Chapter 4

Think Globally, Act Locally

Stan Wagon

In order to find all common points, which are the solutions of our nonlinear equations, we will (in general) have to do neither more nor less than map out the full zero contours of both functions. Note further that the zero contours will (in general) consist of an unknown number of disjoint closed curves. How can we ever hope to know when we have found all such disjoint pieces?
— W. H. Press et al. in "Numerical Recipes" [PTVF92]

Problem 4

What is the global minimum of the function

$$e^{\sin(50x)} + \sin(60e^y) + \sin(70\sin x) + \sin(\sin(80y)) - \sin(10(x+y)) + (x^2+y^2)/4 \ ?$$

4.1 A First Look

Let $f(x, y)$ denote the given function. On a global scale, f is dominated by the quadratic term $\left(x^2 + y^2\right)/4$, since the values of the other five summands lie in the intervals $[1/e, e]$, $[-1, 1]$, $[-1, 1]$, $[-\sin 1, \sin 1]$, and $[-1, 1]$, respectively. Thus the overall graph of f resembles a familiar paraboloid (Figure 4.1). This indicates that the minimum is near $(0, 0)$. But a closer look shows the complexity introduced by the trigonometric and exponential functions. Indeed, as we shall see later, there are 2720 critical points inside the square $[-1, 1] \times [-1, 1]$. From this first look, we see that a solution to the problem breaks into three steps:

1. Find a bounded region that contains the minimum.
2. Identify the rough location of the lowest point in that region.

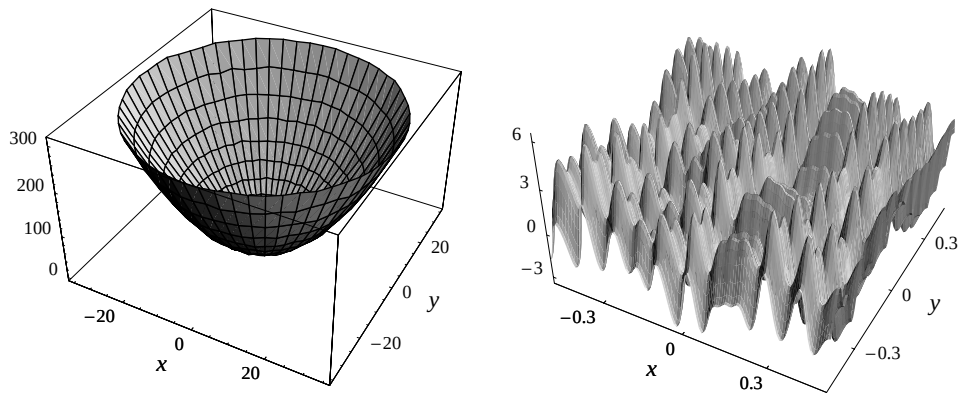

Figure 4.1. *Two views of the graph of $f(x, y)$. The scale of the left-hand view masks the complexity introduced by trigonometric and exponential terms. It is this complexity that makes finding the global minimum a challenge.*

3. Zoom in closer to pinpoint the minimum to high precision.

Step 1 is easy. A quick computation using a grid of modest size yields the information that the function can be as small as -3.24. For example, one can look at the 2601 values obtained by letting x and y range from -0.25 to 0.25 in steps of 0.01.

A Mathematica session. For convenience later we define `f` so that it accepts as inputs either the two numeric arguments or a single list of length two.

```
f[x_, y_] := e^Sin[50 x] + Sin[60 e^y] + Sin[70 Sin[x]] + Sin[Sin[80 y]] -
    Sin[10 (x + y)] + (x^2 + y^2)/4;
f[{x_, y_}] := f[x, y];

Min[Table[f[x, y], {x, -0.25, 0.25, 0.01}, {y, -0.25, 0.25, 0.01}]]
-3.246455170851875
```

This upper bound on the minimum tells us that the global minimum must lie inside the circle of radius 1 centered at the origin, since outside that circle the quadratic and exponential terms are at least $1/e+1/4$ and the four sines are at least $-3-\sin 1$, for a total above -3.23. And step 3 is easy once one is near the minimum: standard optimization algorithms, or root-finding algorithms on the gradient of f, can be used to locate the minimum very precisely. Once we are close, there is no problem getting several hundred digits. The main problem is in step 2: How can be pinpoint a region that contains the correct critical point, and is small enough that

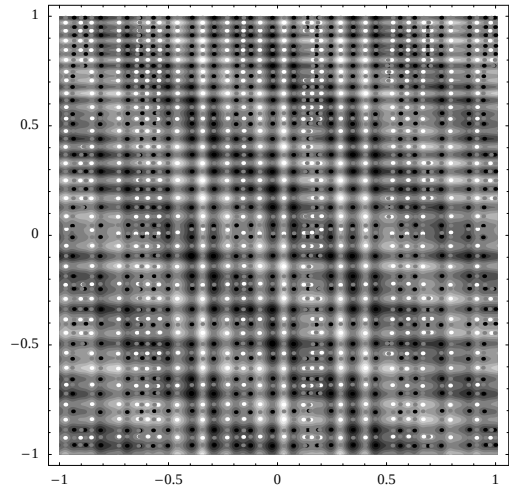

Figure 4.2. *A density plot of $f(x, y)$ with increasing values going from black to white. The local minima are shown as black dots (693), the local maxima as white dots (667), and the saddles (1360) as gray dots. There are 2720 critical points in this square, computed by the method of §4.4.*

it contains no other? Figure 4.2, a contour plot of $f(x, y)$ with the contour lines suppressed, shows what we are up against.

In fact, a slightly finer grid search will succeed in locating the proper minimum; several teams used such a search together with estimates based on the partial derivatives of f to show that the search was fine enough to guarantee capture of the answer. But we will not discuss such methods here, focussing instead on general algorithms that do not require an analysis of the objective function.

4.2 Speedy Evolution

While a grid search can be effective for this particular problem (because the critical points do have a rough lattice structure), it is easy to design a more general search procedure that uses randomness to achieve good coverage of the domain. An effective way to organize the search is to use ideas related to evolutionary algorithms. For each point in the current generation, n random points are introduced, and the n best results of each generation (and its parents) are used to form the new generation. The scale that governs the generation of new points shrinks as the algorithm proceeds.

Algorithm 4.1. Evolutionary Search to Minimize a Function.

Inputs: $f(x, y)$, the objective function;
R, the search rectangle;
n, the number of children for each parent, and the number of points in the

new generation;
ϵ, a bound on the absolute error in the location of the minimum of f in R;
s, a scaling factor for shrinking the search domain.

Output: An upper bound to the minimum of f in R, and an approximation to its location.

Notation: `parents` = current generation of sample points, `fvals` = f-values for current generation.

Step 1: Initialize: Let z be the center of R; `parents` $= \{z\}$; `fvals` $= \{f(z)\}$; $\{h_1, h_2\}$ = side-lengths of R.

Step 2: The main loop:
While $\min(h_1, h_2) > \epsilon$,
For each $p \in$ `parents`, let its children consist of n random points in a rectangle around p; get these points by using uniform random x and y chosen from $[-h_1, h_1]$ and $[-h_2, h_2]$, respectively;
Let `newfvals` be the f-values on the set of all children;
Form `fvals` $\cup$ `newfvals`, and use the n lowest values to determine the points from the children and the previous parents that will survive;
Let `parents` be this set of n points; let `fvals` be the set of corresponding f-values;
Let $h_i = s \cdot h_i$ for $i = 1, 2$.

Step 3: Return the smallest value in `fvals`, and the corresponding parent.

This algorithm is nicely simple and can be programmed in just a few lines in a traditional numerical computing environment. It generalizes with no problem to functions from $\mathbb{R}^n$ to $\mathbb{R}$. The tolerance is set to 10^{-6} in the *Mathematica* code that follows because the typical quadratic behavior at the minimum means that should give about 12 digits of precision of the function value. Note that it is possible for some children to live outside the given rectangle, and therefore the final answer might be outside the rectangle. If that is an issue, then one can add a line to restrict the children to be inside the initial rectangle.

A Mathematica session. Some experimentation will be needed to find an appropriate value of n. We use $n = 50$ here, though this algorithm will generally get the correct digits even with n as low as 30.

```
h = 1; gen = {f[#], #}&/@{{0, 0}};

While[h > 10^-6,
  new = Flatten[Table[#[[2]] + Table[h(2 Random[] - 1), {2}], {50}]&/@gen,
      1]; gen = Take[Sort[Join[gen, {f[#], #}&/@new]], 50];
  h = h/2];
```

```
gen[[1]]
{-3.30686864747396, {-0.02440308163632728, 0.2106124431628402}}
```

Figures 4.3 and 4.4 use a logarithmic scale to show the steady convergence of the points in each generation to the answer. Figure 4.4 shows how the points swarm into the correct one from all directions.

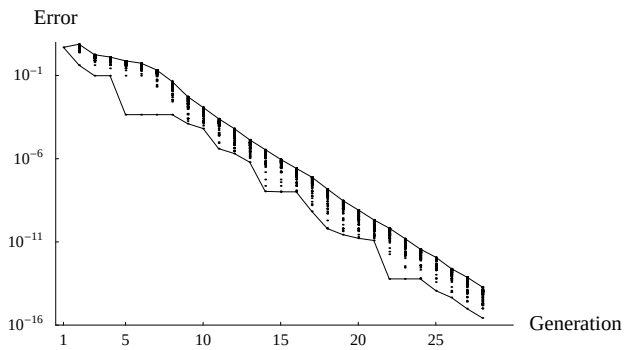

Figure 4.3. *The base-10 logarithm of the f-value error for each point in each generation of a genetic search, in a run with $n = 50$ and tolerance of 10^{-8} in the x-y coordinates. The convergence to the correct result is nicely steady.*

For more confidence one would compute a sequence of values as n increases. When $n = 70$, the algorithm solves the problem with high probability (992 successes in 1000 runs). This approach has the virtue of being easy to program and very fast, and so long as the resolution is fine enough to find the correct local minimum, it will converge to the answer even if one requests many digits (and uses appropriate working precision). Of course, it comes with no guarantee, and repeated trials will usually be necessary to gain confidence in the correctness of the result (the next section will present an algorithm that eliminates the uncertainty inherent in a random search). Searches such as this have limitations — a move to higher dimensions puts stress on the number of search points; a microscopic downward spike would be impossible to find — but such issues are problematic for just about any optimization method. Given the speed and minimal programming effort, this genetic approach is a fine tool for an initial investigation into an optimization problem.

An obvious way to speed things up is to switch at some point to a method that is designed to quickly converge to the nearest local minimum. It can be hard to decide, in an automated fashion, when one is close enough to the right minimum. However, for the problem at hand, switching to either a minimization routine (Brent's method is popular) or, better, a root-finder that finds the zero of the gradient speeds things up a lot if one wants a large number of digits. Once we have a seed that we believe in, whether from a random search or a more reliable interval algorithm as in §4.3, using Newton's method on the gradient is very fast and yields 10 000 digits in under a minute (see Appendix B).

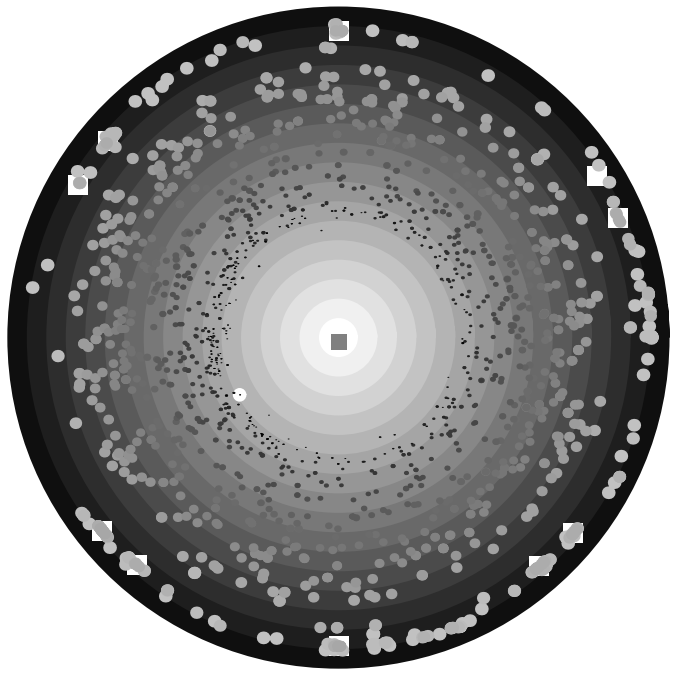

Figure 4.4. *Survival of the fittest: the results of a genetic algorithm with a high working precision, and accuracy goal of* 10^{-13}*; the computation runs through 45 generations with 50 members of each. The scale is logarithmic, with each gray ring representing another power of 10 away from the true minimum location (center). Points of the same hue correspond to the same generation, so the innermost red points are the last generation (28th). They are within* 10^{-6} *of the true best; the point with the lowest* f*-value (which is not the point closest to the answer!) is the one atop a black disc. The outermost points — the first generation — have* f*-values between* -1.3 *and* 4.2*. But by the 9th generation all points are in the vicinity of the correct critical point. The yellow discs mark the location of the 20 lowest local minima; the clustering near those points at the early generations is visible.*

If one wants to use a random search, there are some canned algorithms that can be used. For example, the `NMinimize` function in *Mathematica* can solve a variety of optimization problems, including problems with constraints and in any number of variables. But there are several methods available for it, and several options for each, so, as with all complicated software, it can be tricky finding a combination that works. Simulated annealing and the Nelder–Mead method are available, but they do not seem very good on this sort of continuous problem (they tend to end up at a nonglobal minimum). An evolutionary algorithm called *differential evolution* does work well (for more on this method see §5.2), and will reliably (85 successes in 100 random trials) solve the problem at hand when called in the following form.

A Mathematica session.

```
NMinimize[{f[x, y], x^2 + y^2 ≤ 1}, {x, y},
  Method → {"DifferentialEvolution", "SearchPoints" → 250}]

{-3.3.3068686474752402,
  {x → -0.024403079743737212, y → 0.21061242727591697}}
```

This gives 14 digits of the answer, and cause the objective function to be evaluated only about 6,000 times (the simpler evolutionary algorithm presented earlier can get 15 digits with about 120,000 function evaluations).

Another approach is to use a more comprehensive global optimization package. One such, based on random search and statistically based reasoning, is MathOptimizer, a commercial package for Mathematica developed by Janos Pintér.[1]

4.3 Interval Arithmetic

Let R be the square $[-1,1] \times [-1,1]$, which we know must contain the answer. Random search methods can end up at a local but nonglobal minimum; the best algorithm will be one that is guaranteed to find the global minimum. Such an algorithm arises from a subdivision process. Repeatedly subdivide R into smaller rectangles, retaining only those rectangles that have a chance of containing the global minimum. The identification of these subrectangles can be done by estimating the size of the function and its derivatives on the rectangle. This point of view is really one of interval arithmetic. In interval arithmetic, the basic objects are closed intervals $[a,b]$ on the real line; extended arithmetic is generally used, so that endpoints can be $\pm\infty$. One can design algorithms so that elementary functions can be applied to such intervals (or, in our case, pairs of intervals), and the result is an interval that contains the image of the function on the domain in question. However, the resulting interval is not simply the interval from the minimum to the maximum of the function. Rather it is an enclosing interval, defined so as to make its computation easy and fast; it well generally be larger than the smallest interval containing all the f-values.

For example, it is easy to determine an enclosing interval for $\sin([a,b])$: just check whether $[a,b]$ contains a real number of the form $\frac{\pi}{2} + 2n\pi$ $(n \in \mathbb{Z})$. If so, the upper end of the enclosing interval is 1; if not, it is simply $\max(\sin a, \sin b)$. Finding the lower end is similar. The exponential function is monotonic, so one need look only at the endpoints. For sums, one follows a worst-case methodology and adds the left ends and then the right ends. So if $g(x) = \sin x + \cos x$, then this approach will not yield a very tight result, as $[-2,2]$ is only a loose bound on the actual range of g, $[-\sqrt{2}, \sqrt{2}\,]$. This phenomenon is known as *dependence*, since the two summands are treated as being independent, when they are not. Nevertheless, as the intervals get smaller and the various pieces become monotonic, the interval computations yield tighter results. Products are similar to sums, but with several cases. There are several technicalities that make the implementation of an interval

[1] http://is.dal.ca/~jdpinter/

arithmetic system tedious and difficult to get perfectly correct: one critical point is that one must always round outwards. But *Mathematica* has interval arithmetic built-in, and there are packages available for other languages, such as IntpakX[2] for Maple, Intlab[3] for Matlab, and the public-domain SMATH library[4] for C. Thus one can use a variety of environments to design algorithms that, assuming the interval arithmetic and the ordinary arithmetic are implemented properly, will give digits that are verified to be correct. For the algorithms we present here, we will assume that a comprehensive interval package is available.

One simple application of these ideas is to the estimation exercise used in §4.1 to ascertain that the square R contains the global minimum. The interval output of f applied to the half-plane $-\infty < x \leqslant -1$ is $[-3.223, \infty]$, and the same is true if the input region — the half-plane — is rotated 90°, 180°, or 270° around the origin. This means that in these four regions — the complement of R — the function is greater than -3.23, and so the regions can be ignored. Here is how to do this in *Mathematica*, which has interval arithmetic implemented for elementary functions.

A Mathematica session.

```
f[Interval[{-∞, -1.}], Interval[{-∞, ∞}]]
Interval[{-3.223591543636455, ∞}]
```

Now we can design an algorithm to solve Problem 4 as follows (this approach is based on the solution of the Wolfram Research team, the only team to use interval methods on this problem, and is one of the basic algorithms of interval arithmetic; see [Han92, Chap. 9] and [Kea96, §5.1]). We start with R and the knowledge that the minimum is less than -3.24. We then repeatedly subdivide R, retaining only those subrectangles T that have a chance of containing the global minimum. That is determined by checking the following three conditions. Throughout the interval discussion we use the notation $h[T]$ to refer to an enclosing interval for $\{h(t) : t \in T\}$.

(a) $f[T]$ is an interval whose left end is less than or equal to the current upper bound on the absolute minimum.

(b) $f_x[T]$ is an interval whose left end is negative and right end is positive.

(c) $f_y[T]$ is an interval whose left end is negative and right end is positive.

For (a), we have to keep track of the current upper bound. It is natural to try condition (a) by itself; such a simple approach will get one quickly into the region of the lowest minimum, but the number of intervals then blows up because the flat nature of the function near the minimum makes it hard to get sufficiently tight enclosing intervals for the f-values to discard them. The other two conditions arise from the fact that the minimum occurs at a critical point, and leads to an algorithm that is more aggressive in discarding intervals. Conditions (b) and (c) are easy to implement (the partial derivatives are similar to f in complexity) and

[2] http://www.mapleapps.com/powertools/interval/Interval.shtml
[3] http://www.ti3.tu-harburg.de/~rump/intlab/
[4] http://interval.sourceforge.net/interval/C/smathlib/README.html

the subdivision process then converges quickly to the answer for the problem at hand. While a finer subdivision might sometimes be appropriate, simply dividing each rectangles into four congruent subrectangles is adequate.

In the implementation we must be careful, as it is important to always improve the current upper bound as soon as that is possible. In the algorithm that follows, this improvement is done for the entire round of same-sized rectangles when a_1 is updated. The algorithm is given here for the plane, but nothing new is needed to apply it to n-space; later in this section we will present a more sophisticated algorithm for use in all dimensions.

Algorithm 4.2. Using Intervals to Minimize a Function in a Rectangle.

Assumptions: An interval arithmetic implementation that works for f, f_x, and f_y is available.

Inputs: R, the starting rectangle;
$f(x, y)$, a continuously differentiable function on R;
ϵ, a bound on the absolute error for the final approximation to the lowest f-value on R;
b, an upper bound on the lowest f-value on R, obtained perhaps by a preliminary search;
$i_{\max}$, a bound on the number of subdivisions.

Output: Interval approximations to the location of the minimum and the f-value there, the latter being an interval of size less than ϵ (or a warning that the maximum number of subdivisions has been reached). If the global minimum occurs more than once, then more than one solution will be returned.

Notation: $\mathcal{R}$ is a set of rectangles that might contain the global minimum; a_0 and a_1 are lower and upper bounds, respectively, on the f-value sought, and an *interior* rectangle is one that lies in the interior of R.

Step 1: Initialize: Let $\mathcal{R} = \{R\}$, $i = 0$; $a_0 = -\infty$, $a_1 = b$.
Step 2: The main loop:
While $a_1 - a_0 > \epsilon$ and $i < i_{\max}$:
Let $i = i + 1$;
Let $\mathcal{R}$ = the set of all rectangles that arise by uniformly dividing each rectangle in $\mathcal{R}$ into 4 rectangles;
Let $a_1 = \min(a_1, \min_{T \in \mathcal{R}}(f[T]))$;
Check size: Delete from $\mathcal{R}$ any rectangle T for which the left end of $f[T]$ is not less than a_1;
Check the gradient: Delete from $\mathcal{R}$ any interior rectangle T for which $f_x[T]$ does not contain 0 or $f_y[T]$ does not contain 0;
Let $a_0 = \min_{T \in \mathcal{R}}(f[T])$.
Step 3: Return the centers of the rectangles in $\mathcal{R}$ and of the f-interval for the rectangles in $\mathcal{R}$.

Appendix C.5.3 contains a bare-bones implementation in *Mathematica*, in the simplest form necessary to determine 10 digits of the minimum (both the checking of the number of subdivisions and the distinction involving interior squares are suppressed). Appendix C.4.3 also has code that does the same thing using the Intlab package for Matlab. Moreover, there are self-contained packages available that are devoted to the use of interval algorithms in global optimization; two prominent ones are COCONUT[5] and GlobSol[6], both of which solve this problem with no difficulty. The *Mathematica* version of the algorithm takes under two seconds to solve Problem 4 completely as follows.

A Mathematica session.

```
LowestCriticalPoint[f[x, y], {x, -1, 1}, {y, -1, 1}, -3.24, 10^-9]
{{-3.306868647912808, -3.3068686470376663},
  {{-0.024403079696639973, 0.21061242715950385}}}
```

A more comprehensive routine, with many bells and whistles, is available at the web page for this book. That program includes an option for monitoring the progress of the algorithm (including the generation of graphics showing the candidate rectangles) and for getting high accuracy. A run with the tolerance ϵ set to 10^{-12} takes only a few seconds (all timings in this chapter are on a Macintosh G4 laptop with a one GHz CPU) and uses 47 subdivision rounds (iterations of the main While loop). The total number of rectangles examined is 1372; the total number of interval function evaluations is 2210, and the numbers of candidate rectangles after each subdivision step are:

$$4, 16, 64, 240, 440, 232, 136, 48, 24, 12, 12, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,$$
$$4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4$$

Thus we see that at the third and later subdivisions, some subrectangles were discarded, and after 11 rounds only a single rectangle remained: it was then subdivided 35 times to obtain more accuracy. The interval returned by this computation is -3.30687^{49}_{56}, which determines 12 digits of the answer (its midpoint is correct to 16 digits). Figure 4.5 shows the candidate rectangles at the end of each of the first 12 subdivision steps.

As with the random search of Algorithm 4.1, an obvious way to improve this basic algorithm is to switch to a rootfinder on the gradient once one has the minimum narrowed down to a small region. But there are many other ways to improve and extend this basic interval subdivision process and we shall discuss those later in §4.5 and §4.6.

4.4 Calculus

A natural idea for this problem is to use the most elementary ideas of calculus and try to find all the critical points — the roots of $f_x = f_y = 0$ — in the rectangle R;

[5] http://www.mat.univie.ac.at/~neum/glopt/coconut/branch.html
[6] http://www.mscs.mu.edu/~globsol/

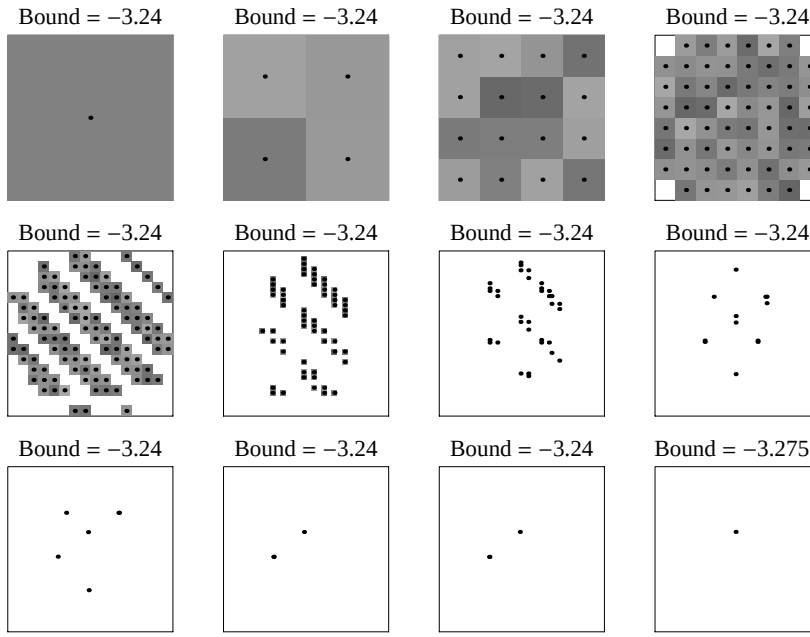

Figure 4.5. *The first 12 iterations of the interval subdivision algorithm. After 47 rounds, 12 digits of the answer are known.*

then the smallest f-value among them is the global minimum. While this is not the most efficient way to attack an optimization problem — most of the critical points are irrelevant, and the interval method of Algorithm 4.2 is more efficient at focusing on a region of interest — we present it here because it works, and it is a good general method for finding the zeros of a pair of functions in a rectangle. Moreover, if the objective function were one for which an interval implementation was not easily available, then the method of this section would be a useful alternative.

The partial derivatives are simple enough, but finding all the zeros of two nonlinear functions in a rectangle is a nontrivial matter (see the quote starting this chapter). Yet it can be done for the problem at hand by a very simple method (§4.6 will address the issue of verifying correctness of the list of critical points). The main idea, if one is seeking the roots of $f = g = 0$, is to generate a plot of the zero-set of f and use the data in the plot to help find the zero [SW97]. If a good contour-generating program is available, one can use it to approximate the zero-set of f and then one can design an algorithm to find all zeros in a rectangle as follows.

Algorithm 4.3. Using Contours to Approximate the Roots of $f = g = 0$.

Assumptions: The availability of a contour-drawing program that allows access to the points of the curves.

Inputs: f and g, continuous functions from $\mathbb{R}^2$ to $\mathbb{R}$;
R, the rectangular domain of interest;
r, the grid-resolution of the contour plot that will be used in step 1.

Output: Numerical approximations to each pair (x, y) that is a root of $f = g = 0$ and lies in R.

Notation: s will contain points that serve as seeds to a root-finder.

Step 1: Generate the curves corresponding to $f = 0$. This is most easily done by a contour plot, set to compute just the zero-level curves, and using r for the resolution in each dimension.

Step 2: For each contour curve, evaluate g on the points that approximate the curve; whenever there is a sign change, put the point just scanned into s.

Step 3: For each point in s, use Newton's method (or a secant method in the case of nondifferentiable functions) to determine a root. Discard the root if it lies outside R.

Step 4: If the number of roots differs from the number of seeds, restart the algorithm with a finer resolution.

Step 5: Return the roots.

The use of a contour routine was given in step 1 because that is quite simple and is adequate to the task at hand. However, there are more geometric (and faster) methods to compute an approximation to a contour curve that could be used. Such path-following algorithms have seen use in the study of bifurcation of differential equations. See [DKK91] and references therein to the AUTO program.

Implementing Algorithm 4.3 is quite simple provided one can get access to the data approximating the curves $f(x, y) = 0$. One might leave step 4 for a manual check, as opposed to making it part of the program. Figure 4.6, which includes the zero-sets for both functions, shows the result for f_x and f_y on a small section of the plane, where a resolution of 450 was used (this refers to the grid size used by the contour algorithm); five seconds were required to compute all 290 roots. The close-up on the right shows that the resolution was good enough to catch all the intersections.

To solve the problem over all of the square $R = [-1, 1]^2$ it makes sense to work on subsquares to avoid the memory consumption of a very finely resolved contour plot. Therefore we break R into 64 subsquares, compute all the critical points in each, and keep track of the one that gives the smallest value of f. Doing this takes under a minute and yields 2720 critical points. The one of interest is near $(-0.02, 0.21)$, where the f-value is near -3.3, the answer to the question. Newton's method can then be used to quickly obtain the function value at this critical point to more digits.

Manual examination of the contour plots on the subsquares to confirm that the resolution setting is adequate is not an efficient method for checking the results. A

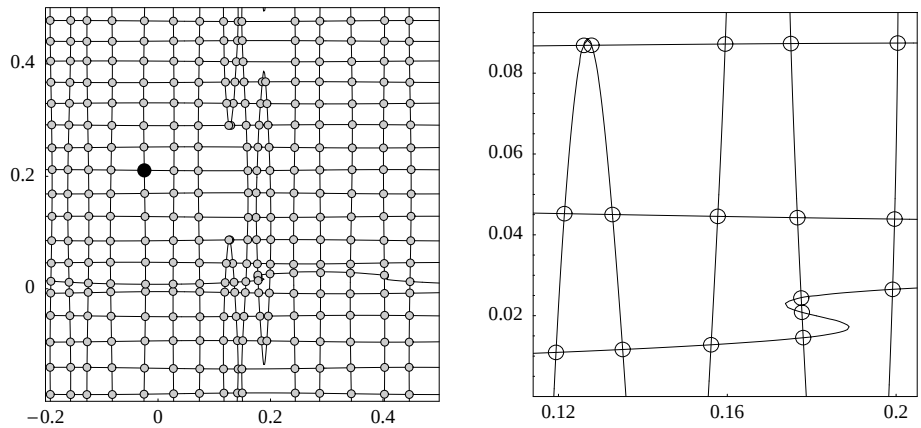

Figure 4.6. *A view of 290 solutions to $f_x = 0$ (vertical curves) and $f_y = 0$ (horizontal), where $f(x, y)$ is the objective function of Problem 4. The black point is the location of the global minimum. The close-up shows that the resolution was adequate for these curves.*

better way to gain confidence in the answer is to run the search multiple times, with increasing resolution, checking for stability; results of such work are in Table 4.1. While even the lowest resolution was enough to find the global minimum, quite high resolution is needed to catch all 2720 critical points in S. Note how, in some cases, 2720 seeds were found but they did not lead to the intended roots, and only 2719 roots were found. The high resolution needed to get 2720 roots (160×160 on each of 64 small squares) justifies the use of the subsquare approach. It is also a simple matter to use the second-derivative test to classify the critical points: 667 are maxima, 693 are minima, and 1360 are saddle points.

Note that the number of extrema ($693 + 667$) coincides with the number of saddles (1360). In some sense this is a coincidence, as it fails for, say, a rectangle that contains but a single critical point; it fails also for the rectangle $[-0.999, 0.999]^2$, which has $692 + 667$ extrema and 1357 saddles. Yet this phenomenon is related to the interesting subject known as Morse theory, which provides formulas for

$$\text{number of maxima} + \text{number of minima} - \text{number of saddles}$$

on a closed surface, such as a sphere or torus. These formulas are related to the Euler characteristic, and, on a torus, this number equals 0 [Mil63]. So let's make our function toroidal by taking f on $R = [-1, 1]^2$ and reflecting the function in each of the four sides, and continuing this reflection process throughout the plane. This gives us a function — call it g — that is doubly periodic on the whole plane, with a fundamental region that is $[-1, 3]^2$; thus g can be viewed as a function on a torus. Each critical point of f in R generates four copies of itself for g. But we have the complication that we have introduced new critical points on the boundary.

Table 4.1. *A resolution of* 160×160 *on each of 64 subsquares is needed to find all 2720 critical points in* $[-1, 1]^2$.

Resolution	Number of Seeds	Number of Roots
20	2642	2286
40	2700	2699
60	2716	2712
80	2714	2711
100	2718	2716
120	2720	2719
140	2720	2719
160	2720	2720
180	2720	2720
200	2720	2719
220	2720	2720
240	2720	2720
260	2720	2719
280	2720	2720
300	2720	2720
320	2720	2720
340	2720	2720
360	2720	2720
380	2720	2720
400	2720	2720

To understand the boundary, consider marching along the four edges of R and looking at each one-dimensional extremum: there will be the same number of maxima as minima (we assume none occurs at a corner, and no inflection points occur). Moreover, because of the symmetry of the reflection, each such maximum will be either a saddle or a maximum for g, and each minimum will be either a minimum or a saddle for g. If we can enumerate all these occurrences, then we can use the toroidal formula for g to deduce the situation for f in R. But a couple of surprising things happen because of the nature of $f(x, y)$. First, f_x is positive on the left and right edges of R and f_y is positive on the top and negative on the bottom. This means that all the maxima on the left edge become saddles and all the minima stay minima, with similar results holding for the other edges. We also need to know the numbers of extrema on the border. These are easily computed and turn out to be: top, 24 maxima, 24 minima; bottom, same; left, 29 maxima, 30 minima; right: same. These facts about f combine to show that the toroidal fomula applies verbatim to f restricted to R, thus explaining the coincidence. More important, this analysis relies only on a one-dimensional computation and so provides supporting

evidence that the counts obtained by the contour method (693, 667, and 1360) are correct.

Figure 4.7 shows another example that arose from a system of differential equations [Col95]. Finding the equilibrium points of the autonomous system

$$x' = 2y\cos\left(y^2\right)\cos(2x) - \cos y, \quad y' = 2\sin(y^2)\sin(2x) - \sin x,$$

is the same as finding all zeros of the right-hand sides. A contour resolution of 60 is sufficient, and the computation of all 73 zeros takes under a second.

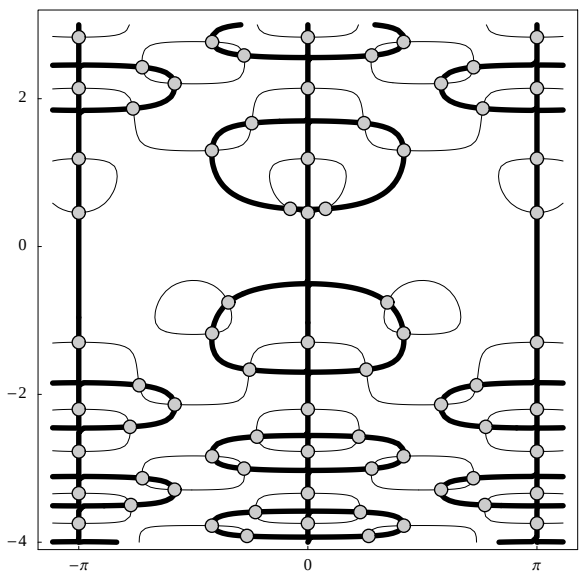

Figure 4.7. *This complicated example of simultaneous equations has 73 solutions in the rectangle shown.*

The total amount of programming needed to implement algorithm 3 is modest, provided one has a good way of accessing the data in the contour plot. In *Mathematica*, `Cases[ContourPlot[***],_Line,∞]` does the job.

A Mathematica session. The following is complete code for generating the zeros for the example in Figure 4.7. For a complete routine there are, as always, other issues to deal with: one should eliminate duplicates in the final result, since it is possible that different seeds will converge to the same zero.

```
g1[x_, y_] := 2 y Cos[y^2] Cos[2 x] - Cos[y];

g2[x_, y_] := 2 Sin[y^2] Sin[2x] - Sin[x];

{a, b, c, d} = {-3.45, 3.45, -4, 3};
```

```
contourdata = Map[First, Cases[Graphics[
       ContourPlot[g1[x, y], {x, a, b}, {y, c, d}, Contours → {0}, PlotPoints → 60]],
     _Line, ∞]];
seeds =
  Flatten[
    Map[
      #[[1 + Flatten[Position[Rest[ss = Sign[Apply[g2, #, 2]]] * Rest[RotateRight[ss]],
              -1]]]]&, contourdata], 1];
roots =
  Select[Map[{x, y}/.FindRoot[{g1[x, y] == 0, g2[x, y] == 0}, {x, #[[1]]}, {y, #[[2]]}]&,
     seeds], a < #[[1]] < b ∧ c < #[[2]] < d&];

Length[roots]
73
```

This technique also applies to finding all zeros of a complex function $f(z)$ in a rectangle by just applying the method to the system $\mathrm{Re}f = \mathrm{Im}f = 0$. The contour technique can fail badly if the zero-curves for f and g are tangent to each other (multiple zeros), since the approximations to the contours probably fail to have the necessary crossings. The technique is also limited to the plane. In §4.6 we will discuss a slower but more reliable method that addresses both of these problems.

4.5 Newton's Method for Intervals

The interval algorithm of §4.3 solves the problem nicely, but we can improve Algorithm 4.2 in several ways. Extensions of the algorithm so that it is faster and applies to functions in higher dimensions is an active area of research. Here we will only outline two ideas, one easy, one subtle, that can be used to improve it. The first, which might be called *opportunistic evaluation,* is quite simple: right after the subdivision, evaluate the objective function at the rectangle centers. This pointwise functional evaluation might pick up a value that can be used to improve the current best. Then the new current best is used in the size-checking step. This might not lead to great improvement in cases where we start with a good approximation to the answer (such as the -3.24 we had found), but if such an approximation is not available, these extra evaluations will help.

The second improvement relies on a root-finding algorithm called the interval Newton method, a beautiful and important idea originated by R. E. Moore in 1966 [Moo66]. We will use the term *boxes* for intervals in $\mathbb{R}^n$. The starting point is the Newton operator on boxes, defined as follows. Suppose $F : \mathbb{R}^n \to \mathbb{R}^n$ is continuously differentiable; then for a box X in n-space, $N(X)$ is defined to be $m - J^{-1} \cdot F(m)$, where m is the center of X and J is the interval Jacobian $F'[X] = (\partial F_i/\partial x_j[X])_{ij}$. Here J^{-1} is obtained by the standard arithmetic interval operations that arise in the computation of a matrix inverse. If $n = 1$, then $N(X)$ is simply $m - F(m)/F'[X]$, where m is the midpoint of X; note that $N(X) = [-\infty, \infty]$ if $0 \in F'[X]$. This definition leads to several important properties that can be used to design an algorithm to find roots. Here are the main points in the case of one dimension, which is much simpler than the general case.

Theorem 4.1. *Suppose $F : \mathbb{R} \to \mathbb{R}$ is a continuously differentiable function, and $X = [a, b]$ is a finite interval. Then:*

(a) *If r is a zero of F in X, then r lies in $N(X)$.*

(b) *If $N(X) \subseteq X$ then F has a zero in $N(X)$.*

(c) *If $N(X) \subseteq X$ then F has at most one zero in X.*

Proof. (a) If $F'(r) = 0$, then $N(X) = [-\infty, \infty]$; otherwise, the mean-value theorem yields $c \in X$ so that $F(r) - F(m) = F'(c)(r - m)$, and this implies that $r = m - F(m)/F'(c) \in N(X)$.

(b) The hypothesis implies that $F'(x) \neq 0$ on X. Then the mean-value theorem yields values c_i in X so that $m - F(m)/F'(c_1) - a = -F(a)/F'(c_1)$ and $b - (m - F(m)/F'(c_2)) = F(b)/F'(c_2)$; because $N(X) \subseteq [a, b]$, the product of the left sides is positive. But $F'(c_1)$ and $F'(c_2)$ have the same sign, and so the product $F(a)F(b)$ must be negative and F therefore has a zero in $[a, b]$.

(c) If there were two zeros, then the derivative would vanish between them, causing $N(X)$ to become infinite. □

This theorem is very powerful since one can use it to design a simple subdivision algorithm to find zeros. Just subdivide the given interval into subintervals and check, for each subinterval, whether the Newton condition — $N(X) \subseteq X$ — holds. If it does, then we know from (c) that there is one and only one zero in X. If it fails in the form $N(X) \cap X = \emptyset$, that is also good, for (a) tells us that there are no zeros in X. If neither situation applies, subdivide and try again. When the Newton condition does hold, we can just iterate the N operator. If we are close enough to the zero, the algorithm converges (see [Kea96, Thm. 1.14]), and in fact will converge quadratically, as does the traditional Newton root-finding method. But some bad things that can happen: we might not be close enough for convergence (the exact condition for this depends on, among other things, the tightness of the interval approximation J to the inverse of F'); or there are zeros for which the Newton condition will never hold. Think of the process as a queue: intervals are removed from the queue while, sometimes, their subdivisions are added to the queue, or the interval is added to a list of answers. If the queue becomes empty, we are done. If a max-iteration counter is exceeded and the queue is not empty, then there are some unresolved intervals. This will happen with multiple roots, such as occurs with $x^2 = 0$; for in such cases $N(X) = [-\infty, \infty]$.

For an application of the interval Newton method to a one-dimensional root-finding problem that arises in Problem 8 of the SIAM 100-Digit Challenge, see §8.3.2.

In higher dimensions, parts (a) and (c) remain true, but one must use a variation of the Newton operator to get (b), which is important to guarantee that a zero exists. One approach is by a preconditioning matrix (see [Neu90, Thm. 5.1.7]). Another approach, which we shall follow here, uses the Krawczyk operator. For more information on this important variation to Newton's method, see [Kea96, Neu90].

Let F, m, and J be as defined for the Newton operator. Consider $P(x) = x - YF(x)$, where Y is some type of approximation to J^{-1}, the interval matrix that is $F'[X]^{-1}$. Two natural choices for Y are: $F'(m)^{-1}$, or the inverse of the matrix of midpoints of the intervals in the matrix J. The latter is faster, since J has to be computed anyway, and that is what we shall use. Then the Krawczyk operator is defined to be $K(X) = m - YF(m) + (I - YJ)(X - m)$, where I is the $n \times n$ identity matrix (the numeric part of this computation should be done in an interval environment, with m replaced by a small interval around m). To understand the rationale behind the K operator, first recall the mean-value theorem. For a continuously differentiable $F : \mathbb{R}^n \to \mathbb{R}^n$, the mean-value theorem takes the following form: Given x and y in a box X, there are points $c_1, c_2, \ldots, c_n \in X$ so that $F(y) - F(x) = (\nabla F_i(c_i))(y - x)$, where $(\nabla F_i(c_i))$ denotes an $n \times n$ matrix with i indexing the rows.

Now $K(X)$ can be viewed as a "mean value extension" of P in the following sense: if $P'[X]$ is an interval enclosure of P' on X, the mean-value theorem implies that the box $Q = P(m) + P'[X](X - m)$ contains $P(X)$, which here denotes the exact image, $\{P(x) : x \in X\}$. But because $P'(x) = I - YF'(x)$, we may take $I - YJ$ to be the enclosure. Then Q becomes precisely $K(X)$ and we have proved that $K(X)$ contains $P(X)$. This smooths out the n-dimensional theory nicely as follows.

Theorem 4.2. *Suppose $F : \mathbb{R}^n \to \mathbb{R}^n$ is a continuously differentiable function, X is a finite box in $\mathbb{R}^n$, $J = F'[X]$ is a componentwise interval enclosure, and Y is the inverse of the matrix of midpoints of the intervals in J; Y is assumed to be nonsingular. Let K be the corresponding Krawczyk operator. Then:*

(a) *If r is a root of $F = 0$ in X, then r lies in $K(X)$.*

(b) *If $K(X) \subseteq X$ then F has a zero in $K(X)$.*

(c) *If $K(X) \subset X$ then F has at most one zero in X.*

Proof. (a) $K(X)$ contains $P(X)$, and so $P(r) \in K(X)$. But $P(r) = r - YF(r) = r$.

(b) Since $P(X) \subseteq K(X) \subseteq X$, P is a contraction mapping, and so the Brouwer fixed-point theorem for continuous functions implies that P has a fixed point in X, which means that F has a zero in X.

(c). This one is subtle. Suppose x and y are distinct roots of $F = 0$ in X. Then the mean-value theorem tells us that $F(x) - F(y) = (\nabla F_i(c_i))(x - y)$ for some points $c_i \in X$. It follows that the matrix is singular, and therefore J contains a singular matrix. However, it is not immediately obvious that this contradicts the Krawczyk condition. Yet it does. For a complete proof, see [Neu90, Thm 5.1.8], and note the hypothesis of strict containment here. □

This theorem immediately gives a local root-finding algorithm, one that will come into play in §4.6. If the Krawczyk condition — $K(X) \subset X$ — holds, then we know there is one and only one root in X, and that that root lies in $K(X)$. Simply iterate K. When we have a small enough interval, we know the root to the desired accuracy.

Here is how root-finding can help in optimization. Once we have the problem reduced to a single box, as happens after 11 subdivisions in the problem at hand, we can check the Krawczyk condition. If it holds, we can use the Newton–Krawczyk iteration to zoom into the unique critical point in the box, and that will give us the answer more quickly than repeated subdivisions (analogous to the advantage Newton's method provides over bisection in one dimension). For the problem at hand and a tolerance of 10^{-6} for the location of the minimum, using this idea (but not opportunistic evaluation) gives a speedup of about 10%.

To be precise, add the following step to the interval method of Algorithm 4.2, right after the gradient check:

> If $\mathcal{R}$ contains only one rectangle X, compute $K(X)$.
>
> If $K(X) \cap X = \emptyset$, then there is no critical point, and the minimum is on the border; do nothing.
>
> If $K(X) \subset X$, then iterate the K operator starting with $K(X)$ until the desired tolerance is reached.
>
> Use the last rectangle to set a_0 and a_1(the loop will then terminate).

A Mathematica session. An implementation of this extension is available at the web page for this book. Here is how that code would be used to get 100 digits of the answer, using interval arithmetic throughout. The switch to root-finding occurs after the 12th round, and then convergence is very fast. The output shown is the center of an interval of length less than 10^{-102}.

```
IntervalMinimize[f[x, y], {x, -1, 1}, {y, -1, 1}, -3.24,
  AccuracyGoal → 102, WorkingPrecision → 110]
-3.30686864747523728007611377089851565716648236147628821750129308550
   30919983788829503582548807528349918619 3
```

The use of interval methods in global optimization is a well-developed area, with techniques that go well beyond the brief introduction given here (see [Han92, Kea96]; Hansen reports success on a wide variety of problems, including a 10-dimensional one). Nevertheless, the fact that the very simplest ideas give a verified answer to Problem 4 in a few seconds and with only a few lines of code shows how powerful these ideas are. And it is noteworthy that interval analysis has played an important role in diverse areas of modern mathematics. For example, W. Tucker won the Moore prize for his recent work using interval analysis to prove that the Lorenz equations really do have a strange attractor; this solved one of Steven Smale's Problems for the 21st Century (see [Tuc02]); Hales and Ferguson [FH98] used interval methods in their resolution of the Kepler conjecture on sphere-packing, and Lanford [Lan82] used intervals to prove the existence of a universal limit — the Feigenbaum constant — in certain sequences of bifurcations.

The most important points of the interval approach to Problem 4 are:

- The algorithm is very general and will find the lowest critical point in the rectangle (provided interval arithmetic is available in a form that applies to the objective function and its partial derivatives).

- The results are verifiably correct if one uses intervals throughout.
- Getting the results to very high precision is not a problem.
- Having an interval root-finding method can lead to improvements in the optimization algorithm.
- And most important: The interval algorithm is a reasonable way to solve the problem whether or not one wants proved results. In short, interval thinking yields both a good algorithm and proved results.
- The basic ideas apply to functions of more variables, but life becomes more difficult in higher dimensions. See [Han92, Kea96] for discussions of various enhancements that can be used to improve the basic algorithm (one example: using the Hessian to eliminate n-dimensional intervals on which the function is not concave up).

4.6 A Validation Method for Roots

The pessimistic quote at the start of this chapter leads to the question: How can we be certain that the collection of 2720 critical points found in §4.4 is correct and complete? There are several ways to do this. One can use intervals to design a root-finding algorithm and check that it finds the same set of critical points. That can be done by a simple subdivision process where we keep only rectangles that have a chance of containing a zero, and constantly check and use the Krawczyk condition, $K(X) \subset X$. Here is a formal description.

Algorithm 4.4. Using Intervals to Find the Zeros of $F : \mathbb{R}^n \to \mathbb{R}^n$ in a Box.

Assumptions: Interval arithmetic is available for F.

Inputs: F, a continuously differentiable function from $\mathbb{R}^n$ to $\mathbb{R}^n$;
R, the box in which we want all zeros of F;
ϵ, an upper bound on the absolute error of the zero in terms of Euclidean distance;
`mtol`, a tolerance for combining boxes;
$i_{\max}$, a bound on the number of subdivision steps.

Output: A set of intervals with each one trapping a zero, and a set of intervals that are unresolved (these might contain none or one or more zeros).

Notation: For an n-dimensional box X, let m be the center of X and let M be a small box containing m. Then $K(X) = M - YF[M] + (I - YJ)(X - M)$ where J is the Jacobian interval matrix $F'[X]$, and Y is the inverse of the matrix obtained from J by replacing each box by its center. "Subdividing a box" here means dividing it into 2^n pieces by bisecting each side. "Combining" a set of boxes means replacing any two that have nonempty intersection with the smallest box that contains them

both, and repeating the process until no intersections remain. The Krawczyk condition for a box X is $K(X) \subset X$.

Step 1: Initialize: Let $\mathcal{R} = \{R\}$, $i = 0$, $a = \emptyset$.
Step 2: The main loop:
While $\mathcal{R} \neq \emptyset$ and $i < i_{\max}$:
Let $i = i + 1$;
Let $\mathcal{R}$ be the set of all boxes that arise by uniformly dividing each box in $\mathcal{R}$ into 2^n boxes;
If all boxes in $\mathcal{R}$ have their maximum side-dimension less than `mtol`, let $\mathcal{R}$ be the result of combining boxes in $\mathcal{R}$ until no further combining is possible;
For each $X \in \mathcal{R}$ compute $K(X)$, the Krawczyk image of X:
if $K(X) \cap X = \emptyset$, delete X from $\mathcal{R}$;
if $K(X) \subset X$,
iterate K starting from $K(X)$ until the tolerance (the size of the box, or the size of its F-image) is as desired;
add the resulting box to a and delete X from $\mathcal{R}$.
Step 3: Return a and $\mathcal{R}$, the latter being the unresolved boxes.

The combining step using `mtol` in the main loop requires some explanation. If a zero is near the edge of a box, then it can happen that the subdivision and Krawczyk contraction process will never succeed in isolating it. This can happen even in one dimension if the initial interval is $[-1, 1]$ and the zero is at 0, for then the subdivision process will produce two intervals with a very small overlap, thus placing the zero very near the edge. The combining step checks whether all remaining boxes are so small that we ought to have isolated the zeros (the merging tolerance is provided by the user); the fact that we have not done so ($\mathcal{R}$ is not empty) means that it would be wise to combine small boxes into larger ones. This will likely place the zero nearer the center of a box (but not at the exact center!), and the iterative process then succeeds. Of course, there is the chance of an infinite loop here. Thus it would be reasonable to run this algorithm first with `mtol` $= 0$ so that no combining takes place. If it fails to validate, it can be tried with a setting of, say, `mtol` $= 10^{-6}$.

For zeros x of F at which $F'(x)$ is singular, the Krawczyk condition is never satisfied, and the algorithm does not find those zeros. However, they are not lost, as they will be trapped within the unresolved intervals that are returned, and the user could investigate those further to try to determine if a zero lives in them.

Algorithm 4.4 succeeds in obtaining all 2720 zeros (in 8 minutes). It is an important technique, especially in higher dimensions, where we might not have another way to get at the roots (see [Kea96]). But in cases where we think we already have the zeros, we should use an interval algorithm for *validation*; this is a central theme in the field of interval analysis: it is often more appropriate to use traditional numerical algorithms and heuristics to get results that are believed to be complete, and then use interval analysis to validate the results.

Here is a second approach, where we use interval analysis as a validation tool. It is a two-step approach based on the theorem about the Krawczyk operator in §4.5, and it works well for the problem at hand. Suppose the set of approximate zeros is r and has size m. First we check that each zero in r is roughly correct: for each zero, let X be a small box centered at the zero and check that the Krawczyk condition $K(X) \subset X$ holds; this technique is called *ϵ-inflation* (originally due to Rump; see [Rum98] and references therein). Once this is done, we know that r contains approximations to a set of m true zeros. Then move to verify completeness by carrying out a subdivision process on the given domain, doing two things only with each box that shows up: if interval arithmetic or the Krawczyk operator shows that the enclosure for the F-values on the box does not contain the zero vector, it is discarded; and if the Krawczyk containment holds, so that the box does contain a zero, then it is again discarded, but a running count is incremented by one. Boxes that remain are subdivided. When no boxes remain, we know that the count equals the number of zeros. If this count equals m, we know the set r was indeed a complete set of approximations to the zeros. Here is a formal description.

Algorithm 4.5. Using Intervals to Validate a Set of Zeros.

Assumptions: Interval arithmetic is available for F and its partial derivatives.

Inputs: F, a continuously differentiable function from $\mathbb{R}^n$ to $\mathbb{R}^n$;
R, the box that is the validation domain;
r, a list, believed to be complete and correct, of the set of zeros of F in R;
ϵ, an upper bound of the absolute error on the zero in terms of Euclidean distance;
`mtol`, a tolerance for combining boxes.

Output: True, if the true zeros in R coincide with the points in r, with the absolute error in each case less than ϵ; False otherwise

Notation: As in Algorithm 4.4.

Step 1: ϵ-inflation:
For each point in r,
let X be the surrounding box of side-length $2\epsilon/\sqrt{n}$;
check that the Krawczyk condition holds for each X.
If there is any failure, stop and return False.

Step 2: Initialize the subdivision process. Let $\mathcal{R} = \{R\}$; $c = 0$.

Step 3: The main loop:
While $\mathcal{R} \neq \emptyset$:
Let $\mathcal{R}$ be the set of boxes obtained by subdividing each box in $\mathcal{R}$;
Delete from $\mathcal{R}$ any box for which the F-interval does not straddle 0;

Delete from $\mathcal{R}$ any box X for which the Krawczyk condition holds,
increase c by 1 for every such box;
Delete from $\mathcal{R}$ any box X for which $K(X) \cap X = \emptyset$;
If all boxes in $\mathcal{R}$ have their maximum side-dimension less than `mtol`,
let $\mathcal{R}$ be the result of combining boxes in $\mathcal{R}$
until no further combining is possible.

Step 4: If $c =$ the number of points in r, return True, otherwise False.

For the gradient of the function f of Problem 4, the contour method locates the 2720 zeros in 30 seconds. The validation method then takes 30 seconds to check that the zeros are correct, and another 5.5 minutes to verify that the set is complete. While the speedup over the method of using intervals from the beginning to find all the zeros is modest, the validation method is a more elegant algorithm and illustrates the important idea that, when possible, one should put off the interval work to the last stage of a computational project.

Algorithms 4.4 and 4.5 can be combined into a single algorithm that finds and validates the zeros. The key observation is that there is no need to iterate the Krawczyk operator when one can just use the traditional Newton method.

Algorithm 4.6. Using Intervals to Find and Validate a Set of Zeros.

Assumptions: Interval arithmetic is available for F and its partial derivatives.

Inputs: F, a continuously differentiable function from $\mathbb{R}^n$ to $\mathbb{R}^n$;
R, the box that is the validation domain;
ϵ, an upper bound on the absolute error of the zeros in terms of Euclidean distance;
`mtol`, a tolerance for combining boxes.

Output: Validated approximations to the zeros, with the absolute error in each case less than ϵ.

Notation: As before, with s being a set of rough approximations to the zeros, and r the set of final approximations.

Step 1: Let $s = \emptyset$; follow steps 1 and 2 of Algorithm 4.4, except that when $K(X) \subset X$ add the center of X to s.

Step 2: Apply Newton's method to each point in s, putting the result in r.

Step 3: Use ϵ-inflation as in step 1 of Algorithm 4.5 to verify that r is a complete set of zeros to the desired tolerance.

Step 4: Return r.

Algorithm 4.6 finds and validates all the critical points to the challenge problem in 5.5 minutes, a 9% time savings over the combination of Algorithms 4.3 and 4.5. The material at the web page for this book includes the *Mathematica* programs

`ValidateRoots` and `FindAndValidateRoots`.

4.7 Harder Problems

What sort of problem could have been asked instead of Problem 4 that would have been a little, or a lot, harder? There certainly would have been additional difficulties if there were more dimensions or if the objective function did not have interval arithmetic easily available. Indeed, the optimization needed to solve Problem 5 is exactly of this sort (4 dimensions, complicated objective), and that is quite a difficult problem for general-purpose minimization algorithms. Yet another sort of problem would arise if the objective function had not a single minimum, but a continuous set, such as a line or a plane (see [Han92]; there are some examples there). But let us look only at the dimension issue for a moment. Here is a slight variation of Problem 4, but in three dimensions: What is the global minimum of

$$g(x,y,z) = e^{\sin(50x)} + \sin(60e^y)\sin(60z) + \sin(70\sin x)\cos(10z) + \\ \sin\sin(80y) - \sin(10(x+z)) + (x^2+y^2+z^2)/4\ ?$$

The methods of this chapter work well on this problem, except the techniques that tried to find all the critical points in a box: there are probably over 100,000 such points. An approach that combines various methods to advantage would proceed as follows:

1. Use the genetic minimization routine with 200 points per generation and a scale factor of 0.9 to discover that g gets as small as -3.327. Differential evolution works too.

2. Use traditional root-finding (Newton) on the gradient to get the more accurate value of -3.32834 (or hundreds of digits if desired).

3. Use interval arithmetic as in §4.3 to then prove that the global minimum is inside the cube $[-0.77, 0.77]^3$.

4. Use the basic interval algorithm of §4.3, with upper bound -3.328 and Krawczyk iteration taking over after 11 rounds, to prove that the minimum is -3.32834, and occurs near $(-0.15, 0.29, -0.28)$. This takes almost a minute, and the total number of boxes examined is 9408.

A Mathematica session.

```
g[x_, y_, z_] := e^Sin[50 x] + Sin[60 e^y] Sin[60 z] + Sin[70 Sin[x]] Cos[10 z] +
    Sin[Sin[80 y]] - Sin[10 (x + z)] + 1/4 (x^2 + y^2 + z^2);

IntervalMinimize[g[x, y, z], {x, -0.77, 0.77}, {y, -0.77, 0.77},
  {z, -0.77, 0.77}, -3.328]
```

```
{Interval[{-3.328338345663281, -3.328338345663262}],
  {{x → Interval[{-0.1580368204689058, -0.1580368204689057}],
     y → Interval[{0.2910230486091526, 0.2910230486091528}],
     z → Interval[{-0.2892977987325703, -0.2892977987325701}]}}}
```

So for at least one complicated example, the algorithm works well in three dimensions. If the dimension is increased in such a way that the number of local minima grows with the the space (that is, exponentially), one would expect a slowdown in the algorithm, though it will be sensitive to the location of the minima and still might work well.

As a final example, we mention that the Krawczyk validation method given at the end of §4.6 can solve the related problem of finding the critical points of $g(x, y, z)$. Because there are many critical points we work in a small box only ($[0, 0.1] \times [0, 0.05] \times [0, 0.1]$) and show some of them, together with the three surfaces $g_x = 0$, $g_y = 0$, $g_z = 0$, in Figure 4.8. There are six zeros in this box.

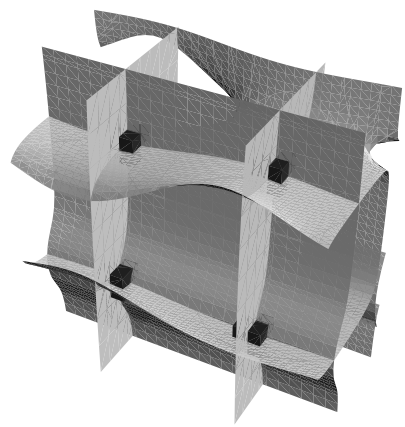

Figure 4.8. *A very small portion — within* $[0, 0.1] \times [0, 0.05] \times [0, 0.1]$ *— of the three surfaces* $g_x = 0$, $g_y = 0$, $g_z = 0$ *whose intersections form the critical points of* g*, with the roots obtained by the interval method shown as small cubes.*

4.8 Summary

Interval arithmetic is powerful and can obtain certifiably correct answers to optimization and root-finding problems. The technique is useful not only as a validation tool, but also as a complete algorithm. For many problems it makes sense to use other techniques first, and then use interval arithmetic to validate the results, if that is desired. Random search methods, especially evolutionary algorithms, can be very efficient at getting approximate results. The contour approach is somewhat specialized, but it too is very efficient at solving this, and other, two-dimensional problems.