# 15 Search

Once you've gathered your data, selected a representation to work with, chosen a framework for function approximation, specified an error metric, and expressed your prior beliefs about the model, then comes the essential step of choosing the best parameters. If they enter linearly, the best global values can be found in one step with a singular value decomposition, but as we saw in the last chapter coping with the curse of dimensionality can require parameters to be inside nonlinearities. This entails an iterative search starting from an initial guess. Such exploration is similar to the challenge faced by a mountaineer in picking a route up a demanding peak, but with two essential complications: the search might be in a 200-dimensional space instead of just 2D, and because of the cost of function evaluation you must do the equivalent of climbing while looking down at your feet, using only information available in a local neighborhood.

The need to search for parameters to make a function extremal occurs in many kinds of optimization. We already saw one nice way to do nonlinear search, the Levenberg–Marquardt method (Section 12.4.1). But this is far from the end of the story. Levenberg–Marquardt assumes that it is possible to calculate both the first and second derivatives of the function to be minimized, that the starting parameter values are near the desired extremum of the cost function, and that the function is reasonably smooth. In practice these assumptions often do not apply, and so in this chapter we will look at ways to relax them. This chapter will cover the *unconstrained optimization* of a cost function; Chapter 17 will add restrictions to the search space for *constrained optimization.*

Perhaps because nature must solve these kinds of optimization problems so frequently, algorithms in this chapter can have a kind of natural familiarity missing in most numerical methods. We'll see blobs (for the downhill simplex search), mass (momentum for avoiding local minima), temperature (via simulated annealing), and reproduction (with genetic algorithms). The cost of this kind of intuition is rigor. While there is some supporting theory, their real justification lies in their empirical performance. Consequently it's important to view them not as fixed received wisdom, but rather as a framework to guide further exploration.

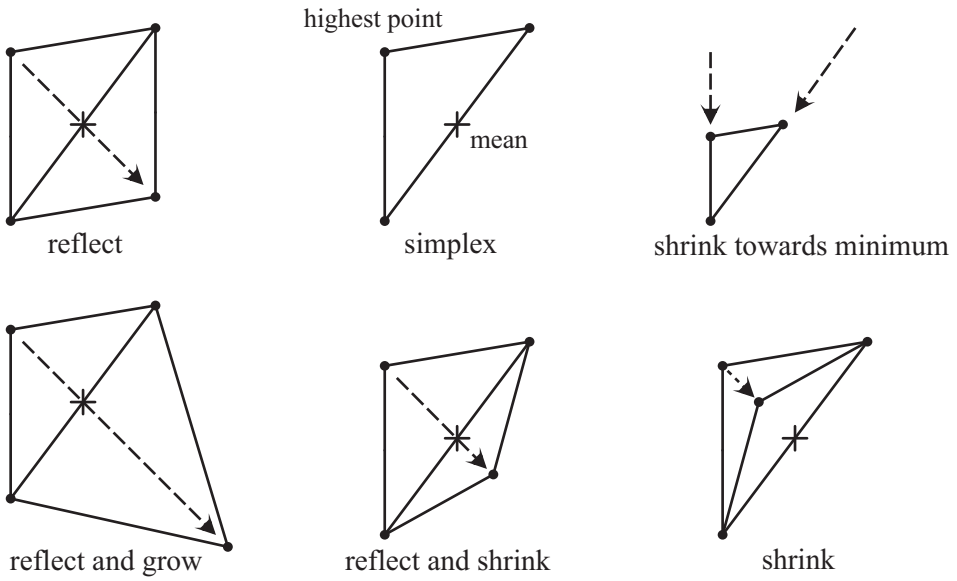


Figure 15.1. Downhill simplex update moves.

## 15.1 MULTIDIMENSIONAL SEARCH

### 15.1.1 Downhill Simplex

As features are added to a model it quickly becomes inconvenient, if not impossible, to analytically calculate the partial derivatives with respect to the parameters (impossible because the function being searched might itself be the result of a numerical simulation or experimental measurement). A simple-minded solution is to evaluate the function at a constellation of points around the current location and use a finite difference approximation to find the partials, but the number of function evaluations required at each step rapidly becomes prohibitive in a high-dimensional space. After each function evaluation there should be some kind of update of the search to make sure that the next function evaluation is as useful as possible. Each function evaluation requires choosing not only the most promising direction but also the length scale to be checked. In Levenberg–Marquardt we used the Hessian to set the scale, but doing that numerically would require still more function evaluations. The *Downhill Simplex* method, also called the *Nelder–Mead* method after its original authors [Nelder & Mead, 1965], is a delightful (although far from optimal) solution to these problems. Of all algorithms it arguably provides the most functionality for the least amount of code, along with the most entertainment.

A *simplex* is a geometrical figure that has one more vertex than dimension: a triangle in 2D, a tetrahedtron in 3D, and so forth. Nelder–Mead starts by choosing an initial simplex, for example by picking a random location and taking unit vectors for the edges, and evaluating the function being searched at the $D + 1$ vertices of the simplex. Then an iterative procedure attempts to improve the vertex with the highest value of the function at each step (assuming that the goal is minimization; for maximization the vertex with the smallest value is updated).

The moves are shown in Figure 15.1. Let $\vec{x}_i$ by the location of the $i$th vertex, ordered so that $f(\vec{x}_1) > f(\vec{x}_2) > \ldots > f(\vec{x}_{D+1})$. The first step is to calculate the center of the face of the simplex defined by all of the vertices other than the one we're trying to improve:

$$\vec{x}_{mean} = \frac{1}{D}\sum_{i=2}^{D+1}\vec{x}_i \quad . \tag{15.1}$$

Since all of the other vertices have a better function value it's a reasonable guess that they give a good direction to move in. Therefore the next step is to reflect the point across the face:

$$\begin{aligned}\vec{x}_1 \to \vec{x}_1^{new} &= \vec{x}_{mean} + (\vec{x}_{mean} - \vec{x}_1) \\ &= 2\vec{x}_{mean} - \vec{x}_1 \quad .\end{aligned} \tag{15.2}$$

If $f(\vec{x}_1^{new}) < f(x_{D+1})$ the new point is now the best vertex in the simplex and the move is clearly a good one. Therefore it's worth checking to see if it's even better to double the size of the step:

$$\begin{aligned}\vec{x}_1 \to \vec{x}_1^{new} &= \vec{x}_{mean} + 2(\vec{x}_{mean} - \vec{x}_1) \\ &= 3\vec{x}_{mean} - 2\vec{x}_1 \quad .\end{aligned} \tag{15.3}$$

If growing like this gives a better function value than just reflecting we keep the move, otherwise we go back to the point found by reflecting alone. If growing does succeed it's possible to try moving the point further still in the same direction, but this isn't done because it would result in a long skinny simplex. For the simplex to be most effective its size in each direction should be appropriate for the scale of variation of the function, therefore after growing it's better to go back and improve the new worse point (the one that had been $\vec{x}_2$).

If after reflecting $\vec{x}_1^{new}$ is neither best nor worst, we keep it and move on to the new worst point. If it is still worst, it means that we overshot the minimum of the function. Therefore instead of reflecting and growing we can try reflecting and shrinking:

$$\begin{aligned}\vec{x}_1 \to \vec{x}_1^{new} &= \vec{x}_{mean} + \frac{1}{2}(\vec{x}_{mean} - \vec{x}_1) \\ &= \frac{3}{2}\vec{x}_{mean} - \frac{1}{2}\vec{x}_1 \quad .\end{aligned} \tag{15.4}$$

If $f(\vec{x}_1^{new}) < f(\vec{x}_2)$ we accept the move and try to improve $f(\vec{x}_2)$; if after reflecting and shrinking $f(\vec{x}_1^{new})$ is still worse we can try just shrinking:

$$\begin{aligned}\vec{x}_1 \to \vec{x}_1^{new} &= \vec{x}_{mean} - \frac{1}{2}(\vec{x}_{mean} - \vec{x}_1) \\ &= \frac{1}{2}(\vec{x}_{mean} + \vec{x}_1) \quad .\end{aligned} \tag{15.5}$$

If after shrinking $f(\vec{x}_1^{new})$ is still worse the conclusion is that the moves we're making are too big to find the minimum, so we give up and shrink all of the vertices towards the best one:

$$\begin{aligned}\vec{x}_i \to \vec{x}_i^{new} &= \vec{x}_i - \frac{1}{2}(\vec{x}_i - \vec{x}_{D+1}) \quad (i = 1, \ldots, D) \\ &= \frac{1}{2}(\vec{x}_i + \vec{x}_{D+1}) \quad .\end{aligned} \tag{15.6}$$

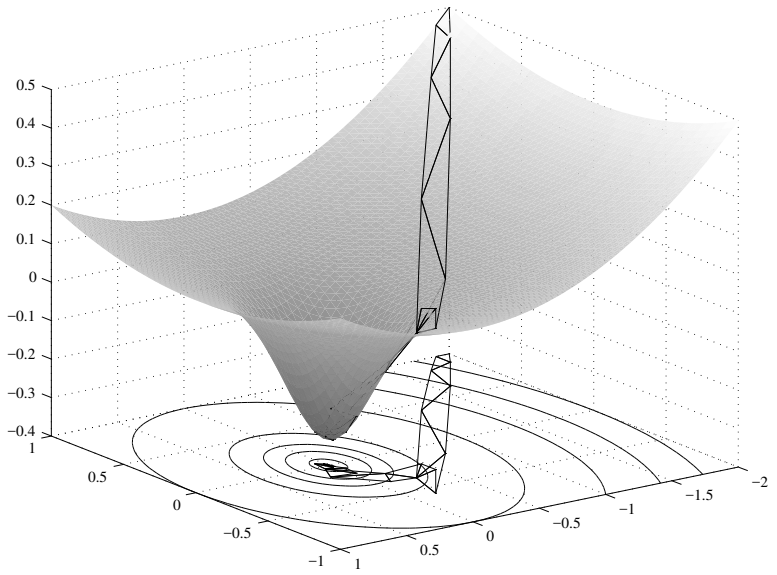


Figure 15.2. Downhill simplex function minimization. Above is the evolution of the simplex shown on the surface; below the 2D projection on a contour map.

Taken together these moves result in a surprisingly adept simplex. As it tumbles downhill it grows and shrinks to find the most advantageous length scale for searching in each direction, squeezing through narrow valleys where needed and racing down smooth regions where possible. When it reaches a minimum it will give up and shrink down around it, triggering a stopping decision when the values are no longer improving. The beauty of this algorithm is that other than the stopping criterion there are no adjustable algorithm parameters to set. Figure 15.2 shows what a simplex looks like searching for the minimum of a function.

Each vertex in the simplex requires storage for $D$ points, and there are $D+1$ of them, for a storage requirement of $D(D+1)$.

### 15.1.2 Line Minimization

Instead of maintaining multiple sets of points, an alternative approach is to maintain multiple search directions. *Line minimizations* can then be sequentially performed to minimize the function in these directions. To see how that's done, start with a triple of points (taken here to be scalar distances in the search direction) $x_0$, $x_1$ and $x_2$, where $x_0$ and $x_2$ are known to bracket the minimum and $x_1$ is between them (we'll see below where these come from).

A new function evaluation will be needed to determine which side of $x_1$ the minimum is on. Let $a = |x_1 - x_0|$ and $b = |x_2 - x_1|$ (Figure 15.3). To keep the search balanced, we can ask for the ratio $a/b$ to remain the same for the new points whichever side it is on. If $c = |x_3 - x_1|$, then we want

$$\frac{a}{b} = \frac{b}{c} \tag{15.7}$$

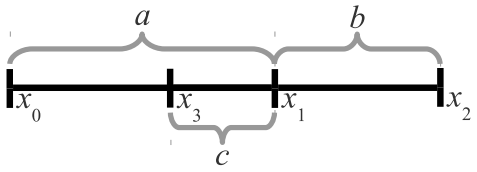


Figure 15.3. Golden section line search.

and

$$\frac{a}{b} = \frac{a-c}{c} \quad . \tag{15.8}$$

Eliminating $c$ between them,

$$\left(\frac{a}{b}\right)^2 - \frac{a}{b} - 1 = 0 \tag{15.9}$$

or

$$\frac{a}{b} = \frac{1+\sqrt{5}}{2} \tag{15.10}$$

i.e., the *golden ratio* (taking the positive root). That's why this is called *golden section bisection search*. Starting with a bracketing triple, a new point is tested at the golden ratio distance into the larger of the two intervals, then based on the function evaluation there it becomes either the central or end point of a new bracketing triple.

The bracketing triple can be found by doing the opposite procedure. Three starting points are chosen with the golden mean spacing, then new points are added in the downhill direction, maintaining the golden mean ratio, until one ends up on the far side of the minimum.

Each successive evaluation improves the location of the minimum by a constant fraction, for a linear convergence rate. If the function being searched is well-behaved, quadratic interpolation can be used for superlinear convergence. If the function being searched is a polynomial

$$f = ax^2 + bx + c \tag{15.11}$$

and we have three values for it $\{(x_1, f_1), (x_2, f_2), (x_3, f_3)\}$, then with those three equations we can solve for the three unknowns in the polynomial. Asking for $df/dx = 0$ then gives

$$x = \frac{(x_2^2 - x_3^2)f_1 + (x_3^2 - x_1^2)f_2 + (x_1^2 - x_2^2)f_3}{2[(x_2 - x_3)f_1 + (x_3 - x_1)f_2 + (x_1 - x_2)f_3]} \tag{15.12}$$

This converges in a single step for the polynomial, and otherwise ends up near it based on how good the polynomial approximation is. *Brent's method* is like the Levenberg-Marquardt method for line search, starting with bisection away from the minimum, and switching to root-finding near it [Press *et al*., 2007].

### 15.1.3 Direction Set

Equipped with a line minimization routine, a naive way to do multidimensional search is to cycle through the axes of the space, minimizing along each direction in turn. This does

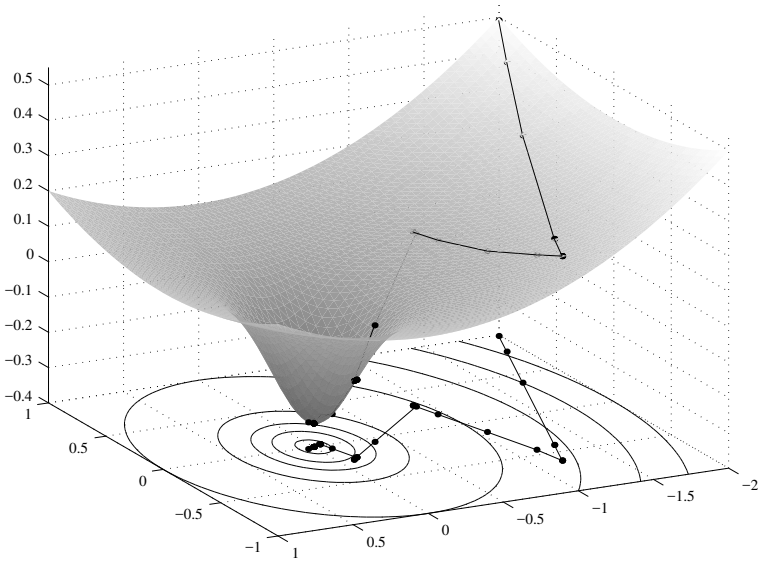


Figure 15.4. Minimization by Powell's method in 2D, illustrated by starting with random search directions. After each pair of line minimizations the net change is taken as a new search direction, replacing the most similar previous direction.

not work well, because each succeeding minimization can influence the optimizations that came before it, leading to a large number of zigs and zags to follow the function along directions that might not be lined up with the axes of the space.

*Powell's method* or the *direction set method* improves on this idea by updating the directions that are searched to try to find a set of directions that don't interfere with each other [Acton, 1990]. Starting from an initial guess $\vec{x}_0$, $D$ line minimizations are performed, initially along each of the $D$ axes of the space. The resulting point $\vec{x}_1$ defines a change vector, $\Delta\vec{x} = \vec{x}_1 - \vec{x}_0$. This is a good direction to keep for future minimizations. Since we need to keep a set of $D$ directions to span the $D$-dimensional space, the new direction should replace the one most similar to it. This could be determined by computing all the dot products among the vectors, or more simply estimated by throwing out the direction in which there was the largest change in $f$ (and hence the direction which most likely made the most important contribution to $\Delta\vec{x}$). An example is shown in Figure 15.4.

Powell's method requires storage for $D$ directions, each with $D$ points, for a total of $D^2$.

### 15.1.4 Conjugate Gradient

Simplex and Powell's method assume that only the function can be evaluated. Naturally, it's possible to do better if the gradient of the function is known. A second-order expansion of the function around a point (or exact expression for a quadratic form) is

$$f \approx f(\vec{x}_0) + \sum_i \left.\frac{\partial f}{\partial x_i}\right|_{\vec{x}_0} x_i + \frac{1}{2}\sum_{i,j} \left.\frac{\partial^2 f}{\partial x_i \partial x_j}\right|_{\vec{x}_0} x_i x_j$$

$$\equiv c - \vec{b} \cdot \vec{x} + \frac{1}{2}\vec{x} \cdot \mathbf{A} \cdot \vec{x} \quad . \tag{15.13}$$

Taking partials,

$$\frac{\partial f}{\partial x_i} \approx \frac{\partial f}{\partial x_i}\bigg|_{\vec{x}_0} + \frac{1}{2}\sum_j \frac{\partial^2 f}{\partial x_i \partial x_j}\bigg|_{\vec{x}_0} x_j + \frac{1}{2}\sum_j \frac{\partial^2 f}{\partial x_j \partial x_i}\bigg|_{\vec{x}_0} x_j \tag{15.14}$$

and so the gradient $\vec{g}$ up to this second-order approximation is

$$\vec{g} = \nabla f \approx \mathbf{A} \cdot \vec{x} - \vec{b} \quad . \tag{15.15}$$

(if $\mathbf{A}^T = \mathbf{A}$, which will hold for a Hessian). Finding where the gradient vanishes thus solves $\mathbf{A} \cdot \vec{x} = \vec{b}$ for a quadratic form, which is a useful approach to solve sparse matrix equations [Golub & Loan, 1989].

Given a starting point $\vec{x}_n$ and direction $\vec{d}_n$, a line minimization will find a new point $\vec{x}_{n+1} = \vec{x}_n + \alpha_n \vec{d}_n$ at a distance $\alpha_n$. The gradient at this new point will be perpendicular to the line minimization direction:

$$\begin{aligned} 0 &= \vec{d}_n \cdot \vec{g}_{n+1} \\ &\approx \vec{d}_n \cdot \left(\mathbf{A} \cdot \vec{x}_{n+1} - \vec{b}\right) \end{aligned} \tag{15.16}$$

because otherwise there would still be a decreasing component in the minimization direction. Going in the direction of the new gradient is called *steepest descent*, and is a bad idea because after the new minimization the old one would no longer hold, potentially requiring many right-angle turns to follow a function. If instead the new search direction is chosen so that

$$\vec{d}_n \cdot \left[\mathbf{A} \cdot \left(\vec{x}_{n+1} + \alpha \vec{d}_{n+1}\right) - \vec{b}\right] = 0 \tag{15.17}$$

is still true then they won't conflict. This will be the case if

$$\vec{d}_n \cdot \mathbf{A} \cdot \vec{d}_{n+1} = 0 \tag{15.18}$$

in which case the vectors are said to be *conjugate* with respect to $\mathbf{A}$

To find this new direction, we can try constructing it from a combination of the new gradient and the old direction:

$$\vec{d}_{n+1} = \vec{g}_{n+1} + \gamma_n \vec{d}_n \tag{15.19}$$

where $\gamma_n$ is a constant to be found. Plugging that into the conjugacy condition gives

$$\begin{aligned} 0 &= \vec{d}_{n+1} \cdot \mathbf{A} \cdot \vec{d}_n \\ &= \vec{g}_{n+1} \cdot \mathbf{A} \cdot \vec{d}_n + \gamma_n \vec{d}_n \cdot \mathbf{A} \cdot \vec{d}_n \\ \Rightarrow \gamma_n &= -\frac{\vec{g}_{n+1} \cdot \mathbf{A} \cdot \vec{d}_n}{\vec{d}_n \cdot \mathbf{A} \cdot \vec{d}_n} \end{aligned} \tag{15.20}$$

Evaluating this requires knowing $\mathbf{A}$, the Hessian, which we don't know, but observe that

$$\begin{aligned} \vec{g}_{n+1} &= \mathbf{A} \cdot \vec{x}_{n+1} - \vec{b} \\ &= \mathbf{A} \cdot \left(\vec{x}_n + \alpha_n \vec{d}_n\right) - \vec{b} \\ &= \vec{g}_n + \alpha_n \mathbf{A} \cdot \vec{d}_n \\ \vec{g}_{n+1} - \vec{g}_n &= \alpha_n \mathbf{A} \cdot \vec{d}_n \quad . \end{aligned} \tag{15.21}$$

Since we're assuming that we do know the gradients we can use that expression to eliminate **A**:

$$\begin{aligned}
\gamma_n &= -\frac{\vec{g}_{n+1} \cdot (\vec{g}_{n+1} - \vec{g}_n) / \alpha_n}{\vec{d}_n \cdot (\vec{g}_{n+1} - \vec{g}_n) / \alpha_n} \\
&= -\frac{\vec{g}_{n+1} \cdot (\vec{g}_{n+1} - \vec{g}_n)}{\vec{d}_n \cdot \vec{g}_{n+1} - \vec{d}_n \cdot \vec{g}_n} \\
&= \frac{\vec{g}_{n+1} \cdot (\vec{g}_{n+1} - \vec{g}_n)}{\vec{d}_n \cdot \vec{g}_n} \\
&= \frac{\vec{g}_{n+1} \cdot (\vec{g}_{n+1} - \vec{g}_n)}{\left(\vec{g}_n + \gamma_{n-1}\vec{d}_{n-1}\right) \cdot \vec{g}_n} \\
&= \frac{\vec{g}_{n+1} \cdot (\vec{g}_{n+1} - \vec{g}_n)}{\vec{g}_n \cdot \vec{g}_n + \gamma_{n-1}\vec{d}_{n-1} \cdot \vec{g}_n} \\
&= \frac{\vec{g}_{n+1} \cdot (\vec{g}_{n+1} - \vec{g}_n)}{\vec{g}_n \cdot \vec{g}_n}
\end{aligned} \tag{15.22}$$

This is the *Polak-Ribiere* conjugate gradient algorithm [Polak & Ribiere, 1969]. If the function is exactly a quadratic form then this will converge after $D$ line minimizations in a $D$-dimensional space, otherwise the iteration can be continued. The earlier *Fletcher-Reeves* algorithm leaves off the evaluation of $\vec{g}_{n+1} \cdot \vec{g}_n$, which vanishes for a quadratic form [Fletcher & Reeves, 1964].

The conjugate gradient algorithm requires keeping track of the old and new search directions, and evaluating the old and new gradients, for a storage proportional to $D$. For a large problem that's quite a difference from the $D^2$ required by the simplex or direction set methods.

### 15.1.5 Stochastic Search

filtered noisy evaluation [Kirk *et al.*, 1993] (analog logic)

stochastic gradient descent

## 15.2 LOCAL MINIMA

In functional optimization, like life, sometimes taking short-term gains does not lead to the best long-term return. Most nontrivial functions have local minima that will trap an unwary algorithm that always accepts a step that improves the function. Consider Figure 15.5, which shows the behavior of downhill simplex search on a function with two minima. Since by definition it always makes moves that improve the local value of the function it necessarily gets caught in the basin around the starting point.

All techniques for coping with local minima must in some way force the search to sometimes make moves that are not locally desirable, a numerical analog of learning from mistakes. A crude but eminently usable way to do this is by adding *momentum* to the search. Just as momentum causes a particle to keep moving in an old direction as a new force is applied, a fraction of whatever update was last done in a search

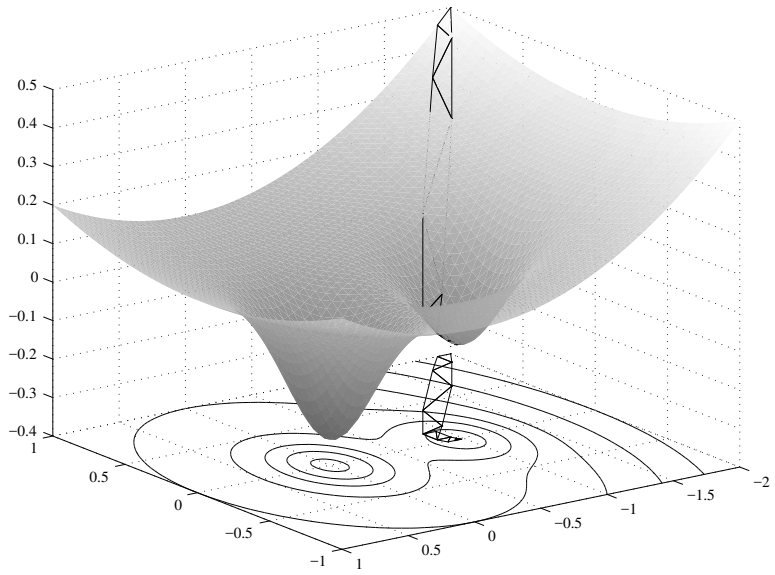


Figure 15.5. Downhill simplex function minimization getting caught in a local mininum.

can be added to whatever new update the algorithm deems best. This is usually done by loose analogy rather than by formally mapping the problem onto classical mechanics.

For downhill simplex, let $\Delta\vec{x}(n)$ be the total change in the $n$th move to the worst point in the simplex, $\vec{x}_1^{new}(n) = \vec{x}_1^{old}(n) + \Delta\vec{x}(n)$. Momentum adds to this some of the change from the previous move

$$\vec{x}_1^{new}(n) = \vec{x}_1^{old}(n) + \Delta\vec{x}(n) + \alpha[\vec{x}_1^{new}(n-1) - \vec{x}_1^{old}(n-1)] \quad . \tag{15.23}$$

As Figure 15.6 shows, this lets the simplex "roll" out of a local minima and find a better one. As $\alpha$ is increased the simplex is able to climb out of deeper minima, but takes longer to converge around the final minimum.

The attraction of using momentum is that it is a trivial addition to almost any other algorithm. Of course including momentum does not guarantee that local minima will be avoided; that depends on the details of how much is used, what the configuration of the search space is, and where the initial condition is located. As more momentum is included it is possible to climb out of deeper minima, and the reasonable hope is that the resulting directions in which the search proceeds will be sufficiently randomized to have a good chance of finding better minima. This is making an implicit statistical assumption about the dynamics of the search, in effect asking that there be enough redirection of the search for it to appear ergodic and hence be able to reach the whole search space. In the rest of this chapter we will turn to explicit statistical assumptions to randomize searches.

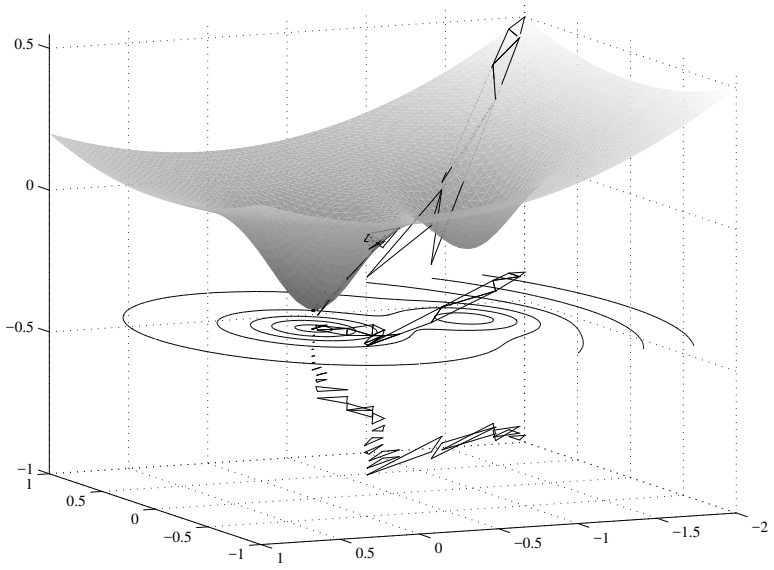


Figure 15.6. Downhill simplex function minimization with momenum added, avoiding a local minimum. At the bottom the updates are shown displaced vertically as a function of the iteration for clarity.

## 15.3 SIMULATED ANNEALING

The growth of a crystal from a liquid solves a difficult optimization problem. If the liquid was instantaneously frozen, the atoms would be trapped in the configuration they had in the liquid state. This arrangement has a higher energy than the crystalline ordering, but there is an energy barrier to be surmounted to reach the crystalline state from a glassy one. If instead the liquid was slowly cooled, the atoms would start at a high temperature exploring many local rearrangements, and then at a lower temperature become trapped in the lowest energy configuration that was reached. The slower the cooling rate, the more likely it is that they will find the ordering that has the lowest global energy. This process of slowly cooling a system to eliminate defects is called *annealing*.

Annealing was introduced to numerical methods in the 1950s by Metropolis and coworkers [Metropolis *et al*., 1953]. They were studying statistical mechanical systems and wanted to include thermodynamic fluctuations. In thermodynamics, the relative probability of seeing a system in a state with an energy $E$ is proportional to an exponential of the energy divided by the temperature times Boltzmann's constant $k$ [Balian, 1991]:

$$p(E) \propto e^{-E/kT} \equiv e^{-\beta E} \quad . \tag{15.24}$$

This is called a *Boltzmann factor*. At $T = 0$ this means that the system will be in the lowest energy state, but at $T > 0$ there is some chance to see it in any of the states. Metropolis proposed that to update a simulation, a new state reachable from the current state be randomly selected and have its energy evaluated. If the energy is lower, the state should always be accepted. But if the energy is higher, it is accepted based on the

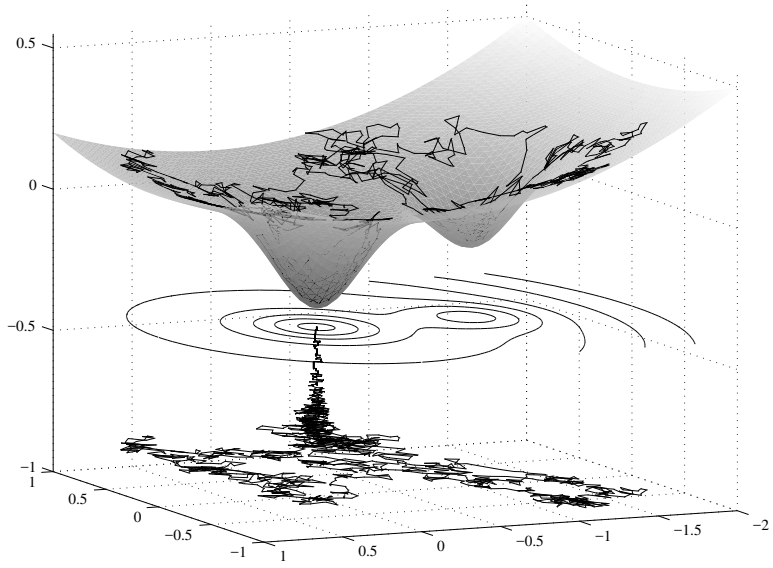


Figure 15.7. Optimization by simulated annealing. As the temperature is decreased, the random walker gets caught in the mininum. In the lower trace the iterations are displaced vertically.

probability to see a fluctuation of that size, which is proportional to the exponential of the change in energy $\exp(-\beta\Delta E)$. This is easily implemented by accepting the move if a random number drawn from a uniform distribution between 0 and 1 is less than a threshold value equal to the desired probability.

In the 1980s Scott Kirkpatrick realized that the same idea could be used to search for solutions to other hard problems, such as finding a good routing of the wires on a printed circuit board [Kirkpatrick *et al*., 1983; Kirkpatrick, 1984]. This idea is called by analogy *simulated annealing*. Now the energy is replaced by a cost function, such as the total length of wire on the circuit board, or a MAP likelihood estimate. Trial moves that update the state are evaluated. Moves that improve the search are always accepted, and moves that make it worse are taken with a probability given by a Boltzmann factor in the change in the cost function. At a high temperature this means that the search indiscriminately accepts any move offered, and hence pays little attention to the function being searched and tumbles around the space. As the temperature is lowered the function becomes more and more significant, until as $T \to 0$ the search becomes trapped in the lowest minima that it has reached. Figure 15.7 shows simulated annealing working on a function with multiple minima.

There are two key elements in implementing simulated annealing. The first is the selection of the trial moves to be evaluated. This could be done randomly, but that makes little sense since in a high-dimensional space the chance of guessing a good direction is very small. Better is to use an algorithm such as downhill simplex or conjugate gradient that makes intelligent choices for the updates, but include the Boltzmann factor to allow it to make mistakes. The second part of implementing simulated annealing is choosing the cooling schedule. Cool too fast and you freeze the system in a bad solution; cool too

slowly and you waste computer time. There is a large literature on techniques motivated by thermodynamics that analyze the fluctuations to determine if the system is being kept near equilibrium. A more modest empirical approach is to try runs at successively slower cooling rates until the answer stops improving.

Simulated annealing requires repeatedly making a probabilistic decision to accept a move proportional to its Boltzmann factor. If $\alpha \in [0, 1]$ is the probability with which we want to accept a move, this can be done simply by drawing a random number uniformly distributed between 0 and 1. If the number is less than $\alpha$ we accept the move, and if the number is greater than $\alpha$ we reject it. By definition, this gives a probability of success of $\alpha$, and for failure of $1 - \alpha$.

## 15.4 GENETIC ALGORITHMS

There is a natural system that solves even harder problems than crystal growth: evolution. Atoms in a crystal have a very limited repertoire of possible moves compared to the options of, say, a bird in choosing how to fly. The bird must determine the size and shape of the wings, their structural and aerodynamic properties, the control strategy for flight, as well as all of the other aspects of its life-cycle that support its ability to fly.

Evolution works by using a large population to explore many options in parallel, rather than concentrating on trying many changes around a single design. The same can be true of numerical methods. Simulated annealing keeps one set of search parameters that are repeatedly updated. An alternative is to keep an ensemble of sets of parameters, spending less time on any one member of the ensemble. Techniques that do this have come to be called *genetic algorithms*, or *GAs*, by analogy with the evolutionary update of the genome [Forrest, 1993].

The state of a GA is given by a population, with each member of the population being a complete set of parameters for the function being searched. The whole population is updated in generations. There are four steps to the update:

- *Fitness*
  The fitness step evaluates the function being searched for the set of parameters for each member of the population.
- *Reproduction*
  The members of the new population are selected based on their fitness. The total size of the population is fixed, and the probability for a member of the previous generation to appear in the new one is proportional to its fitness. A set of parameters with a low fitness might disappear, and one with a high fitness can be duplicated many times. The weighting of how strongly the fitness determines the relative rates of reproduction is analogous to the temperature of simulated annealing. Low selectivity accepts any solution; high selectivity forces one solution to dominate.
- *Crossover*
  In crossover, members of the ensemble can share parameters. After two parents are randomly chosen based on their fitness, the offspring gets its parameter values based on some kind of random selection from the parents. The usefulness of crossover depends on the nature of the function being searched. If it naturally decouples into subproblems,

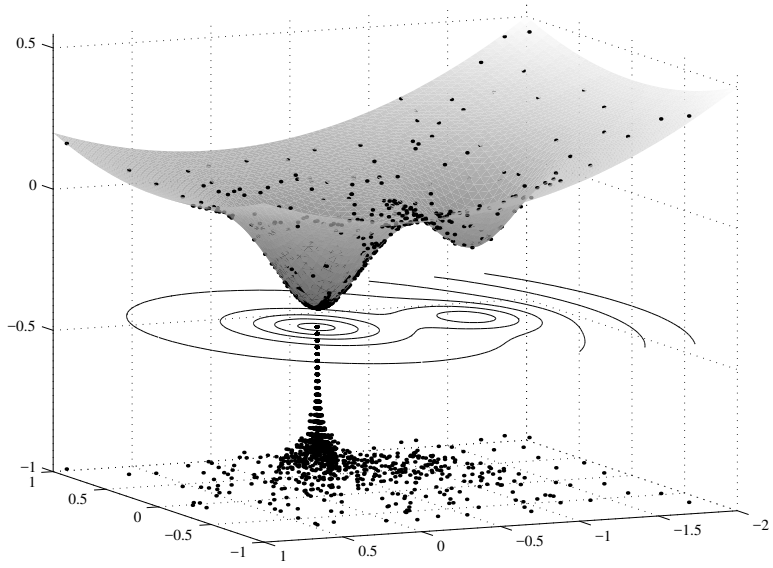


Figure 15.8. Optimization by a genetic algorithm, with populations displaced vertically at the bottom.

then one parent may be good at one part of the problem and the other at another, so taking a block of parameters from each can be advantageous. If on the other hand the parameter values are all intimately linked then crossover has little value and can be skipped. Crossover introduces to the search process a notion of collaboration among members of the ensemble, making it possible to jump to solutions in new parts of the space without having to make a series of discrete moves to get there.

- *Mutation*
  This step introduces changes into the parameters. Like simulated annealing it could be done randomly, but preferably takes advantage of whatever is known about how to generate good moves for the problem.

Figure 15.8 shows the evolution of the population in a GA searching the same function used in the previous examples. Clearly this is a long way from downhill simplex search. There the only choice was when to stop; for a GA there are many decisions to be made about how to implement each of the steps. For simple problems it can be possible to do this optimally [Prügel-Bennett & Shapiro, 1994], but the most important use of GAs is for problems where such calculations are not possible. Even so, plausible choices for each of the options can lead to surprisingly satisfactory performance. After all, evolution has come a long way with many suboptimal choices.

## 15.5 THE BLESSING OF DIMENSIONALITY

This chapter has presented an escalating series of search algorithm enhancements to handle successively more difficult problems. Figure 15.9 illustrates the types of functions to which they apply. On the left is a smooth function with a broad minimum. As long as you have reasonable confidence that you can guess starting parameter values that

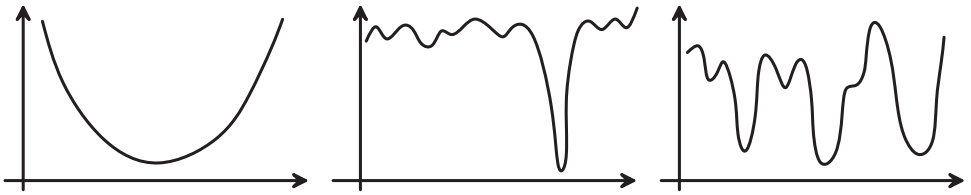

Figure 15.9. Types of search problems. Left, a global minimum that can be found by gradient descent. Right, a function with many equally good nearby minima that can be found with a local stochastic search. And center, a function with a large difference between typical local minima and the global minimum, requiring a global search.

put you within the basin of the desired minimum, variants of gradient descent will get you there (Nelder–Mead or Powell's method if only function evaluation is possible, conjugate gradient if first derivatives are available, Levenberg–Marquardt if first and second derivatives are available). At the opposite extreme is a function with very many equally good local minima. Wherever you start out, a bit of randomness can get you out of the small local minima and into one of the good ones, making this an appropriate problem for simulated annealing. But simulated annealing is less suitable for the problem in the middle, which has one global minimum that is much better than all the other local ones. Here it's very important that you find that basin, and so it's better to spend less time improving any one set of parameters and more time working with an ensemble to examine more parts of the space. This is where GAs are best.

It's not possible to decide which of these applies to a given nontrivial problem, because only a small part of the search space can ever be glimpsed. But a good clue is provided by the statistics of a number of local searches starting from random initial conditions. If the same answer keeps being found the case on the left applies, if different answers are always found but they have similar costs then the case on the right applies, and if there is a large range in the best solutions found then it's the one in the middle.

It should be clear that simulated annealing and genetic algorithms are not disjoint alternatives. Both start with a good generator of local moves. Simulated annealing introduces stochasticity to cope with local minima; genetic algorithms introduce an ensemble to cope with still rougher search landscapes. One idea came from observing how thermodynamic systems solve problems, the other from biology, but divorced from the legacy of their origins both have sensible lessons for numerical search.

Perhaps the biggest surprise of all about high-dimensional nonlinear search is just how well it can work. Variants of simulated annealing are now used routinely for such hard problems as routing fleets of airplanes. The explanation for this success can be called the *blessing of dimensionality*. The kind of landscape shown on the right of Figure 15.9 has come to be well understood through the study of *spin glasses* [Mezard, 1987]. These are physical systems in which atomic spin degrees of freedom have random interaction strengths; Problems 15.2 and 15.3 work through an example. For a large number of spins it is extremely difficult to find the global minimum, but there is an enormous number of local minima that are all almost equally good (remember that the number of configurations is exponential in the number of spins). Almost anywhere you look you will be near a reasonably low-energy solution.

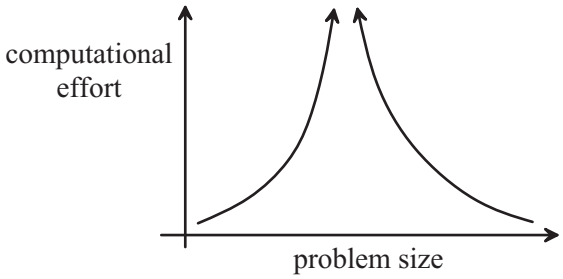


Figure 15.10. Characteristic computational effort as a function of the problem size.

In fact, the really hard problems are not the big ones. If there are enough planes in a fleet, there are many different routings that are comparable. Small problems by definition are not hard; for a small fleet the routing options can be exhaustively checked. What's difficult is the intermediate case, where there are too many planes for a simple search to find the global best answer, but there are too few planes for there to be many alternative acceptable routings. Many other problems have been observed to have this kind of behavior, shown in Figure 15.10. The effort to find an answer goes down for small problems because there are so few possible answers to check, and it goes down for big problems because there are so many different ways to solve a problem. In between is a transition between these regimes that can be very difficult to handle. This crossover has been shown to have many of the characteristics of a phase transition [Cheeseman *et al*., 1991; Kirkpatrick & Selman, 1994], which is usually where the most complex behavior in a system occurs.

Figure 15.10 explains a great deal of the confusion about the relative claims for search algorithms. Many different techniques work equally well on toy problems, because they are so easy. And many techniques can work on what look like hard problems, if they are given a big enough space to work in. But the really hard problems in between can trip up techniques with stellar records at either extreme. This figure helps explain the success of neural networks (Chapter 14): if a model is not going to have a small number of meaningful parameters, then the best thing to do is to give it so many adjustable parameters that there's no trouble finding a good solution, and prevent overfitting by imposing priors. And it helps explain the importance of heuristics and relaxations in problem-solving (Chapter 18): it can be preferable to find the exact solution to an approximate problem rather than an approximate solution to an exact problem.

## 15.6 SELECTED REFERENCES

[Acton, 1990] Acton, Forman S. (1990). *Numerical Methods That Work*. Washington, DC: Mathematical Association of America.

This should be required reading for anyone who uses any search algorithm, to understand why the author feels that "They are the first refuge of the computational scoundrel, and one feels at times that the world would be a better place if they were quietly abandoned."

[Press *et al*., 2007] Press, William H., Teukolsky, Saul A., Vetterling, William T., & Flannery, Brian P. (2007). *Numerical Recipes in C: The Art of Scientific Computing*. 3nd edn. Cambridge: Cambridge University Press.

*Numerical Recipes* has a good collection of search algorithms.

## 15.7 PROBLEMS

(15.1) (*a*) Plot the *Rosenbrock function* [Rosenbrock, 1960]

$$f = (1 - x)^2 + 100(y - x^2)^2 \quad . \tag{15.25}$$

(*b*) Pick a stopping criterion and use the downhill simplex method to search for its minimum starting from $x = y = -1$, plotting the search path, and comparing the computing time and memory used.

(*c*) Repeat with the direction set method.

(*d*) Repeat with the conjugate gradient method.

(*e*) Repeat with the Levenberg-Marquardt method (using the gradient and Hessian of the function).

(15.2) Consider a 1D spin glass defined by a set of spins $S_1, S_2, \ldots, S_N$, where each spin can be either +1 or $-1$. The energy of the spins is defined by

$$E = -\sum_{i=1}^{N} J_i S_i S_{i+1} \quad , \tag{15.26}$$

where the $J_i$'s are random variables drawn from a Gaussian distribution with zero mean and unit variance, and $S_{N+1} = S_1$ (periodic boundary conditions). Find a low-energy configuration of the spins by simulated annealing. The minimum possible energy is bounded by

$$E_{min} = -\sum_{i=1}^{N} |J_i| \quad ; \tag{15.27}$$

compare your result with this. At each iteration flip a single randomly chosen spin, and if the energy increases by $\Delta E$ accept the move with a probability

$$p = e^{-\beta \Delta E} \tag{15.28}$$

(always accept a move that decreases the energy). Take $\beta$ to be proportional to time, $\beta = \alpha t$ (where $t$ is the number of iterations), and repeat the problem for $\alpha = 0.1, 0.01$, and 0.001 for $N = 100$. Choose a single set of values for the $J$'s and the starting values for the spins and use these for each of the cooling rates.

(15.3) Now solve the same problem with a genetic algorithm (keep the same values for the $J$'s as the previous problem). Start with a population of 100 randomly drawn sets of spins. At each time step evaluate the energy of each member of the population, and then assign it a probability for reproduction of

$$p \propto e^{-\beta(E - E_{min})} \quad . \tag{15.29}$$

Generate 100 new strings by, for each string, choosing two of the strings from the previous population by drawing from this probability distribution, choosing a random crossover point, taking the bits to the left of the crossover point in the first string and the bits to the right in the second, and then mutating by randomly

flipping a bit in the resulting string. Plot the minimum energy in the population as a function of time step for $\beta = 10, 1, 0.1, 0.01$.