Minimization

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

Minimization is a simple case of a more general poblem—optimization—which includes finding best available values of some objective function within a given domain.

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

$$\frac{\partial f(\mathbf{x})}{\partial x_i} = 0 \,. \tag{1}$$

Thus one can alternatively solve this system of non-linear equations.

Quasi-Newton's methods

These methods are based on the quadratic approxiamtion of the objective function f in the vicinity of the suspected minimum,

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

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

$$\nabla f(\mathbf{x}) \doteq \left[\frac{\partial f(\mathbf{x})}{\partial x_i}\right] \,, \tag{3}$$

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

$$H(\mathbf{x}) \doteq \left[\frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}\right] \,. \tag{4}$$

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,

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

This gives an approximate step towards the minimum, called the Newton step,

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

Newton's method

The Newton's method is simply the iteration,

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

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

Modified Newton's method

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 algoritm similar to that in the Newton's method for root-finding. One starts with $\lambda = 1$ and than backtracks, $\lambda \leftarrow \lambda/2$, until the *Armijo condition*,

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

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

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

Quasi-Newton methods

Quasi-Newton methods attempt to avoid the calculation of the Hessian matrix at each iteration, trying instead certain updates based on the analysis of the gradient vectors. The update δH is usually chosen to satisfy the condition

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

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 δH. 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.

Broydens's update The Broyden's update is chosen in the form

$$\delta H = \mathbf{c}\delta\mathbf{x}^T\,. \tag{10}$$

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

$$H \to H + \frac{\mathbf{d}-H\delta\mathbf{x}}{\delta\mathbf{x}^T\delta\mathbf{x}}\delta\mathbf{x}^T\,, \tag{11}$$

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

$$H^{-1} \to H^{-1} + \frac{(\delta\mathbf{x}-H^{-1}\mathbf{d})\mathbf{d}^T H^{-1}}{\mathbf{d}^T H^{-1}\delta\mathbf{x}}\,. \tag{12}$$

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

$$\delta H = ss^T\,, \tag{13}$$

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

$$H \to H + \frac{(\mathbf{d}-H\delta\mathbf{x})(\mathbf{d}-H\delta\mathbf{x})^T}{(\mathbf{d}-H\delta\mathbf{x})^T\delta\mathbf{x}}\,. \tag{14}$$

And for the inverse matrix

$$H^{-1} \to H^{-1} + \frac{(\delta\mathbf{x}-H^{-1}\mathbf{d})(\delta\mathbf{x}-H^{-1}\mathbf{d})^T}{(\delta\mathbf{x}-H^{-1}\mathbf{d})^T\mathbf{d}}\,. \tag{15}$$

Table 1: Downhill simplex (Nelder-Mead) algorithm

```
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)
```

Downhill simplex method

The *downhill simplex method* (also called Nelder-Mead method or amoeba method) is a commonnly 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 realtively 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, $\{\mathbf{p}_1, \ldots, \mathbf{p}_{n+1}\}$ (where each point $\mathbf{p}_k$ is an n-dimensional vector).
- Highest point: the vertex, $\mathbf{p}_{\rm hi}$, with the largest value of the function: $f(\mathbf{p}_{\rm hi}) = \max_{(k)} f(\mathbf{p}_k)$.
- Lowest point: the vertex, $\mathbf{p}_{\rm lo}$, with the smallest value of the function: $f(\mathbf{p}_{\rm lo}) = \min_{(k)} f(\mathbf{p}_k)$.
- Centroid: the center of gravity of all points, except for the highest: $\mathbf{p}_{\rm ce} = \frac{1}{n} \sum_{(k \neq {\rm hi})} \mathbf{p}_k$

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

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

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 value;
   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(value[i]>value[hi]) hi=i;
   lo=0; for(int i=1;i<d+1;i++) if(value[i]<value[hi]) 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), value(zeros<vec>(d)), 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++) value[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 < value[lo])
         {
         vec p_ex = p_ce + 2*(p_ce - p[hi]); // try expansion
         double f_ex = f(p_ex);
         if(f_ex<f_re)
            {
            value[hi]=f_ex; p[hi]=p_ex;// accept expansion
            update(); continue;
            }
         }
      if(f_re < value[hi])
         {
         value[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 < value[hi])
         {
         value[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
         value[i]=f(p[i]);
         }
      update();continue;
      }// end while
   }// end downhill
```