1 Minimization and optimization

1.1 Introduction

Minimization (maximization) is a problem of finding the minimum (maximum) of a given — generally non-linear — real valued function $\phi(\mathbf{x})$ (often called the *objective function*) of an n-dimensional argument $\mathbf{x} \doteq \{x_1, \ldots, x_n\}$.

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

Minimization is not unrelated to root-finding: at the minimum all partial derivatives of the objective function vanish,

$$\frac{\partial \phi}{\partial x_i} = 0 \Big|_{i=1,\ldots,n} , \tag{1}$$

and one can alternatively solve this system of (non-linear) equations.

1.2 Local minimization

1.2.1 Newton's methods

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

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

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

$$\nabla\phi(\mathbf{x}) \doteq \left\{ \frac{\partial \phi(\mathbf{x})}{\partial x_i} \right\}_{i=1,\ldots,n} , \tag{3}$$

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

$$\mathbf{H}(\mathbf{x}) \doteq \left\{ \frac{\partial^2 \phi(\mathbf{x})}{\partial x_i \partial x_j} \right\}_{i,j \in 1,\ldots,n} . \tag{4}$$

The minimum of the quadratic form (2), as function of $\Delta\mathbf{x}$, is found at the point where its gradient with respect to $\Delta\mathbf{x}$ vanishes,

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

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

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

The original Newton's method is simply the iteration,

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

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 the Newton's method is modified to take a smaller step $\mathbf{s}$,

$$\mathbf{s} = \lambda\Delta\mathbf{x}, \tag{8}$$

with $0 < \lambda < 1$. The factor 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*,

$$\phi(\mathbf{x}+\mathbf{s}) < \phi(\mathbf{x}) + \alpha\mathbf{s}^{\mathsf{T}}\nabla\phi(\mathbf{x})\,, \tag{9}$$

is satisfied (or the minimal λ is reached, in which case the step is taken unconditionally). The parameter α can be chosen as small as 10^{-4}.

1.2.2 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

$$\nabla\phi(\mathbf{x}+\mathbf{s}) = \nabla\phi(\mathbf{x}) + (\mathbf{H}+\delta\mathbf{H})\mathbf{s}\,, \tag{10}$$

called *secant equation*, which is the Taylor expansion of the gradient.

The secant equation is under-determined 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 practice one typically uses the inverse Hessian matrix (often—but not always—denoted as $\mathbf{B}$) and applies the updates directly to the inverse matrix thus avoiding the need to solve the linear equation (5) at each iteration.

For the inverse Hessian matrix the secant equation (10) reads

$$(\mathbf{B}+\delta\mathbf{B})\mathbf{y} = \mathbf{s}\,, \tag{11}$$

or, in short,

$$\delta\mathbf{B}\mathbf{y} = \mathbf{u}\,, \tag{12}$$

where $\mathbf{B} \doteq \mathbf{H}^{-1}$, $\mathbf{y} \doteq \nabla\phi(\mathbf{x}+\mathbf{s}) - \nabla\phi(\mathbf{x})$, and $\mathbf{u} \doteq \mathbf{s} - \mathbf{B}\mathbf{y}$.

One usually starts with the identity matrix as the zeroth approximation for the inverse Hessian matrix and then applies the updates.

If the minimal λ is reached during the bactracking line-search—which might be a signal of lost precision in the approximate (inverse) Hessian matrix—it is advisable to reset the current inverse Hessian matrix to identity matrix.

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

$$\delta \mathbf{B} = \mathbf{c}\mathbf{s}^{\mathsf{T}} \, . \tag{13}$$

where the vector $\mathbf{c}$ is found from the condition (12),

$$\mathbf{c} = \frac{\mathbf{u}}{\mathbf{s}^{\mathsf{T}}\mathbf{y}} \, . \tag{14}$$

Sometimes the dot-product $\mathbf{s}^{\mathsf{T}}\mathbf{y}$ becomes very small or even zero which results in serious numerical difficulties. One can avoid this by only performing update if the condition $|\mathbf{s}^{\mathsf{T}}\mathbf{y}| > \epsilon$ is satisfied where ϵ is a small number, say 10^{-6}.

Symmetric Broyden's update The Broyden's update (13) is not symmetric (while the Hessian matrix should be) which is an obvious drawback. Therefore a beter approximation might be the symmetric Broyden's update,

$$\delta \mathbf{B} = \mathbf{a}\mathbf{s}^{\mathsf{T}} + \mathbf{s}\mathbf{a}^{\mathsf{T}} \, . \tag{15}$$

The vector $\mathbf{a}$ is again found from the condition (12),

$$\mathbf{a} = \frac{\mathbf{u} - \gamma \mathbf{s}}{\mathbf{s}^{\mathsf{T}}\mathbf{y}} \, , \tag{16}$$

where $\gamma = (\mathbf{u}^T\mathbf{y})/(2\mathbf{s}^{\mathsf{T}}\mathbf{y})$.

Again one only performs the update if $|\mathbf{s}^{\mathsf{T}}\mathbf{y}| > \epsilon$.

A C-implementation of the algorithm is listed in Table 1.

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

$$\delta \mathbf{B} = \mathbf{v}\mathbf{v}^{\mathsf{T}} \, , \tag{17}$$

where the vector $\mathbf{v}$ is again found from the condition (10), which gives

$$\delta \mathbf{B} = \frac{\mathbf{u}\mathbf{u}^T}{\mathbf{u}^T\mathbf{y}} \, . \tag{18}$$

Again, one only performs the update if denominator is not too small, that is, $|\mathbf{u}^{\mathsf{T}}\mathbf{y}| > \epsilon$.

Other popular updates The wikipedia article "Quasi-Newton method" list several other popular updates.

Table 1: C-implementation of a quasi-Neuton method

```
#include<stdio.h>
#include<math.h>
#include<float.h>
#define FOR(i) for(int i=0;i<dim;i++)
#define SET_IDENTITY(B) FOR(i)FOR(j)B[i][j]=0;FOR(i)B[i][i]=1;
static const double DELTA=sqrt(DBL_EPSILON);

void numeric_gradient
(int dim,double F(int dim,double* x),double*x,double*gradient){
 double fx=F(dim,x);
 FOR(i){
  double dx=DELTA, xi=x[i];
  if(fabs(xi)>sqrt(DELTA)) dx=fabs(xi)*DELTA;
  x[i]+=dx; gradient[i]=(F(dim,x)-fx)/dx; x[i]=xi; } }

void broyden(int dim,double F(int,double*), double*x, double acc) {
 int nsteps=0;
 double B[dim][dim]; SET_IDENTITY(B);
 double gx[dim]; numeric_gradient(dim,F,x,gx);
 double fx=F(dim,x),fz;
 while(nsteps<1000){
  double norm2gx=0; FOR(i)norm2gx+=gx[i]*gx[i];
  if(norm2gx<acc*acc) break;
  nsteps++;
  double step[dim],z[dim];
  FOR(i){ step[i]=0; FOR(j) step[i]-=B[i][j]*gx[j]; }
  double lambda=1;
  while(1){
   FOR(i)z[i]=x[i]+step[i]; fz=F(dim,z);
   double sTg=0; FOR(i)sTg+=step[i]*gx[i];
   if(fz<fx+0.01*sTg) break;
   if(lambda<DELTA){ SET_IDENTITY(B); break; }
   lambda/=2; FOR(i)step[i]/=2;
  }
  double gz[dim]; numeric_gradient(dim,F,z,gz);
  double y[dim]; FOR(i)y[i]=gz[i]-gx[i];
  double u[dim]; FOR(i){ u[i]=step[i]; FOR(j) u[i] -= B[i][j]*y[j]; }
  double sTy=0; FOR(i)sTy += step[i]*y[i];
  if(fabs(sTy)>DELTA){
   //FOR(i)FOR(j)B[i][j]+=u[i]*step[j]/sTy; // Broyden update
   double uTy=0; FOR(i) uTy += u[i]*y[i];
   double gamma=uTy/2/sTy; // symmetric Broyden update
   double a[dim]; FOR(i) a[i] = u[i]-gamma*step[i];
   FOR(i)FOR(j)B[i][j] += a[i]*step[j]/sTy;
   FOR(i)FOR(j)B[i][j] += step[i]*a[j]/sTy;
  }
  FOR(i)x[i]=z[i]; FOR(i)gx[i]=gz[i];
  fx=fz;
 } return; }
```

1.2.3 Downhill simplex method

The *downhill simplex method* [1] (also called "Nelder-Mead" or "amoeba") 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, $\mathbf{p}_{\text{hi}}$, with the highest value of the function: $\phi(\mathbf{p}_{\text{hi}}) = \max_k \phi(\mathbf{p}_k)$.
- Lowest point: the vertex, $\mathbf{p}_{\text{lo}}$, with the lowest value of the function: $\phi(\mathbf{p}_{\text{lo}}) = \min_k \phi(\mathbf{p}_k)$.
- 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. $\mathbf{p}_{k \neq \text{lo}} \rightarrow \frac{1}{2}(\mathbf{p}_k + \mathbf{p}_{\text{lo}})$.

Table 2 shows one possible algorithm for the downhill simplex method, and a C-implementation of simplex operations and the amoeba algorithm can be bound in Table 5 and Table 1.3.3.

1.3 Global optimization

Global optimization is the problem of locating (a good approximation to) the global minimum of a given objective function in a search space large enough to prohibit exhaustive enumeration.

Table 2: Downhill simplex (Nelder-Mead) algorithm

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

When only a small sub-space of the search space can be realistically sampled the stochastic methods usually come to the fore.

A good local minimizer converges to the nearest local miminum relatively fast, so one possible global minimizer can be constructed by simply starting the local miminizer from different random starting points.

In the following several popular global minimization algorithms are shortly described.

1.3.1 Simulated annealing

Simulated annealing is a stochastic metaheuristic algorithm for global minimization. The name and inspiration come from annealing—heating up and cooling slowly—in material science. The slow cooling allows a piece of material to reach a state with "lowest energy".

The objective function in the space of states is interpreted as some sort of potential energy and the states—the points in the search space—are interpeted as physical states of some physical system. The system attempts to make transitions from its current state to some randomly sampled nearest states with the goal to eventually reach the state with minimal energy – the global minimum.

The system is attached to a thermal resevoir with certain temperature. Each time the energy of the system is measured the reservoir supplies it with a random amount of thermal energy sampled from the Boltzmann distribution,

$$P(E) = Te^{-E/T} \,. \tag{19}$$

Table 3: Simulated annealing algorithm

```
state ← start_state
T ← start_temperature
energy ← E(state)
REPEAT :
  new_state ← neighbour(state)
  new_energy ← E(new_state)
  IF new_energy < energy :
    state ← new_state
    energy ← new_energy
  ELSE :
    do with probability exp(−(new_energy − energy)/T) :
      state ← new_state
      energy ← new_energy
  reduce_temperature_according_to_schedule(T)
UNTIL terminated
```

If the temperature equals zero the system can only make transitions to the neighboring states with lower potential energy. In this case the algorithm turns merely into a local minimizer with random sampling.

If temperature is finite the system is able to climb up the ridges of the potential energy—about as high as the current temperature—and thus escape from local minima and hopefully eventually reach the global minimum.

One typically starts the simulation with some finite temperature on the order of the height of the typical hills of the potential energy surface, letting the system to wander almost unhindered around the landscape with a good chance to locate if not the best then at least a good enough minimum. The temperature is then slowly reduced following some annealing schedule which may be supplied by the user but must end with $T = 0$ towards the end of the alloted time budget.

Table 3 lists one possible variant of the algorithm.

The function `neigbour(state)` should return a randomly chosen neighbour of the given state.

Downhill simplex method can incorporate simulated annealing by adding the stochastic thermal energy to the values of the objective function at the vertices.

1.3.2 Quantum annealing

Quantum annealing is a general global minimization algorithm which—like simulated annealing—also allows the search path to escape from local minima. However instead of the thermal jumps over the potential barriers quantum annealing allows the system to tunnel through the barriers.

In its simple incarnation the quantum annealing algorithm allows the system to attempt transitions not only to the nearest states but also to distant states within

Table 4: Quantum annealing algorithm

```
state ← start_state
energy ← E(state)
R ← start_radius
REPEAT :
  new_state ← random_neighbour_within_radius(state,R)
  new_energy ← E(new_state)
  IF new_energy < energy :
    state ← new_state
    energy ← new_energy
  reduce_radius_according_to_schedule(R)
UNTIL terminated
```

certain "tunneling distance" from the current state. The transition is accepted only if it reduces the potential energy of the system.

At the beginning of the minimization procedure the tunnelling distance is large—on the order of the size of the region where the global minimum is suspected to be localed—allowing the system to explore the region. The tunneling distance is then slowly reduced according to a schedule such that by the end of the alloted time the tunnelling distance reduces to zero at which point the system hopefully is in the state with minimal energy.

1.3.3 Evolutionary algorithms

Unlike annealing algorithms, which follow the motion of only one point in the search space, the evolutionary algorithms typically follow a set of points called a population of individuals. A bit like the downhill simplex method which follows the motion of a set of points – the simplex.

The population evolves towards more fit individuals where fitness is understood in the sence of minimizing the objective functions. The parameters of the individuals (for example, the coordinates of the points in multi-dimentional minimization of a continuous objective function) are called genes.

The algorithm proceeds iteratively with the population in each iteration called a generation. In each generation the fitness of each individual—typically, the value of the objective function—is evaluated and the new generation is generated stochastically from the gene pool of the current generation through crossovers and mutations such that the genes of more fit individuals have a better chance of propagating into the next generation.

Each new individual in the next generation is produced from a pair of "parent" individuals of the current generation. The use of two "parents" is biologically inspired, in practice more than two "parents" can be used as well. The parents for a new individual are selected from the individuals of the current generation through a fitness based stochastic process where fitter individuals are more likely

to be selected.

The "child" individual shares many characteristics of its "parents". In the simplest case the "child" may get its genes by simply averaging the genes of its parents. Then a certain amount of mutations—random changes in the genes—are added to the "child's" genes.

Generation of "children" continues until the population of the new generation reaches the appropriate size after which the iteration repeats itself.

The algorithm is terminated when the fitness level of the population is deemed sufficient or when the allocated budget is exhausted.

References

[1] J.A.Nelder and R.Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

Table 5: C implementation of simplex operations

```
void reflection
 (double* highest,double* centroid,int dim,double* reflected){
  for(int i=0;i<dim;i++) reflected[i]=2*centroid[i]-highest[i];
}
void expansion
  (double* highest, double* centroid, int dim, double* expanded) {
  for(int i=0;i<dim;i++) expanded[i]=3*centroid[i]-2*highest[i];
}
void contraction
  (double* highest, double* centroid, int dim, double* contracted){
  for(int i=0;i<dim;i++)
    contracted[i]=0.5*centroid[i]+0.5*highest[i];
}
void reduction( double** simplex, int dim, int lo){
  for(int k=0;k<dim+1;k++) if(k!=lo) for(int i=0;i<dim;i++)
      simplex[k][i]=0.5*(simplex[k][i]+simplex[lo][i]);
}
double distance(double* a, double* b, int dim){
  double s=0; for(int i=0;i<dim;i++) s+=pow(b[i]-a[i],2);
  return sqrt(s);
}
double size(double** simplex,int dim){
  double s=0; for(int k=1;k<dim+1;k++){
    double dist=distance(simplex[0],simplex[k],dim);
    if(dist>s) s=dist; }
  return s;
}
```

```
void simplex_update(double** simplex, double* f_values, int d,
int* hi, int* lo, double* centroid) {
 *hi=0; *lo=0; double highest=f_values[0], lowest =f_values[0];
 for(int k=1;k<d+1;k++) {
  double next=f_values[k];
  if(next>highest){highest=next;*hi=k;}
  if(next<lowest) {lowest=next; *lo=k;} }
 for(int i=0;i<d;i++) {
  double s=0;for(int k=0;k<d+1;k++)if(k!=*hi)s+=simplex[k][i];
  centroid[i]=s/d; }
}
void simplex_initiate(
double fun(double*), double** simplex, double* f_values, int d,
int* hi, int* lo, double* centroid) {
  for(int k=0;k<d+1;k++) f_values[k]=fun(simplex[k]);
  simplex_update(simplex,f_values,d,hi,lo,centroid);
}
```

Table 6: C implementation of downhill simplex algorithm

```
int downhill_simplex(
  double F(double*),double**simplex,int d,double simplex_size_goal)
{
int hi,lo,k=0; double centroid[d], F_value[d+1], p1[d], p2[d];
simplex_initiate(F,simplex,F_value,d,&hi,&lo,centroid);
while(size(simplex,d)>simplex_size_goal){
  simplex_update(simplex,F_value,d,&hi,&lo,centroid);
  reflection(simplex[hi],centroid,d,p1); double f_re=F(p1);
  if(f_re<F_value[lo]){ // reflection looks good: try expansion
    expansion(simplex[hi],centroid,d,p2); double f_ex=F(p2);
    if(f_ex<f_re){ // accept expansion
      for(int i=0;i<d;i++)simplex[hi][i]=p2[i]; F_value[hi]=f_ex;}
    else{ // reject expansion and accept reflection
      for(int i=0;i<d;i++)simplex[hi][i]=p1[i]; F_value[hi]=f_re;}}
  else{ // reflection wasn't good
    if(f_re<F_value[hi]){ // ok, accept reflection
      for(int i=0;i<d;i++)simplex[hi][i]=p1[i]; F_value[hi]=f_re;}
    else{ // try contraction
      contraction(simplex[hi],centroid,d,p1); double f_co=F(p1);
      if(f_co<F_value[hi]){ // accept contraction
        for(int i=0;i<d;i++)simplex[hi][i]=p1[i]; F_value[hi]=f_co;}
      else{ // do reduction
        reduction(simplex,d,lo);
        simplex_initiate(F,simplex,F_value,d,&hi,&lo,centroid);}}}
  k++;} return k;
}
```