Chapter 14 Global Search Algorithms

An Introduction to Optimization

Spring, 2014

Wei-Ta Chu

Introduction

- We discuss various search methods that attempts to search throughout the entire feasible set. These methods use only objective function values and do not require derivatives.
- They are applicable to a much wider class of optimization problems.
- They can be used to generate “good” initial points for the iterative methods discussed in earlier chapters.
- Some methods are also used in combinatorial optimization, where the feasible set is finite, but typically large.

The Nelder-Mead Simplex Algorithm

- A ***simplex*** is a geometric object determined by an assembly of $n+1$ points, $\boldsymbol{p}_0, \boldsymbol{p}_1, ..., \boldsymbol{p}_n$, in the n-dimensional space such that

$$\det \begin{bmatrix} \boldsymbol{p}_0 & \boldsymbol{p}_1 & \cdots & \boldsymbol{p}_n \\ 1 & 1 & \cdots & 1 \end{bmatrix} \neq 0$$

- This condition ensures that two points in R do not coincide, three points in R^2 are not colinear, four points in R^3 are not coplanar, and so on. Thus, simplex in R is a line segment, in R^2 it is a triangle, while a simplex in R^3 is a tetrahedron; in each case it encloses a finite n -dimensional volume.

The Nelder-Mead Simplex Algorithm

- Suppose that we wish to minimize $f(\boldsymbol{x}), \boldsymbol{x} \in R^n$. To start the algorithm, we initialize a simplex of $n+1$ points. A possible way to set up a simplex is to start with an initial point $\boldsymbol{x}^{(0)} = \boldsymbol{p}_0$ and generate the remaining points of the initial simplex as follows:

$$\boldsymbol{p}_i = \boldsymbol{p}_0 + \lambda_i \boldsymbol{e}_i, i = 1, 2, ..., n$$

  where the $\boldsymbol{e}_i$ are unit vectors constituting the natural basis of R^n The positive constant coefficients λ_i are selected in such as way that their magnitudes reflect the length scale of the optimization problem.

The Nelder-Mead Simplex Algorithm

- Our objective is to modify the initial simplex stage by stage so that the resulting simplices converge toward the minimizer. In the function minimization process, the point with the largest function value is replaced with another point. The process of modifying the simplex continues until it converges toward the function minimizer.
- We use a two-dimensional example to illustrate the rules. Select the initial set of $n+1$ points that are to form the initial simplex. We next evaluate f at each point and order the $n+1$ vertices to satisfy

$$f(\boldsymbol{p}_0) \leq f(\boldsymbol{p}_1) \leq \cdots \leq f(\boldsymbol{p}_n)$$

The Nelder-Mead Simplex Algorithm

- For the two-dim case we let $\boldsymbol{p}_l, \boldsymbol{p}_{nl}, \boldsymbol{p}_s$ denote the points of the simplex for which f is largest, next largest, and smallest; that is, because we wish to minimize f, the vertex $\boldsymbol{p}_s$ is the best vertex, $\boldsymbol{p}_l$ is the worst vertex, and $\boldsymbol{p}_{nl}$ is the next-worst vertex.
- We next compute $\boldsymbol{p}_g$, the centroid of the best n points:

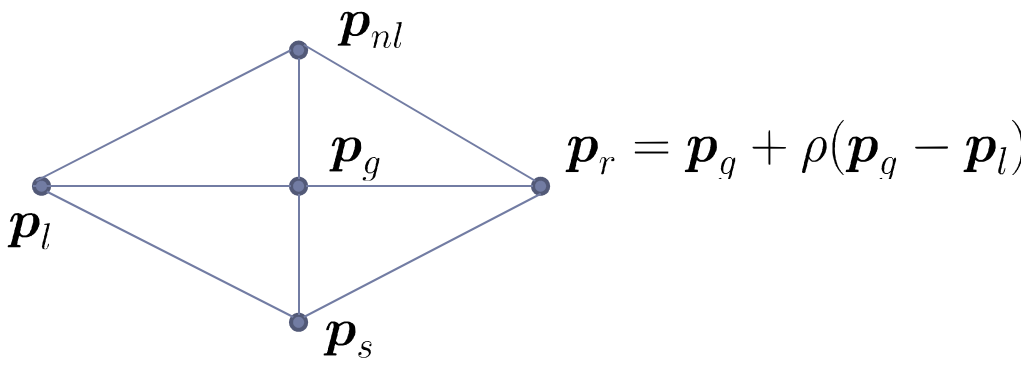
$$\boldsymbol{p}_g = \sum_{i=0}^{n-1} \frac{\boldsymbol{p}_i}{n}$$

  In our two-dim case, $n = 2$, we would have $\boldsymbol{p}_g = \frac{1}{2}(\boldsymbol{p}_{nl} + \boldsymbol{p}_s)$
- We then reflect the worst vertex in $\boldsymbol{p}_g$ using a reflection coefficient $\rho > 0$ to obtain the reflection point

$$\boldsymbol{p}_r = \boldsymbol{p}_g + \rho(\boldsymbol{p}_g - \boldsymbol{p}_l)$$

The Nelder-Mead Simplex Algorithm

- The typical value is $\rho = 1$. We proceed to evaluate f at $\boldsymbol{p}_r$ to obtain $f_r = f(\boldsymbol{p}_r)$. If $f_0 \leq f_r < f_{n-1}$ [i.e., if f_r lies between $f_s = f(\boldsymbol{p}_s)$ and $f_{nl} = f(\boldsymbol{p}_{nl})$], then the point $\boldsymbol{p}_r$ replaces $\boldsymbol{p}_l$ to form a new simplex, and we determine the iteration. (Figure 14.1)
- We proceed to repeat the process. Thus, we compute the centroid of the best n vertices of the new simplex and again reflect the point with the largest function f value in the centroid obtained for the best n points of the new simplex.

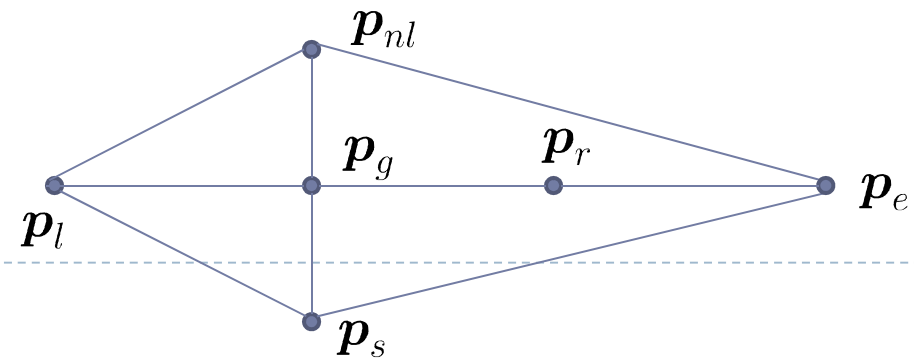


The Nelder-Mead Simplex Algorithm

- If, however, $f_r < f_s = f_0$, so that the point $\boldsymbol{p}_r$ yields the smallest function value among the points of the simplex, we argue that this direction is a good one. In this case we increase the distance traveled using an ***expansion coefficient*** $\chi > 1$ (e.g., $\chi = 2$) to obtain

$$\boldsymbol{p}_e = \boldsymbol{p}_g + \chi(\boldsymbol{p}_r - \boldsymbol{p}_g)$$

- The operation above yields a new point on the line $\boldsymbol{p}_l\boldsymbol{p}_g\boldsymbol{p}_r$ extended beyond $\boldsymbol{p}_r$. If $f_e < f_r$ now, the expansion is declared a success and $\boldsymbol{p}_e$ replaces $\boldsymbol{p}_l$ in the next simplex. If, on the other hand, $f_e \geq f_r$, the expansion is a failure and $\boldsymbol{p}_r$ replaces $\boldsymbol{p}_l$

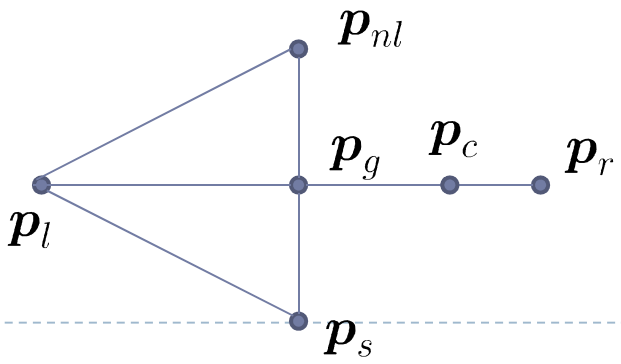


The Nelder-Mead Simplex Algorithm

- Finally, if $f_r \geq f_{nl}$, the reflected point $\boldsymbol{p}_r$ would constitute the point with the largest function value in the new simplex. Then in the next step it would be reflected in $\boldsymbol{p}_g$, probably an unfruitful operation.
- Instead, this case is dealt with by a ***contraction*** operation in one of two ways. First, if $f_r \geq f_{nl}$ and $f_r < f_l$, then we contract $(\boldsymbol{p}_r - \boldsymbol{p}_g)$ with a contraction coefficient $0 < \gamma < 1$ to obtain

$$\boldsymbol{p}_c = \boldsymbol{p}_g + \gamma(\boldsymbol{p}_r - \boldsymbol{p}_g)$$

- We refer to this operation as the ***outside contraction***.

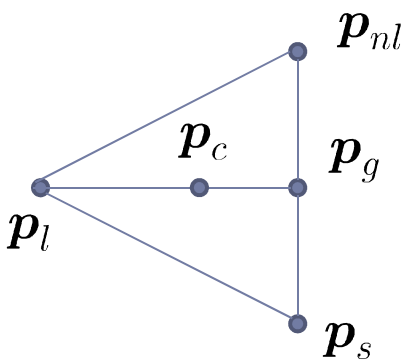


The Nelder-Mead Simplex Algorithm

- If, on the other hand, $f_r \geq f_{nl}$ and $f_r \geq f_l$, then $\boldsymbol{p}_l$ replaces $\boldsymbol{p}_r$ in the contraction operation and we get

$$\boldsymbol{p}_c = \boldsymbol{p}_g + \gamma(\boldsymbol{p}_l - \boldsymbol{p}_g)$$

- This operation, referred to as the ***inside contraction***.

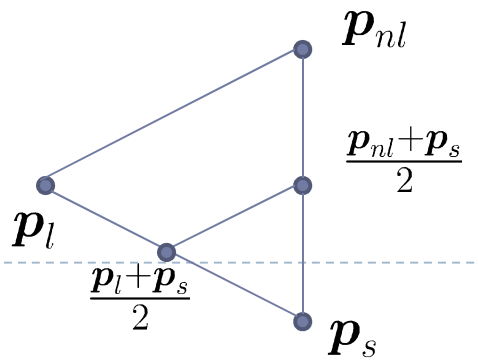


The Nelder-Mead Simplex Algorithm

- If, in either case, $f_c \leq f_l$, the contraction is considered as success, and we replace $\boldsymbol{p}_l$ with $\boldsymbol{p}_c$ in the new simplex. If, however, $f_c > f_l$, the contraction is a failure, and in this case a new simplex can be formed by retaining $\boldsymbol{p}_s$ only and halving the distance from $\boldsymbol{p}_s$ to every other point in the simplex.
- We can refer to this event as a shrinkage operation. In general, the shrink step produces the n new vertices of the new simplex according to the formula

$$\boldsymbol{v}_i = \boldsymbol{p}_s + \sigma(\boldsymbol{p}_i - \boldsymbol{p}_s), i = 1, 2, ..., n$$

where $\sigma = 1/2$. Hence, the vertices of the new simplex are $\boldsymbol{p}_s, \boldsymbol{v}_1, ..., \boldsymbol{v}_n$



The Nelder-Mead Simplex Algorithm

- Figure 14.6 illustrates the simplex search method by showing the first few stages of the search for a minimizer of a function of two variables.
- The starting simplex is composed of the vertices A, B, C. The vertices D, E are obtained by the expansion operation.
- The vertex F is obtained by the reflection operation. The vertex G is obtained using the outside contraction operation, while the vertex I is obtained employing the inside contraction operation.
- For clarity we terminate the process with the simplex composed of the vertices E, H, I.

Simulated Annealing

- Simulated annealing is an instance of a randomized search method. A ***randomized search method***, also called a ***probabilistic search method***, is an algorithm that searches the feasible set of an optimization problem by considering randomized samples of candidate points in the set.
- Suppose that we wish to solve an optimization problem

$$\begin{aligned} &\text{minimize } f(\boldsymbol{x}) \\ &\text{subject to } \boldsymbol{x} \in \Omega \end{aligned}$$

- Typically, we start a randomized search process by selecting a random initial point $\boldsymbol{x}^{(0)} \in \Omega$. Then, we select a random next-candidate point, usually close to $\boldsymbol{x}^{(0)}$

Simulated Annealing

- We assume that for any $\boldsymbol{x} \in \Omega$, there is a set $N(\boldsymbol{x}) \subset \Omega$ such that we can generate a random sample from this set. Typically, $N(\boldsymbol{x})$ is a set of points that are "close" to $\boldsymbol{x}$, and for this reason we usually think of $N(\boldsymbol{x})$ as a "neighborhood" of $\boldsymbol{x}$.
- When speaking of generating a random point in $N(\boldsymbol{x})$, we mean that there is a prespecified distribution over $N(\boldsymbol{x})$, and we sample a point with this distribution. Often, this distribution is chosen to be uniform over $N(\boldsymbol{x})$; other distributions are often used, including Gaussian and Cauchy.

Naïve Random Search

- Naïve random search algorithm
  - 1. Set $k := 0$. Select an initial point $\boldsymbol{x}^{(0)} \in \Omega$
  - 2. Pick a candidate point $\boldsymbol{z}^{(k)}$ at random from $N(\boldsymbol{x}^{(k)})$
  - 3. If $f(\boldsymbol{z}^{(k)}) < f(\boldsymbol{x}^{(k)})$, then set $\boldsymbol{x}^{(k+1)} = \boldsymbol{z}^{(k)}$; else, set $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)}$
  - 4. If stopping criterion satisfied, then stop.
  - 5. Set $k := k + 1$, go to step 2.
- Note that the algorithm has the familiar form $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \boldsymbol{d}^{(k)}$ where $\boldsymbol{d}^{(k)}$ is randomly generated. By design, the direction $\boldsymbol{d}^{(k)}$ either is $\mathbf{0}$ or is a descent direction. Typical stopping criteria include reaching a certain number of iterations, or reaching a certain objective function value.

Simulated Annealing Algorithm

- The main problem of the random search method is that it may get stuck in a region around a local minimizer. For example, if $\boldsymbol{x}^{(0)}$ is a local minimizer and $N(\boldsymbol{x}^{(0)})$ is sufficiently small that all points in it have no smaller objective function value than $\boldsymbol{x}^{(0)}$, then clearly the algorithm will be stuck and will never find a point outside of $N(\boldsymbol{x}^{(0)})$.
- We need to consider points outside this region. One way to achieve this goal is to make sure that at each k, the neighborhood $N(\boldsymbol{x}^{(k)})$ is a very large set. An extreme example is where $N(\boldsymbol{x}^{(k)}) = \Omega$. However, this results in slow search process, because the sampling of candidate points to consider is spread out, making it more unlikely to find a better candidate point.

Simulated Annealing Algorithm

- Another way is to modify the naïve search algorithm so that we can "climb out" of such as region. This means that the algorithm may accept a new point that is ***worse*** than the current point.
- Simulated annealing algorithm
  - 1. Set $k := 0$. Select an initial point $\boldsymbol{x}^{(0)} \in \Omega$
  - 2. Pick a candidate point $\boldsymbol{z}^{(k)}$ at random from $N(\boldsymbol{x}^{(k)})$
  - 3. Toss a coin with probability of HEAD equal to $p(k, f(\boldsymbol{z}^{(k)}), f(\boldsymbol{x}^{(k)}))$ If HEAD, then set $\boldsymbol{x}^{(k+1)} = \boldsymbol{z}^{(k)}$; else, set $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)}$
  - 4. If stopping criterion satisfied, then stop.
  - 5. Set $k := k + 1$, go to step 2.

Simulated Annealing Algorithm

- The simulated anneal algorithm also has the familiar form $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \boldsymbol{d}^{(k)}$, where $\boldsymbol{d}^{(k)}$ is randomly generated. But in simulated annealing the direction $\boldsymbol{d}^{(k)}$ might be an ascent direction. However, as the algorithm progresses, we can keep track of the ***best-so-far point*** – that is a point $\boldsymbol{x}_{\text{best}}^{(k)}$ which, at each k , is equal to a $\boldsymbol{x}^{(j)}$, $j \in \{0, ..., k\}$, such that $f(\boldsymbol{x}^{(j)}) \leq f(\boldsymbol{x}^{(i)})$ for all $i \in \{0, ..., k\}$.
- The best-so-far point can be updated at each step k as follows:

$$\boldsymbol{x}_{best}^{(k)} = \begin{cases} \boldsymbol{x}^{(k)} & \text{if } f(\boldsymbol{x}^{(k)}) < f(\boldsymbol{x}_{best}^{(k-1)}) \\ \boldsymbol{x}_{best}^{(k-1)} & \text{otherwise} \end{cases}$$

Simulated Annealing Algorithm

- By keeping track of the best-so-far point, we can treat the simulated annealing algorithm simply as a search procedure; the best-so-far point is what we eventually use when the algorithm stops.
- The major difference between simulated annealing and naïve random search is that in step 3, there is some probability that we set the next iterate to be equal to the random point selected from the neighborhood, even if that point turns out to be worse than the current iterate. This probability is called the ***acceptance probability***.

Simulated Annealing Algorithm

- The acceptance probability must be chosen appropriately. A typical choice is

$$p(k, f(\boldsymbol{z}^{(k)}), f(\boldsymbol{x}^{(k)})) = \min\{1, \exp(-(f(\boldsymbol{z}^{(k)}) - f(\boldsymbol{x}^{(k)}))/T_k)\}$$

  where $\exp$ is the exponential function and T_k represents a positive sequence called the ***temperature schedule*** or ***cooling schedule***.

- Notice that if $f(\boldsymbol{z}^{(k)}) \leq f(\boldsymbol{x}^{(k)})$, then $p(k, f(\boldsymbol{z}^{(k)}), f(\boldsymbol{x}^{(k)})) = 1$, which means that we set $\boldsymbol{x}^{(k+1)} = \boldsymbol{z}^{(k)}$. However, if $f(\boldsymbol{z}^{(k)}) > f(\boldsymbol{x}^{(k)})$ there is still a positive probability of setting $\boldsymbol{x}^{(k+1)} = \boldsymbol{z}^{(k)}$; this probability is equal to

$$\exp\left(-\frac{f(\boldsymbol{z}^{(k)}) - f(\boldsymbol{x}^{(k)})}{T_k}\right)$$

Simulated Annealing Algorithm

- Note that the larger the difference between $f(\boldsymbol{z}^{(k)})$ and $f(\boldsymbol{x}^{(k)})$, the less likely we are to move to the worse point $\boldsymbol{z}^{(k)}$. Similarly, the smaller the value of T_k, the less likely we are to move to $\boldsymbol{z}^{(k)}$
- It is typical to let the "temperature" T_k be monotonically decreasing to 0 (hence the word ***cooling***). In other words, as the iteration index k increases, the algorithm becomes increasingly reluctant to move to a worse point.
- The intuitive reason for this behavior is that initially we wish to actively explore the feasible set, but with time we would like to be less active in exploration so that we spend more time in a region around a global minimizer.

Simulated Annealing Algorithm

- The term ***annealing*** comes from the field of metallurgy, where it refers to a technique for improving the property of metals. The basic procedure is to heat up a piece of metal and then cool it down in a controlled fashion. When the metal is first heated, the atoms in it become unstuck from their initial positions. Then, as cooling takes place, the atoms gradually configure themselves in states of lower internal energy. Provided that the cooling is sufficiently slow, the final internal energy is lower than the initial energy, thereby refining the crystalline structure and reducing defects.

Simulated Annealing Algorithm

- Hajek shows that an appropriate cooling schedule is

$$T_k = \frac{\gamma}{\log(k+2)}$$

  where $\gamma > 0$ is a problem-dependent constant (large enough to allow the algorithm to "climb out" of regions around local minimizers that are not global minimizers).

- Simulated annealing is often also used in combinatorial optimization, where the feasible set is finite. An example is the celebrated ***traveling salesperson problem***.

Particle Swarm Optimization

- This optimization method is inspired by social interaction principles. The PSO algorithm differs from the randomized search methods in one key way: Instead of updating a single candidate solution $\boldsymbol{x}^{(k)}$ at each iteration, we update a ***population*** (set) of candidate solutions, called a ***swarm***. Each candidate solution in the swarm is called a ***particle***.
- We think of a swarm as an apparently disorganized population of moving individuals that tend to cluster together while each individual seems to be moving in a random direction.

Particle Swarm Optimization

- Suppose that we wish to minimize an objective function over R^n In the PSO algorithm, we start with an initial randomly generated population of points in R^n . Associated with each point in the population is a velocity vector. We think of each point as the position of a particle, moving with an associated velocity.
- We then evaluate the objective function at each point in the population. Based on this evaluation, we create a new population of points together with a new set of velocities.

Particle Swarm Optimization

- Each particle keeps track of its ***best-so-far position***. That is the best position it has visited so far. We will call it ***personal best*** (*pbest*). In contrast, the overall best-so-far position is called a ***global best*** (*gbest*).
- The particles “interact” with each other by updating their velocities according to their individual personal best as well as the global best. In the *gbest* version of the PSO algorithm, the velocity of each particle is changed, at each time step, toward a combination of its *pbest* and the *gbest* locations.
- Typical stopping criteria of the algorithm consist of reaching a certain number of iterations, or reaching a certain objective function value.

Basic PSO Algorithm

- Let $f : R^n \to R$ be the objective function that we wish to minimize. Let d be the population size, and index the particles in the swarm by $i = 1, ..., d$. Denote the position of particle i by $\boldsymbol{x}_i \in R^n$ and its velocity by $\boldsymbol{v}_i \in R^n$. Let $\boldsymbol{p}_i$ be the *pbest* of particle i and $\boldsymbol{g}$ the best.
- It is convenient to introduce the ***Hadamard product*** (or ***Schur product***) operator, denoted by $\circ$. If $\boldsymbol{A}$ and $\boldsymbol{B}$ are matrices with the same dimension, then $\boldsymbol{A} \circ \boldsymbol{B}$ is a matrix of the same dimension as $\boldsymbol{A}$ resulting from entry-by-entry multiplication of $\boldsymbol{A}$ and $\boldsymbol{B}$.

Basic PSO Algorithm

- 1. Set $k := 0$. For $i = 1, ..., d$, generate initial random positions $\boldsymbol{x}_i^{(0)}$ and velocities $\boldsymbol{v}_i^{(0)}$, and set $\boldsymbol{p}_i^{(0)} = \boldsymbol{x}_i^{(0)}$. Set $\boldsymbol{g}^{(0)} = \arg\min_{\boldsymbol{x} \in \{\boldsymbol{x}_1^{(0)}, ..., \boldsymbol{x}_d^{(0)}\}} f(\boldsymbol{x})$
- 2. For $i = 1, ..., d$, generate random n-vectors $\boldsymbol{r}_i^{(k)}$ and $\boldsymbol{s}_i^{(k)}$ with components uniformly in the interval (0, 1), and set

$$\boldsymbol{v}_i^{(k+1)} = \omega \boldsymbol{v}_i^{(k)} + c_1 \boldsymbol{r}_i^{(k)} \circ (\boldsymbol{p}_i^{(k)} - \boldsymbol{x}_i^{(k)}) + c_2 \boldsymbol{s}_i^{(k)} \circ (\boldsymbol{g}^{(k)} - \boldsymbol{x}_i^{(k)})$$
$$\boldsymbol{x}_i^{(k+1)} = \boldsymbol{x}_i^{(k)} + \boldsymbol{v}_i^{(k+1)}$$

- 3. For $i = 1, ..., d$, if $f(\boldsymbol{x}_i^{(k+1)}) < f(\boldsymbol{p}_i^{(k)})$, then set $\boldsymbol{p}_i^{(k+1)} = \boldsymbol{x}_i^{(k+1)}$; else, set $\boldsymbol{p}_i^{(k+1)} = \boldsymbol{p}_i^{(k)}$
- 4. If there exists $i \in \{1, ..., d\}$ such that $f(\boldsymbol{x}_i^{(k+1)}) < f(\boldsymbol{g}_i^{(k)})$, then set $\boldsymbol{g}^{(k+1)} = \boldsymbol{x}_i^{(k+1)}$; else, set $\boldsymbol{g}^{(k+1)} = \boldsymbol{g}^{(k)}$
- 5. If stopping criterion satisfied, then stop.
- 6. Set $k := k + 1$, go to step 2.

Basic PSO Algorithm

- The parameter ω is referred to as an ***inertial constant***. Recommended values are slightly less than 1. The parameters c_1 and c_2 are constants that determine how much the particle is directed toward "good" positions. They represent a "cognitive" and a "social" component, respectively, in that they affect how much the particle's person best and the global best influence its movement. Recommended values are $c_1, c_2 \approx 2$.

Variations

- The PSO techniques have evolved since 1995. Recently Clerc proposed a ***constriction-factor*** version of the algorithm, where the velocity is updated as

$$\boldsymbol{v}_i^{(k+1)} = \kappa\left(\boldsymbol{v}_i^{(k)} + c_1\boldsymbol{r}_i^{(k)} \circ (\boldsymbol{p}_i^{(k)} - \boldsymbol{x}_i^{(k)}) + c_2\boldsymbol{s}_i^{(k)} \circ (\boldsymbol{g}^{(k)} - \boldsymbol{x}_i^{(k)})\right)$$

  where the constriction coefficient κ is computed as

$$\kappa = \frac{2}{|2 - \phi - \sqrt{\phi^2 - 4\phi}|} \qquad \begin{array}{l}\phi = c_1 + c_2 \\ \phi > 4\end{array}$$

- For example, for $\phi = 4.1$, we have $\kappa = 0.729$. The role of the constriction coefficient is to speed up the convergence.

Genetic Algorithms

- A ***genetic algorithm*** is a randomized, population-based search technique that has its roots in the principles of genetics.
- Suppose that we wish to solve an optimization problem of the form

$$\begin{aligned} &\text{maximize } f(\boldsymbol{x}) \\ &\text{subject to } \boldsymbol{x} \in \Omega \end{aligned}$$

- We start with an initial set of points in Ω, denoted by $P(0)$, called the ***initial population***. We then evaluate the objective function at points in $P(0)$. Based on this evaluation, we create a new set of points $P(1)$. The creation of $P(1)$ involves certain operations on points in $P(0)$, called ***crossover*** and ***mutation***. We repeat the procedure iteratively, generating populations $P(2), P(3), \ldots$, until an appropriate stopping criterion is reached.

Genetic Algorithms

- The purpose of the crossover and mutation operations is to create a new population with an average objective function value that is higher than that of the previous population.
- Genetic algorithms do not work directly with points in the set Ω but rather with an ***encoding*** of the points in Ω . Specifically, we need first to map Ω onto a set consisting of strings of symbols, all of equal length. These strings are called ***chromosomes***. Each chromosome consists of elements from a chosen set of symbols, called the ***alphabet***. For example, a common alphabet is the set {0.1}, in which case the chromosomes are simply binary strings.

Genetic Algorithms

- We denote by L the length of chromosomes (i.e., the number of symbols in the strings). To each chromosome there corresponds a value of the objective function, referred to as the ***fitness*** of the chromosome.
- For each chromosome x, we write $f(x)$ for its fitness. We assume that f is a nonnegative function.
- The choice of chromosome length, alphabet, and encoding is called the ***representation scheme*** for the problem. Identification of an appropriate representation scheme is the first step in using genetic algorithms.

Genetic Algorithms

- Once a suitable representation scheme has been chosen, the next phase is to initialize the first population $P(0)$ of chromosomes. This is usually done by a random selection of a set of chromosomes.
- We then apply the operations of crossover and mutation on the population. During each iteration k of the process, we evaluate the fitness $f(\boldsymbol{x}^{(k)})$ of each member $\boldsymbol{x}^{(k)}$ of the population $P(k)$. After the fitness of the entire population has been evaluated, we form a new population $P(k+1)$ in two stages.

Selection and Evolution

- We form a set $M(k)$ with the same number of elements as $P(k)$. This number is called the ***population size***, which we denote by N. The set $M(k)$, called the ***mating pool***, is formed from $P(k)$ using a random procedure as follows.
- Each point $\boldsymbol{m}^{(k)}$ in $M(k)$ is equal to $\boldsymbol{x}^{(k)}$ in $P(k)$ with probability

$$\frac{f(\boldsymbol{x}^{(k)})}{F(k)}$$

where $F(k) = \sum f(\boldsymbol{x}_i^{(k)})$
and the sum is taken over the whole of $P(k)$. In other words, we select chromosomes into the mating pool with probabilities proportional to their fitness.
- The selection scheme is called the ***roulette-wheel scheme***.

Selection and Evolution

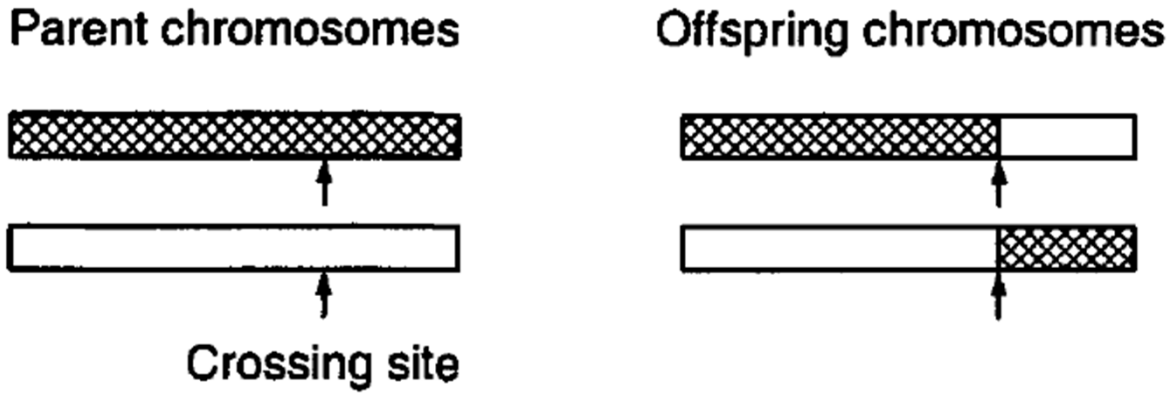
- An alternative selection scheme is the ***tournament scheme***.
- First, we select a pair of chromosomes at random from $P(k)$. We then compare the fitness values of these two chromosomes, and place the fitter of the two into $M(k)$. We repeat this operation until the mating pool $M(k)$ contains N chromosomes.
- The ***crossover operation*** takes a pair of chromosomes, called the ***parents***, and gives a pair of ***offspring chromosomes***. The operation involves exchanging substrings of the two parent chromosomes.

Selection and Evolution

- Pairs of parents for crossover are chosen from the mating pool randomly, such that the probability that a chromosome is chosen for crossover is p_c . We assume that whether or not a given chromosome is chosen is independent of whether or not any other chromosome is chosen for crossover.
- We may randomly choose two chromosomes from the mating pool as parents. If N is the size of the mating pool, then $p_c = 2/N$ Similarly, if we randomly pick $2k$ chromosomes, forming k pairs of parents, we have $p_c = 2k/N$
- Another way is, given a value of p_c, we pick a random number of pairs of parents such that the average number of pairs is $p_c N/2$

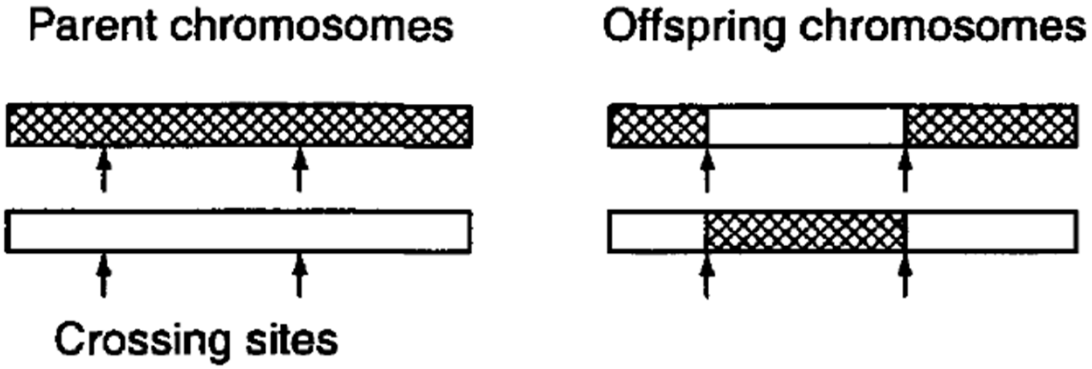
Selection and Evolution

- We apply the crossover operation to the parents. There are many types of crossover operations. The simplest crossover operation is the ***one-point crossover***. We first choose a number randomly between 1 and $L-1$ according to a uniform distribution, where L is the length of chromosomes. We refer to this number as the ***crossing site***. Crossover then involves exchanging substrings of the parents to the left of the crossing site.



Selection and Evolution

- We can also have crossover operations with multiple crossing sites.



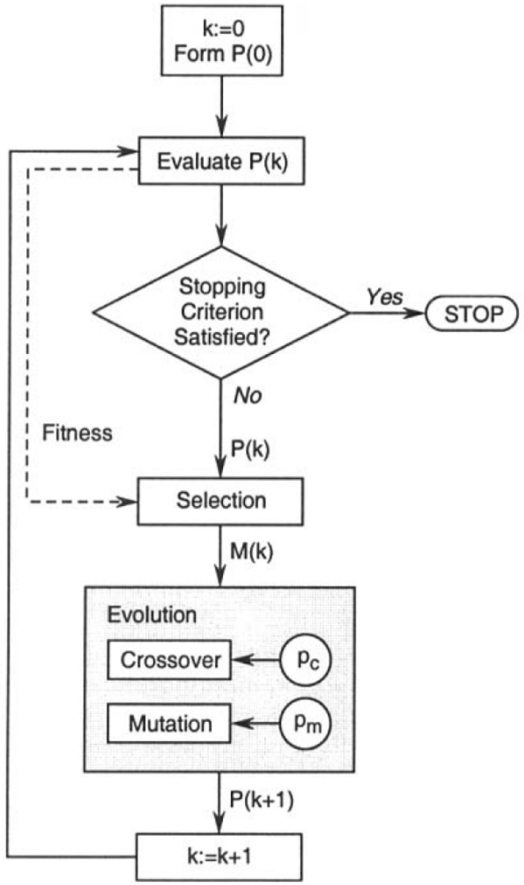
- After the crossover operation, we replace the parents in the mating pool by their offspring. The mating pool has therefore been modified but maintains the same number of elements.

Selection and Evolution

- Next, we apply the ***mutation*** operation, which takes each chromosome from the mating pool and randomly changes each symbol of the chromosome with a given probability p_m
- In the case of binary alphabet, this change corresponds to complementing the corresponding bits. If the alphabet contains more than two symbols, then the change involves randomly substituting the symbol with another symbol from the alphabet.
- Typically, the value of p_m is very small (e.g., 0.01), so that only a few chromosomes will undergo a change due to mutation.

Genetic Algorithm

- 1. Set $k := 0$. Generate an initial population $P(0)$
- 2. Evaluate $P(k)$
- 3. If the stopping criterion is satisfied, then stop.
- 4. Select $M(k)$ from $P(k)$
- 5. Evolve $M(k)$ to form $P(k+1)$
- 6. Set $k := k + 1$, go to step 2.



Genetic Algorithm

- During execution of the genetic algorithm, we keep track of the ***best-so-far*** chromosome, which serves as the candidate for the solution to the original problem. We may even copy the best-so-far chromosome into each new population, a practice referred to as ***elitism***.
- The stopping criterion can be implemented in a number of ways. For example, stop after a prespecified number of iterations, or stop when the fitness for the best-so-far chromosome does not change significantly.

Genetic Algorithm

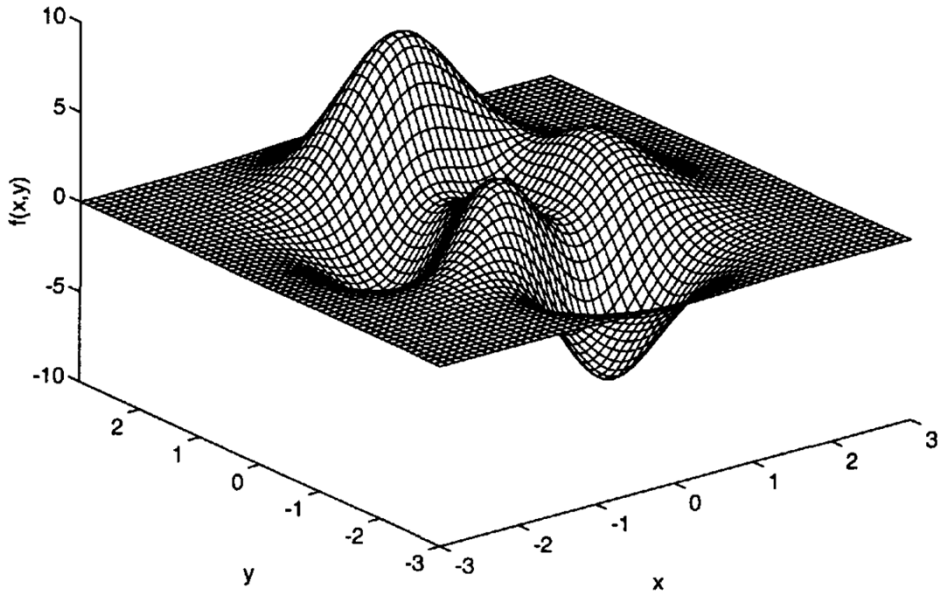
- The genetic algorithm differs from the algorithms discussed in previous chapters in several respects
  - 1. It does not use derivatives of the objective function
  - 2. It uses operations that are random within each iteration
  - 3. It searches from a set of points rather than a single point at each iteration (like the PSO algorithm)
  - 4. It works with an encoding of the feasible set rather with than the set itself

Example

- Consider the MATLAB peaks function $f : R^2 \to R$

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10(\tfrac{x}{5} - x^3 - y^5)e^{-x^2-y^2} - \frac{e^{-(x+1)^2-y^2}}{3}$$

We wish to maximize f over the set $\Omega = \{[x,y]^T \in R^2 : -3 \le x, y \le 3\}$ Using the MATLAB function fminunc (from the Optimization Toolbox), we found the optimal point to be $[-0.0093, 1.5814]^T$, with objective function value 8.1062.



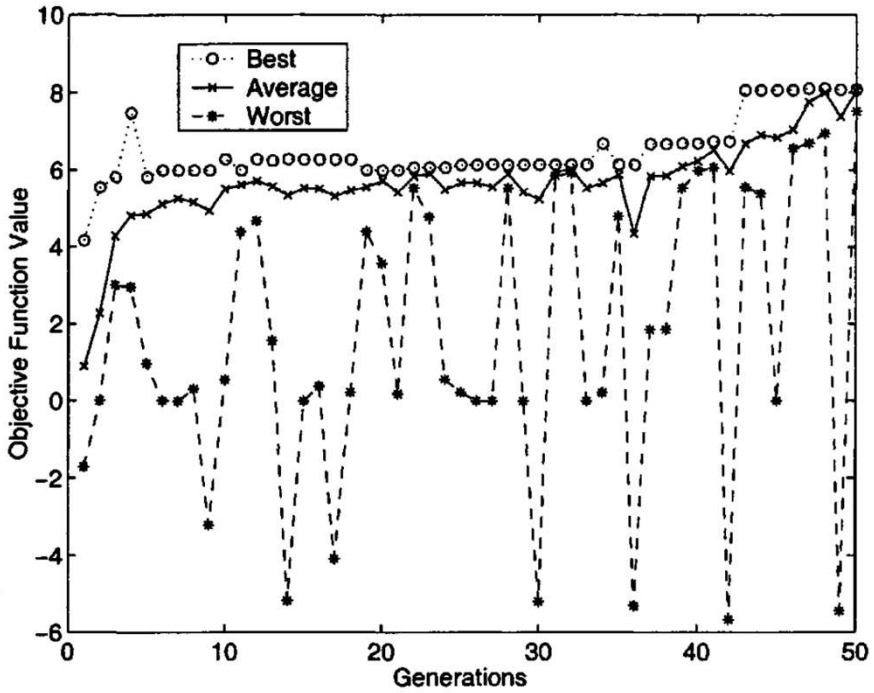
Example

- We use a binary representation scheme with length L=32, where the first 16 bits encode the x component, whereas the remaining 16 bits encode the y component. We first map the interval [-3, 3] onto the interval $[0, 2^{16} - 1]$, via a simple translation and scaling. The integers in the interval $[0, 2^{16} - 1]$ are then expressed as binary 16-bit strings.
- The chromosome is obtained by juxtaposing the two 8-bit strings. For example, the point $[x, y]^T = [-1, 3]^T$ is encoded as

$$\underbrace{0101010101010101}_{\text{Encoded } x = -1}\underbrace{1111111111111111}_{\text{Encoded } y = 3}$$

Example

- Using a population size of 20, we apply 50 iterations of the genetic algorithm. We used values of $p_c = 0.75$ and $p_m = 0.0075$
- The best-so-far solution obtained at the end of the 50 iterations is [0.0615, 1.5827], with objective function value 8.1013. Note that his solution and objective function value are very close to those obtained using MATLAB.



Analysis of Genetic Algorithms

- For convenience, we only consider chromosomes over the binary alphabet. The notion ***schema*** is a set of chromosomes with certain common features. For example, the notion 1*01 represents the schema

$$1 * 01 = \{1001, 1101\}$$

and the notation 0*101* represents the schema

$$0 * 101 * = \{001010, 001011, 011010, 011011\}$$

Thus, a schema describes a set of chromosomes that have certain specified similarities.

- If a schema has r "don't care" symbols, then it contains 2^r chromosomes. Moreover, any chromosome of length L belongs to 2^L schemata.

Analysis of Genetic Algorithms

- Given a schema that represents good solutions to our optimization problem, we would like the number of matching chromosomes in the population $P(k)$ to grow as k increases. This growth is affected by several factors. We assume throughout that we are using the roulette-wheel selection method.
- If a schema has chromosomes with better-than-average fitness, then the expected (mean) number of chromosomes matching this schema in the mating pool $M(k)$ is larger than the number of chromosomes matching this schema in the population $P(k)$

Analysis of Genetic Algorithms

- To quantify this assertion, let H be a given schema, and let $e(H,k)$ be the number of chromosomes in $P(k)$ that match H; that is, $e(H,k)$ is the number of elements in the set $P(k) \bigcap H$
- Let $f(H,k)$ be the average fitness of chromosomes in $P(k)$ that match schema H. This means that if $H \bigcap P(k) = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_{e(H,k)}\}$

$$f(H,k) = \frac{f(\boldsymbol{x}_1) + \cdots + f(\boldsymbol{x}_{e(H,k)})}{e(H,k)}$$

- Let N be the number of chromosomes in the population and $F(k)$ be the sum of the fitness values of chromosomes in $P(k)$. Denote by $\bar{F}(k)$ the average fitness of chromosomes in the population

$$\bar{F}(k) = F(k)/N = \frac{1}{N}\sum f(\boldsymbol{x}_i^{(k)})$$

Analysis of Genetic Algorithms

- Let $m(H,k)$ be the number of chromosomes in $M(k)$ that match H in other words, the number of elements in the set $M(k) \bigcap H$
- Lemma 14.1: Let H be a given schema and $M(H,k)$ be the expected value of $m(H,k)$ given $P(k)$, then

$$M(H,k) = \frac{f(H,k)}{\bar{F}(k)} e(H,k)$$

- This lemma quantifies that if a schema H has chromosomes with better than average fitness, i.e., $f(H,k)/\bar{F}(k) > 1$, then the expected number of chromosomes matching H in the mating pool is larger than the number of chromosomes matching H in the population.

Analysis of Genetic Algorithms

- We now analyze the effect of the evolution operations on the chromosomes in the mating pool. The ***order*** $o(S)$ of a schema S is the number of fixed symbols in its representation. If the ***length*** of chromosomes in S is L, then $o(S)$ is L minus the number of * symbols in S . For example,

$$o(1*01) = 4 - 1 = 3 \qquad o(0*1*01) = 6 - 2 = 4$$

- The ***length*** $l(S)$ of a schema S is the distance between the first and last fixed symbols.

$$l(1*01) = 4 - 1 = 3 \qquad l(0*101*) = 5 - 1 = 4 \qquad l(**1*) = 0$$

Analysis of Genetic Algorithms

- The order $o(S)$ is a number between 0 and L, and the length $l(S)$ is a number between 0 and $L-1$. The order of a schema containing no * symbols is L, e.g., $o(1011) = 4 - 0 = 4$. The length of a schema with fixed symbols in its first and last positions is $L-1$, e.g., $l(0**1) = 4 - 1 = 3$
- Given a chromosome in $M(k) \bigcap H$, the probability that it leaves H after crossover is bounded above by a quantity that is proportional to p_c and $l(H)$
- Lemma 14.2: Given a chromosome in $M(k) \bigcap H$, the probability that it is chosen for crossover and neither of its offspring is in H is bounded above by

$$p_c \frac{l(H)}{L-1}$$

Analysis of Genetic Algorithms

- From Lemma 14.2, we conclude that given a chromosome in $M(k) \bigcap H$, the probability that either it is not selected for crossover or that at least one of its offspring is in H after the crossover operation, is bounded below by

$$1 - p_c \frac{l(H)}{L-1}$$

- Lemma 14.3: Given a chromosome in $M(k) \bigcap H$, the probability that it remains in H after the mutation operation is given by

$$(1 - p_m)^{o(H)} \approx 1 - p_m o(H)$$

Analysis of Genetic Algorithms

- Theorem 14.1: Let H be a given schema and $\mathcal{E}(H, k+1)$ be the expected value of $e(H, k+1)$ given $P(k)$, then

$$\mathcal{E}(H, k+1) \geq \left(1 - p_c \frac{l(H)}{L-1}\right)(1-p_m)^{o(H)} \frac{f(H,k)}{\bar{F}(k)} e(H,k)$$

- Theorem 14.1 indicates how the number of chromosomes in a given schema changes from one population to the next.
  - 1. the role of average fitness of the given schema – the higher the average fitness, the higher the expected number of matches in the next population.
  - 2. the effect of crossover – the smaller the term, the higher the expected number of matches in the next population
  - 3. the effect of mutation – the larger the term, the higher the expected number of matches in the next population

Analysis of Genetic Algorithms

- In summary, a schema that is short, low order, and has above-average fitness will have on average an increasing number of its representatives in the population from iteration to iteration.
- Observe that the encoding is relevant to the performance of the algorithm. Specifically, a good encoding is one that results in high-fitness schemata having small lengths and orders.

Real-Number Genetic Algorithms

- The genetic algorithms described thus far operate on binary strings, representing elements of the feasible set Ω. However, there are some disadvantages to operating on binary strings. To see this, let $\boldsymbol{g} : \{0,1\}^L \to \Omega$ represent the binary "decoding" function; that is, if $\boldsymbol{x}$ is a binary chromosome, $\boldsymbol{g}(\boldsymbol{x}) \in \Omega$ is the point in the feasible set $\Omega \in R^n$ whose encoding is $\boldsymbol{x}$. Therefore, the objective function being maximized by the genetic algorithm is not f itself but rather the composition of f and the decoding function $\boldsymbol{g}$. In other words, the optimization problem being solved is

$$\begin{aligned} &\text{maximize} && f(\boldsymbol{g}(\boldsymbol{x})) \\ &\text{subject to} && \boldsymbol{x} \in \{\boldsymbol{y} \in \{0,1\}^L : \boldsymbol{g}(\boldsymbol{y}) \in \Omega\} \end{aligned}$$

Real-Number Genetic Algorithms

- This optimization problem may be more complex than the original optimization problem. For example, it may have extra maximizers, making the search for a global maximizer more difficult.
- In real-number algorithms, for crossover, we have several options. The simplest is to use averaging: for a pair of parents $\boldsymbol{x}$ and $\boldsymbol{y}$, the offspring is $\boldsymbol{z} = (\boldsymbol{x} + \boldsymbol{y})/2$. This offspring can then replace one of the parents.
- Alternatively, we may produce two offspring as follows
  $\boldsymbol{z}_1 = (\boldsymbol{x} + \boldsymbol{y})/2 + \boldsymbol{w}_1 \quad \boldsymbol{z}_2 = (\boldsymbol{x} + \boldsymbol{y})/2 + \boldsymbol{w}_2$
  where $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ are two randomly generated vectors (with zero mean).

Real-Number Genetic Algorithms

- A third option is to take random convex combinations of the parents. Specifically, we generate a random number $\alpha \in (0, 1)$ and then produce two offspring $\boldsymbol{z}_1 = \alpha\boldsymbol{x} + (1 - \alpha)\boldsymbol{y}$ and $\boldsymbol{z}_2 = (1 - \alpha)\boldsymbol{x} + \alpha\boldsymbol{y}$
- A fourth option is the perturb the two points by some random amount: $\boldsymbol{z}_1 = \alpha\boldsymbol{x} + (1 - \alpha)\boldsymbol{y} + \boldsymbol{w}_1 \qquad \boldsymbol{z}_2 = (1 - \alpha)\boldsymbol{x} + \alpha\boldsymbol{y} + \boldsymbol{w}_2$
- For mutation, a simple implementation is to add a random vector to the chromosome. Specifically, given a chromosome $\boldsymbol{x}$ we produce this mutation as $\boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{w}$. This mutation is also called a ***real number creep***.
- An alternative: $\boldsymbol{x}' = \alpha\boldsymbol{x} + (1 - \alpha)\boldsymbol{w} \qquad \alpha \in (0, 1)$ $\boldsymbol{w} \in \Omega$

Example

- Consider again the previous example. We apply a real-number genetic algorithm to find a maximizer of f using a crossover operation of the fourth type described above and a mutation operation of the second type above.
- With a population size of 20, we apply 50 iterations. As before, we used parameter values of $p_c = 0.75$ and $p_m = 0.0075$. The best-so-far solution obtained at the end of the 50 iterations is $[-0.0096, 1.5845]^T$, with objective function value 8.1061, which is close to the result described previously.