarray, shape (n,)] Vector with which the Hessian matrix of the quadratic model of the objective function is multiplied from the right.

Returns

numpy.ndarray, shape (n,)
Right product of the Hessian matrix of the quadratic model of the objective function with *v*.

cobyqa.models.Models.reset_models

`Models.reset_models()`

Set the quadratic models of the objective function, nonlinear inequality constraints, and nonlinear equality constraints to the alternative quadratic models.

Raises

numpy.linalg.LinAlgError
If the interpolation system is ill-defined.

cobyqa.models.Models.shift_x_base

Models.**shift_x_base**(*new_x_base*, *options*)

Shift the base point without changing the interpolation set.

Parameters

new_x_base
[numpy.ndarray, shape (n,)] New base point.

options
[dict] Options of the solver.

cobyqa.models.Models.update_interpolation

Models.**update_interpolation**(*k_new*, *x_new*, *fun_val*, *cub_val*, *ceq_val*)

Update the interpolation set.

This method updates the interpolation set by replacing the *knew*-th interpolation point with *xnew*. It also updates the function values and the quadratic models.

Parameters

k_new
[int] Index of the updated interpolation point.

x_new
[numpy.ndarray, shape (n,)] New interpolation point. Its value is interpreted as relative to the origin, not the base point.

fun_val
[float] Value of the objective function at *x_new*. Objective function value at *x_new*.

cub_val
[numpy.ndarray, shape (m_nonlinear_ub,)] Values of the nonlinear inequality constraints at *x_new*.

ceq_val
[numpy.ndarray, shape (m_nonlinear_eq,)] Values of the nonlinear equality constraints at *x_new*.

Raises

numpy.linalg.LinAlgError
If the interpolation system is ill-defined.

The *framework* module implements the trust-region framework used by COBYQA.

3.3 Trust-region framework

This module implements classes for representing the trust-region framework used by COBYQA.

TrustRegion(pb, options)	Trust-region framework.

3.3.1 cobyqa.framework.TrustRegion

class cobyqa.framework.**TrustRegion**(*pb*, *options*)

Trust-region framework.

Attributes

best_index
Index of the best interpolation point.

ceq_best
Values of the nonlinear equality constraints at `x_best`.

cub_best
Values of the nonlinear inequality constraints at `x_best`.

fun_best
Value of the objective function at `x_best`.

m_linear_eq
Number of linear equality constraints.

m_linear_ub
Number of linear inequality constraints.

m_nonlinear_eq
Number of nonlinear equality constraints.

m_nonlinear_ub
Number of nonlinear inequality constraints.

models
Models of the objective function and constraints.

n
Number of variables.

penalty
Penalty parameter.

radius
Trust-region radius.

resolution
Resolution of the trust-region framework.

x_best
Best interpolation point.

Methods

decrease_penalty()	Decrease the penalty parameter.
get_constraint_linearizations(x)	Get the linearizations of the constraints at a given point.
get_geometry_step(k_new, options)	Get the geometry-improving step.
get_index_to_remove([x_new])	Get the index of the interpolation point to remove.
get_reduction_ratio(step, fun_val, cub_val, ...)	Get the reduction ratio.
get_second_order_correction_step(step, options)	Get the second-order correction step.
get_trust_region_step(options)	Get the trust-region step.
increase_penalty(step)	Increase the penalty parameter.
lag_model(x)	Evaluate the Lagrangian model at a given point.
lag_model_curv(v)	Evaluate the curvature of the Lagrangian model along a given direction.
lag_model_grad(x)	Evaluate the gradient of the Lagrangian model at a given point.
lag_model_hess()	Evaluate the Hessian matrix of the Lagrangian model at a given point.
lag_model_hess_prod(v)	Evaluate the right product of the Hessian matrix of the Lagrangian model with a given vector.
merit(x[, fun_val, cub_val, ceq_val])	Evaluate the merit function at a given point.
reduce_resolution(options)	Reduce the resolution of the trust-region framework.
set_best_index()	Set the index of the best point.
set_multipliers(x)	Set the Lagrange multipliers.
shift_x_base(options)	Shift the base point to `x_best`.
sqp_ceq(step)	Evaluate the linearization of the nonlinear equality constraints.
sqp_cub(step)	Evaluate the linearization of the nonlinear inequality constraints.
sqp_fun(step)	Evaluate the objective function of the SQP subproblem.
update_radius(step, ratio)	Update the trust-region radius.

cobyqa.framework.TrustRegion.decrease_penalty

TrustRegion.**decrease_penalty**()

Decrease the penalty parameter.

cobyqa.framework.TrustRegion.get_constraint_linearizations

TrustRegion.**get_constraint_linearizations**(*x*)

Get the linearizations of the constraints at a given point.

Parameters

x

[`numpy.ndarray`, shape (n,)] Point at which the linearizations of the constraints are evaluated.

Returns

`numpy.ndarray`, shape (m_linear_ub + m_nonlinear_ub, n)

Left-hand side matrix of the linearized inequality constraints.

numpy.ndarray, shape (m_linear_ub + m_nonlinear_ub,)
Right-hand side vector of the linearized inequality constraints.

numpy.ndarray, shape (m_linear_eq + m_nonlinear_eq, n)
Left-hand side matrix of the linearized equality constraints.

numpy.ndarray, shape (m_linear_eq + m_nonlinear_eq,)
Right-hand side vector of the linearized equality constraints.

cobyqa.framework.TrustRegion.get_geometry_step

`TrustRegion.get_geometry_step`(*k_new*, *options*)

Get the geometry-improving step.

Three different geometry-improving steps are computed and the best one is returned. For more details, see Section 5.2.7 of [1].

Parameters

k_new
[int] Index of the interpolation point to be modified.

options
[dict] Options of the solver.

Returns

numpy.ndarray, shape (n,)
Geometry-improving step.

Raises

numpy.linalg.LinAlgError
If the computation of a determinant fails.

References

[1]

cobyqa.framework.TrustRegion.get_index_to_remove

`TrustRegion.get_index_to_remove`(*x_new=None*)

Get the index of the interpolation point to remove.

If *x_new* is not provided, the index returned should be used during the geometry-improvement phase. Otherwise, the index returned is the best index for included *x_new* in the interpolation set.

Parameters

x_new
[numpy.ndarray, shape (n,), optional] New point to be included in the interpolation set.

Returns

int
Index of the interpolation point to remove.

float
Distance between `x_best` and the removed point.

Raises

numpy.linalg.LinAlgError
If the computation of a determinant fails.

cobyqa.framework.TrustRegion.get_reduction_ratio

`TrustRegion.get_reduction_ratio`(*step*, *fun_val*, *cub_val*, *ceq_val*)

Get the reduction ratio.

Parameters

step
[`numpy.ndarray`, shape (n,)] Trust-region step.

fun_val
[float] Objective function value at the trial point.

cub_val
[`numpy.ndarray`, shape (m_nonlinear_ub,)] Nonlinear inequality constraint values at the trial point.

ceq_val
[`numpy.ndarray`, shape (m_nonlinear_eq,)] Nonlinear equality constraint values at the trial point.

Returns

float
Reduction ratio.

cobyqa.framework.TrustRegion.get_second_order_correction_step

`TrustRegion.get_second_order_correction_step`(*step*, *options*)

Get the second-order correction step.

Parameters

step
[`numpy.ndarray`, shape (n,)] Trust-region step.

options
[dict] Options of the solver.

Returns

`numpy.ndarray`, shape (n,)
Second-order correction step.

cobyqa.framework.TrustRegion.get_trust_region_step

`TrustRegion.get_trust_region_step`(*options*)

Get the trust-region step.

The trust-region step is computed by solving the derivative-free trust-region SQP subproblem using a Byrd-Omojokun composite-step approach. For more details, see Section 5.2.3 of [1].

Parameters

options
[dict] Options of the solver.

Returns

`numpy.ndarray`, shape (n,)
Normal step.

`numpy.ndarray`, shape (n,)
Tangential step.

References

[1]

cobyqa.framework.TrustRegion.increase_penalty

`TrustRegion.increase_penalty(step)`

Increase the penalty parameter.

Parameters

step
[`numpy.ndarray`, shape (n,)] Trust-region step.

cobyqa.framework.TrustRegion.lag_model

`TrustRegion.lag_model(x)`

Evaluate the Lagrangian model at a given point.

Parameters

x
[`numpy.ndarray`, shape (n,)] Point at which the Lagrangian model is evaluated.

Returns

float
Value of the Lagrangian model at x.

cobyqa.framework.TrustRegion.lag_model_curv

`TrustRegion.lag_model_curv(v)`

Evaluate the curvature of the Lagrangian model along a given direction.

Parameters

v
[`numpy.ndarray`, shape (n,)] Direction along which the curvature of the Lagrangian model is evaluated.

Returns

float
Curvature of the Lagrangian model along v.

cobyqa.framework.TrustRegion.lag_model_grad

`TrustRegion.lag_model_grad(x)`

Evaluate the gradient of the Lagrangian model at a given point.

Parameters

x
[`numpy.ndarray`, shape (n,)] Point at which the gradient of the Lagrangian model is evaluated.

Returns

`numpy.ndarray`, shape (n,)
Gradient of the Lagrangian model at x.

cobyqa.framework.TrustRegion.lag_model_hess

`TrustRegion.lag_model_hess()`

Evaluate the Hessian matrix of the Lagrangian model at a given point.

Returns

numpy.ndarray, shape (n, n)
Hessian matrix of the Lagrangian model at *x*.

cobyqa.framework.TrustRegion.lag_model_hess_prod

`TrustRegion.lag_model_hess_prod(v)`

Evaluate the right product of the Hessian matrix of the Lagrangian model with a given vector.

Parameters

v
[numpy.ndarray, shape (n,)] Vector with which the Hessian matrix of the Lagrangian model is multiplied from the right.

Returns

numpy.ndarray, shape (n,)
Right product of the Hessian matrix of the Lagrangian model with *v*.

cobyqa.framework.TrustRegion.merit

`TrustRegion.merit(x, fun_val=None, cub_val=None, ceq_val=None)`

Evaluate the merit function at a given point.

Parameters

x
[numpy.ndarray, shape (n,)] Point at which the merit function is evaluated.

fun_val
[float, optional] Value of the objective function at *x*. If not provided, the objective function is evaluated at *x*.

cub_val
[numpy.ndarray, shape (m_nonlinear_ub,), optional] Values of the nonlinear inequality constraints. If not provided, the nonlinear inequality constraints are evaluated at *x*.

ceq_val
[numpy.ndarray, shape (m_nonlinear_eq,), optional] Values of the nonlinear equality constraints. If not provided, the nonlinear equality constraints are evaluated at *x*.

Returns

float
Value of the merit function at *x*.

cobyqa.framework.TrustRegion.reduce_resolution

`TrustRegion.reduce_resolution(options)`

Reduce the resolution of the trust-region framework.

Parameters

options
[dict] Options of the solver.

cobyqa.framework.TrustRegion.set_best_index

`TrustRegion.set_best_index()`

Set the index of the best point.

cobyqa.framework.TrustRegion.set_multipliers

`TrustRegion.set_multipliers(x)`

Set the Lagrange multipliers.

This method computes and set the Lagrange multipliers of the linear and nonlinear constraints to be the QP multipliers.

Parameters

x
[`numpy.ndarray`, shape (n,)] Point at which the Lagrange multipliers are computed.

cobyqa.framework.TrustRegion.shift_x_base

`TrustRegion.shift_x_base(options)`

Shift the base point to `x_best`.

Parameters

options
[dict] Options of the solver.

cobyqa.framework.TrustRegion.sqp_ceq

`TrustRegion.sqp_ceq(step)`

Evaluate the linearization of the nonlinear equality constraints.

Parameters

step
[`numpy.ndarray`, shape (n,)] Step along which the linearization of the nonlinear equality constraints is evaluated.

Returns

`numpy.ndarray`, shape (m_nonlinear_ub,)
Value of the linearization of the nonlinear equality constraints along *step*.

cobyqa.framework.TrustRegion.sqp_cub

TrustRegion.**sqp_cub**(*step*)

Evaluate the linearization of the nonlinear inequality constraints.

Parameters

step
[numpy.ndarray, shape (n,)] Step along which the linearization of the nonlinear inequality constraints is evaluated.

Returns

numpy.ndarray, shape (m_nonlinear_ub,)
Value of the linearization of the nonlinear inequality constraints along *step*.

cobyqa.framework.TrustRegion.sqp_fun

TrustRegion.**sqp_fun**(*step*)

Evaluate the objective function of the SQP subproblem.

Parameters

step
[numpy.ndarray, shape (n,)] Step along which the objective function of the SQP subproblem is evaluated.

Returns

float
Value of the objective function of the SQP subproblem along *step*.

cobyqa.framework.TrustRegion.update_radius

TrustRegion.**update_radius**(*step*, *ratio*)

Update the trust-region radius.

Parameters

step
[numpy.ndarray, shape (n,)] Trust-region step.

ratio
[float] Reduction ratio.

The *subsolvers* module implements the subproblem solvers used by COBYQA.

3.4 Subproblem solvers

This module implements the subproblem solvers of COBYQA.

The trust-region subproblems, i.e., the normal and tangential Byrd-Omojokun subproblems, are solved approximately using variations of the truncated conjugate gradient method. The function below implements these methods.

normal_byrd_omojokun(aub, bub, aeq, beq, xl, ...)	Minimize approximately a linear constraint violation subject to bound constraints in a trust region.
tangential_byrd_omojokun(grad, hess_prod, ...)	Minimize approximately a quadratic function subject to bound constraints in a trust region.
constrained_tangential_byrd_omojokun(grad, ...)	Minimize approximately a quadratic function subject to bound and linear constraints in a trust region.

3.4.1 cobyqa.subsolvers.normal_byrd_omojokun

`cobyqa.subsolvers.`**`normal_byrd_omojokun`**(*aub*, *bub*, *aeq*, *beq*, *xl*, *xu*, *delta*, *debug*, ***kwargs*)

Minimize approximately a linear constraint violation subject to bound constraints in a trust region.

This function solves approximately

$$\min_{s \in \mathbb{R}^n} \quad \frac{1}{2} \big(\lVert \max\{A_{\mathcal{I}} s - b_{\mathcal{I}}, 0\} \rVert^2 + \lVert A_{\mathcal{E}} s - b_{\mathcal{E}} \rVert^2 \big) \quad \text{s.t.} \quad \left\{ \begin{array}{l} l \le s \le u, \\ \lVert s \rVert \le \Delta, \end{array} \right.$$

using a variation of the truncated conjugate gradient method.

Parameters

aub
[`numpy.ndarray`, shape (m_linear_ub, n)] Matrix $A_{\mathcal{I}}$ as shown above.

bub
[`numpy.ndarray`, shape (m_linear_ub,)] Vector $b_{\mathcal{I}}$ as shown above.

aeq
[`numpy.ndarray`, shape (m_linear_eq, n)] Matrix $A_{\mathcal{E}}$ as shown above.

beq
[`numpy.ndarray`, shape (m_linear_eq,)] Vector $b_{\mathcal{E}}$ as shown above.

xl
[`numpy.ndarray`, shape (n,)] Lower bounds l as shown above.

xu
[`numpy.ndarray`, shape (n,)] Upper bounds u as shown above.

delta
[float] Trust-region radius Δ as shown above.

debug
[bool] Whether to make debugging tests during the execution.

Returns

`numpy.ndarray`, shape (n,)
Approximate solution s.

Other Parameters

improve
[bool, optional] If True, a solution generated by the truncated conjugate gradient method that is on the boundary of the trust region is improved by moving around the trust-region boundary on the two-dimensional space spanned by the solution and the gradient of the quadratic function at the solution (default is True).

Notes

This function implements Algorithm 6.4 of [1]. It is assumed that the origin is feasible with respect to the bound constraints and that *delta* is finite and positive.

References

[1]

3.4.2 cobyqa.subsolvers.tangential_byrd_omojokun

cobyqa.subsolvers.**tangential_byrd_omojokun**(*grad*, *hess_prod*, *xl*, *xu*, *delta*, *debug*, ***kwargs*)

Minimize approximately a quadratic function subject to bound constraints in a trust region.

This function solves approximately

$$\min_{s \in \mathbb{R}^n} \quad g^{\mathsf{T}} s + \frac{1}{2} s^{\mathsf{T}} H s \quad \text{s.t.} \quad \begin{cases} l \le s \le u,\\ \|s\| \le \Delta, \end{cases}$$

using an active-set variation of the truncated conjugate gradient method.

Parameters

grad
[numpy.ndarray, shape (n,)] Gradient g as shown above.

hess_prod
[callable] Product of the Hessian matrix H with any vector.

```
hess_prod(s) -> `numpy.ndarray`, shape (n,)
```

returns the product Hs.

xl
[numpy.ndarray, shape (n,)] Lower bounds l as shown above.

xu
[numpy.ndarray, shape (n,)] Upper bounds u as shown above.

delta
[float] Trust-region radius Δ as shown above.

debug
[bool] Whether to make debugging tests during the execution.

Returns

numpy.ndarray, shape (n,)
Approximate solution s.

Other Parameters

improve
[bool, optional] If True, a solution generated by the truncated conjugate gradient method that is on the boundary of the trust region is improved by moving around the trust-region boundary on the two-dimensional space spanned by the solution and the gradient of the quadratic function at the solution (default is True).

Notes

This function implements Algorithm 6.2 of [1]. It is assumed that the origin is feasible with respect to the bound constraints and that *delta* is finite and positive.

References

[1]

3.4.3 cobyqa.subsolvers.constrained_tangential_byrd_omojokun

cobyqa.subsolvers.**constrained_tangential_byrd_omojokun**(*grad*, *hess_prod*, *xl*, *xu*, *aub*, *bub*, *aeq*, *delta*, *debug*, ***kwargs*)

Minimize approximately a quadratic function subject to bound and linear constraints in a trust region.

This function solves approximately

$$\min_{s \in \mathbb{R}^n} \quad g^{\mathsf{T}} s + \frac{1}{2} s^{\mathsf{T}} H s \quad \text{s.t.} \quad \begin{cases} l \le s \le u,\\ A_{\mathcal{I}} s \le b_{\mathcal{I}}, \; A_{\mathcal{E}} s = 0,\\ \lVert s \rVert \le \Delta, \end{cases}$$

using an active-set variation of the truncated conjugate gradient method.

Parameters

grad
[numpy.ndarray, shape (n,)] Gradient g as shown above.

hess_prod
[callable] Product of the Hessian matrix H with any vector.

```
hess_prod(s) -> `numpy.ndarray`, shape (n,)
```

returns the product Hs.

xl
[numpy.ndarray, shape (n,)] Lower bounds l as shown above.

xu
[numpy.ndarray, shape (n,)] Upper bounds u as shown above.

aub
[numpy.ndarray, shape (m_linear_ub, n)] Coefficient matrix $A_{\mathcal{I}}$ as shown above.

bub
[numpy.ndarray, shape (m_linear_ub,)] Right-hand side $b_{\mathcal{I}}$ as shown above.

aeq
[numpy.ndarray, shape (m_linear_eq, n)] Coefficient matrix $A_{\mathcal{E}}$ as shown above.

delta
[float] Trust-region radius Δ as shown above.

debug
[bool] Whether to make debugging tests during the execution.

Returns

numpy.ndarray, shape (n,)
Approximate solution s.

Other Parameters

improve
[bool, optional] If True, a solution generated by the truncated conjugate gradient method that is on the boundary of the trust region is improved by moving around the trust-region boundary on the two-dimensional space spanned by the solution and the gradient of the quadratic function at the solution (default is True).

Notes

This function implements Algorithm 6.3 of [1]. It is assumed that the origin is feasible with respect to the bound and linear constraints, and that *delta* is finite and positive.

References

[1]

The geometry-improving subproblems are solved approximately using techniques developed by Powell for his solver BOBYQA [DS1]. The functions below implement these techniques.

cauchy_geometry(const, grad, curv, xl, xu, ...)	Maximize approximately the absolute value of a quadratic function subject to bound constraints in a trust region.
spider_geometry(const, grad, curv, xpt, xl, ...)	Maximize approximately the absolute value of a quadratic function subject to bound constraints in a trust region.

3.4.4 cobyqa.subsolvers.cauchy_geometry

cobyqa.subsolvers.**cauchy_geometry**(*const*, *grad*, *curv*, *xl*, *xu*, *delta*, *debug*)

Maximize approximately the absolute value of a quadratic function subject to bound constraints in a trust region.

This function solves approximately

$$\max_{s \in \mathbb{R}^n} \quad \left| c + g^{\mathsf{T}} s + \frac{1}{2} s^{\mathsf{T}} H s \right| \quad \text{s.t.} \quad \left\{ \begin{array}{l} l \le s \le u, \\ \|s\| \le \Delta, \end{array} \right.$$

by maximizing the objective function along the constrained Cauchy direction.

Parameters

const
[float] Constant c as shown above.

grad
[numpy.ndarray, shape (n,)] Gradient g as shown above.

curv
[callable] Curvature of H along any vector.

```
curv(s) -> float
```

returns $s^{\mathsf{T}} H s$.

xl
[numpy.ndarray, shape (n,)] Lower bounds l as shown above.

xu
[numpy.ndarray, shape (n,)] Upper bounds u as shown above.

delta
[float] Trust-region radius Δ as shown above.

debug
[bool] Whether to make debugging tests during the execution.

Returns

numpy.ndarray, shape (n,)
Approximate solution s.

Notes

This function is described as the first alternative in Section 6.5 of [1]. It is assumed that the origin is feasible with respect to the bound constraints and that *delta* is finite and positive.

References

[1]

3.4.5 cobyqa.subsolvers.spider_geometry

`cobyqa.subsolvers.`**`spider_geometry`**(*const*, *grad*, *curv*, *xpt*, *xl*, *xu*, *delta*, *debug*)

Maximize approximately the absolute value of a quadratic function subject to bound constraints in a trust region.

This function solves approximately

$$\max_{s \in \mathbb{R}^n} \quad \left| c + g^{\mathsf{T}} s + \frac{1}{2} s^{\mathsf{T}} H s \right| \quad \text{s.t.} \quad \begin{cases} l \le s \le u, \\ \|s\| \le \Delta, \end{cases}$$

by maximizing the objective function along given straight lines.

Parameters

const
[float] Constant c as shown above.

grad
[`numpy.ndarray`, shape (n,)] Gradient g as shown above.

curv
[callable] Curvature of H along any vector.

```
curv(s) -> float
```

returns $s^{\mathsf{T}} H s$.

xpt
[`numpy.ndarray`, shape (n, npt)] Points defining the straight lines. The straight lines considered are the ones passing through the origin and the points in *xpt*.

xl
[`numpy.ndarray`, shape (n,)] Lower bounds l as shown above.

xu
[`numpy.ndarray`, shape (n,)] Upper bounds u as shown above.

delta
[float] Trust-region radius Δ as shown above.

debug
[bool] Whether to make debugging tests during the execution.

Returns

`numpy.ndarray`, shape (n,)
Approximate solution s.

Notes

This function is described as the second alternative in Section 6.5 of [1]. It is assumed that the origin is feasible with respect to the bound constraints and that *delta* is finite and positive.

References

[1]

Version
1.0

Useful links
Issue tracker | Mailing list

Authors
Tom M. Ragonneau | Zaikun Zhang

COBYQA is a derivative-free optimization solver designed to supersede COBYLA. Using only functions values, and no derivatives, it aims at solving problems of the form

$$\min_{x \in \mathcal{X}} \quad f(x) \quad \text{s.t.} \quad \begin{cases} A_{\mathcal{I}} x \le b_{\mathcal{I}}, \ A_{\mathcal{E}} x = b_{\mathcal{E}}, \\ c_{\mathcal{I}}(x) \le 0, \ c_{\mathcal{E}}(x) = 0, \end{cases}$$

where $\mathcal{X} = \{x \in \mathbb{R}^n : l \le x \le u\}$. COBYQA always respect the bound constraints throughout the optimization process. Hence, the nonlinear functions f, $c_{\mathcal{I}}$, and $c_{\mathcal{E}}$ do not need to be well-defined outside $\mathcal{X}$. In essence, COBYQA is a derivative-free trust-region SQP method based on quadratic models obtained by underdetermined interpolation. For a more detailed description of the algorithm, see the *framework description*.

To install COBYQA, run in your terminal

```
pip install cobyqa
```

For more details on the installation and the usage of COBYQA, see the *user guide*.

CHAPTER

FOUR

CITING COBYQA

If you would like to acknowledge the significance of COBYQA in your research, we suggest citing the project as follows.

- T. M. Ragonneau. "Model-Based Derivative-Free Optimization Methods and Software." PhD thesis. Hong Kong, China: Department of Applied Mathematics, The Hong Kong Polytechnic University, 2022. URL: https://theses.lib.polyu.edu.hk/handle/200/12294.
- T. M. Ragonneau and Z. Zhang. COBYQA Version 1.0.0. 2024. URL: https://www.cobyqa.com.

The corresponding BibTeX entries are given hereunder.

```
@phdthesis{rago_thesis,
    title        = {Model-Based Derivative-Free Optimization Methods and Software},
    author       = {Ragonneau, T. M.},
    school       = {Department of Applied Mathematics, The Hong Kong Polytechnic University},
    address      = {Hong Kong, China},
    year         = 2022,
    url          = {https://theses.lib.polyu.edu.hk/handle/200/12294},
}

@misc{razh_cobyqa,
    author       = {Ragonneau, T. M. and Zhang, Z.},
    title        = {{COBYQA} {V}ersion 1.0.0},
    year         = 2024,
    url          = {https://www.cobyqa.com},
}
```

CHAPTER

FIVE

STATISTICS

As of January 10, 2024, COBYQA has been downloaded 2,193 times, including

- 0 times on GitHub, and
- 2,193 times on PyPI (mirror downloads excluded).

CHAPTER SIX

ACKNOWLEDGMENTS

This work was partially supported by the Research Grants Council of Hong Kong under Grants PF18-24698, PolyU 253012/17P, PolyU 153054/20P, PolyU 153066/21P, and The Hong Kong Polytechnic University.

BIBLIOGRAPHY

[UU1] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Ser. Oper. Res. Financ. Eng. Springer, New York, NY, USA, second edition, 2006. doi:10.1007/978-0-387-40065-5.

[UU2] M. J. D. Powell. A direct search optimization method that models the objective and constraint functions by linear interpolation. In S. Gomez and J.-P. Hennart, editors, *Advances in Optimization and Numerical Analysis*, volume 275 of Math. Appl., pages 51–67. Springer, Dordrecht, Netherlands, 1994. doi:10.1007/978-94-015-8330-5_4.

[UF1] T. M. Ragonneau. *Model-Based Derivative-Free Optimization Methods and Software*. PhD thesis, Department of Applied Mathematics, The Hong Kong Polytechnic University, Hong Kong, China, 2022. URL: https://theses.lib.polyu.edu.hk/handle/200/12294.

[US1] T. M. Ragonneau. *Model-Based Derivative-Free Optimization Methods and Software*. PhD thesis, Department of Applied Mathematics, The Hong Kong Polytechnic University, Hong Kong, China, 2022. URL: https://theses.lib.polyu.edu.hk/handle/200/12294.

[1] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Ser. Oper. Res. Financ. Eng. Springer, New York, NY, USA, second edition, 2006. doi:10.1007/978-0-387-40065-5.

[2] M. J. D. Powell. A direct search optimization method that models the objective and constraint functions by linear interpolation. In S. Gomez and J.-P. Hennart, editors, *Advances in Optimization and Numerical Analysis*, volume 275 of Math. Appl., pages 51–67. Springer, Dordrecht, Netherlands, 1994. doi:10.1007/978-94-015-8330-5_4.

[3] T. M. Ragonneau. *Model-Based Derivative-Free Optimization Methods and Software*. PhD thesis, Department of Applied Mathematics, The Hong Kong Polytechnic University, Hong Kong, China, 2022. URL: https://theses.lib.polyu.edu.hk/handle/200/12294.

[1] M. J. D. Powell. The NEWUOA software for unconstrained optimization without derivatives. In G. Di Pillo and M. Roma, editors, *Large-Scale Nonlinear Optimization*, volume 83 of Nonconvex Optim. Appl., pages 255–297. Springer, Boston, MA, USA, 2006. doi:10.1007/0-387-30065-1_16.

[1] M. J. D. Powell. On updating the inverse of a KKT matrix. Technical Report DAMTP 2004/NA01, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, Cambridge, UK, 2004.

[1] T. M. Ragonneau. *Model-Based Derivative-Free Optimization Methods and Software*. PhD thesis, Department of Applied Mathematics, The Hong Kong Polytechnic University, Hong Kong, China, 2022. URL: https://theses.lib.polyu.edu.hk/handle/200/12294.

[1] T. M. Ragonneau. *Model-Based Derivative-Free Optimization Methods and Software*. PhD thesis, Department of Applied Mathematics, The Hong Kong Polytechnic University, Hong Kong, China, 2022. URL: https://theses.lib.polyu.edu.hk/handle/200/12294.

[1] T. M. Ragonneau. *Model-Based Derivative-Free Optimization Methods and Software*. PhD thesis, The Hong Kong Polytechnic University, Hong Kong, China, 2022.

[1] T. M. Ragonneau. *Model-Based Derivative-Free Optimization Methods and Software*. PhD thesis, The Hong Kong Polytechnic University, Hong Kong, China, 2022.

[1] T. M. Ragonneau. *Model-Based Derivative-Free Optimization Methods and Software*. PhD thesis, The Hong Kong Polytechnic University, Hong Kong, China, 2022.

[1] T. M. Ragonneau. *Model-Based Derivative-Free Optimization Methods and Software*. PhD thesis, The Hong Kong Polytechnic University, Hong Kong, China, 2022.

[1] T. M. Ragonneau. *Model-Based Derivative-Free Optimization Methods and Software*. PhD thesis, The Hong Kong Polytechnic University, Hong Kong, China, 2022.

[DS1] M. J. D. Powell. The BOBYQA algorithm for bound constrained optimization without derivatives. Technical Report DAMTP 2009/NA06, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, Cambridge, UK, 2009.

PYTHON MODULE INDEX

C

INDEX