2 The Differential Evolution Algorithm

2.1 Overview

2.1.1 Population Structure

DE's most versatile implementation maintains a pair of vector populations, both of which contain *Np* *D*-dimensional vectors of real-valued parameters. The current population, symbolized by $\mathrm{P}_{\mathbf{x}}$, is composed of those vectors, $\mathbf{x}_{i,g}$, that have already been found to be acceptable either as initial points, or by comparison with other vectors:

$$\begin{aligned} \mathrm{P}_{\mathbf{x},g} &= \left(\mathbf{x}_{i,g}\right), \quad i = 0,1,...,Np-1, \;\; g = 0,1,...,g_{\max}, \\ \mathbf{x}_{i,g} &= \left(x_{j,i,g}\right), \; j = 0,1,...,D-1. \end{aligned} \tag{2.1}$$

Indices start with 0 to simplify working with arrays and modular arithmetic. The index, $g = 0, 1, ..., g_{\max}$, indicates the generation to which a vector belongs. In addition, each vector is assigned a population index, *i*, which runs from 0 to $Np - 1$. Parameters within vectors are indexed with *j,* which runs from 0 to $D - 1$.

Once initialized, DE mutates randomly chosen vectors to produce an intermediary population, $\mathrm{P}_{\mathbf{v},g}$, of *Np* mutant vectors, $\mathbf{v}_{i,g}$:

$$\begin{aligned} \mathrm{P}_{\mathbf{v},g} &= \left(\mathbf{v}_{i,g}\right), \quad i = 0,1,...Np-1, \;\; g = 0,1,...,g_{\max}, \\ \mathbf{v}_{i,g} &= \left(v_{j,i,g}\right), \; j = 0,1,...,D-1. \end{aligned} \tag{2.2}$$

Each vector in the current population is then recombined with a mutant to produce a trial population, $\mathrm{P}_{\mathbf{u}}$, of *Np* trial vectors, $\mathbf{u}_{i,g}$:

$$\begin{aligned} \mathrm{P}_{\mathbf{u},g} &= \left(\mathbf{u}_{i,g}\right), \quad i = 0,1,...,Np-1, \;\; g = 0,1,...,g_{\max}, \\ \mathbf{u}_{i,g} &= \left(u_{j,i,g}\right), \quad j = 0,1,...,D-1. \end{aligned} \tag{2.3}$$

During recombination, trial vectors overwrite the mutant population, so a single array can hold both populations.

2.1.2 Initialization

Before the population can be initialized, both upper and lower bounds for each parameter must be specified. These 2*D* values can be collected into two, *D*-dimensional initialization vectors, $\mathbf{b}_{\mathrm{L}}$ and $\mathbf{b}_{\mathrm{U}}$, for which subscripts L and U indicate the lower and upper bounds, respectively. Once initialization bounds have been specified, a random number generator assigns each parameter of every vector a value from within the prescribed range. For example, the initial value ($g = 0$) of the j^{th} parameter of the i^{th} vector is

$$x_{j,i,0} = \mathrm{rand}_j(0,1) \cdot \left(b_{j,\mathrm{U}} - b_{j,\mathrm{L}}\right) + b_{j,\mathrm{L}}. \tag{2.4}$$

The random number generator, $\mathrm{rand}_j(0,1)$, returns a uniformly distributed random number from within the range [0,1), i.e., $0 \le \mathrm{rand}_j(0,1) < 1$. The subscript, *j*, indicates that a new random value is generated *for each parameter*. Even if a variable is discrete or integral, it should be initialized with a real value since DE internally treats all variables as floating-point values regardless of their type.

2.1.3 Mutation

Once initialized, DE mutates and recombines the population to produce a population of *Np* trial vectors. In particular, *differential mutation* adds a scaled, randomly sampled, vector difference to a third vector. Equation 2.5 shows how to combine three different, randomly chosen vectors to create a mutant vector, $\mathbf{v}_{i,g}$:

$$\mathbf{v}_{i,g} = \mathbf{x}_{r0,g} + F \cdot \left(\mathbf{x}_{r1,g} - \mathbf{x}_{r2,g}\right). \tag{2.5}$$

The scale factor, $F \in (0,1+)$, is a positive real number that controls the rate at which the population evolves. While there is no upper limit on *F*, effective values are seldom greater than 1.0.

The *base vector* index, *r*0, can be determined in a variety of ways, but for now it is assumed to be a randomly chosen vector index that is different from the *target vector* index, *i*. Except for being distinct from each other and from both the base and target vector indices, the *difference vector* indices, *r*1 and *r*2, are also randomly selected once per mutant. Figure 2.1 illustrates how to construct the mutant, $\mathbf{v}_{i,g}$, in a two-dimensional parameter space.

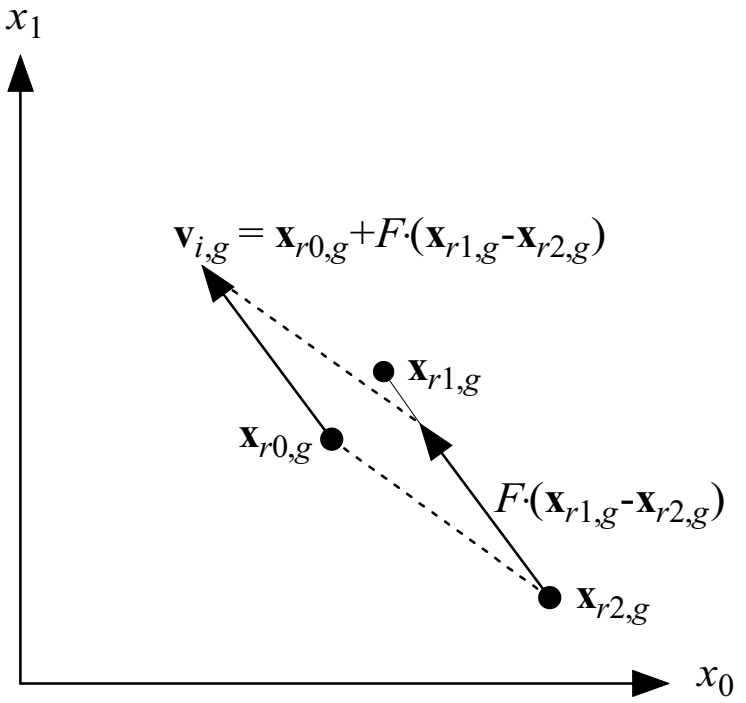

Fig. 2.1. Differential mutation: the weighted differential, $F \cdot (\mathbf{x}_{r1,g} - \mathbf{x}_{r2,g})$, is added to the base vector, $\mathbf{x}_{r0,g}$, to produce a mutant, $\mathbf{v}_{i,g}$.

2.1.4 Crossover

To complement the differential mutation search strategy, DE also employs *uniform crossover*. Sometimes referred to as *discrete recombination*, (dual) crossover builds trial vectors out of parameter values that have been copied from two different vectors. In particular, DE crosses each vector with a mutant vector:

$$\mathbf{u}_{i,g} = u_{j,i,g} = \begin{cases} v_{j,i,g} & \text{if } \left(\text{rand}_j(0,1) \le Cr \text{ or } j = j_{\text{rand}}\right) \\ x_{j,i,g} & \text{otherwise.} \end{cases} \tag{2.6}$$

The crossover probability, $Cr \in [0,1]$, is a user-defined value that controls the fraction of parameter values that are copied from the mutant. To determine which source contributes a given parameter, uniform crossover

compares Cr to the output of a uniform random number generator, $\text{rand}_j(0,1)$. If the random number is less than or equal to Cr, the trial parameter is inherited from the mutant, $\mathbf{v}_{i,g}$; otherwise, the parameter is copied from the vector, $\mathbf{x}_{i,g}$. In addition, the trial parameter with randomly chosen index, j_{rand}, is taken from the mutant to ensure that the trial vector does not duplicate $\mathbf{x}_{i,g}$. Because of this additional demand, Cr only approximates the true probability, p_{Cr}, that a trial parameter will be inherited from the mutant. Figure 2.2 plots the possible trial vectors that can result from uniformly crossing a mutant vector, $\mathbf{v}_{i,g}$, with the vector $\mathbf{x}_{i,g}$.

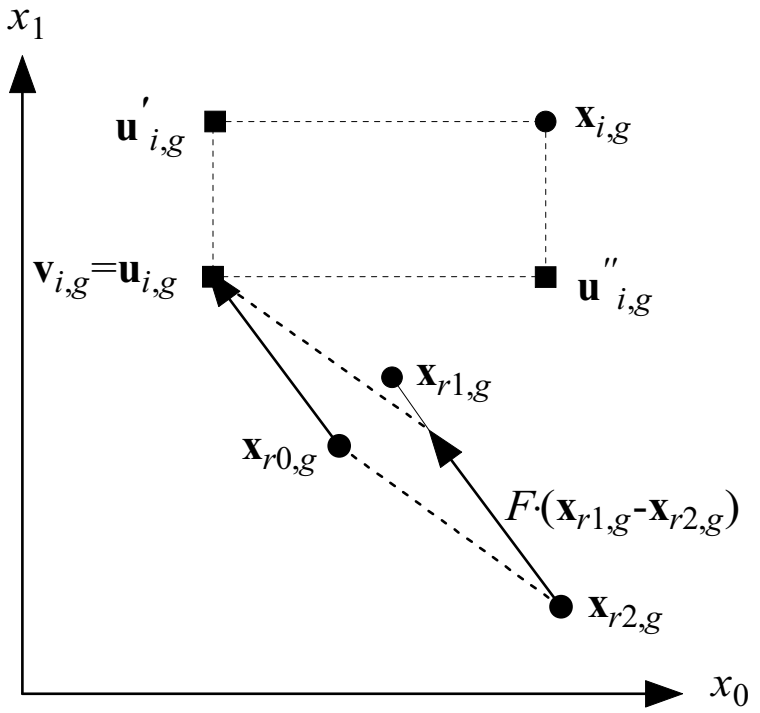

Fig. 2.2. The possible additional trial vectors $\mathbf{u}'_{i,g}$, $\mathbf{u}''_{i,g}$ when $\mathbf{x}_{i,g}$ and $\mathbf{v}_{i,g}$ are uniformly crossed

2.1.5 Selection

If the trial vector, $\mathbf{u}_{i,g}$, has an equal or lower objective function value than that of its target vector, $\mathbf{x}_{i,g}$, it replaces the target vector in the next generation; otherwise, the target retains its place in the population for at least one

more generation (Eq. 2.7). By comparing each trial vector with the target vector from which it inherits parameters, DE more tightly integrates recombination and selection than do other EAs:

$$\mathbf{x}_{i,g+1} = \begin{cases} \mathbf{u}_{i,g} & \text{if } f(\mathbf{u}_{i,g}) \le f(\mathbf{x}_{i,g}) \\ \mathbf{x}_{i,g} & \text{otherwise.} \end{cases} \tag{2.7}$$

Once the new population is installed, the process of mutation, recombination and selection is repeated until the optimum is located, or a prespecified termination criterion is satisfied, e.g., the number of generations reaches a preset maximum, $g_{\max}$.

2.1.6 DE at a Glance

Here are three different ways to describe the DE algorithm known as "classic DE".

Generate-and-Test

The simplicity of DE's generate-and-test loop becomes apparent once Eqs. 2.5–2.7 are combined:

$$u_{j,i,g} = \begin{cases} x_{j,r0,g} + F \cdot (x_{j,r1,g} - x_{j,r2,g}), & \text{if } (\text{rand}_j(0,1) \le Cr \text{ or } j = j_{\text{rand}}), \\ x_{j,i,g} & \text{otherwise.} \end{cases} \tag{2.8}$$

$$j = 0,1,...,D-1; \quad j_{\text{rand}} \in \{0,1,...,D-1\}$$

$$i = 0,1,...,Np-1$$

$$g = 0,1,...,g_{\max}$$

$$r0,r1,r2 \in \{0,1,...,Np-1\}, \quad r0 \ne r1 \ne r2 \ne i$$

$$\mathbf{x}_{i,g+1} = \begin{cases} \mathbf{u}_{i,g} & \text{if } f(\mathbf{u}_{i,g}) \le f(\mathbf{x}_{i,g}), \\ \mathbf{x}_{i,g} & \text{otherwise.} \end{cases}$$

C-Style Pseudo-code

Figure 2.3 presents C-style pseudo-code for classic DE. The vector indices $r0$, $r1$ and $r2$ are all different and distinct from the target index, i. In addition, selection is delayed until the trial population is complete.

```
// initialize...

do // generate a trial population
{
   for (i=0; i<Np; i++)  // r0!=r1!=r2!=i
   {
      do r0=floor(rand(0,1)*Np); while (r0==i);
      do r1=floor(rand(0,1)*Np); while (r1==r0 or r1==i);
      do r2=floor(rand(0,1)*Np); while (r2==r1 or r2==r0 or r2==i);
      jrand=floor(D*rand(0,1));

      for (j=0; j<D; j++)  // generate a trial vector
      {
         if (rand(0,1)<=Cr or j==jrand)
         {
            u_{j,i}=x_{j,r0}+F*(x_{j,r1}-x_{j,r2});  //check for out-of-bounds ?
         }
         else
         {
            u_{j,i}=x_{j,i};
         }
      }
   }

   // select the next generation

   for (i=0; i<Np; i++)
   {
      if ( f(u_i)<=f(x_i) ) x_i=u_i;
   }
} while (termination criterion not met);
```

Fig. 2.3. Classic DE; $0 \leq \text{rand}(0,1) < 1$ so that indices never equal *Np*.

Flow Chart

Figure 2.4 shows a flow chart of DE. That *r*0, *r*1, *r*2 and *i* are distinct indices is not made explicit in this figure.

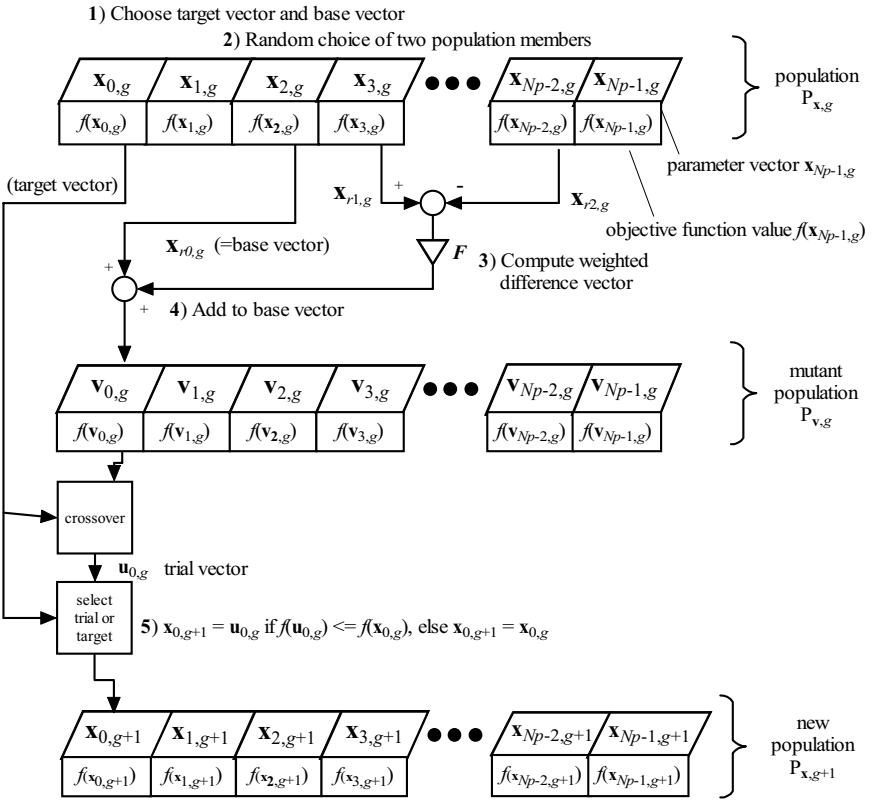

Fig. 2.4. A flow chart of DE's generate-and-test loop

2.1.7 Visualizing DE

The Difference Vector Distribution

Figure 2.5a shows the difference vectors formed by all possible pairings of nine vectors. Transporting the difference vectors to a common origin more clearly shows their distribution (Fig. 2.5b). Because all difference vectors have both a negative counterpart and an equal chance of being chosen, their distribution's mean is zero.

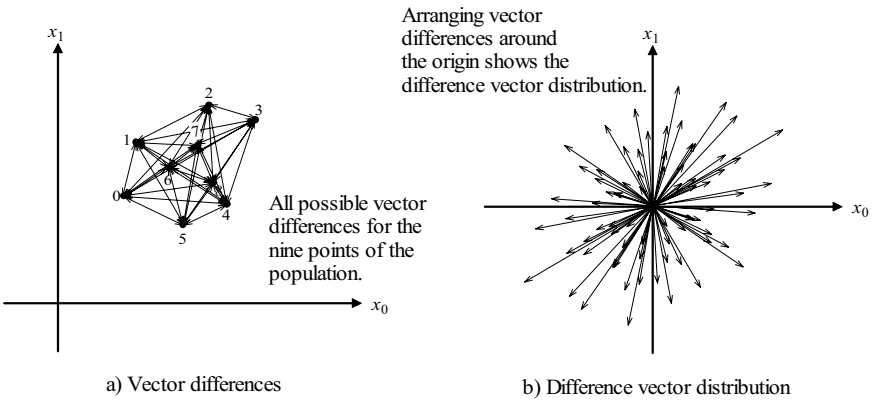

Fig. 2.5. Nine vectors **a**, and their corresponding difference distribution **b**

Scaling vector differences ensures that trial vectors do not duplicate existing points (Fig. 2.6a). In addition, scaling can shift the focus of the search between local and global. Figure 2.6b illustrates that the difference vector distribution contains a substantial number of vectors whose considerable length reduces the probability that vectors will become trapped in a local minimum.

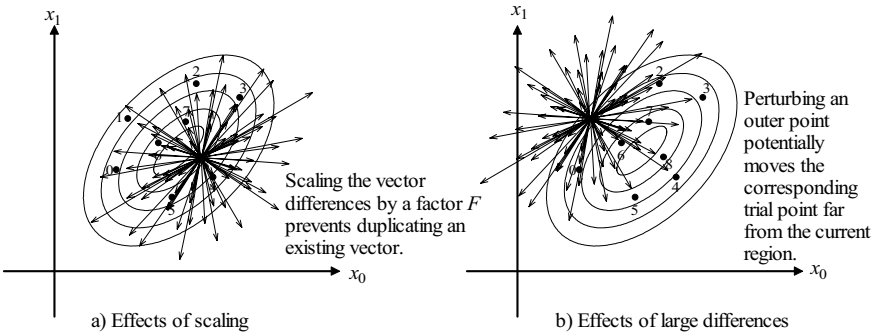

Fig. 2.6. The effects of scaling **a**, and large vector differences **b**

Contour Matching

One of the biggest advantages that difference vectors afford is that both a step's size and its orientation automatically adapt to the objective function landscape. The series of plots in Figs. 2.7–2.13 demonstrate this property

for the "peaks" function (Eq. 1.16). For clarity, the difference vector distribution plot only shows the difference vector endpoints.

As it evolves, the population coalesces around competing minima (Figs. 2.7–2.10). During this phase, the difference distribution is multi-modal, like the function itself. It contains not only steps adapted to searching within each basin, but also larger steps capable of transporting vectors between basins and beyond. Once the population settles into the optimal basin (Figs. 2.11–2.13), the difference vector distribution becomes uni-modal and steps exhibit both a scale and an orientation that is appropriate for a local search.

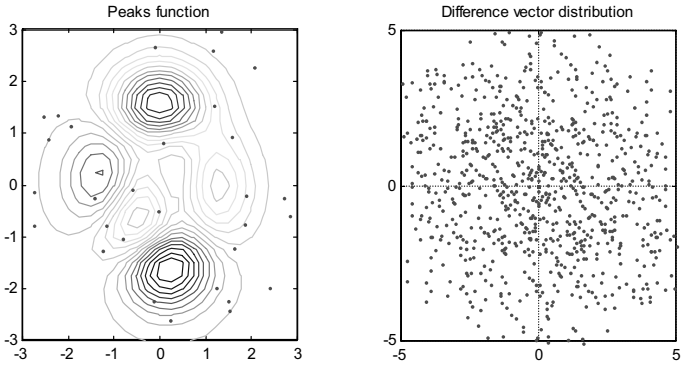

Fig. 2.7. Generation 1: DE's population and difference vector distributions

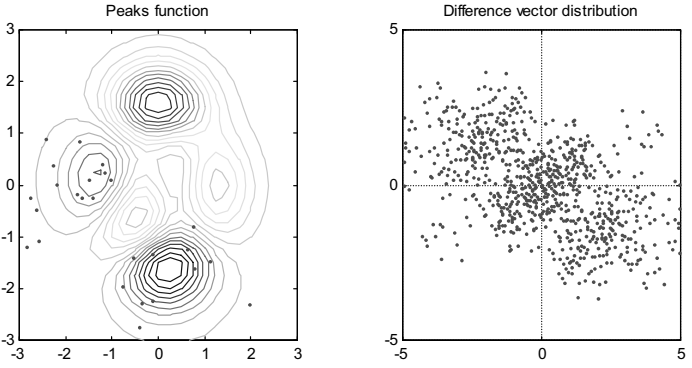

Fig. 2.8. Generation 6: The population coalesces around the two main minima

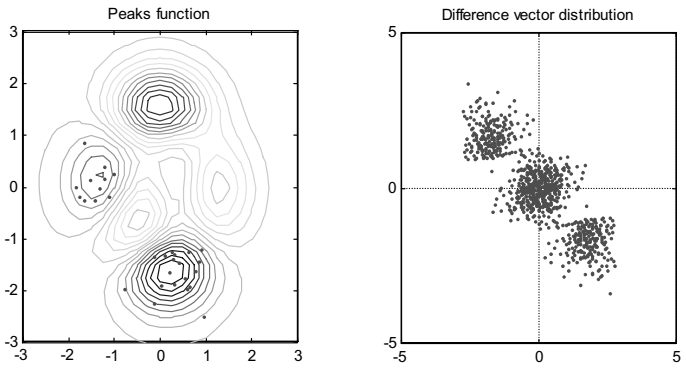

Fig. 2.9. Generation 12: The difference vector distribution contains three main clouds – one for local searches and two for moving between the two main minima.

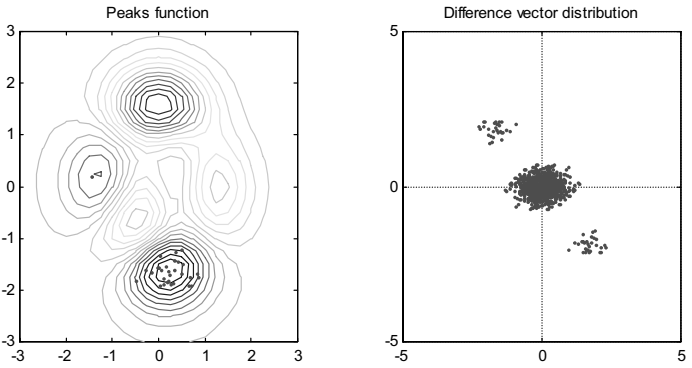

Fig. 2.10. Generation 16: The population is concentrated on the main minimum.

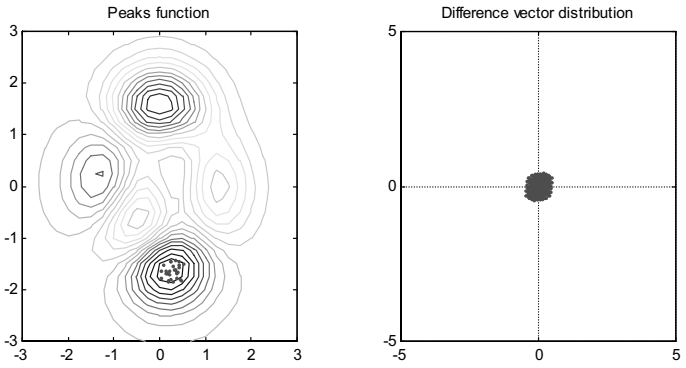

Fig. 2.11. Generation 20: Convergence is imminent. The difference vectors automatically shorten for a fine-grained, local search.

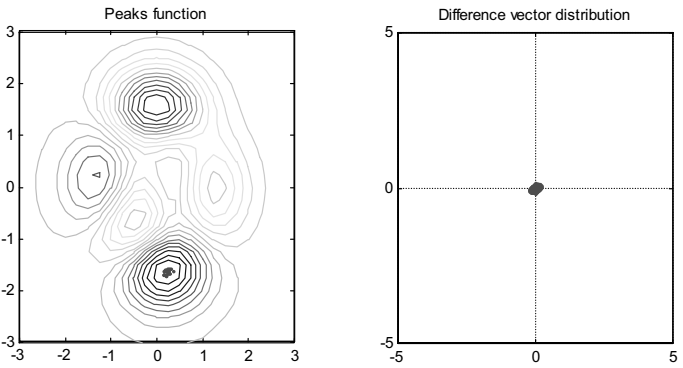

Fig. 2.12. Generation 26: The population has almost converged.

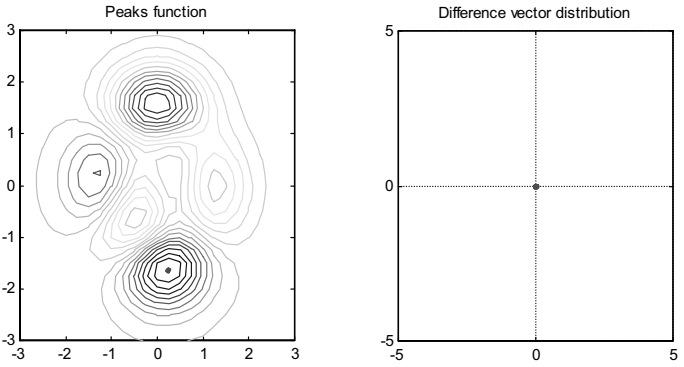

Fig. 2.13. Generation 34: DE finds the global minimum.

2.1.8 Notation

The technical name for the method illustrated in this overview is "DE/rand/1/bin" because the base vector is *rand*omly chosen, *1* vector difference is added to it and because the number of parameters donated by the mutant vector closely follows a *bin*omial distribution. More often, however, this book refers to this method simply as "classic DE". This version will probably suffice for most applications, but a number of variations are possible, each with its own strengths and weaknesses. The most successful of these alternative strategies will be explored later in this chapter, but first the next few sections examine the details missing from this brief overview.

2.2 Parameter Representation

DE encodes all parameters as floating-point numbers, regardless of their type. Even integer and discrete variables are encoded as real values to add diversity to their difference distributions. Specific advice for handling integer and discrete variables is given in Sect. 4.2. The point being made here is that encoding continuous parameters as floating-point numbers and manipulating them with arithmetic operators offer several significant advantages over the traditional GA "bit flipping" approach to continuous parameter optimization. Advantages include:

- ease of use
- efficient memory utilization
- lower computational complexity – scales better on large problems
- lower computational effort – faster convergence
- greater freedom in designing a mutation distribution.

The next subsection exposes the shortcomings of the standard GA coding scheme, while the subsequent subsection elaborates the advantages that floating-point arithmetic confers on a real-parameter optimizer.

2.2.1 Bit Strings

Standard GA Encoding

Typically, GAs encode a continuous parameter, x, as an integer string of q bits, a_k, $k = 0, 1, \ldots q - 1$, each of which is a coefficient for a power of 2:

$$x = b_{\mathrm{L}} + \frac{\left(b_{\mathrm{U}} - b_{\mathrm{L}}\right)}{2^q - 1} \cdot \sum_{k=0}^{q-1} a_k 2^k . \tag{2.9}$$

When decoded, integers are normalized by a factor of $2^q - 1$ and multiplied by $b_{\mathrm{U}} - b_{\mathrm{L}}$ so that values span the range between a parameter's upper and lower bounds, b_{U} and b_{L}, respectively. Assuming that equal resources are devoted to each parameter, a vector of D parameters will require $l = q \cdot D$ bits in all.

For functions with independent parameters, both theory and experiment suggest that the optimal mutation rate, i.e., the probability that a bit should be inverted, or "flipped", is $p_m = 1/l$ (Mühlenbein 1992; Potter and DeJong 1994). The problem with the GA approach is that even on uni-modal objective functions, the computational effort to optimize a parameter is a

function of l that depends on a parameter's value. For example, if the initial parameter value is $x = 15$ ($x = 01111$ binary) and the optimal value is $x = 16$ ($x = 10000$ binary), then 5 bits must *simultaneously* be flipped to make the final improving move. When $p_m = 1/l$, the probability of this event is $p = (1/l)^5$. Because this "Hamming cliff" prevents *incremental* improvement, $x = 15$ is one of many *local* optima even if the objective function is uni-modal. In effect, the function that maps bit strings to real-parameter values is itself multi-modal (Bäck 1993).

By contrast, progress does not depend on simultaneously flipping multiple bits if the optimum happens to be $x = 0$. Instead, inverting non-zero bits in any *sequence* produces a series of parameter values each of which is closer to $x = 0$ than the last. If the objective function is separable and uni-modal, these intermediate steps constitute improving moves. In this *very* special case, the computational complexity to optimize a parameter is constant at $O(l \cdot \ln(l))$ (Salomon 1996b). The factor, $\ln(l)$, occurs because the neighborhoods around parameters that are already optimized continue to be re-sampled (Salomon 1997). In the worst case scenario, however, all q bits must be inverted to make an improving move, so the upper bound on the computational complexity for optimizing an independent parameter of a *uni-modal* function becomes $O(l^q \cdot \ln(l))$.

The requirement that all q bits simultaneously be inverted is also a demand when the objective function is separable and *multi-modal*. For example, it may be that two competing local minima are positioned at points whose representations differ at each bit position. Since an improving move from one local minimum to the other must simultaneously change all q bits, the complexity is $O(l^q \cdot \ln(l))$ (Salomon 1996b). That the computational complexity to optimize an independent parameter is the same in the worst case regardless of whether the function is uni- or multi-modal reflects the aforementioned fact that the standard GA coding scheme imposes multi-modality on even uni-modal objective functions (Bäck 1993).

Gray Codes

Gray codes eliminate Hamming cliffs by reassigning bit groupings to integers so that representations for adjacent integers differ by a single bit, i.e., so that the Hamming distance between consecutive integers is 1 (Wright 1991). As long as the objective function is both uni-modal and separable, sequentially flipping single bits in Gray-coded variables can always produce monotonously decreasing objective function values regardless of both the starting point and the optimal parameter value. Since it no longer matters what the optimal parameter value is, the complexity for optimizing a separable, uni-modal function with Gray codes when $p_m = 1/l$ is constant at

$O(l \cdot \ln(l))$ (Salomon 1996b). Because of their constant low complexity, Gray codes are more efficient than the standard GA representation when the objective function is uni-modal (Bäck 1993). If, however, the objective function is multi-modal, then all bits must be inverted simultaneously in the worst case scenario, so the computational complexity again rises to $O(l^q \cdot \ln(l))$ – the same complexity demonstrated for standard GA coding (Salomon 1996b).

2.2.2 Floating-Point

Unlike the standard GA representation in which all bits are potentially significant, the floating-point format retains only a limited number of significant digits. For example, the ANSI C **float** data type encodes a real number with $q = 32$ bits. Twenty-four bits are dedicated to precision, while the remaining eight bits are assigned to an exponent that locates the decimal point. By contrast, a fixed-point integer variable requires 256+ bits to span as many orders of magnitude as the **float** data type. In the final answer, most of the bits in this very long integer format will be either leading zeros or bits of unneeded precision. By contrast, the floating-point format retains only a limited number of significant bits while spanning a vast dynamic range with minimal resources.

The floating-point format is convenient not only because it can efficiently handle parameter values that span a wide dynamic range, but also because most modern programming languages support common floating-point formats. No special routines are needed to define, input, manipulate or output a floating-point value. When representing continuous parameters in floating-point, the encoding process is transparent to the user.

Logical Versus Arithmetic Operators

GAs typically operate on bit strings with *logical* operators like the XOR (exclusive or) which has the effect of inverting specified bits. By contrast, DE and other floating-point optimizers *add* a floating-point deviation to one or more parameters. Compared to bit flipping, arithmetic provides two benefits: it reduces the complexity of the algorithm and it provides greater flexibility in designing a mutation distribution.

The most efficient way for an EA to optimize a function with independent parameters is to change one parameter at a time before evaluating the result (Salomon 1996a). Typically, both standard and Gray-coded GAs implement this strategy by setting $p_m = 1/l$ so that, on average, only one parameter value changes per function evaluation (Potter and DeJong 1994).

For EAs that add a small deviation to a floating-point parameter, the corresponding mutation probability is $p_m = 1/D$ – a value which also, on average, perturbs just one parameter before evaluating the result (Mühlenbein and Schlierkamp-Voosen 1993; Salomon 1996a).

When the objective function is multi-modal, all bits in an *independent* floating-point parameter may have to be set to the correct value to make progress. If there are q bits in the floating-point representation, then the probability of making progress in this worst case scenario is $(1/2)^q$. While this number may be very small, it is constant and independent of D. As a result, the computational complexity for optimizing separable, multi-modal functions with floating-point representations and arithmetic operators is $O(D \cdot \ln(D))$ (Salomon 1996a). Compared to the $O(l^q \cdot \ln(l))$ complexity for optimizing an independent, Gray-coded parameter of a multi-modal function, the floating-point format representation is faster by a factor of up to $q \cdot l^{q-1} \cdot (1 + \ln(q)/\ln(l))$. The rules of complexity mathematics (Beckman 1980), however, replace *leading* constants, like q, with 1 and substitute 0 for terms like $\ln(q)/\ln(l)$ that are negligible for large l. Under these rules, the ratio of Gray to floating-point complexities reduces to l^{q-1} (Salomon 1996b).

Parameter dependence amplifies this disparity between the computational complexity of the Gray and floating-point approaches. For example, if a multi-modal function has two parameters that depend on each other, then progress in the worst case scenario will require flipping all bits in both parameters simultaneously. The probability of this event is $p = (1/l)^{2q}$ and the corresponding computational complexity is $O(l^{2q} \cdot \ln(l))$. Under similar circumstances, all bits in both parameters' floating-point representations also must be changed. When $p_m = 1/D$, this event occurs with a probability of $(1/D)^2$, so the computational complexity for optimizing two, dependent, floating-point parameters of a multi-modal function is $O(D^2 \cdot \ln(D))$. Under the rules of complexity mathematics, the gain over Gray-coded parameters rises to $l^{2(q-1)}$.

Crafting a Mutation Distribution

Arguably the most important advantage that floating-point arithmetic confers on a real-parameter optimizer is the freedom to decide how perturbations are distributed. Because floating-point's computational complexity does not depend on the mutation operator's probability density, distributions can be crafted to implement a particular search strategy (Salomon 1996b). For example, the Breeder Genetic Algorithm perturbs parameters with non-adaptive step sizes that are distributed according to a power law

(Mühlenbein and Schlierkamp-Voosen 1993). Evolution Strategies (Bäck and Schwefel) and Fast Evolution Strategies (Yao and Liu 1997) adaptively modify steps sampled from Gaussian and Cauchy distributions, respectively. The same freedom that these floating-point optimizers enjoy also allows DE to tap the pool of vector differences as its mutation distribution.

2.2.3 Floating-Point Constraints

The number of bits that a floating-point format dedicates to an exponent limits the minimum and maximum values that it can represent. These limits are rarely exceeded in practical applications because physical properties of such extreme magnitude are uncommon. Of greater consequence for DE is the number of significant digits (precision) that a format supports. If the objective function contains terms that differ by many orders of magnitude, contributions from smaller terms will be lost if there are not enough significant bits available. For example, the **float** data type holds about seven decimal digits of precision. If two numbers differ by more than seven decimal orders of magnitude, then the smaller contribution is not taken into account.

$$\begin{aligned} &x = 1.2 \times 10^{10}, \\ &y = 17, \\ &x + y = 12000000017 \rightarrow 1.200000 \times 10^{10} = x. \end{aligned} \tag{2.10}$$

For the same reason, the lack of precision can be a problem not only when computing the objective function, but also when forming vector differences. Because DE relies on vector differences, the inability to record the effect of small perturbations might cause DE to stagnate (Zimmons n.d.).

In most cases, the **double** format with 15 digits of decimal precision will be enough. Because they evaluate high-order polynomials, however, functions like the high-dimensional versions of the Chebyshev function (see Appendix) require **long double**s. Except for requiring additional memory and bandwidth, there is little penalty for declaring **long double**s and their 19 digits of decimal precision because floating-point units compute values to full precision by default.

2.3 Initialization

In order for DE to work, the initial population must be distributed throughout the problem space. One-point optimizers do not require this initial diversity and even the (1, λ)-ES begins with a single point. If, however, DE is initialized with Np replicas of a single vector, uniform crossover and differential mutation will only clone more replicas. Consequently, DE requires a predefined *probability distribution function*, or PDF, to seed the initial population. When specifying an initial distribution, steps must be taken to ensure that its scale sufficiently broad.

2.3.1 Initial Bounds

As a matter of convenience, test function parameters are often initialized with values that are constrained to lie between a single set of upper and lower bounds. By contrast, bounds for parameters that define real-world objective functions are seldom equal, often because the parameters they delimit correspond to different physical or mathematical entities. In many cases, the existence of natural physical limits or logical constraints makes prescribing bounds for each parameter straightforward. For example, ordinary optical glass can never have an index of refraction less than or equal to 1, nor can a gear have less than one tooth. In cases like these where parameter limits are inviolable, initialization bounds should not only delimit the initial population, but also constrain the subsequent search. Section 4.3.1 discusses several methods for keeping parameters constrained within pre-specified bounds.

Far Initialization

When parameters exhibit no obvious limits, their upper and lower bounds, $b_{j,\mathrm{U}}$ and $b_{j,\mathrm{L}}$, respectively, should be set so that the initial bounding box they define encompasses the optimum. If the optimum's general location is uncertain, then the possibility exists that it lies outside the initial bounding box. Figure 2.14 shows an example of *far initialization* in which the upper parameter limit has been reduced to the point where the initial bounding box no longer contains the optimum, $\mathbf{x}^*$. In cases of far initialization, bounds on otherwise unconstrained parameters must be ignored once the population has been initialized so that DE can explore beyond the initial bounding box.

Table 2.1 records the effect that far initialization has on DE's ability to discover the optima of ten common test functions (descriptions of test

functions can be found in the Appendix). Although each function has a different set of initialization bounds, in each case, these bounds define a *D*-dimensional box that encloses the function's global optimum.

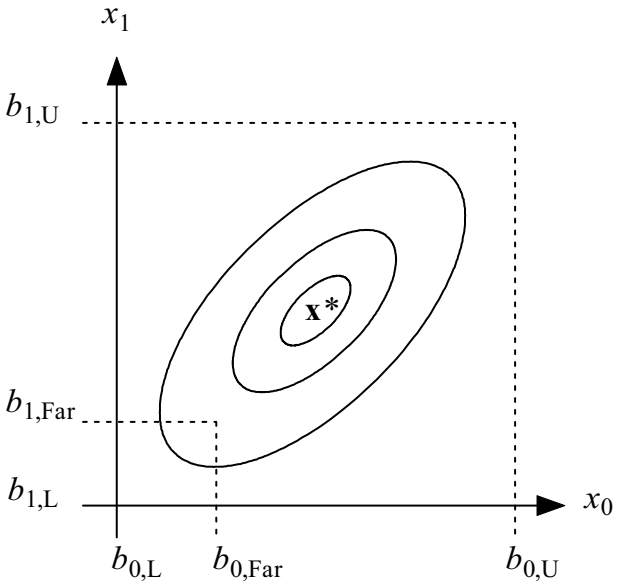

Fig. 2.14. Far initialization shrinks the initial bounding box so that it no longer contains the optimum, $\mathbf{x}^*$.

For the results in Table 2.1, each parameter was initialized with a uniformly distributed random value from within a range that has been reduced by a factor, *h*, when compared to the originally prescribed bounds:

$$x_{j,i,0} = b_{j,\mathrm{L}} + h \cdot \mathrm{rand}_j(0,1) \cdot \left(b_{j,\mathrm{U}} - b_{j,\mathrm{L}}\right). \tag{2.11}$$

After far initializing the population with the given value of *h*, bounds were relaxed to their normal values to constrain the subsequent search.

For each of the functions in Table 2.1, the initial bounding box encloses the optimum when $h = 1$. Setting $h \leq 0.1$ far initializes the population by restricting it to a corner of the original bounding box where it cannot surround the optimum. Table 2.1 reports the average number of function evaluations ("Evals.") taken to find a point whose objective function value differs from the optimum objective function value by less than a preset minimum. Finding such a point within the maximum allowed number of generations constitutes a success; otherwise, the trial is considered to be a failure. (See the Appendix for details on the minimum function value to reach.) Only "successes" contribute to the results in Table 2.1. The fraction

of successful trials, P, records the impact of failures. Results are 100-trial averages obtained using classic DE with $F = Cr = 0.9$ and $r0 \neq r1 \neq r2 \neq i$ (distinct indices).

Table 2.1. The effects of far initializing DE with a uniformly random population

Function	D	Np	$h = 1$		$h = 0.1$		$h = 0.01$	
			Evals.	P	Evals.	P	Evals.	P
Sphere	10	30	30,994.5	1	31,514.9	1	31,722.2	1
Ridge	10	30	48,520.2	1	48,825.5	1	48,820.2	1
Rosenbrock	10	30	59,643.4	1	59,721.9	1	60,315.4	1
Chebyshev	9	30	69,522.1	1	72,211.3	1	71,068.5	1
Ackley	10	30	48,385.2	1	49,853.3	0.90	–	0
Rastrigin	5	100	59,840.4	1	60,199.2	1	–	0
Schwefel	5	100	16,245.6	1	22,432.4	0.25	–	0
Griewangk	5	100	19,4202	0.98	19,5551	0.99	19,4921	0.99
Langerman	5	100	38,405.7	0.98	37,196.1	0.54	34,873.2	0.21
Michalewicz	5	100	27,749.5	1	29,291.6	0.95	32,061.6	0.96

As Table 2.1 shows, far initialization's effect on the sphere, ridge, Rosenbrock, Chebyshev, Michalewicz and Griewangk functions is minimal. In most cases, far initialization penalizes these six functions with a very slight increase in the average number of function evaluations and a very slight decrease in the estimated probability of success. For both the sphere and ridge functions, this result is not surprising. Both functions are uni-modal and convex, so neither poses obstacles to the population's expansion toward the minimum. (Pictures of the two-dimensional versions for many of the test functions used in this book appear in the Appendix.) Rosenbrock's function is also uni-modal, but unlike the sphere it is non-convex. At least in the case of Rosenbrock's function, non-convexity does not impede DE's ability to locate the minimum when far initialized.

Unlike the sphere, ridge or Rosenbrock functions, the remaining functions in Table 2.1 are all multi-modal. Optimal parameter values for the Chebyshev function vary greatly in magnitude and restricting initial values to a small range means that some parameter values must inflate many orders of magnitude to be on par with their optimal values. Table 2.1 shows that except for a slight increase in the number of function evaluations, diminishing the value of h did not significantly impact DE's ability to converge on the Chebyshev optimum. Similarly, far initialization did not significantly affect DE's performance on either Michalewicz's or Griewangk's function.

DE became unreliable, however, when far initializing Langerman's function and failed altogether on the Ackley, Rastrigin and Schwefel func-

tions once $h = 0.01$. For these highly multi-modal functions, the entire initial population can land inside a single, non-optimal, local basin of attraction when h becomes too small. If competing basins are sufficiently far apart, then classic DE cannot generate difference vectors large enough to escape the local basin. Thus, it is important to use a bounding box of sufficient size when initializing multi-modal functions with a uniform random distribution.

Initializing with a Constant

Occasionally, it may prove productive to experiment with a design by holding one or more of its parameters constant while optimizing the remaining variables. DE automatically leaves a parameter unchanged during optimization if every vector is initialized with the same value for the given parameter. When all vectors have the same value for a parameter, every differential they combine to create for that parameter will be zero. Furthermore, uniform crossover does not change parameter values, so a parameter initialized with a single constant value will never change.

2.3.2 Initial Distributions

DE can be initialized with either a uniform or a non-uniform distribution. The decision regarding which to use depends on how much is known about the location of the optimum. If the optimum's location is fairly well known, a Gaussian distribution may prove somewhat faster, although it may also increase the probability that the population will converge prematurely. In general, uniform distributions are preferred, since they best reflect the lack of knowledge about the optimum's location. The next section looks at two common uniform distributions.

Uniform Distributions

Distributing initial points with random uniformity is not mandatory, but experience has shown $\text{rand}_j(0,1)$ to be very effective in this regard. In general, any distribution that uniformly covers the search domain and contains a degree of irregularity or randomness should serve well for initializing the vector population. For example, Hammersley and Halton point sets are often used in the field of numerical integration (Halton and Weller 1964). Based on prime numbers, these pseudo-random distributions are both uniform and irregular, but lack points in close proximity, i.e., they have a minimum resolution that increases as the number of points in the sample increases. Figure 2.15 gives C-style pseudo-code for computing Halton

points in up to ten dimensions. The function, halton(*i*,*j*), takes the population and parameter indices as input and returns a (rational) number belonging to the interval [0,1).

```
halton(i,j)
{
   prime[10]=[2,3,5,7,11,13,17,19,23,29];
   p1=prime[j];
   p2=p1;
   sum=0;

   do
   {
      x=i%p1; // "%" is the modulo operator
      sum=sum+x/p2;
      i=floor(i/p1);
      p2=p2*p1;
   }while (i>0);

   return(sum);
}
```

Fig. 2.15. C-style pseudo-code for generating Halton point sets, $D \leq 10$

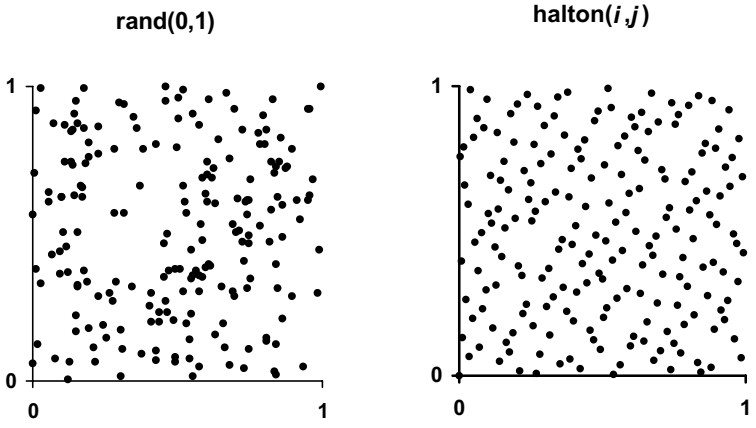

Fig. 2.16. Two hundred points distributed with random uniformity (left) and according to a two-dimensional Halton point set (right).

Figure 2.16 compares the uniform random and Halton distributions in two dimensions. The Halton distribution is more even, but the random distribution displays a wider range of difference vector magnitudes.

Table 2.2 shows how the random and Halton distributions affect DE's performance by reporting the average number of function evaluations ("Evals.") taken to find a point whose objective function value differs from the optimum objective function value by less than a preset minimum. Finding such a point within the maximum allowed number of generations constitutes a success; otherwise, the trial is considered to be a failure. (See Appendix for details on the minimum function value to reach.) Only "successes" contribute to the results in Table 2.2. The fraction of successful trials, P, records the impact of failures. Results are 100-trial averages obtained with classic DE, $F = Cr = 0.9$, distinct indices and with bound constraints imposed. Results for the random uniform distribution have been copied from Table 2.1 ($h = 1$).

Table 2.2. Comparing the effects of uniform initial distributions on performance

Function	D	Np	$\text{rand}_j(0,1)$		halton(i,j)	
			Evals.	P	Evals.	P
Sphere	10	30	30,994.5	1	30,971.1	1
Ridge	10	30	48,520.2	1	48,346.8	1
Rosenbrock	10	30	59,643.4	1	59,406.2	1
Chebyshev	9	30	69,522.1	1	72,611.6	1
Ackley	10	30	48,385.2	1	48,354.7	1
Rastrigin	5	100	59,840.4	1	60,019.2	1
Schwefel	5	100	16,245.6	1	16,203.2	1
Griewangk	5	100	19,4202	0.98	18,8279	1
Langerman	5	100	38,405.7	0.98	39,610.2	0.99
Michalewicz	5	100	27,749.5	1	27,130.7	0.98

As Table 2.2 shows, it matters little whether the population is initialized with $\text{rand}_j(0,1)$, or according to

$$x_{j,i,0} = b_{j,\text{L}} + \text{halton}_j(i,j) \cdot \left(b_{j,\text{U}} - b_{j,\text{L}}\right). \tag{2.12}$$

In every case, both the fraction of successful trials and the average number of function evaluations they required were virtually the same regardless of which uniform distribution initialized the population. To generate a different point set, a different range of prime numbers should be used.

Gaussian Distribution

Uniform distributions are reliable, but populations can also be non-uniformly initialized. For example, Fig. 2.17 plots 200 points distributed according to a two-dimensional multi-normal distribution whose mean vector value is $\boldsymbol{\mu} = 0.5$ and whose covariance matrix is $\mathbf{C} = \sigma^2 \cdot \mathbf{I}$, where $\sigma = 0.5$ and $\mathbf{I}$ is the identity matrix, i.e., N(**0.5**, 0.25·**I**). This choice centers the (symmetrical) distribution in the bounding box and places standard deviates (along coordinate axes) on its surface. Unlike the Halton distribution in Eq. 2.12, the multi-normal distribution, whose output is a vector, is sampled only once per initial vector.

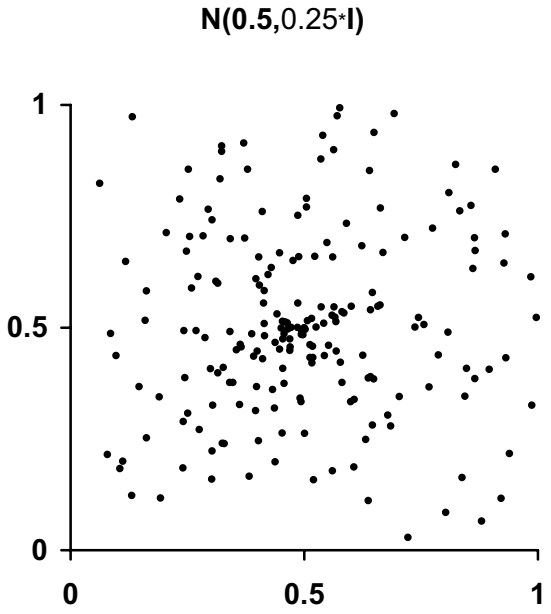

Fig. 2.17. A two-dimensional Gaussian-distributed initial population with mean of 0.5 and a standard deviation of 0.5, i.e., N(**0.5**, 0.25·**I**)

Table 2.3 details how classic DE's performs when the initial population is distributed according to a multi-normal distribution. Unlike Eq. 2.12 in which a new random value is generated for each parameter, the distribution used in both Fig. 2.17 and Table 2.3 generates a single instance of a multi-normally distributed random vector for each initial point. In both cases, the distribution's mean vector is $\boldsymbol{\mu} = (0.5, 0.5, \ldots 0.5)$ and its covariance matrix is $\mathbf{C} = 0.25 \cdot \mathbf{I}$. A comparison with Table 2.2 shows that when

the population is *not* far initialized ($h = 1$), it makes little difference whether the initial distribution is uniform or Gaussian. The sole exception is Ackley's function. Although initializing Ackley's function with a Gaussian distribution left convergence speed unchanged, it significantly degraded DE's probability of success.

Once the population is far initialized ($h \leq 0.1$), failures become more likely. When compared with Table 2.1, the results in Table 2.3 show that a population far initialized with a Gaussian distribution is less likely to be successful on multi-modal functions than a uniformly distributed one. In every case where uniform distributions failed, the Gaussian-distributed population failed more often.

Table 2.3. Far initialization with a ten-dimensional multi-normal distribution

Function	*D*	*Np*	$h = 1$		$h = 0.1$		$h = 0.01$	
			Evals.	P	Evals.	P	Evals.	P
Sphere	10	30	31,801.1	1	31,937.7	1	32,967.6	1
Ridge	10	30	48,483.9	1	48,919.4	1	49,569.9	1
Rosenbrock	10	30	60,198.4	1	60,897.4	1	61,056.4	1
Chebyshev	9	30	72,972.6	1	72,233.6	1	70,129.5	1
Ackley	10	30	48,472.5	0.02	–	0	–	0
Rastrigin	5	100	59,627.1	1	61,125.9	0.71	–	0
Schwefel	5	100	17,406.2	0.97	33,750.1	0.67	–	0
Griewangk	5	100	19,0872	1	19,6362	0.99	19,2462	0.99
Langerman	5	100	34,005.4	0.60	32,630.6	0.09	32,254.3	0.04
Michalewicz	5	100	28,219.8	0.96	31,005.8	0.98	30,828.1	0.73

Clustering the initial population significantly decreased success probabilities not only for Ackley's function, but also for the Rastrigin, Schwefel and Langerman functions, although in each case the average number of function evaluations was not seriously affected. This result reinforces the idea that when the objective function is multi-modal, it is important to disperse the initial population widely enough to contain the optimum. Results also suggest that the penalty for expanding bounds is a small increase in the average number of function evaluations but the reward is often a significantly enhanced probability of success.

DE is based on evolution with vector differences, so it is not surprising that the way in which differences are chosen can have an impact on the optimization process. The following section examines what happens when the base and difference vectors are chosen both with and without restrictions.

2.4 Base Vector Selection

There are four vector indices in classic DE's generating equation (e.g., Eq. 2.8). The target index, *i*, specifies the vector with which the mutant is recombined and against which the resulting trial vector competes. The remaining three indices, *r*0, *r*1 and *r*2, determine which vectors combine to create the mutant. Typically, both the base index, *r*0, and the difference vector indices, *r*1 and *r*2, are chosen anew for each trial vector from the range [0, *Np* − 1].

When indices are randomly selected, the possibility exists that some vectors may be chosen repeatedly while others may be omitted altogether. Both omitted and duplicated indices affect DE's performance. Duplicating an index can reduce DE's novel search strategy to a conventional one, while omitting an index may deprive a vector of the opportunity to serve as a base vector. After presenting several alternative schemes for selecting base vectors, this section explores the effects of degenerate vector combinations.

2.4.1 Choosing the Base Vector Index, *r*0

Random Without Restrictions

The base index, *r*0, specifies the vector to which the scaled differential is added. The classic version of DE employs a uniform distribution to randomly select *r*0 anew for each trial vector. To ensure that the index is always less than *Np*, $\text{rand}_i(0,1)$ must return a value that is strictly less than 1.

```
r0=floor(rand_i(0,1)*Np);
```

Fig. 2.18. Base vector selection without restrictions

While base index selection without restrictions (Fig. 2.18) treats all vectors equally in a statistical sense, it may pick some vectors more than once per generation, causing others to be omitted. Stochastic universal sampling provides a more representative population sample.

Stochastic Universal Sampling

Randomly selecting the base vector without restrictions is known in EA parlance as *roulette wheel selection*. Roulette wheel selection chooses *Np*

vectors by conducting *Np* separate random trials, much like *Np* passes at a roulette wheel whose slots are proportional in size to the selection probability of the vector they represent. In many GAs, selection probabilities are biased toward better solutions, meaning that better vectors are assigned proportionally wider slots, but in classic DE, each vector has the same chance of being chosen as a base vector, so all slots are of equal size, just like a real roulette wheel.

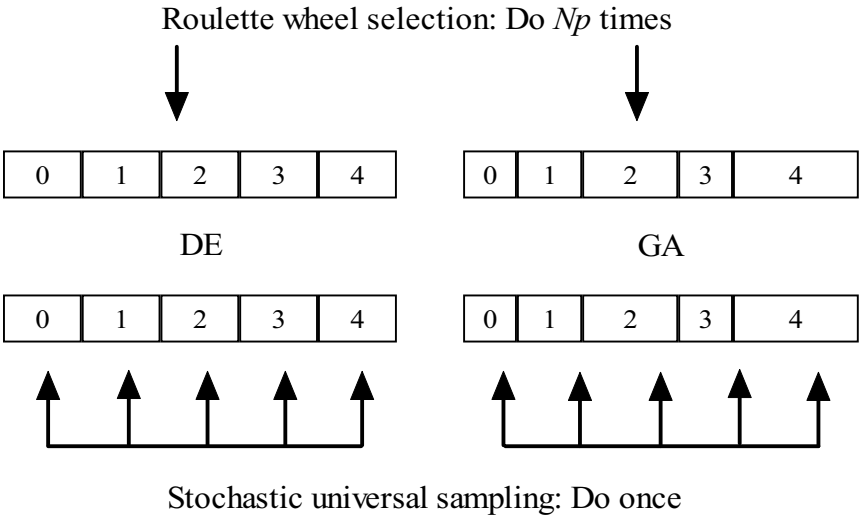

Fig. 2.19. Stochastic universal sampling and roulette wheel selection compared. The fraction of the space allotted to a vector in DE is constant, but in the GA it depends on the vector's objective function value.

Because samples drawn by roulette wheel selection suffer from a large variance, the preferred method for sampling a distribution is *stochastic universal sampling* because it guarantees a minimum spread in the sample (Baker 1987; Eiben and Smith 2003). The relation of stochastic universal sampling to roulette wheel selection is best illustrated if the ball used in real roulette is replaced with a stationary pointer. Once the roulette wheel stops, the vector corresponding to the slot pointed to is selected. Instead of spinning a roulette wheel *Np* times to select *Np* vectors with a single pointer, stochastic universal sampling uses *Np* equally spaced pointers and spins the roulette wheel just once. In the GA, slot sizes are based on a vector objective function value, with better vectors being assigned more space. In DE, each candidate has the same probability of being accepted, so slots are of equal size. Consequently, each of the *Np* pointers selects one

and only one vector regardless of how the roulette wheel is spun (Fig. 2.19)

The following vector selection methods adhere to stochastic universal sampling as it applies to DE since all vectors serve as base vectors once and only once per generation. Both methods described below also establish the one-to-one correspondence needed to pair each target vector with a unique base vector.

2.4.2 One-to-One Base Vector Selection

Permutation Selection

To ensure that each vector serves as a base vector just once per generation, permutation selection draws consecutive base vector indices from an array containing a random permutation of the sequence [0, 1,..., $Np - 1$]. In this scheme, the (target) vector with index i is crossed with is the base vector whose index is the i^{th} element of the permutation. The permutation array can be initialized with consecutive integers and $r0$ can be computed with a single call to a uniform random number generator and one swap of array elements. Another way to permute base vectors assigns to i the vector whose index is the product, modulo Np, of i and an integer that is relatively prime to Np. Details of both methods can be found in Sect. 5.2.

Random Offset Selection

The random offset method is another way to stochastically assign each target vector a unique base vector. Simpler than the permutation method, the random offset method computes $r0$ as the sum, modulo Np, of the target index and a randomly generated offset, r_g. The modulo operator, %, in Fig. 2.20 divides the operand, $(i + r_g)$, by Np and returns the integral remainder.

```
r0=(i+rg)%Np;
```

Fig. 2.20. The base vector is the sum, modulo Np, of the target index, i, and the randomly generated offset, r_g (see Fig. 2.21).

```
rg=floor(randg(0,1)*Np);
```

Fig. 2.21. The random offset, r_g, is chosen anew *at the start of each generation.*

Each of the *Np* possible values for r_g defines a one-to-one mapping between target and base vectors. These *Np* rotational mappings are a subset of the set of *Np*! permutations. The symbol, "!" is the factorial operator. The value of *n*! is just the product of all of the positive integers less than or equal to *n*. Figure 2.22 gives examples for each of the aforementioned base vector assignment methods. The target index is the population's running index, *i*, so each method automatically ensures that each vector serves as a target vector once per generation. Only the last two methods, however, also ensure that each vector serves as a base vector once per generation. permute[*i*] refers to the i^{th} element of an array containing a randomly generated permutation of the sequence [0, 1,…, *Np* − 1] (*Np* = 7 in Fig. 2.22).

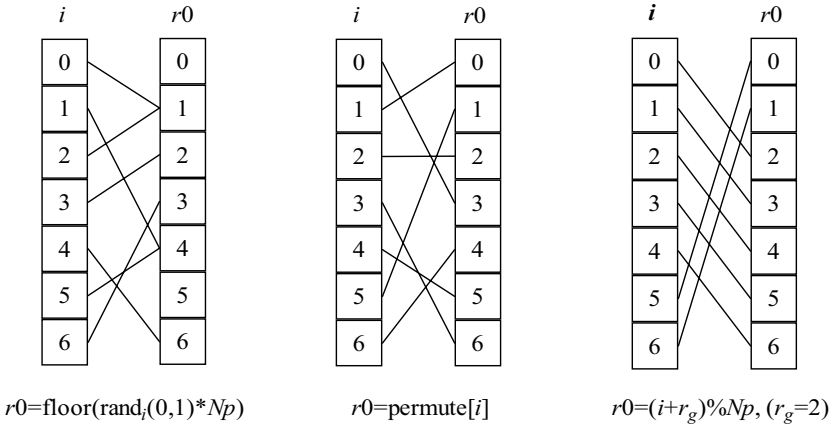

Fig. 2.22. Three ways to stochastically pair base and target vectors

2.4.3 A Comparison of Random Base Index Selection Methods

Using the ten-dimensional sphere as a test function, Table 2.4 compares the performance of the three stochastic base vector selection methods. (See the Appendix for test function details.) As Table 2.4 shows, all vector selection methods respond similarly when *Np* is increased. Before convergence becomes regular, increasing *Np* not only improves the probability of success, but also *decreases* the number of function evaluations needed to reach the optimum. Once convergence becomes regular, however, additional increases in *Np* only marginally improve the probability of convergence while the number of function evaluations begins to climb. As a result, each method exhibits an optimal population size for which the number of function evaluations is a minimum. In the case of the ten-

dimensional sphere, all three stochastic selection methods perform best when $Np = 9$, ($F = Cr = 0.9$ and $i \neq r0 \neq r1 \neq r2$), with each converging reliably in about 6000 function evaluations. Some performance disparities arise, however, once degenerate vector combinations are allowed.

Table 2.4. When best efforts are compared, all the three stochastic selection methods perform similarly. Results are 1000-trial averages of the number of function evaluations needed to reach the optimum to within a pre-specified limit and within the maximum allowed number of generations (see the Appendix for the function value to reach). P is the fraction of trials that were successful. For these results, $F = Cr = 0.9$ and $i \neq r0 \neq r1 \neq r2$.

Np	$r0 = \text{floor}(\text{rand}_i(0,1) \cdot Np)$		$r0 = \text{permute}[i]$		$r0 = (i + r_g)\%Np$	
	Evals.	P	Evals.	P	Evals.	P
5	36,616.0	0.001	17,929.0	0.004	–	0
6	14,215.0	0.074	16,804.2	0.309	17,627.9	0.583
7	12,917.2	0.889	10,017.1	0.961	9047.00	0.977
8	7097.05	0.982	6582.3	0.979	7086.11	0.995
9	6006.70	0.994	5954.05	0.995	5927.24	0.998
10	6039.08	0.996	5969.34	1.0	6669.14	1.0
11	6433.55	0.998	6431.55	0.999	6843.09	1.0
12	71,10.87	0.999	7195.95	1.0	8213.57	1.0
13	79,86.33	0.999	8031.48	1.0	8856.00	1.0
14	90,15.09	1.0	9040.13	1.0	10,509.7	1.0
15	10,095.4	1.0	10,214.1	1.0	11,557.2	1.0

2.4.4 Degenerate Vector Combinations

If indices are chosen without restrictions, there is no guarantee that i, $r0$, $r1$ and $r2$ will be distinct. When these indices are not mutually exclusive, DE's novel trial vector-generating strategy reduces to uniform crossover only, duplication of the base vector, an alternative form of recombination, or mutation only. These possibilities are explored below, first by looking at the three degenerate combinations of indices that comprise the mutant vector, $r0$, $r1$ and $r2$, and then by considering the three interactions of the target index, i, with the mutant indices.

Degenerate Combinations of Mutant Indices: r0, r1, r2

***r*1 = *r*2: No Mutation.** If $r1 = r2$, then the differential formed by the corresponding vectors will be zero and the base vector, $\mathbf{x}_{r0,g}$, will not be mutated:

$$r1 = r2\,(= r0): \quad \mathbf{v}_{i,g} = \mathbf{x}_{r0,g}. \tag{2.13}$$

When indices are chosen without restrictions, $r1$ will equal $r2$ on average once per generation, i.e., with probability $1/Np$. The probability that all three indices will be equal is $(1/Np)^2$, but either way, the result is the same: a randomly chosen base vector that has *not* undergone mutation is recombined with the target vector by means of conventional uniform crossover:

$$\mathbf{u}_{i,g} = u_{j,i,g} = \begin{cases} x_{j,r0,g} & \text{if } \left(\text{rand}_j(0,1) \le Cr \vee j = j_{\text{rand}}\right); \\ x_{j,i,g} & \text{otherwise.} \end{cases} \tag{2.14}$$

Requiring the base vector to contribute a parameter when $j = j_{\text{rand}}$ ensures that the trial vector will not simply reproduce the vector with which it is compared, i.e., the target vector, $\mathbf{x}_{i,g}$. If, however, Cr is greater than 0, the possibility exists that the trial vector will *duplicate* the base vector. When $Cr = 1$, and $r1 = r2$, duplication is a certainty:

$$r1 = r2\,(= r0) \wedge Cr = 1: \quad \mathbf{u}_{i,g} = \mathbf{v}_{i,g} = \mathbf{x}_{r0,g}. \tag{2.15}$$

More generally, the probability that the base vector will be duplicated is the product of the probability that $r1 = r2$ and the probability that all parameters are inherited from the mutant, $\mathbf{v}_{i,g}$. Since Cr mediates a random process having just two possible outcomes (mutant or target), the number of parameters inherited from the mutant is governed by a binomial distribution. Thus, the probability of inheriting x mutant parameters in n tries is

$$p(X = x) = \frac{n!}{x!(n-x)!} Cr^x (1 - Cr)^{n-x}, \qquad n! = \prod_{k=1}^{n} k. \tag{2.16}$$

Since one parameter is certain to be taken from the mutant, $n = D - 1$. Thus, the probability, given Cr, that all $D - 1$ of the remaining parameters will also be inherited from the mutant ($x = D - 1$) is

$$p(X = D-1) = \frac{(D-1)!}{(D-1)!\,0!} Cr^{D-1} (1 - Cr)^0 = Cr^{D-1}, \qquad 0! \equiv 1. \tag{2.17}$$

When difference indices are chosen without restrictions, the probability that the base vector will not be mutated is $1/Np$, making Cr^{D-1}/Np the probability that a base vector will be duplicated.

***r*1 = *r*0 or *r*2 = *r*0: Arithmetic Recombination.** Another special case occurs when either of the difference indices, $r1$ or $r2$, equals the base index, $r0$. When indices are chosen without restrictions, each coincidence occurs on average once per generation. Equation 2.18 elaborates the two possibilities that result when DE's three-vector mutation formula (Eq. 2.5) reduces to a linear relation between the base vector and a single difference vector:

$$r1 = r0: \quad \mathbf{v}_{i,g} = \mathbf{x}_{r0,g} + F \cdot \left(\mathbf{x}_{r0,g} - \mathbf{x}_{r2,g}\right), \tag{2.18}$$

$$r2 = r0: \quad \mathbf{v}_{i,g} = \mathbf{x}_{r0,g} + F \cdot \left(\mathbf{x}_{r1,g} - \mathbf{x}_{r0,g}\right).$$

Each two-vector linear combination defines a line that connects the base vector to one of the two difference vectors (Fig. 2.23). F plays the role of a coefficient of combination that determines which point along the line is targeted. In the parlance of evolutionary computation, this "line search" is usually called either *continuous* or *arithmetic recombination*. This book adopts the term "arithmetic recombination". Section 2.6 explores this process more thoroughly.

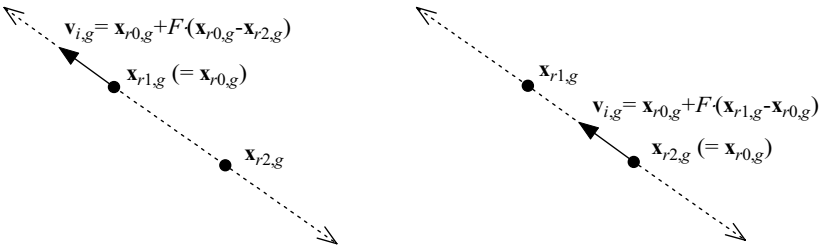

Fig. 2.23. Mutation degenerates into two-vector arithmetic recombination when either $r1 = r0$ (left) or $r2 = r0$ (right).

Degenerate Combinations Involving the Target Index, i

***r*0 = *i*: Mutation Only.** If the base index, $r0$, is not different from the target index, i, then crossover reduces to mutation of the target vector. In this scenario, Cr plays the role of a mutation probability:

$$u_{j,i,g} = \begin{cases} x_{j,i,g} + F \cdot \left(x_{j,r1,g} - x_{j,r2,g} \right) & \text{if } \left(\text{rand}_j(0,1) \le Cr \vee j = j_{\text{rand}} \right), \\ x_{j,i,g} & \text{otherwise.} \end{cases} \quad (2.19)$$

When base vector indices are randomly selected without restrictions, these degenerate vector combinations occur with probability $1/Np$.

***i* = *r*1 or *i* = *r*2.** Each of the coincidental events, $i = r1$ and $i = r2$, occurs with probability $1/Np$ when indices are chosen without restrictions. Neither coincidence reduces DE's generating process to a conventional one; mutants are still three-vector combinations and crossover recombines distinct base and target vectors (assuming $r0 \neq i$).

Table 2.5 summarizes the possible degenerate vector combinations that can occur when difference indices are chosen without restrictions, i.e., index = floor(rand$_i$(0,1)·Np).

Table 2.5. First-order degenerate combinations

Event	Degenerate process	Prob.	Result
$r1 = r2$	Uniform crossover	$1/Np$	$\mathbf{v}_{i,g} = \mathbf{x}_{r0,g}$
	Duplication of base vector	Cr^{D-1}/Np	$\mathbf{u}_{i,g} = \mathbf{x}_{r0,g}$
$r0 = r1$	Intermediate recombination	$1/Np$	$\mathbf{v}_{i,g} = \mathbf{x}_{r0,g} + F \cdot (\mathbf{x}_{r0,g} - \mathbf{x}_{r2,g})$
$r0 = r2$	Intermediate recombination	$1/Np$	$\mathbf{v}_{i,g} = \mathbf{x}_{r0,g} + F \cdot (\mathbf{x}_{ri,g} - \mathbf{x}_{r0,g})$
$i = r0$	Differential mutation	$1/Np$	$\mathbf{v}_{i,g} = \mathbf{x}_{i,g} + F \cdot (\mathbf{x}_{r1,g} - \mathbf{x}_{r2,g})$
$i = r1$	None	$1/Np$	$\mathbf{v}_{i,g} = \mathbf{x}_{r0,g} + F \cdot (\mathbf{x}_{i,g} - \mathbf{x}_{r2,g})$
$i = r2$	None	$1/Np$	$\mathbf{v}_{i,g} = \mathbf{x}_{r0,g} + F \cdot (\mathbf{x}_{r1,g} - \mathbf{x}_{i,g})$

Higher Order Degenerate Combinations

The above index pairings are first-order degenerate combinations in which only two of four indices are coincident. If indices are chosen without restrictions, the same index may be chosen more than twice. In practice, the effects of higher order degenerate combination are small because their probability is inversely proportional to powers of $Np \ge 2$.

2.4.5 Implementing Mutually Exclusive Indices

Enforcing i ≠ r0

If base indices are chosen randomly, as they are in classic DE, then $r0 = i$ can be prevented by using a "do–while" loop to reselect $r0$ until it no longer equals the target vector index (Fig. 2.24).

```
do
{
   r0=floor(rand_i(0,1)*Np);
}while(r0==i);
```

Fig. 2.24. To ensure that base and target vectors are different, $r0$ should be reselected.

Similarly, if base vectors are the elements of a permutation, then $r0$ can be redrawn from the remaining list of unused indices, except when i is the last element of the permutation. In the random offset method, choosing r_g from the more restricted range [1, $Np - 1$], ensures that $r0 = i$ does not occur.

Mutually Exclusive Indices: $i \neq r0 \neq r1 \neq r2$

Once the base vector has been determined, difference indices can be chosen. Perhaps the simplest way to implement mutually exclusive indices is to use a pair of "do–while" loops (Fig. 2.25) to reselect any difference index that happens to equal the target, base or a previously chosen difference index.

```
do
{
   r1=floor(rand_i(0,1)*Np);
}while(r1==i || r1==r0); // "||" is "or"
do
{
   r2=floor(rand_i(0,1)*Np);
}while(r2==i || r2==r0 || r2==r1);
```

Fig. 2.25. Given $r0 \neq i$, distinct indices should be selected with a pair of do–while loops.

Distinct difference indices can be taken from arrays of random permutations of the sequence [0, 1,…, $Np - 1$]. Methods for generating random permutations are presented in Sect. 5.2 and as an option in the Matlab code on this book's companion CD-ROM.

2.4.6 Gauging the Effects of Degenerate Combinations: The Sphere

Table 2.6 calls upon the ten-dimensional sphere to reveal how the presence of degenerate combinations affects both the speed and probability with which each of three stochastic base index selection methods converges. Because it is simple, the sphere provides a good way to interpret the effect of degenerate vector combinations. Performance is measured at the value of Np that minimizes the average number of function evaluations. The first row of results, labeled "All", shows the combined effect of all degenerate combinations (any $r0$, $r1$, $r2$). For the final row of results, labeled "None", indices are mutually exclusive ($i \neq r0 \neq r1 \neq r2$) and degenerate combinations are forbidden. The middle rows record what happens when *only* the designated index coincidence is permitted.

Table 2.6. DE's performance is influenced by the way in which trial vector indices are chosen. Here, the effects of degenerate vector combinations on the three base index selection schemes are compared. Results are 1000-trial averages, with $F = Cr = 0.9$. The value of Np is that which yields the answer in the fewest number of function evaluations, while P is the corresponding probability of success. Data for the last row "None", has been copied from Table 2.4.

Allowed event	$r0 = \text{floor}(\text{rand}_i(0,1)\cdot Np)$			$r0 = \text{permute}[i]$			$r0 = (i + r_g)\%Np$		
	Np	Evals.	P	Np	Evals.	P	Np	Evals.	P
All	14	6479.79	0.992	14	6359.32	0.996	13	6549.55	1.0
$r1 = r2$	13	6585.71	0.797	14	6522.19	0.881	13	6568.55	0.993
$r0 = r1$	9	6388.09	0.967	9	6341.93	0.976	9	6371.36	0.994
$r0 = r1$	13	5832.56	1.0	13	5743.42	1.0	13	5769.72	1.0
$i = r0$	9	6067.18	0.991	9	5940.59	0.993	10	10,829.5	1.0
$i = r1$	9	6009.60	0.992	9	6007.01	0.998	9	6204.13	0.999
$i = r2$	9	5955.17	0.992	9	5847.14	0.998	9	5729.59	1.0
None	9	6006.70	0.994	9	5954.05	0.995	9	5927.24	0.998

All: Any r0, r1 and r2

The first row of data summarizes the combined influence of all degenerate combinations, including higher order degenerate combinations. Although the large optimal population size helps to keep convergence probability competitive, it also slows convergence speed.

r1 = r2

Except for the anomalous behavior of the random offset method when $r0 = i$, all three base index selection methods exhibit their worst performance when equal difference indices are allowed ($r1 = r2$). At $Cr = 0.9$, a significant fraction of these events (about 39%) duplicate base vectors. Re-evaluating duplicated vectors wastes time and accepting them reduces the population's effective size. Indeed, Table 2.6 shows that when $r1 = r2$ is allowed, all three base index selection methods require relatively large populations. In this case, increasing Np to compensate for duplicated entries slows convergence without making it reliable.

r0 = r1

Because difference vector $\mathbf{x}_{r2,g}$ is preceded by a minus sign, $r0 = r1$ places the recombinant farther away from $\mathbf{x}_{r2,g}$ than was $\mathbf{x}_{r1,g}$ whenever $F > 0$ (refer back to Fig. 2.23). For the sphere, accepting this recombinant slows convergence and compromises reliability. This form of recombination also tends to slow convergence on multi-modal functions, but its effect on the probability of convergence will not always be detrimental.

r0 = r2

By contrast, all three base index selection methods performed best when they allowed $r0 = r2$ to transform differential mutation into arithmetic recombination. This is because when $0 < F < 2$, $r2 = r0$ produces a recombinant that lies closer to $\mathbf{x}_{r1,g}$ than was $\mathbf{x}_{r2,g}$. This contractile mapping improves optimization speed even though Np must be increased to compensate for the additional convergence "pressure". Allowing this index combination typically speeds optimizations of multi-modal functions as well, but unlike the case of the sphere, it is less common that convergence probability will also improve.

i = r0

A careful examination of Table 2.6 shows that the random offset method (last column) exhibits the best probability of convergence under all circumstances. In addition, its speed of convergence is competitive except for the case $i = r0$, when the number of function evaluations balloons to nearly twice that of the other two methods. It may seem curious that permitting the combination $r0 = i$ affects the performance of the random offset method so much more than it does the random selection method, even though $r0 = i$ occurs on average once per generation in both cases. The

performance disparity arises because when the random offset equals zero ($r_g = 0$), an entire generation of target–base pairings is turned into degenerate combinations, whereas unrestricted random selection spreads them uniformly over the generations. When allowed, the same "identity mapping" of target and base vectors also occurs in the permutation method, but its effect is negligible since it occurs on average only once every $Np!$ generations.

i = r1 and i = r2

The influence of $i = r1$ and $i = r2$ is more difficult to analyze than that of the corresponding pair of events $r0 = r1$ and $r0 = r2$, but it mirrors their behavior, with one event speeding convergence ($i = r2$) and the other retarding it ($i = r1$). Although its convergence speed distinguishes $i = r1$ from $i = r2$, both events have little effect on either convergence probability or optimal population size when compared to the case of mutually exclusive indices (i.e., "None").

None

Excluding all degenerate target, base and difference vector combinations, i.e., $i \neq r0 \neq r1 \neq r2$, enables DE to achieve both good convergence speed and probability with a relatively small population. Imposing restrictions eliminates the function-dependent effects of degenerate search strategies and ensures that both crossover and differential mutation play a role in the creation of each trial vector.

The effect that degenerate vector combinations have on DE's performance depends in some degree on the objective function. For the hypersphere, however, only $i = r0$ dramatically affected DE's performance. In practice, even these first-order degenerate combinations play only a limited role in the optimization process simply because they become increasingly infrequent as the population grows.

2.4.7 Biased Base Vector Selection Schemes

In GAs, better vectors are more likely to be chosen for recombination (Holland 1973). Similarly, some versions of DE select the base vector based on its objective function value. For example, the algorithm DE/best/1/bin (Storn 1996) always selects the best-so-far vector (*best*) as the base vector, adds a single (*1*) scaled vector difference to it, then creates a trial vector by uniformly crossing (*bin*) the resulting mutant with the tar-

get vector. In this algorithm, the base vector always has the lowest objective function value in the current population

$$r0 = \text{best, if } \forall i \in (0,1,...,Np-1),\ f\left(\mathbf{x}_{\text{best},g}\right) \le f\left(\mathbf{x}_{i,g}\right). \tag{2.20}$$

When compared to random base vector selection *at the same Np*, best-so-far base vector selection usually speeds convergence, reduces the odds of stagnation and lowers the probability of success. Chapter 3 examines this trade-off between speed and reliability when the performance of DE/rand/1/bin and DE/best/1/bin are compared.

Two alternative base vector selection schemes have been proposed that bias solutions toward better vectors without creating the intense selection pressure that the "best" method applies. In Price (1997), a base vector's objective function value must be less than or equal to that of the target vector, $\mathbf{x}_{i,g}$:

$$r0 = \text{better, if } f\left(\mathbf{x}_{\text{better},g}\right) \le f\left(\mathbf{x}_{i,g}\right). \tag{2.21}$$

The other method, DE/target-to-best/1/bin (called "rand-to-best" in (Storn 1996)), uses arithmetic recombination (see Sect. 2.6.3) to generate a base vector that lies on a line between the target vector and the best-so-far vector:

$$\mathbf{x}_{r0,g} = \mathbf{x}_{i,g} + k \cdot \left(\mathbf{x}_{\text{best},g} - \mathbf{x}_{i,g}\right), \quad k \in [0,1] = \text{constant}. \tag{2.22}$$

The constant, k, in Eq. 2.22 controls the bias toward the best-so-far solution.

Compensating for Lost Diversity

Compared to random base vector selection, setting $r0$ = best lowers the diversity of the pool of potential trial vectors. Increasing the population size is both a simple and effective way to enhance the diversity of the pool of potential trial vectors, but several other schemes have also been proposed. One idea was to expand the set of vector differences by adding *two difference vectors* together (Price 1996; Storn 1996). Because they are larger than their single difference counterparts, differentials composed of two differences typically require a smaller F to match the convergence rate that one-difference differentials produce. Except for a few early successes on relatively simple functions, this method has not shown much promise, perhaps because adding difference vectors destroys the correlation that the objective function's topography imparts to the one-difference vector differentials (see contour matching in Sect. 2.17).

Making F a random variable is another way to enhance the pool of potential trial vectors. This technique, which is covered extensively in Sect. 2.5.2, has proven useful in cases where stagnation threatens, or when convergence is very slow. In particular, R. Storn has found randomizing the scale factor, F, to be crucial when designing digital filters (Sect. 7.8).

2.5 Differential Mutation

Most dictionaries define mutation as an alteration or change. In the context of genetics and EAs, however, mutation is also seen as change with a random element. Thus, real-valued EAs typically simulate the effects of mutation with additive increments that are randomly generated by a predefined *probability distribution function*, or PDF. DE, however, uses a uniform PDF not to generate increments, but to randomly sample vector differences:

$$\Delta \mathbf{x}_{r1,r2} = \left(\mathbf{x}_{r1} - \mathbf{x}_{r2} \right). \tag{2.23}$$

In a population of Np distinct vectors, there will be $Np \cdot (Np - 1)$ non-zero vector differences and Np null differences having zero magnitude giving a total of Np^2 vector differences. Figure 2.26 pictures an arbitrary population of 5 vectors and the sheaf of 20 non-null difference vectors that they generate.

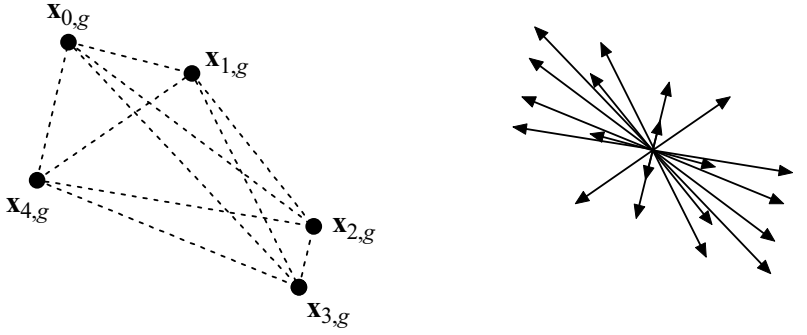

Fig. 2.26. The figure on the right displays the sheaf of 20 vector differences generated by the population of 5 vectors shown on the left. Here, differentials have been scaled by half (F = 0.5), and transported to a common origin. Note that the distribution is symmetric about zero.

The distribution of difference vectors will depend on the distribution of vectors and this will be different for each objective function. Each distribu-

tion, however, will be symmetric about zero because every pair of vectors gives rise to two opposite but equal difference vectors, since reversing the order of the vectors in the differential reverses the sign of the differential:

$$\Delta\mathbf{x}_{r1,r2,g} = \left(\mathbf{x}_{r1,g} - \mathbf{x}_{r2,g}\right) = -\left(\mathbf{x}_{r2,g} - \mathbf{x}_{r1,g}\right) = -\Delta\mathbf{x}_{r2,r1,g}\,. \tag{2.24}$$

Since each difference vector can be paired with a differential of equal value but opposite sign, and since all vector differences are equally probable, both the sum and average over all Np^2 difference vectors are zero. Equation 2.25 sums the Np^2 vector differences (including the Np cases when $i = k$ and $\Delta\mathbf{x} = 0$) and normalizes the result. The brackets, $\langle\ \rangle$, indicate that $\Delta\mathbf{x}$ is an (ensemble) average taken over all population members, not an expectation or a time average:

$$\left\langle \Delta\mathbf{x} \right\rangle_g = \frac{1}{Np^2}\sum_{i,k=0}^{Np\text{-}1}\left(\mathbf{x}_{i,g} - \mathbf{x}_{k,g}\right) = 0\,. \tag{2.25}$$

2.5.1 The Mutation Scale Factor: *F*

Limits on F

Upper. The stated range for F is (0,1), although 1.0 is an empirically derived upper limit in the sense that no function that has been successfully optimized has required $F > 1$. This is not to say that solutions are not possible when $F > 1$, but only that they tend to be both more time consuming and less reliable than if $F < 1$. When $F = 1$ exactly, otherwise distinct vector combinations become indistinguishable:

$$\mathbf{x}_{r0,g} + \mathbf{x}_{r1,g} - \mathbf{x}_{r2,g} = \begin{cases} \mathbf{x}_{r0,g} + F\cdot(\mathbf{x}_{r1,g} - \mathbf{x}_{r2,g}) \\ \mathbf{x}_{r1,g} + F\cdot(\mathbf{x}_{r0,g} - \mathbf{x}_{r2,g}) \end{cases} \quad \text{when } F = 1. \tag{2.26}$$

This discontinuity at $F = 1$ reduces the number of mutants by half and can result in erratic convergence unless $Cr < 1$, since $Cr = 1$ further restricts the pool of possible trial vectors by not crossing mutant and target parameters.

Lower. In general, selection tends to reduce the diversity of a population, whereas mutation increases it. To avoid premature convergence, it is crucial that F be of sufficient magnitude to counteract this selection pressure. Zaharie (2002) recently demonstrated the existence of what is effectively a lower limit for F, finding that if F is too small, the population can con-

verge even if selection pressure is absent. In her study, Zaharie measured population diversity as the variance of its parameter values. Because all variables are independent in the absence of selection pressure, population diversity can be measured by tracking the variance of a single parameter of the population. In Eq. 2.27, the subscript, "*x*", in $\mathrm{P}_{x,g}$ is set in italics to emphasize that the variance and mean are computed using one *parameter* from each vector in the population (the particular parameter is not specified):

$$\mathrm{Var}(\mathrm{P}_{x,\mathrm{g}}) = \frac{1}{Np}\sum_{i=0}^{Np-1}\left(x_{i,g} - \langle x\rangle_g\right)^2; \quad \langle x\rangle_g = \frac{1}{Np}\sum_{i=0}^{Np-1} x_{i,g}\,. \tag{2.27}$$

Using a methodology pioneered by H.-G. Beyer (1999), Zaharie computed the expected variance of DE's mutant and trial populations given the variance of the population. The goal was to determine which combinations of DE control parameters were likely to result in premature convergence due solely to the inability of the algorithm to generate a trial population as diverse as the population. To simplify her analysis, Zaharie dropped DE's usual demand that base and target vectors be different, although the requirement that base and difference vectors be distinct was retained. By dropping the demand that at least one trial parameter be inherited from the mutant, Zaharie also assumed that *Cr* is a true crossover probability, p_{Cr}. In order to compute the expected population variance, Zaharie further modified the standard DE algorithm by multiplying *F* by a Gaussian random variable, ξ_j, that is chosen anew *for each parameter*

$$v_{j,i,g} = x_{j,r0,g} + \widetilde{F}_j \cdot \left(x_{j,r1,g} - x_{j,r2,g}\right); \qquad \widetilde{F}_j = F \cdot \xi_j, \quad \xi_j \approx \mathrm{N}(0,1)\,. \tag{2.28}$$

With these caveats, Zaharie determined that the expected variance of the mutant population is related to the variance of the population by the formula:

$$E\left(\mathrm{Var}\left(\mathrm{P}_{v,g}\right)\right) = \left(2F^2 + \frac{Np-1}{Np}\right)\mathrm{Var}\left(\mathrm{P}_{x,g}\right). \tag{2.29}$$

If this mutant population is then crossed with the original population, the expected trial population variance becomes:

$$E\left(\mathrm{Var}\left(\mathrm{P}_{u,g}\right)\right) = \left(2F^2 p_{Cr} - \frac{2p_{Cr}}{Np} + \frac{p_{Cr}^{\ 2}}{Np} + 1\right)\mathrm{Var}\left(\mathrm{P}_{x,g}\right). \tag{2.30}$$

Consequently, DE control parameter combinations that satisfy the equation:

$$2F^2 - \frac{2}{Np} + \frac{p_{Cr}}{Np} = 0 \tag{2.31}$$

can be considered to be critical since they result in a population whose variance remains constant except for random fluctuations. When selection is "turned off", Eq. 2.31 predicts that F will display a critical value, F_{crit}, such that the population variance decreases when $F < F_{\text{crit}}$ and increases when $F > F_{\text{crit}}$. Solving Eq. 2.31 for F gives F_{crit} as

$$F_{\text{crit}} = \sqrt{\frac{\left(1 - p_{Cr}/2\right)}{Np}}\,. \tag{2.32}$$

Thus, F_{crit} establishes a lower limit for F in the sense that smaller values will induce convergence even on a level objective function landscape. Figure 2.27 confirms the prediction by Zaharie that $F = 0.1341$ is a critical value when $Np = 50$ and $p_{Cr} = 0.2$.

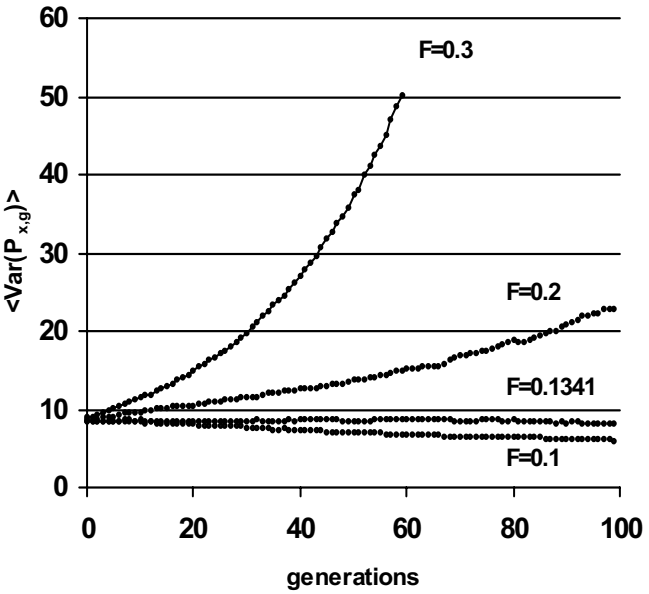

Fig. 2.27. The evolution of the variance of a single parameter is displayed for four different values of F. Note that $F \sim 0.134$ is critical in the sense that the variance is nearly constant. Results are for evolution on a flat surface, i.e., all trial vectors are accepted (no selection pressure). These results are 100-trial averages and were generated using Zaharie's modified version of DE, with $Np = 50$ and $p_{Cr} = 0.2$.

Objective function landscapes are seldom flat. In practice, F must be larger than F_{crit} to counteract the additional reduction in variance that selection induces. For example, Zaharie empirically examined three test functions using $Np = 50$, $p_{Cr} = 0.2$ and found that $F \sim 0.3$ was the smallest reliable scale factor and that $F_{\text{crit}} = 0.1341$ was too small to forestall premature convergence.

Figure 2.28 illustrates the effect of this additional selection pressure produced by the 30-dimensional Rastrigin function on the population's variance over time at several different values for F.

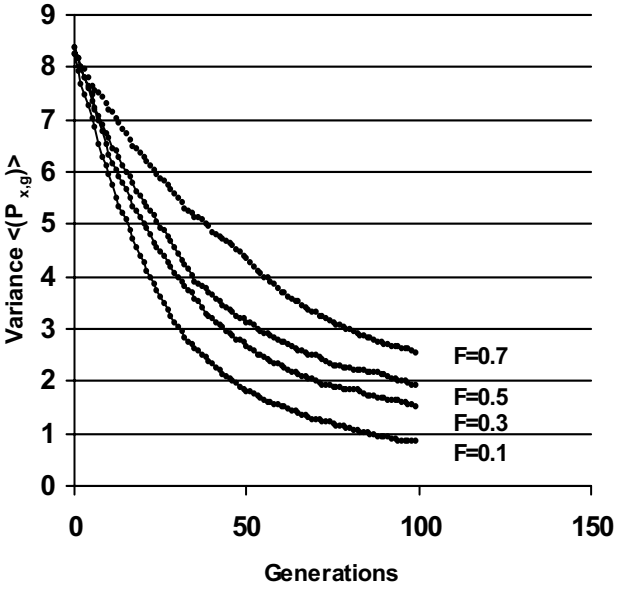

Fig. 2.28. Even though F is above the critical value, the population variance still decreases over time due to the selection pressure exerted by the objective function, in this case the thirty-dimensional Rastrigin function. Results are 100-trial averages obtained with Zaharie's version of DE, with $Np = 50$ and $p_{Cr} = 0.2$.

A DE control parameter study by Gamperle et al. (2002) explored DE's performance on two of the same test functions that Zaharie used and concluded that $F < 0.4$ was not useful. In Ali and Törn (2000), C–Si clusters were optimized with F never falling below $F = 0.4$. On the other hand, Chakraborti et al. (2001; Sect. 7.1) had success minimizing the binding energy of Si–H clusters using values for F ranging from 0.0001 to 0.4, with $F = 0.2$ often proving effective. Such low values for F, however, appear to be atypical. The lower limits suggested by Zaharie and Gamperle et al. more

accurately reflect the norm. Zaharie concluded that for the test functions examined, modifying vector differences with a Gaussian distribution did not significantly alter DE's performance compared to when F is held constant. The next section tests this claim and examines several other methods for transforming F into a random variable.

2.5.2 Randomizing the Scale Factor

When compared to the ES, DE shifts the responsibility for adapting step sizes from the mutation distribution's pre-factors to the distribution itself. More specifically, the ES adapts pre-factors (strategy parameters) and multiplies them by the output from a stationary, multi-dimensional PDF, whereas DE multiplies the constant pre-factor, F, by a sample vector difference from an adaptive distribution. Whereas the ES "strategy" parameters adapt to the *absolute* step size, F only affects the *relative* step size since the distribution of vector differences is itself adaptive. Thus, F can be kept constant during optimization without compromising DE's ability to generate steps of the required size. Indeed, keeping F constant has proven effective in the sense that no function that has been solved has required F to be a random variable. Nevertheless, randomizing F offers potential benefits.

Transforming F into a random variable effectively broadens the spectrum of vector differentials beyond the possibilities allowed for by combining vectors. Such an enhanced distribution of differentials might be useful if the population is small and/or symmetrically distributed, since without access to a mutation distribution of sufficient diversity, DE can *stagnate*. When stagnant, DE can no longer find improved solutions because no combination of vector and vector difference leads to a better solution. Instead of coalescing to a single solution, a stagnant population of vectors remains static while still distributed throughout the problem space. The case explored by Lampinen and Zelinka (2000) is hypothetical and subsequent attempts to induce stagnation in test functions with classic DE have been unsuccessful. Nevertheless, randomizing the scale factor is a way to increase the pool of potential trial vectors and minimize the risk of stagnation without increasing the population size.

Transforming F into a random variable also makes the analysis of DE dynamics tractable. By invoking the normal (Gaussian) distribution, Zaharie succeeded not only in predicting critical control parameter combinations, but also in constructing a limited convergence proof (Zaharie 2002). Zaharie based her proof on the general EA convergence criteria set forth by G. Rudolph (1996). Briefly, an evolutionary search algorithm can be

proven to converge to within $\varepsilon > 0$ of the global optimum in the long-time limit if its operators fulfill two (sufficient, but not necessary) conditions:

1. The transition probability, through mutation, between any two points in the problem space is strictly positive.

2. Selection is elitist, i.e., that the best-so-far solution is always retained.

DE selection is elitist because the population's current best vector can only be replaced by a better vector. By multiplying F by a normally distributed variable, Zaharie ensured that the unbounded, multi-normal distribution could access any point given enough time. The possibility does exist, however, that all members of a population may have the same value for one or more parameters, in which case no new possibilities for that parameter are generated. Zaharie considers this set to be of zero measure and that it has no impact on the proof that DE is convergent when mutation is augmented by a Gaussian random variable.

Converting F into a random variable, however, involves both selecting a PDF and deciding how often it should be sampled. Zaharie, for example, sampled a zero-mean, normally distributed random variable anew for each parameter, but this is not the only possibility. The next two subsections explore how both the sampling frequency and PDF affect the optimization process.

PDF Sampling Frequency: Dither and Jitter

In Zaharie's version of DE, F_j is a normally distributed random variable that is generated anew for each parameter. For convenience, the practice of generating a new value of F for every *parameter* is called *jitter* and it is signified by subscripting F with the parameter index, j. Alternatively, choosing F anew for each *vector*, or *dithering*, is indicated by subscripting F with the population's running index, i. Dithering scales the length of vector differentials because the same factor, F_i, is applied to all components of a difference vector (Fig. 2.29). As such, dithering does not dramatically depart from traditional DE in which each component of a differential is scaled by the same constant, F. Jitter, however, multiplies each component of the difference vector by a different scale factor, F_j, and this changes not only the scale of the differential, but also its orientation. The rotation that it introduces makes jitter a fundamentally different process than classic DE mutation with F = constant.

When $Cr = 0$, only one trial vector parameter is inherited from the mutated base vector, so it impossible to distinguish jitter from dither, since in

both cases only a single instance of F as a random variable occurs per trial vector. In order to compare how jitter and dither affect the optimization process, it is necessary to plot DE's performance *versus Cr*.

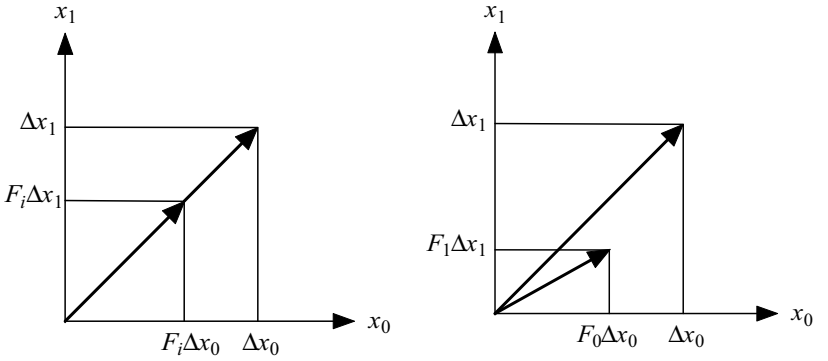

Fig. 2.29. Dithering (left) scales vector differentials, while jitter (right) both scales and rotates them.

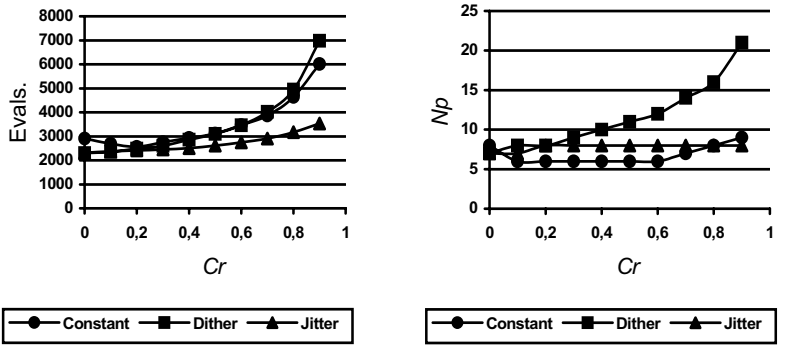

Fig. 2.30. The graph on the left illustrates how implementing jitter and dither with the N(0,1) PDF affects convergence speed compared to holding F constant. Plotted as a function of Cr is the minimum number of function evaluations required to optimize the ten-dimensional hyper-sphere. The graph on the right plots the corresponding optimal Np at which the minimum number of evaluations occurred. For example, at $Cr = 0.8$, the graph on the left shows that jitter took a little more than 3000 evaluations, while the graph on the right shows that the population used to produce this result was $Np = 8$ (at $Cr = 0.8$). Results are 1000-trial averages which, except for the indicated randomization method, were obtained with $F = 0.9$, $r0 = \text{rand}_i(0,1) \cdot Np$, except $r0 \neq r1 \neq r2 \neq i$ (i.e., classic DE).

Using the normal (Gaussian) PDF, N(0,1), to drive dither and jitter, Fig. 2.30 plots both the minimum number of function evaluations and the cor-

responding optimal population size at which the minimum occurred *versus* *Cr* for each of the three methods when applied to the ten-dimensional hyper-sphere objective function. An inspection of the graphs in Fig. 2.30 reveals that:

- As expected, both jitter and dither exhibit the same number of function evaluations and the same optimal population size ($Np = 7$) when $Cr = 0$.
- At $Cr = 0.2$ (Zaharie's choice), all three methods require about the same number of function evaluations, with both jitter and dither also having the same optimal population size ($Np = 8$).
- Over the range of *Cr*, jitter was the fastest technique and the optimal population size was virtually constant at $Np = 8$.
- In terms of the number of function evaluations, F = constant and dither perform similarly, but dither requires a larger population.

The data in Fig. 2.30 casts suspicion on Zaharie's contention that multiplying each component of a differential by a normally distributed variable does not affect DE's performance. Even for a function as simple as the hyper-sphere, classic DE with its constant F and Zaharie's method of jitter perform similarly only when $Cr = 0.2$ and this performance discrepancy grows as *Cr* increases.

Not shown in Fig. 2.30 is the fact that all trials conducted with both dither and jitter at their optimal *Np* were successful, but convergence was less than perfect when F was kept constant, as Table 2.7 shows. A slight increase in *Np*, however, would put the convergence probability on a par with that of dither and jitter, but then the average number of function evaluations would also increase.

Table 2.7. The fraction of trials that were successful when optimizing the ten-dimensional hyper-sphere using classic DE and F = constant = 0.9. By contrast, all trials with dither and jitter were successful at the specified optimal *Np*.

Cr	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
P	0.966	0.884	0.95	0.931	0.916	0.881	0.865	0.957	0.971	0.991	1

Also not shown in Fig. 2.30 are the data points associated with $Cr = 1$. These points are not plotted in Fig. 2.30 simply because the large values for dither and F = constant would overwhelm the data for $Cr \leq 0.9$. Instead, Table 2.8 reports both the average minimum number of function evaluations and the population size at which this minimum occurred for each of the three methods when *Cr* was set equal to 1. When compared to

trials using $Cr \leq 0.9$, all three methods required significantly larger populations to offset the loss of diversity that occurs when $Cr = 1$, exactly. In this case, the penalty for enlisting larger populations is slower convergence.

Table 2.8. When $Cr = 1$, both the optimal population size and the number of function evaluations balloon for both dither and F = constant and even jitter takes twice as long to converge as it does when $Cr = 0.9$. Results are 1000-trial averages for the ten-dimensional hyper-sphere using classic DE except for the indicated randomization method using a normal distribution: N(0,1).

Process	Evaluations	*Np*
F=constant	49,809.5	41
Dither	33,640.1	109
Jitter	6037.11	13

The hyper-ellipsoid (Eq. 2.33) poses a stiffer challenge to optimization algorithms because unlike the symmetrical hyper-sphere, the optimal step size depends on the direction in which the step is taken:

$$f_{\text{ellipsoid}}(\mathbf{x}) = \sum_{j=0}^{D-1} 2^j x_j^2 \ . \tag{2.33}$$

Figure 2.31 shows a single contour of constant function value for the two-dimensional version of this function.

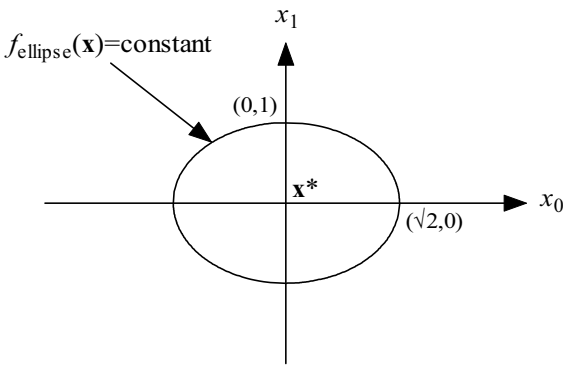

Fig. 2.31. The ellipse is a single contour of the two-dimensional version of the ellipsoidal function described by Eq. 2.33. The optimum, $\mathbf{x}^*$, is located at the origin, (0,0). The principal axes of the ellipse are aligned with the coordinate axes. Taking large steps along x_0 and smaller steps along x_1 efficiently optimizes this function.

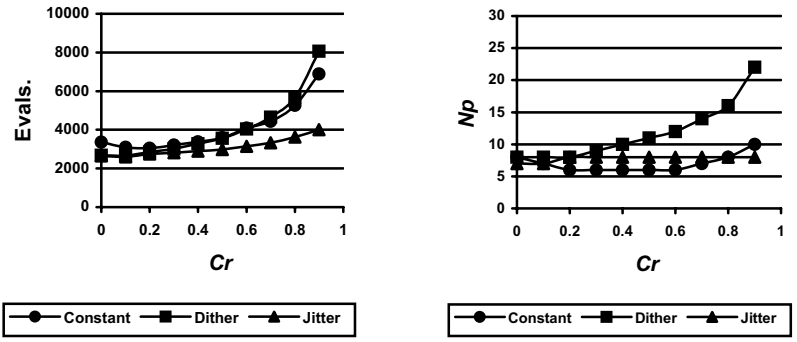

Fig. 2.32. The graph on the left illustrates the effects of jitter, dither and F = constant by plotting, as a function of Cr, the minimum number of function evaluations needed to optimize the ten-dimensional hyper-ellipsoid. The graph on the right plots the corresponding optimal Np at which the minimum number of evaluations occurred. For example, at Cr = 0.9, the graph on the left shows that jitter took about 4000 function evaluations when using the population indicated in the graph on the right at Cr = 0.9, i.e., Np = 8. Results are 1000-trial averages, F = 0.9 and classic DE except for randomizing F with a normal distribution N(0,1).

Figure 2.32 profiles how both jitter and dither influence DE's ability to optimize the ten-dimensional hyper-ellipsoid. Except for requiring roughly 15% more function evaluations, the performance profiles for the hyper-ellipsoid are virtually indistinguishable from those generated for the hyper-sphere.

Before taking these profiles to be universal, it is instructive to perform the same experiment, except that this time, trial vectors are evaluated in a coordinate system that has been rotated 45° with respect to the principal axes of the ellipse (Fig. 2.33).

In two dimensions, this rotated version of the ellipse defined by Eq. 2.33 is

$$f_{\text{ellipse}}(\mathbf{x}) = 2\left(x_0^2 - x_0 x_1 + x_1^2\right). \tag{2.34}$$

As a result of this rotation, the ellipse, which is separable as defined in Eq. 2.33, becomes nonlinear, i.e., parameters become dependent. The cross-term, $x_0 x_1$, in Eq. 2.34 embodies this parameter dependence (see Sects. 1.2.3 and 2.6.2). Even though rotation does not alter the objective function's topography, the parameter dependence that it induces compromises DE's efficiency in the presence of jitter.

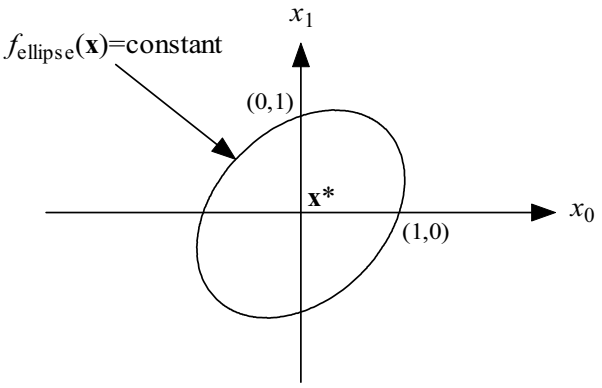

Fig. 2.33. Once rotated, the parameters of the ellipse function become dependent. An efficient search of the long axis along the diagonal now requires that large steps in both coordinate directions occur simultaneously, i.e., that they be *correlated*.

As Fig. 2.34 illustrates, transforming the hyper-ellipsoid from a separable function into one with dependent parameters *via* a coordinate system rotation dramatically alters the performance profiles of all three methods. In particular, the data plotted in Fig. 2.34 show that:

- In contrast to results for the separable hyper-ellipsoid, the fastest solutions now occur at high *Cr*.
- Jitter is now the worst performing method even though population sizes remain relatively small.
- *F* = constant is the fastest method except when *Cr* = 1, in which case dithering is faster.

At *Cr* < 0.5, the fastest run-times begin to occur at population sizes that are too small to produce reliable convergence. To be fair, the number of function evaluations for different methods must be compared at the same probability of convergence. In previous examples, convergence probabilities were so close to 1 that the small differences between them did not compromise the validity of the performance comparisons. A method for evaluating algorithm performance that gives proper weight to convergence probabilities will be presented in Chap. 3 when several version of DE are tested. For now, higher run-times and worse convergence probabilities make it easy to say that DE's performance on the parameter-dependent (rotated) hyper-ellipsoid deteriorates at low *Cr*.

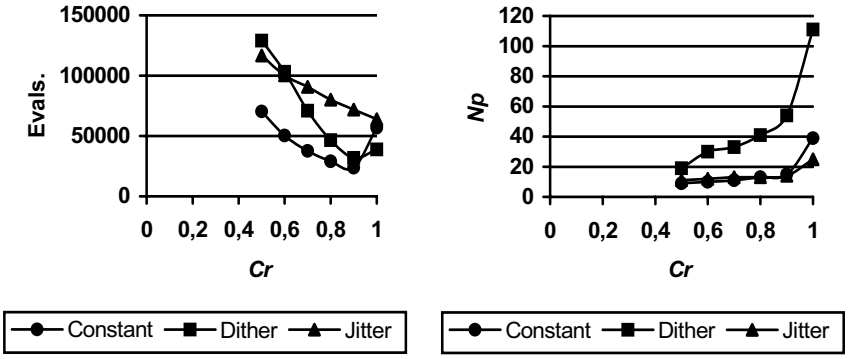

Fig. 2.34. Once rotation induces parameter dependence in the ten-dimensional hyper-ellipsoid, the three techniques become inefficient at low *Cr*. Population sizes used to produce the graph on the left are plotted in the graph on the right at the corresponding value of *Cr*. For example, jitter still uses small populations but is slow nonetheless. Despite using large populations, dither is more efficient than jitter when $Cr > 0.6$ and more efficient than F = constant = 0.9 when $Cr = 1$. Keeping F constant, however, uses relatively small populations and gives the overall fastest result at $Cr = 0.9$. All results are 1000-trial averages with classic DE, except for the indicated randomization scheme. For this experiment, the PDF was the normal distribution, N(0,1).

Since it was the fastest method when the hyper-ellipsoid was separable and was competitive with both dither and F = constant on the rotated hyper-ellipsoid at $Cr = 1$, jitter would seem to be a good strategy as long as *Cr* is chosen wisely. The case of the Chebyshev polynomial, however, suggests differently (see the Appendix for a function description). Like the rotated hyper-ellipsoid, the Chebyshev polynomial fitting problem is a function with dependent parameters that requires correlating step sizes that differ greatly in magnitude from one parameter to the next. Unlike the hyper-ellipsoid, the Chebyshev function is multi-modal. Figure 2.35 compares the number of function evaluations taken by dither to those needed by F = constant to find the coefficients of the nine-dimensional Chebyshev polynomial.

The results in Fig. 2.35 are remarkably similar to those displayed by dither and F = constant for the rotated hyper-ellipsoid in Fig. 2.34, except that now dither gives the overall fastest solution (when $Cr = 1$). Missing from Fig. 2.35 are the results for jitter. Like the case of the rotated hyper-ellipsoid, jitter was most effective when $Cr = 1$, but unlike the case of the rotated hyper-ellipsoid, run-times at this optimal crossover setting were not

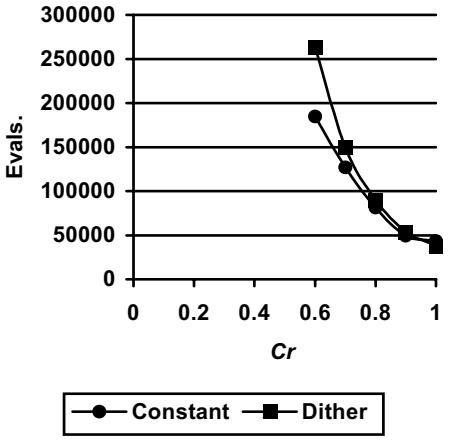

Fig. 2.35. Dither and F = constant perform similarly on the nine-dimensional Chebyshev function, with dither converging in fewer function evaluations than F = constant at $Cr = 1$. Results for jitter are not shown, as run-times were in excess of 6 million function evaluations and convergence was erratic even for large populations. Results are 100-trial averages with $Np = 40$. Both dither and jitter (not shown) used the normal PDF; otherwise, the algorithm was classic DE.

competitive with those turned in by either dither or F = constant. Not only was convergence erratic even with large populations, but the number of function evaluations taken by successful trials never averaged less than 6 *million*, making jitter over 100 times slower than either dither or F = constant. Clearly, these results refute Zaharie's contention that DE's performance is not significantly affected by transforming F into a Gaussian random variable that is sampled anew for each parameter.

Although jitter is effective on separable functions, its poor performance on non-separable, multi-modal functions makes it a questionable strategy for non-linear global optimization with DE *unless the deviations it generates are very small*, e.g., $d = 0.001$ in the case of uniform jitter (see next subsection). The next subsection explores this possibility with some alternatives to the Gaussian PDF.

Other Distributions

The effectiveness of both jitter and dither can be improved by moderating the amount of variation in F_j and F_i, respectively. The problem with Zaharie's formulation in this regard is that as the standard deviation, σ, of the normal (Gaussian) distribution approaches zero, so does F_j (or F_i):

$$F_j = F \cdot \mathrm{N}_j(0,\sigma);\ \ \lim_{\sigma \to 0}\left(F_j\right) = 0 \tag{2.35}$$

To circumvent this difficulty, F can be multiplied by a PDF whose average value is 1, not 0. This way, both dither and jitter revert to the F = constant model as the amount of variation, e.g., σ, approaches zero. Furthermore, the order in which difference vectors are chosen determines the sign of a differential, so a PDF need only generate positive values in order to scale differential magnitudes. A normal distribution can be given an average value of 1 simply by adding one to the zero mean normal PDF, N(0,1), but the resulting distribution will still generate both positive and negative values. The traditional PDF for perturbing scale factor magnitudes is the *log-normal* distribution.

Log-normal. In the ES, not only are the objective function variables mutated and recombined, but so too are the components of the adaptive correlation matrix. Of the correlation matrix's D^2 components, D are scale factors while the remaining $D \cdot (D - 1)$ are rotation angles. Although the ES perturbs rotation angles with normally distributed random variables, it turns to the *log-normal* PDF to mutate the strategy parameters that regulate step sizes (Bäck 1996). An instance of a log-normal random variable *for DE* can be computed as

$$F_j = F \exp\left(\tau\left(N_j(0,1) - \frac{\tau}{2}\right)\right). \tag{2.36}$$

The factor, τ, controls the spread of the distribution while the term $\tau/2$ is an empirically derived factor that normalizes the expected value of the distribution to 1.0. When τ = 0, the average value of the log-normal PDF is the constant value 1, so all $F_j = F$. In this model, the distribution's variance can be controlled independently of F. Figure 2.36 shows how the spread of the log-normal distribution affects DE's ability to optimize both the rotated hyper-ellipsoid and the Chebyshev polynomial fitting problem.

In both plots, jitter requires an increasing number of function evaluations as τ increases. For the Chebyshev polynomial fitting problem, this increase is explosive. By contrast, dither actually shows a slight *decrease* in the number of function evaluations when compared to F = constant, with the best performance occurring near τ = 0.4. The improvement amounts to roughly 10% for the rotated hyper-ellipsoid and just over 40% for the Chebyshev problem.

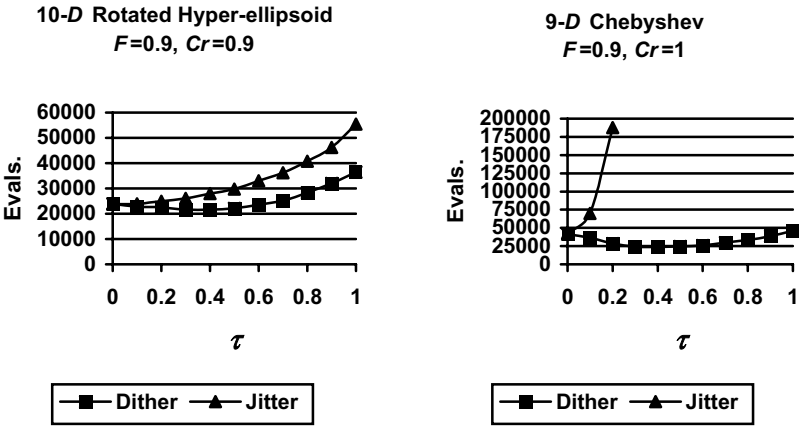

Fig. 2.36. Jitter performs worse as the variance of the log-normal distribution is increased from zero. By contrast, dither is faster than F = constant ($\tau = 0$) on both the rotated hyper-ellipsoid (while $\tau > 0.6$) and the Chebyshev problem (while $\tau \leq 0.9$). In both cases, the fastest convergence occurs near $\tau = 0.4$. Data points are 1000-trial averages for the rotated hyper-ellipsoid and 100-trial averages for the Chebyshev problem. Results were obtained using classic DE except for the indicated randomization method with a log-normal PDF. $Np = 40$.

Uniform. The uniform distribution can also be transformed into a PDF whose average value is F and whose spread is an independent variable. Equation 2.37 illustrates one possibility:

$$F_j = F + d \cdot \left(\text{rand}_j(0,1) - 0.5\right), \quad d < 2F. \tag{2.37}$$

To keep F_i positive, d must be less $2F$. Like τ in the log-normal PDF, d controls the amount of variation in the uniform PDF. The log-normal PDF, however, occasionally generates both very large and very small perturbations, both of which can degrade DE's performance because they tend to slow progress toward the optimum. The uniform distribution with $d \sim F$ effectively eliminates these extremes. Figure 2.37 compares DE's performance on both the rotated hyper-ellipsoid and Chebyshev polynomial fitting problem as a function of the spread, d.

Figure 2.37 shows that as long as $d < 0.1$, jitter remains competitive, although once $d > 0.2$, its performance quickly deteriorates. It should be emphasized, however, that a very small amount of jitter can prove useful, sometimes providing solutions that would otherwise be impossible with F = constant.

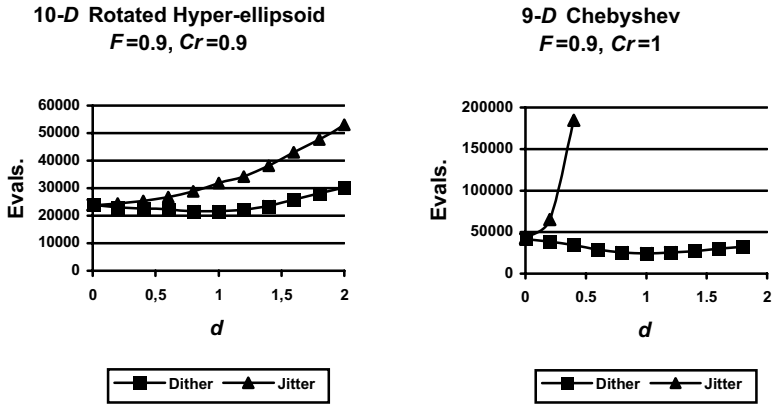

Fig. 2.37. The profiles generated by the uniform and log-normal PDFs are very similar. Jitter's performance worsens as the variation increases and dither converges faster than F = constant (d = 0) when $0 < d < 1.4$. Dither's best performance in this case occurs when $d = 0.9$ (not plotted). Results are 1000-trial averages for the rotated hyper-ellipsoid and 100-trial averages for the Chebyshev problem. In both cases, all trials were successful. The algorithm was classic DE except for the specified randomization technique with the uniform PDF. $Np = 40$.

In particular, experiments with the digital filter design program FIWIZ (Sect. 7.8). have shown that uniform jitter on the order of $d = 0.001$ is often indispensable. In addition, jitter can reduce the size of the population that DE needs to solve a given problem.

Dither's performance changes little when log-normal noise replaces the uniform PDF. The slightly larger optimal population size posted by the log-normal PDF suggests that the small steps present in the log-normal PDF but excluded by the uniform PDF only marginally inflate the optimal population size. The similarly of the two results also suggests that the very large steps generated when $\tau = 0.4$ are too infrequent to have much impact on convergence speed.

Power Law. Just as choosing a PDF complicates the optimization task, so too does having to decide what level of variability is suitable for the normal, log-normal and uniform models. One PDF that avoids this difficulty is based on a power law. An instance of a power law variable can be generated by raising a uniformly distributed random value, rand(0,1), to the power, q, where $q = (1/F) - 1$:

$$F_j = \text{pow}\left(\text{rand}_j(0,1), q\right) = \text{rand}_j(0,1)^q, \quad q = \frac{1}{F} - 1. \tag{2.38}$$

This distribution has F as its average value and when F is between 0 and 1, F_j will also lie in this interval. For example, when $F = 0.5$ and $q = 1$, the distribution is uniform between (0,1). As F approaches either 1 or 0, the amount of variation decreases so that when $F = 1$ all $F_j = 1$ and when $F = 0$ all $F_j = 0$. When $F > 1$, q is negative and all F_j are greater than 1. Table 2.9 reports DE's performance on both the rotated hyper-ellipsoid and the Chebyshev polynomial fitting problems when F_j is a random variable distributed according to the power law in Eq. 2.38.

Table 2.9. At $F = 0.9$, the power law distribution has a small variance, so results for jitter and dither on the ten-dimensional rotated hyper-ellipsoid are close to those for F_i = constant. Nevertheless, the variation is large enough to inflate jitter's function evaluations to twice that of dither in the case of the Chebyshev polynomial fitting problem. Results are 1000-trial averages for the rotated hyper-ellipsoid and 100-trial averages for the Chebyshev function. The algorithm was classic DE except for the stated randomization method using a power law PDF. All trials were successful (P = 1.0).

Function	Rotated hyper-ellipsoid $Cr = 0.9$			Chebyshev $Cr = 1$		
Process	*Np*	Evals.	P	*Np*	Evals.	P
F = constant = 0.9	16	23,208.2	1.0	36	43,608.8	1.0
Dither	16	23,060.5	1.0	34	35,966.8	1.0
Jitter	15	25,212.4	1.0	22	70,358.4	1.0

In both cases, dither's fast convergence did not require a compensating increase in population size. Jitter, although competitive on the rotated hyper-ellipsoid, took twice as many function evaluations to solve the Chebyshev problem as did dither, even though it operated with a smaller population. Still, this is much faster than the 6 million function evaluations that jitter took when driven by the normal PDF, N(0,1). In this model, the amount of jitter cannot be chosen independently of F. For example, using a very small amount of jitter will require F to be very close to 1.

2.6 Recombination

Recombination randomly exchanges or merges parameters from two or more vectors to create one or more trial vectors. *Discrete recombination*, also known as *crossover*, is an operation in which trial vector parameters are copied from randomly selected vectors. Since it only copies information, crossover can be applied to binary, real-valued or even symbolic data. By contrast, *continuous* or *arithmetic recombination* expresses trial vectors

as linear combinations of vectors, so it is inapplicable to symbolic data and inappropriate for binary variables. Both crossover and arithmetic recombination have a variety of implementations. Those with particular relevance to DE are described below.

2.6.1 Crossover

It was originally thought that crossover could exponentially increase the probability of above-average parameter groupings (alleles) while exponentially decreasing the likelihood of less than average groupings (Holland 1973). More recent analysis shows that growth is not exponential because the selective advantage of a parameter grouping decreases as it becomes more prevalent (Macready and Wolpert 1998). Empirical evidence also exists suggesting that (uniform) crossover does not decrease the time complexity of an EA but merely speeds convergence by a constant factor (Mühlenbein and Schlierkamp-Voosen 1993). Nevertheless, crossover plays a significant role in most EAs.

Global discrete recombination refers to the case where both vectors are chosen anew for each trial *parameter* (Bäck and Schwefel 1993). The ES globally recombines its strategy variables, but like DE and most GAs, it crosses objective function parameters from just two vectors (*dual crossover*). Both DE and ES also use crossover to create a single trial vector, whereas most GAs cross two vectors to produce two trial vectors, often by one-point crossover.

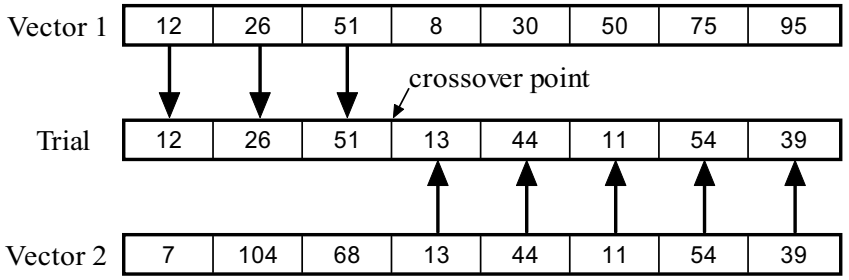

Fig. 2.38. One-point crossover. Each string represents a vector of parameters. In this figure, $D = 8$ and values are integral, although real-valued or symbolic data could also have been used. Each vector contributes a contiguous series of parameter values to the trial vector. The crossover point is randomly chosen. In this case, it occurs between the third and fourth parameters.

One-Point Crossover

There are several ways to assign donors to trial parameters. For example, *one-point crossover* randomly selects a single *crossover point* such that all parameters to the left of the crossover point are inherited from vector 1, while those to the right are copied from the vector 2 (Fig. 2.38) (Holland 1995). GAs often construct a second trial vector by reversing the roles of the vectors, with vector 2 contributing the parameters to the left of the crossover point and vector 1 supplying all trial parameters to the right of the crossover point.

N-Point Crossover

N-point crossover randomly subdivides the trial vector into $n + 1$ partitions such that parameters in adjacent partitions are inherited from different vectors. If n is odd (e.g., one-point crossover), parameters near opposite ends of a trial vector are less likely to be taken from the same vector than when n is even (e.g., $n = 2$) (Eshelman et al. 1989). This dependence on parameter separation is known as *representational* or *positional bias*, since the particular way in which parameters are ordered within a vector affects algorithm performance. Studies of *n*-point crossover have shown that recombination with an even number of crossover points reduces the representational bias at the expense of increasing the disruption of parameters that are closely grouped (Spears and DeJong 1991). To reduce the effect of their individual biases, DE's exponential crossover employs both one- and two-point crossover.

Exponential Crossover

DE's exponential crossover achieves a similar result to that of one- and two-point crossover, albeit by a different mechanism. One parameter is initially chosen at random and copied from the mutant to the corresponding trial parameter so that the trial vector will be different from the vector with which it will be compared (i.e., the target vector, $\mathbf{x}_{i,g}$). The source of subsequent trial parameters is determined by comparing Cr to a uniformly distributed random number between 1 and 0 that is generated anew for each parameter, i.e., $\text{rand}_j(0,1)$. As long as $\text{rand}_j(0,1) \leq Cr$, parameters continue to be taken from the mutant vector, but the *first time* that $\text{rand}_j(0,1) > Cr$, the current *and all remaining* parameters are taken from the target vector. The example in Fig. 2.39 illustrates a case in which the exponential crossover model produced two crossover points.

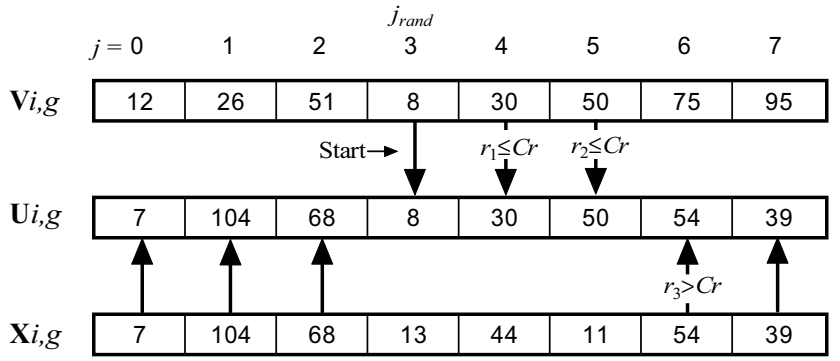

Fig. 2.39. Exponential crossover. Starting at the randomly chosen parameter index, j_{rand} (= 3), trial parameters are inherited from the mutant, $\mathbf{v}_{i,g}$, as long as $\text{rand}_j(0,1) \le Cr$ (e.g., $j = 4, 5$). The first time that $\text{rand}_j(0,1) > Cr$, *all* the remaining trial parameters (e.g., $j = 6, 7, 0, 1, 2$) are inherited from the target vector, $\mathbf{x}_{i,g}$. Indices are computed modulo $D = 8$.

Figure 2.40 describes the process in C-style pseudo-code. Parameter indices are computer modulo D. The exponential method's name reflects the fact that the number of inherited mutant parameters is an exponentially distributed random variable. For example, the probability that the initial, randomly chosen parameter is the trial vector's only mutant parameter is equal to the chance that the first comparison of $\text{rand}_j(0,1)$ and Cr results in a failure, i.e., that $\text{rand}_j(0,1) > Cr$. Thus, the odds of crossover resulting in exactly one mutant parameter are

$$p(x = 1) = 1 - Cr. \tag{2.39}$$

```
jr=floor(rand(0,1)*D); // 0<=jr<D
j=jr;
do
{
   u_{j,i}=v_{j,i};  // Child inherits a mutant parameter
   j=(j+1)%D;// Increment j, modulo D
}while(rand(0,1)<Cr && j!=jr); // Take another mutant parameter?
while(j!=jr) //Take the rest, if any, from the target
{
   u_{j,i}=v_{j,i};
   j=(j+1)%D;
}
```

Fig. 2.40. C-style pseudo-code for DE's exponential crossover scheme

Similarly, the probability that two mutant parameters are inherited is the same as the chance that there will be one success before the first failure:

$$p(x=2)=(1-Cr)\cdot Cr. \tag{2.40}$$

In general, the probability that the trial vector will inherit exactly n mutant parameters is

$$p(x=n)=(1-Cr)\cdot Cr^{n-1}=Cr^{n-1}-Cr^{n}. \tag{2.41}$$

Summing these terms gives the cumulative distribution function. Once summed, only the first and last terms remain, since consecutive contributions contain identical terms of opposite sign that cancel. As a result, the probability that *n or fewer* parameters are inherited from the mutant is

$$p(x\le n)=\sum_{k=1}^{n} Cr^{k-1}-Cr^{k}=1-Cr^{n}. \tag{2.42}$$

One way to eliminate any representational bias associated with the crossover process is to shuffle the vector indices, perform crossover and then un-shuffle the trial vector indices (Caruana et al. 1989). Alternatively, the representational bias inherent in n-point crossover can be eliminated if donors are determined by D independent random trials. This alternative, known as *uniform* crossover, is the discrete recombination method that DE employs most often.

Uniform (Binomial) Crossover

G. Syswerda defined uniform crossover as a process in which independent random trials determine the source for each trial parameter (Syswerda 1989). Crossover is uniform in the sense that each parameter, regardless of its location in the trial vector, has the same probability, p_{Cr}, of inheriting its value from a given vector. For this reason, uniform crossover does not exhibit a representational bias. Syswerda's original definition also allows for the possibility that donors are chosen with different probabilities, but p_{Cr} = 0.5 is the most commonly cited value (both donors are equally probable).

When the vectors being crossed are randomly chosen from the same population, p_{Cr} and $1 - p_{Cr}$ create the same pool of trial vectors. For example, both p_{Cr} = 0.3 and p_{Cr} = 0.7 produce a vector that on average inherits 30% of its parameters from one vector and 70% from another. In particular, when two vectors, A and B, are crossed with p_{Cr} = 0.3, trial vectors will inherit, on average, 30% of their parameters from A and 70% from B. It is equally probable, however, that B will be drawn first and A second, in which case trial vectors inherit, on average, 30% of their parameters from

B and 70% from A. These trial vectors could also have been generated by taking A first, B second and $p_{Cr} = 0.7$. Reversing the roles of the donor vectors has the same effect as using $1 - p_{Cr}$ instead of p_{Cr}. Since the order in which vectors are chosen is random, p_{Cr} potentially generates the same population as does $1 - p_{Cr}$. DE on the other hand crosses vectors from different populations and their order of crossover is not random. In DE, each value of $Cr \sim p_{Cr}$ generates a different trial population.

As with exponential crossover, DE's version of uniform crossover begins by taking a randomly chosen parameter from the mutant so that the trial vector will not simply replicate the target vector. Comparing Cr to $\text{rand}_j(0,1)$ determines the source for each remaining trial parameter. If $\text{rand}_j(0,1) \le Cr$, then the parameter comes from the mutant; otherwise, the target is the source. Figure 2.41 illustrates the process.

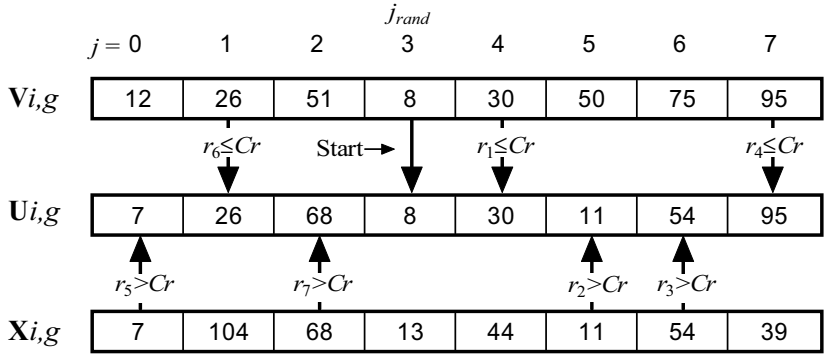

Fig. 2.41. Uniform crossover. Once an initial, randomly chosen parameter is inherited from the mutant (e.g., $j_{rand} = 3$), $D - 1$ independent trials are conducted to determine the source of the remaining parameters. If $\text{rand}_j(0,1) \le Cr$, the mutant donates a parameter value; otherwise, parameters are copied from the target.

The number of inherited mutant parameters follows a binomial distribution, since parameter origins are determined by a finite number of independent trials having two outcomes with constant probabilities. In particular, the odds of successfully inheriting only one parameter from the mutant is the probability that there will be $D - 1$ "failures" occurring with probability $1 - Cr$

$$p(x = 1) = (1 - Cr)^{D-1}. \tag{2.43}$$

More generally, the probability, given D, that exactly n parameters are inherited from the mutant is

$$p(x = n; D) = {}_{D-1}C_{n-1} \cdot Cr^{n-1} \cdot (1 - Cr)^{D-n}, \tag{2.44}$$

$$\text{where } {}_{D\text{-}1}C_{n\text{-}1} \equiv \frac{(D-1)!}{(n-1)! \cdot (D-n)!}.$$

The term ${}_{D-1}C_{n-1}$ represents the number of combinations of $D - 1$ items taken $n - 1$ at a time. Summing the first n terms of Eq. 2.44 gives the probability that the trial vector will inherit at least n mutant parameters. Unlike exponential crossover, the cumulative binomial distribution does not reduce to a simple expression. Because the distribution of inherited mutant parameters is binomial, most DE literature refers to this method as "binomial crossover" to distinguish it from exponential crossover.

Lampinen and Zelinka (2000) have shown that the number of possible trial vectors, n_{trial}, that can be created with DE's uniform (binomial) crossover is

$$n_{\text{trial}} = \begin{cases} Np^3 - 3Np^2 + 2Np & \text{if } Cr = 1 \\ D \cdot Np \cdot \left(Np^3 - 3Np^2 + 2Np\right) & \text{if } Cr = 0 \\ 2^D \cdot Np \cdot \left(Np^3 - 3Np^2 + 2Np\right) & \text{otherwise.} \end{cases} \tag{2.45}$$

Although the number of possible trial vectors is constant when $0 < Cr < 1$, uniform crossover suffers from a *distribution bias* because not all configurations are equally likely (Spears and DeJong 1991). DE does not eliminate distribution bias but relies on Cr to provide the means for controlling it. At one extreme, $Cr \sim 0$ minimizes disruption by incrementally altering just a few parameters of a vector at a time, while at the other extreme, $Cr \sim 1$ favors exploration by drawing most trial vectors directly from the mutant population. The next section examines the conditions under which reinforcement and incremental change are useful and in what contexts exploration becomes crucial.

2.6.2 The Role of *Cr* in Optimization

Despite mediating a crossover process, Cr can also be thought of as a *mutation rate*, i.e., the (approximate) probability that a parameter will be inherited from a mutant. In DE, the average number of parameters mutated for a given Cr depends on the crossover model (e.g., exponential or binomial) but in each, a low Cr corresponds to a low mutation rate. Many GAs recommend a mutation rate of $1/D$, meaning that, on average, only one

trial parameter is mutated (Potter and DeJong 1994). Indeed, Zaharie's results for Rastrigin, Griewangk and the sphere, as well as those for the simple hyper-ellipsoid in Fig. 2.33, consistently found low Cr to be the most effective values. Similarly, optimizing the extensive test beds in Storn and Price (1997) showed that all functions could be solved with either $0 \le Cr \le 0.2$ or $0.9 \le Cr \le 1$. The reason for the bifurcation of the crossover control space was not at first appreciated until it was realized that functions solvable with low Cr were inevitably decomposable, while those requiring $Cr \sim 1$ were not.

Limitations of a Low Mutation Rate

As Sect. 1.2.3 mentioned, a decomposable function can be written as a sum of D one-dimensional functions (not necessarily all the same)

$$f(\mathbf{x}) = \sum_{j=0}^{D-1} f_j(x_j). \tag{2.46}$$

Decomposability simplifies the task of optimization because each parameter can be optimized *independently*, allowing the task of optimizing a single D-dimensional function to be broken up into D one-dimensional problems. Once the optima of the D one-dimensional functions have been located, they can be combined to specify the optimum of the original D-dimensional function

$$f(\mathbf{x}^*) = f(x_0^*, x_1^*, ..., x_{D-1}^*), \quad \min(f_j(x_j)) = f_j(x_j^*), \quad j = 0,1,...,D-1. \tag{2.47}$$

For such functions, changing just one parameter (e.g., $Cr = 0$) before each evaluation can be viewed as a single step in an independent, one-dimensional optimization. If the parameter being modified is randomly selected, then the D one-dimensional optimizations proceed as arbitrarily sequenced tasks (Salomon 1996a).

Any decomposable uni- or multi-modal function can be optimized in linear time, $O(D)$, but randomly interleaving the order in which these one-dimensional optimization tasks are executed causes EAs to incur an additional penalty of $\ln(D)$, raising their total computational complexity for decomposable functions to $O(D \cdot \ln(D))$ (Salomon 1996a). Thus, DE and other GAs with low mutation rates should not be expected to compete with dedicated decomposable function solvers. Such was the case at the First International Contest of Evolutionary Optimizers, held in Kyoto, Japan, where DE finished behind a method that exploited the fact that the contest functions were decomposable (Storn and Price 1996). Even so, the $\ln(D)$ penalty incurred by EAs when using low mutation rates on decomposable

functions is not prohibitive. Once parameters become dependent, however, the penalty incurred by algorithms using low mutation rates does become prohibitive.

Salomon provides two reasons why a low mutation rate is an ill-advised strategy when optimizing parameter-dependent functions (Salomon 1996). The first reason, mentioned briefly in conjunction with the rotated ellipse of Sect. 2.5.2, is illustrated by Fig. 2.42. The picture on the left shows contours of an elliptical objective function whose principal axes are parallel to the coordinate axes. Any trial vector that is interior to the contour on which $\mathbf{x}_i$ resides constitutes an improving move. If only one parameter is changed per evaluation, then $\mathbf{x}_i$ can move at most Δx_0 in the x_0 direction *or* Δx_1 in the x_1 direction before it produces an unacceptable result. For this ellipse, these intervals are large enough to permit the optimum to be located in just two moves, first to either $\mathbf{x}_i + 0.5 \cdot \Delta \mathrm{x}_0$ or to $\mathbf{x}_i + 0.5 \cdot \Delta \mathrm{x}_1$, and then to $\mathbf{x}^*$ on the next move.

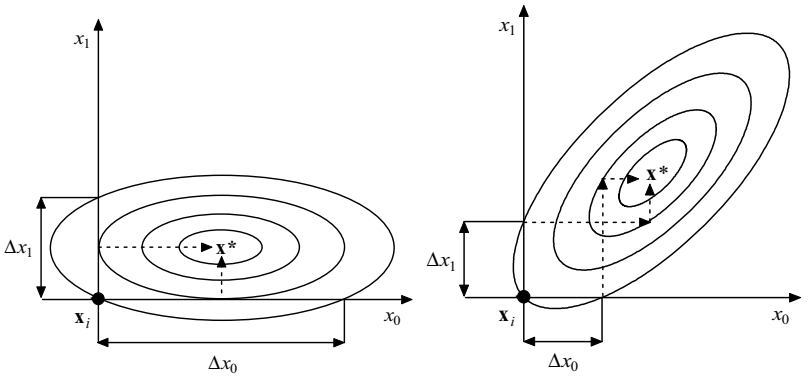

Fig. 2.42. When the principal axes of the ellipse are aligned along coordinate axes, improvement intervals are large compared to those available once the coordinate axes have been rotated by 45°. In the figure on the left, a single pair of moves executed in either order would be able to reach the minimum, but in the figure on the right it takes at least three moves parallel to the coordinate axes to reach the optimum.

By contrast, rotation shortens the improvement intervals to the point where the optimum can no longer be reached in just two consecutive moves *if each step is taken parallel to a coordinate axis*. These additional steps slow convergence and raise the algorithm's time complexity above $O(D \cdot \ln D)$. Both the dimension and eccentricity of the hyper-ellipsoid exacerbate this performance loss. Indeed, the experiments in Sect. 2.5.2 con-

firmed Salomon's predictions that low *Cr*, though efficient on the decomposable ellipse, is inefficient on its rotated, non-decomposable counterpart.

As the example of the rotated ellipse in Fig. 2.42 demonstrates, a low-*Cr* DE strategy can suffer a loss of performance even if the function is unimodal. Salomon's second reason for not using low mutation rates applies only to multi-modal functions whose local minima are not aligned with the coordinate axes. Figure 2.43 shows the contours of a hypothetical multimodal function having two local optima located on a diagonal. The only way to reach the optimum at **x*** from inside the penultimate basin of attraction is by moving in both the (positive) x_0 and x_1 directions *simultaneously*. Since the current vector is in a local optimum, no single move parallel to a coordinate axis will be acceptable and improving moves into a basin of equal or lower function value will have components in both axes.

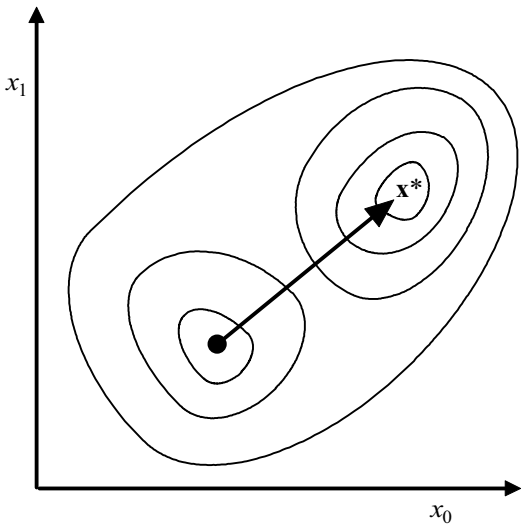

Fig. 2.43. Multi-modal functions with dependent parameters pose additional challenges to low-*Cr* strategies. The only improving move out of the penultimate basin of attraction requires making changes in both coordinates simultaneously.

Salomon has shown that at $O(D^D) = O(\exp(D \cdot \ln(D)))$, a low mutation rate can actually take longer than a random search to optimize a parameter-dependent, multi-modal function (Salomon 1997). Time complexity of this order is prohibitive in all but the most trivial cases.

In summary, the role of *Cr* is to provide the means to exploit decomposability, if it exists, and to provide extra diversity to the pool of possible trial vectors, especially near $Cr = 1$. In the general case of parameter-

dependent functions, *Cr* should be close to 1 so that the performance losses associated with using a low mutation rate are minimized.

Rotational Invariance

An algorithm whose performance depends on the objective function being aligned with a privileged coordinate system is a poor choice in general because it is unlikely that the optimal orientation will be known in advance. What is needed instead is a search algorithm that is *rotationally invariant* – one whose performance does not depend on the orientation of the coordinate system in which the objective function is evaluated. For classic DE, this means that *Cr* = 1, i.e., mutation only and no crossover.

That crossover is not rotationally invariant can be seen in Fig. 2.44, which plots the trial vectors generated by a pair of vectors both before and after a coordinate rotation. Although rotation leaves the position of the vectors with respect to one another unaltered, trial vector placement relative to the vector population depends on the angle of rotation. Since each angle samples different regions of the objective function, performance is rotation dependent.

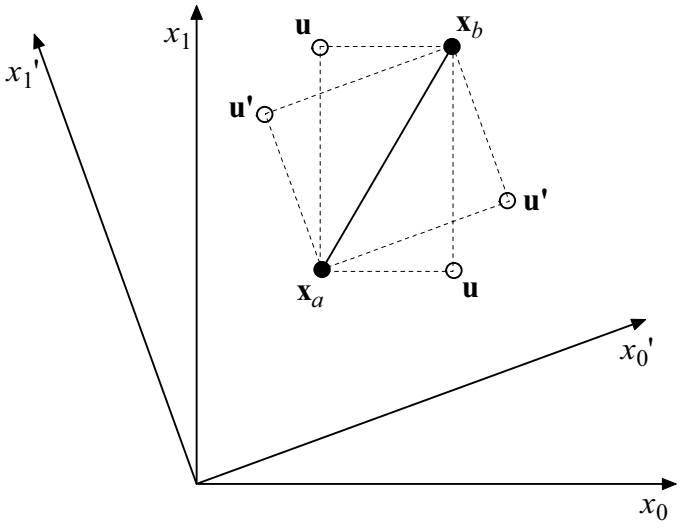

Fig. 2.44. Crossover is not a rotationally invariant process. The trial vectors derived by crossover from vectors $\mathbf{x}_a$ and $\mathbf{x}_b$ change from $\mathbf{u}$ to $\mathbf{u}'$ as the coordinate system is reoriented.

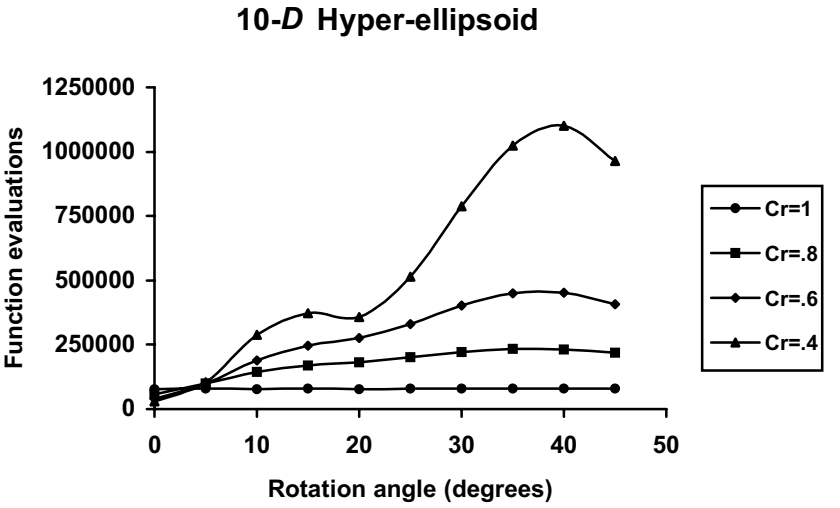

Fig. 2.45. The average number of function evaluations to solve the ten-dimensional hyper-ellipsoid is a function of the angle between the hyper-ellipsoid's principal axes and the axes of the coordinate system in which it is evaluated. Only when $Cr = 1$ (mutation only) is the algorithm's performance independent of the rotation angle. Results are 100-trial averages obtained with classic DE (DE/rand/1/bin) and $F = 0.9$.

Figure 2.45 shows how the time taken by classic DE to optimize the ten-dimensional hyper-ellipsoid depends on the orientation of the hyper-ellipsoid's principal axes with respect to the coordinate system axes in which the trial vector is evaluated. Only when $Cr = 1$ is the number of function evaluations independent of the coordinate system orientation.

Without crossover, classic DE operates by mutation alone. Setting $Cr = 1$, however, ensures that mutation is rotationally invariant only if jitter is absent. For example, Fig. 2.46 shows the regions where jitter relocates the head of a difference vector when $F_j = F + d \cdot (\text{rand}_j(0,1) - 0.5)$ where $d = 0.5$. Because it permits each differential component to be perturbed independently, jitter is an angle-dependent search. The relatively large random deviation illustrated in Fig. 2.46 is necessary to clearly illustrate jitter's rotational dependence, but in practice, such a large value for d would seriously degrade DE's performance on epistatic objective functions. In practice d should be much smaller, e.g., $d = 0.001$.

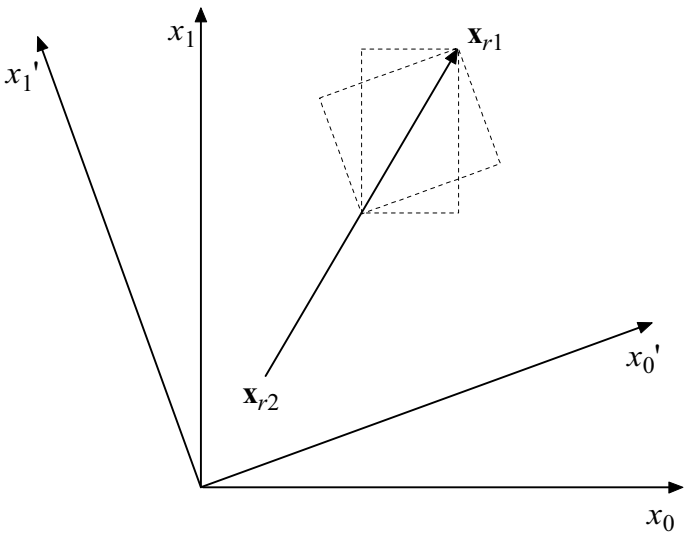

Fig. 2.46. Jitter is not a rotationally invariant process because components of the differential are altered independently. Dashed boxes outline the areas in which jitter with $F_j = 0.5 + 0.5 \cdot \text{rand}_j(0,1)$ can place the head of the difference vector, $\mathbf{x}_{r1} - \mathbf{x}_{r2}$. As the coordinate axes are reoriented, the range of possibilities changes.

Figure 2.47 shows that even with a mutation-only strategy, DE's performance is rotationally dependent if jitter is present (top line). The magnitude of the dependence increases as the magnitude of jitter's deviation increases. On the other hand, dither, like the F = constant model profiled in Fig. 2.45, is rotationally invariant as the lower line in Fig. 2.47 shows. The middle line shows that when jitter is very small (e.g., $d = 0.001$), the penalty for rotational invariance is also small.

Salomon's warnings notwithstanding, DE performs well on parameter-dependent multi-modal functions in practice as long as rotationally invariant processes are the dominant strategies, e.g., when Cr is "close" to 1, say, $Cr = 0.98$, and when jitter's PDF has a "small" variance, e.g., $d = 0.001$ in Eq. 2.37.

The value of such a small value for jitter appears to be that the diversity it adds to the pool of trial vectors lowers the odds that DE will stagnate, particularly when Np is relatively small. This added diversity seems to be of particular benefit to the algorithm DE/best/1/bin, for which reliance on the best-so-far vector as a base vector lowers diversity in the pool of possible trial vectors. In addition, jitter with a suitable PDF makes DE provably convergent. It should be emphasized, however, that jitter's practical value is still a matter of debate.

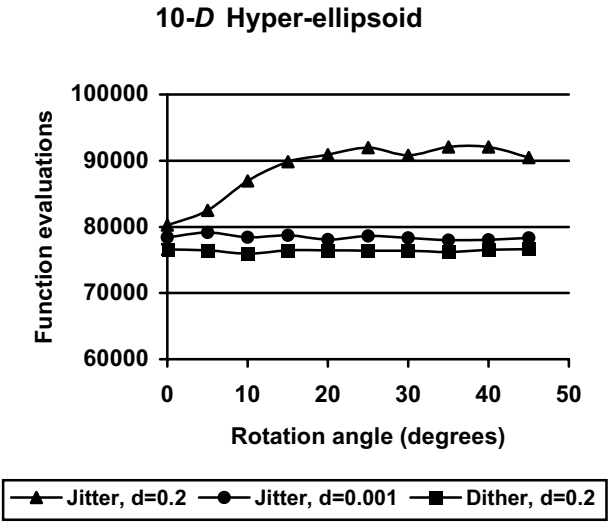

Fig. 2.47. When using jitter, DE's performance on the ten-dimensional hyper-ellipsoid depends on the orientation of the coordinate system relative to the principal axes of the hyper-ellipsoid. Plotted are the number of function evaluations that DE needed to optimize the ten-dimensional hyper-ellipsoid using both jitter and dither in a mutation-only strategy ($Cr = 1$). Unlike jitter, dither is rotationally invariant, but when the level of variation in jitter is very small ($d = 0.001$), rotation does not significantly affect run-times. Results were obtained using $Np = 50$, $Cr = 1$ and classic DE except that $F_i = 0.9 + d \cdot (\text{rand}_i(0,1) - 0.5)$ with $d = 0.2$ for dither and $F_j = 0.9 + d \cdot (\text{rand}_j(0,1) - 0.5)$ with both $d = 0.2$ and $d = 0.001$ for jitter.

If a strictly rotationally invariant scheme is demanded, then $Cr = 1$ and the pool of potential trial vectors is limited to the mutant population. Without crossover or jitter, the only rotationally invariant way to increase the pool of potential trial vectors is by increasing Np or by using dither. If, however, dither's PDF has a high proportion of small perturbations, then optimal population sizes may be larger than if no dithering is used at all. Alternatively, certain forms of *arithmetic recombination* – unlike discrete recombination – can add diversity and complement the mutation search strategy without becoming rotationally dependent.

2.6.3 Arithmetic Recombination

Although crossover creates new combinations of parameters, it leaves the parameter values themselves unchanged. *Continuous* or *arithmetic recombination*, however, operates on individual trial parameter values by ex-

pressing them as linear combinations of parameters. Arithmetic recombination's global variant selects both vectors anew for each parameter of a recombinant vector, $\mathbf{w}_{i,g}$ (Bäck and Schwefel 1993), but most EAs select just one set of vectors for all parameters of $\mathbf{w}_{i,g}$:

$$\mathbf{w}_{i,g} = \mathbf{x}_{r0,g} + k_i\left(\mathbf{x}_{r1,g} - \mathbf{x}_{r0,g}\right). \tag{2.48}$$

The *coefficient of combination*, k_i, can be a constant (e.g., $k_i = 0.5$ is *uniform arithmetic recombination* (Eiben and Smith 2003)), or a random variable (e.g., rand(0,1)). More generally, if k_i is either constant or a random variable that is sampled anew for each vector, then the resulting process is called *line recombination* (Eq. 2.48) (Mühlenbein and Schlierkamp-Voosen 1993). If, however, the coefficient of combination is sampled anew for each parameter, then the process is known as *intermediate recombination* (Mühlenbein and Schlierkamp-Voosen 1993):

$$w_{j,i,g} = x_{j,r1,g} + k_j\left(x_{j,r2,g} - x_{j,r1,g}\right). \tag{2.49}$$

Not all sources agree on this terminology. For example, in ES terminology, the coefficient of combination is chosen anew for each parameter only in the global version, i.e., when vectors are also chosen anew for each parameter (Bäck and Schwefel 1993). This book equates intermediate recombination with the two-vector linear combination in Eq. 2.49, where k_j is a random variable that is sampled anew for each parameter, but vectors are chosen once per trial vector. If k_j is allowed to assume values outside the range (0,1), then the process is called *extended intermediate recombination* (Mühlenbein and Schlierkamp-Voosen 1993).

Figure 2.48 compares the regions searched by discrete, line and intermediate recombination when the coefficient of combination is distributed with random uniformity between 0 and 1. The two vectors occupy opposing corners of a hypercube whose remaining corners are the trial vectors created by discrete recombination. Line recombination, as its name suggests, searches along the axis connecting vectors, while intermediate recombination explores the entire D-dimensional volume contained within the hypercube.

Since the hypercube's corners are the possible outcomes of discretely recombining two vectors, intermediate recombination, like both jitter and crossover, is not a rotationally invariant process. Rotation relocates the hypercube's corners, which in turn redefine the area that intermediate recombination searches. On the other hand, line recombination is rotationally invariant. Given that both differential mutation and line recombination are rotationally invariant schemes for adding a weighted vector difference to

an existing vector, the question arises: what real difference is there between the two operations?

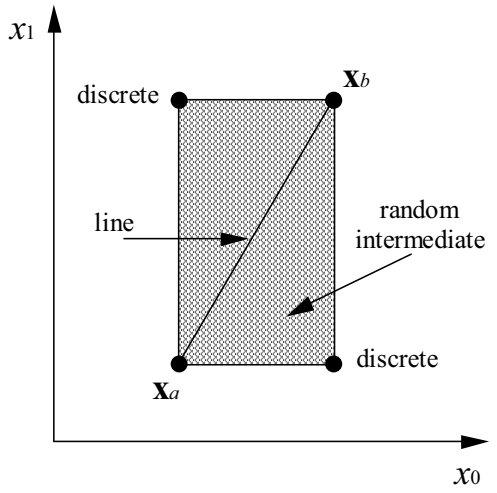

Fig. 2.48. Domains of the possible recombinant vectors generated using discrete, line and intermediate recombination. The coefficient of combination is drawn from the interval [0,1].

Distinguishing Line Recombination from Differential Mutation

Why should some vector differences be associated with recombination and the others not? The reason is that the presence of the base vector in recombination differentials constitutes a bias that makes recombination's dynamics different from those of differential mutation. For example, shifting the base vector's position with respect to the population does not influence its mutation differentials, but it does alter the size and orientation of its recombination differentials. Figure 2.49 shows that if the base vector moves from the population's outer boundary to a more central position, its recombination differentials will become shorter and more symmetrically distributed, whereas mutation differentials – defined by the remaining vectors whose positions are unchanged – are unaffected.

Recombination's positional dependence allows trial vectors to be deliberately placed into the population in locations that mutation can reach only by chance. For example, $k_i = 0.5$ (Eq. 2.48) places the trial vector midway between the base vector and the vector $\mathbf{x}_{r1}$. Moreover, $k_i = 1$ reduces re-

combination to a replacement operation by placing the trial vector at $\mathbf{x}_{r1}$. By contrast, non-zero mutation differentials place trial vectors on, between or in relation to other vectors only by chance, not by intention.

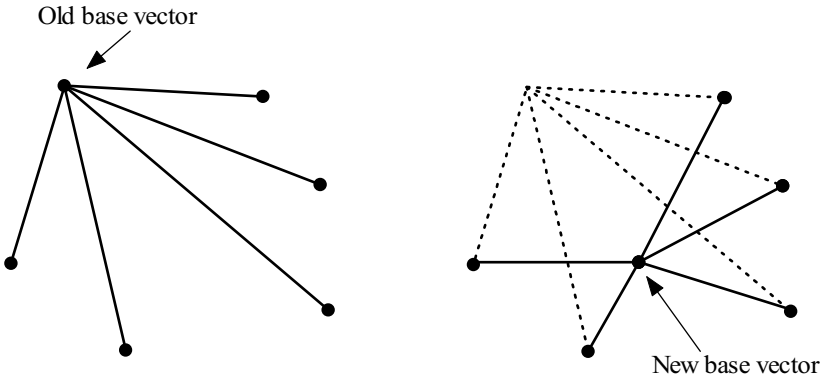

Fig. 2.49. Recombination differentials change in response to a shift in the base vector's position relative to the population.

When a trial vector is a linear combination of only two vectors, the differential's dependence on the base vector is inevitable. For example, let **u** be a trial vector that is a linear combination of two, randomly chosen vectors

$$\mathbf{u} = k_0 \cdot \mathbf{x}_{r0} + k_1 \cdot \mathbf{x}_{r1}. \tag{2.50}$$

To prevent trial vectors from expanding ($k_0 + k_1 > 1$) or contracting ($k_0 + k_1 < 1$) over the course of many generations due only to the generating process itself, the coefficients k_0 and k_1 are subject to a *normalization constraint* that requires their sum to equal 1. For a linear combination of m vectors,

$$\begin{aligned} &\sum_{i=0}^{m-1} k_i = 1, \\ &k_0 + k_1 = 1, \quad (m = 2). \end{aligned} \tag{2.51}$$

Substituting $1 - k_1$ for k_0 in Eq. 2.50 yields the familiar formula for line recombination in which the base vector, $\mathbf{x}_{r0}$, *also appears in the difference term*:

$$\begin{aligned} \mathbf{u} &= (1 - k_1) \cdot \mathbf{x}_{r0} + k_1 \cdot \mathbf{x}_{r1}, \\ &= \mathbf{x}_{r0} + k_1 \cdot (\mathbf{x}_{r1} - \mathbf{x}_{r0}). \end{aligned} \tag{2.52}$$

Once three vectors are linearly combined, however, the positional bias inherent in two-vector combinations can be eliminated. For example, a mutant is a three-vector linear combination that is subject to two constraints. The normalization constraint, $k_0 + k_1 + k_2 = 1$, eliminates one of the three coefficients of combination (k_0) and reduces the expression for a general linear combination of three vectors to

$$\mathbf{u} = \mathbf{x}_{r0} + k_1 \cdot (\mathbf{x}_{r1} - \mathbf{x}_{r0}) + k_2 \cdot (\mathbf{x}_{r2} - \mathbf{x}_{r0}). \tag{2.53}$$

Imposing the *mutation constraint*

$$\begin{aligned} &\sum_{i=1}^{m-1} k_i = 0, \\ &k_1 = -k_2, \quad (m = 3), \end{aligned} \tag{2.54}$$

both eliminates a second coefficient (k_1) *and* removes $\mathbf{x}_{r0}$ from the difference term

$$\mathbf{u} = \mathbf{x}_{r0} + k_2 \cdot (\mathbf{x}_{r1} - \mathbf{x}_{r2}) = \mathbf{v}. \tag{2.55}$$

Satisfying Eq. 2.54 cancels out the base vector's contribution to the $m - 1$ differential terms. The one remaining coefficient of combination, k_2, is the mutation scale factor, F. Like the increments generated by a PDF, the mutation differentials contain no reference to the vector they modify.

Two-vector line recombination's positional dependence complements a mutation-driven search, but the existence of only $Np - 1$ possible recombination axes limits its explorative power. More than two vectors can be recombined and elevating line recombination to a three-vector process places it on an equal footing with differential mutation as both consist of a linear combination of three vectors.

Three-Vector Recombination

Equation 2.51 appears to be missing the differential mutation operator because it expresses a trial vector as the sum of the base vector and two recombination differentials that contain the base vector. The reciprocal roles played by recombination and mutation in three-vector linear combinations become clearer once Eq. 2.53 is rewritten with a change of variables that decomposes any normalized, three-vector linear combination into separate recombination and mutation components, K and F, respectively. First, let

$$k_1 \equiv \frac{(K+F)}{\sqrt{2}}, \qquad k_2 \equiv \frac{(K-F)}{\sqrt{2}}. \tag{2.56}$$

Replacing k_1 and k_2 in Eq. 2.53 with the expressions in 2.56 yields

$$\mathbf{u} = \mathbf{x}_{r0} + \frac{(K+F)}{\sqrt{2}} \cdot (\mathbf{x}_{r1} - \mathbf{x}_{r0}) + \frac{(K-F)}{\sqrt{2}} \cdot (\mathbf{x}_{r2} - \mathbf{x}_{r0}). \tag{2.57}$$

Multiplying out the expressions in Eq. 2.57 and collecting terms reorganizes Eq. 2.53 into a recombination term that contains the base vector and a mutation term from which the base vector is absent:

$$\mathbf{u} = \mathbf{x}_{r0} + \frac{K}{\sqrt{2}} \cdot (\mathbf{x}_{r1} + \mathbf{x}_{r2} - 2\mathbf{x}_{r0}) + \frac{F}{\sqrt{2}} \cdot (\mathbf{x}_{r1} - \mathbf{x}_{r2}). \tag{2.58}$$

The change of variables laid out in Eq. 2.56 defines a 45° rotation of the *K*–*F* plane with respect to the k_1–k_2 plane (Fig. 2.50). The mutation constraint, $k_1 = -k_2$, defines a mutation axis, *F*, that passes through the origin and has a slope of −1, while the *recombination constraint*, $k_1 = k_2$, defines a recombination axis, *K*, that also passes through the origin but has a slope of +1. The advantage of the *K*–*F* decomposition is that it permits two search processes with different dynamics to be controlled independently.

The coordinates, (k_1,k_2), locate the trial vector, **u**, *relative to* the base vector $\mathbf{x}_{r0}$ using two-vector recombination differentials as basis vectors. Coordinate k_1 measures the distance of the trial vector from the base vector in the direction of the differential $(\mathbf{x}_{r1} - \mathbf{x}_{r0})$, while k_2 measures the distance from $\mathbf{x}_{r0}$ in the direction of the differential $(\mathbf{x}_{r2} - \mathbf{x}_{r0})$. Similarly, *K* and *F* measure the distance of the trial vector from the base vector along the direction of the three-vector recombination and mutation axes, respectively.

The medial line (the *K*–axis in Figs. 2.50 and 2.51) plays an important role in the two-dimensional version of the Nelder–Mead algorithm. As Sect. 1.2.3 explained, the Nelder–Mead strategy tests a point located on the axis defined by the vector being modified (the worst vector in Nelder–Mead, but $\mathbf{x}_{r0}$ in this case) and the centroid of a simplex consisting of *D* additional vectors. When *D* = 2, this axis is a medial line that passes through not only the centroid, but also the average position of $\mathbf{x}_{r1}$ and $\mathbf{x}_{r2}$.

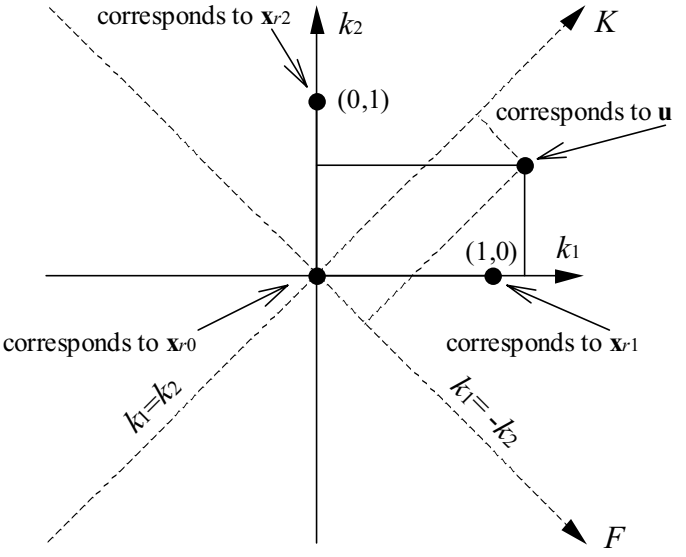

Fig. 2.50. Decomposing the position of a trial vector into separate mutation and recombination components in the *K–F* plane (refer to Eqs. 2.51 and 2.56). The rotation angle between the k_1–k_2 and *K–F* coordinate systems is 45°.

Figure 2.51 illustrates some of the important features of the *K–F* plane. Coordinates are given as (*K,F*) where *K* and *F* are a vector's coordinates along the recombination and mutation axes, respectively. The base vector, $\mathbf{x}_{r0}$, corresponds to the origin, (*K,F*) = (0,0). The remaining two vectors correspond to (0.5,0.5) and (0.5, –0.5). Together, the three vectors form an inverted triangle whose sides and their extensions constitute the three axes along which three-vector combinations reduce to two-vector line recombination. This triangle of vectors is inscribed inside a larger triangle whose vertices are the three *mutation points* (0,1), (0, –1) and (1,0) corresponding to the vectors $\mathbf{x}_{r0} + \mathbf{x}_{r1} - \mathbf{x}_{r2}$, $\mathbf{x}_{r0} + \mathbf{x}_{r2} - \mathbf{x}_{r1}$ and $\mathbf{x}_{r1} + \mathbf{x}_{r2} - \mathbf{x}_{r0}$, respectively.

Only the order in which vectors are combined distinguishes these three strategies and as long as vectors are randomly selected, the three mutation points are dynamically indistinguishable, i.e., the three strategies cannot be distinguished based on their performance. Similarly, the sides of this larger triangle represent the three possible mutation axes and its three medial lines represent the three possible recombination axes. The figure is bilaterally symmetric (left and right sides are mirror images, with the mirror aligned on the *K* axis) about the vertical recombination axis because $\mathbf{x}_{r1} - \mathbf{x}_{r2} = -(\mathbf{x}_{r2} - \mathbf{x}_{r1})$. The centroid of both the large and small triangles lies at (1/3,0).

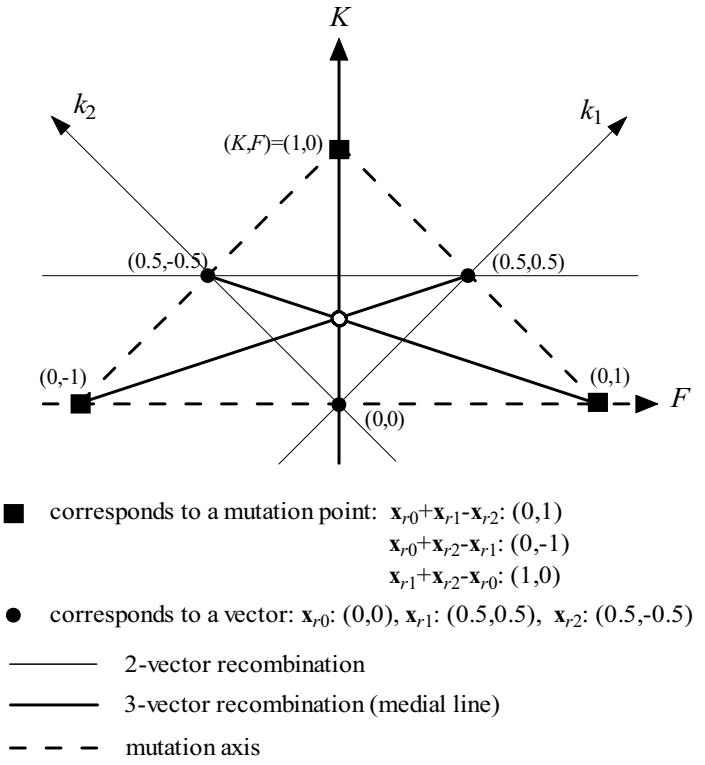

Fig. 2.51. The *K–F* plane exhibits three axes along which two-vector recombination produces trial vectors. Squares plot the three dynamically equivalent mutation points. The vertical axis measures the component along the medial axis while the horizontal mutation axis measures the component in the direction of the difference vector $\mathbf{x}_{r1} - \mathbf{x}_{r2}$. Note that the coordinate values are expressed in the *K–F* coordinate system. Note also that the vectors that correspond to these points are also mentioned. As an example, the point (1,0) corresponds to the vector $\mathbf{x}_{r1} + \mathbf{x}_{r2} - \mathbf{x}_{r0}$.

Because they represent varying fractions of recombination and mutation, points in the *K–F* plane also represent different search strategies. For example, classic DE with $Cr = 1$ includes all the points on the mutation axis where trial vectors are pure mutants, whereas those that lie along the medial axes are pure three-vector recombinants similar to those produced by the two-dimensional Nelder–Mead algorithm. Off-axis points possess attributes of vectors that have been subjected to both differential mutation and three-vector line recombination. For optimization, the most important questions regarding *K* and *F* are whether they are correlated and whether successful strategies consistently cluster around landmarks in the *K–F*

plane. The phase portrait is designed to provide insights into these questions.

2.6.4 Phase Portraits

Phase portraits are a visual aid for exploring relationships between control parameters, in this case K and F. Each point, (K,F), in the K–F plane locates a point representing a trial vector generating strategy that is iterated over many generations. If the point at (K,F) is plotted when the strategy it represents is successful within the allotted number of generations, then a "portrait" forms revealing the location of effective control variable combinations for the given test function. Rosenbrock's function, for example, displays the portrait in Fig. 2.52.

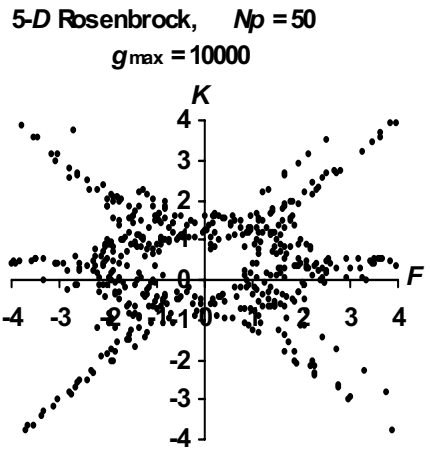

Fig. 2.52. The phase portrait for the five-dimensional generalized Rosenbrock function. Points were sampled with random uniformity, i.e., $F = 8\cdot(\text{rand}(0,1) - 0.5)$ and $K = 8\cdot(\text{rand}(0,1) - 0.5)$. The function rand(0,1) lacks a subscript to indicate that a single value is generated anew *for each optimization run*. One optimization was run for each point. If the optimization was successful within the allotted number of generations and with the chosen population size, the point was plotted. Results were obtained with DE/rand/1/bin, $Cr = 1$ and $i \neq r0 \neq r1 \neq r2$.

Regions that are most densely populated correspond to strategies that have the highest probability of convergence. Points in the central triangular void are strategies that converged prematurely for the given Np, while those in the vacant space surrounding the portrait did not converge within the allowed number of generations.

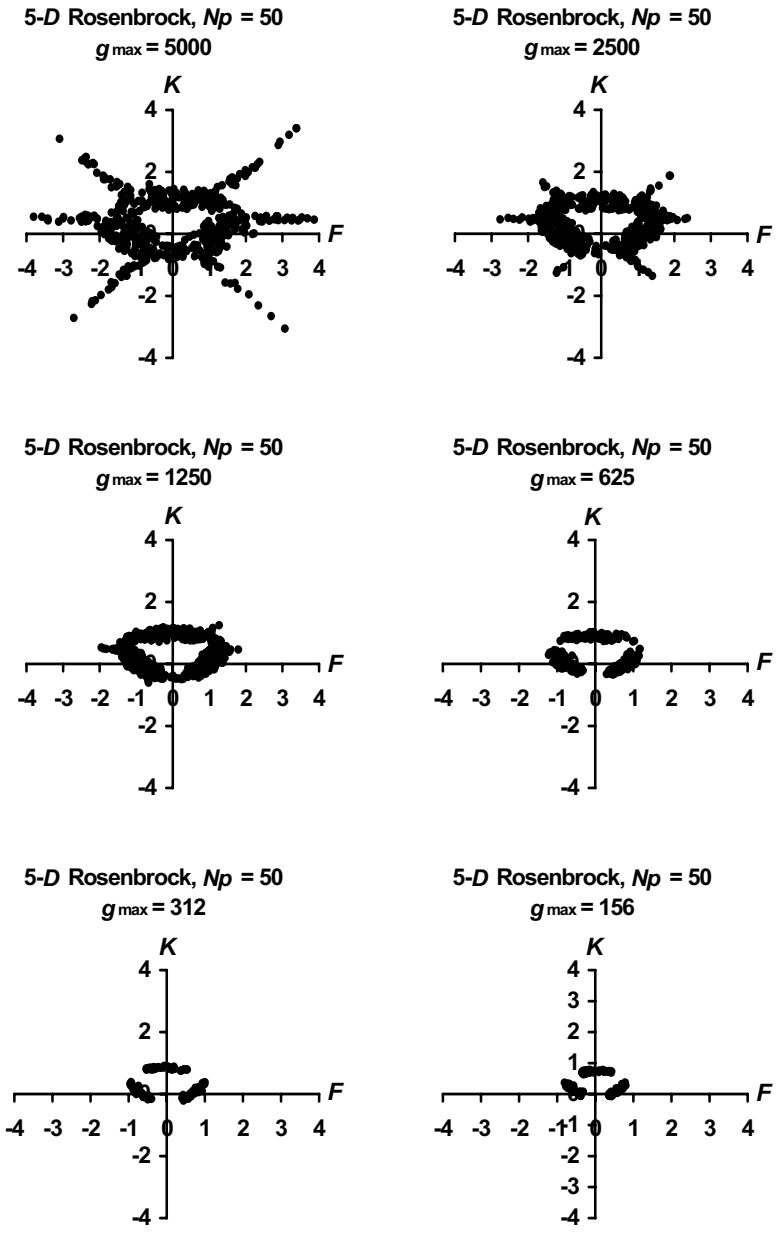

Fig. 2.53. Reducing the maximum allowed number of generations reveals that the fastest solutions are the most interior ones. The three clusters represent symmetric solutions.

The distribution of successful strategies highlights several important features of the *K–F* plane shown in Fig. 2.51. For example, the distribution of successful strategies is bilaterally symmetric about the vertical recombination axis. In addition, the six spikes correspond to the three cases of two-vector line recombination. Their presence in Rosenbrock's portrait shows that even two-vector line recombination is sufficient to solve this unimodal function if the coefficient of combination (e.g., k_1 in Eq. 2.50) is large enough.

By successively halving the maximum allowed number of generations, g_{max}, in successive portraits, Fig. 2.53 shows that the solutions obtained by two-vector recombination are relatively time consuming and that the fastest solutions are the most interior ones.

Figure 2.54 shows the final Rosenbrock portrait in Fig. 2.53 at expanded scale with medial lines and lines of two-vector recombination drawn for reference.

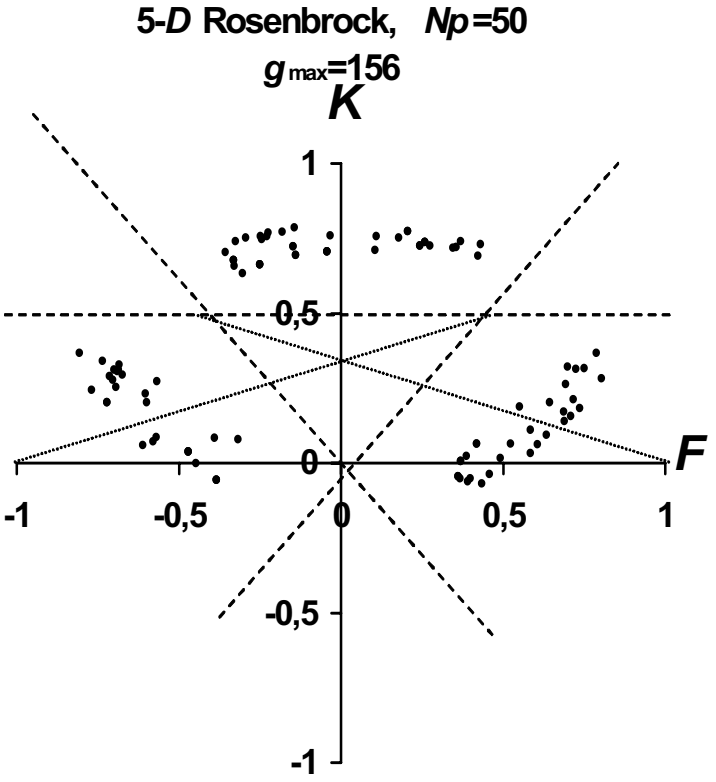

Fig. 2.54. Clusters for Rosenbrock's functions are bisected by a medial line and constrained by the lines of two-vector recombination.

Even though they possess very different topography, many other functions display portraits similar to Rosenbrock's. As Fig. 2.55 shows, the phase portraits for the hyper-ellipsoid, Ackley, Whitley and Lennard-Jones functions all look remarkably similar to Rosenbrock's portrait when g_{max}= 5000.

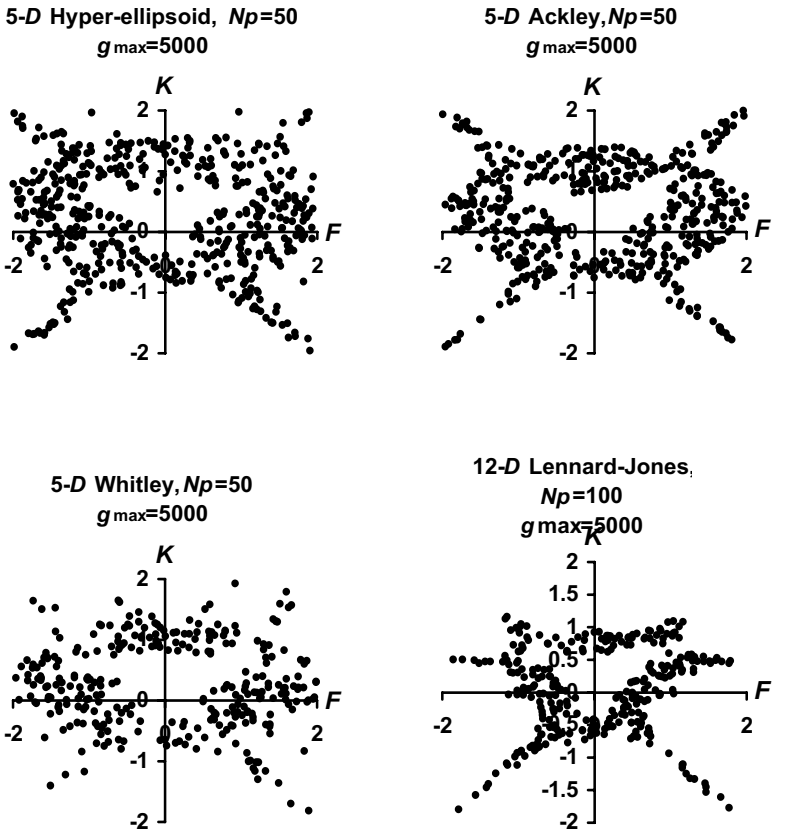

Fig. 2.55. Despite having radically different topographies, these functions produce portraits similar to Rosenbrock's.

Other portraits, such as those of the Chebyshev and Hilbert functions in Fig. 2.56, produce images similar to plots of Rosenbrock's fastest strategies. In each case there are three clusters centered on a medial line that are constrained by the lines representing two-point recombination.

Not all functions conform to this pattern and some have portraits with clusters that lie predominantly along *either* the mutation axis *or* the three-

vector recombination axis. Figure 2.57 shows that most of the successful strategies for the Shekel and odd square functions lie on the mutation axis.

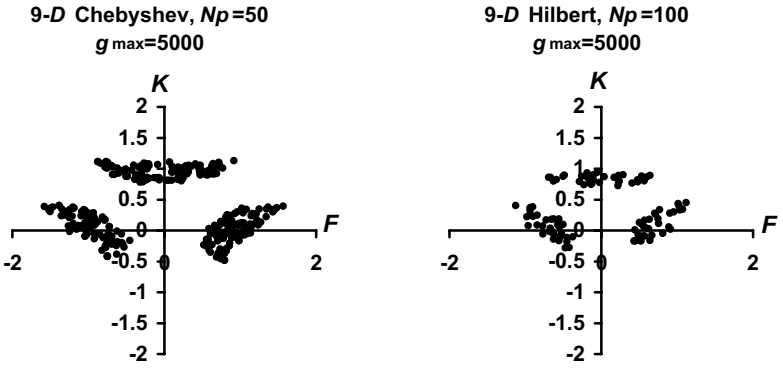

Fig. 2.56. Portraits for both the Chebyshev and Hilbert functions are almost indistinguishable from Rosenbrock's innermost strategies.

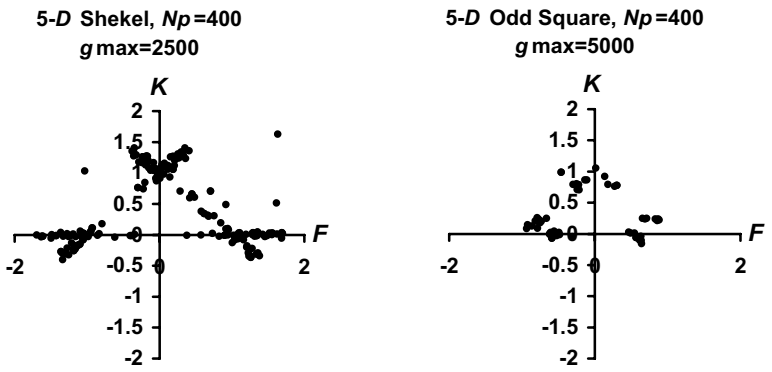

Fig. 2.57. Solutions for both the Shekel and odd square functions lie almost entirely on the mutation axes. Recombination is effective on the Shekel function as long as K is close to 1, but it is an ineffective strategy when applied to the odd square.

At the other extreme, Griewangk's function shows a distribution of points centered on the medial lines that has only a few outlying points approaching the mutation axis, most notably near $F = 1$ and $F = 0.5$.

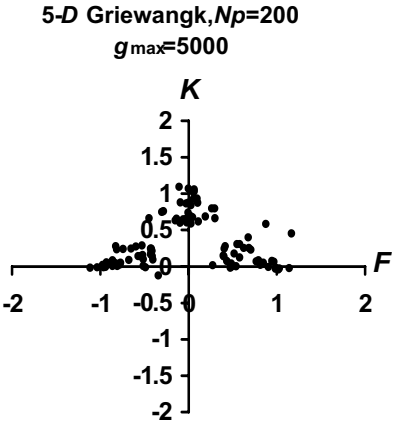

Fig. 2.58. Although a few cluster points intersect the mutation axes, the most robust strategies lie on the medial axes.

These phase portraits show that mutation and recombination differentials do indeed have different effects on the optimization dynamic. For functions like the odd square, mutation is the only viable option, while for those like Griewangk, recombination is a better strategy. Reliance on the wrong operation is likely to result in poor performance for a significant number of functions, but many functions are *generic*, meaning that either mutation or recombination makes an effective strategy. Given the range of behaviors displayed in the phase portraits, what is the best strategy in general?

2.6.5 The Either/Or Algorithm

All portraits in the previous section displayed clusters of successful strategies that were bisected by either a recombination or a mutation axis. In the generic case, both axes intersected clusters. Furthermore, there was no case in which a cluster *only* occupied the spaces between axes. Because these isolated, off-axis clusters are not observed, the best strategy for locating a central cluster point is to look along the mutation axis, the recombination axis, or both, but not between them. Compared to searching the entire two-

dimensional *K*–*F* plane, a dual-axis search reduces the effort to find a successful strategy because it restricts the search to a pair of one-dimensional axes.

The simplest way to implement a dual-axis search is to define a mutation probability such that trial vectors that are pure mutants occur with probability p_F and those that are pure recombinants occur with probability $1 - p_F$:

$$\mathbf{u}_{i,g} = \begin{cases} \mathbf{v}_{i,g} = \mathbf{x}_{r0,g} + F \cdot \left(\mathbf{x}_{r1,g} - \mathbf{x}_{r2,g}\right) & \text{if } \text{rand}_i(0,1) < p_F, \\ \mathbf{w}_{i,g} = \mathbf{x}_{r0,g} + K \cdot \left(\mathbf{x}_{r1,g} + \mathbf{x}_{r2,g} - 2\mathbf{x}_{r0,g}\right) & \text{otherwise.} \end{cases} \tag{2.59}$$

This scheme accommodates functions that are best solved by either mutation only ($p_F = 1$) or recombination only ($p_F = 0$), as well as generic functions that can be solved by randomly interleaving both operations ($0 < p_F < 1$). Figure 2.59 gives pseudo-code for this "either/or" algorithm.

```
...
if (rand_i(0,1)<P_F)            // mutate or recombine ?
{
   u_i=x_r0+F*(x_r1-x_r2);      // mutate
}
else
{
   u_i=x_r0+K*(x_r1+x_r2-2*x_r0); // recombine
}
...
```

Fig. 2.59. Pseudo-code for creating a trial vector with the "either/or" algorithm. From experience $K = 0.5 \cdot (F + 1)$ can be recommended as a good first choice for K given F.

2.7 Selection

There are primarily two stages in the evolutionary process where selection can be applied to a population. Some GAs (Goldberg 1989) employ *parent selection* to decide which vectors will undergo recombination. Typically, vectors with the best function values are assigned the highest *selection probability*, making them the most likely to be chosen for mating. This strategy mimics the one employed by breeders and botanists who try to

improve traits by selectively breeding individuals with superior characteristics. In practice, methods for assigning selection probabilities involve additional assumptions about how to map objective function values to a set of probabilities. Instead of selecting mates based on objective function value, both ES and classic DE select mates with equal probability. In the ES, each vector has the same chance to be chosen for mutation and/or recombination. Similarly, classic DE randomly selects base vectors without regard for their objective function values (see Sect. 2.4).

In contrast to parent selection, s*urvivor selection*, also called *replacement*, chooses the next generation of vectors from the current generation of vectors and trial vectors. Most EAs apply selection pressure either when choosing vectors to recombine or when choosing survivors. GAs typically bias selection in favor of better vectors, whereas DE, ES and other EAs, however, combine randomly chosen vectors and apply selection pressure only when picking survivors. Using both parent (base vector) and survivor selection can cause premature convergence to a local optimum.

The remainder of this section is primarily concerned with survivor selection and it will be convenient for the following discussion to assume that the current and trial populations can have different sizes. In keeping with the naming traditions established by the ES community, μ will denote the size of the current population and λ will represent the size of the trial population.

2.7.1 Survival Criteria

In some algorithms, age alone determines which individuals survive. Here, age distinguishes vectors in the current population from those in the (younger) trial population. More often, however, both a vector's objective function value and the luck of the draw are also factors. The simple GA, however, determines survivors by their age alone.

Age Only

The simple GA replaces μ vectors with $\lambda = \mu$ trial vectors without regard to whether the trial vectors actually have lower function values than those in the current generation (Goldberg 1989). This *age-based replacement* scheme only works if parent selection is driven by an objective function-based criterion. Without the feedback that an objective function-based parent selection rule provides, there is no bias to drive the population toward better solutions. For example, the (1,1)-ES with its age-based selection is nothing more than a random walk in which each trial vector replaces the

current vector regardless of its objective function value (Bäck et al. 1993). Similarly, age-based replacement is unsuitable for classic DE because its parent selection scheme, i.e., random base vector selection, does not choose vectors based on objective function value.

Objective Function Value Only

When only trial vectors are allowed to advance, there is no guarantee that the best-so-far solution will not be lost. Retaining the best-so-far solution is known as *elitism* and part of the task of proving that an algorithm will converge to the global optimum in the long-time limit is proving that it is elitist (Rudolph 1996). For this reason, and because of the speed improvement that it offers, most EAs, including DE, evolutionary programming (EP) and some versions of ES and genetic programming (GP) (Koza 1992), include the current population when determining the membership of the next generation. For example, the $(\mu + \lambda)$-selection scheme (see Sect. 1.2.3) ranks all vectors in both the current and trial populations from best to worst and then populates the next generation with the best μ individuals. Similarly, EP tournament selection (see subsection 2.7.2) compares the objective function value of vectors randomly chosen from the current and trial populations. In both cases, a vector's age is irrelevant and the best-so-far result is always retained.

Age and Objective Function Value

In ES (μ, λ)-selection (see Sect. 1.2.3), age dictates that only trial vectors can survive, while objective function values determine which trial vectors are among the μ best. Using objective function values to pick the μ best survivors from a pool of λ trial vectors biases evolution toward better solutions, unlike the simple GA in which $\mu = \lambda$ trial vectors survive regardless of their objective function values. Since surviving trial vectors can overwrite better current vectors, (μ, λ)-selection is not elitist. Forgetting prior results, however, allows the population both to escape local optima and to track dynamic ones. In addition, (μ, λ)-selection lessens the chance that ES "strategy" parameters will prematurely adapt to a good but sub-optimal solution (Bäck and Schwefel 1995).

As the next section shows, both age and objective function value also play a role in DE selection. Age is a factor because trial vectors can only compete against members of the current population, while their objective function values determine which vector survives. The DE scheme is elitist since the best vector in the current and trial populations always survives.

2.7.2 Tournament Selection

In general, any parent selection scheme can also be adapted for survivor selection, but in practice nearly all EAs, including DE, determine survivors by some form of tournament selection or ranking, which is a special case of tournament selection. The next subsection explores DE selection in the context of the tournament survivor selection method employed by the EP algorithm (Fogel et al. 1966; Fogel 1991).

In EP-style tournament selection, each vector competes against T opponents drawn at random from a selection pool of Ns vectors (Saravanan and Fogel 1997). In deterministic tournaments, vectors are assigned a "win" for each pair wise competition in which they have the lower objective function value (in non-deterministic tournaments, the best vector wins with a user-defined probability). The μ vectors that accumulate the most wins populate the next generation.

The main control variable in tournament selection is the tournament size, T, where $2 \leq T \leq Ns$. A typical tournament size for the EP algorithm is $T = 10$. DE, however, conducts Np, *binary* tournaments ($T = 2$) in which only two individuals compete. In general, the selection pressure increases as T increases, i.e., increasing T speeds convergence, so compared to EP tournament selection ($T = 10$), DE selection is gentler. DE's lower selection pressure helps avoid premature convergence without the introduction of variation operators to enhance the diversity of the pool of potential trial vectors.

Ranking (e.g., $(\mu + \lambda)$-selection in which both the current and trial populations are sorted based on objective function value) is a special case of EP tournament selection for which $T = Ns$. For example, if one vector is better than another, the better vector will win all the same tournaments that the inferior vector wins plus the tournament with the inferior vector itself. Since better vectors always have more wins than inferior vectors, conducting $T = Ns$ tournaments for each vector ensures that ranking vectors by the number of wins also ranks them by objective function value. In practice, ranking is accomplished more efficiently by sorting the population from best to worst based on objective function value and then taking the top μ individuals. Efficient sorting reduces the computational complexity of the each-against-all tournament process from $O(Ns^2)$ to $O(Ns \cdot \log(Ns))$ (Blahut 1984).

Tournament selection is very versatile because it only depends on knowing which of two solutions that have been paired for competition is better. Because it only depends on the difference between objective function values, tournament selection is unaffected when a constant is added to every

vector's objective function value (transposition) (Eiben and Smith 2003). By contrast, *fitness proportional selection* selects an individual with a probability based on its function value (Holland 1992)

$$P_i = \frac{f(\mathbf{x}_i)}{\sum_{i=0}^{\mu-1} f(\mathbf{x}_i)} \tag{2.60}$$

and adding a constant to each vector's objective function value will change its fitness proportional selection probability.

Tournament selection is also well suited for co-evolutionary optimization tasks in which the quality of a given solution is defined only in the context of its performance with respect to the rest of the population. For example, it is difficult to rate an arbitrary checkers strategy, but it is a simple matter to determine which of two strategies is better by actually using them to play one or more games against each other. Similarly, tournament selection is the most effective way to evolve solutions to "subjective" objective functions, like those used in evolutionary art. In such environments, it is easier to decide, for example, which of two pictures is more pleasing than it is to decide how pleasing a picture is in an absolute sense. In addition, tournament selection permits the concept of Pareto-dominance to be implemented for both constraint functions and for multi-objective optimizations (see Sects. 4.3 and 4.6).

A single competition in an EP tournament might select two current population vectors, a current and a trial vector, or two trial vectors to compete against one another. DE, however, restricts tournament selection to this last possibility in which each competition pits a trial vector against a vector in the current population (the target) with the additional proviso that the target and trial vectors are also related by crossover. The next subsection explores this special class of deterministic, binary tournaments, known as one-to-one selection for the way in which population and trial vectors are paired for competition.

2.7.3 One-to-One Survivor Selection

Besides pairing competitors based on age, DE's one-to-one replacement scheme differs from EP tournament selection in other ways. For example, an EP-style binary tournament conducts $2Np$ competitions by pairing each vector in the current population and each trial vector with a randomly selected competitor. Each vector competes once in its own tournament and

possibly one or more times as a competitor in another vector's tournament. Consequently, not every vector that wins advances and not every vector that loses fails to advance. For example, a very good vector will lose if it is chosen as a competitor in the best vector's tournament. If, however, an average competitor is chosen to compete in the very good vector's tournament, then the very good vector will win. Even though it loses in competition with the best vector, the very good vector still wins its own tournament, giving it the same chance to enter the next generation as the best vector because both vectors won their tournaments and scored one win apiece. Furthermore, it is possible for more than Np vectors to win their tournaments depending on how competitors are chosen, in which case all winning vectors cannot advance. For example, although improbable, every member of the current population and every member of the trial population might randomly pick the very worst vector in the combined populations as a competitor. In that case, there would be $2Np - 1$ vectors with one win and one vector with no wins. In such cases, the best vector can be lost unless steps are taken to preserve it.

By contrast, DE's one-to-one selection holds only Np "knock-out" competitions. Any vector that loses the single competition in which it competes is eliminated and vectors that win are assured of a spot in the next generation. This form of binary, deterministic, one-to-one tournament selection in which competitors are chosen from different populations is not unique to DE. Like DE, the Particle Swarm Optimization (PSO) algorithm also conducts Np competitions that compare the trial vector with population index i to the best performing vector at population index i (Kennedy and Eberhart 1995). In DE, the best performing vector at the i^{th} position is just the i^{th} vector in the current population, i.e., the target vector $\mathbf{x}_{i,g}$. In both DE and PSO, the trial vector replaces the best-so-far vector with the same index only if it has an equal or lower objective function value.

Comparing each trial vector to the best performing vector at the same index ensures that both DE and PSO retain not only the best vector at each index, but also the very best-so-far solution at any index. Even so, a trial vector that is better than most of the current population will be rejected if its target is even better. Trial vectors that are worse than the worst vector in the current population, however, are never accepted.

2.7.4 Local Versus Global Selection

Local Selection

When an objective function is known to exhibit multiple global optima, some algorithms subdivide the selection pool into subpopulations. Each subpopulation evolves in isolation to prevent the entire population from coalescing about a single optimum. Selection is local because survivors can only replace members of the same subpopulation. For example, in the simple GA with a (μ, λ) survivor selection scheme, age determines the interacting subpopulation, or *selection neighborhood*, because only trial vectors are allowed to compete. In general, the smaller the selection neighborhood is, the lower the selection pressure will be. Just as increasing λ or T increases selection pressure, increasing the size of the population from which the base vector is drawn speeds convergence.

If DE's base and target vectors are the same, vectors evolve in isolation as though there were Np subpopulations. Selection will be local because each population vector will be compared to a mutated version of itself. Although the mutation differential is still drawn from the population at large, there is no interaction with other population members – no comparisons to solutions evolving in other parts of the solution space.

Global Selection

When seeking a single, global optimum, care must be taken to ensure that information about the best solutions can reach all members of the population. If base vectors are randomly chosen, then each vector in the current population is compared to and possibly crossed with the mutated version of another vector. Compared to local selection, global selection speeds convergence and minimizes the risk of stagnation.

2.7.5 Permutation Selection Invariance

When base vectors are the elements of a random permutation of the sequence (0, 1,…, $Np - 1$), the roles played by the base vector and survivor selection become interchangeable. If a permutation of the sequence (0, 1,…, $Np - 1$) indexes base vectors, then each vector in the current population serves as a base vector once per generation (Sect. 2.4.2). Each vector in the current population also serves as a target vector once per generation. As such, it makes no difference whether the random permutation indexes

either base vectors or target vectors. Either way, each vector in the current population vector is mutated, then matched by permutation to a vector with which it is both crossed and compared and which it potentially replaces. For example, the first expression in Eq. 2.61 shows the traditional DE approach in which permuted indices, permute[i], select the base vector, while the running index, i, points to the target vector. The second expression shows the situation reversed, in which the running index specifies the base vector and the i^{th} permutation entry locates the target vector. For clarity, Eq. 2.61 expresses this symmetry as a vector relationship ($Cr = 1$):

$$\mathbf{x}_i \text{ vs. } \mathbf{x}_{\text{permute[i]}} + F\left(\mathbf{x}_{r1} - \mathbf{x}_{r2}\right) \quad \Leftrightarrow \quad \mathbf{x}_{\text{permute[i]}} \text{ vs. } \mathbf{x}_i + F\left(\mathbf{x}_{r1} - \mathbf{x}_{r2}\right). \quad (2.61)$$

These two approaches based on the permutation selection method give identical results, i.e., optimizer performance is the same regardless of which method is employed. In both cases, each vector in the current population is mutated and then crossed with and compared to another vector in the current population not assigned to any other mutant. As such, random assignments derived from permutations can be performed either during parent (base vector) selection (left side of Eq. 2.61), or when selecting a target vector with which to cross and compete (right side of Eq. 2.61).

The "urn" permutation algorithm (see Sect. 5.2) helps illustrate the symmetry between these two selection options. For example, let urn 1 hold Np marbles, each of which is numbered with a unique vector index, $i \in [0, Np - 1]$. Urn 2 also contains Np marbles numbered 0 through $Np - 1$, but this time, numbers indicate a vector in the current population that has been mutated. Permutation selection matches vectors in the current population with mutants for crossing and competing by drawing a marble (at random and without replacement) from each urn. Once all marbles have been randomly paired this way, the final mapping between population and mutant indices will define a permutation. It does not matter to which urn the role of the permutation is assigned, just as it does not matter whether targets or mutants have their indices permuted.

2.7.6 Crossover-Dependent Selection Pressure

Because DE selection compares each trial vector to the vector in the current population with which it is crossed, replacing a vector in the current population can change the population's composition by as little as one parameter ($Cr = 0$), or by as many as D ($Cr = 1$). If, unlike DE, each vector

in the current population is compared to and replaced by a trial vector with whom it shares no common parameters, then the composition of (one member of) the vector in the current population changes by D parameters. Similarly, when $(\mu + \lambda)$-selection replaces a vector in the current population with a trial vector, the two vectors usually share no parameter values in common. By contrast, the number of parameters changed when classic DE accepts a trial vector is a function of Cr.

Figure 2.60 compares the selection pressure exerted by classic DE and two other selection schemes, both of which change D parameters each time they replace a vector in the current population. Classic DE (DE/rand/1/bin) generated the trial vectors in each case and algorithms differed only in how they selected survivors. Data points were only plotted if all 20 trials were successful. The top line shows that classic DE selection is the slowest of the three schemes when $Np = 60$, although it is the only method whose selection pressure is gentle enough to prevent premature convergence at small values of Cr. The middle line corresponds to a selection scheme that pairs vectors in the current population and trial vectors with a random permutation. This use of random permutation to pair vectors and trial vectors is distinctly different from the permutation selection method described in Sect. 2.7.5. Instead of drawing vectors from the first urn and mutants from the second, this selection method draws completed trial vectors that have already been crossed with another vector from the second urn. As such, vectors in the current population and the trial vectors that compete against them share no parameters through crossover. Its greater rate of convergence in Fig. 2.60 shows that in the case of the hyper-ellipsoid, accepting D new parameters per trial vector creates more selection pressure than does classic DE selection, except at $Cr = 1$ where both algorithms change the current population by D parameter values for each trial vector accepted. The $(\mu + \lambda)$-selection scheme (bottom line) generates the highest selection pressure because it not only changes D parameters per accepted trial vector, it also uses $T = Np$ tournaments instead of just $T = 2$.

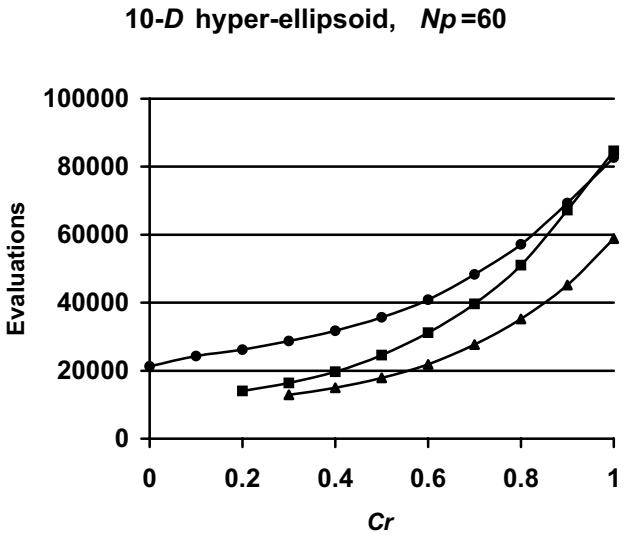

Fig. 2.60. Classic DE selection (top line) is weaker than ($\mu + \lambda$)-selection (bottom line) but both share a similar profile. Pairing target and trial vector (not mutant) adversaries with a random permutation provides an intermediate level of selection pressure (middle line). DE/rand/1/bin (Np = 60, F = 0.9) generated trial vectors, but survivors were selected by the indicated selection method. Data points are 20-trial averages.

2.7.7 Parallel Performance

Not all survivor selection methods are equally well suited to parallel implementations. For example, ($\mu + \lambda$)-selection is time consuming when implemented as tournament selection without replacement. If instead, ($\mu + \lambda$)-selection is done by sorting, it becomes difficult to implement efficiently in parallel because some comparisons must be performed before others. DE, however, is ideally suited for parallel computing, primarily because each vector in the current population competes in a single tournament against a trial vector that belongs to an intermediate population. Section 5.1 describes several schemes for distributing DE across a network of processors. In addition, Sect. 7.6 describes how DE was implemented in

parallel to perform image registration. In that application, performance scaled quasi-linearly.

2.7.8 Extensions

The presence of constraint functions and multiple objectives in an optimization task make it difficult to compare solutions based on a single objective function value. For this reason, J. Lampinen (2002) has extended DE's selection criteria so that solutions can be compared based on the notion of Pareto-dominance (Sect. 4.6). Instead of replacing a vector in the current population with a trial vector whose objective function value is equal or lower, Lampinen's method replaces a vector in the current population when the trial vector dominates it. Lampinen's method is easy to apply to problems with multiple constraints (Sect. 4.3), those with multiple objectives (Sect 4.6) and multi-objective problems with multiple constraints (Sect 4.6). Among its principal advantages are that objectives and constraints do not need to be weighted. Details on Lampinen's method can be found in the sections indicated above.

In summary, DE's one-to-one selection offers numerous advantages beyond its simplicity. It does not require mapping objective function values to selection probabilities. It is elitist, easy to implement in parallel, compensates for increased acceptance rates at low *Cr* and has all the traditional advantages of tournament selection's versatility which include invariance to objective function transposition. DE selection is also flexible, allowing either target or base indices to be randomly specified by permutations, or the criterion of "less than or equal" to be replaced by "Pareto-dominant" when problems have multiple objective and/or multiple constraints.

2.8 Termination Criteria

Sometimes it is obvious when an optimization should be halted. For example, in constraint satisfaction problems (Sects. 4.3 and 4.5) the optimization is over when all constraints are satisfied, i.e., when a feasible vector is found. In multi-objective optimization (Sect. 4.6), however, objectives often conflict. Satisfying one objective leaves another unfulfilled, so it is not always clear when to stop searching for a better compromise. This section briefly describes some halting criteria and the scenarios in which they are appropriate.

2.8.1 Objective Met

In some optimization tasks, the objective function's minimum value is already known. For example, the goal when designing telescope optics is to reduce the geometric spot size of a star's image to a point. The wave nature of light, however, renders meaningless any improvement beyond certain well-known limits. Consequently, an optical system optimization can be halted when spot sizes fall below the limits set by the wave nature of light. The same is true of other error functions for which the tolerable error is given. This is also the method used when working with test functions whose minima are known. If the best-so-far vector's objective function value is within a specified tolerance of the global minimum, the optimization halts.

2.8.2 Limit the Number of Generations

Usually, the objective function minimum is not known in advance. Even for many test functions, only the best-known results are reported. In these cases, optimizations can be terminated after g_{max} generations. When testing optimizers with functions whose optima *are* known, setting g_{max} may halt optimizations that do not reach the objective function minimum within the specified tolerance. Finding a value of g_{max} that is large enough to give an optimizer enough time to find the optimum, but not so long that a second trial would be a better way to invest computer time, involves some guess work.

Alternatively, an optimization can be halted when Δg_{max} generations have passed without a trial vector being accepted. Again, some experimentation may be needed to find a good value for Δg_{max}. Long periods without improvement are perhaps more common in DE than other EAs, so it is important that Δg_{max} not be set too low.

2.8.3 Population Statistics

An optimization can also be terminated when a population statistic reaches a pre-specified value. For example, an optimization can be halted when the difference between the population's worst and best objective function values falls below some predetermined limit. This method needs to be applied with caution because it can cause an optimization to halt prematurely. For example, if the optimization is halted when the difference between the population's worst and best objective function values is less than, e.g.,

1.0×10^{-6}, the population's best objective function value might not yet be within 1.0×10^{-6} of the minimum value. Thus, the interruption is premature because DE may still be making progress even though the range of objective function values is small. When using this criterion, it is usually a good idea to make the difference between the population's worst and best objective function values several orders of magnitude lower than the tolerance set for locating the optimum. The same advice applies when monitoring the standard deviation of population vectors or the longest vector difference as termination criteria.

2.8.4 Limited Time

Sometimes only a limited amount of time is available for an optimization. In such cases, the optimization must terminate regardless of the state of the population or the number of generations. For example, in on-line optimization, only a small amount of time may available to adjust manufacturing process parameters (e.g., Sect. 7.12). Similarly, it may be that computer time is limited or simply that a deadline must be met. Monitoring and manual intervention can help determine whether the available time is best spent completing an ongoing optimization or running a new trial.

2.8.5 Human Monitoring

Because of the inherent uncertainties in knowing when an optimization is over, it usually helps to personally monitor time-consuming optimization tasks. The feedback from the best objective function value, number of trial vectors accepted per generation, the distribution of the population, etc., usually makes it clear when no more improvement is possible or when time might be better spent running a new trial. In addition, human monitoring allows the optimization to be altered in response to perceived opportunities.

2.8.6 Application Specific

Finally some applications will have their own termination criterion. In evolutionary art, for example, an optimization to find the most pleasing picture might end when interest in the exhibit wanes, or when a certain group of people have participated.

References

Ali MM, Törn A (2000) Optimization of carbon and silicon cluster geometry for Tersoff potential using differential evolution. In: Floudas CA, Pardalos PM (eds) Optimisation in computational chemistry and molecular biology. Kluwer Academic, Dordrecht, pp 1–15

Bäck T (1993) Optimal mutation rates in genetic search. In: Forrest F (ed) Proceedings of the fifth international conference on genetic algorithms. Morgan-Kaufmann, San Mateo, CA, pp 2–8

Bäck T (1996) Evolutionary algorithms in theory and in practice. Oxford University Press

Bäck T, Schwefel H-P (1993) An overview of evolutionary algorithms for parameter optimization. Evolutionary Computation 1(1):1–23

Bäck T, Schwefel H-P (1995) Evolution strategies I: variants and their computational implementation. In: Periaux J, Winter G (eds) Genetic algorithms in engineering and computer science. Wiley, Chichester, Chap. 6

Bäck T, Rudolph G, Schwefel H-P (1993) Evolutionary programming and evolution strategies: similarities and differences. In: Fogel DG, Atmar W (eds) Second annual conference on evolutionary programming, February. Evolutionary Programming Society, La Jolla, CA, pp 11–22

Baker JE (1987) Reducing bias and inefficiency in the selection algorithm. In: Grenfenstette JJ (ed) Proceedings of the first international conference on genetic algorithms and their applications. Lawrence Erlbaum, Hillsdale, NJ, pp 14–21

Beckman FS (1980) Mathematical foundations of programming. Addison-Wesley, Reading, MA

Beyer HG (1999) On the dynamics of EAs without selection. In: Banzaf W, Reeves C (eds) Foundations of genetic algorithms. Morgan Kaufmann, San Mateo, CA, pp 5–26

Blahut RE (1984) Fast algorithms for digital signal processing. Addison-Wesley, Reading, MA, p 329

Caruana RA, Eshelman LJ, Schaffer JD (1989) Representation and hidden bias II: eliminating defining length bias in genetic search via shuffle crossover. In: Sidharan NS (ed) Eleventh international joint conference on artificial intelligence, Morgan Kaufmann, San Mateo, CA, vol 1, pp 750–755

Chakraborti N, Misra K, Bhatt P, Barman N, Prasad R (2001) Tight-binding calculations of Si–H clusters using genetic algorithms and related techniques: studies using differential evolution. Journal of Phase Equilibria 22(5):525–530

Eiben AE, Smith JE (2003) Introduction to evolutionary computing. Springer, Berlin Heidelberg New York

Eshelman LJ, Caruana RA, Schaffer JD (1989) Biases in the crossover landscape. In: Schaffer JD (ed) Proceedings of the third international conference on genetic algorithms. Morgan Kaufmann, San Francisco, pp 10–19

Fogel DB (1991) System identification through simulated evolution: a machine learning approach to modeling. Ginn Press, Needham Heights, MA

Fogel LJ, Owens AJ, Walsh MJ (1966) Artificial intelligence through simulated evolution. Wiley, New York

Gamperle G, Mueller SD, Koumoutsakos P (2002) A parameter study for differential evolution. In: Grmela A, Mastorakis NE (eds) Advances in intelligent systems, fuzzy systems, evolutionary computation. WSEAS Press, Athens, pp 293–298

Goldberg DE (1989) Genetic algorithms in search, optimization and machine learning. Addison-Wesley, Reading, MA

Halton J, Weller G (1964) Algorithm 247: radical inverse quasi-random point sequence. Communications of the ACM, 7(12):701–702

Holland, JH (1973) Genetic algorithms and the optimal allocation of trials. SIAM Journal of Computing 2:88–105

Holland J (1992) Adaptation in natural and artificial systems. MIT Press, Cambridge, MA. First edition 1975, The University of Michigan Press, Ann Arbor

Kennedy J, Eberhart RC (1995) Particle swarm optimization. In: Proceedings of the international conference on evolutionary computation, Perth, Australia. Invited paper

Koza J (1992) Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge, MA

Lampinen J (2002) Multi-constrained nonlinear optimization by the differential evolution algorithm. In: Rajkumar Roy, Mario Köppen, Seppo Ovaska, Takeshi Furuhashi, Frank Hoffmann (eds) Soft computing and industry: recent advances. Springer, Berlin Heidelberg New York, pp 305–318 (Proceedings of the 6th online world conference on soft computing in industrial applications (WSC6), September 10–24, 2001. Available at: http://vision.fhg.de/wsc6)

Lampinen J, Zelinka I (2000) On stagnation of (the) differential evolution algorithm. In: Ošmera P (ed) Proceedings of MENDEL 2000, sixth international Mendel conference on soft computing, June 7–9, Brno, Czech Republic. Brno University of Technology, Faculty of Mechanical Engineering, Institute of Automation and Computer Science, Brno, pp 76–83. Available at: http://www.lut.fi/~jlampine/MEND2000.ps

Macready WG, Wolpert DH (1998) Bandit problems and the exploration/exploitation tradeoff. IEEE Transactions on Evolutionary Computing 2(1):2–22

Michalewicz Z (1996) Genetic algorithms + data structures = evolution programs, 3rd edn. Springer, Berlin Heidelberg New York

Mühlenbein H (1992) How genetic algorithms really work I: mutation and hill climbing. In: Schwefel H-P, Männer R (eds) Proceedings of the second international conference on parallel problem solving from nature, Springer, Berlin Heidelberg New York, pp 15–26

Mühlenbein H, Schlierkamp-Voosen D (1993) Predictive models for the breeder genetic algorithm. Evolutionary Computation 1(1):25–50

Peitgen H-O, Saupe D (eds) (1998) The science of fractal images. Springer, Berlin Heidelberg New York

Potter MA, DeJong KA (1994) A cooperative co-evolutionary approach to function optimization. In: Davidor Y, Schwefel H-P, Männer R (eds) Proceedings of parallel problems solving from nature 3. Springer, Berlin Heidelberg New York, pp 249–257

Price KV (1996) Differential evolution: a fast and simple numerical optimizer. In: Smith MH, Lee MA, Keller J, Yen J (eds) Proceedings of the 1996 biennial conference of the North American fuzzy information processing society – NAFIPS, June 19–22, Berkeley, CA, USA. IEEE Press, New York, pp 524–527

Price KV (1997) Differential evolution vs. the functions of the second ICEO. In: Proceedings of the 1997 IEEE international conference on evolutionary computation, Indianapolis, Indiana, USA. IEEE Press, New York, pp 153–157

Rudolph G (1996) Convergence of evolutionary algorithms in general search spaces. In: Proceedings of the third IEEE conference on evolutionary computation, IEEE Press, New York, pp 50–54

Saravanan N, Fogel DB (1997) Multi-operator evolutionary programming: a preliminary study on function optimization. In: Angeline PJ, Reynolds RG, McDonnell JR, Eberhart R (eds) Evolutionary programming 6: sixth international conference, Indianapolis, Indiana, USA, April. Springer, Berlin Heidelberg New York, pp 215–221

Salomon R (1996a) Re-evaluating genetic algorithm performance under coordinate rotation of benchmark functions: a survey of some theoretical and practical aspects of genetic algorithms. Biosystems 39(3):263–278

Salomon R (1996b) The influence of different coding schemes on the computational complexity of genetic algorithms in function optimization. In: Voigt H-M, Ebeling E, Rechenberg I, Schwefel H-P (eds) Proceedings of the fourth international conference on parallel problem solving from nature, Springer, Berlin Heidelberg New York, pp 227–235

Salomon R (1997) Raising theoretical questions about the utility of genetic algorithms. In: Proceedings of the sixth international conference on evolutionary programming. Lecture notes in computer science, vol 1213. Springer, Berlin Heidelberg New York, pp 275–284

Spears WM, DeJong KA (1991) An analysis of multi-point crossover. In: Rawlins G (ed) Foundations of genetic algorithms. Morgan Kaufmann, San Francisco, pp. 301–315

Storn R (1996) On the usage of differential evolution for function optimization. In: Smith MH, Lee MA, Keller J, Yen J (eds) Proceedings of the 1996 biennial conference of the North American fuzzy information processing society – NAFIPS, June 19–22, Berkeley, CA, USA. IEEE Press, New York, pp 519–523

Storn R, Price KV (1996) Minimizing the real functions of the ICEC'96 contest by differential evolution. In: Proceedings of the 1996 IEEE international conference on evolutionary computation, Nagoya, Japan. IEEE Press, New York, pp 842–844

Storn R, Price KV (1997) Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. Journal of Global Optimization 11:341–359

Syswerda G (1989) Uniform crossover in genetic algorithms. In: Schaffer JD (ed) Proceedings of the third international conference on genetic algorithms. Morgan Kaufmann, San Francisco, pp 2–9

Wright AH (1991) Genetic algorithms for real parameter optimization. In: Rawlins GJE (ed) Foundations of genetic algorithms. Morgan-Kaufmann, San Mateo, CA, pp 205–218

Yang J-M, Chen Y-P, Horng J-T, Kao C-M, Chen Y-P, Horng J-T, Kao C-Y (1997) Applying family competition to evolution strategies for constrained optimization. In: Angeline PJ, Reynolds RG, McDonnell JR, Eberhart R (eds) Evolutionary programming 6: sixth international conference, Indianapolis, Indiana, USA, April. Springer, Berlin Heidelberg New York, pp 201–211

Yao X, Liu Y (1997) Fast evolution strategies. In: Angeline PJ, Reynolds RG, McDonnell JR and Eberhart R (eds) Proceedings of the sixth international conference on evolutionary programming, Springer, Berlin Heidelberg New York, pp 151–161

Zaharie D (2002) Critical values for the control parameters of differential evolution algorithms. In: Matoušek R, Ošmera P (eds) Proceedings of MENDEL 2002, 8th international conference on soft computing, June 5–7, 2002, Brno, Czech Republic. Brno University of Technology, Faculty of Mechanical Engineering, Institute of Automation and Computer Science, Brno, pp 62–67

Zimmons P (n.d.) Polynomial fitting with differential evolution. Available at: http://www.cs.unc.edu/~zimmons/cs258/poly.html