The Nelder–Mead algorithm

Andreas Sorge

September 3, 2014

The Nelder–Mead algorithm [1] attempts to minimize a goal function $f : \mathbb{R}^n \to \mathbb{R}$ of an unconstrained optimization problem. As it only evaluates function values, but no derivatives, the Nelder–Mead algorithm is a *direct search method* [2]. Although the method generally lacks rigorous convergence properties [3] [4], in practice the first few iterations often yield satisfactory results [5]. Typically, each iteration evaluates the goal function only once or twice [6], which is why the Nelder–Mead algorithm is comparatively fast if goal function evaluation is the computational bottleneck [5].

1 The algorithm

Nelder & Mead [1] refined a simplex method by Spendley et al. [7]. A simplex is the generalization of triangles in $\mathbb{R}^2$ to n dimensions: in $\mathbb{R}^n$, a simplex is the convex hull of $n+1$ vertices $x_0, \ldots, x_n \in \mathbb{R}^n$. Starting with an initial simplex, the algorithm attempts to decrease the function values $f_i := f(x_i)$ at the vertices by a sequence of elementary transformations of the simplex along the local landscape. The algorithm *succeeds* when the simplex is sufficiently small (*domain convergence test*), and/or when the function values f_i are sufficiently close (*function-value convergence test*). The algorithm *fails* when it did not succeed after a given number of iterations or function evaluations. See Singer & Nead [5] and references therein for a complete description of the algorithm and the simplex transformations.

2 Uncertainties in parameter estimation

For parameter estimation, Spendley et al. [7] and Nelder & Mead [1] provide a method to estimate the uncertainties. Fitting a quadratic surface to the vertices and the midpoints of the edges of the final simplex yields an estimate for the variance–covariance matrix. The variance–covariance matrix is $\mathbf{Q}\mathbf{B}^{-1}\mathbf{Q}^T$ as originally given by Nelder & Mead [1], despite the erratum on the original paper. The errors are the square roots of the diagonal terms [8].

3 Implementation

Scientific Python [9] [10] implements the Nelder–Mead method for the scipy.optimize.minimize function. Note that this implementation only returns the vertex with the lowest function value, but not the whole final simplex.

```
In [1]: # ******NOTICE***************
        # optimize.py module by Travis E. Oliphant
        #
        # You may copy and use this module as you see fit with no
        # guarantee implied provided you keep this notice in all copies.
        # *****END NOTICE************

        # numpy
        import numpy
        import scipy.optimize

        from numpy import (
            asfarray
        )
```

```
from scipy.optimize.optimize import (
    _check_unknown_options, wrap_function, _status_message, OptimizeResult
)


def _neldermead_errors(
    sim, fsim, func
):
    # fit quadratic coefficients
    fun = func

    n = len(sim) - 1

    x = 0.5 * (sim[numpy.mgrid[0:6, 0:6]][1] + sim[numpy.mgrid[0:6, 0:6]][0])

    for i in range(n + 1):
        assert(numpy.array_equal(x[i,i], sim[i]))
        for j in range(n + 1):
            assert(numpy.array_equal(x[i,j], 0.5 * (sim[i] + sim[j])))

    y = numpy.nan * numpy.ones(shape=(n + 1, n + 1))
    for i in range(n + 1):
        y[i, i] = fsim[i]
        for j in range(i + 1, n + 1):
            y[i, j] = y[j, i] = fun(x[i, j])

    y0i = y[numpy.mgrid[0:6, 0:6]][0][1:,1:, 0]
    for i in range(n):
        for j in range(n):
            assert y0i[i, j] == y[0, i + 1], (i, j)

    y0j = y[numpy.mgrid[0:6, 0:6]][0][0, 1:, 1:]
    for i in range(n):
        for j in range(n):
            assert y0j[i, j] == y[0, j + 1], (i, j)

    b = 2 * (y[1:, 1:] + y[0, 0] - y0i - y0j)
    for i in range(n):
        assert abs(b[i, i] - 2 * (fsim[i + 1] + fsim[0] - 2 * y[0, i + 1])) < 1e-12
        for j in range(n):
            if i == j:
                continue
            assert abs(b[i, j] - 2 * (y[i + 1, j + 1] + fsim[0] - y[0, i + 1] -
                y[0, j + 1])) < 1e-12

    q = (sim - sim[0])[1:].T
    for i in range(n):
        assert numpy.array_equal(q[:, i], sim[i + 1] - sim[0])

    varco = numpy.dot(q, numpy.dot(numpy.linalg.inv(b), q.T))
    return numpy.sqrt(numpy.diag(varco))

def minimize_neldermead_witherrors(
    fun, x0, args=(), callback=None,
    xtol=1e-4, ftol=1e-4, maxiter=None, maxfev=None,
    disp=False, return_all=False, with_errors=True,
    **unknown_options):
```

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

    Options for the Nelder-Mead algorithm are:
        disp : bool
            Set to True to print convergence messages.
        xtol : float
            Relative error in solution `xopt` acceptable for convergence.
        ftol : float
            Relative error in ``fun(xopt)`` acceptable for convergence.
        maxiter : int
            Maximum number of iterations to perform.
        maxfev : int
            Maximum number of function evaluations to make.

    This function is called by the `minimize` function with
    `method=minimize_neldermead_with_errors`. It is not supposed to be called directly.
    """
    maxfun = maxfev
    retall = return_all

    fcalls, func = wrap_function(fun, args)
    x0 = asfarray(x0).flatten()
    N = len(x0)
    rank = len(x0.shape)
    if not -1 < rank < 2:
        raise ValueError("Initial guess must be a scalar or rank-1 sequence.")
    if maxiter is None:
        maxiter = N * 200
    if maxfun is None:
        maxfun = N * 200

    rho = 1
    chi = 2
    psi = 0.5
    sigma = 0.5
    one2np1 = list(range(1, N + 1))

    if rank == 0:
        sim = numpy.zeros((N + 1,), dtype=x0.dtype)
    else:
        sim = numpy.zeros((N + 1, N), dtype=x0.dtype)
    fsim = numpy.zeros((N + 1,), float)
    sim[0] = x0
    if retall:
        allvecs = [sim[0]]
    fsim[0] = func(x0)
    nonzdelt = 0.05
    zdelt = 0.00025
    for k in range(0, N):
        y = numpy.array(x0, copy=True)
        if y[k] != 0:
            y[k] = (1 + nonzdelt)*y[k]
        else:
            y[k] = zdelt

        sim[k + 1] = y
```

```
        f = func(y)
        fsim[k + 1] = f

    ind = numpy.argsort(fsim)
    fsim = numpy.take(fsim, ind, 0)
    # sort so sim[0,:] has the lowest function value
    sim = numpy.take(sim, ind, 0)

    iterations = 1

    while (fcalls[0] < maxfun and iterations < maxiter):
        if (numpy.max(numpy.ravel(numpy.abs(sim[1:] - sim[0]))) <= xtol and
                numpy.max(numpy.abs(fsim[0] - fsim[1:])) <= ftol):
            break

        xbar = numpy.add.reduce(sim[:-1], 0) / N
        xr = (1 + rho) * xbar - rho * sim[-1]
        fxr = func(xr)
        doshrink = 0

        if fxr < fsim[0]:
            xe = (1 + rho * chi) * xbar - rho * chi * sim[-1]
            fxe = func(xe)

            if fxe < fxr:
                sim[-1] = xe
                fsim[-1] = fxe
            else:
                sim[-1] = xr
                fsim[-1] = fxr
        else:  # fsim[0] <= fxr
            if fxr < fsim[-2]:
                sim[-1] = xr
                fsim[-1] = fxr
            else:  # fxr >= fsim[-2]
                # Perform contraction
                if fxr < fsim[-1]:
                    xc = (1 + psi * rho) * xbar - psi * rho * sim[-1]
                    fxc = func(xc)

                    if fxc <= fxr:
                        sim[-1] = xc
                        fsim[-1] = fxc
                    else:
                        doshrink = 1
                else:
                    # Perform an inside contraction
                    xcc = (1 - psi) * xbar + psi * sim[-1]
                    fxcc = func(xcc)

                    if fxcc < fsim[-1]:
                        sim[-1] = xcc
                        fsim[-1] = fxcc
                    else:
                        doshrink = 1

                if doshrink:
                    for j in one2np1:
```

```
                                sim[j] = sim[0] + sigma * (sim[j] - sim[0])
                                fsim[j] = func(sim[j])

                    ind = numpy.argsort(fsim)
                    sim = numpy.take(sim, ind, 0)
                    fsim = numpy.take(fsim, ind, 0)
                    if callback is not None:
                        callback(sim[0])
                    iterations += 1
                    if retall:
                        allvecs.append(sim[0])

                x = sim[0]
                fval = numpy.min(fsim)
                warnflag = 0
                errors = None

                if fcalls[0] >= maxfun:
                    warnflag = 1
                    msg = _status_message['maxfev']
                    if disp:
                        print('Warning: ' + msg)
                elif iterations >= maxiter:
                    warnflag = 2
                    msg = _status_message['maxiter']
                    if disp:
                        print('Warning: ' + msg)
                else:
                    msg = _status_message['success']
                    errors = _neldermead_errors(sim, fsim, func)
                    if disp:
                        print(msg)
                        print("         Current function value: %f" % fval)
                        print("         Iterations: %d" % iterations)
                        print("         Function evaluations: %d" % fcalls[0])

                result = OptimizeResult(fun=fval, nit=iterations, nfev=fcalls[0],
                                        status=warnflag, success=(warnflag == 0),
                                        message=msg, x=x, errors=errors, sim=sim,
                                        fsim=fsim)

                if retall:
                    result['allvecs'] = allvecs
                return result

        x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
        res = scipy.optimize.minimize(scipy.optimize.rosen, x0, method='Nelder-Mead')
        print(res)

        res_witherrors = scipy.optimize.minimize(
            scipy.optimize.rosen,
            x0,
            method=minimize_neldermead_witherrors
        )
        print(res_witherrors)
```

```
x: array([ 0.99910115,  0.99820923,  0.99646346,  0.99297555,  0.98600385])
     nit: 141
 success: True
```

```
    fun: 6.6174817088845322e-05
message: 'Optimization terminated successfully.'
   nfev: 243
 status: 0
success: True
   fsim: array([  6.61748171e-05,   6.64266969e-05,   6.66640269e-05,
        6.69424827e-05,   6.70671859e-05,   6.70870519e-05])
    sim: array([[ 0.99910115,  0.99820923,  0.99646346,  0.99297555,  0.98600385],
       [ 0.99909442,  0.99820319,  0.99645812,  0.99302291,  0.98608273],
       [ 0.99908478,  0.99820409,  0.9964653 ,  0.99296511,  0.9859391 ],
       [ 0.99909641,  0.99824295,  0.99644354,  0.9929541 ,  0.98596017],
       [ 0.99907389,  0.998211  ,  0.99645267,  0.99294537,  0.98597485],
       [ 0.99907893,  0.99819173,  0.99644522,  0.99298154,  0.986004  ]])
    fun: 6.6174817088845322e-05
    nit: 141
      x: array([ 0.99910115,  0.99820923,  0.99646346,  0.99297555,  0.98600385])
 errors: array([ 0.12236908,  0.22373152,  0.43670037,  0.86737782,  1.72549539])
message: 'Optimization terminated successfully.'
 status: 0
   nfev: 258
```


References

[1] J. A. Nelder and R. Mead, The Computer Journal **7**, 308 (1965).

[2] T. G. Kolda, R. M. Lewis, and V. Torczon, SIAM Review **45**, 385 (2003).

[3] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright, SIAM Journal on Optimization **9**, 112 (1998).

[4] C. J. Price, I. D. Coope, and D. Byatt, *Journal of Optimization Theory and Applications*, **113**, 5 (2002).

[5] S. Singer and J. Nelder, Scholarpedia **4**, 2928+ (2009).

[6] S. Singer and S. Singer, Applied Numerical Analysis & Computational Mathematics **1**, 524 (2004).

[7] W. Spendley, G. R. Hext, and F. R. Himsworth, Technometrics **4**, 441 (1962).

[8] P. R. Bevington and D. K. Robinson, *Data Reduction and Error Analysis for the Physical Sciences*, 3rd ed. (McGraw-Hill, Boston, 2003).

[9] E. Jones, T. Oliphant, and P. Peterson, "SciPy: Open source scientific tools for Python," (2001–), http://www.scipy.org/.

[10] T. E. Oliphant, Computing in Science & Engineering **9**, 10 (2007).