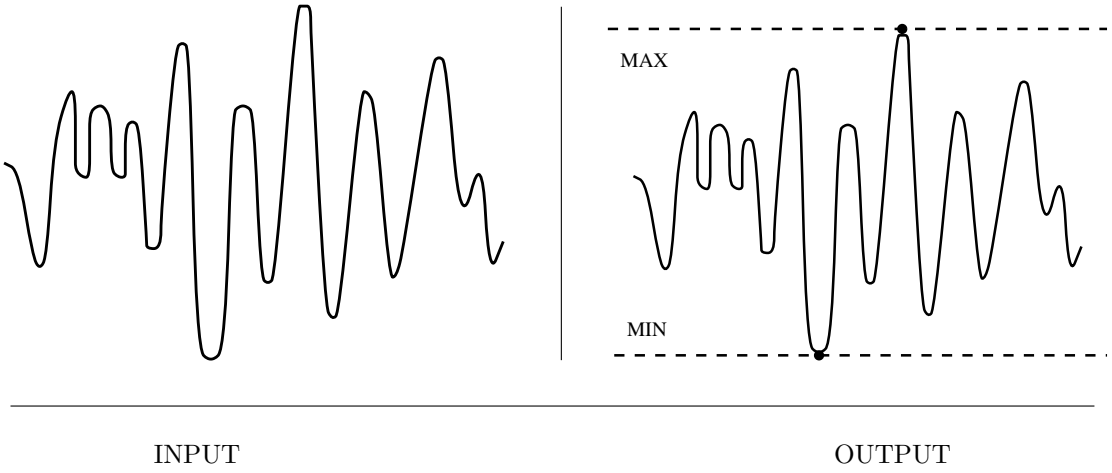

13.5 Constrained and Unconstrained Optimization

Input description: A function $f(x_1, \ldots, x_n)$.

Problem description: What point $p = (p_1, \ldots, p_n)$ maximizes (or minimizes) the function f?

Discussion: Most of this book concerns algorithms that optimize one thing or another. This section considers the general problem of optimizing functions where, due to lack of structure or knowledge, we are unable to exploit the problem-specific algorithms seen elsewhere in this book.

Optimization arises whenever there is an objective function that must be tuned for optimal performance. Suppose we are building a program to identify good stocks to invest in. We have certain financial data available to analyze—such as the price-earnings ratio, the interest rate, and the stock price—all as a function of time t. The key question is how much weight we should give to each of these factors, where these weights correspond to coefficients of a formula:

$$\text{stock-goodness}(t) = c_1 \times \text{price}(t) + c_2 \times \text{interest}(t) + c_3 \times \text{PE-ratio}(t)$$

We seek the numerical values c_1, c_2, c_3 whose stock-goodness function does the best job of evaluating stocks. Similar issues arise in tuning evaluation functions for any pattern recognition task.

Unconstrained optimization problems also arise in scientific computation. Physical systems from protein structures to galaxies naturally seek to minimize their "energy" or "potential function." Programs that attempt to simulate nature thus often define potential functions assigning a score to each possible object configuration, and then select the configuration that minimizes this potential.

Global optimization problems tend to be hard, and there are lots of ways to go about them. Ask the following questions to steer yourself in the right direction:

- *Am I doing constrained or unconstrained optimization?* – In unconstrained optimization, there are no limitations on the values of the parameters other than that they maximize the value of f. However, many applications demand constraints on these parameters that make certain points illegal, points that might otherwise be the global optimum. For example, companies cannot employ less than zero employees, no matter how much money they think they might save doing so. Constrained optimization problems typically require mathematical programming approaches, like linear programming, discussed in Section 13.6 (page 411).

- *Is the function I am trying to optimize described by a formula?* – Sometimes the function that you seek to optimize is presented as an algebraic formula, such as finding the minimum of $f(n) = n^2 - 6n + 2^{n+1}$. If so, the solution is to analytically take its derivative $f'(n)$ and see for which points p' we have $f'(p') = 0$. These points are either local maxima or minima, which can be distinguished by taking a second derivative or just plugging p' back into f and seeing what happens. Symbolic computation systems such as Mathematica and Maple are fairly effective at computing such derivatives, although using computer algebra systems effectively is somewhat of a black art. They are definitely worth a try, however, and you can always use them to plot a picture of your function to get a better idea of what you are dealing with.

- *Is it expensive to compute the function at a given point?* – Instead of a formula, we are often given a program or subroutine that evaluates f at a given point. Since we can request the value of any given point on demand by calling this function, we can poke around and try to guess the maxima.

  Our freedom to search in such a situation depends upon how efficiently we can evaluate f. Suppose that $f\ (x_1, \ldots, x_n)$ is the board evaluation function in a computer chess program, such that x_1 is how much a pawn is worth, x_2 is how much a bishop is worth, and so forth. To evaluate a set of coefficients as a board evaluator, we must play a bunch of games with it or test it on a library of known positions. Clearly, this is time-consuming, so we would have to be frugal in the number of evaluations of f we use to optimize the coefficients.

- *How many dimensions do we have? How many do we need?* – The difficulty in finding a global maximum increases rapidly with the number of dimensions (or parameters). For this reason, it often pays to reduce the dimension by ignoring some of the parameters. This runs counter to intuition, for the naive programmer is likely to incorporate as many variables as possible into their evaluation function. It is just too hard, however, to optimize such complicated

functions. Much better is to start with the three to five most important variables and do a good job optimizing the coefficients for these.

- *How smooth is my function?* The main difficulty of global optimization is getting trapped in local optima. Consider the problem of finding the highest point in a mountain range. If there is only one mountain and it is nicely shaped, we can find the top by just walking in whatever direction is up. However, if there are many false summits, or other mountains in the area, it is difficult to convince ourselves whether we really are at the highest point. *Smoothness* is the property that enables us to quickly find the local optimum from a given point. We assume smoothness in seeking the peak of the mountain by walking up. If the height at any given point was a completely random function, there would be no way we could find the optimum height short of sampling every single point.

The most efficient algorithms for unconstrained global optimization use derivatives and partial derivatives to find local optima, to point out the direction in which moving from the current point does the most to increase or decrease the function. Such derivatives can sometimes be computed analytically, or they can be estimated numerically by taking the difference between the values of nearby points. A variety of *steepest descent* and *conjugate gradient* methods to find local optima have been developed—similar in many ways to numerical root-finding algorithms.

It is a good idea to try several different methods on any given optimization problem. For this reason, we recommend experimenting with the implementations below before attempting to implement your own method. Clear descriptions of these algorithms are provided in several numerical algorithms books, in particular *Numerical Recipes* [PFTV07].

For constrained optimization, finding a point that satisfies all the constraints is often the difficult part of the problem. One approach is to use a method for unconstrained optimization, but add a penalty according to how many constraints are violated. Determining the right penalty function is problem-specific, but it often makes sense to vary the penalties as optimization proceeds. At the end, the penalties should be very high to ensure that all constraints are satisfied.

Simulated annealing is a fairly robust and simple approach to constrained optimization, particularly when we are optimizing over combinatorial structures (permutations, graphs, subsets). Techniques for simulated annealing are described in Section 7.5.3 (page 254).

Implementations: The world of constrained/unconstrained optimization is sufficiently confusing that several guides have been created to point people to the right codes. Particularly nice is Hans Mittlemann's *Decision Tree for Optimization Software* at *http://plato.asu.edu/guide.html*. Also check out the selection at GAMS, the NIST *Guide to Available Mathematical Software*, at http://gams.nist.gov.

NEOS (Network-Enabled Optimization System) provides a unique service—the opportunity to solve your problem remotely on computers and software at Argonne

National Laboratory. Linear programming and unconstrained optimization are both supported. Check out *http://www-neos.mcs.anl.gov/* when you need a solution instead of a program.

Several of the *Collected Algorithms of the ACM* are Fortran codes for unconstrained optimization, most notably Algorithm 566 [MGH81] , Algorithm 702 [SF92] , and Algorithm 734 [Buc94] . Algorithm 744 [Rab95] does unconstrained optimization in Lisp. They are all available from Netlib (see Section 19.1.5 (page 659)).

General purpose simulated annealing implementations are available, and probably are the best place to start experimenting with this technique for constrained optimization. Feel free to try my code from Section 7.5.3 (page 254). Particularly popular is *Adaptive Simulated Annealing (ASA)*, written in C by Lester Ingber and available at *http://asa-caltech.sourceforge.net/*.

Both the Java Genetic Algorithms Package (JGAP) (*http://jgap.sourceforge.net/*) and the C Genetic Algorithm Utility Library (GAUL) (*http://gaul.sourceforge.net/*) are designed to aid in the development of applications that use genetic/evolutionary algorithms. I am highly skeptical about genetic algorithms (see Section 7.8 (page 266)), but other people seem to find them irresistible.

Notes: Steepest-descent methods for unconstrained optimization are discussed in most books on numerical methods, including [BT92, PFTV07]. Unconstrained optimization is the topic of several books, including [Bre73, Fle80].

Simulated annealing was devised by Kirkpatrick et al. [KGV83] as a modern variation of the Metropolis algorithm [MRRT53]. Both use Monte Carlo techniques to compute the minimum energy state of a system. Good expositions on all local search variations, including simulated annealing, appear in [AL97].

Genetic algorithms were developed and popularized by Holland [Hol75, Hol92]. More sympathetic expositions on genetic algorithms include [LP02, MF00]. Tabu search [Glo90] is yet another heuristic search procedure with a devoted following.

Related Problems: Linear programming (see page 411), satisfiability (see page 472).