scipy / **scipy**

Branch: master ▾ **scipy** / scipy / optimize / **lbfgsb.py** Find file Copy path

pv ENH: optimize: call callback with a copy for scipy.optimize 7f43e34 4 days ago

19 contributors

469 lines (397 sloc) 16.5 KB

```python
"""
Functions
---------
.. autosummary::
   :toctree: generated/

    fmin_l_bfgs_b

"""

## License for the Python wrapper
## ==============================

## Copyright (c) 2004 David M. Cooke <cookedm@physics.mcmaster.ca>

## Permission is hereby granted, free of charge, to any person obtaining a
## copy of this software and associated documentation files (the "Software"),
## to deal in the Software without restriction, including without limitation
## the rights to use, copy, modify, merge, publish, distribute, sublicense,
## and/or sell copies of the Software, and to permit persons to whom the
## Software is furnished to do so, subject to the following conditions:

## The above copyright notice and this permission notice shall be included in
## all copies or substantial portions of the Software.

## THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
## IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
## FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
## AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
## LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
## FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
## DEALINGS IN THE SOFTWARE.

## Modifications by Travis Oliphant and Enthought, Inc. for inclusion in SciPy

from __future__ import division, print_function, absolute_import

import numpy as np
from numpy import array, asarray, float64, int32, zeros
from . import _lbfgsb
from .optimize import (approx_fprime, MemoizeJac, OptimizeResult,
                       _check_unknown_options, wrap_function,
                       _approx_fprime_helper)
from scipy.sparse.linalg import LinearOperator

__all__ = ['fmin_l_bfgs_b', 'LbfgsInvHessProduct']


def fmin_l_bfgs_b(func, x0, fprime=None, args=(),
                  approx_grad=0,
                  bounds=None, m=10, factr=1e7, pgtol=1e-5,
                  epsilon=1e-8,
                  iprint=-1, maxfun=15000, maxiter=15000, disp=None,
                  callback=None, maxls=20):
    """
    Minimize a function func using the L-BFGS-B algorithm.

    Parameters
    ----------
    func : callable f(x,*args)
        Function to minimise.
    x0 : ndarray
```

```
        Initial guess.
    fprime : callable fprime(x,*args), optional
        The gradient of `func`.  If None, then `func` returns the function
        value and the gradient (``f, g = func(x, *args)``), unless
        `approx_grad` is True in which case `func` returns only ``f``.
    args : sequence, optional
        Arguments to pass to `func` and `fprime`.
    approx_grad : bool, optional
        Whether to approximate the gradient numerically (in which case
        `func` returns only the function value).
    bounds : list, optional
        ``(min, max)`` pairs for each element in ``x``, defining
        the bounds on that parameter. Use None or +-inf for one of ``min`` or
        ``max`` when there is no bound in that direction.
    m : int, optional
        The maximum number of variable metric corrections
        used to define the limited memory matrix. (The limited memory BFGS
        method does not store the full hessian but uses this many terms in an
        approximation to it.)
    factr : float, optional
        The iteration stops when
        ``(f^k - f^{k+1})/max{|f^k|,|f^{k+1}|,1} <= factr * eps``,
        where ``eps`` is the machine precision, which is automatically
        generated by the code. Typical values for `factr` are: 1e12 for
        low accuracy; 1e7 for moderate accuracy; 10.0 for extremely
        high accuracy. See Notes for relationship to `ftol`, which is exposed
        (instead of `factr`) by the `scipy.optimize.minimize` interface to
        L-BFGS-B.
    pgtol : float, optional
        The iteration will stop when
        ``max{|proj g_i | i = 1, ..., n} <= pgtol``
        where ``pg_i`` is the i-th component of the projected gradient.
    epsilon : float, optional
        Step size used when `approx_grad` is True, for numerically
        calculating the gradient
    iprint : int, optional
        Controls the frequency of output. ``iprint < 0`` means no output;
        ``iprint = 0``    print only one line at the last iteration;
        ``0 < iprint < 99`` print also f and ``|proj g|`` every iprint iterations;
        ``iprint = 99``   print details of every iteration except n-vectors;
        ``iprint = 100``  print also the changes of active set and final x;
        ``iprint > 100``  print details of every iteration including x and g.
    disp : int, optional
        If zero, then no output.  If a positive number, then this over-rides
        `iprint` (i.e., `iprint` gets the value of `disp`).
    maxfun : int, optional
        Maximum number of function evaluations.
    maxiter : int, optional
        Maximum number of iterations.
    callback : callable, optional
        Called after each iteration, as ``callback(xk)``, where ``xk`` is the
        current parameter vector.
    maxls : int, optional
        Maximum number of line search steps (per iteration). Default is 20.

    Returns
    -------
    x : array_like
        Estimated position of the minimum.
    f : float
        Value of `func` at the minimum.
    d : dict
        Information dictionary.

        * d['warnflag'] is

          - 0 if converged,
          - 1 if too many function evaluations or too many iterations,
          - 2 if stopped for another reason, given in d['task']

        * d['grad'] is the gradient at the minimum (should be 0 ish)
        * d['funcalls'] is the number of function calls made.
        * d['nit'] is the number of iterations.

    See also
    --------
    minimize: Interface to minimization algorithms for multivariate
        functions. See the 'L-BFGS-B' `method` in particular. Note that the
        `ftol` option is made available via that interface, while `factr` is
        provided via this interface, where `factr` is the factor multiplying
        the default machine floating-point precision to arrive at `ftol`:
        ``ftol = factr * numpy.finfo(float).eps``.

    Notes
    -----
    License of L-BFGS-B (FORTRAN code):

    The version included here (in fortran code) is 3.0
    (released April 25, 2011).  It was written by Ciyou Zhu, Richard Byrd,
    and Jorge Nocedal <nocedal@ece.nwu.edu>. It carries the following
    condition for use:

    This software is freely available, but we expect that all publications
    describing work using this software, or all commercial products using it,
    quote at least one of the references given below. This software is released
    under the BSD License.

    References
    ----------
    * R. H. Byrd, P. Lu and J. Nocedal. A Limited Memory Algorithm for Bound
      Constrained Optimization, (1995), SIAM Journal on Scientific and
      Statistical Computing, 16, 5, pp. 1190-1208.
    * C. Zhu, R. H. Byrd and J. Nocedal. L-BFGS-B: Algorithm 778: L-BFGS-B,
      FORTRAN routines for large scale bound constrained optimization (1997),
      ACM Transactions on Mathematical Software, 23, 4, pp. 550 - 560.
    * J.L. Morales and J. Nocedal. L-BFGS-B: Remark on Algorithm 778: L-BFGS-B,
      FORTRAN routines for large scale bound constrained optimization (2011),
      ACM Transactions on Mathematical Software, 38, 1.

    """
    # handle fprime/approx_grad
    if approx_grad:
        fun = func
        jac = None
    elif fprime is None:
        fun = MemoizeJac(func)
        jac = fun.derivative
    else:
        fun = func
        jac = fprime

    # build options
    if disp is None:
        disp = iprint
    opts = {'disp': disp,
            'iprint': iprint,
            'maxcor': m,
            'ftol': factr * np.finfo(float).eps,
            'gtol': pgtol,
            'eps': epsilon,
            'maxfun': maxfun,
            'maxiter': maxiter,
            'callback': callback,
            'maxls': maxls}

    res = _minimize_lbfgsb(fun, x0, args=args, jac=jac, bounds=bounds,
                           **opts)
    d = {'grad': res['jac'],
         'task': res['message'],
         'funcalls': res['nfev'],
         'nit': res['nit'],
         'warnflag': res['status']}
    f = res['fun']
    x = res['x']

    return x, f, d


```

```python
def _minimize_lbfgsb(fun, x0, args=(), jac=None, bounds=None,
                     disp=None, maxcor=10, ftol=2.2204460492503131e-09,
                     gtol=1e-5, eps=1e-8, maxfun=15000, maxiter=15000,
                     iprint=-1, callback=None, maxls=20, **unknown_options):
    """
    Minimize a scalar function of one or more variables using the L-BFGS-B
    algorithm.

    Options
    -------
    disp : bool
       Set to True to print convergence messages.
    maxcor : int
        The maximum number of variable metric corrections used to
        define the limited memory matrix. (The limited memory BFGS
        method does not store the full hessian but uses this many terms
        in an approximation to it.)
    ftol : float
        The iteration stops when ``(f^k -
        f^{k+1})/max{|f^k|,|f^{k+1}|,1} <= ftol``.
    gtol : float
        The iteration will stop when ``max{|proj g_i | i = 1, ..., n}
        <= gtol`` where ``pg_i`` is the i-th component of the
        projected gradient.
    eps : float
        Step size used for numerical approximation of the jacobian.
    disp : int
        Set to True to print convergence messages.
    maxfun : int
        Maximum number of function evaluations.
    maxiter : int
        Maximum number of iterations.
    maxls : int, optional
        Maximum number of line search steps (per iteration). Default is 20.

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

    """
    _check_unknown_options(unknown_options)
    m = maxcor
    epsilon = eps
    pgtol = gtol
    factr = ftol / np.finfo(float).eps

    x0 = asarray(x0).ravel()
    n, = x0.shape

    if bounds is None:
        bounds = [(None, None)] * n
    if len(bounds) != n:
        raise ValueError('length of x0 != length of bounds')
    # unbounded variables must use None, not +-inf, for optimizer to work properly
    bounds = [(None if l == -np.inf else l, None if u == np.inf else u) for l, u in bounds]

    if disp is not None:
        if disp == 0:
            iprint = -1
        else:
            iprint = disp

    n_function_evals, fun = wrap_function(fun, ())
    if jac is None:
        def func_and_grad(x):
            f = fun(x, *args)
            g = _approx_fprime_helper(x, fun, epsilon, args=args, f0=f)
            return f, g
    else:
        def func_and_grad(x):
            f = fun(x, *args)
            g = jac(x, *args)
            return f, g

    nbd = zeros(n, int32)
    low_bnd = zeros(n, float64)
    upper_bnd = zeros(n, float64)
    bounds_map = {(None, None): 0,
                  (1, None): 1,
                  (1, 1): 2,
                  (None, 1): 3}
    for i in range(0, n):
        l, u = bounds[i]
        if l is not None:
            low_bnd[i] = l
            l = 1
        if u is not None:
            upper_bnd[i] = u
            u = 1
        nbd[i] = bounds_map[l, u]

    if not maxls > 0:
        raise ValueError('maxls must be positive.')

    x = array(x0, float64)
    f = array(0.0, float64)
    g = zeros((n,), float64)
    wa = zeros(2*m*n + 5*n + 11*m*m + 8*m, float64)
    iwa = zeros(3*n, int32)
    task = zeros(1, 'S60')
    csave = zeros(1, 'S60')
    lsave = zeros(4, int32)
    isave = zeros(44, int32)
    dsave = zeros(29, float64)

    task[:] = 'START'

    n_iterations = 0

    while 1:
        # x, f, g, wa, iwa, task, csave, lsave, isave, dsave = \
        _lbfgsb.setulb(m, x, low_bnd, upper_bnd, nbd, f, g, factr,
                       pgtol, wa, iwa, task, iprint, csave, lsave,
                       isave, dsave, maxls)
        task_str = task.tostring()
        if task_str.startswith(b'FG'):
            # The minimization routine wants f and g at the current x.
            # Note that interruptions due to maxfun are postponed
            # until the completion of the current minimization iteration.
            # Overwrite f and g:
            f, g = func_and_grad(x)
        elif task_str.startswith(b'NEW_X'):
            # new iteration
            n_iterations += 1
            if callback is not None:
                callback(np.copy(x))

            if n_iterations >= maxiter:
                task[:] = 'STOP: TOTAL NO. of ITERATIONS REACHED LIMIT'
            elif n_function_evals[0] > maxfun:
                task[:] = ('STOP: TOTAL NO. of f AND g EVALUATIONS '
                           'EXCEEDS LIMIT')
        else:
            break

    task_str = task.tostring().strip(b'\x00').strip()
    if task_str.startswith(b'CONV'):
        warnflag = 0
    elif n_function_evals[0] > maxfun or n_iterations >= maxiter:
        warnflag = 1
    else:
        warnflag = 2

    # These two portions of the workspace are described in the mainlb
    # subroutine in lbfgsb.f. See line 363.
    s = wa[0: m*n].reshape(m, n)
    y = wa[m*n: 2*m*n].reshape(m, n)

    # See lbfgsb.f line 160 for this portion of the workspace.
    # isave(31) = the total number of BFGS updates prior the current iteration;
    n_bfgs_updates = isave[30]

    n_corrs = min(n_bfgs_updates, maxcor)
    hess_inv = LbfgsInvHessProduct(s[:n_corrs], y[:n_corrs])

    return OptimizeResult(fun=f, jac=g, nfev=n_function_evals[0],
                          nit=n_iterations, status=warnflag, message=task_str,
                          x=x, success=(warnflag == 0), hess_inv=hess_inv)


class LbfgsInvHessProduct(LinearOperator):
    """Linear operator for the L-BFGS approximate inverse Hessian.

    This operator computes the product of a vector with the approximate inverse
    of the Hessian of the objective function, using the L-BFGS limited
    memory approximation to the inverse Hessian, accumulated during the
    optimization.

    Objects of this class implement the ``scipy.sparse.linalg.LinearOperator``
    interface.

    Parameters
    ----------
    sk : array_like, shape=(n_corr, n)
        Array of `n_corr` most recent updates to the solution vector.
        (See [1]).
    yk : array_like, shape=(n_corr, n)
        Array of `n_corr` most recent updates to the gradient. (See [1]).

    References
    ----------
    .. [1] Nocedal, Jorge. "Updating quasi-Newton matrices with limited
       storage." Mathematics of computation 35.151 (1980): 773-782.

    """
    def __init__(self, sk, yk):
        """Construct the operator."""
        if sk.shape != yk.shape or sk.ndim != 2:
            raise ValueError('sk and yk must have matching shape, (n_corrs, n)')
        n_corrs, n = sk.shape

        super(LbfgsInvHessProduct, self).__init__(
            dtype=np.float64, shape=(n, n))

        self.sk = sk
        self.yk = yk
        self.n_corrs = n_corrs
        self.rho = 1 / np.einsum('ij,ij->i', sk, yk)

    def _matvec(self, x):
        """Efficient matrix-vector multiply with the BFGS matrices.

        This calculation is described in Section (4) of [1].

        Parameters
        ----------
        x : ndarray
            An array with shape (n,) or (n,1).

        Returns
        -------
        y : ndarray
            The matrix-vector product

        """
        s, y, n_corrs, rho = self.sk, self.yk, self.n_corrs, self.rho
        q = np.array(x, dtype=self.dtype, copy=True)
        if q.ndim == 2 and q.shape[1] == 1:
            q = q.reshape(-1)

        alpha = np.zeros(n_corrs)

        for i in range(n_corrs-1, -1, -1):
            alpha[i] = rho[i] * np.dot(s[i], q)
            q = q - alpha[i]*y[i]

        r = q
        for i in range(n_corrs):
            beta = rho[i] * np.dot(y[i], r)
            r = r + s[i] * (alpha[i] - beta)

        return r

    def todense(self):
        """Return a dense array representation of this operator.

        Returns
        -------
        arr : ndarray, shape=(n, n)
            An array with the same shape and containing
            the same data represented by this `LinearOperator`.

        """
        s, y, n_corrs, rho = self.sk, self.yk, self.n_corrs, self.rho
        I = np.eye(*self.shape, dtype=self.dtype)
        Hk = I

        for i in range(n_corrs):
            A1 = I - s[i][:, np.newaxis] * y[i][np.newaxis, :] * rho[i]
            A2 = I - y[i][:, np.newaxis] * s[i][np.newaxis, :] * rho[i]

            Hk = np.dot(A1, np.dot(Hk, A2)) + (rho[i] * s[i][:, np.newaxis] *
                                                        s[i][np.newaxis, :])
        return Hk
```