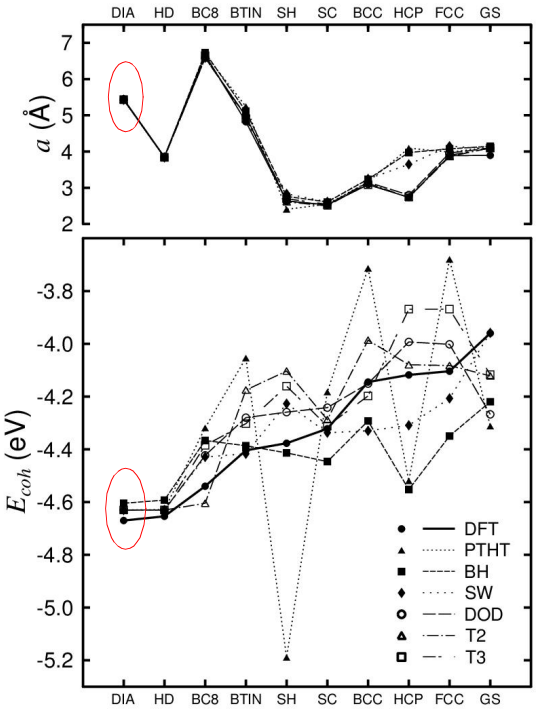
Minimization of functions

• We have a number of atoms at positions $\{\mathbf{r}_i\}$, $i = 1, 2, \ldots, N$, and they interact via potential energy $V(\{\mathbf{r}_i\})$.

- Find the configuration $\{\mathbf{r}_i\}$ that minimizes the potential energy $V(\{\mathbf{r}_i\})$ → find the equilibrium structure of the material.
  (Well equilibrium in zero Kelvin.)

Si: E_{pot} for different structures by different pot. models



• Function minimization in general:

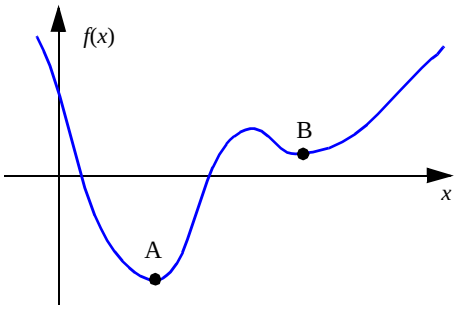
Find the argument of the function f that gives the smallest possible function value.

- f may have many variables → multidimensional minimization
- Function maximization by minimization of $-f$.
- Function values may be continuous of discrete.
- Constrained or unconstrained minimization.
- Minimization = optimization.

Minimization of functions

• Usually one is interested in the **global minimum** (A).

- There may also be **local minima** (B).
- Local minima are easy to find.
- Global minimum is harder.
  - One way is vary the starting point (see below) and pick the lowest minimum.
  - Simulated annealing and genetic algorithms may give you the global minimum.



• In this section we introduce a couple of algorithms for unconstrained minimization continuous functions.

• What method to choose depends often on whether you can or want to compute the derivatives of the function.

• In general, minimization methods are iterative:

1) First guess a starting point.
2) Use the algorithm to decide where to proceed and how long.

Minimization of functions: golden section search

• Remember bisection rule in root finding:

1) Isolate the root to interval $[a, b]$.
2) Split the interval into two.
3) Take as a new interval the one that contains the root.

• Golden section search (**GSS**) is application of the very same idea to function minimization.

- To isolate a minimum we need three points a, b, c so that

$$a < b < c$$
$$f(b) < f(a)$$
$$f(b) < f(c)$$

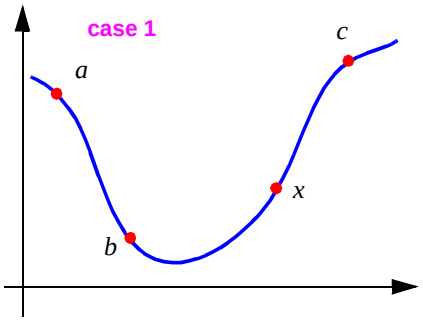
- Choose a new point x either from subinterval $[a, b]$ or $[b, c]$.

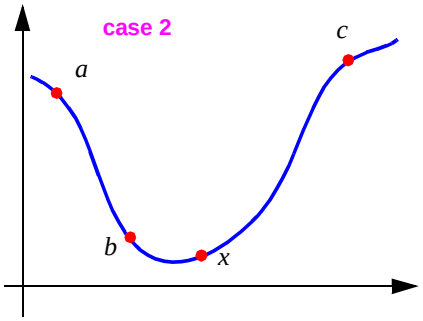
- Assume the new point is in $[b, c]$ (see the figure).

- If $f(b) < f(x)$ $\rightarrow$ the new triplet is (a, b, x) (**case 1**)
  otherwise it is (b, x, c) (**case 2**)

- Always choose the triplet whose middle point has the lowest function value.

- This iteration is continued until the required accuracy is reached.
  $\rightarrow$ the interval $[a, c]$ is small enough.





Minimization of functions: golden section search

- How small is small enough?

- If the minimum is at $x = b$ we might think that it can be localized into interval

$$[(1-\varepsilon)b, (1+\varepsilon)b]^{1},$$

where ε is the machine epsilon.

- Near the minimum the function has the form (1st derivative almost zero)

$$f(x) \approx f(b) + \frac{1}{2}f''(b)(x-b)^2$$

- The 2nd term on the right hand size is negligible compared with the 1st term when

$$\frac{1}{2}(x-b)^2 f''(b) < |f(b)|\varepsilon$$

Minimum: $f''(b) > 0$

- This can be massaged into form

$$\frac{|x-b|}{|b|} < \sqrt{\varepsilon}\sqrt{\frac{2|f(b)|}{b^2 f''(b)}}$$

- Assuming that the term under square root is of the order of unity we get the result that

$$\frac{|x-b|}{|b|} < \sqrt{\varepsilon}$$

- I.e. the (relative) accuracy we can reach is only $\sqrt{\varepsilon}$.

Float: $\varepsilon = 1.192\times10^{-7}$ $\sqrt{\varepsilon} = 3.453\times10^{-4}$

Double: $\varepsilon = 2.220\times10^{-16}$ $\sqrt{\varepsilon} = 1.490\times10^{-8}$

1. Or $[b, (1+\varepsilon)b]$?

Minimization of functions: golden section search

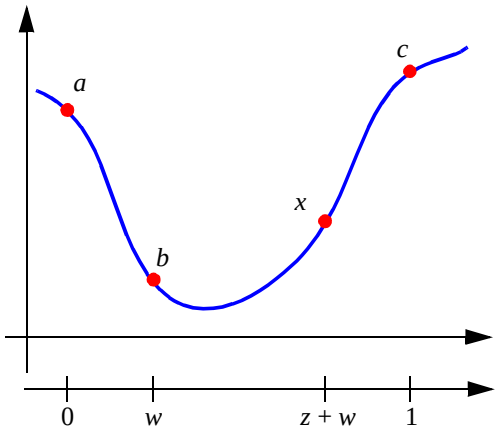
- What is then the optimum way to choose the new point x when a, b, c are known?

- Let's denote with w the relative position of b at the interval $[a, c]$:

$$w = \frac{b-a}{c-a}, \quad 1-w = \frac{c-b}{c-a}$$

- Similarly the relative position of the new point x from b is z:

$$z = \frac{x-b}{c-a}.$$



- Now the length of the next interval is (in relative units) either $z + w$ ($[a, x]$) or $1 - w$ ($[b, c]$).

- Optimal (on the average) choice is to set these two equal:

$$z + w = 1 - w \Rightarrow z = 1 - 2w.$$

- One can see that the new point x is symmetrical with the point b:

$$|b-a| = |x-c|.$$

- Also we can see that x is in the larger one of the subintervals $[a, b]$ and $[b, c]$.

Minimization of functions: golden section search

- The point x can be calculated in the following way:

- Assume that we have already used the GSS algorithm.
  - This means that the distance of x from b compared to interval $[b, c]$[1] = the distance of b from a compared to interval $[a, c]$:

$$\frac{x-b}{c-b} = \frac{b-a}{c-a}$$

  - Or in relative units

$$\frac{z}{1-w} = \frac{w}{1}$$

  - Combining this with our optimal choice we get

$$\begin{cases} z = 1-2w \\ \dfrac{z}{1-w} = w \end{cases}$$

$$\rightarrow \quad w^2 - 3w + 1 = 0$$

$$\rightarrow \quad w = \frac{3-\sqrt{5}}{2} \approx 0.381966$$

1. Assumed to be the larger subinterval.

Minimization of functions: golden section search

- Thus, the optimum triplet of points (a, b, c) is such that b is at the distance $0.381966(c-a)$ from one end of interval $[a, c]$ and at the distance $0.618034(c-a)$ from the other.

- The algorithm is the following

1. We have points a, b, and c from the previous iteration.
2. Choose the new point x such that it is at the relative distance of 0.381966 from b to the direction of the larger subinterval (either $[a, b]$ or $[b, c]$).
3. The new triplet is the one that has the lower value at the midpoint.

- At each iteration step the interval decreases by a factor 0.618034.

- Convergence is linear; i.e. sufficiently near to the minimum x^* the errors $\varepsilon_k = x_k - x^*$ in the position of the minimum x_k at two consecutive iterations are related as

$$|\varepsilon_{k+1}| \leq C|\varepsilon_k|,$$

where C is a constant that depends on the function being miminized.

Minimization of functions: golden section search

- Below an example of an implementation (modified from NR routine)

```
#include <math.h>
#define R 0.6180339887498949 /* (sqrt(5)-1)/2 */
#define C (1.0-R)
#define SHFT2(a,b,c) (a)=(b);(b)=(c)
#define SHFT3(a,b,c,d) (a)=(b);(b)=(c);(c)=(d)

double golden(double ax, double bx, double cx,
              double (*f)(double),
             double tol, double *xmin, int *j, int maxiter)
{
  double f1,f2,x0,x1,x2,x3;
  int i;
  double fi;
  fi=0.0;
  x0=ax;
  x3=cx;
  if (fabs(cx-bx) > fabs(bx-ax)) {
    x1=bx;
    x2=bx+C*(cx-bx);
  } else {
    x2=bx;
    x1=bx-C*(bx-ax);
  }
  f1=(*f)(x1);
  f2=(*f)(x2);
  i=1;
  while (fabs(x3-x0) > tol*(fabs(x1)+fabs(x2))) {
    fi=0.05*i;
    i++;
    if (f2 < f1) {
      SHFT3(x0,x1,x2,R*x1+C*x3);
      SHFT2(f1,f2,(*f)(x2));
    } else {
      SHFT3(x3,x2,x1,R*x2+C*x0);
      SHFT2(f2,f1,(*f)(x1));
    }
    if (i>=maxiter) {
      if (f1 < f2) {
        *xmin=x1;
        *j=i;
        return f1;
      } else {
        *j=i;
        *xmin=x2;
        return f2;
      }
    }
  }
  if (f1 < f2) {
    *xmin=x1;
    *j=i;
    return f1;
  } else {
    *xmin=x2;
    *j=i;
    return f2;
  }
}
```

Minimization of functions: golden section search

- And the main program

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

double golden(double ax, double bx, double cx, double (*f)(double),
        double tol, double *xmin, int *iter, int maxiter);
double func(double x) {return j0(x);}

int main(int argc, char **argv)
{
  int i,maxiter,iter;
  double ax,bx,cx,xmin,gold,bren,tol;
  if (argc!=6) {
    fprintf(stderr,"Usage: %s tol a b c maxiter\n",argv[0]);
    return (1);
  }
  tol=atof(*++argv);
  ax=atof(*++argv);
  bx=atof(*++argv);
  cx=atof(*++argv);
  maxiter=atoi(*++argv);
  gold=golden(ax,bx,cx,func,tol,&xmin,&iter, maxiter);
  printf("Golden: %12.6g %8d %18.12g %18.12g\n",tol,iter,xmin,gold);
  return 0;
}
```

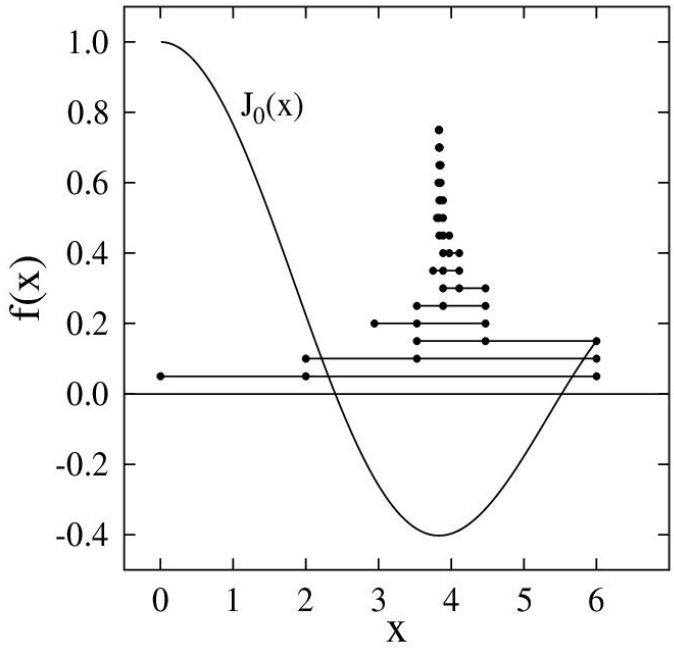
- Function `j0(x)` is the Bessel function $J_0(x)$; it is included (at least) in Linux and HP Tru64 Unix environments.
- In GSL the function it is `gsl_sf_bessel_j0 (double x).`
- In SLATEC: `besj0(x)`

Minimization of functions: golden section search

- Compilation and run:

```
golden> gcc -o goldenmain goldenmain.c golden.c -lm
golden> ./goldenmain 1e-6 0 1 6 1000
Golden:        1e-06       30      3.83170618864    -0.402759395703
```

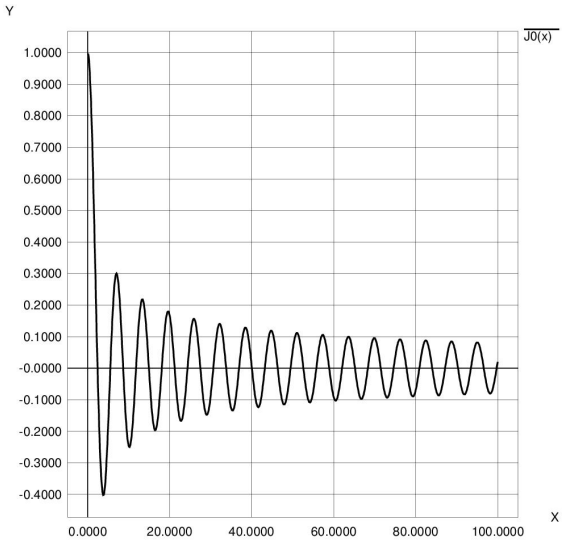
- Iteration graphically:



Minimization of functions: golden section search

- Note that the Bessel function $J_0(x)$ has many minima and where we end up depends on the initial triplet of points:

```
golden> ./goldenmain 1e-8 0 1 2 1000 1
Golden:        1e-08       39        1.99999998855885019       0.223890785739630616
golden> ./goldenmain 1e-8 10 11 12 1000 1
Golden:        1e-08       35        10.1734681153456474      -0.249704877057843139
golden> ./goldenmain 1e-8 5 6 7 1000 1
Golden:        1e-08       36        5.00000002995331894      -0.177596761502255868
```



Minimization of functions: parabolic interpolation

• As we have seen near the minimum the function can be approximated as a parabola

$$f(x) \approx f(b) + \frac{1}{2} f''(b)(x-b)^2$$

- This can directly be used in searching the minimum.

- Assume again that we have three points a, b, and c $(a < b < c)$.

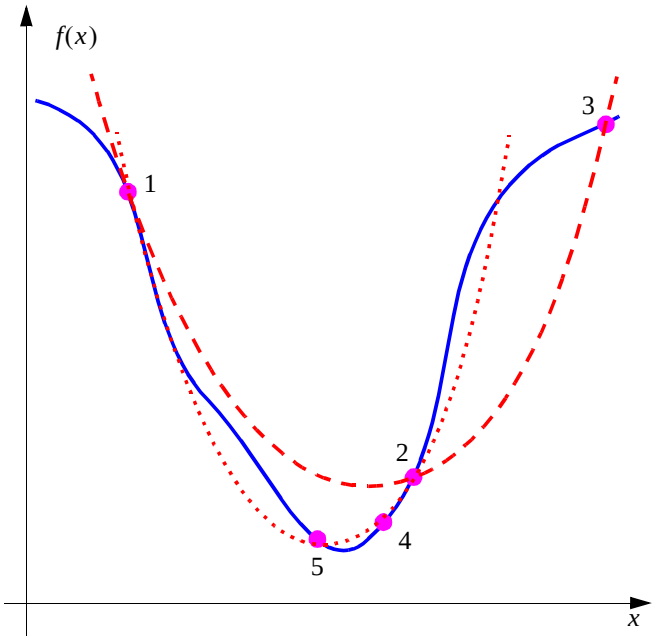
- Let's draw a parabola through them and compute its minimum;

$$x = b - \frac{1}{2}\frac{(b-a)^2[f(b)-f(c)] - (b-c)^2[f(b)-f(a)]}{(b-a)[f(b)-f(c)] - (b-c)[f(b)-f(a)]}$$

- This algorithm does not work if the points are collinear.

- Moreover, it does not distinguish minima from maxima.

- Dashed line through points $1, 2, 3$ is the parabola. It has minimum at point 4. $\rightarrow$ The new triplet is $1, 4, 2$.
- Dotted line is the parabola through this new triplet. Its minimum is at point 5.



Minimization of functions: parabolic interpolation

• As in the case of root finding combining different methods is a good way to obtain fast but robust algorithms.

- One used in minimization is **Brent's method**.

- Normally use parabolic interpolation.

  - If the convergence
    1) is not fast enough or
    2) the new trial minimum is outside the bracketing limits

    switch to golden section rule.

Minimization of functions: parabolic interpolation

- The algorithm is implemented in the GNU Scientific Library (**GSL**).
- Example code (from the GSL info documentation.)

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_min.h>

double fn1 (double x, void * params) {
  return cos(x) + 1.0;}

int main (void)
{
  int status;
  int iter = 0, max_iter = 100;
  const gsl_min_fminimizer_type *T;
  gsl_min_fminimizer *s;
  double m = 2.0, m_expected = M_PI;
  double a = 0.0, b = 6.0;
  gsl_function F;

  F.function = &fn1;
  F.params = 0;
  T = gsl_min_fminimizer_brent;
  s = gsl_min_fminimizer_alloc(T);
  gsl_min_fminimizer_set(s,&F,m,a,b);

  printf ("using %s method\n",
       gsl_min_fminimizer_name(s));

  printf ("%5s [%9s, %9s] %9s %10s %9s\n",
    "iter", "lower", "upper", "min",
    "err", "err(est)");
  printf ("%5d [%.7f, %.7f] %.7f %+.7f %.7f\n",
    iter,a,b,m,m-m_expected,b-a);

  do
    {
      iter++;
      status = gsl_min_fminimizer_iterate(s);
      m = gsl_min_fminimizer_x_minimum(s);
      a = gsl_min_fminimizer_x_lower(s);
      b = gsl_min_fminimizer_x_upper(s);
      status=
         gsl_min_test_interval(a,b,0.001,0.0);
      if (status == GSL_SUCCESS)
            printf ("Converged:\n");
      printf ("%5d [%.7f, %.7f] %.7f %.7f %+.7f %.7f\n",
        iter, a, b,m, m_expected, m - m_expected, b - a);
    }
  while (status == GSL_CONTINUE && iter < max_iter);
  return status;
}
```

Minimization of functions: parabolic intepolation

- Compilation and run:

```
minimization> gcc -o brent_gsl brent_gsl.c -lm -lgsl -lgslcblas
minimization> ./brent_gsl
using brent method
 iter [    lower,     upper]       min        err  err(est)
    0 [0.0000000, 6.0000000] 2.0000000 -1.1415927 6.0000000
    1 [2.0000000, 6.0000000] 3.5278640 3.1415927 +0.3862713 4.0000000
    2 [2.0000000, 3.5278640] 3.1748217 3.1415927 +0.0332290 1.5278640
    3 [2.0000000, 3.1748217] 3.1264576 3.1415927 -0.0151351 1.1748217
    4 [3.1264576, 3.1748217] 3.1414743 3.1415927 -0.0001183 0.0483641
    5 [3.1414743, 3.1748217] 3.1415930 3.1415927 +0.0000004 0.0333474
Converged:
    6 [3.1414743, 3.1415930] 3.1415927 3.1415927 +0.0000000 0.0001187
```


Minimization of functions: Brent's method

• Another GSL example:

```
int main (int argc, char **argv)
{
  int status;
  int iter, max_iter, ialg;
  const gsl_min_fminimizer_type *T;
  gsl_min_fminimizer *s;
  double a,b,c,tol,f;
  gsl_function F;
  struct func_params p;

  if (argc!=8) {
    fprintf(stderr,"Usage: %s tol a b c maxiter ifunc ialg\n",
argv[0]);
    return (1);
  }
  tol=atof(*++argv);
  a=atof(*++argv);
  b=atof(*++argv);
  c=atof(*++argv);
  max_iter=atoi(*++argv);
  p.ifunc=atoi(*++argv);
  ialg=atoi(*++argv);

  switch (ialg) {
  case 1: T=gsl_min_fminimizer_goldensection; break;
  case 2: T=gsl_min_fminimizer_brent; break;
  default: T=gsl_min_fminimizer_goldensection; break;
  }

  F.function = &func;
  F.params = &p;
  s = gsl_min_fminimizer_alloc(T);
  gsl_min_fminimizer_set(s,&F,b,a,c);

  iter=0;

  printf("# using %s method\n",
     gsl_min_fminimizer_name(s));
  printf("#  iter       lower              upper");
  printf("               min            err(est)\n");
  printf("%5d   %18.10e, %18.10e  %18.10e %18.10e\n",
     iter,a,c,b,c-a);

  do
    {
      iter++;
      status = gsl_min_fminimizer_iterate(s);
      b = gsl_min_fminimizer_x_minimum(s);
      a = gsl_min_fminimizer_x_lower(s);
      c = gsl_min_fminimizer_x_upper(s);
      status=gsl_min_test_interval(a,c,tol,0.0);
      if (status == GSL_SUCCESS)
             printf ("# Converged:\n");
      printf ("%5d   %18.10e, %18.10e
               %18.10e %18.10e\n",iter,a,c,b,c-a);
    }
  while (status==GSL_CONTINUE && iter<max_iter);
  return status;
}
```

```
struct func_params {int ifunc;};
double func(double x, void * p) {
  struct func_params
       *params=(struct func_params *)p;
  double ff;
  switch (params->ifunc) {
  case 1: ff=j0(x); break;
  case 2: ff=sin(-x)*exp(-x*x); break;
  case 3: ff=sin(x); break;
  case 4: ff=cos(x) + 1.0; break;
  default: ff=0.0; break;
  }
  return ff;}
```

Minimization of functions: Newton's method

• The methods described so far only use the function values for minimization.

• If the derivative of the function is available one can utilize it in minimization.

- The basic idea is to approximate the function by a polynomial (again!) and find its minimum.

- Because a line doesn't have a minimum we have to use a parabola.

- Let's write the function as a Taylor series around the result x_k at the iteration step k:

$$f(x_k + p) = f(x_k) + pf'(x_k) + \frac{1}{2}p^2 f''(x_k) + \dots$$

- Now minimize this

$$\begin{aligned} f(x^*) &= \min_x[f(x)] \\ &= \min_p[f(x_k + p)] \\ &= \min_p\left[f(x_k) + pf'(x_k) + \frac{1}{2}p^2 f''(x_k) + \dots\right] \\ &\approx \min_p\left[f(x_k) + pf'(x_k) + \frac{1}{2}p^2 f''(x_k)\right] \end{aligned}$$

Minimization of functions: Newton's method

- The minimum of the parabola is obtained by setting its derivative with respect to p zero

$$p = -\frac{f'(x_k)}{f''(x_k)} .$$

- Thus the algorithm is simple: at every iteration step update the position of the minimum by

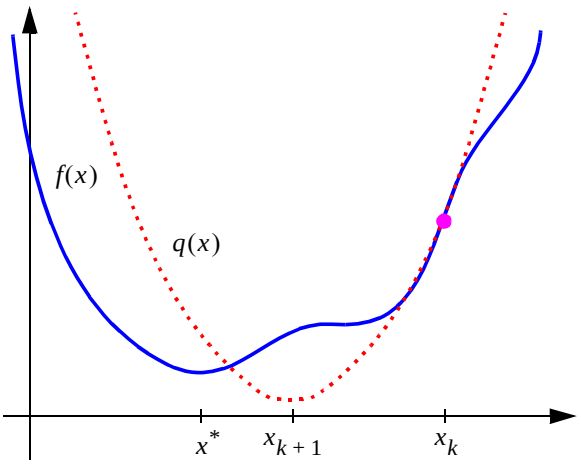
$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} .$$

- Note that this is exactly the Newton's method for root finding; now for function $f'(x)$.

- Graphically the method can be perceived as follows:

  - The interpolating parabola $q(x)$ fulfills the following conditions:

$$\begin{cases} q(x_k) = f(x_k) \\ q'(x_k) = f'(x_k) \\ q''(x_k) = f''(x_k) \end{cases}$$



Minimization of functions: Newton's method

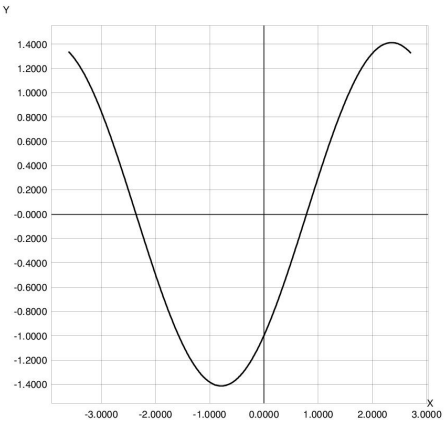
- As we remember from root finding the degree of convergence of Newton's method is 2:

$$\left|x_{k+1} - x^*\right| \le C\left|x_k - x^*\right|^2$$

- Let's take an example

$$f(x) = \sin x - \cos x$$

start of iteration: $x_0 = -0.5$



- Derivatives are

$$\begin{cases} f'(x) = \cos x + \sin x \\ f''(x) = -\sin x + \cos x \end{cases}$$

- Exact result is $x^* = -\frac{\pi}{4} \approx -0.78539816$

- The first iteration step:

$$f'(x_0) = \cos(-0.5) + \sin(-0.5) = 0.39815702$$

$$f''(x_0) = -\sin(-0.5) + \cos(-0.5) = 1.35700810$$

$$p = -\frac{f'(x_0)}{f''(x_0)} = -\frac{0.39815702}{1.35700810} = -0.29340799$$

$$x_1 = x_0 + p = -0.5 - 0.29340799 = -0.79340799$$

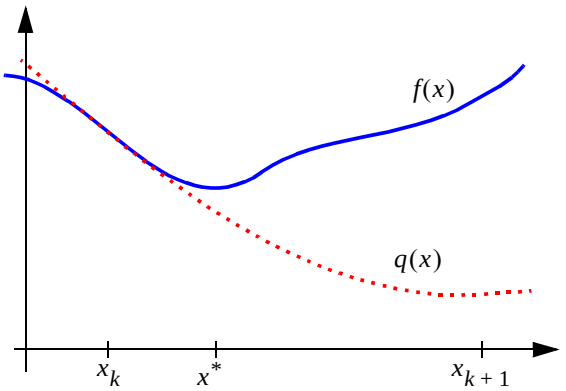
Minimization of functions: Newton's method

- The results of the three first steps show the fast convergence:

k	x_k	$\left\|x_k - x^*\right\|$
0	–0.50000000	2.8×10^{-1}
1	–0.79340799	8.0×10^{-3}
2	–0.78539799	1.7×10^{-7}
3	–0.78539816	1.3×10^{-17}

- As we have stated before, Newton's method has its bad sides:

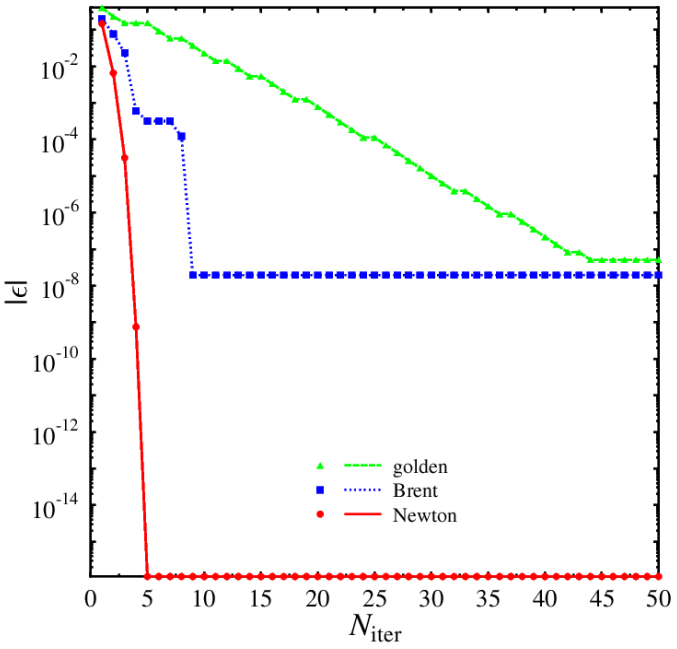
1. If the parabola is not a good approximation for the function it is not guaranteed that the new point nearer to the minimum than the old point.
2. Newton's method can also converge to a maximum.
3. One must be able to calculated the first two derivatives of the function.

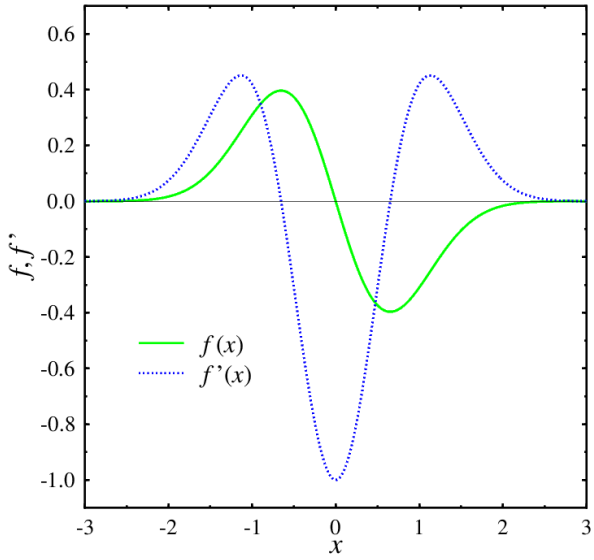


- As in the case of root finding the best strategy is to combine Newton's method with golden section search or use the information on the derivative in some other way.

Minimization of functions: Newton's method

- Comparison of the GSS, Brent and Newton's methods: Plotted is the error which in the case of GSS and Brent is $|\varepsilon| = |a - c|$ and for Newton's method $|\varepsilon| = |x_i - x_{i-1}|$.
- The function to be minimized is $f(x) = \sin(-x)e^{-x^2}$.





Minimization of functions: multidimensional minimization

• Now we have to minimize the function

$$f(x_1, x_2, \ldots, x_N) \equiv f(\mathbf{x})$$

or to find the vector $\mathbf{x}^*$ that satisfies

$$f(\mathbf{x}^*) \leq f(\mathbf{x})$$
$$\forall \mathbf{x} \in \mathbf{R}^N$$

- The most algorithms that do this work in the following way:
  1. Choose the intial point (or points).
  2. Choose the direction $\mathbf{d}_k$ where the minimum is searched.
  3. Find the step length $\lambda_k > 0$ such that $f(\mathbf{x}_k + \lambda_k \mathbf{d}_k)$ is minimized.
     This is so the called line minimization.
     1D methods described above can be used to accomplish this.
  4. If the minimum of $f(\mathbf{x})$ has not bee found go to step 2.

- Many methods utilize the Taylor series of the function near the iteration point:

$$f(\mathbf{x} + \mathbf{h}) = f(\mathbf{x}) + \sum_{i=1}^{N} \frac{\partial f(\mathbf{x})}{\partial x_i} h_i + \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} h_i \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j} h_j + \ldots \quad .$$

$$= f(\mathbf{x}) + [\nabla f(\mathbf{x})]^T \cdot \mathbf{h} + \frac{1}{2} \mathbf{h}^T \mathbf{H}(\mathbf{x}) \mathbf{h} + \ldots$$

- Here

$$\mathbf{H}(\mathbf{x}), \quad [\mathbf{H}(\mathbf{x})]_{ij} = \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}$$

is so called **Hessian matrix** (after Ludwig Otto Hesse, a German mathematician from the 19th century.).

Minimization of functions: multidimensional minimization

• Note that in multidimensional minimization bracketing the minimum is not necessarily possible.

• Convergence criterion is not as simple as in 1D case (the bracketing interval $[a, b]$ is small enough).

- Commonly one may use two criteria:

1. Change in the iterated vector is sufficiently small, i.e. $|\mathbf{x}_k - \mathbf{x}_{k-1}| < \varepsilon_1$, where k is the iteration step.
2. Change in the function value is small enough: $|f(\mathbf{x}_k) - f(\mathbf{x}_{k-1})| < \varepsilon_2$.

• In the following we present a few algorithms that are used for multidimensional minimization.

- One point distinguishing these methods from each other is the whether they use the Hessian matrix of the function. (I.e. use of the second derivative.)

- In large minimization problems constructing the Hessian matrix is prohibitively expensive (in terms of CPU time and memory).

- Example: potential energy minimization of atomic systems: N may be as large as 10^5 $\rightarrow$ Hessian has 10^{10} elements. (Well, in most cases this matrix is very sparse. However, computing the second derivative consumes a lot of CPU time.)

Minimization of functions: downhill simplex method

• This method is a simple but slowly converging algorithm that only uses the function values, not its derivatives.

- It is initialized by creating a geometrical figure *simplex* in N dimensions consisting of $N + 1$ points and their interconnecting line segments and polygonal faces.

- In 2D the simplex is a triangle and in 3D a tetrahedron.

- The simplex is nondegenerate i.e. it encloses a finite inner volume.

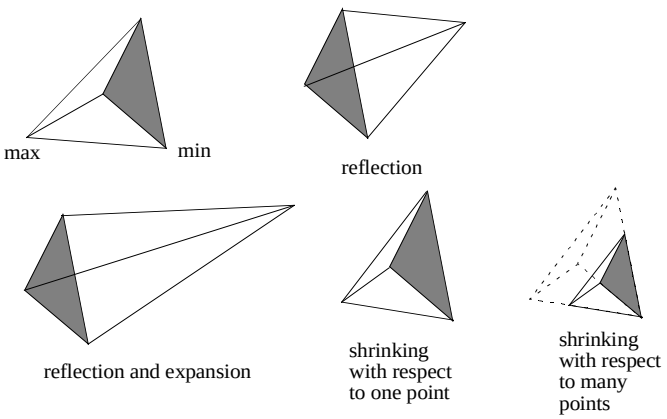
- One way to generate the initial simplex is

$$\mathbf{P}_i = \mathbf{P}_0 + \lambda \mathbf{e}_i ,$$

where $\mathbf{e}_i$ are the N unit vectors.

- The algorithm uses the following moves:

- The method is slowly converging but can also applied to not so nice functions.



Minimization of functions: downhill simplex method

- From *Numeeriset menetelmät käytännössä*, J. Haataja *et al.*, CSC, 1999:

Polytooppi sisältää $n+1$ kärkipistettä $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{n+1})$. Merkitsemme pienintä ja suurinta funktion arvoa polytoopin kärkipisteissä $f_l = \min_i f(\mathbf{x}_i)$ ja $f_h = \max_i f(\mathbf{x}_i)$. Olkoon lisäksi $f_i = f(\mathbf{x}_i)$.

Laskemme polytoopin n parhaan kärjen painopisteen $\mathbf{x}_c = \frac{1}{n}\sum_{i \neq h} \mathbf{x}_i$. Peilauskerroin on $\alpha = 1 > 0$, laajennuskerroin $\gamma = 2 > 1$ ja pienennyskerroin $\beta = 0.5 \in [0, 1]$. Kuva 10.11 havainnollistaa polytooppihaussa käytettyjä skaalaus- ja peilausoperaatioita.

Algoritmi 10.8.4 (Polytooppihaku)

1: Peilaa huonoin kärki $\mathbf{x}_h$ vastapäätä olevan sivun suhteen:

$$\mathbf{x}_r = (1+\alpha)\,\mathbf{x}_c - \alpha \mathbf{x}_h\,.$$

Jos $f_r > f_i$ kaikilla $i \neq h$, mene vaiheeseen 3. Jos taas $f_r \in [f_l, f_h]$, aseta $\mathbf{x}_h = \mathbf{x}_r$ ja toista vaihe 1 saadulle uudelle polytoopille. Muuten jatka vaiheesta 2.

2: Jos $f_r < f_l$, suurenna polytooppia löydettyyn suuntaan:

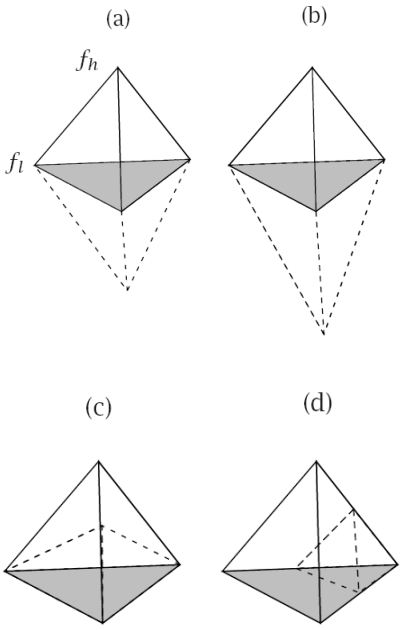
$$\mathbf{x}_e = \gamma \mathbf{x}_r + (1-\gamma)\,\mathbf{x}_c\,.$$

Jos $f_e < f_l$, aseta $\mathbf{x}_l = \mathbf{x}_e$, muuten $\mathbf{x}_h = \mathbf{x}_r$ (polytoopin suurentaminen ei kannata). Jatka vaiheesta 1.

3: Jos $f_r < f_h$, aseta $\mathbf{x}_h = \mathbf{x}_r$. Pienennä polytooppia:

$$\mathbf{x}_s = \beta \mathbf{x}_h + (1-\beta)\,\mathbf{x}_c\,.$$

Jos $f_s < \min\{f_h, f_r\}$, aseta $\mathbf{x}_h = \mathbf{x}_s$. Muussa tapauksessa aseta $\mathbf{x}_i = (\mathbf{x}_i + \mathbf{x}_l)/2$ eli siis puolita polytoopin sivut. Mene vaiheeseen 1.



Minimization of functions: steepest descent

• We have a starting point $\mathbf{P}$ in a N dimensional space.

- We proceed from this point to direction $\mathbf{n}$.

- Using minimization algorithms for 1D functions we can minimize function $f(\mathbf{x})$ in this direction.

- Many different multidimensional minimization algorithms can be constructed by choosing different ways determine the direction $\mathbf{n}$.

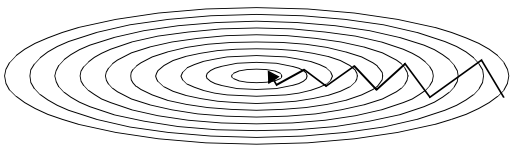
- The **steepest descent** method is maybe the most obvious one: direction is the in which the function decreases the most (locally), i.e. the opposite direction to the gradient $\mathbf{n} = -\nabla f(\mathbf{x}_k)$.

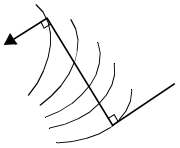
- Algorithm is the following:

  1. Find the step size $\lambda_k > 0$ that minimizes $f(\mathbf{y}_k)$, where $\mathbf{y}_k = \mathbf{x}_k + \lambda_k(-\nabla f(\mathbf{x}_k))$.
  2. Set $\mathbf{x}_{k+1} = \mathbf{y}_k$. Go to step 1 if the required accuracy is not reached.

- This method converges linearly and in some cases may be really inefficient.

  - In narrow valleys the path of the iteration is a zig-zag line.





Minimization of functions: Newton's method

• Assume that our function f is quadratic:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b}^T\mathbf{x} + c\,.$$

- We can readily obtain the gradient of the function

$$\nabla f(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}\,.$$

and the Hessian matrix

$$\mathbf{H}(\mathbf{x}) = \nabla^2 f(\mathbf{x}) = \mathbf{A}\,.$$

- The gradient of the function at point $\mathbf{x} + \mathbf{s}$ is

$$\begin{aligned}\nabla f(\mathbf{x}+\mathbf{s}) &= \mathbf{A}(\mathbf{x}+\mathbf{s}) + \mathbf{b} \\ &= (\mathbf{A}\mathbf{x} + \mathbf{b}) + \mathbf{A}\mathbf{s} \\ &= \nabla f(\mathbf{x}) + \mathbf{H}(\mathbf{x})\mathbf{s}\end{aligned}\,.$$

- From the equation

$$\nabla f(\mathbf{x}+\mathbf{s}) = 0$$

we obtain the condition for the step that minimizes the quadratic function

$$\mathbf{H}(\mathbf{x})\mathbf{s} = -\nabla f(\mathbf{x})\,.$$

Minimization of functions: Newton's method

- Newton's iteration works also for other than quadratic functions if we expand the function as a Taylor series around the iteration point:

$$f(\mathbf{x}+\mathbf{d}) \approx f(\mathbf{x}) + [\nabla f(\mathbf{x})]^T\mathbf{d} + \frac{1}{2}\mathbf{d}^T\mathbf{H}(\mathbf{x})\mathbf{d}\,.$$

- At every iteration step the following is done:

1. Solve $\mathbf{s}_k$ from equation
   $$\mathbf{H}(\mathbf{x}_k)\mathbf{s}_k = -\nabla f(\mathbf{x}_k)$$
2. Update the iteration point:
   $$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$$

- As an example take function

$$f(\mathbf{x}) = \frac{1}{2}x_1^2 + \frac{9}{2}x_2^2$$

- Hessian matrix is

$$\mathbf{H}(\mathbf{x}) = \begin{bmatrix}1 & 0\\ 0 & 9\end{bmatrix}$$

and gradient

$$\nabla f(\mathbf{x}) = \begin{bmatrix}x_1\\ 9x_2\end{bmatrix}\,.$$

Minimization of functions: Newton's method

- Because this function is quadratic Newton's method finds its minimum in one iteration.

- Initial point is $\mathbf{x}_1 = (9, 1)$

- The equation $\mathbf{H}(\mathbf{x})\mathbf{s} = -\nabla f(\mathbf{x})$ takes the form

$$\begin{bmatrix} 1 & 0 \\ 0 & 9 \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} = - \begin{bmatrix} 9 \\ 9 \end{bmatrix}$$

which has the solution

$$\mathbf{s} = (-9, -1).$$

- This gives the minimum

$$\mathbf{x}_2 = (0, 0).$$

- Newton's method converges quadratically.

Minimization of functions: quasi-Newton's methods

• Newton's method does not necessarily always bring us nearer to the minimum.

- In order to approach the minimum the function value must decrease in the iteration step:

$$\mathbf{s}_k^T \nabla f(\mathbf{x}_k) < 0$$

- Or expressed in terms of the Hessian matrix:

$$\mathbf{s}_k^T \mathbf{H}(\mathbf{x}_k)\mathbf{s}_k > 0$$

(This means that the Hessian matrix must be *positive definite*.)

- When far from the minimum there are no guarantees that the Hessian is positive definite.
- In the worst case the iteration step might take us farther from the minimum.

- In the quasi-Newton method the Hessian matrix is not calculated explicitly.

- Using some initial guess for the Hessian it is update so that it approaches the exact Hessian during the simulation.

$$\lim_{i \to \infty} \mathbf{H}_i = \mathbf{H}$$

- A good initial guess is e.g. unit matrix $\mathbf{H}_1 = \mathbf{1}$.

- Different quasi-Newton algorithms differ on the method of updating the Hessian.

Minimization of functions: quasi-Newton's methods

- Below we present two ways to update the Hessian.

- When far from the minimum the algorithms guarantee that the Hessian is positive definite.
- When we are near the minimum the Hessian approaches the real values of Hessian and we obtain quadratic convergence.

- Because Hessian is never calculated explicitly it is sufficient to use its inverse $\mathbf{H}^{-1}$ in all computations.

- Remember that $\mathbf{H}(\mathbf{x})\mathbf{s} = -\nabla f(\mathbf{x})$. From this we get

$$\mathbf{x}_{i+1} - \mathbf{x}_i = \mathbf{H}^{-1}[\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_i)]$$

- This is the exact Hessian. It is natural to assume that updating our approximation is of similar form:

$$\mathbf{x}_{i+1} - \mathbf{x}_i = \mathbf{H}_{i+1}^{-1}[\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_i)].$$

- We can also think that the update consists of expressions including terms

$\mathbf{x}_{i+1} - \mathbf{x}_i$ and
$\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_i)$.

Minimization of functions: quasi-Newton's methods

- In the so called Davidon-Fletcher-Powell (**DFP**) algorithm update is done as

$$\mathbf{H}_{i+1}^{-1} = \mathbf{H}_i^{-1} + \frac{(\mathbf{x}_{i+1} - \mathbf{x}_i) \bullet (\mathbf{x}_{i+1} - \mathbf{x}_i)}{(\mathbf{x}_{i+1} - \mathbf{x}_i)^T [\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_i)]} - \frac{\{\mathbf{H}_i^{-1}[\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_i)]\} \bullet \{\mathbf{H}_i^{-1}[\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_i)]\}}{[\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_i)]^T \mathbf{H}_i^{-1}[\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_i)]}$$

here $\mathbf{u} \bullet \mathbf{v}$ means a matrix formed from vectors: $(\mathbf{u} \bullet \mathbf{v})_{ij} = u_i v_j$.

- The Broyden-Fletcher-Goldfarb-Shanno (**BFGS**) algorithm is similar to DFP but includes an additional term

$$\ldots + [\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_i)]^T \mathbf{H}_i^{-1}[\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_i)] \times (\mathbf{u} \bullet \mathbf{u})$$

where vector $\mathbf{u}$ is

$$\mathbf{u} = \frac{\mathbf{x}_{i+1} - \mathbf{x}_i}{(\mathbf{x}_{i+1} - \mathbf{x}_i)^T [\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_i)]} - \frac{\mathbf{H}_i^{-1}[\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_i)]}{[\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_i)]^T \mathbf{H}_i^{-1}[\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_i)]} .$$

Minimization of functions: quasi-Newton's methods

- To give an example we use the Numerical Recipes routine `dfpmin` (BFGS algorithm).
- The routine definition is (source NR):

```
void dfpmin(float p[], int n, float gtol, int *iter, float *fret,
    float(*func)(float []), void (*dfunc)(float [], float []))
```

Given a starting point `p[1..n]` that is a vector of length `n`, the Broyden-Fletcher-Goldfarb-Shanno variant of Davidon-Fletcher-Powell minimization is performed on a function `func`, using its gradient as calculated by a routine `dfunc`. The convergence requirement on zeroing the gradient is input as `gtol`. Returned quantities are `p[1..n]` (the location of the minimum), `iter` (the number of iterations that were performed), and `fret` (the minimum value of the function). The routine `lnsrch` is called to perform approximate line minimizations.

```
{
    void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
        float *f, float stpmax, int *check, float (*func)(float []));
    int check,i,its,j;
    float den,fac,fad,fae,fp,stpmax,sum=0.0,sumdg,sumxi,temp,test;
    float *dg,*g,*hdg,**hessin,*pnew,*xi;
```

```
void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
    float *f, float stpmax, int *check, float (*func)(float []))
```

Given an `n`-dimensional point `xold[1..n]`, the value of the function and gradient there, `fold` and `g[1..n]`, and a direction `p[1..n]`, finds a new point `x[1..n]` along the direction `p` from `xold` where the function `func` has decreased "sufficiently." The new function value is returned in `f`. `stpmax` is an input quantity that limits the length of the steps so that you do not try to evaluate the function in regions where it is undefined or subject to overflow. `p` is usually the Newton direction. The output quantity `check` is false (0) on a normal exit. It is true (1) when `x` is too close to `xold`. In a minimization algorithm, this usually signals convergence and can be ignored. However, in a zero-finding algorithm the calling program should check whether the convergence is spurious. Some "difficult" problems may require double precision in this routine.

```
{
    int i;
    float a,alam,alam2,alamin,b,disc,f2,rhs1,rhs2,slope,sum,temp,
        test,tmplam;
```


Minimization of functions: quasi-Newton's methods

- The function to be minimized is

$$f(x, y) = 1 - e^{-x^2/4}\sin^2 y$$

and its derivatives

$$\frac{\partial f}{\partial x} = \frac{1}{2}xe^{-x^2/4}\sin^2 y$$

$$\frac{\partial f}{\partial y} = -2e^{-x^2/4}\cos y \sin y$$

Minimization of functions: quasi-Newton's methods

- The main program looks like (note its float instead of double):

```
#include <stdio.h>
#include <math.h>
#define NRANSI
#include "nr.h"
#include "nrutil.h"
#define A 4.0
#define NDIM 2
#define GTOL 1.0e-4

static int nfunc,ndfunc;

float func(float x[])
{
  nfunc++;
  return 1.0-
       exp(-x[1]*x[1]/A)*sin(x[2])*sin(x[2]);
}

void dfunc(float x[],float df[])
{
  float t;
  ndfunc++;
  t=exp(-x[1]*x[1]/A);
  df[1]=2.0/A*x[1]*t*sin(x[2])*sin(x[2]);
  df[2]=-2.0*t*cos(x[2])*sin(x[2]);
}
```

```
int main(int argc, char **argv)
{
  int iter;
  float *p,fret;
  if (argc!=3) {
    fprintf(stderr,"Usage: %s x0 y0 \n",argv[0]);
    return (1);
  }
  p=vector(1,NDIM);
  p[1]=atof(*++argv);
  p[2]=atof(*++argv);
  nfunc=ndfunc=0;
  printf("Starting vector: (%7.4f,%7.4f)\n",
            p[1],p[2]);
  dfpmin(p,NDIM,GTOL,&iter,&fret,func,dfunc);
  printf("Iterations: %3d\n",iter);
  printf("Func. evals: %3d\n",nfunc);
  printf("Deriv. evals: %3d\n",ndfunc);
  printf("Solution vector: (%9.6f,%9.6f)\n",
         p[1],p[2]);
  printf("Func. value at solution %14.6g\n",fret);
  free_vector(p,1,NDIM);
  return 0;
}
```

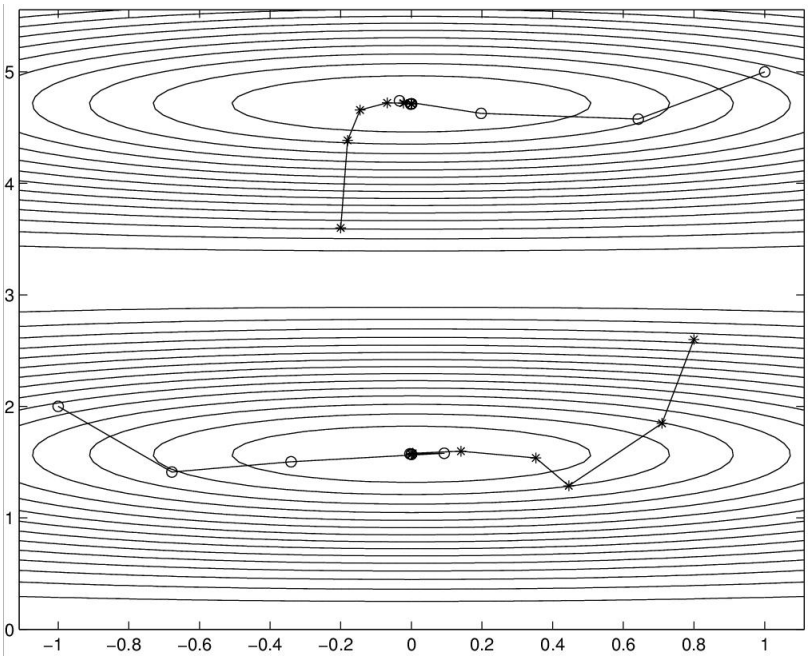

Minimization of functions: quasi-Newton's methods

- Compilation and run (routine `dfpmin` was changed by adding the output of iteration points):

```
dfpmin> cc -o xdfpmin xdfpmin.c dfpmin.c lnsrch.c nrutil.c -lm
xdfpmin.c:
dfpmin.c:
lnsrch.c:
nrutil.c:
dfpmin> xdfpmin -0.2 3.6
Starting vector: (-0.2000, 3.6000)
ITE: 0 -0.2 3.6 0.806124
ITE: 1 -0.180612 4.38577 0.110225
ITE: 2 -0.145471 4.65777 0.00824103
ITE: 3 -0.0685644 4.72128 0.00125362
ITE: 4 -0.0227184 4.71944 0.000178732
ITE: 5 0.000113919 4.71283 2.02031e-07
ITE: 6 2.49532e-05 4.71241 6.16609e-10
ITE: 7 1.98841e-07 4.71239 1.0103e-14
Iterations:   7
Func. evals:  10
Deriv. evals:   8
Solution vector: ( 0.000000, 4.712389)
Func. value at solution     1.0103e-14
dfpmin>
```

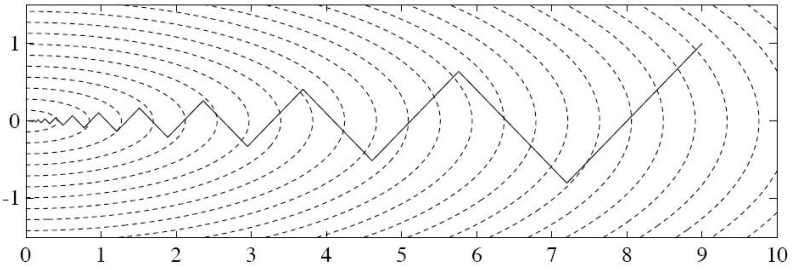
Minimization of functions: quasi-Newton's methods

- Below a couple of iteration paths:



Minimization of functions: conjugate gradients

• As we saw the steepest descent method has the drawback of ending up to a zig-zag path:



- In the conjugate gradients (**CG**) method the new directions in the iteration are chosen in such a way that they do not ruin the minimization in the directions we have already done (direction are conjugate to the previous ones).

Minimization of functions: conjugate gradients

• Minimization and systems of linear equations[1]:

- Assume we have a function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x} + c$$

- Its derivative (gradient) is

$$f'(\mathbf{x}) = \left[\frac{\partial}{\partial x_1}f(\mathbf{x}) \ldots \frac{\partial}{\partial x_N}f(\mathbf{x})\right]^T = \frac{1}{2}\mathbf{A}^T\mathbf{x} + \frac{1}{2}\mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{A}\mathbf{x} - \mathbf{b}$$

If $\mathbf{A}$ symmetric

- Assume $\mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b}$ and $\mathbf{p}$ is an arbitrary vector. One can show that if $\mathbf{A}$ is symmetric then

$$f(\mathbf{p}) = f(\mathbf{x}^*) + \frac{1}{2}(\mathbf{p} - \mathbf{x}^*)^T\mathbf{A}(\mathbf{p} - \mathbf{x}^*)$$

- If $\mathbf{A}$ is positive definite i.e. $\forall \mathbf{x}, \mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ then $\mathbf{x}^*$ is the minimum of $f(\mathbf{x})$ [because $(\mathbf{p} - \mathbf{x}^*)^T\mathbf{A}(\mathbf{p} - \mathbf{x}^*) > 0$].

- Conclusion: If $\mathbf{A}$ is a positive definite and symmetric matrix then the following two are equivalent:

1. Vector $\mathbf{x}^*$ is the solution of $\mathbf{A}\mathbf{x} - \mathbf{b} = 0$.
2. Vector $\mathbf{x}^*$ minimizes $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x} + c$.

1. Note the slight differences in the notation as compared to other material in this chapter.

Minimization of functions: conjugate gradients

• What is really meant by two directions being conjugate to each other? Consider an arbitrary function $f(\mathbf{x})$ of N dimensional argument, and construct its Taylor-series around a point $\mathbf{P}$:

$$f(\mathbf{x}) = f(\mathbf{P}) + \sum_i \frac{\partial f}{\partial x_i}x_i + \frac{1}{2}\sum_{i,j}\frac{\partial^2 f}{\partial x_i \partial x_j}x_i x_j + \ldots \approx c - \mathbf{b}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}$$

$$\text{where} \quad c \equiv f(\mathbf{P}) \qquad \mathbf{b} = -\nabla f\big|_{\mathbf{P}} \qquad \mathbf{A} = \left.\frac{\partial^2 f}{\partial x_i \partial x_j}\right|_{\mathbf{P}}$$

- The matrix $\mathbf{A}$ is the so called Hessian matrix. In this approximation the gradient of f is $\nabla f = \mathbf{A}\mathbf{x} - \mathbf{b}$, and a change in the gradient ∇f over some distance $\delta\mathbf{x}$ is again

$$\delta(\nabla f) = \mathbf{A}(\delta\mathbf{x})$$

- The previous direction in which we have moved is $\mathbf{u}$, gradient is $\mathbf{g}$. How to construct the next direction $\mathbf{v}$?

- In the current point: $\mathbf{g} \perp \mathbf{u}$.

- After the next step we still want $\mathbf{g}' \perp \mathbf{u}$ $\rightarrow$ the change in the gradient $\delta(\nabla f)$ should be perpendicular to $\mathbf{u}$:

$$\mathbf{u}^T\delta(\nabla f) = 0 \Rightarrow \mathbf{u}^T\mathbf{A}\mathbf{v} = 0$$

- If this is valid, the directions $\mathbf{u}$ and $\mathbf{v}$ are considered to be *conjugated* (or $\mathbf{A}$ conjugated).

Minimization of functions: conjugate gradients

- In the **CG** method two vectors $\mathbf{g}$ and $\mathbf{h}$ are used to calculate the new direction into which to move.
- $\mathbf{h}$ is the actual direction into which the line minimization is carried out.
- In solving linear equations, these are iterated as follows:

$$\mathbf{g}_{i+1} = \mathbf{g}_i - \lambda_i(\mathbf{A}\mathbf{h}_i) \quad \text{and} \quad \mathbf{h}_{i+1} = \mathbf{g}_{i+1} + \gamma_i \mathbf{h}_i$$

where

$$\lambda_i = \frac{\mathbf{g}_i^T \mathbf{g}_i}{\mathbf{g}_i^T \mathbf{A}\mathbf{h}_i}, \quad \gamma_i = \frac{\mathbf{g}_{i+1}^T \mathbf{A}\mathbf{h}_i}{\mathbf{h}_i^T \mathbf{A}\mathbf{h}_i}.$$

One can see that these can be written as

$$\lambda_i = \frac{\mathbf{g}_i^T \mathbf{h}_i}{\mathbf{h}_i^T \mathbf{A}\mathbf{h}_i}, \quad \gamma_i = \frac{\mathbf{g}_{i+1}^T \mathbf{g}_{i+1}}{\mathbf{g}_i^T \mathbf{g}_i} = \frac{(\mathbf{g}_{i+1} - \mathbf{g}_i)^T \mathbf{g}_{i+1}}{\mathbf{g}_i^T \mathbf{g}_i}$$

- The vectors $\mathbf{g}$ and $\mathbf{h}$ fulfill the orthogonality and conjugation requirements[1]:

$$\mathbf{g}_i^T \mathbf{g}_j = 0 \qquad \mathbf{h}_i^T \mathbf{A}\mathbf{h}_j = 0 \qquad \mathbf{g}_i^T \mathbf{h}_j = 0, \text{ when } i \neq j$$

- Not suitable for large systems: the $N \times N$ matrix $\mathbf{A}$!
- However, suppose that we have $\mathbf{g}_i = -\nabla f(\mathbf{x}_i)$. Further, suppose we proceed from $\mathbf{x}_i$ to direction $\mathbf{h}_i$ to the local minumum of $f(\mathbf{x})$ located at $\mathbf{x}_{i+1} = \mathbf{x}_i + \lambda \mathbf{h}_i$, and set $\mathbf{g}_{i+1} = -\nabla f(\mathbf{x}_{i+1})$.
- Now this vector $\mathbf{g}_{i+1}$ is the same as would have been constructed by the above iteration because[2]

$$\mathbf{g}_i = -\mathbf{A}\mathbf{x}_i + \mathbf{b}$$

$$\mathbf{g}_{i+1} = -\mathbf{A}(\mathbf{x}_i + \lambda \mathbf{h}_i) + \mathbf{b} = -\mathbf{A}\mathbf{x}_i - \lambda \mathbf{A}\mathbf{h}_i + \mathbf{b} = \mathbf{g}_i - \lambda \mathbf{A}\mathbf{h}_i \qquad \textbf{(a)}$$

Line minimization: $\mathbf{h}_i^T \nabla f(\mathbf{x}_{i+1}) = -\mathbf{h}_i^T \mathbf{g}_{i+1} = 0$ **(b)**

From **(a)** and **(b)** we can solve $\lambda = (\mathbf{h}_i^T \mathbf{g}_i) / (\mathbf{h}_i^T \mathbf{A}_i \mathbf{h}_i)$.

1. For proof see e.g. E. Polak, *Computational Methods in Optimization: A Unified Approach*, Academic Press, 1971, Chapter 2.3
2. Here we assume that our function is of the quadratic form: $c - \mathbf{b}^T \mathbf{x} + \mathbf{x}^T \mathbf{A}\mathbf{x}/2$.

Minimization of functions: conjugate gradients

- Using these equations we obtain the following algorithm for conjugate gradient minimization:

1. Get the initial point for iteration $\mathbf{x}_1$. Set $\mathbf{g}_1 = -\nabla f(\mathbf{x}_1)$, $\mathbf{h}_1 = \mathbf{g}_1$, $i = 1$.

2. Minimize $f(\mathbf{x})$ in direction $\mathbf{h}_i$; i.e. minimize $f(\mathbf{x}_i + \lambda \mathbf{h}_i)$ with respect to λ. Set $\mathbf{x}_{i+1} = \mathbf{x}_i + \lambda \mathbf{h}_i$.

3. Set $\mathbf{g}_{i+1} = -\nabla f(\mathbf{x}_{i+1})$.

4. Compute $\gamma_i = \dfrac{(\mathbf{g}_{i+1} - \mathbf{g}_i)^T \mathbf{g}_{i+1}}{\mathbf{g}_i^T \mathbf{g}_i}$.

5. Update the direction: $\mathbf{h}_{i+1} = \mathbf{g}_{i+1} + \gamma_i \mathbf{h}_i$.

- The above is the original, so called Fletcher-Reeves - algorithm. In some cases it is more efficient to use the so called Polak-Ribiere- version, which is identical to the above except that step 4 is:

4. Calculate $\gamma_i = \dfrac{(\mathbf{x}_i + \mathbf{g}_i)^T \mathbf{x}_i}{\mathbf{g}_i^T \mathbf{g}_i}$.

- It is also possible to reset the iteration (if something seems to go wrong) by resetting the search direction to the opposite of the gradient.

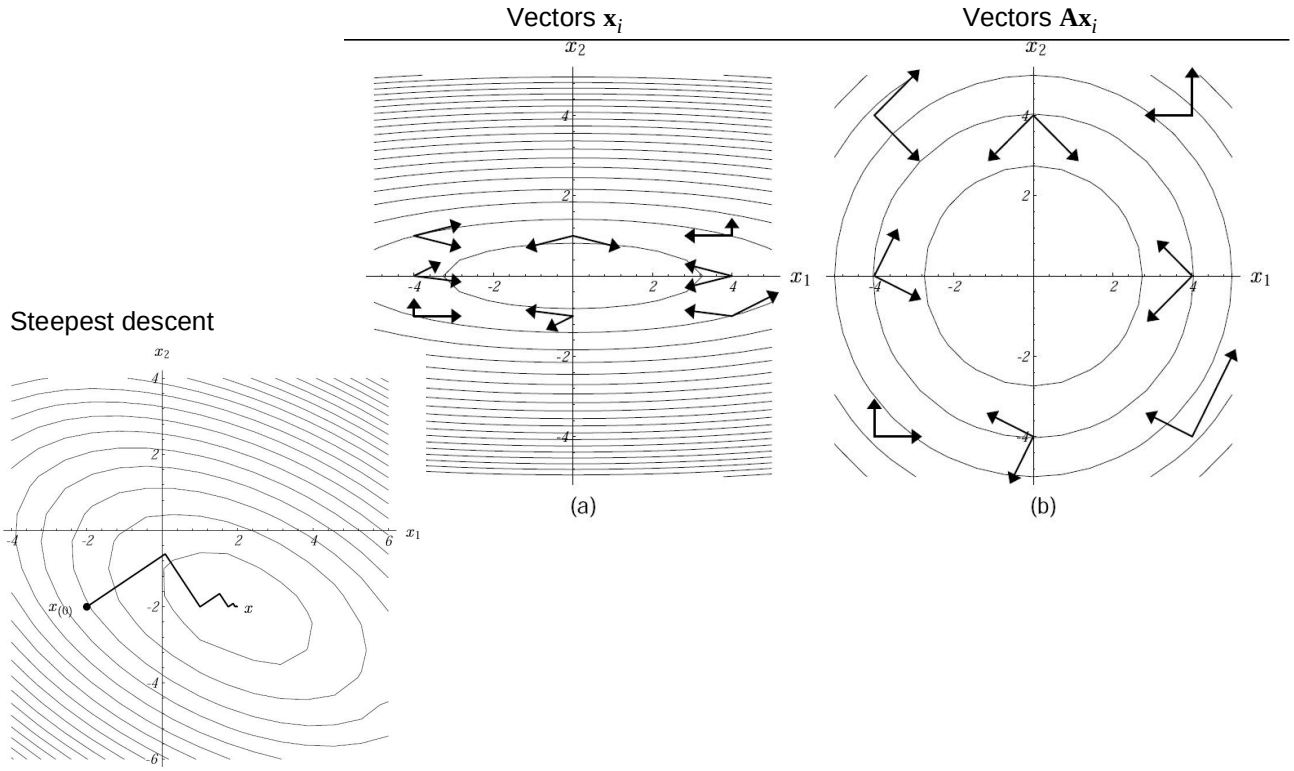
Minimization of functions: conjugate gradients

- Note that the CG method is exact for quadratic functions in the sense that they are minimized in exactly N iterations. (N is the dimension of the problem).

- Near the minimum quadratic is a good approximation.

Minimization of functions: conjugate gradients

- The concept of $\mathbf{A}$ conjugated vectors can be illustrated by the following[1].



1. J. R. Shewchuk: An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, www.cs.cmu.edu/~quake-papers/ painless-conjugate-gradient.pdf

Minimization of functions: conjugate gradients

- As an example we take the familiar function $f(x, y) = 1 - e^{-x^2/4}\sin^2 y$ (using the NR CG routine `frprmn`):

```
#include <stdio.h>
#include <math.h>
#define NRANSI
#include "nr.h"
#include "nrutil.h"
#define A 4.0
#define NDIM 2
#define FTOL 1.0e-6

static int nfunc,ndfunc;

float func(float x[])
{
  nfunc++;
  return 1.0-exp(-x[1]*x[1]/A)*sin(x[2])*sin(x[2]);
}

void dfunc(float x[],float df[])
{
  float t;
  ndfunc++;
  t=exp(-x[1]*x[1]/A);
  df[1]=2.0/A*x[1]*t*sin(x[2])*sin(x[2]);
  df[2]=-2.0*t*cos(x[2])*sin(x[2]);
}

int main(int argc, char **argv)
{
  int iter;
  float fret,*p;

  p=vector(1,NDIM);
  if (argc!=3) {
    fprintf(stderr,"Usage: %s x0 y0 \n",argv[0]);
    return (1);
  }
  p=vector(1,NDIM);
  p[1]=atof(*++argv);
  p[2]=atof(*++argv);
  printf("Starting vector: (%7.4f,%7.4f)\n",p[1],p[2]);
  frprmn(p,NDIM,FTOL,&iter,&fret,func,dfunc);
  printf("Iterations: %3d\n",iter);
  printf("Func. evals: %3d\n",nfunc);
  printf("Deriv. evals: %3d\n",ndfunc);
  printf("Solution vector: (%9.6f,%9.6f)\n",p[1],p[2]);
  printf("Func. value at solution %14.6g\n",fret);
  free_vector(p,1,NDIM);
  return 0;
}
```

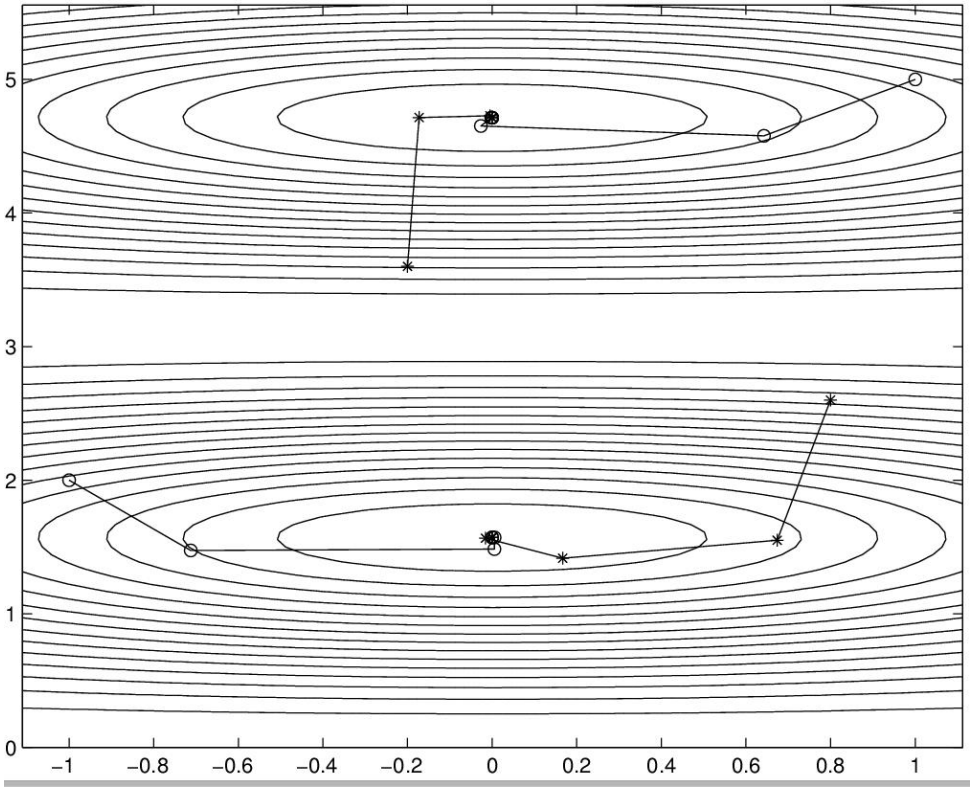

Minimization of functions: conjugate gradients

- Compilation and run:

```
frprmn> cc -o xfrprmn xfrprmn.c frprmn.c mnbrak.c linmin.c nrutil.c
frprmn> xfrprmn -0.2 3.6
Starting vector: (-0.2000, 3.6000)
ITE: 0 -0.2 3.6 0.806124
ITE: 1 -0.172527 4.71345 0.00741493
ITE: 2 -0.00504783 4.72753 0.000235587
ITE: 3 8.32696e-05 4.7124 1.78478e-09
ITE: 4 1.24055e-08 4.71239 2.22045e-16
Iterations:   5
Func. evals:  68
Deriv. evals:   5
Solution vector: (-0.000000, 4.712389)
Func. value at solution    2.22045e-16
```

Minimization of functions: conjugate gradients

- And the iteration graphically:



Minimization of functions: conjugate gradients

• CG is a suitable method for large problems because the Hessian matrix is not needed at all.

- It has been widely applied e.g. in the field of atomistic simulations where the dimension of the problem may easily be of the order of 10^5.

Minimization of functions: GSL

• Quasi-Newton method is also implemented in the GNU Scientific Library (**GSL**) minimization routines:

- In Xemacs: **Help** → **Info** → **Info Contents** → **gsl-ref**

- In GNU Emacs: **Help** → **Manuals** → **Browse Manuals with Info** → **gsl-ref**

- Shell: **info** → **gsl-ref**

- GSL is included in most Linux distributions (e.g. RH/Fedora packages `gsl` and `gsl-devel`.
- It is also installed on **punk.helsinki.fi** .

- Compilation: `cc -o prog prog.c -lm -lgsl -lgslcblas`

- Note that you may have to use compiler options `-I` and `-L` if the header files or libraries are in nonstandard directories.

- GSL home page: **http://www.gnu.org/software/gsl/**

Minimization of functions: stochastic methods

• Methods described above are deterministic in the sense that the initial point of the iteration determines the outcome of the computation.

• In stochastic methods iteration is done in a random fashion so that you (very) probably find the minimum.
[Well, you can always repeat the iteration exactly by starting the random number generator with the same seed number(s).]

• Here we present two stochastic methods:

1. Simulated annealing
2. Genetic algorithms

• These methods are particularly suitable for discrete optimization (combinatorial problems) but can as well be used in continuous minimizations.

• Both methods are inspired by nature:

Simulated annealing: heating and cooling of materials in order to reach their equilibrium structure
Genetic algorithms : evolution of species

Minimization of functions: simulated annealing

• Annealing: heat material to high temperatures and then slowly cool it → material attains its equilibrium structure
  - I.e. (free) energy is minimized.

• Simulated annealing (**SA**):
  - energy $E(\mathbf{r})$ ↔ function to be minimized $f(\mathbf{x})$
  - temperature ↔ control parameter

• Physical system follows the Boltzmann distribution[1]

$$P(E(\mathbf{r})) \propto e^{-(E(\mathbf{r}))/T}, \qquad \textbf{(BD1)}$$

where $P(E)$ is the probability that the system is in the state with energy E.

- In Metropolis Monte Carlo (MC) simulations we generate configurations that have the distribution (BD1).

- In simulate annealing we generate configurations or vectors $\mathbf{x}$ that have the distribution

$$P(f) \propto e^{-f(\mathbf{x})/c}, \qquad \textbf{(BD2)}$$

where c is a control parameter having the role of temperature.

- One can show that when $c \to 0$ distribution (BD2) approaches the minimum configuration

$$P_{\min}(f) \propto \delta(\mathbf{x} - \mathbf{x}_{\min}),$$ where $\mathbf{x}_{\min}$ is the position of the function minimum.

1. In this chapter we set Boltzmann constant $k_B = 1$.

Minimization of functions: simulated annealing

• How do we generate points $\mathbf{x}$ with distribution?

- Metropolis MC.

  - Assume we have the iteration point $\mathbf{x}_i$.

  - We add a perturbation $\delta\mathbf{x}$ to it.

  - Accept the new iteration point $\mathbf{x}_j = \mathbf{x}_i + \delta\mathbf{x}$ by probability

$$P_{\text{acc}}(i \to j) = \begin{cases} 1, & \delta f_{ij} \le 0 \\ \exp\left(-\dfrac{\delta f_{ij}}{c}\right), & \delta f_{ij} > 0 \end{cases}, \qquad \textbf{(MMC)}$$

  where $\delta f_{ij} = f(\mathbf{x}_j) - f(\mathbf{x}_i)$

  - In other words:
    If we would go downhill we accept the new state right away.
    However, if we would go uphill there is a finite probability to accept the new state.

  - The last point is important in the sense that with **SA** it is possible to reach other function minima than the nearest local one.

Minimization of functions: simulated annealing

- That equation (MMC) really generates states with the right distribution can be proved as follows.

  - We take a discrete notation here. It can be generalized to continuous variables.

  - Let the system probability distribution be denoted by the vector $\mathbf{P}^{(i)}$.
    - Its elements tell the probability of each state at iteration i.

  - We have a Markov process[1] that is generated by the stochastic matrix $\mathbf{Q}$.
    - Element Q_{ij} gives the probability for transition $i \to j$.
    - In general a stochastic matrix fulfills the following conditions

      $$\sum_j Q_{ij} = 1, \quad Q_{ij} > 0.$$

    - We can proceed forward in the Markov chain by operating by the stochastic matrix

      $$\mathbf{P}^{(i+1)} = \mathbf{P}^{(i)}\mathbf{Q}.$$

    - In long time limit of the probability distribution is obtained by

      $$\mathbf{P} = \lim_{\tau \to \infty} \mathbf{P}^{(1)}\mathbf{Q}^{\tau}$$

    - This equilibrium distribution fulfills the eigen value equation

      $$\mathbf{P} = \mathbf{P}\mathbf{Q}$$

      i.e. we have a stationary state.

  - By expanding this we get

    $$\sum_m P_m Q_{mn} = P_n. \qquad \textbf{(EQD)}$$

1. The new state only depends on the current one, not the previous states; i.e. no memory.

Minimization of functions: simulated annealing

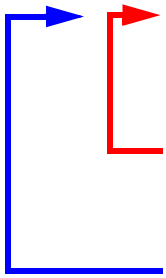
- Instead of using exactly equation (EQD) for out algorithm we take a more restricting condition; so called **detailed balance** on which we base our algorithm:

  $$P_i Q_{ij} = P_j Q_{ji} \qquad \textbf{(DB)}$$

- It is easy to see that **(EQD)** $\Rightarrow$ **(DB)**

- Another condition for the algorithm is that one must ne able to reach all the possible states (ergodicity).
- Moreover, the trial probability for the iteration must be symmetric.

- The algorithm looks like this:



1. Set the initial value for the control parameter $c = c_0$.
2. Set $i = 1$. Set the initial state $\mathbf{x}_i = \mathbf{x}_0$.
3. Change the system state: $\mathbf{x}_i \leftarrow \mathbf{x}_i + \delta\mathbf{x}$.
4. Compute the change in the function value: $\delta f = f(\mathbf{x}_i + \delta\mathbf{x}) - f(\mathbf{x}_i)$.
5. Generate an evenly distributed random ξ number in the interval $[0, 1]$.
6. If $\xi < \exp(-\delta f / c)$ accept the new state.
7. Set $i \leftarrow i + 1$. If $i \le i_{\max}$ go to step 3, otherwise go to step 8.
8. Lower the control parameter; e.g. $c \leftarrow \alpha c$, $0 < \alpha < 1$.
9. If $c < c_{\min}$ stop.
10. Set $i \leftarrow 1$ and go to step 3.

Minimization of functions: simulated annealing

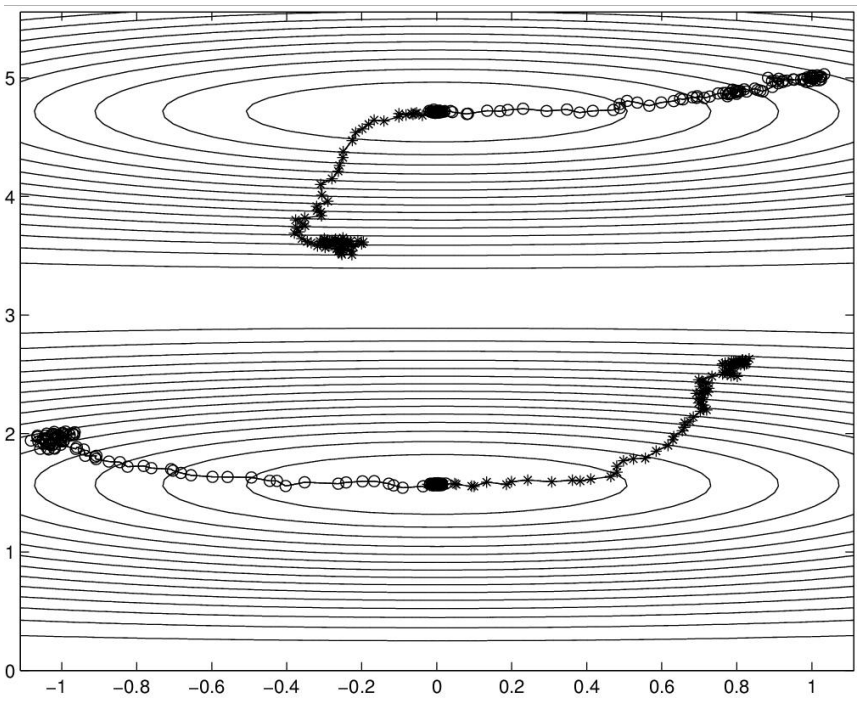
- For an example let's take the familiar function $f(x, y) = 1 - e^{-x^2/4}\sin^2 y$

```
#include <stdio.h>
#include <math.h>
#define NRANSI
#include "nr.h"
#include "nrutil.h"
#define NDIM 2
#define MAXSTEP 1000
#define A 4.0
static int nfunc=0;
int metropx(float de, float t, long *seed);
float func(float x[]) {
  nfunc++;return 1.0-
   exp(-x[1]*x[1]/A)*sin(x[2])*sin(x[2]);}

int main(int argc, char **argv)
{
  int niter,iter,i,j,ans;
  float fret,*x,Tfrac,Tini,Tmin,
      T,dx,ddx,f1,f2,df;
  long seed;
  x=vector(1,NDIM);
  if (argc!=9) {
    fprintf(stderr,"Usage: %s x0 y0 niter
    Tfrac Tini Tmin seed dx\n",
    argv[0]); return (1);
  }
  x=vector(1,NDIM);
  x[1]=atof(*++argv); x[2]=atof(*++argv);
  niter=atoi(*++argv); Tfrac=atof(*++argv);
  Tini=atof(*++argv); Tmin=atof(*++argv);
  seed=atoi(*++argv); dx=atof(*++argv);
  T=Tini;
  printf("i: %d %g %g %g %g\n",
       0,T,x[1],x[2],func(x));
  for (i=1;i<=MAXSTEP;i++) {
    for (iter=1;iter<=niter;iter++) {
      for (j=1;j<=NDIM;j++) {
     f1=func(x);
     ddx=dx*(2.0*ran3(&seed)-1.0);
     x[j]+=ddx;
     f2=func(x);
     df=f2-f1;
     ans=metropx(df,T,&seed);
     if (!ans) x[j]-=ddx;
        }
    }
    T*=Tfrac;
    if (T<Tmin) {
      printf("Iteration ended at step %d:
           T = %g\n",i,T);
      printf("Minimun = %g %g\nF(xmin) = %g\n",
           x[1],x[2],func(x));
      printf("Function evaluations =  %d\n",
           nfunc);
      return;
    }
  }
}
```


Minimization of functions: simulated annealing

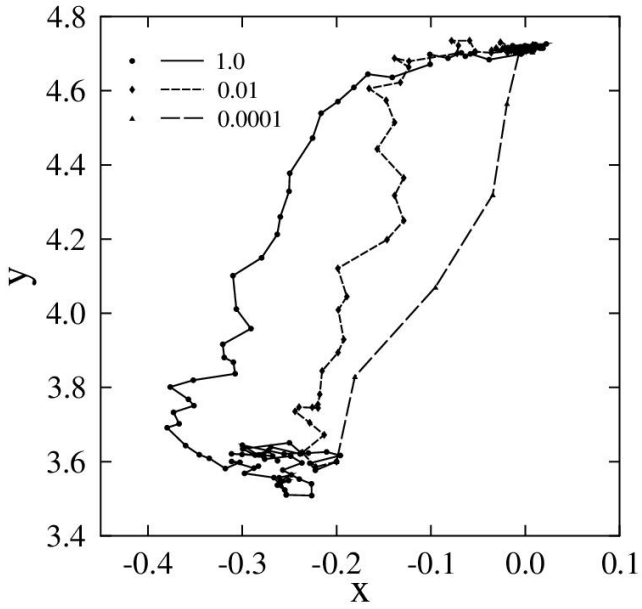
- Below is shown how the iteration proceeds.
  `niter=1000`, `Tini=1.0`, `Tfrac=0.95`, and `Tmin=0.00001`.



- We see that the convergence is very slow.

Minimization of functions: simulated annealing

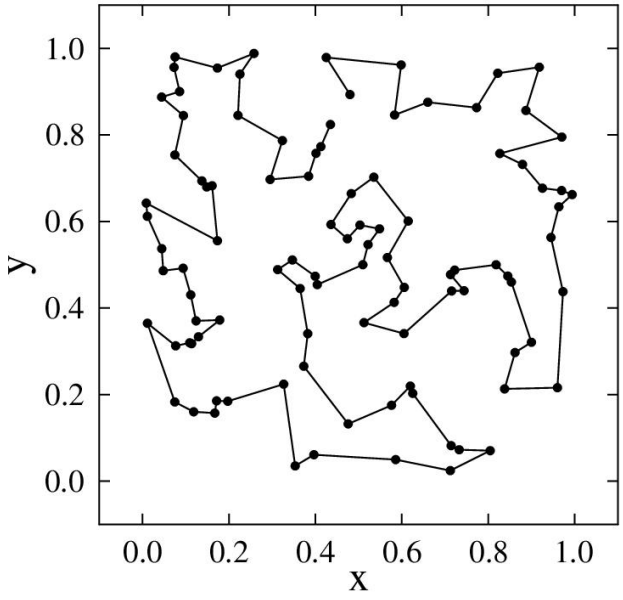
- By adjusting the parameters we get somewhat faster convergence: i.e. dependence on the initial temperature



- This example was only to demonstrate the method. 2D minimization is best done using CG of quasi-Newton's methods.

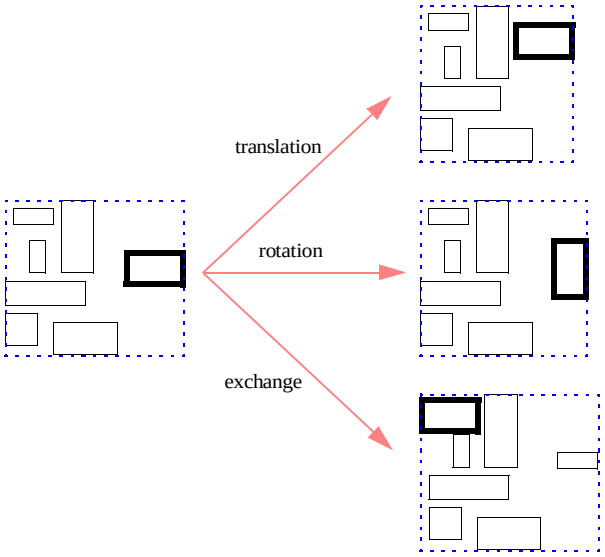
Minimization of functions: simulated annealing

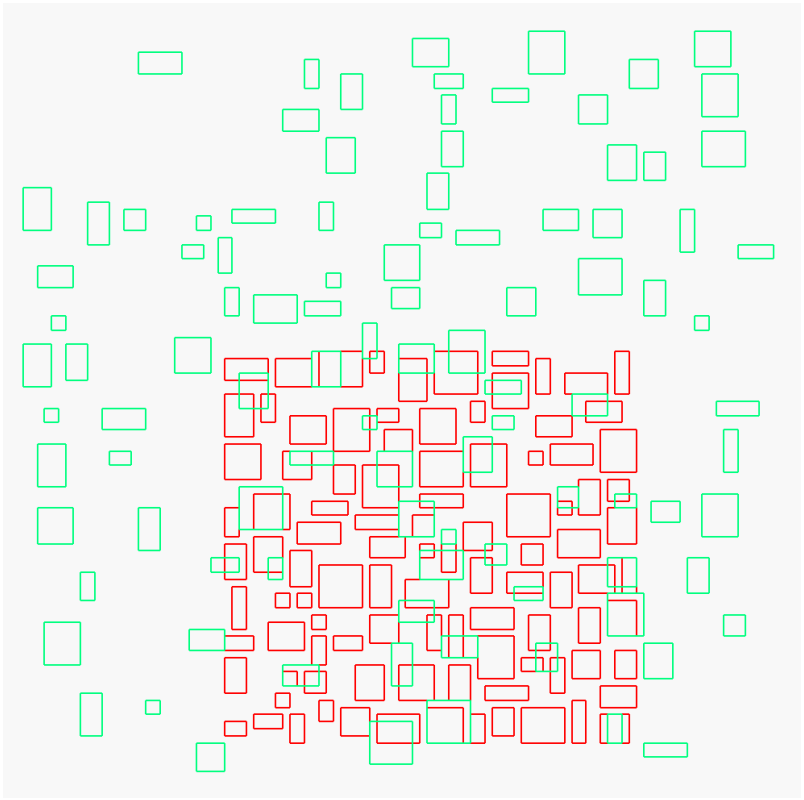
- As an example of a more suitable problem is the traveling salesman problem:



Minimization of functions: simulated annealing

- Another example: packing of rectangles.





Minimization of functions: genetic algorithms

• Genetic algorithms (**GA**) have obtained their inspiration from Darwin's theory of evolution.
- The idea is to perform natural selection for some group of parameters G (population) which describes well the real system.
- The group is allowed to breed by mating, after which natural selection is carried out (i.e. the poorest adapted species are killed).
- Also mutations may be introduced.

• The parameters G can be considered to correspond to a gene sequence, DNA.

• As a concrete example let's take as an example an application of GA to chemistry[1]:structure of molecule dimers.

- Let us consider the interaction between two molecules A and B.
- The relative position and orientation of the molecules is described by coordinates and angles: $(x, y, z, \alpha, \theta, \phi)$.
- If we now discretize the possible positions and angles, using e.g. 16 possibilities for each dimension, the state of the molecule can be described with 24 bits of information, for instance

  (4.5 Å,5.0 Å,9.0 Å,120^o, 100^o,60^o)=(1001:1010:1110:0110:0101:0011).
- The breeding operation is defined such that the binary string is exchanged from some point forward ("crossover"). So if we have two parents

  P1 = (1001:1010:1110:0110:0101:0011)
  P2 = (1001:1010:1110:0100:1011:1110)

  and the exchange position is chosen to be 21, we get the children

  C1 = (1001:1010:1110:0110:0101: 1110)
  C2 = (1001:1010:1110:0100:1011: 0011)

1. Xiao and Williams, *Chem. Phys. Lett.* **215** (1993) 17.

Minimization of functions: genetic algorithms

- The algorithm can be described as:

0. Start. Create the initial population $\mathbf{G} = \{G_1, G_2, \ldots, G_N\}$, $G_i = (x, y, z, \alpha, \theta, \phi)_i$.

1. Mating and breeding. Select two well-adjusted parents for breeding. This is done by selecting a given parent i with state G_i with the probability

$$P(G_i) \propto e^{-E(G_i)/T_\mathrm{m}}$$

where the mating 'temperature' T_m is selected as the range of energies among the whole population $\{G_i\}$. Exchange the gene sequence of a parents with another starting from a random position.

2. Mutation. With a given probability exchange the state of a bit (0→1 or 1→0) for all bits in all individuals.

***[3. Minimize the energy** of the child to the closest local minimum. This is done by e.g. CG.]*

4. Natural selection. If the child has **lower energy** than any of the parents, allow it to stay alive. Then check that its energy does not match the energy of any parent within an energy range δE. If this is true, include it in the population, and kill the least-well adapted parent (the one with the highest E).

5. Convergence test. If convergence has not been reached, return to stage **1.**

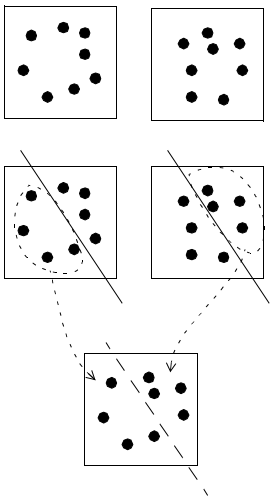
- The authors found the equilibrium structures for benzene, naphtalene, and anthracene dimers.

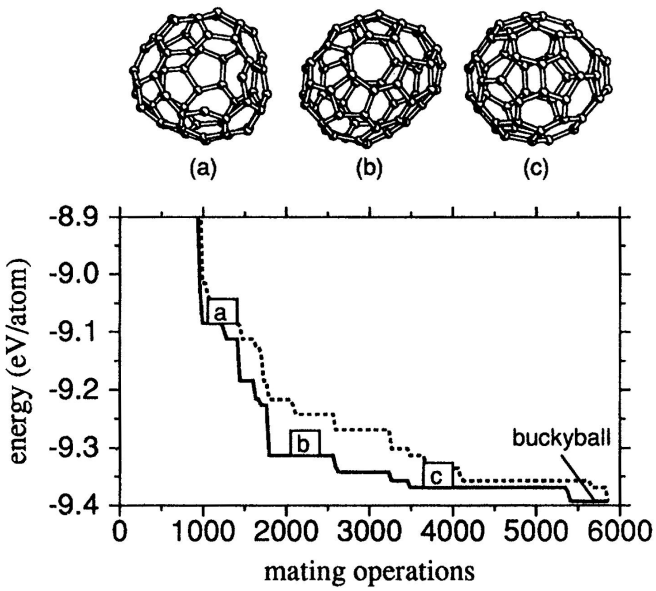
Minimization of functions: genetic algorithms

- Using a rather similar GA the equilibrium structure of carbon clusters was determined in Deaven and Ho, *Phys. Rev. Lett.* **75** (1995) 288.

  - Their method of mating and breeding can be described by the figure on the right.

  - They could obtain the C_{60} buckyball structure by starting from a random configuration.





Minimization of functions: stochastic optimization

• Simulated annealing:
  + Easy to implement
  + May work well both for discrete and continuous problems
  - Convergence may be slow
  - Convergence may depend on cooling scheme

• Genetic algorithms
  + May work well for problems where others fail: finding of global minimum.
  + May work well both for discrete and continuous problems
  - Convergence may be slow
  - Building the model (genes, breeding, mutations)
  - May be difficult to tell beforehand whether the method works when you face a new problem.