4

Optimization by Evolution

Organic life, we are told, has developed gradually from the protozoon to the philosopher, and this development, we are assured, is indubitably an advance. Unfortunately it is the philosopher, not the protozoon, who gives us this assurance.
***Bertrand Russell,* Mysticism & Logic**

Computer scientists often encounter problems that are not amenable to numerical methods of solution. A common problem is that of finding input value that produces a minimum or maximum output from a function. It's quite easy to optimize a function that maps a single high or low value; things become quite a bit more difficult when a function generates several such values, as shown in Figure 4-1.

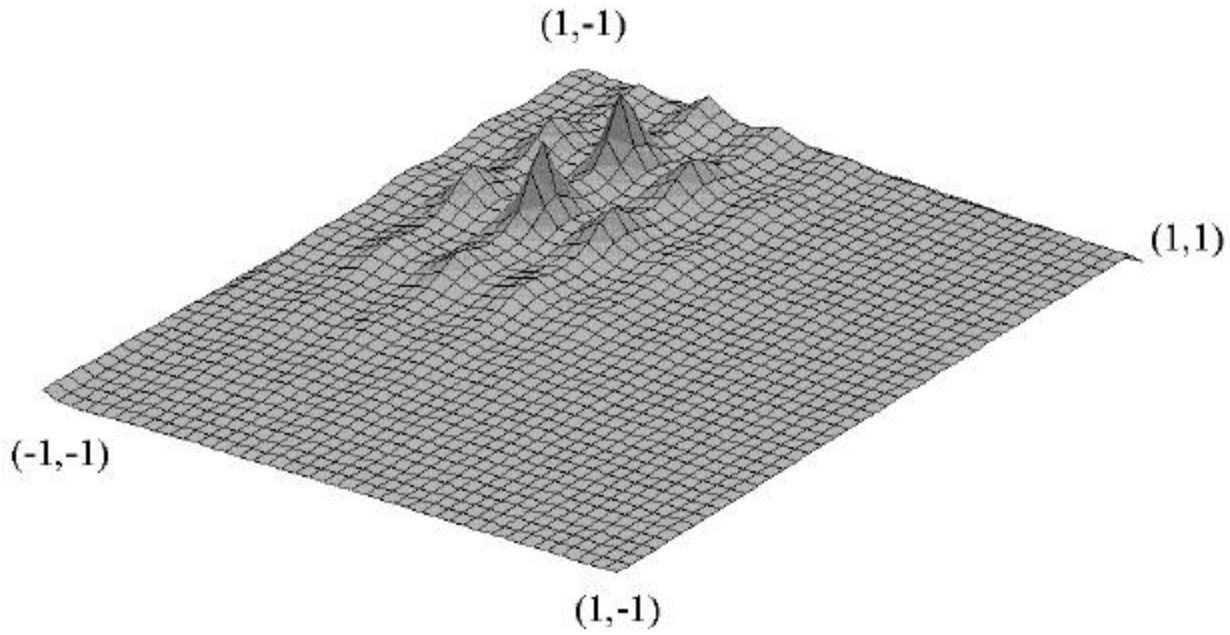

Figure 4-1 Function with several maximums

The graph in Figure 4-1 was generated by applying the following formula to the ranges -1 <= x <= 1 and -1 <= y <= 1:

$$f(x,y) = \frac{1}{(x+0.5)^2 + 2(y-0.5)^2 - 0.3\cos(3\pi x) - 0.4\cos(4\pi y) + 0.8}$$

How should a program go about finding the maximum for $f(x,y)$? The graph shows several local maxima in the upper quadrant, including two tall spikes in close proximity. Traditional approaches to finding function maxima or minima—a process known as optimization—use a variety of techniques that rely on the ability to climb "upward" to a solution. In optimizing $f(x,y)$, most optimization techniques (hill climbing, for example) become "trapped" on the smaller "hills"—unless they begin looking for maxima in just the right place.

A genetic algorithm begins with a set of randomly-selected points from which it selects the best performers through fitness testing. Crossover combines the best attributes from the most successful members of the population, and random mutation introduces new characteristics that may produce better solutions. As I'll demonstrate, a genetic algorithm is particularly effective in finding optimal solutions to functions.

Characteristics of a Genetic Algorithm

The population size is the most important factor affecting the run-time of a genetic algorithm. More chromosomes mean more time spent in fitness calculation—a fitness calculation is the most time-consuming component of most genetic algorithms. And while a large population size provides more chromosomes for testing, it also dilutes the fitness of the best chromosomes. If we have ten chromosomes with fitness of 10, 8, 5, 1, 1, 1, 1, 1, 1 and 1, the three most-fit chromosomes have a combined 77% chance of reproducing; if another ten chromosomes of fitness 1 were included in the population, the reproduction probability for the three best drops to 57%. A balance must be struck between diversity (large populations) and relative fitness (small populations.)

Numerical Precision

In the real world of science, few calculations involve values of incredible precision; in fact, Heisenberg's Uncertainty Principles guarantees that we'll never know everything "exactly." Scientists work with the concept of significant digits—the number of valid non-zero decimal digits in scientific notation. The accuracy of a result is only as good as the accuracy of the operands used in the calculation.

For example, if you ask C++ to perform these calculations:

```
double a = 1.5;
double b = 2.01;
double c = a * b;
```

The compiler will dutifully assign **c** the result of multiplying 1.5 by 2.01, which is 3.015. However, the accuracy of that result must be compared against the accuracy of the numbers used to calculate it. In this case, while **b** has four decimal digits, we only know the accuracy of *a* to two decimal places. Unless we know that *ib* is *exactly* equal to 1.5, we cannot assume that *c* is *exactly* equal to 3.015. It's a case of being sure of our results. If we aren't certain of **a**'s value beyond the second digit, then we cannot be certain beyond the second decimal place about the value of any result which is calculated using **a**.

It's common in science to know a value to a specific number of decimal places, without certainty that the value is exact. So long as there is doubt about the absolute accuracy of a value, all calculations involving that number are limited to the number of digits that we know are absolutely correct. In the example above, the correct value for `c` is 3.0, since the least accurate of

operand used in its calculation (a) had only two digits of accuracy. The compiler, of course, knows nothing about significant digits. It blithely goes about its business of assigning 3.015 to *c*. Now, what happens later in the program when another calculation involving c is made?

```
double d = c * 250.0; // assume 250.0 is exact
```

d is assigned the value 3.015 * 250, or 753.75. The error continues to mount! The correct result should be 750, since **c** is only known reliably to 2 digits of accuracy (making it 3.0). The problem only grows worse as calculations continue.

To solve this problem, I extended the rounding features of C++ to set a specific number of significant digits in a value. Here's the double versions of the **sigdig** function, and a support function named **round_nearest** that rounds a number at a specific number of significant digits.

```
double Coyote::round_nearest(double x)
{
    double i, f, dummy;

    f = fabs(modf(x,&i));

    if (f == 0.0)
        return i;

    if (f == 0.5)
    {
        if (modf(i / 2.0, &dummy) != 0.0)
        {
            if (x < 0.0)
                i -= 1.0;
            else
                i += 1.0;
        }
    }
    else
    {
        if (f > 0.5)
        {
            if (x < 0.0)
                i -= 1.0;
            else
                i += 1.0;
        }
    }

    return i;
}

//--------------------------------------------------
// set number of significant digits in a value

double Coyote::sigdig(double x,size_t n)
{
    double s, result;

    // a very small number rounds to zero
    if (fabs(x) < 1.0E-300)
        result = 0.0;
    else
    {
        // is asking for no digits, or more didgits than in double
        // simply return x
```

```
            if ((n == 0U) || (n > DBL_DIG))
                result = x;
            else
            {
                // find a factor of ten such that all significant digits will
                // be in the integer part of the double
                s = pow(10.0,double((int)n - 1 - (int)floor(log10(fabs(x)))));

                // scale up, round, and scale down
                result = round_nearest(x * s) / s;
            }
        }

        return result;
    }
```

The setting of significant digits in the dialog box will cause numbers to be rounded and truncated at the specified decimal position by **sigdig**. It also affects the output precision of floating-point numbers. The default of 8 digits is usually more than adequate, for the simple reason that most C++ numerical functions have limited precision.

The inexact nature of IEEE floating-point numbers combines with the limitations of C++ functions to produce inexact results. For example, in practical terms, 1.23456E-76 is such a small number that it generates the same result from the **sin** function as does zero. Since very tiny numbers and zero produce the same result, their fitness as chromosomes is the same, thus preventing a genetic algorithm from distinguishing the two values.

Perhaps it would make sense to convert very tiny number to zero automatically—but it might very well be that the function actually *does* produce a peak at a very tiny number and not at zero. In general, a 64-bit IEEE **double** can only be used to analyze a search grid with a precision of about **DBL_EPSILON**, and any number smaller than **DBL_EPSILON** can be considered zero.

The range settings set constraints on the *x* and *y* values being analyzed. Set these ranges to bracket the search area; if you want to find a local maxima instead of a global one, set these values to limit the search to your area of interest.

Optimizing the Equation

The namespace OptByEvol encapsulates types and functions used in optimizing the equation in Figure 4-1.

```
namespace OptByEvol
{
    enum ScalingMode
    {
        stNONE,
        stEXPONENTIAL,
        stWINDOW,
        stLINEAR
    };

    enum FunctionType
    {
        ftF6,
        ftF7,
        ftF8,
        ftChapter4
    };
```

```
    void testOptByEvol(size_t       populationSize,
                       size_t       numGenerations,
                       double       crossoverRate,
                       double       mutationRate,
                       bool         elitismEnabled,
                       ScalingMode  fitnessScaling,
                       FunctionType function);

} // end namespace OptByEvol
```

The testOptByEvol function has several parameters that define the operation of the algorithm:

- **populationSize** declares the number of chromosomes in the population
- **numGenerations** is a limit on the number of "cycles" the simulation will run
- **crossoverRate** is the percentage chance (0.0 to 1.0) of crossover when a new chromosome is bred
- **mutationRate** is another percentage defining the chance of mutation
- **elitismEnabled** determines whether the "best" chromosome is saved from one generation to the next
- **fitnessScaling** turns fitness scaling on or off
- **function** selects one of four different equations to be optimized; **ftChapter4** is the equation in Figure 4-1

Another Type of Fitness Scaling

When fitness scaling is on, several other factors come into play. The three most common types of fitness scaling are: *windowed*, *exponential*, and *linear normalization*. The first two techniques were covered in Chapter Two; linear normalization is something new. Fitness scaling emphasizes the reproductive chances of the most-fit chromosomes in a population; linear normalization accomplishes this by changing fitness values to reflect a gradation of values. For example, here is a set fitness values for five chromosomes, as calculated directly by the fitness function:

```
1: 0.255
2: 0.773
3: 0.405
4: 0.928
5: 0.318
```

While chromosome 4 is obviously the most fit, its reproductive chance relative to the entire population is only 35 percent. Assuming a base value of 20, a decrement of 8, and a minimum value of 1, linear normalization would assign new fitness values as shown in Table 4-1. For comparison, I've also included the fitness values as scaled by windowing and exponentiation.

Chromosome	***Original Fitness***	***Scaled by Windowing***	***Scaled by Exponential***	***Scaled by Linear Norm***
1	0.255 (10%)	0.000 (0%)	1.575 (13%)	1.00 (3%)
2	0.773 (29%)	0.518 (37%)	3.144 (26%)	12.00 (31%)

3	0.405 (15%)	0.150 (11%)	1.974 (16%)	4.00 (10%)
4	0.928 (35%)	0.673 (48%)	3.717 (31%)	20.00 (53%)
5	0.318 (11%)	0.063 (4%)	1.737 (14%)	1.00 (3%)

Table 4-1 Fitness Scaling Example

Windowing increases the reproductive chances of the strongest chromosomes, but eliminates the least-fit chromosome from producing offspring. Exponential fitness scaling (adding one to the original fitness and squaring the result) increases the reproductive capability of less-fit chromosomes, while maintaining the superiority of the best. Linear normalization puts a premium on success, enhancing the reproductive chances of the best chromosomes while still maintain a possibility the less-fit chromosomes might produce offspring.

Why not have the most-fit chromosome produce all members of the new population? Because the most-fit chromosomes may not have all the components necessary for reaching an optimal solution. As I discussed in Chapter Two, crossover mixes the most fit parts of different chromosomes; it may be that a few bits of a less-fit chromosome may be essential to creating the optimum fitness. And in some cases, we don't want to eliminate the least-fit chromosomes since they may just have an essential piece of the final solution.

What I call exponential fitness scaling is my own—or, at least, I haven't seen it discussed in the genetic algorithm literature. At first glance, the exponential scaling might seem counterproductive in that it "evens out" the reproductive chances within a population. In testing, I've found the exponential method to work quite well, particularly in populations where the fitness is heavily biased toward specific values. Or, to put it another way: When the landscape of a function includes steep approaches to maxima, some values will have dramatically higher fitnesses than their neighbors; exponential scaling can often prevent getting stuck in a sub-optimal peak by allowing apparently unfit chromosomes a chance at reproducing.

Picking an Equation

The program supports the optimization of four different functions, three classic and one of my own invention. The first three functions come from the work of I.O. Bohachevsky, M.E. Johnson, and M.L. Stein in a 1986 paper that analyzes techniques for function optimization. The functions were selected because they generate a "wavy" landscape that contain various configurations of local maxima. Properly, these three functions are known in the genetic algorithm literature as F6, F7, and F8. Functions one through five, developed by K. A. DeJong in 1975, lack several local maxima, and are thus less useful than six through eight in testing the effectiveness of optimization algorithms.

Even these three functions suffer from significant problems. To begin with, all three produce an bowl-shaped plot in which the minimum value is located at $x = 0$, $y = 0$. I'm never thrilled with test examples with such easy answers; and, since the functions produce a minimum instead of a maximum, fitness values need to be adjusted by "flipping" the plot upside down, so that the minimum becomes a peak. Otherwise, the most fit chromosomes will produce the smallest fitness value—zero.

I played around a bit, and eventually came up with the equation the produces the plot shown in Figure 4-1. My equation is quite tricky to optimize, since it has a broad plain of low maxima

punctuated by some very strong peaks. Essentially, I modified the equation F6 to offset its maximum away from the origin to a point that is not easily predicted by induction or guesswork.

Implementation

Now for the meat, the implementation of the **testOptByEvol** function. And here it is:

```
static const double PI = 3.141592653589793238;

template <class T> inline T sqr(const T & n) { return n * n; }

static double fitnessF6(double x,double y)
{
    return  0.7 + sqr(x)
          + 2.0 * sqr(y)
          - 0.3 * cos(3.0 * PI * x)
          - 0.4 * cos(4.0 * PI * y);
}

static double fitnessF7(double x,double y)
{
    return  0.3 + sqr(x)
          + 2.0 * sqr(y)
          - 0.3 * (cos(3.0 * PI * x) * cos(4.0 * PI * y));
}

static double fitnessF8(double x,double y)
{
    return  0.3 + sqr(x)
          + 2.0 * sqr(y)
          - 0.3 * (cos(3.0 * PI * x)
          + cos(4.0 * PI * y));
}

static double fitnessC4(double x,double y)
{
    return 1.0 / (0.8 + sqr(x + 0.5)
         + 2.0 * sqr(y - 0.5)
         - 0.3 * cos(3.0 * PI * x)
         - 0.4 * cos(4.0 * PI * y));
}

void OptByEvol::testOptByEvol(size_t populationSize,
                              size_t numGenerations,
                              double crossoverRate,
                              double mutationRate,
                              bool   elitismEnabled,
                              ScalingMode  fitnessMode,
                              FunctionType function)
{
    cout << "Function Optimization (Peak Search)" << endl
         << "-----------------------------------" << endl << endl;

    // adjust any invalid parameters
    if (populationSize < 10)
        populationSize = 10;

    if (numGenerations < 1)
        numGenerations = 1;

    if (crossoverRate < 0.0F)
        crossoverRate = 0.0F;
```

```
else
    if (crossoverRate > 1.0F)
        crossoverRate = 1.0;

if (mutationRate < 0.0F)
    mutationRate = 0.0F;
else
    if (mutationRate > 1.0F)
        mutationRate = 1.0;

// display parameters for this run
cout << "   Equation: ";

switch (function)
{
case ftF6:
    cout << "f6(x,y) = x²+2y²-0.3cos(3px)-0.4cos(4py)+0.7";
    break;
case ftF7:
    cout << "f7(x,y) = x²+2y²-0.3[cos(3px)cos(4py)]+0.3";
    break;
case ftF8:
    cout << "f8(x,y) = x²+2y²-0.3[cos(3px)+cos(4py)]+0.3";
    break;
case ftChapter4:
    cout << "f(x,y) = 1/((x+0.5)²+2(y-0.5)²-0.3cos(3px)-0.4cos(4py)+0.8)";
}

// display parameters for this run
cout << endl
     << "  population size: " << populationSize << endl
     << " # of generations: " << numGenerations << endl
     << "   crossover rate: " << crossoverRate * 100.0F << "%" << endl
     << "    mutation rate: " << mutationRate * 100.0F << "%" << endl
     << "  elitism enabled: " << elitismEnabled << endl
     << "fitness algorithm: ";

switch(fitnessMode)
{
case stNONE:
    cout << "None";
    break;
case stEXPONENTIAL:
    cout << "Exponential";
    break;
case stWINDOW:
    cout << "Windowing";
    break;
case stLINEAR:
    cout << "Linear Normalization";
    break;
}

// create random deviate and mutation objects
Random<double> randNum;
FloatBreeder mutator;

// constants to define ranges for fitness scaling
const double FIT_BASE = 100.0;
const double FIT_MIN  =  10.0;
const double FIT_DEC  =  10.0;

// ranges for X,Y grid
const double MIN_XY   = -10.0;
```

```
const double MAX_XY   =  10.0;
const double RANGE_XY = MAX_XY - MIN_XY;

// other constants
const size_t SIG_DIGITS =   8;

cout << endl << endl << setprecision(SIG_DIGITS) << dec;

// allocate population and fitness arrays
double * x       = new double [populationSize];
double * xnew    = new double [populationSize];
double * y       = new double [populationSize];
double * ynew    = new double [populationSize];
double * fitness = new double [populationSize];

double * ptrf = fitness - 1;
double * ptrx =  x  - 1;
double * ptry =  y  - 1;

// various variables
double bestFitness, lowFitness, fitn, vf, vx, vy;
size_t i, j, inc, genCounter, bestIndex, parent1, parent2;

// generate initial X & Y values
for (i = 0; i < populationSize; ++i)
{
    x[i] = sigdig(RANGE_XY * randNum() + MIN_XY,8);
    y[i] = sigdig(RANGE_XY * randNum() + MIN_XY,8);
}

// do the generations
for (genCounter = 0; genCounter < numGenerations; ++genCounter)
{
    // calculate fitness for x values
    bestFitness = DBL_MIN;
    lowFitness  = DBL_MAX;
    bestIndex   = 0;

    for (i = 0; i < populationSize; ++i)
    {
        switch (function)
        {
        case ftF6:
            fitness[i] = 1.0 - fitnessF6(x[i],y[i]);
            break;
        case ftF7:
            fitness[i] = 1.0 - fitnessF7(x[i],y[i]);
            break;
        case ftF8:
            fitness[i] = 1.0 - fitnessF8(x[i],y[i]);
            break;
        case ftChapter4:
            fitness[i] = fitnessC4(x[i],y[i]);
        }

        fitness[i] = sigdig(fitness[i],SIG_DIGITS);

        // track bestFitness fitness
        if (fitness[i] > bestFitness)
        {
            bestFitness  = fitness[i];
            bestIndex = i;
        }
```

```
            // track lowest fitness
            if (fitness[i] < lowFitness)
                lowFitness = fitness[i];
        }

        // display bestFitness solution so far
        if ((genCounter % 10) == 0)
        {
            cout.setf(ios::internal | ios::showpoint);

            cout << setw(6) << genCounter << setfill('0')
                 << ": (" << setw(SIG_DIGITS+1) << x[bestIndex] << "," <<
setw(SIG_DIGITS+1) << y[bestIndex]
                 << ") fitness = " << setw(SIG_DIGITS+1) << bestFitness << endl
<< setfill(' ') ;

            cout.unsetf(ios::internal);
        }

        // sort by fitness if linear normalization
        if (stLINEAR == fitnessMode)
        {
            // shell sort three arrays in order of fitness
            fitn = FIT_BASE;

            for (inc = 1; inc <= populationSize / 9; inc = 3 * inc + 1) ;

            for ( ; inc > 0; inc /= 3)
            {
                for (i = inc + 1; i <= populationSize; i += inc)
                {
                    vf = ptrf[i];
                    vx = ptrx[i];
                    vy = ptry[i];

                    j  = i;

                    while ((j > inc) && (ptrf[j - inc] < vf))
                    {
                        ptrf[j] = ptrf[j - inc];
                        ptrx[j] = ptrx[j - inc];
                        ptry[j] = ptry[j - inc];

                        j -= inc;
                    }

                    ptrf[j] = vf;
                    ptrx[j] = vx;
                    ptry[j] = vy;
                }
            }
        }

        if (fitnessMode != stNONE)
        {
            for (i = 0; i < populationSize; ++i)
            {
                // fitness scaling
                switch (fitnessMode)
                {
                case stEXPONENTIAL:
                    fitness[i] = sqr(fitness[i] + 1.0);
                    break;
                case stWINDOW:
```

```
            fitness[i] -= lowFitness;
            break;
        case stLINEAR:
            fitness[i] = fitn;

            if (fitn > FIT_MIN)
            {
                fitn -= FIT_DEC;

                if (fitn < FIT_MIN)
                    fitn = FIT_MIN;
            }
            break;
        }
    }
}

// create roulette wheel for reproduction selection
RouletteWheel<double> * selector;
selector = new RouletteWheel<double> (fitness,populationSize);

// if elitist, include bestFitness from orig. population
if (elitismEnabled)
{
    if (stLINEAR == fitnessMode)
    {
        xnew[0] = x[0];
        ynew[0] = y[0];
    }
    else
    {
        xnew[0] = x[bestIndex];
        ynew[0] = y[bestIndex];
    }

    i = 1;
}
else
    i = 0;

// create new population of x's
for ( ; i < populationSize; ++i)
{
    // create a new x
    parent1 = selector->get_index();

    if (randNum() <= crossoverRate)
    {
        parent2 = selector->get_index();
        xnew[i] = mutator.crossover(x[parent1],x[parent2]);
    }
    else
        xnew[i] = x[parent1];

    // create a new y
    parent1 = selector->get_index();

    if (randNum() <= crossoverRate)
    {
        parent2 = selector->get_index();
        ynew[i] = mutator.crossover(y[parent1],y[parent2]);
    }
    else
        ynew[i] = y[parent1];
```

```
                // mutate X
                if (randNum() <= mutationRate)
                    xnew[i] = mutator.mutate(xnew[i]);

                // mutate Y
                if (randNum() <= mutationRate)
                    ynew[i] = mutator.mutate(ynew[i]);

                // make sure x & y fitness ranges
                if (xnew[i] > MAX_XY)
                    xnew[i] = MAX_XY;

                if (xnew[i] < MIN_XY)
                    xnew[i] = MIN_XY;

                if (ynew[i] > MAX_XY)
                    ynew[i] = MAX_XY;

                if (ynew[i] < MIN_XY)
                    ynew[i] = MIN_XY;

                // truncate digits
                xnew[i] = sigdig(xnew[i],SIG_DIGITS);
                ynew[i] = sigdig(ynew[i],SIG_DIGITS);
            }

            // remove roulette wheel
            delete selector;

            // copy new population
            memcpy(x,xnew,populationSize * sizeof(double));
            memcpy(y,ynew,populationSize * sizeof(double));
        }

        // delete buffers
        delete [] fitness;
        delete [] ynew;
        delete [] y;
        delete [] xnew;
        delete [] x;
    }
```

The algorithm begins by creating a configuration object and displaying the parameters. Next, I allocate buffers to hold fitness values and populations of x and y chromosomes. Initial populations contains random values distributed between specified minimum and maximum values.

A loop, containing the main algorithm, then counts the generations. The first part of processing a generation is to calculate fitness values for the population, calling the selected fitness function for each x-y pair in the population. The loop also tracks the highest and lowest fitness values for reporting and later fitness scaling. To implement linear normalization, a shell sort orders the chromosomes by their calculated fitness, before the algorithm assigns new values from top to bottom.

Producing a new population requires the selection of parents, who are combined by crossover and then mutated based on the chosen configuration. After the generation loop ends, the algorithm deletes dynamically-allocated buffers and displays its output.

What You'll See

The output of **testOptByEvol** will look like this (I've manually inserted ellipses for repetitive cycles, so you just see when the fitness changed):

```
Function Optimization (Peak Search)
-----------------------------------

   Equation: f(x,y) = 1/((x+0.5)²+2(y-0.5)²-0.3cos(3px)-0.4cos(4py)+0.8)
  population size: 1000
 # of generations: 1000
   crossover rate: 90.000000%
    mutation rate: 10.000000%
  elitism enabled: 1
fitness algorithm: Windowing

     0: (-0.53235396,0.40350782) fitness = 1.6967843
    10: (-0.65735396,0.49727308) fitness = 7.9261430
    20: (-0.65735396,0.49921094) fitness = 7.9405427
    30: (-0.65625914,0.49921091) fitness = 7.9440813
    40: (-0.65637741,0.49945533) fitness = 7.9444938
    50: (-0.65625917,0.49945508) fitness = 7.9447716
    60: (-0.65637738,0.49969922) fitness = 7.9449309
    70: (-0.65625535,0.49969946) fitness = 7.9452182
    80: (-0.65625532,0.49970707) fitness = 7.9452278
   .
   .
   .
   110: (-0.65625496,0.49970682) fitness = 7.9452283
   120: (-0.65625103,0.49970700) fitness = 7.9452374
   130: (-0.65625137,0.49994349) fitness = 7.9454118
   140: (-0.65625101,0.49997383) fitness = 7.9454180
   150: (-0.65625101,0.49997788) fitness = 7.9454184
   .
   .
   .
   200: (-0.65527589,0.49991281) fitness = 7.9467265
   .
   .
   .
   230: (-0.65527599,0.49992253) fitness = 7.9467298
   240: (-0.65527592,0.49997976) fitness = 7.9467417
   250: (-0.65527365,0.49997205) fitness = 7.9467420
   .
   .
   .
   280: (-0.65500604,0.49995912) fitness = 7.9468029
   290: (-0.65497993,0.49997338) fitness = 7.9468042
   .
   .
   .
   310: (-0.65497312,0.49998268) fitness = 7.9468047
   .
   .
   .
   330: (-0.65497503,0.49998268) fitness = 7.9468048
   .
   .
   .
   350: (-0.65500412,0.49998278) fitness = 7.9468058
   .
   .
   .
```

```
   390: (-0.65499792,0.49999136) fitness = 7.9468062
   .
   .
   .
   420: (-0.65499812,0.49999706) fitness = 7.9468064
   .
   .
   .
   950: (-0.65501338,0.49999690) fitness = 7.9468065
   .
   .
   .
```

For the functions F6 through F8, **testOptByEvol** performs remarkably well, finding the approximate answer in fewer than 10 generations.

```
Function Optimization (Peak Search)
-----------------------------------

   Equation: f6(x,y) = x²+2y²-0.3cos(3px)-0.4cos(4py)+0.7
  population size: 1000
 # of generations: 1000
   crossover rate: 90%
    mutation rate: 10%
  elitism enabled: 1
fitness algorithm: Windowing

     0: (-0.68040694,9.2169483) fitness = 2.2250739e-308
    10: (7.7943307e-010,6.0263372e-154) fitness = 1.0000000
   .
   .
   .


 Function Optimization (Peak Search)
-----------------------------------

  Equation: f7(x,y) = x²+2y²-0.3[cos(3px)cos(4py)]+0.3
  population size: 1000
 # of generations: 1000
   crossover rate: 90.000000%
    mutation rate: 10.000000%
  elitism enabled: 1
fitness algorithm: Windowing

     0: (0.21172395,-0.051639709) fitness = 0.55136173
    10: (-1.8036076e-038,3.5417319e-019) fitness = 1.0000000
   .
   .
   .

Function Optimization (Peak Search)
-----------------------------------

   Equation: f8(x,y) = x²+2y²-0.3[cos(3px)+cos(4py)]+0.3
  population size: 1000
 # of generations: 1000
   crossover rate: 90.000000%
    mutation rate: 10.000000%
  elitism enabled: 1
fitness algorithm: Windowing

     0: (0.35563143,-0.45448293) fitness = 0.11926576
```

```
 10: (3.7216025e-019,2.7317749e-019) fitness = 1.3000000
.
.
.
```

For the simpler functions, the genetic algorithm quickly zeros in on the optimal value. My custom function presents a more difficult problem, as evidenced by the GA's slower performance. The highest possible fitness value, to fifteen decimal places, is 7.94680648572638, which is generated by x and y values close to x = -0.655 and y = 0.5. If you're looking for six to eight digits of precision, the peak is usually found in a few hundred generations. Being a stochastic process, the genetic algorithm doesn't always produce identical performance. Play a bit with the configuration, selecting a variety of parameters to gain a feel for how they affect performance. Reducing the influence of chance will be one of the topics in Chapter Six, along with further analysis of genetic algorithm performance and parameter selection.

Onward

Genetic algorithms can do more than optimize a function. The next chapter shows how to use genetic algorithms for finding optimum strategies—and I'll describe how scientists use evolutionary programming to explore mysterious aspects of the universe.