6. Global Search Optimization Algorithms.

6.1. Introduction

The gradient-based numerical optimization algorithms that we discussed in Chapter 5 are local search algorithms where a design is continuously improved/updated based on derivatives. For differentiable functions, it has been shown that even if the derivatives are calculated by finite difference methods, gradient-based algorithms are better than ones that do not use derivatives. However, gradient-based algorithms are limited to continuously differentiable functions. If the objective function or constraint functions are not smooth or have some discontinuities, we have to rely on gradient-free optimization algorithms.

Another challenge of gradient-based algorithms is that they find local optimum designs. Since design moves from an initial design to an improved design, the local search algorithms tend to converge the local optimum that is close to the initial design. Therefore, different initial designs may end up different local optima. Indeed, it is often suggested to repeat the gradient-based optimization algorithms with different initial designs in order to increase the chance of finding a global optimum design. However, the lack of capability in exploring the entire design space is considered as the major drawback of gradient-based optimization algorithms.

In order to overcome the requirements of smoothness and local optima, gradient-free optimization algorithms have gained a lot of attention recently. These algorithms are relatively easy to implement as they do not require gradient or Hessian information and the line search for step-size determination. In addition, since they explore the entire design space, they have a better chance to find the global optimum. However, most gradient-free algorithms are computational expensive as they require thousands of function evaluations. In addition, these algorithms often come with many parameters to be tuned and the performance strongly depends on these parameters. Therefore, an algorithm that works great for one optimization problem may not work well for other problems. Typically, these algorithms are developed based on some physical phenomena. Some examples are genetic algorithms, simulated annealing algorithm, and particle swarm optimization.

In the literature, these algorithms are often referred to as global optimization algorithms, but they are referred to as global search algorithms in this text. This is because there is no guarantee to find the global optimum design. They are indeed a structured random search, which increases the chance of finding global or near global optimum designs. These methods tend to be numerically robust as they generate a population of design and choose the best one. An important aspect of these group of algorithms is that they can provide a number of good designs instead of a single optimum design.

Some global search algorithms are deterministic, while most others are probabilistic. Nelder-Mead sequential simplex algorithm in Section 6.2 and the DIRECT method in Section 6.3 are two examples of deterministic algorithms. These algorithms are deterministic in the sense that the same optimum design can be found when the algorithm is used with the same initial condition. However, they still search globally to find the best design. Most other algorithms are stochastic in their nature where different results are expected with the algorithm is repeated. These algorithms include the genetic algorithm in Section 6.4, the particle swarm algorithm in Section 6.5, and simulated annealing algorithm in Section 6.6.

Depending on how they generate population and not get trapped in local optima, probabilistic optimization algorithms can be categorized into three groups. The first group is based on pure random search, including the hill-climbing and taboo search algorithms. These algorithms essectially jump around the design space and accept designs with some probability. The second group is based on evolutionary algorithms, such as the genetic algorithm that try to imitate biological evolution. The last one tries to

imitate swarm intelligence, particle swarm optimization, which will cover essentially imitates birds and bees while ant colony optimization imitates ants.

The major bottleneck of global search algorithms is computational cost. The cost of local optimization algorithms usually increases somewhere between linearly and quadratically proportional to the number of variables. However, the global search algorithms are often considered as NP-hard [53], which means that as the dimension of a problem increases, the cost of solving it increases faster than any polynomial orders, which means that the computational cost increases exponentially.

Another interesting observation in global search algorithms is no-free-lunch theorem [54], where for certain types of mathematical problems, the computational cost of finding a solution, averaged over all problems in the class, is the same for any solution method. In global optimization, it means that no single algorithm will perform well on all problems. The performance of global search algorithms changes significantly by changing their parameters. Therefore, it is possible to make an algorithm perform well for a specific problem, but it may not work efficiently for other problems. This is a great opportunity for engineers who can use problem specific knowledge to tailor algorithms to their problems, something that they can rarely do for local optimization algorithms. However, it is difficult to claim that one algorithm is better than the other.

The complexity of a function can be categorized in the perspective of difficulty associated with finding the global optimum. Thomas Weise [55] illustrates different characteristics of optimization problems of increasing challenge. The function in Figure 6-1(a) shows a very benign function, where local minima are very shallow and most local optimization algorithms may be able to find the global minimum. Figure 6-1(b) shows many significant local optima, where it is likely that gradient-based algorithms may find the closest local optimum from the intial design. However, a good global algorithm may find the global optimum in such a problem. Gradient-based algorithm can still find the global optimum by repeating the algorithm with different initial design. Figure 6-1(c) is a challenging problem in the sense that most optimization algorithms may trap in the wide flat region as we do not know the functional form of the entire design space. In Figure 6-1(d), the global optimum is hidden in a narrow region. Even a good global search optimizer may take forever to find the optimum.

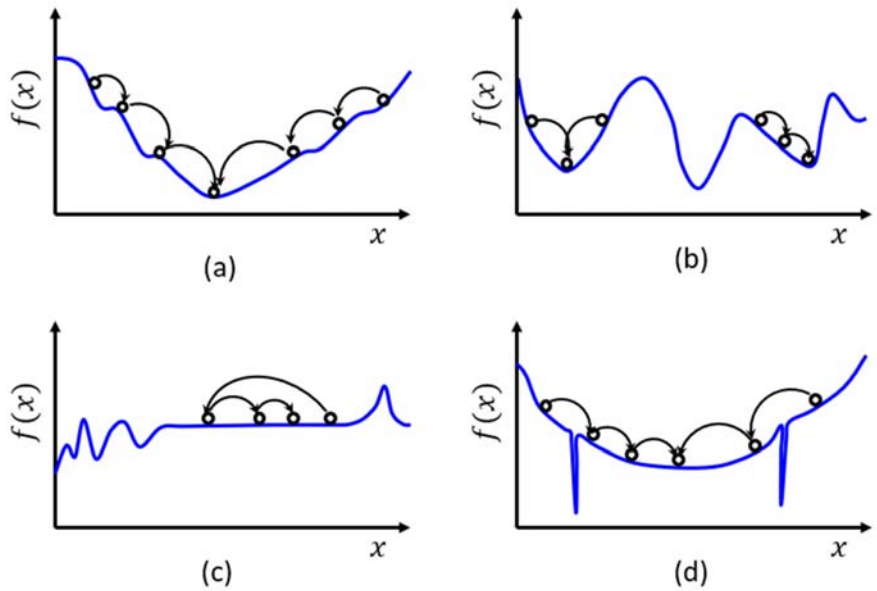


Figure 6-1: Complexity of functions and global optimization.

6.2. Nelder-Mead sequential simplex algorithm

The first gradient-free algorithm that we will discuss in this section is the Nelder-Mead sequential simplex algorithm, which does not require the objective function to be differentiable or even continuous. The algorithm was initially invented by Spendley et al. [56], and then, refined by British mathematicians Nelder and Mead [57]. It is considered the simplest algorithm in a random search by adding simple logics, which is referred to as a structured random search. It is easy to implement, robust, and computationally efficient. This algorithm is currently implemented in Matlab function fminsearch. Matlab refers to a paper on its convergence properties in one or two dimensions [58].

The algorithm has random search characteristic as it is based on generating population in the design space. However, it is not purely random search because it moves the design in a direction that can reduce the objective function. This sounds like a gradient-based algorithm, but it moves the design without requiring the gradient information.

The Nelder-Mead sequential simplex algorithm utilizes the geometry of simplex, which is the simplest body in N dimensional design space with $N + 1$ vertices (triangle in 2D and tetrahedron in 3D). The initial vertices are randomly generated in the design space, but the performance of the algorithm is relatively insensitive to these initial choices. The objective function is evaluated at each vertex. The basic idea of the Nelder-Mead algorithm is to move away from the worst point in the direction of the other points. Then, the algorithm is composed of the following four simple logics to update the design:

- Reflection: the point with the highest objective function value is reflected to the opposite side (preserve volume, nondegeneracy). Specifically, the worst point moves in the direction to the center of gravity of the other points. Most of the time, this point may be an improvement on the worst point, but not on the best point. In that case, the new vertex is the point that is the mirror image of the worst point on the other side.
- Expansion: If the reflected point is the best design (better than the objective functions at all vertices), this means that the reflection direction is promising and the reflected point should go further along that direction; i.e., expand the simplex further (possibly optimum is further in that direction).
- Contraction: If the reflected point is worse than all the existing points, it means the reflected point possibly goes too far and it would be necessary to contract the simplex (possible valley floor).
- Reduction: If the contracted point is still worst, reduce the size of the simplex (possibly the best point is within the simplex)

Figure 6-2 shows an illustration of the Nelder-Mead algorithm for two-dimensional design space. The initial simplex is the blue triangle composed of b_1-b_2-b_3. At these three vertices, the objective function has the highest value at b_3, which is reflected against b_1-b_2. The reflected point is b_r. If the objective value at b_r is between that of b_1 and b_2, it becomes a new vertex, replacing b_3. If the objective value at b_r is less than that of b_1 and b_2, it is further extended to b_e. If the objective value at b_r is greater than that of b_1 and b_2, it is contracted to b_c. If the objective function value at the contracted point b_c is greater than that of b_1 and b_2, reduce the size of simplex by half.

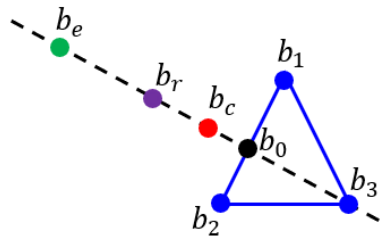


Figure 6-2: Nelder-Mead sequential simplex algorithm in two-dimensional design space.

With reference to Figure 6-2, the Nelder-Mead sequential simplex algorithm in N-dimensional design space starts with $N+1$ randomly selected vertices of a simplex. The objective functions at all vertices are evaluated and ordered by function values: $f(\mathbf{b}_1) \le f(\mathbf{b}_2) \le \cdots \le f(\mathbf{b}_{N+1})$. That is, $\mathbf{b}_1$ is the best design and $\mathbf{b}_{N+1}$ is the worst design for the given simplex. Then, the following algorithm is repeated until the size of simplex is less than a threshold.

1. Calculate $\mathbf{b}_0$ at the center of gravity of all the points except for $\mathbf{b}_{N+1}$

$$\mathbf{b}_0 = \frac{1}{N}\sum_{i=1}^{N} \mathbf{b}_i$$

2. (Reflection) Reflect the worst point $\mathbf{b}_{N+1}$ about $\mathbf{b}_0$ ($\alpha = 1$)

$$\mathbf{b}_r = \mathbf{b}_0 + \alpha(\mathbf{b}_0 - \mathbf{b}_{N+1})$$

   If $\mathbf{b}_r$ is better than the second worst, but not the best, use it to replace with the worst and go back to Step 1 ($\mathbf{b}_{N+1} = \mathbf{b}_r$)

3. (Expansion) If $\mathbf{b}_r$ is the best; i.e., $f(\mathbf{b}_r) < f(\mathbf{b}_1)$, move further in the direction to $\mathbf{b}_r$ ($\gamma = 2$)

$$\mathbf{b}_e = \mathbf{b}_0 + \gamma(\mathbf{b}_0 - \mathbf{b}_{N+1})$$

   If $f(\mathbf{b}_e) < f(\mathbf{b}_r)$, then replace $\mathbf{b}_{N+1}$ by $\mathbf{b}_e$. Otherwise, replace $\mathbf{b}_{N+1}$ by $\mathbf{b}_r$. Go back to Step 1

4. (Contraction) If $\mathbf{b}_r$ is the worst; i.e., $f(\mathbf{b}_r) \ge f(\mathbf{b}_N)$, move less in the direction to $\mathbf{b}_r$ ($\rho = 0.5$)

$$\mathbf{b}_c = \mathbf{b}_0 + \rho(\mathbf{b}_0 - \mathbf{b}_{N+1})$$

   If $f(\mathbf{b}_c) < f(\mathbf{b}_{N+1})$, then replace $\mathbf{b}_{N+1}$ by $\mathbf{b}_c$.

5. (Reduction) If $f(\mathbf{b}_c) \ge f(\mathbf{b}_{N+1})$, contract all points about the best one ($\sigma = 0.5$)

$$\mathbf{b}_i = \mathbf{b}_1 + \sigma(\mathbf{b}_i - \mathbf{b}_1), i = 2, \cdots, N+1$$

   Go back to Step 1

Example 6-1
Consider the minimization of the Rosenbrock function, commonly called a banana function:

$$\text{minimize } f(\mathbf{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

It is known that the function has its minimum at $\mathbf{x}^* = (1,1)$ with $f(\mathbf{x}^*) = 0$. With the initial design at $(-1.2,1.0)$, perform 20 optimization iterations using the Matlab fminsearch function (Nelder-Mead sequential simplex method).

Solution:
First, the following Matlab function banana is stored in the banana.m file:

```
function [y]=banana(x)
```

```
global z1 z2 yg count
y=100*(x(2)-x(1)^2)^2+(1-x(1))^2;
z1(count)=x(1); z2(count)=x(2);
yg(count)=y;
count=count+1;
```

The banana function does not only codes the Banana function, but it also stores the coordinates and the function values every time it is called. This allows us to plot the history of design evolution. The following Matlab script calls fminsearch with a limit on the number of function evaluations set at 20

```
global z1 z2 yg count
count =1;
options=optimset('MaxFunEvals',20)
[b,fval] = fminsearch(@banana,[-1.2, 1],options)
mat=[z1;z2;yg]
```

The array mat stores the history of design variables and objective function:

```
mat =
  Columns 1 through 8
   -1.200   -1.260   -1.200   -1.140   -1.080   -1.080   -1.020   -0.960
    1.000    1.000    1.050    1.050    1.075    1.125    1.1875   1.150
   24.20     39.64    20.05    10.81     5.16     4.498    6.244    9.058

  Columns 9 through 16
   -1.020   -1.020   -1.065   -1.125   -1.046   -1.031   -1.007   -1.013
    1.125    1.175    1.100    1.100    1.119    1.094    1.078    1.113
    4.796    5.892    4.381    7.259    4.245    4.218    4.441    4.813
```

The first two rows are design variables, x_1 and x_2, while the last row is the objective function. In order to construct the initial simplex (triangle in 2D problem), fminsearch perturbs the initial design by 5% in each design direction. Figure 6-3 shows the first seven design points generated by fminsearch function in Matlab. The first three points are the initial simplex. Here the first reflection, $\mathbf{x}^{(4)}$, is very successful, in that the value of the function, at 10.81, is better than any of the previous points, so we double the distance and get an even better result of 5.16 at $\mathbf{x}^{(5)}$. At the step, the simplex is composed by $\mathbf{x}^{(5)}$, $\mathbf{x}^{(3)}$, and $\mathbf{x}^{(2)}$. The second iteration again starts with a reflection of the worst point $f\left(\mathbf{x}^{(2)}\right) = 24.2$ about the middle of the line connecting the other two points, $\mathbf{x}^{(5)}$ and $\mathbf{x}^{(3)}$. This gives us an improvement to 4.49 at $\mathbf{x}^{(6)}$, so we try expansion again to $\mathbf{x}^{(7)}$. However, this time we do not get an improvement, so the simplex for the next iteration will include $\mathbf{x}^{(6)}$, $\mathbf{x}^{(5)}$, and $\mathbf{x}^{(3)}$.

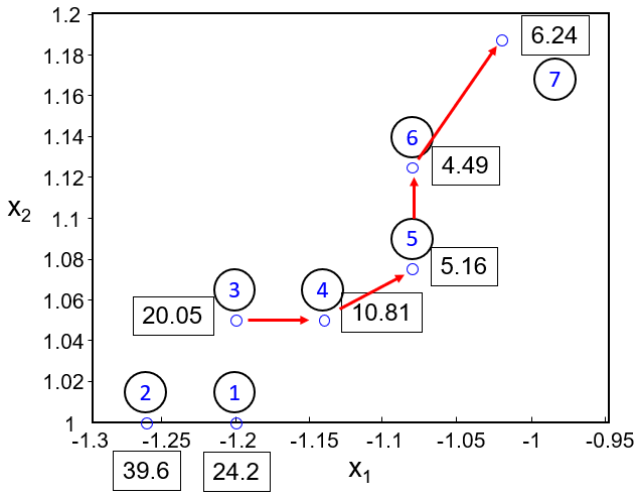


Figure 6-3: First seven design points generated by fminsearch in Matlab.

6.3. DIRECT method

Although the Nelder-Mead sequential simplex method is a gradient-free algorithm, it has a limitation to be called a global search algorithm as it starts with an initial simplex and update the simplex based on function values. If a function has many local optima, it tends to find a local optimum depending on the size and location of the initial simplex. A global search algorithm should have a capability to explore the entire design space for a possible, better optimum design.

In this section, we will discuss a new algorithm, called DIviding RECTangles (DIRECT) algorithm, originally proposed by Jones et al. [59]. The DIRECT algorithm is an example of systematic exploration of the design space by dividing it into regions, and then into sub-regions. The regions would be sections of a line in 1D, rectangles in 2D, and cuboids in 3D. In higher-dimensions, they are referred to hyper-boxes. Compared to other global search algorithms, the DIRECT algorithm is purely deterministic, which means that the algorithm will produce the same optimum design even if it runs multiple times. In order to understand the DIRECT optimization algorithm, it would be necessary to discuss Lipschitzian optimization first, followed by how the DIRECT algorithm extends it.

Lipschitzian optimization

Lipschitzian optimization was originally proposed by Shubert [60], which is designed for seeking the global minimum of a function. It is named after Rudolf Otto Sigismund Lipschitz (14 May 1832 – 7 October 1903) who was a German mathematician and professor at the University of Bonn from 1864. Lipschitzian optimization is a predecessor of the DIRECT optimization algorithm that divides the space into boxes, samples at the center of the boxes and divides boxes further based on the function's Lipschitz constant. The Lipschitz constant is an upper bound on the rate of change of the function, and it can be used even for functions that are not even differentiable.

A function $f: D \in R^n \to R$ is called Lipschitz continuous if there exists a positive constant $K \in R^+$ such that

$$|f(x) - f(x')| \leq K|x - x'|, \qquad \forall x, x' \in D \tag{6.1}$$

where the lowest K is called the Lipschitz constant. This basically excludes those functions whose absolute value of the slope approaches infinite. The upper bound of the slope is the Lipschitz constant. As long as the objective function is a bounded function of design variables, it is Lipschitz continuous. For $x \in [a, b]$, replacing a and b with x' in the previous equation obtains

$$\begin{aligned} f(x) &\geq f(a) - K(x - a) \\ f(x) &\geq f(b) + K(x - b) \end{aligned} \tag{6.2}$$

In the first equation, the right-hand side is the straight line from $f(a)$ with a slope of $-K$. In the second equation, the right-hand side is the straight line from $f(b)$ with a slope of $+K$. Therefore, the function value is above the two straight lines. As shown in Figure 6-4(a), the intersection of these two lines is the possible lowest value if the function is decreased by a rate of K from both ends. Note that given K the function cannot be possibly lower than the intersection value. The intersection point can be calculated as

$$\begin{aligned} x_1(a, b, f, K) &= \frac{a + b}{2} + \frac{f(a) - f(b)}{2K} \\ f_1(a, b, f, K) &= \frac{f(a) + f(b)}{2} - \frac{K(b - a)}{2} \end{aligned} \tag{6.3}$$

The Lipschitz optimization repeats this process by reducing the intervals. As shown in Figure 6-4(b), the process is repeated with two intervals that were created by adding x_1; that is, (a, x_1) and (x_1, b). Figure 6-4(b) shows that the process predicts a lower bound on the function in (a, x_1) that is better than the one in (x_1, b). Therefore, we select the intersection there and get x_2. We can repeat the process at the additional two intervals that were created by adding x_2, that is (a, x_2) and (x_2, x_1). Among all intersection points, we choose the lowest on as x_3 as shown in Figure 6-4(c).

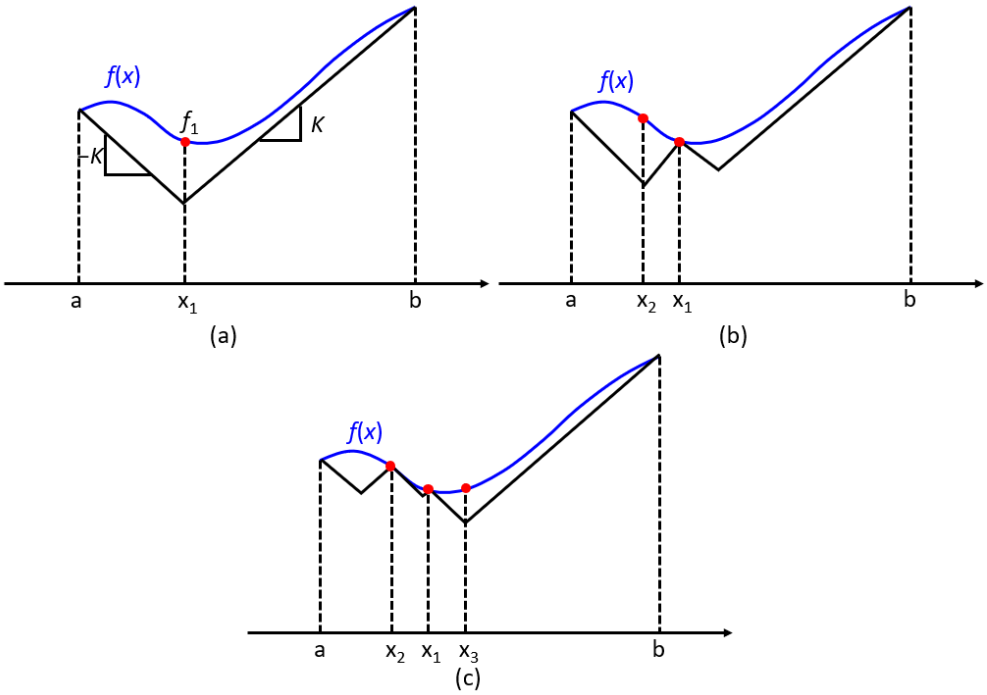


Figure 6-4: Lipschitzian optimization of 1D function.

The Lipschitzian optimization algorithm is simple and straightforward, and yet, provides a robust global searching capability. The algorithm is deterministic, which means there is no need for multiple runs. In addition, the algorithm requires a few tuning parameters. The only parameter it needs is the Lipschitz constant K, which depends on the function. On the other hand, the determination of the Lipschitz constant may not be an easy task as the global behavior of the function is unknown. In practice, the Lipschitz constant affects the convergence speed as well as global searching capability. A Large K can help global search but it slows down the convergence speed. On the other hand, a small K may end up searching for local optima. Lastly, the algorithm may have a computational complexity for high dimensions. Since the algorithm requires to evaluate function values at all corners of the design space, it requires a large number of function evaluations $\sim O(2^n)$ for a high-dimensional design space.

DIRECT in 1D

The direct algorithm is inspired by Lipschitzian optimization. The algorithm divides the design space into hyper-boxes (lines in 1D, rectangles in 2D, cuboids in 3D, etc.) and samples the function at the center of each hyper-box. Based on Jones et al. [59], another challenge in Lipschitzian optimization is the determination of the Lipschitzian constant K. In the DIRECT algorithm, instead of finding the intersection of two lines as in Figure 6-4(a), the function value is evaluated at the center of the region. In addition, in order to recycle the center of the region in subsequent subregions, the region of design is

divided into three subregions and the function values are evaluated at the center of each subregion. As shown in Figure 6-5(a), the center point P in the initial region is recycled after the region is divided into three subregions. From the perspective of Lipschitzian optimization, the DIRECT algorithm uses the following Lipschitz bound:

$$\begin{aligned} f(x) &\geq f(c) + K(x-a) \quad \text{for } x \leq c \\ f(x) &\geq f(c) - K(x-c) \quad \text{for } x \geq c \end{aligned} \tag{6.4}$$

Since the function value at the center of the region is used instead of the two ends, it is unnecessary to find the intersection of two lines, as shown in Figure 6-5(b). Also, it is unnecessary to define the Lipschitz constant K that is the upper bound of the slope of the function.

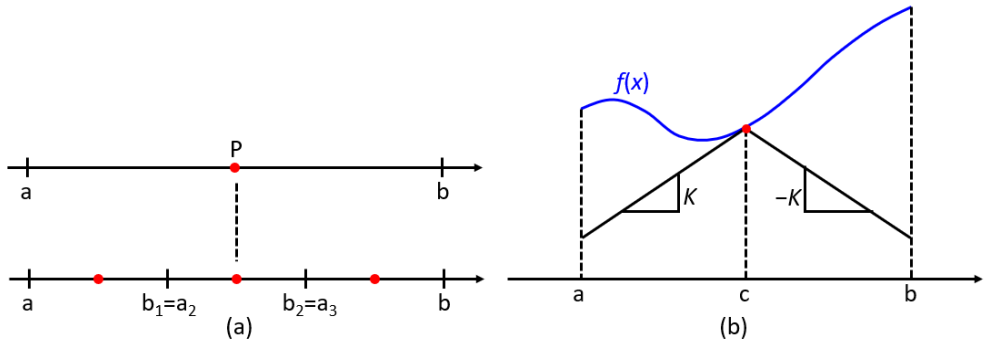


Figure 6-5: The DIRECT algorithm in 1D design space.

DIRECT algorithm

The DIRECT algorithm starts from one region of the entire design space and sequentially subdivides the region. At a given iteration, the design space is divided into subregions, and the objective function is evaluated at the center of each subregion. Figure 6-6 illustrates an example of the subdivision process of the DIRECT algorithm in a two-dimensional design space. The horizontal axis is x_1 and the vertical axis is x_2. First, the design space is normalized such that the box is square. The yellow boxes represent the regions that need to be subdivided, and the red circles represent the location of function evaluation. Initially, a single box is used for the entire design space, and the function is evaluated at the center of the design space as shown in Figure 6-6(a). In the implementation of Jones et al. [59], the rule of the subdivision is such that the longer dimension is subdivided into three subregions. When a box is a square, the region is subdivided into three regions in the direction of x_2 first, and then, only the center region is subdivided into three regions in the direction of x_1 as shown in Figure 6-6(b). Such scheme only requires four additional function evaluations. After comparing all five function values, it is determined that the bottom box is further subdivided. The rule of selecting the regions to be subdivided will be discussed later. In this case, the subdivision is only performed in the direction of x_1 because the bottom box is rectangle and has a longer length in the direction of x_1. In the next iteration as shown in Figure 6-6(c), both the top rectangle and the bottom-center square are selected. In the optimization term, subdividing a large box corresponds to exploring the design space, while subdividing a small box corresponds to refining the design space. The top rectangle is subdivided into three squares in the direction of x_1, while the bottom-center square is subdivided into both x_1 and x_2 directions, as shown in Figure 6-6(d). In the subsequent evaluations, the top-center and bottom-right boxes will be further subdivided in the next iteration.

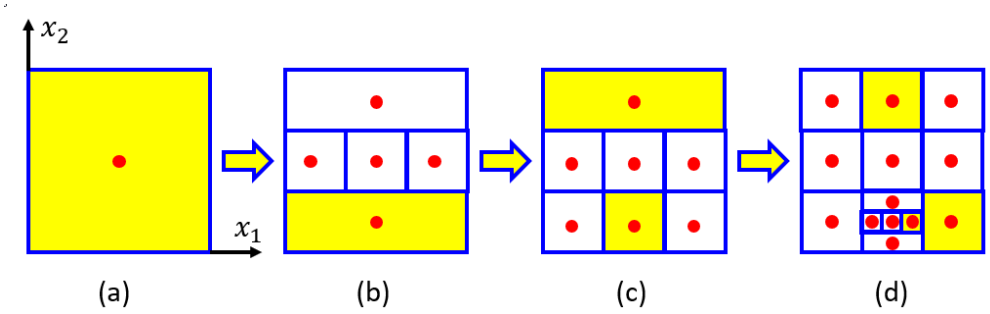


Figure 6-6: DIRECT algorithm in 2D design space.

A critical question of the DIRECT algorithm is which boxes should be subdivided. As mentioned before, this subdivision is used to explore the design space and refine the optimum design. In the DIRECT algorithm, this determination is done using a graph of box-size versus objective function as shown in Figure 6-7. In the graph, the box-size represents the largest dimension of a box, while the ordinate is the value of the objective function at the center of the box. With reference to Figure 6-7(a), if the smallest box with the lowest objective function is subdivided, it can look for a better design within the current best box, which corresponds to refining the current optimum design. On the other hand, the lowest objective functions of larger boxes may have higher values, but they have a chance to yield a better design if they are subdivided, which corresponds to exploring the design space.

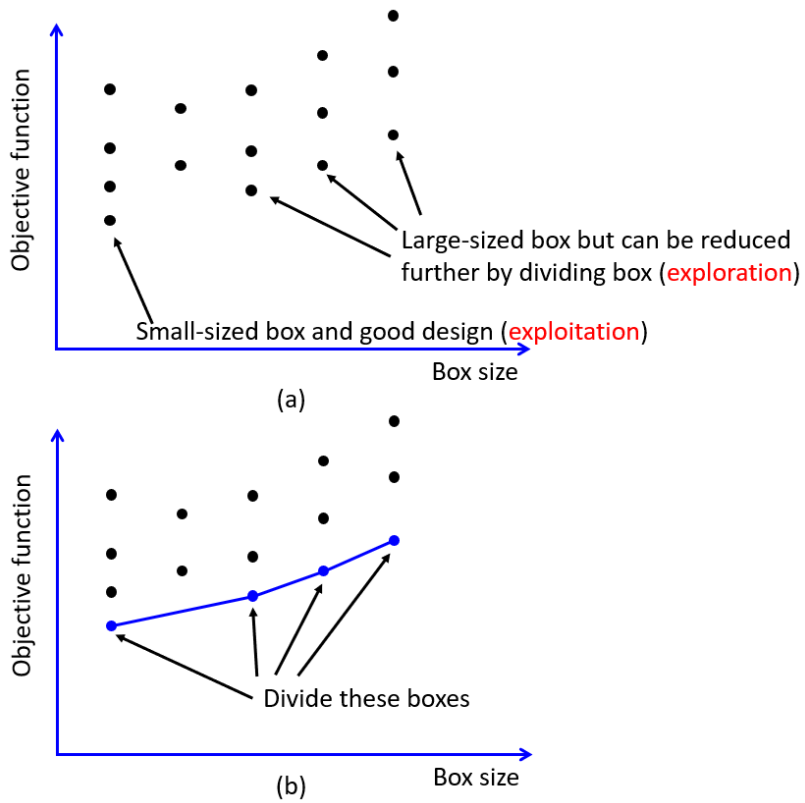


Figure 6-7: Determination of subdivision boxes in the DIRECT algorithm.

In the DIRECT algorithm, subdivisions are made at those points that have the best possibility of improving the optimum design. These points can be selected by constructing a convex hull of box sizes as

shown in Figure 6-7(b). Those points on the convex hull are subdivided and function values are evaluated at the center of the new boxes. If enough number of iterations are performed, every region in design space will be eventually divided into a small box. Therefore, the algorithm is guaranteed to find the global optimum even if the objective function has a difficult shape to find the global optimum design. In practice, however, it will require a huge number of function evaluations.

Figure 6-7 only shows a two-dimensional illustration. However, the number of boxes (i.e., the number of function evaluations) increases exponentially with dimension. For example, consider the case of ten design variables. Having ten regions in each design variable is not considered a high resolution. Even with such a coarse resolution, the number of function evaluations can reach 10^{10}, which is practically impossible for most functions. Therefore, there is a serious question as to how practical is DIRECT for a high-dimensional problem, which is discussed by Cox et al. [61]. Therefore, this algorithm is appropriate for an optimization problem with a small number of design variables.

The Matlab code of the DIRECT algorithm from Dr. Daniel Finkel is available in the website (https://ctk.math.ncsu.edu/Finkel_Direct/). The code is based on the algorithm published by Jones et al. [59]. The website also has user's guide as well as sample optimization problems. The Matlab code is also available in the companion website of this book.

Example 6-2

Optimize the following Goldstein-Price function in the design space of $-2 \le x_1, x_2 \le 2$ using the DIRECT algorithm:

$$\text{minimize } f(\mathbf{x}) = \left(1 + (x_1 + x_2 + 1)^2 * (19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)\right) * \left(30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)\right)$$

Solution:

The Goldsten-Price function has its global minimum at $(x_1, x_2) = (0, -1)$ with the minimum value $f_{min} = 3.0$. In order to solve the optimization problem, it would be necessary to define the objective function as an m-file. The following gp.m file is defined:

```
function value = gp(x)
x1 = x(1); x2 = x(2);
value =(1+(x1+x2+1).^2.*(19-14.*x1+3.*x1.^2-14.*x2+6.*x1.*x2+3.*x2.^2))...
.*(30+(2.*x1-3.*x2).^2.*(18-32.*x1+12.*x1.^2+48.*x2-36.*x1.*x2+27.*x2.^2));
```

The above gp.m file will calculate the objective function value at a given design (x_1, x_2). Then, the following Matlab code will define the optimization problem and call the Direct.m Matlab function to find the optimum design:

```
clear all;
bounds = [-2 2;-2 2];                    % lower- and upper-bounds of design
options.testflag  = 1;                          % Global minimum is known
options.globalmin = 3;                      % The value of known global min
options.showits   = 1;                           % Show iteration history
options.tol       = 0.01;                           % Stopping criterion
Problem.f = 'gp';                               % Objective function name
[fmin,xmin,hist] = Direct(Problem,bounds,options);% Calling DIRECT algorithm
plot(hist(:,2),hist(:,3))                      % Plot iteration statistics
xlabel('Fcn Evals'); ylabel('f_{min}');
```

The optimum design is stored at xmin, and the optimum objective value is at fmin. It turns out $x_{min} = (0, -1.0005)$, and $f_{min} = 3.0001$, which is within the tolerance. Figure 6-8(a) shows the contour of the

objective function, which shows $(x_1, x_2) = (0, -1)$ as the global optimum design. Figure 6-8(b) shows the history of the objective function versus the number of function evaluations. Although optimization took only 14 iterations, it requires a total 191 function evaluations. As boxes are refined, more function evaluations are required at a single iteration.

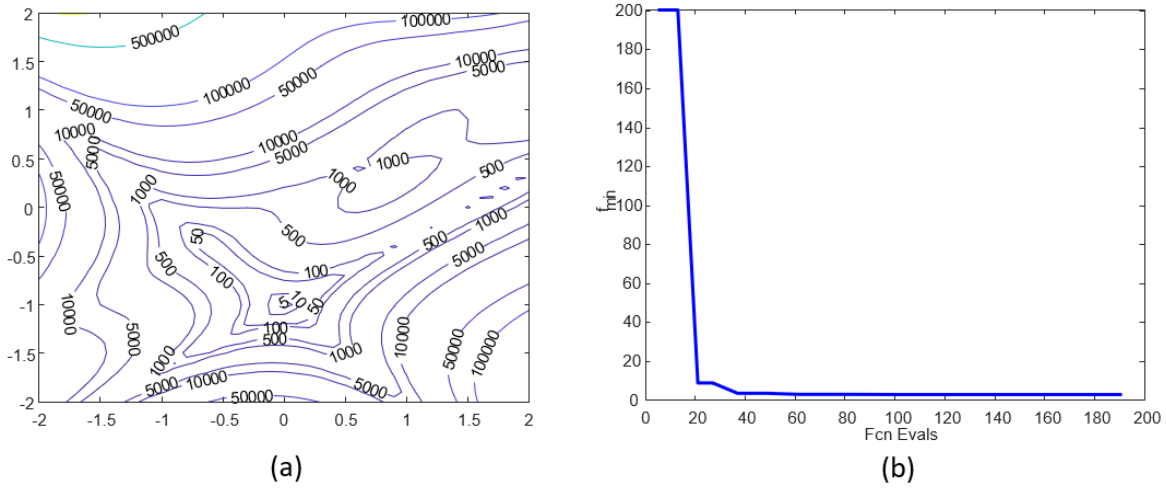


Figure 6-8: Contour of objective function and optimization history of **Example 6-2**.

6.4. Genetic algorithms

The sequential simplex method and the DIRECT method in the previous sections have the global searching capability, but they are deterministic in a sense that they will produce the same optimum design when the algorithm is repeated. That is, there is no randomness in the searching process. This is not a desirable feature of global searching algorithms as randomness can increase the chance of finding the global optimum. Therefore, many global searching algorithms have randomness imbedded in their iteration process. Among many global searching algorithms, genetic algorithms are the most popular ones. Genetic algorithms were developed in 1975 by John Holland from the University of Michigan [62]. However, similar algorithms were developed in Europe around the same time with the name of evolutionary strategies. The main difference is that genetic algorithms are usually associated with discrete variables, while evolutionary strategies with continuous ones.

Genetic algorithms are inspired by Darwin's principle of evolution, where a population of individuals can adapt to its environment because individuals that possess traits that make them less vulnerable than others are more likely to have descendants and therefore to pass on their desirable traits to the next generation. One can think of this process of adaptation as an optimization process that probabilistically creates fitter individuals through selection and recombination of good characters. Genetic algorithms are simplified computer models of evolution, where the environment is emulated by the objective function to maximize/minimize, and the design variables play the role of the individuals. Moving from parents to children corresponds to one iteration in optimization, and the goodness is measured in terms of the objective function. The key ideas that distinguish genetic algorithms from traditional search algorithms are the use of population of designs, the mating of pairs of design to create child designs in a process called crossover, and the use of mutations (i.e., randomness) to enhance exploration. Together they create an optimization process that has a strong random component, so every time the algorithms are run, they produce different optimum designs.

Representation of design

An important difference between the genetic algorithms and conventional engineering optimization is the representation of design. In engineering design, a design variable is represented using a number, either continuous or discrete. In genetic algorithms, a design variable is stored in the chain of DNA (deoxyribonucleic acid) chromosomes, often using binary digits. Each design encodes of a particular candidate structure in the form of one or several chromosomes, which are strings of finite length. A design variable is represented in the form of a string of binary numbers. Each chromosome can have a value of 0 or 1. The mapping between binary numbers and the state of design variable is called coding. As an example, let a discrete design variable have four different states: red, green, blue, and yellow. In order to represent this design variable, two chromosomes should be enough as they can represent four different states: 00, 01, 10, and 11. Therefore, by coding 00 as red, 01 as green, 10 as blue, and 11 as yellow, the design variable can have all possible states. Although binary coding is most common, real number coding is also possible but requires special treatment.

The same binary digits can be used to store integer variables in a similar way that a computer stores numbers in the form of a binary numeral system. Figure 6-9 shows an eight-digit binary number. In the binary system, each bit represents an increasing power of 2, with the rightmost bit representing 2^0, the next representing 2^1, then 2^2, and so on. The value of a binary number is the sum of the powers of 2 represented by each "1" bit. For example, the binary number 100101 is converted to decimal form as follows:

$$100101_2 = (1 \times 2^5) + (0 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 37$$

As expected, the eight-digit binary numbers can represent up to 256 states of a design variable. A great number of bits are required to represent a design variable with a large range.

Value of cell	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	128	64	32	16	8	4	2	1
Binary number	0	0	1	0	0	1	0	1

Figure 6-9: Representation of 37 in the form of binary numbers.

Example 6-3

Four integer variables are used for the following optimization problem definition

$$\text{minimize } f(\mathbf{x}), \qquad \mathbf{x} = \{x_1, x_2, x_3, x_4\}$$
$$\{0 \le x_1, x_4 \le 15\}, \{0 \le x_2 \le 7\}, \{0 \le x_3 \le 3\}$$

When $x_1 = 6, x_2 = 5, x_3 = 3, x_4 = 11$, represent the integer design variables using binary coding.

Solution:

Since x_1 and x_4 are between 0 and 15, all their states can be represented using four bits. Similarly, x_2 needs three bits and x_3 needs two bits. Therefore, all four design variables are represented using 13 bits as shown in

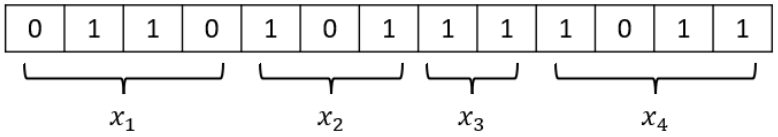


Figure 6-10: Representation of four integer variables.

Many engineering optimization problems include real-valued design variables. It is a challenge to store a continuous variable using binary coding. Instead of storing continuous variables, binary coding stores the variable with resolution or interval. Let a continuous design variable has a lower- and upper-bound, $x^L \le x \le x^U$. We want to store this variable with a resolution of x^{inc}. Then the number m of the required digits can be found from

$$2^m \ge \frac{x^U - x^L}{x^{inc}} + 1 \tag{6.5}$$

For example, if $\{0.01 \le x \le 1.81\}$ with $x^{inc} = 0.001$, the required digits should be

$$2^m \ge \frac{1.81 - 0.01}{0.001} + 1 = 1801 \tag{6.6}$$

Therefore, the smallest integer $m = 11$ and the actual intervals should be $x^{inc} = (1.81 - 0.01)/2^{11} = 0.00088$. It is noted that the interval is adjusted such that the entire 11 digits are used to represent the range. As another example, let a design variable is the height of a stiffener, which can change from 1 inch to 2 inches with 0.1-inch intervals. Even if the design variable can have 11 different values, binary coding requires four digits with 16 values. Therefore, the entire range is divided into 15 intervals, each $1/15$ inch wide. Therefore, 0000 denotes 1", 0001 denotes $16/15$", … 1111 denotes 2". Genetic algorithms, like most global search algorithms, are not effective in getting high-precision designs. It is better to go for a coarse grid of real values.

A common application of discrete variables is stacking sequence optimization of composite materials, where the sequence of four different ply angles, $0°, \pm 45°$ or $90°$, are determined to achieve the best performance of the composite panel. Since the ply thickness is fixed due to the manufacturing process, the design problem is to select the number of plies and select the ply angles for each ply from the four choices. Coding the stacking sequence of a composite laminate that has a finite number of possible angles is typically done by assigning a number to each angle, or even just using the angle itself. For the convenience of explanation, natural coding will be used instead of common binary coding. In natural coding, the four possible ply angles are assigned as $(0° \to 1, 45° \to 2, -45° \to 3, 90° \to 4)$. If binary coding is used, then two bits can be used instead. Therefore, the following stacking sequence can be represented using natural coding as

$$(45/-45/90/0)_s \to (2/3/4/1)$$

The subscribed 's' stands for symmetric stacking, where the total stacking sequence is $(45/-45/90/0/0/90/-45/45)$. When the laminate is symmetric, only one half is coded.

There is often also a balance condition that requires that the laminate has an equal number of positive and negative angles. This means that the number of $45°$ plies needs to be equal to the number of $-45°$ plies. This is a difficult constraint to enforce, and an easy way out is to require that plies come in groups of two: $(0°_2 \to 1, \pm 45° \to 2,\ 90°_2 \to 3)$. This, however, may mean wasted plies for $0°$ or $90°$ directions.

Genetic operators

A key factor of genetic algorithms is working with a population of designs that can mate and create child designs. Among the population, two parents are selected based on their goodness (fitness) of their objective function values. Then, child designs are created using genetic operations, which takes some genes from one parent and some from a second parent. Figure 6-11 shows four basic genetic operators.

The first genetic operator is selection, where good parents are chosen to produce children in the next generation. The basic idea is to pass designs with higher fitness to the next generation. This selection operation increases population of high fitness design, while removing low fitness designs. Therefore, the mean fitness of the population is improved.

The crossover operator exchanges some genes from the two parents to produce children. This corresponds to the case when a child takes a part of their gene from one parent and the remaining part from the other parent. First some chromosomes of the two parents are sliced and exchanged. This operation diversifies the population, while the mean fitness is preserved. If the two parent designs are good (i.e., having a good fitness value), the chance is high that their child design can produce a better fitness. Selection plus crossover form together an exploitation mechanism that looks for combination of good designs. Randomness can be included in the selection process of the two parents and in determining the crossover location. The genetic representation of designs with a fixed chromosome length makes the crossover convenient as their parts can be easily aligned.

Mutation changes a chromosome randomly and adds an element of exploration. These two operators are used in practically every genetic algorithm. Permutation reverses the order of a portion of chromosomes. In the case stacking sequence optimization of composite material, it preserves the in-plane properties and changes only the bending properties. Addition/deletion adds/removes one chromosome from a design. This operation is useful when the design space needs to be expanded to satisfy design constraints.

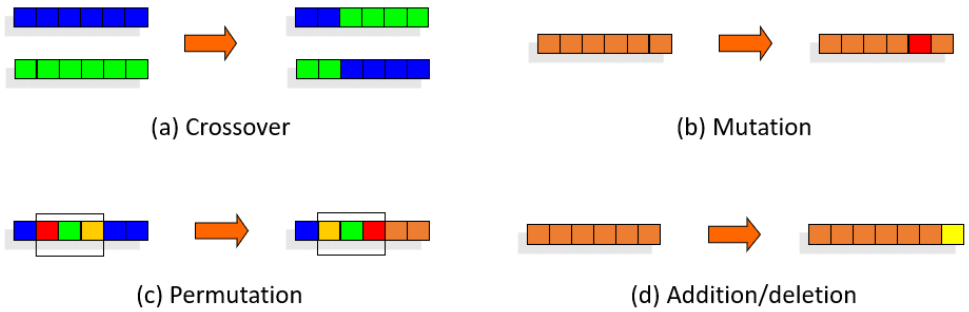


Figure 6-11: Genetic operations.

Example 6-4

In the stacking sequence design of a composite material, two parents are given in $[\pm 45_2/0_4/90_2]$s and $[\pm 45_4/0_2]$s with 5-digit genes. When the crossover occurred at the second digit, calculate their child designs.

Solution:

For coding, let us assign $0_2 \rightarrow 1, \pm 45 \rightarrow 2, 90_2 \rightarrow 3$. Then, the two parents can be represented as

$$\text{Parent 1} : [2/2/1/1/3]$$

$$\text{Parent 2} : [2/2/2/2/1]$$

After the crossover at the second digit, the two children become

$$\text{Child 1} : [2/2/1/2/1]$$

$$\text{Child 2} : [2/2/1/1/3]$$

Therefore, the stacking sequences of the two children are $[\pm 45_2/0_2/\pm 45/0_2]$s and $[\pm 45_2/0_4/90_2]$s.

Example 6-5
The first child in Example 6-4 has a mutation at the fourth digit, where a random value of 3 is assigned, followed by a permutation between 2-3-4 digits. Calculate the resulting stacking sequence.

Solution:
From Example 6-4, Child 1 has a design of [2/2/1/2/1]. After a mutation at the fourth digit, it becomes [2/3/1/2/1]. Then, a permutation between 2-3-4 digits yields [2/2/1/3/1]. This design corresponds to a stacking sequence of $[\pm 45_2/0_2/90_2/0_2]$s.

Procedure of genetic algorithms

Figure 6-12 illustrates the basic procedure of genetic algorithms using the genetic operators. The algorithms start with a population of designs, and the fitness of each design is calculated based on its objective function and possibly constraint satisfaction. Then, parent designs are selected based on their fitness but with randomness thrown in. From the illustrative figure, the best (green) design was selected three times, as was the second best (red). The third best design (blue) was selected twice and the worst design (orange) not at all. Once parents are selected, children are created genetic operations. In the figure, only crossover is used to create children. After all children are created, they become parents in the next generation (iteration). This process is continued until stopping criteria are satisfied.

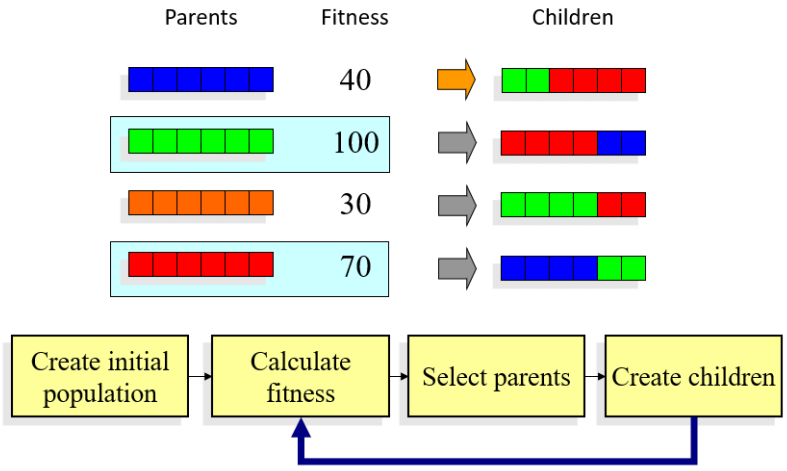


Figure 6-12: Illustration of the procedure of genetic algorithms.

The procedure of genetic algorithms can be summarized as follows:

- Create a random initial population
- Generate the next population based on the following steps
  - Scores each design of the current population based on its *fitness*
  - Selects members, called *parents*, based on their fitness
  - Choose lower/higher fitness members as *elite*. These elite members are passed to the next population
  - Produces *children* from the parents using *genetic operations*

  - Replaces the current population with children to form the next generation
- Repeat the process until the *stopping criteria* are satisfied

The procedure of genetic algorithms starts from creating initial population of designs randomly. In the case of Figure 6-12, the initial population consists of four designs with six digits each. In order to generate random designs, random number generator in common computer programs can be used. For example, Matlab uses the rand function to generate random numbers from a uniform distribution$\sim U[0,1]$. However, most random number generators in computer programs are pseudo random based on the initial seed. If a same initial seed is used, the same sequence of random numbers are generated. The initial seed may reflect some internal computer state, time of day, or some other data available to the generator. It is selected so that it is practically random, and different results are expected at different trials. However, seed is useful when we want to generate the same sequence of random numbers. In most cases, it is necessary to interpret/transform random numbers to design values.

Once a population of designs is randomly generated, the next step is to calculate the fitness of each design. In unconstrained optimization, the fitness corresponds to the objective function to minimize. In the case of a constrained optimization problem, the following three-term augmented objective function is commonly used:

$$f^* = f + pv - bm + \text{sign}(v)\Delta \tag{6.7}$$

where f is the objective function, v is the maximum violation of constraints, m is the minimum margin of satisfying constraints, and $p, b,$ and Δ are constants. By having a large value of p, the objective function is penalized proportional to violated constraints. In addition, there is a small penalty Δ if there is any violated constraints, to give a preference to designs that satisfy the constraints over that have very small violations. Finally, there is a term that rewards margin with respect the constraint with a bonus. This is important when there are multiple designs that have the same value of the objective function, but some satisfy the constraint with a larger margin.

Some objective functions vary in a wide range, while others vary in a small range. In order to make a robust algorithm, it would be necessary to select the fitness so that the difference between the best design and the poorest design is large even if their objective function values are close. This prevents stagnation in the evolution as we near the optimum. Accentuating the difference may be done by normalizing it as

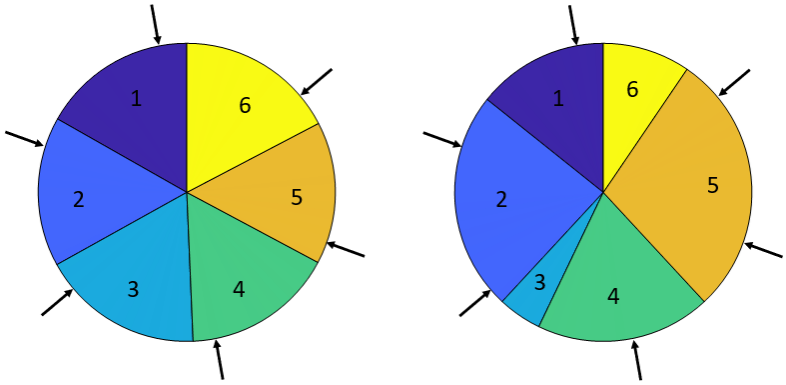
$$\text{fitness} = \frac{|f^* - f^*_{\max}|}{|f^*_{\min} - f^*_{\max}|} \tag{6.8}$$

The normalized fitness is insensitive to the range of objective function.

The two most common ways of selecting parents are roulette wheel and tournament. In this section, only the roulette wheel approach will be explained. Tournament selection is based on selecting randomly two individuals from the population and picking the best one as one parent, then repeating to select the second parent. Some genetic algorithms pursue elitist strategies, where the top design in a generation is always selected to be moved to the next generation. This guarantees monotonic progress, but it has not been proven to lead to faster convergence. In fact, putting too much preference to the top design means focusing on refining local optimum but limiting exploration of a better design far away from the current good one. Finally, often genetic algorithms have a no twin rule, in that both parents cannot be identical.

In a roulette wheel approach, each design occupies a slice of the roulette wheel that is proportional to its fitness. Spinning the roulette wheel is simulated by generating a random number from a uniform distribution$\sim U[0,1]$. Figure 6-13(a) shows an illustration of roulette wheel selection with six fitness values, $\{0.62, 0.60, 0.65, 0.61, 0.57, 0.64\}$. Since the six fitnesses are close, the slices allocated to the six designs are close in size. Therefore, if they are randomly selected (arrows in the figure), they all have a

similar probability to be selected. If, on the other hand, the fitness is made proportional to the reverse rank, then the top design will get a slice of 6/21 of the wheel, the second best 5/21, and the poorest design 1/21. Therefore, the following reverse rank $\{3/21, 5/21, 1/21, 4/21, 6/21, 2/21\}$ is used to make the roulette wheel, as shown in Figure 6-13(b). The figure also shows the effect of generating six random numbers and selecting individuals based on these numbers. With the original fitnesses, each design is selected once. With the reverse rank-based fitness, the first and second best designs are selected twice, while the third and fourth are selected once, and fifth and sixth are not selected at all.



(a) (b)

Figure 6-13: Example of roulette wheel selection method.

Once parents are selected, genetic operators are applied to create child designs. The process of moving from parents to children is referred to as a generation, which is equivalent to iteration in optimization. While the conventional optimization updates a design to a better design, genetic algorithms modify a population of designs via selection and genetic operations, by which, the population evolves toward an optimal design. Once child designs are created, they become a new population and the process is repeated until the algorithm satisfies stopping criteria.

Stopping criteria determine when to stop optimization iteration. Although, the users can determine when and how to stop optimization iteration, below are common stopping criteria that are widely used.

- Stop when the maximum *number* of generations reaches
- Stop after running for the maximum computational *time* reaches
- Stop when the *best* fitness becomes less than or equal to the limit
- Stop when the *relative change* in fitness is less than the tolerance
- Stop if there is *no improvement* during an interval of time
- Stop if the average *relative change* in fitness is less than the tolerance.

Genetic algorithm in Matlab

Genetic algorithms can be applied to solve optimization problems that are not well suited for standard optimization algorithms, including problems in which the objective function is discontinuous, non-differentiable, stochastic, or highly nonlinear. Matlab provides the ga function for minimizing an objective function with linear and nonlinear constraints using genetic algorithms. The ga function can be called as

```
[x,fval,exitflag,output,population,scores] = …
ga(fun,nvars,A,b,[],[],lb,ub,nonlcon,IntCon,options)
```

The ga function takes various input parameters. The fun is the function handle of the fitness function, nvars is the number of design variables, A and b are for linear inequality constraints in the form of $\mathbf{Ax} \leq \mathbf{b}$, lb and ub are lower- and upper-bounds of design variables, nonlcon is the function handle of the nonlinear inequality constraints, IntCon is for integer variable, and options are optimization option.

Once the ga algorithm is finished, it returns with various output results. The variable x includes the optimum design, fval is the value of the objective function at the optimum design, population includes the final population of designs, and scores include the fitness function values of the population. When the exitflag is positive, it means the algorithm stopped normally, while when it is negative, it means the algorithm stopped due to erroneous reasons.

Example 6-6

The Rosenbrock function is frequently used for testing optimization algorithms. The functional has two design variables, as

$$f = 100(x_1^2 - x_2)^2 + (1 - x_1)^2 \tag{6.9}$$

It has a minimum value of zero at design $\mathbf{x} = \{1,1\}$. Find the optimum design of the Rosenbrock function using the Matlab ga function. Use the range of design variables $-3 \leq x_1, x_2 \leq 3$.

Solution:

Figure 6-14(a) shows the plot of the Rosenbrock function with the optimum design at $\mathbf{x} = \{1,1\}$. Although the function varies over a wide range $[0, 15{,}000]$, it does not vary near the optimum design. Therefore, this function is difficult to identify the optimum design. Since the function is quite steep, it would be easier to visualize it if the logarithm of the function is plotted instead. Figure 6-14(b) shows the plot of the modified Rosenbrock function $\hat{f} = \log(100 + 100(x_1^2 - x_2)^2 + (1 - x_1)^2)$. A value of one hundred is added to the function in order to prevent the function from approaching negative infinity. The figure also shows the optimum design at $\mathbf{x} = \{1,1\}$.

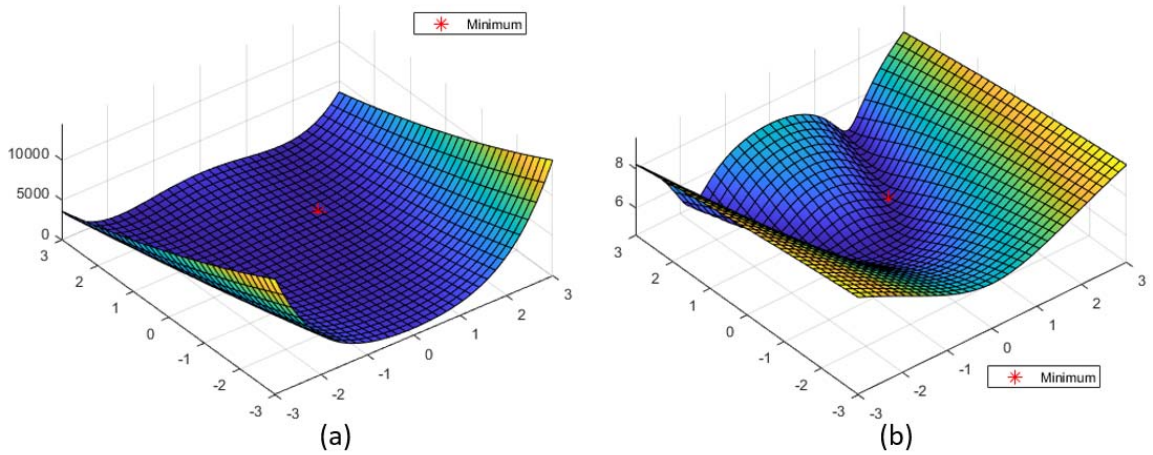


Figure 6-14: Plot of Rosenbrock function in logarithm.

Since this is unconstrained optimization, the objective function is used as the fitness function. To minimize the fitness function using ga, the following fitness function Rosenbrock.m is created:

```
function y = Rosenbrock(x)
```

```
y = 100 * (x(1)^2 - x(2))^2 + (1 - x(1))^2;
```

For given input design variables $\mathbf{x} = \{x_1, x_2\}$, the Rosenbrock function returns the value of the objective function. The following Matlab script defines the lower- and upper-bounds of the design and calls the ga function.

```
rng default     % For reproducibility
ndv = 2;
lb = [-3,-3];
ub = [3,3];
[x,fval] = ga(@Rosenbrock,ndv,[],[],[],[],lb,ub)
```

The rng function sets the seed of the random number generator default so that the same result can be obtained. After launching the ga function, the optimization stops with the following output:

```
Optimization terminated: maximum number of generations exceeded.
x = 1.5083    2.2781
fval = 0.2594
```

The x returned by the solver is the best point in the final population computed by ga. The fval is the value of the function Rosenbrock evaluated at the point x. ga did not find an especially good solution. The maximum generation in Matlab is $100 * ndv = 200$. Therefore, the algorithm stops after 200 generations. It turns out that the optimum design is far away from the global optimum. In fact, 200 generations are often not enough to find a good optimum design. In order to allow more generations, the following optimoptions function is used to change the maximum allowed generations:

```
options = optimoptions('ga','MaxGenerations',10000);
[x,fval] = ga(@Rosenbrock,ndv,[],[],[],[],lb,ub,[],[],options)
```

With the increased number of generations, the ga algorithm stops with the following output:

```
Optimization terminated: average change in the fitness value less than
options.FunctionTolerance.
x = 1.0354    1.0721
fval = 0.0013
```

At this time, the algorithm stops because the average fitness value does not change much. Now, the optimum design is much closer to the global optimum.

When to use genetic algorithm?

Genetic algorithms are computationally expensive compared to deterministic algorithms. Consequently, their domain of application is problems that would be difficult to solve by gradient-based algorithms or problems that would be out of range for these algorithms. Some of the main advantages of genetic algorithms are the following:

- Genetic algorithms do not require continuity or differentiability of the objective function. They do not need gradients;
- They are robust: they are very insensitive to noise;

- They are modular and therefore portable; because the evolutionary mechanism is separate from the problem representation they can be transferred from problem to problem;
- They are particularly efficient on discrete and combinatorial problems, which are typically difficult to solve by conventional algorithms; they can be used for problems involving both discrete and continuous variables;
- Because they explore the design space with populations, genetic algorithms are "naturally" amenable to parallelization.

6.5. Particle swarm optimization

Many global search algorithms mimic the natural behaviors of animals or insects [*]. Among them particle swarm algorithm is introduced by Kennedy and Eberhart [63] in 1995, inspired by social behavior and movement dynamics of insects, birds and fish. Particle swarm optimization imitates the basic strategy of swarm of bees seeking food. The search direction relies on remembered good locations from the past. For each member of the swarm, the search direction is a compromise between the best point visited by the swarm and the best point visited by the individual.

Similar to genetic algorithms, this algorithm is also based on a population of designs. However, instead of creating new population, the particle swarm algorithm moves the existing designs to better ones. Since this movement to a new design is continuous, this algorithm has an advantage of handling continuous design variables. Similar to other global search algorithm, this algorithm also have stochastic property in search. The algorithm is an efficient and robust global search capability. There is some tuning of its parameters that is needed for peak efficiency, but commonly used values often give reasonably good results. The performance of the algorithm is comparable to that of genetic algorithms, and it is highly suitable for parallel computation. The algorithm has successfully been applied to a wide variety of problems, including neural networks, structural optimization, and shape and topology optimization.

Figure 6-15 illustrates an example of a three-bar truss with two cross-sectional area design variables. Initially 20 particles are randomly distributed in the design space. Each particle will have a position, which are the values of the design variables. As the search progresses, the swarm coalesces at the presumed optimum as is shown in the bottom right figure.

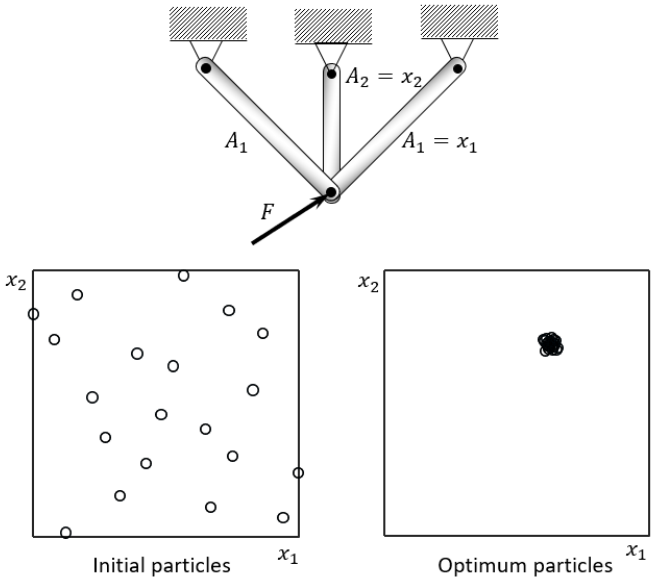


Figure 6-15: Initial and optimum particles of particle swarm optimization.

In the particle swarm optimization algorithm, a particle means a design, and the collection of particles is the population. Let us assume that n be the dimension of design space and m be the number of particles (i.e., designs). These particles are represented by $\mathbf{x}^i = \{x_1, x_2, \cdots, x_n\}^i, i = 1, \cdots, m$. The particle swarm algorithm considers a design as a location of a particle. A design at a point moves to a different location using a design change. Let subscript k represent the current design iteration. Then, the location of a i-th particle at the next iteration can be updated as

$$\mathbf{x}_{k+1}^i = \mathbf{x}_k^i + \Delta\mathbf{x}_{k+1}^i \tag{6.10}$$

where $\Delta\mathbf{x}_{k+1}^i$ is the design change of particle i at iteration $k+1$. The computational cost of the particle swarm algorithm is proportional to the number of particles and the number of iterations: $m \times k$.

The main idea of the particle swarm algorithm is to choose the design change such that the new design can improve the objective function and move toward to global optimum design. In the gradient-based optimization algorithms in Chapter 5, the design change is calculated based on the gradient information. In the particle swarm algorithm, the design change is calculated based on individual and group experiences of the best design. Let $\mathbf{p}^i$ be the best design of particle i up to the current iteration and $\mathbf{p}^g$ be the best design of the group up to the current iteration. The former represents an individual experience, while the latter represents the group experience. Therefore, the design change can be directed toward these well-experienced designs. Therefore, the design change can be calculated as

$$\Delta\mathbf{x}_{k+1}^i = w\Delta\mathbf{x}_k^i + c_1 r_1\left(\mathbf{p}^i - \mathbf{x}_k^i\right) + c_2 r_2\left(\mathbf{p}^g - \mathbf{x}_k^i\right) \tag{6.11}$$

where $w, c_1,$ and c_2 are algorithmic parameters, which determine the importance of each term. The first term $w\Delta\mathbf{x}_k^i$ represents the inertial effect, where the particle tends to move in the direction of the previous iteration. A typical value of w (the inertia fraction) is 0.5. To prevent divergence of the algorithm, the inertia fraction must be smaller than 1. The second term $c_1 r_1\left(\mathbf{p}^i - \mathbf{x}_k^i\right)$ is based on moving towards $\mathbf{p}^i$, the best past position of the ith particle. The size of the move is random, controlled by the random number r_1, uniformly distributed in [0,1]. The multiplier c_1 is used to control the mean of the move distance. A typical value is $c_1 = 2$, in which case the mean of the move is at $\mathbf{p}^i$, and there is a 50% chance of overshooting or undershooting it. The third term $c_2 r_2\left(\mathbf{p}^g - \mathbf{x}_k^i\right)$ is exactly the same structure as the second term, but the target is $\mathbf{p}^g$, which is the group (swarm) best. Again, a typical value for c_2 is also 2. With all three terms present, we get motion in an intermediate direction between the three directions of the three terms.

The procedure of the particle swarm optimization algorithm can be summarized as follows:

1. Initialize Population
   (a) Set algorithmic parameters w, c_1, c_2
   (b) Set $i = 1, k = 1$
   (c) Randomly generate particles $\mathbf{x}_k^i$
2. Optimization iteration
   (a) Evaluate fitness value f_k^i using design particle $\mathbf{x}_k^i$
   (b) If $f_k^i < f_{\text{best}}^i$, then $f_{\text{best}}^i = f_k^i$, $\mathbf{p}^i = \mathbf{x}_k^i$
   (c) If $f_k^i < f_{\text{best}}^g$, then $f_{\text{best}}^g = f_k^i$, $\mathbf{p}^g = \mathbf{x}_k^i$
   (d) If the stopping condition is satisfied go to 3.
   (e) Update particle design change $\Delta\mathbf{x}_{k+1}^i$
   (f) Update particle position vector $\mathbf{x}_{k+1}^i$
   (g) $i = i + 1$. If $i > m$ then increment $k = k + 1$, set $i = 1$.

(h) Go to 2(a).

3. Report results and terminate

Similar to most population-based optimization algorithms, it would be hard to determine the stopping criteria of the particle swarm algorithm. It is partly because there are no optimality criteria for these kinds of algorithms. In addition, there is no mathematical theory available to distinguish the global optimum from local optima. With these difficulties, the convergence of the particle swarm optimization algorithm can be defined in two different ways. (a) All particles converged to a point in the design space, which may or may not be the optimum design, and (b) the individual optimum $\mathbf{p}^i$ and the swarm's best-known position $\mathbf{p}^g$ does not change for many iterations, regardless of how the swarm behaves. In contrast to the gradient-based algorithms, the first corresponds to the case when the design change is less than a threshold, while the second corresponds to the case when the change of the objective function is less than a threshold.

Matlab supports the particle swarm optimization based on the original algorithm of Kennedy and Eberhart [63] with modifications suggested in Mezura-Montes and Coello Coello [64] and in Pedersen [65]. The particle swarm algorithm begins by creating the initial particles, and assigning them initial velocities (design change). It evaluates the objective function at each particle location, and determines the best (lowest) function value and the best location. It chooses new velocities, based on the current velocity, the particles' individual best locations, and the best locations of their neighbors. It then iteratively updates the particle locations (the new location is the old one plus the velocity, modified to keep particles within bounds), velocities, and neighbors.

The Matlab function for the particle swarm algorithm is particleswarm. The following calling convention can be used:

```
[x,fval,exitflag,output] = particleswarm(fun,nvars,lb,ub,options)
```

The particleswarm function takes various input parameters. The fun is the function handle of the fitness function, nvars is the number of design variables, lb and ub are lower- and upper-bounds of design variables, and options are optimization options. As mentioned before, the algorithm depends on the number of particles, which can be changed using the optimization option:

```
options = optimoptions('particleswarm','SwarmSize',50);
```

Once the particle swarm algorithm is finished, it returns with various output results. The variable x includes the optimum design, and fval is the value of the objective function at the optimum design. When the exitflag is positive, it means the algorithm stopped normally, while when it is negative, it means the algorithm stopped due to erroneous reasons. The output structure includes the summary of the solution process, including the number of iterations, the number of function evaluations, and the stopping conditions.

Example 6-7

Optimize the Rosenbrock function in **Example 6-6** using the particle swarm algorithm in Matlab. Try with different swarm sizes and discuss the optimum design, iterations, the number of function evaluations and stopping criteria.

Solution:

The Rosenbrock problem in **Example 6-6** has two design variables in the range of $-3 \le x_1, x_2 \le 3$. The objective function is available in Rosenbrock.m file, and the following Matlab script can be used to solve the optimization problem using the particle swarm algorithm with 10 particles:

```
rng default  % For reproducibility
nvars = 2;
lb = [-3,-3];
ub = [3,3];
options = optimoptions('particleswarm','SwarmSize',10);
[x,fval,exitflag,output] = particleswarm(@Rosenbrock,nvars,lb,ub,options)
```

The optimization algorithm stopped because the relative change in the objective value is less than the tolerance of 1×10^{-1}. The following outputs are returned from the algorithm:

```
x =    0.9683    0.9375
fval =    0.0010
exitflag =     1
output =
    iterations: 67
     funccount: 680
```

The optimum design is close to the global optimum design $\mathbf{x}^* = [1, 2]$ with the value of minimum objective function value of 0. The algorithm is converged after 67 iterations. Since there are 10 particles, each iteration takes 10 function evaluations. Including 10 initial function evaluations, the total number of function evaluations become 680.

The same optimization algorithm can be used with 50 particles by changing option 'SwarmSize' to 50. At this time, the algorithm stopped after consuming the maximum function evaluations 2,000.

```
x =    1.0308    1.0622
fval =   9.6164e-04
exitflag =     1
output =
    iterations: 39
    funccount: 2000
```

Although the final design and final objective function are close to the previous ones, it only performed 39 iterations. This is because each iteration requires 50 function evaluations. Figure 6-16 shows the particle locations at the initial and optimum designs of the Rosenbrock function.

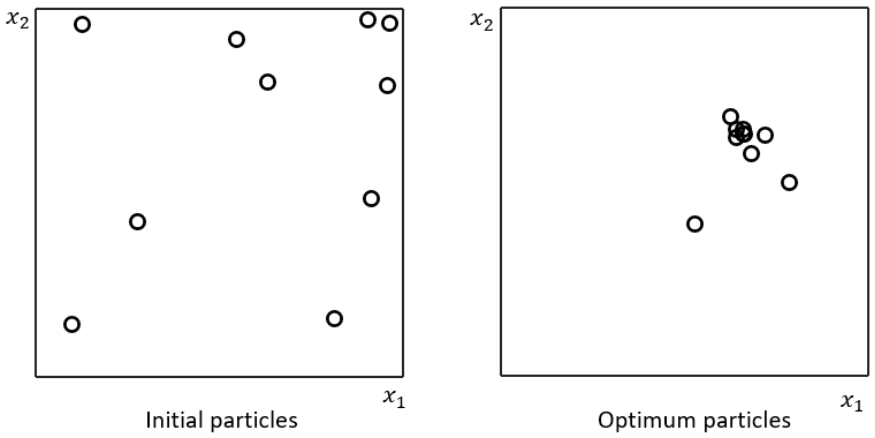


Figure 6-16: Initial and optimum particle distributions of Rosenbrock optimization.

6.6. Simulated annealing optimization

The particle swarm optimization in the previous section is based on social behavior of a bird flock or fish school. The other group of global search algorithms is based on statistical process commonly found in nature. Among them, the simulated annealing algorithm in this section is motivated by studies in statistical mechanics which deal with the equilibrium of a large number of atoms in solids and liquids by gradually decreasing temperature. During the solidification of metals or formation of crystals, for example, a number of solid states with different internal atomic or crystalline structures that correspond to different energy levels can be achieved depending on the rate of cooling. If the system is cooled too rapidly, it is likely that the resulting solid state would have a small margin of stability because the atoms will assume relative positions in the lattice structure to reach an energy state which is only locally minimal. In order to reach a more stable, globally minimum energy state, the process of annealing is used in which the metal is reheated to a high temperature and cooled slowly, allowing the atoms enough time to find positions that minimize a steady-state potential energy. It is observed in the natural annealing process that during the time spent at a given temperature, it is possible to have the system jump to a higher energy state temporarily before the steady state is reached. This characteristic of the annealing process makes it possible to achieve near global minimum energy states.

A computational algorithm that simulates the annealing process was proposed by Metropolis et al. [66] and is referred to as the Metropolis algorithm. Simulated annealing efficiently predicts the behavior of a collection of atoms in equilibrium at a given temperature. At a given temperature T, the algorithm perturbs the position of an atom randomly and computes the resulting change in the energy ΔE of the system. If the energy E at the perturbed state is lower than that of the initial state, $\Delta E < 0$, then the new position of the atom is accepted. On the other hand, if the perturbed position causes an increase in the energy $\Delta E \geq 0$, the new position is accepted with a probability inversely proportional to the energy increase. Accepting higher energy state with a probability is the key mechanism to explore the design space for possible better design far away from the current design.

First, the positive energy change is defined as $\Delta E_+ = \max(0, \Delta E)$. The probability of acceptance, $P(\Delta E)$, of a higher energy state is computed as

$$P(\Delta E) = e^{\left(-\frac{\Delta E_+}{T}\right)} \tag{6.12}$$

Note that when $\Delta E < 0$, $P(\Delta E) = e^0 = 1$. Therefore, the probability is in the range of $0 < P(\Delta E) \leq 1$. If the energy is decreased, it will be accepted with the probability of one. On the other hand, if the energy is increased, it will be accepted with a probability of $P(\Delta E) < 1$. The probability is inversely proportional to the positive energy change. If the temperature of the system is high, then the probability of acceptance of a higher energy state increase. If, on the other hand, the temperature is close to zero, then the probability of acceptance becomes very small. The decision to accept or reject is made by randomly selecting a number $u \sim U(0,1)$ and comparing it with $P(\Delta E)$. If $u < P(\Delta E)$, then the perturbed state is accepted, if $u \geq P(\Delta E)$, the state is rejected.

The temperature T becomes a control parameter and is used to first "melt" the system at a high temperature. At each temperature, a pool of atomic structures is generated by randomly perturbing positions until a steady state energy level is reached (commonly referred to as thermal equilibrium). The iterative simulation is then run at this temperature until the system is considered to be in equilibrium, and only then is the temperature reduced by a small fraction. This cooling and equilibrium process continues until no further improvement becomes possible and the system is considered "frozen" or crystallized, with the design parameters at this condition optimal. The temperature reduction sequence and number of iterations allowed for the system to reach equilibrium are considered analogues to an annealing schedule.

The analogy between the simulated annealing and the optimization of functions with many variables was established recently by Kirkpatrick et al. [67], and Cerny [68]. By replacing the energy state with an objective function f, and using variables x for the configurations of the particles, we can apply the Metropolis algorithm to optimization problems. The method requires only function values. The move in the design space from one point, x^i to another x^j causes a change in the objective function, Δf^{ij} . The temperature T now becomes a control parameter that regulates the convergence of the process. Important elements that affect the performance of the algorithm are the selection of the initial value of the "temperature", T_0, and how to update it. In addition, the number of iterations (or combinations of design variables) needed to achieve "thermal equilibrium" must be decided before the T can be reduced. These parameters are collectively referred to as the "cooling schedule".

The procedure of the simulated annealing algorithm can be summarized as follows:

- Set initial design $\mathbf{x} = \mathbf{x}_0$, objective function $f(\mathbf{x})$, the maximum number of iterations M, and initial temperature T_0
- Loop for cooling schedule
  - Calculate temperature T from the cooling schedule
  - Loop for thermal equilibrium
    - Generate a new candidate design $\mathbf{x}_{\text{new}}$ randomly near the current design $\mathbf{x}$
    - Calculate objective function $f(\mathbf{x}_{\text{new}})$, and $\Delta f = f(\mathbf{x}_{\text{new}}) - f(\mathbf{x})$
    - Calculate the probability of acceptance $P(\Delta f) = e(-\Delta f_+/T)$ with $\Delta f_+ = \max(0, \Delta f)$ and generate a random sample $u \sim U(0,1)$
    - If $P(\Delta f) \geq u$, set $\mathbf{x} = \mathbf{x}_{\text{new}}$
  - End of the thermal equilibrium loop
- End of the cooling schedule loop
- Finish the algorithm with optimum design $\mathbf{x}$ and optimum objective function $f(\mathbf{x})$.

From the algorithm, it is obvious that the algorithm has to determine the maximum number of iterations M, initial temperature T_0, and the cooling schedule.

The definition of the cooling schedule begins with the selection of the initial temperature T_0. If a low value of T_0 is used, the algorithm would have a low probability of reaching a global minimum. The initial

value of T_0 must be high enough to permit virtually all moves in the design space to be acceptable so that almost a random search is performed. Typically, T_0 is selected such that the acceptance ratio X (defined as the ratio of the number of accepted moves to the total number of proposed moves) is approximately $X_0 = 0.95$ [69]. Johnson et al. [70] determined T_0 by calculating the average increase in the objective function, $\overline{\Delta f}^{(+)}$, over a predetermined number of moves and solved

$$X_0 = e^{\left(-\frac{\overline{\Delta f}^{(+)}}{T_0}\right)} \tag{6.13}$$

leading to

$$T_0 = \frac{\overline{\Delta f}^{(+)}}{\ln(X_0^{-1})} \tag{6.14}$$

However, since $\overline{\Delta f}^{(+)}$ is difficult to estimate, the above formula will be an approximate.

Once the initial temperature is set, a number of moves in the variable space are performed by perturbing the design. The number of moves at a given temperature must be large enough to allow the solution to escape from a local minimum. One possibility is to move until the value of the objective function does not change for a specified number, M, of successive iterations. Another possibility suggested by Aarts [71] for discrete-valued design variables is to make sure that every possible combination of design variables in the neighborhood of a steady-state design is visited at least once with a probability of P. That is, if there are S neighboring designs, then

$$M = S \ln\left(\frac{1}{1-P}\right) \tag{6.15}$$

where $P = 0.99$ for $S > 100$, and $P = 0.995$ for $S < 100$. For discrete-valued variables, there are often many options for defining the neighborhood of the design. One possibility is to define it as all the designs that can be obtained by changing one design variable to its next higher or lower value. A broader immediate neighborhood can be defined by changing more than one design variable to its next higher or lower values. For an n-variable problem, the immediate neighborhood has

$$S = 3^n - 1 \tag{6.16}$$

Once convergence is achieved at a given temperature, generally referred to as thermal equilibrium, the temperature is reduced, and the process is repeated.

Many different schemes have been proposed for updating the temperature (cooling schedule). A frequently used rule is a constant cooling update

$$T_{k+1} = \alpha T_k, \qquad k = 0, 1, \cdots, K \tag{6.17}$$

where $0.5 \leq \alpha \leq 0.95$. Nahar [72] fixes the number of decrement steps K, and suggests the determination of the values of the T_k experimentally. It is also possible to divide the interval $[0, T_0]$ into a fixed K number of steps and use

$$T_k = \frac{K-k}{K} T_0, \qquad k = 0, 1, \cdots, K \tag{6.18}$$

The number of intervals typically ranges from 5 to 20.

The use of simulated annealing for structural optimization has been quite recent. Elperin [73] applied the method to the design of a ten-bar truss problem where member cross-sectional dimensions were to be selected from a set of discrete values. Kincaid and Padula [74] used it for minimizing the distortion and internal forces in a truss structure. A 6-story 156-member frame structure with discrete valued variables

was considered by Balling and May [75]. Optimal placement of active and passive members in a truss structure was investigated by Chen et al. [76] to maximize the finite-time energy dissipation to achieve increased damping properties.

An important difference from iterative improvement procedures such as gradient-based methods is that the Metropolis method need not get stuck since transitions out of a local optimum is always possible at a nonzero temperature. Like genetic algorithms, this population-based method is relatively insensitive to noise and does not require gradient information. A major drawback to this method, however, is the problem dependency of algorithm parameters. That is, it is important to use an annealing schedule that is tailored to the type of optimization problem being solved, otherwise the optimizer will perform poorly. Various statistical methods have been proposed, all with limited success, to obtain optimum cooling rates to avoid entrapment in local optima.

Matlab provides the `simulannealbnd` function for solving a minimization problem using the simulated annealing algorithm. The calling convention is similar to other optimization algorithms:

```
[x,fval,exitflag,output] = simulannealbnd(fun,x0,lb,ub,options)
```

The Matlab implementation of `simulannealbnd` is for solving unconstrained optimization problems with bounds. The annealing schedule can be provided as an option, including the initial temperature in `InitialTemperature` and cooling schedule in `TemperatureFcn`.

Example 6-8
Find the minimum of a function $f(\mathbf{x}) = (4 - 2.1x_1^2 + x_1^4/3)x_1^2 + x_1x_2 + (-4 + 4x_2^2)x_2^2$ with the lower- and upper-bounds of design $-64 \leq x_1, x_2 \leq 64$ using the `simulannealbnd` function of Matlab.

Solution:
To implement the objective function calculation, the Matlab file obj.m has the following code:

```
function f = objfun(x)
x1 = x(1);
x2 = x(2);
f = (4-2.1.*x1.^2+x1.^4./3).*x1.^2+x1.*x2+(-4+4.*x2.^2).*x2.^2;
```

The objective function computes the scalar value of the objective function and returns it in its single output argument `f`. To minimize the objective function using `simulannealbnd`, pass in a function handle to the objective function and a starting point `x0` as the second argument. For reproducibility, set the random number stream.

```
x0 = [0.5 0.5];   % Starting point
lb = [-64 -64];
ub = [64 64];
rng default % For reproducibility
[x,fval,exitFlag,output] = simulannealbnd(@objfun,x0,lb,ub)
```

The outputs from Matlab are as follows:

```
x = -0.0896    0.7127
fval = -1.0316
exitFlag = 1
```

```
output =
     iterations: 2465
      funccount: 2484
        message: 'Optimization terminated: change in best function value
                  less than options.FunctionTolerance.'
       rngstate: [1×1 struct]
    problemtype: 'boundconstraints'
    temperature: [2×1 double]
      totaltime: 0.5566
```

It shows that the optimization converged as the change in the objective function is less than the tolerance. The optimum design was $\mathbf{x}^* = (-0.0896, 0.7127)$, where the optimum objective function was -1.0316. The algorithm took 2,465 iterations with 2,484 function evaluations. When the algorithm stopped, the cooling temperatures for both variables were $(0.2058 \times 10^{-1}, 0.2207 \times 10^{-3})$, which was stored in `output.temperature`.

6.7. Exercise

1. For a 2D problem, the current simplex has $f(0,0) = 1, f(1,0) = 2, f(0,1) = 3$. (a) Where will you evaluate f next? (b) If the next two points gave us function values of 4 and 5, respectively, where will you evaluate the functions next? (c) If instead of scenario (b), the next point (reflection point) gave us a function value of 0, where will you evaluate the function next?

2. The first four points selected by the Nelder-Mead algorithm are $(x_1, x_2, f) = 1: (0,0,3),\ 2: (0,1,7),\ 3: (1,1,6),\ 4: (1,0,2)$. Calculate where the algorithm will select the next point, also naming the operation that leads to this point (i.e., reflection, expansion, contraction, or shrinkage).

3. Track and plot the first two iterations of fminsearch on Himmelblau's function starting from (1,1). The form of Himmelblau's function is defined as
$$f(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$$
The function has one local maximum at $\mathbf{x} = (-0.270845, -0.923939)$ with $f_{\max} = 181.617$ and four identical local minima at $\mathbf{x} = (3.0, 2.0), (-2.805118, 3.131312), (-3.779310, -3.283186), (3.584428, -1.848126)$.

4. Consider the following pairs of objective functions to be minimized: $(3,9), (4,1), (10, -3), (-7,11), (-8,14), (7, -17), (6,7), (5,8)$. (a) Divide them into ranks, (b) Calculate the crowding distance of the points in the second rank.

5. Global optimization balances exploration and exploitation. How is that reflected in genetic algorithms?

6. What are all possible child designs of $[0_2/\pm 45/90]s$ and $[\pm 45_2/0]s$ that are balanced and symmetric with uniform crossover?

7. .