# Golden section search I

## Finding the minimum of a function in one dimension

The simplest strategy for finding a minimum of a function is bracketing, similar to bisection in root finding. But in contrast to root finding where the best strategy is to continuously half the search interval, the selection of an optimal new abscissa point is different in the case of minimization.

While a root is bracketed by two points $a$ and $b$ if the signs of $f(a)$ and $f(b)$ are opposite, we need three points to bracket a minimum: $a < b < c$ with the property $f(a) > f(b)$ and $f(c) > f(b)$. Now if we choose a new point $x$ between $b$ and $c$, we can have $f(b) < f(x)$ leading to the new bracketing triplet $(a, b, x)$, or $f(b) > f(x)$ leading to the triplet $(b, x, c)$.

# Golden section search II

Now for a strategy to choose the new point $x$ given $(a, b, c)$. If $b$ is a fraction $w$ of the way between $a$ and $c$:

$$\frac{b-a}{c-a} = w \qquad \frac{c-b}{c-a} = 1-w \tag{1}$$

and the new trial point $x$ is an additional fraction $z$ beyond $b$:

$$\frac{x-b}{c-a} = z \tag{2}$$

Then the next bracketing segment will be either $w+z$ or $1-w$ in length. In order to minimize the worst case possibility, we will try to make them equal:

$$z = 1-2w \tag{3}$$

This makes $|b-a|$ equal to $|x-c|$. But now $w$ is still undetermined. We can find it by demanding that $w$ was also chosen optimally.

# Golden section search III

The scale similarity implies that $x$ should be the same fraction of the way from $b$ to $c$ as was $b$ from $a$ to $c$, or

$$\frac{z}{1-w} = w \tag{4}$$

Together, Eqs. (3) and (4) yield

$$w^2 - 3w + 1 = 0 \curvearrowright w = \frac{3-\sqrt{5}}{2} \approx 0.38197 \tag{5}$$

Thus in a bracketing triplet $(a, b, c)$, $b$ has a relative distance of 0.38197 from $a$ and of 0.61803 from $c$. These fractions correspond to the golden section so that the minimization is also called **golden section search**. The convergence of this method is linear, meaning that additional significant figures are won linearly with additional function evaluations.

# Precision is machine limited

It is important to note that determination of a minimum can only be done up to a precision corresponding to the square root of the machine precision; *e.g.* for `double` $3 \cdot 10^{-8} \approx \sqrt{10^{-15}}$. This can be understood considering the Taylor expansion close to the minimum

$$f(x) \approx f(b) + \frac{1}{2} f''(b)(x-b)^2 \tag{6}$$

The second term will be negligible against the first, *i.e.* a factor of the floating point precision $\epsilon$ smaller, if

$$|x-b| < \sqrt{\epsilon}|b| \sqrt{\frac{2|f(b)|}{b^2 f''(b)}} \tag{7}$$

# Interpolation methods I

A method that can converge faster but cannot be used exclusively is **parabolic interpolation**. The idea is simple: If we draw a parabola through three points $(a, f(x))$, $(b, f(b))$, $(c, f(c))$ bracketing a minimum, we can determine an approximation to the minimum analytically via

$$x = b - \frac{1}{2} \frac{(b-a)^2[f(b)-f(c)] - (b-c)^2[f(b)-f(a)]}{(b-a)[f(b)-f(c)] - (b-c)[f(b)-f(a)]} \tag{8}$$

This doesn't work for three collinear points, and it doesn't distinguish between a minimum and a maximum.

# Interpolation methods II

A good marriage between golden section search and parabolic interpolation is **Brent's method**. It is keeping track of six function points (not necessarily all distinct), $a$, $b$, $u$, $v$, $w$, and $x$, defined as follows: The minimum is bracketed between $a$ and $b$; $x$ is the point with the very least function value found so far; $w$ is the point with the second least function value; $v$ is the previous value of $w$; $u$ is the point at which the function was evaluated most recently. The principle of the algorithm: Parabolic interpolation is attempted, fitting through the points $x$, $v$ and $w$. To be acceptable, the parabolic step must (i) fall within the bounding interval $(a, b)$, and (ii) imply a movement from the best current value $x$ that is less than half the movement of the step before last. This second criterion ensures that the parabolic steps are converging to something and not just bouncing around. If parabolic interpolation is rejected, intersection is used.

# Steepest Descent I

The first idea for minimization in $N$ dimensions is to reduce the task to subsequent onedimensional minimizations.

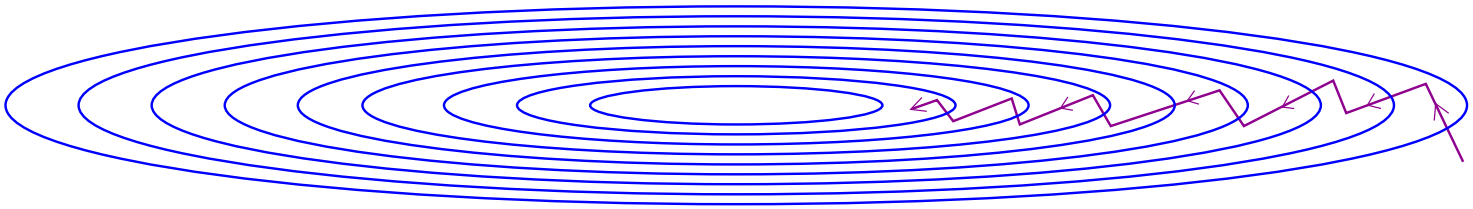
Algorithm of the **steepest descent** method:

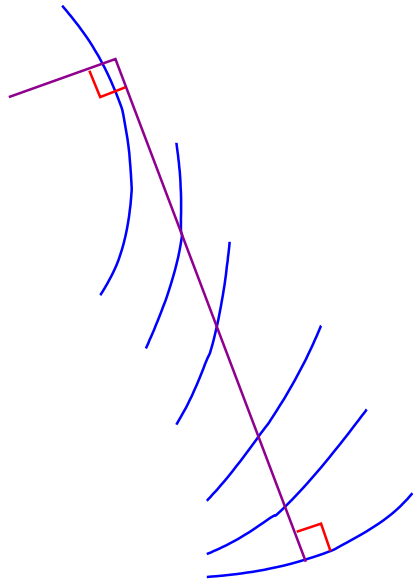
Start at point $\mathbf{P}_0$. As many times as needed, move from point $\mathbf{P}_i$ to the point $\mathbf{P}_{i+1}$ by minimizing along the line from $\mathbf{P}_i$ in the direction of the local downhill gradient $-\nabla f(\mathbf{P}_i)$.

This algorithm is not very good as it will perform many small steps in going down a long, narrow valley even if the valley has perfect quadratic analytic form.

# Steepest Descent II

The figures show how the steepest descent directions zigzag, and how a descent starts off perpendicular to a contour line and proceeds until it is parallel to another in its local minimum, forcing a right angle turn.





The problem of this method is that we need to cycle many times through all $N$ basis vectors in order to reach the minimum. It would be desirable to improve the choice of minimization directions for the $N$ dimensional function, in order to proceed along valley directions or to choose "non-interfering" directions in which minimization along one direction doesn't spoil the previous minimizations along other directions.

# Conjugate Gradient I

Such "non-interfering" directions are called "conjugate". To express this concept mathematically, we write down a Taylor series approximation of $f(\mathbf{x})$ in point $\mathbf{P}$:

$$\begin{aligned} f(\mathbf{x}) &= f(\mathbf{P}) + \sum_i \frac{\partial f}{\partial x_i} x_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 f}{\partial x_i \partial x_j} x_i x_j + \dots \\ &\approx c - \mathbf{b} \cdot \mathbf{x} + \frac{1}{2} \mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} \end{aligned} \tag{9}$$

with $c \equiv f(\mathbf{P})$, $\mathbf{b} = -\nabla f(\mathbf{x})\big|_{\mathbf{x}=\mathbf{P}}$, $[\mathbf{A}]_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}\big|_{\mathbf{x}=\mathbf{P}}$. In the approximation, the gradient of $f$ is

$$\nabla f(\mathbf{x}) = \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \tag{10}$$

How does the gradient $\nabla f(\mathbf{x})$ change as we move along some direction?

$$\delta \nabla f = \mathbf{A} \cdot (\delta \mathbf{x}) \tag{11}$$

# Conjugate Gradient II

Suppose we have moved along some direction **u** to a minimum and now propose to move along some new direction **v**. The condition that motion along **v** not spoil our minimization along **u** is that the gradient stay perpendicular to **u**, *i.e.* that the change in gradient be perpendicular to **u**:

$$0 = \mathbf{u} \cdot \delta \nabla f = \mathbf{u} \cdot \mathbf{A} \cdot \mathbf{v} \tag{12}$$

If this equation holds for **u** and **v** they are called **conjugate**. If this relation holds pairwise for all members of a set of vectors, this is called a conjugate set. For functions that are quadratic forms, $N$ line minimizations in mutually conjugate directions will arrive exactly at the minimum (where $N$ is the dimension of the function).

# Conjugate Gradient III

Side note: The solution of linear systems of equations can be formulated as a minimization problem.
To show that, we have to prove that
i) Solving $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ and
ii) Minimizing $f(\mathbf{x}) = \frac{1}{2}\mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} - \mathbf{x} \cdot \mathbf{b}$
are equivalent. (Note that we write $\mathbf{x} \cdot \mathbf{b} \equiv \mathbf{x}^{\mathrm{T}}\mathbf{b}$, the result of which is a scalar). We take $\mathbf{A}$ to be a symmetric positive definite matrix.
For that purpose we define the auxiliary function

$$E(\mathbf{x}) = \frac{1}{2}(\mathbf{A} \cdot \mathbf{x} - \mathbf{b}) \cdot \mathbf{A}^{-1} \cdot (\mathbf{A} \cdot \mathbf{x} - \mathbf{b}) \tag{13}$$

As $\mathbf{A}^{-1}$ is also positive definite, we have $E(\mathbf{x}) \geq 0$. Thus $E(\mathbf{x})$ is minimal if and only if $\mathbf{A}\mathbf{x} - \mathbf{b} = 0$.
Some algebra on $E(\mathbf{x})$ using $A^{\mathrm{T}} = A$ yields

$$E(\mathbf{x}) = f(\mathbf{x}) + \frac{1}{2}\mathbf{b} \cdot \mathbf{A}^{-1} \cdot \mathbf{b} \tag{14}$$

# Conjugate Gradient IV

As $\mathbf{b} \cdot \mathbf{A}^{-1} \cdot \mathbf{b} \geq 0$ we find that $E(\mathbf{x})$ and $f(\mathbf{x})$ are minimal at the same position, *i.e.*

$$f(\mathbf{x}) \stackrel{!}{=} \min . \Longleftrightarrow \mathbf{A} \cdot \mathbf{x} - \mathbf{b} = 0 \tag{15}$$

The negative gradient of a function points in the direction of the steepest descent; for $f(\mathbf{x})$

$$\nabla f(\mathbf{x}) = \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \tag{16}$$

Thus we need to search for the point $\mathbf{x}$ in which the gradient of $f(\mathbf{x})$ disappears.
In order to spell out the conjugate gradient method of minimization, let's remember that, close to its minimum, a function f can be approximated as a quadratic form:

$$f(\mathbf{x}) = c - \mathbf{b} \cdot \mathbf{x} + \frac{1}{2}\mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} \tag{17}$$

where $\mathbf{A}$ is the Hessian matrix.

# Conjugate Gradient V

Starting with an arbitrary vector $\mathbf{g}_0$ and letting $\mathbf{h}_0 = \mathbf{g}_0$, two sequences of vectors are constructed from the recurrence:

$$\mathbf{g}_{i+1} = \mathbf{g}_i - \lambda_i \mathbf{A} \cdot \mathbf{h}_i \qquad \mathbf{h}_{i+1} = \mathbf{g}_{i+1} - \gamma_i \mathbf{h}_i \quad i = 0, 1, 2, \ldots \tag{18}$$

The vectors satisfy the orthogonality and conjugacy conditions:

$$\mathbf{g}_i \cdot \mathbf{g}_j = 0 \quad \mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_j = 0 \quad \mathbf{g}_i \cdot \mathbf{h}_j = 0 \quad j < i \tag{19}$$

The scalars $\lambda_i$ and $\gamma_i$ are given by

$$\lambda_i = \frac{\mathbf{g}_i \cdot \mathbf{g}_i}{\mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_i} = \frac{\mathbf{g}_i \cdot \mathbf{h}_i}{\mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_i} \qquad \gamma_i = \frac{\mathbf{g}_{i+1} \cdot \mathbf{g}_{i+1}}{\mathbf{g}_i \cdot \mathbf{g}_i} \tag{20}$$

Now suppose we knew the Hessian matrix $\mathbf{A}$ in (17). Then we could use Eq. (18) to construct successively conjugate directions $\mathbf{h}_i$ along which to line-minimize. But so far we don't know $\mathbf{A}$.

# Conjugate Gradient VI

A theorem helps out of this fix: suppose we have $\mathbf{g}_i = -\nabla f(\mathbf{P}_i)$ for some point $\mathbf{P}_i$ where $f$ is of the form (17). Suppose we proceed from $\mathbf{P}_i$ along the direction $\mathbf{h}_i$ to the local minimum of $f$ located at a point $\mathbf{P}_{i+1}$ and set $\mathbf{g}_{i+1} = -\nabla f(\mathbf{P}_{i+1})$. Then, this $\mathbf{g}_{i+1}$ is the same vector as would have been constructed by Eq. (18) (but it was done without knowledge of $\mathbf{A}$!).
Proof: By Eq. (10), $\mathbf{g}_i = -\mathbf{A} \cdot \mathbf{P}_i + \mathbf{b}$, and

$$\mathbf{g}_{i+1} = -\mathbf{A} \cdot (\mathbf{P}_i + \lambda \mathbf{h}_i) + \mathbf{b} = \mathbf{g}_i - \lambda \mathbf{A} \cdot \mathbf{h}_i \tag{21}$$

with $\lambda$ chosen to take us to the line minimum. But at the line minimum, $\mathbf{h}_i \cdot \nabla f = -\mathbf{h}_i \cdot \mathbf{g}_{i+1} = 0$. Together with Eq. (21), we can solve for $\lambda$ and arrive at Eq. (20). With this value of $\lambda$, Eq. (21) becomes equal to Eq. (18) which was to be shown.
The algorithm defined by Eq. (18)-(20) is known as Fletcher-Reeves version of the conjugate gradient algorithm.

# Conjugate Gradient VII

A small change introduced by Polak and Ribiere

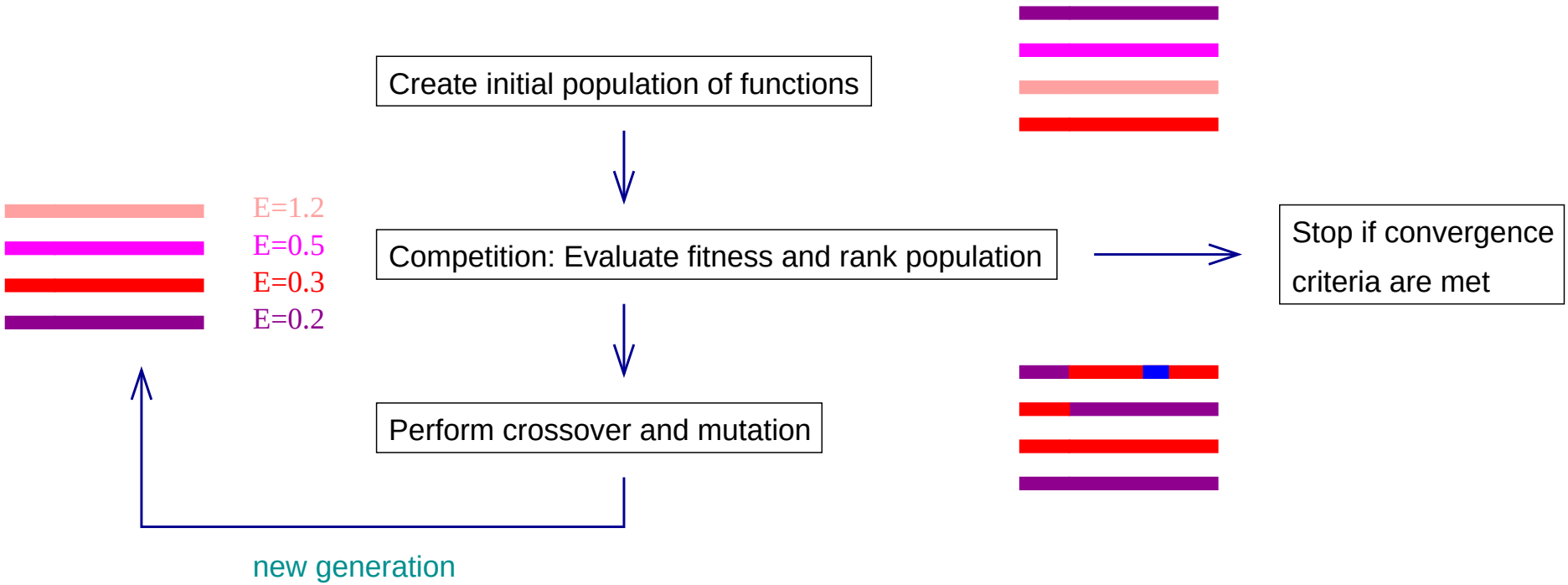
$$\gamma_i = \frac{(\mathbf{g}_{i+1} - \mathbf{g}_i) \cdot \mathbf{g}_{i+1}}{\mathbf{g}_i \cdot \mathbf{g}_i} \tag{22}$$

performs better for functions that are not exactly quadratic forms.

- Conjugate gradient methods are very efficient and recommended. But they share one problem with all gradient-based methods: They cannot distinguish between relative and absolute minima of a function. This means that they proceed to the nearest (relative or absolute) minimum and get stuck there.
- Example: Crystal structure optimization: Potential energy surfaces are very complicated multidimensional functions, and they can possess numerous relative minima (metastable crystal structures).
- There are two types of methods that can overcome this problem:
  1. Molecular dynamics at finite temperature with friction.
  2. Stochastic methods: Monte Carlo, genetic algorithms.

# Genetic Algorithms

## Flow chart of GA optimization




These steps are common to many GA methods; they differ by

1. the way the system to be optimized is represented
2. the rules of the competition
3. the way crossover and mutation are implemented

# Calculating the ground state of a quantum system I

A Hamiltonian in one dimension is given by

$$H = -\frac{1}{2}\nabla^2 + V(x) \tag{23}$$

where $V(x)$ is the external potential. The fitness function describing the quality of a trial wave function is

$$E[\psi] = \frac{\langle\psi|H|\psi\rangle}{\langle\psi|\psi\rangle} \tag{24}$$

For the random initial population, Gaussian like functions obeying the boundary conditions $\psi_j(a) = 0$ and $\psi_j(b) = 0$ are chosen:

$$\psi_j(x) = A \exp\left[-\frac{x - x_j)^2}{\sigma_j^2}\right](x - a)(b - x) \tag{25}$$

# Calculating the ground state of a quantum system II

A smooth crossover between randomly chosen parent functions is defined by

$$\psi_1^{(n+1)}(x) = \psi_1^{(n)}\mathrm{St}(x) + \psi_2^{(n)}[1 - \mathrm{St}(x)] \tag{26}$$

$$';././^{(n+1)}(x) = \psi_2^{(n)}\mathrm{St}(x) + \psi_1^{(n)}[1 - \mathrm{St}(x)] \tag{27}$$

where $\mathrm{St}(x)$ is a smooth step function, *e.g.*

$$St(x) = \frac{1 + \tanh\frac{x-p}{w}}{2} \tag{28}$$

where $p$ and $w$ are used to adjust the position and smoothness of the crossover.
See I. Grigorenko and M.E. Garcia, Physica A **284**, 131 (2000) and **291**, 439 (2001).

# GA Program for solving the 1D Schroedinger equation I

```
#include <complex>
#include "defs.hh"
// Missing: prototypes for NAG routines

int main() {
  int population = 100;
  int generation_number = 100;
  int mesh_size = 1000;
  DMatrix wavefunction(mesh_size,population);
  DVector energy(population);
  DVector current_energy(generation_number);
  double x_range = 100.0;
  double x_step = x_range/mesh_size;
  double perturbation = 0.1;
  double dummy = 0.0; // argument for g05caf
  // start random number generator
  int iseed=5; g05cbf_(&iseed);
  initialize_population(wavefunction,x_range);
```

# GA Program for solving the 1D Schroedinger equation II

```
  for(int generation_i = 0; generation_i < generation_number;
      generation_i++) {
    for(int pop_i=0; pop_i<population; pop_i++)
      energy[pop_i] = fitness(wavefunction,pop_i,x_range);
    competition(wavefunction,energy);
    current_energy[generation_i]
      = fitness(wavefunction,0,x_range);
    for(int pop_i=0; pop_i<(int)(population-2)/2; pop_i+=2) {
      double random_num=1000.0*g05caf_(&dummy);
      if(random_num > 10.0)
        crossover(wavefunction,pop_i,x_range);
      else
        mutation(wavefunction,pop_i,x_range,perturbation);
    }
  }
  // Missing: Output of first wave function as solution
}
```

# Initialization: Population of wave functions is created with Gaussians of random width and normalized

```
void initialize_population(DMatrix& wavefunction,
                           double x_range) {
  int population = wavefunction.column_number;
  int mesh_size = wavefunction.row_number;
  double x_step=x_range/mesh_size;
  double dummy = 0.0; // argument for g05caf
  double sigma;
  double average_width=x_range/20.0;
  wavefunction.zero();
  for(int pop_i=0; pop_i<population; pop_i++) {
    sigma=average_width*g05caf_(&dummy);
    for(int minimum_i=0; minimum_i < 5; minimum_i++)
      for(int mesh_i=0; mesh_i < mesh_size; mesh_i++)
        wavefunction(mesh_i,pop_i)+=exp(-sqr((x_step*mesh_i
          - x_range/2.0+6.0*(minimum_i-2))/sigma)/2.0)
          * mesh_i*(mesh_size-mesh_i+1);
    normalize(wavefunction,pop_i,x_range);
  }
```

# Normalization

```
void normalize(DMatrix& wavefunction, int pop_i,
               double x_range) {
  int mesh_size = wavefunction.row_number;
  double x_step=x_range/mesh_size;
  double sum=0.0;
  for(int mesh_i=0; mesh_i < mesh_size; mesh_i++)
    sum += sqr(wavefunction(mesh_i,pop_i));
  double weight = sqrt(sum*x_step);
  for(int mesh_i=0; mesh_i < mesh_size; mesh_i++)
    wavefunction(mesh_i,pop_i)/=weight;
}
```

# Fitness: Applying Hamiltonian to wave function yields the energy that will be minimal for the ground state solution

```
double fitness(DMatrix& wavefunction, int pop_i, double x_range) {
  int mesh_size = wavefunction.row_number;
  double x_step=x_range/mesh_size; double sum=0.0, sum1=0.0;
  for(int mesh_i=1; mesh_i < mesh_size-1; mesh_i++)
    sum += sqr((wavefunction(mesh_i+1,pop_i)
                - wavefunction(mesh_i-1,pop_i))/2.0);
  sum+=(sqr((wavefunction(1,pop_i)-wavefunction(0,pop_i))/2.0)
        + sqr((wavefunction(mesh_size-1,pop_i)
        - wavefunction(mesh_size-2,pop_i))/2.0))/2.0;
  for(int mesh_i=1; mesh_i < mesh_size-1; mesh_i++)
    sum1 += sqr(wavefunction(mesh_i,pop_i))
            * potential(mesh_i,mesh_size,x_range);
  sum1 += (sqr(wavefunction(0,pop_i))
           * potential(0,mesh_size,x_range)
           + sqr(wavefunction(mesh_size-1,pop_i))
           * potential(mesh_size-1,mesh_size,x_range))/2.0;
  return((sum/x_step + sum1*x_step)/2.0);
}
```

# Potential $V(x)$

Here: a potential with five minima, representing a 1D Xe cluster.

```
double potential(int mesh_i,int mesh_size,double x_range) {
  double x_step=x_range/mesh_size;
  double charge = 54.0;
  double sum = 0.0;
  int well_num=5;
  for(int well_i=0; well_i<well_num; well_i++)
    sum -= charge/sqrt(sqr(mesh_i*x_step-x_range/2.0+6.0
           * (well_i-(int)(well_num/2)))+sqr(2.55))
           * exp(-0.04*sqr(mesh_i*x_step-x_range/2.0+6.0
           * (well_i-(int)(well_num/2))));
  return(sum);
}
```

# Competition: Wave functions are ranked by energy, the best are copied into lower half of population indices

```
void competition(DMatrix& wavefunction, DVector& energy) {
  int population = wavefunction.column_number;
  int mesh_size = wavefunction.row_number;
  int m1=1, m2=population; char order='A';
  static IVector irank(population);
  int ifail=0;
  m01daf_(energy.getpointer(),&m1,&m2,&order, irank.getpointer(),&ifail);
  if(ifail!=0) complain_and_exit("NAG m01daf failed");

  int pop_j = 0;
  for(int pop_i=0; pop_i<population; pop_i++)
    if((irank[pop_i]<(int)(population/2)+1)&&(pop_i!=pop_j)) {
      // copy parent pop_i to position pop_j
      for(int mesh_i=0; mesh_i < mesh_size; mesh_i++)
        wavefunction(mesh_i,pop_j)=wavefunction(mesh_i,pop_i);
      pop_j++;
    }
}
```

# Crossover: Two smooth new wave functions are formed from two old ones

```
void crossover(DMatrix& wavefunction, int pop_i,
               double x_range) {
  int population = wavefunction.column_number;
  int mesh_size = wavefunction.row_number;
  double x_step=x_range/mesh_size;
  double dummy = 0.0;  // argument for g05caf
  int rand_pos = (int)((mesh_size-1)*g05caf_(&dummy)+0.5)+1;
  double rand_width = (0.2 + 0.05*(0.5-g05caf_(&dummy)))*x_range;
  int rand_individual1 = (int)((population/2-1)*g05caf_(&dummy)+0.5)+1;
  int rand_individual2 = (int)((population/2-1)*g05caf_(&dummy)+0.5)+1;
```

# Crossover II

```
  for(int mesh_i=0; mesh_i < mesh_size; mesh_i++) {
    double rand_tanh=blend(mesh_i,x_step,rand_pos,rand_width);
    wavefunction(mesh_i,population/2+pop_i)
      = wavefunction(mesh_i,rand_individual2)*rand_tanh
      + wavefunction(mesh_i,rand_individual1)*(1.0-rand_tanh);
    wavefunction(mesh_i,population/2+pop_i+1)
      = wavefunction(mesh_i,rand_individual1)*rand_tanh
      + wavefunction(mesh_i,rand_individual2)*(1.0-rand_tanh);
  }
  normalize(wavefunction,population/2+pop_i,x_range);
  normalize(wavefunction,population/2+pop_i+1,x_range);
}

double blend(int mesh_i,double x_step,int rand_pos,
             double rand_width) {
  return((1.0+tanh((mesh_i-rand_pos)*x_step/rand_width))/2.0);
}
```

# Mutation: With low probability, a Gaussian of random width and position is added to a pair of wave functions

```
void mutation(DMatrix& wavefunction,int pop_i,double x_range,
      double perturbation) {
  int population = wavefunction.column_number;
  int mesh_size = wavefunction.row_number;
  double x_step=x_range/mesh_size;
  double dummy = 0.0; // argument for g05caf
  double sigma=0.03*x_range*(1.5-g05caf_(&dummy));
  int rand_individual1
    = (int)((population/2-1)*g05caf_(&dummy)+0.5)+1;
  double signed_rand = 0.5-g05caf_(&dummy);
  int rand_pos = (int)((mesh_size-1)*g05caf_(&dummy)+0.5)+1;
  for(int mesh_i=0; mesh_i < mesh_size; mesh_i++)
    wavefunction(mesh_i,population/2+pop_i)
      = wavefunction(mesh_i,rand_individual1)
      + perturbation*signed_rand
      *exp(-sqr((mesh_i-rand_pos)*x_step/sigma))
      *(mesh_size-mesh_i)*mesh_i*sqr(x_step);
```

# Mutation II

```
  sigma=0.1*x_range*(1.5-g05caf_(&dummy));
  rand_individual1
    = (int)((population/2-1)*g05caf_(&dummy)+0.5)+1;
  signed_rand = 0.5-g05caf_(&dummy);
  rand_pos = (int)((mesh_size-1)*g05caf_(&dummy)+0.5)+1;
  for(int mesh_i=0; mesh_i < mesh_size; mesh_i++)
    wavefunction(mesh_i,population/2+pop_i+1)
      = wavefunction(mesh_i,rand_individual1)
      + perturbation*signed_rand
      *exp(-sqr((mesh_i-rand_pos)*x_step/sigma))
      *(mesh_size-mesh_i)*mesh_i*sqr(x_step);
  normalize(wavefunction,population/2+pop_i,x_range);
  normalize(wavefunction,population/2+pop_i+1,x_range);
}
```

# Genetic Algorithms: Practice