Differential Evolution

Dr. Dobb's Journal April 1997

A simple evolution strategy for fast optimization

By Kenneth Price and Rainer Storn

Ken holds a B.S. in physics from Rensselaer Polytechnic Institute. He is currently engaged in research on artificial intelligence and system modeling. Ken can be reached at kprice@solano.community.net. Rainer received his Ph.D. from the University of Stuttgart. Currently, he is a postdoctoral fellow at the International Computer Science Institute in Berkeley, California. He can be contacted at storn@icsi.berkeley.edu or rainer.storn@zfe.siemens.de.

Since their inception three decades ago, genetic algorithms have evolved like the species they try to mimic. Just as competition drives each species to adapt to a particular environmental niche, so, too, has the pressure to find efficient solutions across the spectrum of real-world problems forced genetic algorithms to diversify and specialize. A genetic algorithm that is well adapted to solving a combinatorial task like the traveling salesman problem may fail miserably when used to minimize functions with real variables and many local minima. For the task of optimizing functions of real variables, Evolution Strategies (ES) have emerged as one of the fittest contenders for the title "King of the Jungle." Although they have been around in one form or another for about 30 years, these genetic algorithms for real numbers are still unknown to many investigators. Not only are ES significantly faster at numerical optimization than traditional genetic algorithms, they are also much more likely to find a function's true global optimum.

Until now, the price to be paid for an efficient numerical optimizer has been mathematical complexity. Differential Evolution (DE), however, is an exceptionally simple ES that promises to make fast and robust numerical optimization accessible to everyone. Remarkably, DE's main search engine can be easily written in less than 20 lines of C code and involves nothing more exotic than a uniform random-number generator and a few floating-point arithmetic operations.

Numerical Optimization

Nearly every engineering effort seeks to maximize a design's beneficial aspects while minimizing the impact of its undesirable features. As such, many scientific and engineering tasks can be formulated as optimization problems in which the minimum or maximum of an objective function with real parameters is sought. When the objective function is nonlinear and contains many real variables, the surface it defines often resembles a tortured, multidimensional landscape with many peaks and valleys. If the objective function is to be minimized, the goal of the optimization process will be to find the coordinates of the lowest point in the lowest valley.

Suppose, for example, the minimum value assumed by a spherical surface is sought. Example 1(a) illustrates this for a sphere in three dimensions. Thus, each potential solution consists of a vector with three real-valued components: $\underline{X}=\{X_1,X_2,X_3\}$. The problem is to find a vector, $\underline{X}_{min}$, at which the function attains a minimum. Obviously, the 3-D sphere has a single minimum $f(\underline{X}_{min})=0$, where $\underline{X}_{min}=\{0,0,0\}$. While a local minimum like that of a sphere can easily be found by any method of steepest descent, finding the global minimum of a function with many local minima is fraught with pitfalls.

One way to optimize these "multimodal" functions is with nonrandom numerical techniques. The simplex method, the method of Nelder and Mead, and the Levenberg-Marquardt algorithm are just a few of the most frequently used deterministic search algorithms. While these methods are effective on many problems, they seldom perform well on difficult optimizations. For large problems, random search or stochastic algorithms are often the only viable strategy. Randomization gives a search algorithm the ability to break the curse of dimensionality that makes nonrandom and exhaustive-search methods exponentially tedious on functions with many parameters. In exchange for their efficient search strategies, stochastic algorithms sacrifice the guarantee that their best solution is a true global optimum. Nevertheless, a stochastic algorithm's best result will almost certainly be better than one obtained with a nonrandom method when the problem at hand is nonlinear and has many variables.

Elements of a Genetic Algorithm

Genetic algorithms interpret the value of a function at a point as a measure of that point's fitness as an optimum. In maximization problems, "fitness" aptly describes a vector's suitability, since vectors with large objective function values are indeed the most fit. "Cost," however, is a better way to describe the objective function in minimization problems, since minimizing cost is more familiar than minimizing fitness. Maximization problems are easily transformed into minimization problems and vice versa, so we'll deal with one type of problem -- minimizing functions. Consequently, the objective function will henceforth be referred to as the "cost function." Guided by cost, genetic algorithms attempt to transform an initial population of randomly generated vectors into a solution vector through repeated cycles of mutation, recombination, and selection.

Mutation is an operation that makes small random alterations to one or more parameters of an existing population vector. In genetic algorithms that represent parameters as integer bit strings, mutation usually involves flipping a few randomly chosen bits. Mutation is crucial for maintaining diversity in a population, although excessive mutation, like that resulting from too much radiation, has the potential for erasing evolutionary progress. Even so, early attempts at numerical optimization with ES relied on mutation exclusively. Evolution by mutation alone is not without parallel in nature. For example, whiptail lizard populations are known to propagate by parthenogenesis, or self-cloning. Just as nature discovered the survival value of sex, researchers in ES realized that a faster and more robust algorithm results when a knowledge-sharing operator like recombination is used in conjunction with mutation.

Recombination (or crossover) provides an alternative and complementary means of creating viable vectors. Designed to resemble the natural process by which a child inherits DNA from its parents, recombination builds new parameter combinations from the components of existing vectors. A host of different recombination methods have been proposed, but each combines parameter values from two or more parents in its own characteristic way. One of the most frequently used schemes for numerical optimization is uniform crossover. In this form of recombination, the child inherits parameter values from its parents with equal probability. DE, however, uses a nonuniform crossover that can take child vector parameters from one parent more often than it does from the other. By using components of existing population members to construct trial vectors, recombination efficiently shuffles information about successful combinations, enabling the search for an optimum to focus on the most promising areas of the solution space.

Once new trial solutions have been generated, selection determines which among them will survive into the next generation. Most ES selection schemes are deterministic, meaning that only the fittest candidates from the combined parent and offspring populations are allowed to advance. Other selection mechanisms take a randomized approach. For example, tournament selection pits randomly paired vectors in a winner-takes-all competition in which the victor moves on and the vanquished is eliminated. DE's acceptance strategy resembles tournament selection except that each child is pitted against one of its parents, not against a randomly chosen competitor. Only the fitter of the two is then allowed to advance into the next generation.

Integer versus Floating Point

The first step in implementing any genetic algorithm is to decide how functional parameters should be encoded as genes. While many genetic algorithms have used integers to approximate continuous parameters, this choice limits the resolution with which an optimum can be located to the precision set by the number of bits in the integer. If the problem at hand exhibits a wide dynamic range, then very long integers must be used even though many bits probably won't be significant in the final answer. Instead of wasting bits on unused precision, most ES encode continuous parameters as floating-point numbers. Floating point not only uses computer resources efficiently, it also makes input and output transparent for the user. Parameters can be input, manipulated, and output as ordinary floating-point numbers without ever being reformatted as genes with a different binary representation.

XOR versus Add

Just as floating-point numbers are more appropriate than integers for representing points in continuous space, addition is more appropriate than random bit flipping for searching the continuum. Consider, for example, the consequences of using the logical exclusive OR (XOR) operator for mutating integers. To change a binary 16 (10000) into a binary 15 (01111) with an XOR operation requires inverting all five bits. In most bit-flipping schemes, a mutation of this magnitude would be rare even though transitions between adjacent integers ought to be among the most frequent. Consequently, disparities between the binary representations of adjacent numbers can be problematic when trying to conduct an incremental search. An early attempt to circumvent this adjacency problem was the development of Gray codes, named for Frank Gray, who patented the idea for use in shaft encoders. Gray's idea was to assign each integer an alternative binary representation so that adjacent integers could be transformed into one another with a single bit flip. Although using Gray codes often helps, gains are not dramatic.

A simpler way to restore the adjacency of neighboring points is to use addition as a mutation operator. Under addition, 16 becomes 15 simply by adding -1. Now the odds of a given transition depend only on the arithmetic difference between the numbers involved and not the bit patterns that are used to encode them. Adopting addition as a mutation operator to restore the adjacency of nearby points is not, however, a panacea. Once the switch from logical to arithmetic operators is made, the fundamental question concerning mutation is no longer which bits to flip, but rather how much to add. The simple adaptive scheme used by DE ensures that these mutation increments are automatically scaled to the correct magnitude.

DE at a Glance

The overall structure of the DE algorithm resembles that of most other population-based searches. The parallel version of DE maintains two arrays, each of which holds a population of *NP*, D-dimensional, real-valued vectors. The primary array holds the current vector population while the secondary array accumulates vectors that are selected for the next generation.

In each generation, *NP* competitions are held to determine the composition of the next generation. In particular, the *i*th competition pits the *i*th population vector, known as the "target," against its adversary, the trial vector. Not only does the target vector compete against the trial vector, it is also one of the trial vector's parents. The trial vector's other parent is a randomly chosen population vector to which a weighted random difference vector has been added. Mating between this noisy random vector and the target vector is controlled by a nonuniform crossover operation that determines which trial vector parameters are inherited from which parent. Should the cost of the trial vector turn out to be less than or equal to that of its parent target vector, the trial vector replaces the target as the *i*th population vector in the next generation. Figure 1 illustrates this procedure. In all, just three factors control evolution under DE: the population size, *NP*; the weight applied to the random differential, *F*; and the constant that mediates the crossover operation, *CR*.

Initialization

Parameters in real-world problems generally exhibit restricted ranges over which it is sensible to search for a solution. Before DE can set about optimizing a function, these parameter limits should be established. In general, limits ought to be chosen so that they constrain parameters to physically attainable values. In optical design, for example, there are natural limits on how thick a lens can be, how steep curvatures can become, how closely components can be spaced, and so on. If, however, parameters do not correspond to physical quantities, then parameter ranges should be large enough to contain the suspected optimum. Unless instructed not to, DE can search beyond its specified limits, so even if the optimum is not contained within the initial bounding region, DE may still find it.

Once limits have been set, each parameter in every primary array vector is initialized with a uniformly distributed random value from within its allowed range. To determine and preserve the cost of the resulting initial population, each primary array vector is evaluated and the results are stored in the array, *cost[]*.

Mutating with Vector Differentials

Early Evolutionary Strategies mutated vectors by adding zero-mean Gaussian noise to them. One consequence of this scheme is that the standard deviation of the Gaussian distribution determines the average mutation step size. When this step size is comparable to the standard deviation of the actual distribution of parameters in the population, mutation efficiency improves significantly. Unfortunately, the distributions of different parameters often exhibit different standard deviations because of disparities in scale and sensitivity. To remain efficient, an ES mutation scheme must adjust the standard deviation of the Gaussian noise to suit each parameter.

Not only is the most effective mutation step size a function of the parameter to which it is being applied -- it is also a function of time. Consequently, any source of mutating noise must also adapt to a vector population's evolving shape in solution space. Although measures can be taken to accommodate this mutation step size variability, the effort can be substantial.

A more convenient source of appropriately scaled perturbations is the population itself. Every pair of vectors, $(\underline{X}_a,\underline{X}_b)$, defines a vector differential: $\underline{X}_a-\underline{X}_b$. When $\underline{X}_a$ and $\underline{X}_b$ are chosen randomly, their weighted difference can be used in place of Gaussian noise to perturb another vector, $\underline{X}_c$. This process, which can be expressed mathematically as

$$\underline{X}_c'=\underline{X}_c+F\cdot(\underline{X}_a-\underline{X}_b)$$

is illustrated graphically in Figure 2. The scaling factor, F, is a user-supplied constant in the range: $(0<F\le 1.2)$. The upper limit of 1.2 was determined empirically. The optimal value of F for most functions lies in the range of 0.4 to 1.0.

By mutating vectors with population-derived noise, DE ensures that the solution space will be efficiently searched in each dimension. For example, if the population becomes compact in one dimension but remains widely dispersed along another, the differentials sampled from it will be small in one dimension yet large in the other. Whatever the case, step sizes will be comparable to the breadth of the region they are being used to search, even as these regions expand and contract during the course of evolution.

Recombination

Once in every generation, each primary array vector, $\underline{X}_i$ for i=0,1, is targeted for recombination with a vector like $\underline{X}_c'$ to produce a trial vector, $\underline{X}_t$. Thus, the trial vector is the child of two parents: a noisy random vector and the target vector against which it must compete.

Which parent contributes which trial vector parameter is determined by a series of D-1 binomial experiments. Each experiment, whose outcome is either success or failure, is mediated by the crossover constant, CR, where $0\le CR\le 1$. Starting at a randomly selected parameter, the source of each trial vector parameter is determined by comparing CR to a uniformly distributed random number from within the interval [0,1). If the random number is greater than CR, the trial vector gets its parameter from the target, $\underline{X}_i$, otherwise, the parameter comes from the noisy random vector, $\underline{X}_c'$ (Figure 3). When CR=1, for example, every trial vector parameter is certain to come from $\underline{X}_c'$, making the trial vector an exact replica of the noisy random vector. If, on the other hand, CR=0, all but one trial vector parameter comes from the target vector. To ensure that X_t differs from $\underline{X}_i$ by at least one parameter, the final trial vector parameter always comes from the noisy random vector, even when CR=0.

Selection

Unlike many genetic algorithms, DE does not use proportional selection, ranking, or even an annealing criterion that would allow occasional uphill moves. Instead, the cost of each trial vector is compared to that of its parent target vector. The vector with the lower cost is rewarded by being allowed to advance to the secondary array. In addition, if the trial vector wins, its cost is stored in *cost[i].*

After each primary array vector has been a target for mutation, recombination, and selection, array pointers are swapped so that the roles of the two arrays are reversed. Thus, vectors in what was the secondary array become targets for transformation, while the former primary array now awaits winners of the next generation's competitions.

DE Pseudocode

Differential Evolution is one of those rare algorithms that possesses both simplicity and power. Nowhere is DE's simplicity better illustrated than in the 19 lines of C-style pseudocode in Listing One This accounting includes code for mutation, recombination, and selection, but excludes input/output routines, function descriptions, and initialization code. (The complete source code for both a C and a Matlab implementation of DE is available electronically; see "Availability," page 3, and at http://www.ICSI.Berkeley.edu/~storn/code.html.)

While algorithmic simplicity may be a virtue, without performance, it is irrelevant. The following example, while elementary, nevertheless demonstrates several powerful features of DE.

A Nasty Little Function

What makes a numerical optimization problem hard? Many dimensions, multiple local minima, flat surfaces, and nondifferentiability to name a few reasons. Although it is only two dimensional, the following problem generates a fitness landscape that is nondifferentiable at some points and has many local minima that are partially surrounded by a very flat surface. Given the function in Example 1(b), the problem is to find values for p_1 and p_2 such that *f(x)* will be constrained to the light-colored area of Figure 4.

Problems of this ilk occur in robotics where the goal is to steer a vehicle with the output of a simple function. Alternatively, *f(x)* could represent a filter function with a response that must conform to certain design constraints. Either way, the lack of an obvious analytical solution makes this problem an ideal candidate for DE's evolutionary approach.

Potential solutions to this problem are two-dimensional vectors with components that are the real-valued parameters, p_1 and p_2. A vector's quality can be evaluated by sampling the function it describes at evenly spaced points along the x-axis as indicated in Figure 4. Whenever the function strays outside of design limits, the deviations can be squared and then summed to provide a measure of that function's cost. Other measures, like the maximum deviation, will also work, but the sum of squared errors is more reliable because it provides information about the function's acceptability at more than one point. The landscape generated by this problem's cost function, $g(p_1,p_2)$ is shown in Figure 5.

Despite this problem's treacherous fitness landscape, DE is up to the challenge. With settings of *NP*=10, *CR*=0.9, *F*=0.7, DE located this function's minima with an error of 1.0E-6 after a 100-trial average of only 314 function evaluations. Because it does not require resolution constraints, DE can continue to refine a solution until it exhausts the precision of its floating-point format.

Practical Advice

Choosing *NP*, *F*, and *CR* is seldom difficult and some general guidelines can be given. Ordinarily, *NP* ought to be about five to ten times the number of parameters in a vector. For the two-dimensional example mentioned, this rule suggests that *NP*=10 was a good starting point (*NP*=5**D*=10). As for *F*, try *F*=0.5 initially and then increase *F* and/or *NP* if the population converges prematurely. Values of *F* smaller than 0.4, like those greater than 1, are only occasionally effective.

A good first choice for *CR* is 0.1, but in general *CR* should be as large as possible without causing the population to become devoid of diversity. Consequently, you may first want to try *CR*=0.9 or *CR*=1.0 to see if a quick solution is possible before trying *CR*=0.1 or 0. In the end, considerable latitude is often available when choosing effective control constant combinations.

DE and DeJong

Although it is no longer considered to be a demanding test suite for numerical optimizers, the ubiquitous five-function DeJong test suite (Figure 6) nevertheless provides some insight into DE's abilities. Table 1 summarizes DE's performance as the average number of function evaluations needed to find each function's global minimum. Results are 100-trial averages. In each case, convergence to the global optimum was fast and regular.

A much more difficult set of test functions was the focus of a recent international competition for evolutionary optimizers in which DE was a participant. Among conference entries, DE proved to be the fastest evolutionary algorithm, although it did place third in speed behind two deterministic methods of limited application.

Conclusion

Differential Evolution is a design tool of great utility that is immediately accessible for practical applications. Among DE's advantages are its simple structure, ease of use, speed, and robustness.

Already, DE has been used to design several complex digital filters, the largest of which required tuning 60 parameters. Power like this enables nonspecialists to discover effective solutions to nearly intractable problems without appealing to expert knowledge or complex design algorithms. All one needs is a problem and a way to quantify the quality of its potential solutions. In some cases, this may be a formidable task, but if a system is amenable to being rationally evaluated, DE can provide the means for extracting the best possible performance from it.

DDJ

Listing One

```
while (count < gen_max)          /* Halt after gen_max generations. */{
   for (i=0; i<NP; i++)          /* Start loop through population. */
   {
                              /********** Mutate/recombine **********/
      do a=rnd_uni()*NP; while (a==i); /* Randomly pick 3 vectors, */
      do b=rnd_uni()*NP; while (b==i || b==a);    /* all different */
      do c=rnd_uni()*NP; while (c==i || c==a || c==b);  /* from i. */
      j=rnd_uni()*D;          /* Randomly pick the first parameter. */
      for (k=1; k<=D; k++)       /* Load D parameters into trial[]. */
      {                             /* Perform D-1 binomial trials. */
         if (rnd_uni() < CR || k==D)      /* Source for trial[j] is */
         {          /* a random vector plus weighted differential */
            trial[j]=x1[c][j]+F*(x1[a][j]-x1[b][j]);      /* or... */
         }          /* trial parameter comes from x1[i][j] itself. */
         else trial[j]=x1[i][j];
         j=(j+1)%D;                /* get next parameter, modulo D. */
      }    /* Last parameter (k=D) comes from noisy random vector. */
                               /********** Evaluate/select **********/
      score=evaluate(trial); /* Evaluate trial with your function. */
      if (score<=cost[i])        /* If trial[] improves on x1[i][], */
      {                          /* move trial[] to secondary array */
         for (j=0; j<D; j++) x2[i][j]=trial[j];
         cost[i]=score;                  /* and store improved cost */
      }             /* otherwise, move x1[i][] to secondary array. */
      else for (j=0; j<D; j++) x2[i][j]=x1[i][j];
   }                /* Mutate/recombine next primary array vector. */


          /********** End of population loop; swap arrays **********/
   for (i=0; i<NP; i++)                  /*  After each generation, */
   {                   /* move secondary array into primary array. */
      for (j=0; j<D; j++) x1[i][j]=x2[i][j];
   }                      /* ...or just swap pointers (not shown). */
   count++;              /* End of generation...increment counter. */
                            /********** End of generation **********/
}                                                /* End of program. */
```

Back to Article