

Minimization

Introduction

Minimization is a problem of finding the minimum (or the maximum) of a given, generally non-linear, real valued function $f(\mathbf{x})$ of an n -dimensional argument $\mathbf{x} \doteq \{x_1, \dots, x_n\}$.

Minimization is a simple case of a more general problem—*optimization*—which includes finding best available values of some objective function within a given domain and subject to given constraints.

Minimization is related to root-finding as at the minimum all partial derivatives of the objective function vanish,

$$\frac{\partial f}{\partial x_i} = 0. \quad (1)$$

One can alternatively solve this system of non-linear equations.

Newton's methods

Newton's method is based on the quadratic approximation of the objective function f in the vicinity of the suspected minimum,

$$f(\mathbf{x} + \Delta\mathbf{x}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^T \Delta\mathbf{x} + \frac{1}{2} \Delta\mathbf{x}^T \mathbf{H}(\mathbf{x}) \Delta\mathbf{x}, \quad (2)$$

where the vector $\nabla f(\mathbf{x})$ is the gradient of the objective function at the point \mathbf{x} ,

$$\nabla f(\mathbf{x}) \doteq \left[\frac{\partial f(\mathbf{x})}{\partial x_i} \right]_{i=1, \dots, n}, \quad (3)$$

and $\mathbf{H}(\mathbf{x})$ is the *Hessian matrix*—a square matrix made of second-order partial derivatives of the objective function at the point \mathbf{x} ,

$$\mathbf{H}(\mathbf{x}) \doteq \left[\frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j} \right]_{i, j \in 1, \dots, n}. \quad (4)$$

The minimum of this quadratic form as function of $\Delta\mathbf{x}$ is found at the point where its gradient with respect to $\Delta\mathbf{x}$ vanishes,

$$\nabla f(\mathbf{x}) + \mathbf{H}(\mathbf{x}) \Delta\mathbf{x} = 0. \quad (5)$$

This gives an approximate step towards the minimum, called the *Newton's step*,

$$\Delta\mathbf{x} = -\mathbf{H}(\mathbf{x})^{-1} \nabla f(\mathbf{x}). \quad (6)$$

The original Newton's method is simply the iteration,

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{H}(\mathbf{x}_k)^{-1} \nabla f(\mathbf{x}_k), \quad (7)$$

where at each iteration the full Newton's step is taken and the Hessian matrix is recalculated. In practice, instead of calculating \mathbf{H}^{-1} one rather solves the linear equation (5).

Usually Newton's method is modified to include a smaller step size, $\lambda \Delta\mathbf{x}$, with $0 < \lambda < 1$. The step-size λ can be found by a backtracking algorithm similar to that in the Newton's method for root-finding. One starts with $\lambda = 1$ and then backtracks, $\lambda \leftarrow \lambda/2$, until the *Armijo condition*,

$$f(\mathbf{x} + \lambda \Delta\mathbf{x}) < f(\mathbf{x}) + \alpha \nabla f(\mathbf{x}) \lambda \Delta\mathbf{x}, \quad (8)$$

is satisfied. The parameter α can be chosen as small as 10^{-4} .

Quasi-Newton methods

Quasi-Newton methods are variations of the Newton's method which attempt to avoid recalculation of the Hessian matrix at each iteration, trying instead certain updates based on the analysis of the gradient vectors. The update $\delta\mathbf{H}$ is usually chosen to satisfy the condition

$$\nabla f(\mathbf{x} + \delta\mathbf{x}) = \nabla f(\mathbf{x}) + (\mathbf{H} + \delta\mathbf{H})\delta\mathbf{x}, \quad (9)$$

called *secant equation*, which is the Taylor expansion of the gradient. This secant condition is underdetermined in more than one dimension as it consists of only n equations for the n^2 unknown elements of the update $\delta\mathbf{H}$. Various quasi-Newton methods use different choices for the form of the solution of the secant equation.

In quasi-Newton methods one often starts with a unity matrix as the zeroth approximation for the Hessian matrix and then applies the updates. In this case one can actually use the inverse Hessian matrix—thus avoiding the need to solve the linear equation at each iteration—and apply the updates directly to the inverse matrix.

Broyden's update

The Broyden's update is chosen in the form

$$\delta H = c\delta\mathbf{x}^T. \quad (10)$$

Substituting this ansatz into (9) leads to the Broyden's update,

$$H \rightarrow H + \frac{\mathbf{d} - H\delta\mathbf{x}}{\delta\mathbf{x}^T\delta\mathbf{x}}\delta\mathbf{x}^T, \quad (11)$$

where $\mathbf{d} \doteq \nabla f(\mathbf{x} + \delta\mathbf{x}) - \nabla f(\mathbf{x})$. The update for the inverse matrix is given as

$$H^{-1} \rightarrow H^{-1} + \frac{(\delta\mathbf{x} - H^{-1}\mathbf{d})\mathbf{d}^T H^{-1}}{\mathbf{d}^T H^{-1}\delta\mathbf{x}}. \quad (12)$$

SR1 update

The symmetric-rank-1 update (SR1) is chosen in the form

$$\delta H = ss^T, \quad (13)$$

where the vector s is found from the condition (9). The resulting update is

$$H \rightarrow H + \frac{(\mathbf{d} - H\delta\mathbf{x})(\mathbf{d} - H\delta\mathbf{x})^T}{(\mathbf{d} - H\delta\mathbf{x})^T\delta\mathbf{x}}. \quad (14)$$

And for the inverse matrix

$$H^{-1} \rightarrow H^{-1} + \frac{(\delta\mathbf{x} - H^{-1}\mathbf{d})(\delta\mathbf{x} - H^{-1}\mathbf{d})^T}{(\delta\mathbf{x} - H^{-1}\mathbf{d})^T\mathbf{d}}. \quad (15)$$

Downhill simplex method

The *downhill simplex method* (also called Nelder-Mead method or amoeba method) is a commonly used minimization algorithm, where the minimum of a function in an n -dimensional space is found by transforming a simplex—a polytope with $n+1$ vertexes—according to the function values at the vertexes, moving it downhill until it converges towards the minimum.

The advantages of the downhill simplex method is its stability and the lack of use of derivatives. However, the convergence is relatively slow as compared to Newton's methods.

In order to introduce the algorithm we need the following definitions:

- Simplex: a figure (polytope) represented by $n+1$ points, called vertexes, $\{\mathbf{p}_1, \dots, \mathbf{p}_{n+1}\}$ (where each point \mathbf{p}_k is an n -dimensional vector).

Method	δH	δH^{-1}
DFP		
BFGS		
Broyden	$\frac{\mathbf{d}-H\delta\mathbf{x}}{\delta\mathbf{x}^T\delta\mathbf{x}}\delta\mathbf{x}^T$	$\frac{(\delta\mathbf{x}-H^{-1}\mathbf{d})\mathbf{d}^TH^{-1}}{\mathbf{d}^TH^{-1}\delta\mathbf{x}}$
SR1	$\frac{(\mathbf{d}-H\delta\mathbf{x})(\mathbf{d}-H\delta\mathbf{x})^T}{(\mathbf{d}-H\delta\mathbf{x})^T\delta\mathbf{x}}$	$\frac{(\delta\mathbf{x}-H^{-1}\mathbf{d})(\delta\mathbf{x}-H^{-1}\mathbf{d})^T}{(\delta\mathbf{x}-H^{-1}\mathbf{d})^T\mathbf{d}}$

Table 1: Downhill simplex (Nelder-Mead) algorithm

<pre> REPEAT : find highest, lowest, and centroid points of the simplex try reflection IF f(reflected) < f(lowest) : try expansion IF f(expanded) < f(reflected) : accept expansion ELSE : accept reflection ELSE : IF f(reflected) < f(highest) : accept reflection ELSE : try contraction IF f(contracted) < f(highest) : accept contraction ELSE : do reduction UNTIL converged (e.g. size(simplex)<tolerance) </pre>

- Highest point: the vertex, \mathbf{p}_{hi} , with the largest value of the function: $f(\mathbf{p}_{\text{hi}}) = \max_{(k)} f(\mathbf{p}_k)$.
- Lowest point: the vertex, \mathbf{p}_{lo} , with the smallest value of the function: $f(\mathbf{p}_{\text{lo}}) = \min_{(k)} f(\mathbf{p}_k)$.
- Centroid: the center of gravity of all points, except for the highest: $\mathbf{p}_{\text{ce}} = \frac{1}{n} \sum_{(k \neq \text{hi})} \mathbf{p}_k$

The simplex is moved downhill by a combination of the following elementary operations:

1. Reflection: the highest point is reflected against the centroid, $\mathbf{p}_{\text{hi}} \rightarrow \mathbf{p}_{\text{re}} = \mathbf{p}_{\text{ce}} + (\mathbf{p}_{\text{ce}} - \mathbf{p}_{\text{hi}})$.
2. Expansion: the highest point reflects and then doubles its distance from the centroid, $\mathbf{p}_{\text{hi}} \rightarrow \mathbf{p}_{\text{ex}} = \mathbf{p}_{\text{ce}} + 2(\mathbf{p}_{\text{ce}} - \mathbf{p}_{\text{hi}})$.
3. Contraction: the highest point halves its distance from the centroid, $\mathbf{p}_{\text{hi}} \rightarrow \mathbf{p}_{\text{co}} = \mathbf{p}_{\text{ce}} + \frac{1}{2}(\mathbf{p}_{\text{hi}} - \mathbf{p}_{\text{ce}})$.
4. Reduction: all points, except for the lowest, move towards the lowest points halving the distance. $\mathbf{p}_{k \neq \text{lo}} \rightarrow \frac{1}{2}(\mathbf{p}_k + \mathbf{p}_{\text{lo}})$.

Finally, Table 1 shows one possible algorithm for the downhill simplex method.

```

#include<armadillo>
#include<vector>
#include<functional>
using namespace std;
using namespace arma;

struct amoeba{
    int d,hi,lo; vector<vec> p; vec p_ce; vec values;
    std::function<double(vec)> f;
    amoeba(function<double(vec)> fun, vector<vec> simplex);
    void update(); double size(); void downhill(double simplex_size_goal);};

void amoeba::update(){
    hi=0; for(int i=1;i<d+1;i++) if(values[i]>values[hi]) hi=i;
    lo=0; for(int i=1;i<d+1;i++) if(values[i]<values[lo]) lo=i;
    p_ce = zeros<vec>(d);
    for(int i=0;i<d+1;i++)if(i!=hi) p_ce += p[i];
    p_ce /= d;}

amoeba::amoeba(function<double(vec)> fun, vector<vec> simplex)
:d(simplex.size()-1), f(fun), values(zeros<vec>(d+1)), p_ce(zeros<vec>(d)){
    for(int i=0;i<d+1;i++) p.push_back(simplex[i]);
    for(int i=0;i<d+1;i++) values[i] = f(p[i]);
    update(); }

double amoeba::size(){
    double s=0;
    for(int i=0;i<d+1;i++)if(i!=lo){
        double n = norm(p[i]-p[lo],2); if(n>s) s=n;}
    return s; }

void amoeba::downhill(double simplex_size_goal){
    while(size(>simplex_size_goal){
        vec p_re = p_ce + (p_ce - p[hi]); // try reflection
        double f_re = f(p_re);
        if(f_re < values[lo])
        {
            vec p_ex = p_ce + 2*(p_ce - p[hi]); // try expansion
            double f_ex = f(p_ex);
            if(f_ex<f_re)
            {
                values[hi]=f_ex; p[hi]=p_ex; // accept expansion
                update(); continue;
            }
        }
        if(f_re < values[hi])
        {
            values[hi]=f_re; p[hi]=p_re; // accept reflection
            update(); continue;
        }
        vec p_co = p_ce+0.5*(p[hi]-p_ce); // try contraction
        double f_co = f(p_co);
        if(f_co < values[hi])
        {
            values[hi]=f_co; p[hi]=p_co; // accept contraction
            update(); continue;
        }
        for(int i=0;i<d+1;i++)if(i!=lo)
        {
            p[i]=0.5*(p[i]+p[lo]); // do reduction
            values[i]=f(p[i]);
        }
        update();continue;
    } // end while
} // end downhill

```