Introduction to `nloptr`: an R interface to NLopt *

Jelmer Ypma

July 31, 2013

Abstract

This document describes how to use `nloptr`, which is an R interface to NLopt. NLopt is a free/open-source library for nonlinear optimization started by Steven G. Johnson, providing a common interface for a number of different free optimization routines available online as well as original implementations of various other algorithms. The NLopt library is available under the GNU Lesser General Public License (LGPL), and the copyrights are owned by a variety of authors.

1 Introduction

NLopt addresses general nonlinear optimization problems of the form:

$$\begin{aligned} & \min_{x \in R^n} f(x) \\ s.t. \quad & g(x) \leq 0 \\ & h(x) = 0 \\ & x_L \leq x \leq x_U \end{aligned}$$

where $f(\cdot)$ is the objective function and x represents the n optimization parameters. This problem may optionally be subject to the bound constraints (also called box constraints), x_L and x_U. For partially or totally unconstrained problems the bounds can take values $-\infty$ or ∞. One may also optionally have m nonlinear inequality constraints (sometimes called a nonlinear programming problem), which can be specified in $g(\cdot)$, and equality constraints that can be specified in $h(\cdot)$. Note that not all of the algorithms in NLopt can handle constraints.

This vignette describes how to formulate minimization problems to be solved with the R interface to NLopt. If you want to use the C interface directly or are interested in the Matlab interface, there are other sources of documentation avialable. Some of the information here has been taken from the NLopt website[1], where more details are available. All credit for implementing the C code for the different algorithms availalbe in NLopt should go to the respective authors. See the website[2] for information on how to cite NLopt and the algorithms you use.

*This package should be considered in beta and comments about any aspect of the package are welcome. This document is an R vignette prepared with the aid of `Sweave` (Leisch, 2002). Financial support of the UK Economic and Social Research Council through a grant (RES-589-28-0001) to the ESRC Centre for Microdata Methods and Practice (CeMMAP) is gratefully acknowledged.

[1] `http://ab-initio.mit.edu/nlopt`

[2] `http://ab-initio.mit.edu/wiki/index.php/Citing_NLopt`

2 Installation

This package is on CRAN and can be installed from within R using

```
> install.packages("nloptr")
```

You should now be able to load the R interface to NLopt and read the help.

```
> library('nloptr')
> ?nloptr
```

The most recent experimental version of `nloptr` can be installed from R-Forge using

```
> install.packages("nloptr",repos="http://R-Forge.R-project.org")
```

or from source using

```
> install.packages("nloptr",type="source",repos="http://R-Forge.R-project.org")
```

3 Minimizing the Rosenbrock Banana function

As a first example we will solve an unconstrained minimization problem. The function we look at is the Rosenbrock Banana function

$$f(x) = 100 \left(x_2 - x_1^2\right)^2 + (1 - x_1)^2 ,$$

which is also used as an example in the documentation for the standard R optimizer `optim`. The gradient of the objective function is given by

$$\nabla f(x) = \begin{pmatrix} -400 \cdot x_1 \cdot (x_2 - x_1^2) - 2 \cdot (1 - x_1) \\ 200 \cdot (x_2 - x_1^2) \end{pmatrix} .$$

Not all of the algorithms in NLopt need gradients to be supplied by the user. We will show examples with and without supplying the gradient. After loading the library

```
> library(nloptr)
```

we start by specifying the objective function and its gradient

```
> ## Rosenbrock Banana function
> eval_f <- function(x) {
      return( 100 * (x[2] - x[1] * x[1])^2 + (1 - x[1])^2 )
  }
> ## Gradient of Rosenbrock Banana function
> eval_grad_f <- function(x) {
      return( c( -400 * x[1] * (x[2] - x[1] * x[1]) - 2 * (1 - x[1]),
                  200 * (x[2] - x[1] * x[1]) ) )
  }
```

We define initial values

```
> # initial values
> x0 <- c( -1.2, 1 )
```

and then minimize the function using the `nloptr` command. This command runs some checks on the supplied inputs and returns an object with the exit code of the solver, the optimal value of the objective function and the solution. Before we can minimize the function we need to specify which algorithm we want to use

```
> opts <- list("algorithm"="NLOPT_LD_LBFGS",
               "xtol_rel"=1.0e-8)
```

Here we use the L-BFGS algorithm (Liu and Nocedal, 1989; Nocedal, 1980). The characters `LD` in the algorithm show that this algorithm looks for local minima (`L`) using a derivative-based (`D`) algorithm. Other algorithms look for global (`G`) minima, or they don't need derivatives (`N`). We also specified the termination criterium in terms of the relative x-tolerance. Other termination criteria are available (see Appendix A for a full list of options). We then solve the minimization problem using

```
> # solve Rosenbrock Banana function
> res <- nloptr( x0=x0,
                 eval_f=eval_f,
                 eval_grad_f=eval_grad_f,
                 opts=opts)
```

We can see the results by printing the resulting object.

```
> print( res )
Call:

nloptr(x0 = x0, eval_f = eval_f, eval_grad_f = eval_grad_f, opts = opts)



Minimization using NLopt version 2.3.0

NLopt solver status: 1 ( NLOPT_SUCCESS: Generic success
return value. )

Number of Iterations....: 56
Termination conditions:  xtol_rel: 1e-08
Number of inequality constraints:  0
Number of equality constraints:    0
Optimal value of objective function:  6.0983317523442e-23
Optimal value of controls: 1 1
```

Sometimes the objective function and its gradient contain common terms. To economize on calculations, we can return the objective and its gradient in a list. For the Rosenbrock Banana function we have for instance

```
> ## Rosenbrock Banana function and gradient in one function
> eval_f_list <- function(x) {
     common_term <- x[2] - x[1] * x[1]
     return( list( "objective" = 100 * common_term^2 + (1 - x[1])^2,
                   "gradient"  = c( -400 * x[1] * common_term - 2 * (1 - x[1]),
                                     200 * common_term) ) )
 }
```

which we minimize using

```
> res <- nloptr( x0=x0,
                 eval_f=eval_f_list,
                 opts=opts)
> print( res )
Call:
nloptr(x0 = x0, eval_f = eval_f_list, opts = opts)


Minimization using NLopt version 2.3.0

NLopt solver status: 1 ( NLOPT_SUCCESS: Generic success
return value. )

Number of Iterations....: 56
Termination conditions:  xtol_rel: 1e-08
Number of inequality constraints:  0
Number of equality constraints:    0
Optimal value of objective function:  6.0983317523442e-23
Optimal value of controls: 1 1
```

This gives the same results as before.

4 Minimization with inequality constraints

This section shows how to minimize a function subject to inequality constraints. This example is the same as the one used in the tutorial on the NLopt website. The problem we want to solve is

$$\begin{aligned} & \min_{x \in R^n} \sqrt{x_2} \\ s.t. \quad & x_2 \geq 0 \\ & x_2 \geq (a_1 x_1 + b_1)^3 \\ & x_2 \geq (a_2 x_1 + b_2)^3, \end{aligned}$$

where $a_1 = 2$, $b_1 = 0$, $a_2 = -1$, and $b_2 = 1$. In order to solve this problem, we first have to re-formulate the constraints to be of the form $g(x) \leq 0$. Note that the first constraint is a bound on x_2, which we will add later. The other two constraints can be re-written as

$$\begin{aligned} (a_1 x_1 + b_1)^3 - x_2 &\leq 0 \\ (a_2 x_1 + b_2)^3 - x_2 &\leq 0. \end{aligned}$$

First we define R functions to calculate the objective function and its gradient

```
> # objective function
> eval_f0 <- function( x, a, b ){
      return( sqrt(x[2]) )
  }
> # gradient of objective function
> eval_grad_f0 <- function( x, a, b ){
      return( c( 0, .5/sqrt(x[2]) ) )
  }
```

If needed, these can of course be calculated in the same function as before. Then we define the two constraints and the jacobian of the constraints

```
> # constraint function
> eval_g0 <- function( x, a, b ) {
      return( (a*x[1] + b)^3 - x[2] )
  }
> # jacobian of constraint
> eval_jac_g0 <- function( x, a, b ) {
      return( rbind( c( 3*a[1]*(a[1]*x[1] + b[1])^2, -1.0 ),
                     c( 3*a[2]*(a[2]*x[1] + b[2])^2, -1.0 ) ) )
  }
```

Note that all of the functions above depend on additional parameters, `a` and `b`. We have to supply specific values for these when we invoke the optimization command. The constraint function `eval_g0` returns a vector with in this case the same length as the vectors `a` and `b`. The function calculating the jacobian of the constraint should return a matrix where the number of rows equal the number of constraints (in this case two). The number of columns should equal the number of control variables (two in this case as well).

After defining values for the parameters

```
> # define parameters
> a <- c(2,-1)
> b <- c(0, 1)
```

we can minimize the function subject to the constraints with the following command

```
> # Solve using NLOPT_LD_MMA with gradient information supplied in separate function
> res0 <- nloptr( x0=c(1.234,5.678),
                  eval_f=eval_f0,
                  eval_grad_f=eval_grad_f0,
                  lb = c(-Inf,0),
                  ub = c(Inf,Inf),
                  eval_g_ineq = eval_g0,
                  eval_jac_g_ineq = eval_jac_g0,
                  opts = list("algorithm" = "NLOPT_LD_MMA",
                              "xtol_rel"=1.0e-8,
                              "print_level" = 2,
```

```
                            "check_derivatives" = TRUE,
                            "check_derivatives_print" = "all"),
                a = a,
                b = b )
Checking gradients of objective function.
Derivative checker results: 0 error(s) detected.

  eval_grad_f[ 1 ] = 0.000000e+00 ~ 0.000000e+00   [0.000000e+00]
  eval_grad_f[ 2 ] = 2.098323e-01 ~ 2.098323e-01   [1.422937e-09]

Checking gradients of inequality constraints.
Derivative checker results: 0 error(s) detected.

  eval_jac_g_ineq[ 1, 1 ] =  3.654614e+01 ~  3.654614e+01   [1.667794e-08]
  eval_jac_g_ineq[ 2, 1 ] = -1.642680e-01 ~ -1.642680e-01   [2.103453e-07]
  eval_jac_g_ineq[ 1, 2 ] = -1.000000e+00 ~ -1.000000e+00   [0.000000e+00]
  eval_jac_g_ineq[ 2, 2 ] = -1.000000e+00 ~ -1.000000e+00   [0.000000e+00]

iteration: 1
        f(x) = 2.382855
        g(x) = ( 9.354647, -5.690813 )
iteration: 2
        f(x) = 2.356135
        g(x) = ( -0.122989, -5.549587 )
iteration: 3
        f(x) = 2.245864
        g(x) = ( -0.531886, -5.038655 )
iteration: 4
        f(x) = 2.019102
        g(x) = ( -3.225103, -3.931195 )
iteration: 5
        f(x) = 1.740934
        g(x) = ( -2.676263, -2.761136 )
iteration: 6
        f(x) = 1.404206
        g(x) = ( -1.674056, -1.676216 )
iteration: 7
        f(x) = 1.022295
        g(x) = ( -0.748790, -0.748792 )
iteration: 8
        f(x) = 0.685203
        g(x) = ( -0.173206, -0.173207 )
iteration: 9
        f(x) = 0.552985
        g(x) = ( -0.009496, -0.009496 )
iteration: 10
        f(x) = 0.544354
        g(x) = ( -0.000025, -0.000025 )
iteration: 11
```

```
        f(x) = 0.544331
        g(x) = ( 0.000000, 0.000000 )
iteration: 12
        f(x) = 0.544331
        g(x) = ( -0.000000, 0.000000 )
iteration: 13
        f(x) = 0.544331
        g(x) = ( -0.000000, 0.000000 )
iteration: 14
        f(x) = 0.544331
        g(x) = ( -0.000000, 0.000000 )
iteration: 15
        f(x) = 0.544331
        g(x) = ( -0.000000, 0.000000 )
iteration: 16
        f(x) = 0.544331
        g(x) = ( -0.000000, 0.000000 )
iteration: 17
        f(x) = 0.544331
        g(x) = ( -0.000000, 0.000000 )
iteration: 18
        f(x) = 0.544331
        g(x) = ( -0.000000, 0.000000 )
iteration: 19
        f(x) = 0.544331
        g(x) = ( -0.000000, 0.000000 )
iteration: 20
        f(x) = 0.544331
        g(x) = ( -0.000000, 0.000000 )
iteration: 21
        f(x) = 0.544331
        g(x) = ( 0.000000, -0.000000 )
iteration: 22
        f(x) = 0.544331
        g(x) = ( 0.000000, -0.000000 )
iteration: 23
        f(x) = 0.544331
        g(x) = ( 0.000000, -0.000000 )
iteration: 24
        f(x) = 0.544331
        g(x) = ( 0.000000, -0.000000 )
iteration: 25
        f(x) = 0.544331
        g(x) = ( 0.000000, -0.000000 )
> print( res0 )
Call:

nloptr(x0 = c(1.234, 5.678), eval_f = eval_f0, eval_grad_f = eval_grad_f0,
    lb = c(-Inf, 0), ub = c(Inf, Inf), eval_g_ineq = eval_g0,
```

```
    eval_jac_g_ineq = eval_jac_g0, opts = list(algorithm = "NLOPT_LD_MMA",
        xtol_rel = 1e-08, print_level = 2, check_derivatives = TRUE,
        check_derivatives_print = "all"), a = a, b = b)


Minimization using NLopt version 2.3.0

NLopt solver status: 4 ( NLOPT_XTOL_REACHED: Optimization
stopped because xtol_rel or xtol_abs (above) was reached.
)

Number of Iterations....: 25
Termination conditions:  xtol_rel: 1e-08
Number of inequality constraints:  2
Number of equality constraints:    0
Optimal value of objective function:  0.54433104799443
Optimal value of controls: 0.3333333 0.2962963
```

Here we supplied lower bounds for x_2 in `lb`. There are no upper bounds for both control variables, so we supply `Inf` values. If we don't supply lower or upper bounds, plus or minus infinity is chosen by default. The inequality constraints and its jacobian are defined using `eval_g_ineq` and `eval_jac_g_ineq`. Not all algorithms can handle inequality constraints, so we have to specifiy one that does, `NLOPT_LD_MMA` (Svanberg, 2002).

We also specify the option `print_level` to obtain output during the optimization process. For the available `print_level` values, see `?nloptr`. Setting the `check_derivatives` option to `TRUE`, compares the gradients supplied by the user with a finite difference approximation in the initial point (`x0`). When this check is run, the option `check_derivatives_print` can be used to print all values of the derivative checker (`all` (default)), only those values that result in an error (`errors`) or no output (`none`), in which case only the number of errors is shown. The tolerance that determines if a difference between the analytic gradient and the finite difference approximation results in an error can be set using the option `check_derivatives_tol` (default = 1e-04). The first column shows the value of the analytic gradient, the second column shows the value of the finite difference approximation, and the third column shows the relative error. Stars are added at the front of a line if the relative error is larger than the specified tolerance.

Finally, we add all the parameters that have to be passed on to the objective and constraint functions, `a` and `b`.

We can also use a different algorithm to solve the same minimization problem. The only thing we have to change is the algorithm that we want to use, in this case `NLOPT_LN_COBYLA`, which is an algorithm that doesn't need gradient information (Powell, 1994, 1998).

```
> # Solve using NLOPT_LN_COBYLA without gradient information
> res1 <- nloptr( x0=c(1.234,5.678),
                  eval_f=eval_f0,
                  lb = c(-Inf,0),
                  ub = c(Inf,Inf),
```

```
                eval_g_ineq = eval_g0,
                opts = list("algorithm"="NLOPT_LN_COBYLA",
                            "xtol_rel"=1.0e-8),
                a = a,
                b = b )
> print( res1 )
Call:

nloptr(x0 = c(1.234, 5.678), eval_f = eval_f0, lb = c(-Inf, 0),
    ub = c(Inf, Inf), eval_g_ineq = eval_g0, opts = list(algorithm = "NLOPT_LN_COBYLA"
        xtol_rel = 1e-08), a = a, b = b)


Minimization using NLopt version 2.3.0

NLopt solver status: 4 ( NLOPT_XTOL_REACHED: Optimization
stopped because xtol_rel or xtol_abs (above) was reached.
)

Number of Iterations....: 50
Termination conditions:  xtol_rel: 1e-08
Number of inequality constraints:  2
Number of equality constraints:    0
Optimal value of objective function:  0.544331053951819
Optimal value of controls: 0.3333333 0.2962963
```

5 Derivative checker

The derivative checker can be called when supplying a minimization problem to `nloptr`, using the options `check_derivatives`, `check_derivatives_tol` and `check_derivatives_print`, but it can also be used separately. For example, define the function `g`, with vector outcome, and its gradient `g_grad`

```
> g <- function( x, a ) {
     return(
         c( x[1] - a[1],
            x[2] - a[2],
           (x[1] - a[1])^2,
           (x[2] - a[2])^2,
           (x[1] - a[1])^3,
           (x[2] - a[2])^3
         )
     )
 }
> g_grad <- function( x, a ) {
     return(
         rbind(
             c( 1, 0 ),
             c( 0, 1 ),
```

```
            c( 2*(x[1] - a[1]), 0 ),
            c( 2*(x[1] - a[1]), 2*(x[2] - a[2]) ),
            c( 3*(x[1] - a[2])^2, 0 ),
            c( 0, 3*(x[2] - a[2])^2 )
        )
    )
 }
```

`a` is some vector containing data. The gradient contains some errors in this case. By calling the function `check.derivatives` we can check the user-supplied analytic gradients with a finite difference approximation at a point `.x`.

```
> res <- check.derivatives(
                          .x=c(1,2),
                          func=g,
                          func_grad=g_grad,
                          check_derivatives_print='all',
                          a=c(.3, .8) )
Derivative checker results: 2 error(s) detected.

  grad_f[ 1, 1 ] = 1.00e+00 ~ 1.00e+00   [0.000000e+00]
  grad_f[ 2, 1 ] = 0.00e+00 ~ 0.00e+00   [0.000000e+00]
  grad_f[ 3, 1 ] = 1.40e+00 ~ 1.40e+00   [9.579318e-09]
* grad_f[ 4, 1 ] = 1.40e+00 ~ 0.00e+00   [1.400000e+00]
* grad_f[ 5, 1 ] = 1.20e-01 ~ 1.47e+00   [9.183673e-01]
  grad_f[ 6, 1 ] = 0.00e+00 ~ 0.00e+00   [0.000000e+00]
  grad_f[ 1, 2 ] = 0.00e+00 ~ 0.00e+00   [0.000000e+00]
  grad_f[ 2, 2 ] = 1.00e+00 ~ 1.00e+00   [0.000000e+00]
  grad_f[ 3, 2 ] = 0.00e+00 ~ 0.00e+00   [0.000000e+00]
  grad_f[ 4, 2 ] = 2.40e+00 ~ 2.40e+00   [1.179675e-08]
  grad_f[ 5, 2 ] = 0.00e+00 ~ 0.00e+00   [0.000000e+00]
  grad_f[ 6, 2 ] = 4.32e+00 ~ 4.32e+00   [2.593906e-08]
```

The errors are shown on screen, where the option `check_derivatives_print` determines the amount of output you see. The value of the analytic gradient and the value of the finite difference approximation at the supplied point is returned in a list.

```
> res
$analytic
     [,1] [,2]
[1,] 1.00 0.00
[2,] 0.00 1.00
[3,] 1.40 0.00
[4,] 1.40 2.40
[5,] 0.12 0.00
[6,] 0.00 4.32

$finite_difference
     [,1] [,2]
[1,] 1.00 0.00
```

```
[2,] 0.00 1.00
[3,] 1.40 0.00
[4,] 0.00 2.40
[5,] 1.47 0.00
[6,] 0.00 4.32
```

Note that not all errors will be picked up by the derivative checker. For instance, if we run the check with `a = c(.5, .5)`, one of the errors is not flagged as an error.

6 Notes

The `.R` scripts in the `tests` directory contain more examples. For instance, `hs071.R` and `systemofeq.R` show how to solve problems with equality constraints. See also `http://ab-initio.mit.edu/wiki/index.php/NLopt_Algorithms#Augmented_Lagrangia` for more details. Please let me know if you're missing any of the features that are implemented in NLopt.

Sometimes the optimization procedure terminates with a message `maxtime was reached` without evaluating the objective function. Submitting the same problem again usually solves this problem.

References

Steven G. Johnson. The NLopt nonlinear-optimization package. `http://ab-initio.mit.edu/nlopt`.

Friedrich Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In Wolfgang Härdle and Bernd Rönz, editors, *Compstat 2002 — Proceedings in Computational Statistics*, pages 575–580. Physica Verlag, Heidelberg, 2002. URL `http://www.stat.uni-muenchen.de/ leisch/Sweave`. ISBN 3-7908-1517-9.

D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Math. Programming*, 45:503–528, 1989.

J. Nocedal. Updating quasi-Newton matrices with limited storage. *Math. Comput.*, 35:773–782, 1980.

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*, pages 51–67. Kluwer Academic, Dordrecht, 1994.

M. J. D. Powell. Direct search algorithms for optimization calculations. *Acta Numerica*, 7:287–336, 1998.

Krister Svanberg. A class of globally convergent optimization methods based on conservative convex separable approximations. *SIAM J. Optim.*, 12(2): 555–573, 2002.

A Description of options

```
> nloptr.print.options()
algorithm
        possible values: NLOPT_GN_DIRECT, NLOPT_GN_DIRECT_L,
                         NLOPT_GN_DIRECT_L_RAND, NLOPT_GN_DIRECT_NOSCAL,
                         NLOPT_GN_DIRECT_L_NOSCAL,
                         NLOPT_GN_DIRECT_L_RAND_NOSCAL,
                         NLOPT_GN_ORIG_DIRECT, NLOPT_GN_ORIG_DIRECT_L,
                         NLOPT_GD_STOGO, NLOPT_GD_STOGO_RAND,
                         NLOPT_LD_SLSQP, NLOPT_LD_LBFGS_NOCEDAL,
                         NLOPT_LD_LBFGS, NLOPT_LN_PRAXIS, NLOPT_LD_VAR1,
                         NLOPT_LD_VAR2, NLOPT_LD_TNEWTON,
                         NLOPT_LD_TNEWTON_RESTART,
                         NLOPT_LD_TNEWTON_PRECOND,
                         NLOPT_LD_TNEWTON_PRECOND_RESTART,
                         NLOPT_GN_CRS2_LM, NLOPT_GN_MLSL, NLOPT_GD_MLSL,
                         NLOPT_GN_MLSL_LDS, NLOPT_GD_MLSL_LDS,
                         NLOPT_LD_MMA, NLOPT_LN_COBYLA, NLOPT_LN_NEWUOA,
                         NLOPT_LN_NEWUOA_BOUND, NLOPT_LN_NELDERMEAD,
                         NLOPT_LN_SBPLX, NLOPT_LN_AUGLAG, NLOPT_LD_AUGLAG,
                         NLOPT_LN_AUGLAG_EQ, NLOPT_LD_AUGLAG_EQ,
                         NLOPT_LN_BOBYQA, NLOPT_GN_ISRES
        default value:   none

        This option is required. Check the NLopt website for a description of
        the algorithms.

stopval
        possible values: -Inf <= stopval <= Inf
        default value:   -Inf

        Stop minimization when an objective value <= stopval is found.
        Setting stopval to -Inf disables this stopping criterion (default).

ftol_rel
        possible values: ftol_rel > 0
        default value:   0.0

        Stop when an optimization step (or an estimate of the optimum)
        changes the objective function value by less than ftol_rel multiplied
        by the absolute value of the function value. If there is any chance
        that your optimum function value is close to zero, you might want to
        set an absolute tolerance with ftol_abs as well. Criterion is
        disabled if ftol_rel is non-positive (default).

ftol_abs
        possible values: ftol_abs > 0
        default value:   0.0
```

```
        Stop when an optimization step (or an estimate of the optimum)
        changes the function value by less than ftol_abs. Criterion is
        disabled if ftol_abs is non-positive (default).

xtol_rel
        possible values: xtol_rel > 0
        default value:   1.0e-04

        Stop when an optimization step (or an estimate of the optimum)
        changes every parameter by less than xtol_rel multiplied by the
        absolute value of the parameter. If there is any chance that an
        optimal parameter is close to zero, you might want to set an absolute
        tolerance with xtol_abs as well. Criterion is disabled if xtol_rel is
        non-positive.

xtol_abs
        possible values: xtol_abs > 0
        default value:   rep( 0.0, length(x0) )

        xtol_abs is a vector of length n (the number of elements in x) giving
        the tolerances: stop when an optimization step (or an estimate of the
        optimum) changes every parameter x[i] by less than xtol_abs[i].
        Criterion is disabled if all elements of xtol_abs are non-positive
        (default).

maxeval
        possible values: maxeval is a positive integer
        default value:   100

        Stop when the number of function evaluations exceeds maxeval. This is
        not a strict maximum: the number of function evaluations may exceed
        maxeval slightly, depending upon the algorithm. Criterion is disabled
        if maxeval is non-positive.

maxtime
        possible values: maxtime > 0
        default value:   0.0

        Stop when the optimization time (in seconds) exceeds maxtime. This is
        not a strict maximum: the time may exceed maxtime slightly, depending
        upon the algorithm and on how slow your function evaluation is.
        Criterion is disabled if maxtime is non-positive (default).

tol_constraints_ineq
        possible values: tol_constraints_ineq > 0.0
        default value:   rep( 1e-8, num_constraints_ineq )

        The parameter tol_constraints_ineq is a vector of tolerances. Each
        tolerance corresponds to one of the inequality constraints. The
```

tolerance is used for the purpose of stopping criteria only: a point x is considered feasible for judging whether to stop the optimization if eval_g_ineq(x) <= tol. A tolerance of zero means that NLopt will try not to consider any x to be converged unless eval_g_ineq(x) is strictly non-positive; generally, at least a small positive tolerance is advisable to reduce sensitivity to rounding errors. By default the tolerances for all inequality constraints are set to 1e-8.

tol_constraints_eq
possible values: tol_constraints_eq > 0.0
default value: rep(1e-8, num_constraints_eq)

The parameter tol_constraints_eq is a vector of tolerances. Each tolerance corresponds to one of the equality constraints. The tolerance is used for the purpose of stopping criteria only: a point x is considered feasible for judging whether to stop the optimization if abs(eval_g_ineq(x)) <= tol. For equality constraints, a small positive tolerance is strongly advised in order to allow NLopt to converge even if the equality constraint is slightly nonzero. By default the tolerances for all equality constraints are set to 1e-8.

print_level
possible values: 0, 1, 2, or 3
default value: 0

The option print_level controls how much output is shown during the optimization process. Possible values: 0 (default): no output; 1: show iteration number and value of objective function; 2: 1 + show value of (in)equalities; 3: 2 + show value of controls.

check_derivatives
possible values: TRUE or FALSE
default value: FALSE

The option check_derivatives can be activated to compare the user-supplied analytic gradients with finite difference approximations.

check_derivatives_tol
possible values: check_derivatives_tol > 0.0
default value: 1e-04

The option check_derivatives_tol determines when a difference between an analytic gradient and its finite difference approximation is flagged as an error.

check_derivatives_print
possible values: 'none', 'all', 'errors',
default value: all

The option check_derivatives_print controls the output of the derivative checker (if check_derivatives==TRUE). All comparisons are shown ('all'), only those comparisions that resulted in an error ('error'), or only the number of errors is shown ('none').

print_options_doc
possible values: TRUE or FALSE
default value: FALSE

If TRUE, a description of all options and their current and default values is printed to the screen.

population
possible values: population is a positive integer
default value: 0

Several of the stochastic search algorithms (e.g., CRS, MLSL, and ISRES) start by generating some initial population of random points x. By default, this initial population size is chosen heuristically in some algorithm-specific way, but the initial population can by changed by setting a positive integer value for population. A population of zero implies that the heuristic default will be used.

ranseed
possible values: ranseed is a positive integer
default value: 0

For stochastic optimization algorithms, pseudorandom numbers are generated. Set the random seed using ranseed if you want to use a 'deterministic' sequence of pseudorandom numbers, i.e. the same sequence from run to run. If ranseed is 0 (default), the seed for the random numbers is generated from the system time, so that you will get a different sequence of pseudorandom numbers each time you run your program.