

to the usual method by any factor substantially larger than the “tender-loving-care factor” (which reflects the programming effort of the proponents).

Problems where the objective function and/or one or more of the constraints are replaced by expressions nonlinear in the variables are called *nonlinear programming problems*. The literature on such problems is vast, but outside our scope. The special case of quadratic expressions is called *quadratic programming*. Optimization problems where the variables take on only integer values are called *integer programming problems*, a special case of *discrete optimization* generally. The next section looks at a particular kind of discrete optimization problem.

CITED REFERENCES AND FURTHER READING:

- Bland, R.G. 1981, *Scientific American*, vol. 244 (June), pp. 126–144. [1]
 Dantzig, G.B. 1963, *Linear Programming and Extensions* (Princeton, NJ: Princeton University Press). [2]
 Kolata, G. 1982, *Science*, vol. 217, p. 39. [3]
 Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley), Chapters 7–8.
 Cooper, L., and Steinberg, D. 1970, *Introduction to Methods of Optimization* (Philadelphia: Saunders).
 Gass, S.T. 1969, *Linear Programming*, 3rd ed. (New York: McGraw-Hill).
 Murty, K.G. 1976, *Linear and Combinatorial Programming* (New York: Wiley).
 Land, A.H., and Powell, S. 1973, *Fortran Codes for Mathematical Programming* (London: Wiley-Interscience).
 Kuenzi, H.P., Tzschach, H.G., and Zehnder, C.A. 1971, *Numerical Methods of Mathematical Optimization* (New York: Academic Press). [4]
 Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §4.10.
 Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [5]

10.9 Simulated Annealing Methods

The *method of simulated annealing* [1,2] is a technique that has attracted significant attention as suitable for optimization problems of large scale, especially ones where a desired global extremum is hidden among many, poorer, local extrema. For practical purposes, simulated annealing has effectively “solved” the famous *traveling salesman problem* of finding the shortest cyclical itinerary for a traveling salesman who must visit each of N cities in turn. (Other practical methods have also been found.) The method has also been used successfully for designing complex integrated circuits: The arrangement of several hundred thousand circuit elements on a tiny silicon substrate is optimized so as to minimize interference among their connecting wires [3,4]. Surprisingly, the implementation of the algorithm is relatively simple.

Notice that the two applications cited are both examples of *combinatorial minimization*. There is an objective function to be minimized, as usual; but the space over which that function is defined is not simply the N -dimensional space of N continuously variable parameters. Rather, it is a discrete, but very large, configuration

space, like the set of possible orders of cities, or the set of possible allocations of silicon “real estate” blocks to circuit elements. The number of elements in the configuration space is factorially large, so that they cannot be explored exhaustively. Furthermore, since the set is discrete, we are deprived of any notion of “continuing downhill in a favorable direction.” The concept of “direction” may not have any meaning in the configuration space.

Below, we will also discuss how to use simulated annealing methods for spaces with continuous control parameters, like those of §§10.4–10.7. This application is actually more complicated than the combinatorial one, since the familiar problem of “long, narrow valleys” again asserts itself. Simulated annealing, as we will see, tries “random” steps; but in a long, narrow valley, almost all random steps are uphill! Some additional finesse is therefore required.

At the heart of the method of simulated annealing is an analogy with thermodynamics, specifically with the way that liquids freeze and crystallize, or metals cool and anneal. At high temperatures, the molecules of a liquid move freely with respect to one another. If the liquid is cooled slowly, thermal mobility is lost. The atoms are often able to line themselves up and form a pure crystal that is completely ordered over a distance up to billions of times the size of an individual atom in all directions. This crystal is the state of minimum energy for this system. The amazing fact is that, for slowly cooled systems, nature is able to find this minimum energy state. In fact, if a liquid metal is cooled quickly or “quenched,” it does not reach this state but rather ends up in a polycrystalline or amorphous state having somewhat higher energy.

So the essence of the process is *slow* cooling, allowing ample time for redistribution of the atoms as they lose mobility. This is the technical definition of *annealing*, and it is essential for ensuring that a low energy state will be achieved.

Although the analogy is not perfect, there is a sense in which all of the minimization algorithms thus far in this chapter correspond to rapid cooling or quenching. In all cases, we have gone greedily for the quick, nearby solution: From the starting point, go immediately downhill as far as you can go. This, as often remarked above, leads to a local, but not necessarily a global, minimum. Nature’s own minimization algorithm is based on quite a different procedure. The so-called Boltzmann probability distribution,

$$\text{Prob}(E) \sim \exp(-E/kT) \quad (10.9.1)$$

expresses the idea that a system in thermal equilibrium at temperature T has its energy probabilistically distributed among all different energy states E . Even at low temperature, there is a chance, albeit very small, of a system being in a high energy state. Therefore, there is a corresponding chance for the system to get out of a local energy minimum in favor of finding a better, more global, one. The quantity k (Boltzmann’s constant) is a constant of nature that relates temperature to energy. In other words, the system sometimes goes *uphill* as well as downhill; but the lower the temperature, the less likely is any significant uphill excursion.

In 1953, Metropolis and coworkers [5] first incorporated these kinds of principles into numerical calculations. Offered a succession of options, a simulated thermodynamic system was assumed to change its configuration from energy E_1 to energy E_2 with probability $p = \exp[-(E_2 - E_1)/kT]$. Notice that if $E_2 < E_1$, this probability is greater than unity; in such cases the change is arbitrarily assigned a probability $p = 1$, i.e., the system *always* took such an option. This general scheme,

of always taking a downhill step while *sometimes* taking an uphill step, has come to be known as the Metropolis algorithm.

To make use of the Metropolis algorithm for other than thermodynamic systems, one must provide the following elements:

1. A description of possible system configurations.
2. A generator of random changes in the configuration; these changes are the “options” presented to the system.
3. An objective function E (analog of energy) whose minimization is the goal of the procedure.
4. A control parameter T (analog of temperature) and an *annealing schedule* which tells how it is lowered from high to low values, e.g., after how many random changes in configuration is each downward step in T taken, and how large is that step. The meaning of “high” and “low” in this context, and the assignment of a schedule, may require physical insight and/or trial-and-error experiments.

Combinatorial Minimization: The Traveling Salesman

A concrete illustration is provided by the traveling salesman problem. The proverbial seller visits N cities with given positions (x_i, y_i) , returning finally to his or her city of origin. Each city is to be visited only once, and the route is to be made as short as possible. This problem belongs to a class known as *NP-complete* problems, whose computation time for an *exact* solution increases with N as $\exp(\text{const.} \times N)$, becoming rapidly prohibitive in cost as N increases. The traveling salesman problem also belongs to a class of minimization problems for which the objective function E has many local minima. In practical cases, it is often enough to be able to choose from these a minimum which, even if not absolute, cannot be significantly improved upon. The annealing method manages to achieve this, while limiting its calculations to scale as a small power of N .

As a problem in simulated annealing, the traveling salesman problem is handled as follows:

1. *Configuration*. The cities are numbered $i = 1 \dots N$ and each has coordinates (x_i, y_i) . A configuration is a permutation of the number $1 \dots N$, interpreted as the order in which the cities are visited.
2. *Rearrangements*. An efficient set of moves has been suggested by Lin [6]. The moves consist of two types: (a) A section of path is removed and then replaced with the same cities running in the opposite order; or (b) a section of path is removed and then replaced in between two cities on another, randomly chosen, part of the path.
3. *Objective Function*. In the simplest form of the problem, E is taken just as the total length of journey,

$$E = L \equiv \sum_{i=1}^N \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} \quad (10.9.2)$$

with the convention that point $N + 1$ is identified with point 1. To illustrate the flexibility of the method, however, we can add the following additional wrinkle: Suppose that the salesman has an irrational fear of flying over the Mississippi River. In that case, we would assign each city a parameter μ_i , equal to +1 if it is east of the

Mississippi, -1 if it is west, and take the objective function to be

$$E = \sum_{i=1}^N \left[\sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} + \lambda(\mu_i - \mu_{i+1})^2 \right] \quad (10.9.3)$$

A penalty 4λ is thereby assigned to any river crossing. The algorithm now finds the shortest path that avoids crossings. The relative importance that it assigns to length of path versus river crossings is determined by our choice of λ . Figure 10.9.1 shows the results obtained. Clearly, this technique can be generalized to include many conflicting goals in the minimization.

4. *Annealing schedule.* This requires experimentation. We first generate some random rearrangements, and use them to determine the range of values of ΔE that will be encountered from move to move. Choosing a starting value for the parameter T which is considerably larger than the largest ΔE normally encountered, we proceed downward in multiplicative steps each amounting to a 10 percent decrease in T . We hold each new value of T constant for, say, $100N$ reconfigurations, or for $10N$ successful reconfigurations, whichever comes first. When efforts to reduce E further become sufficiently discouraging, we stop.

The following traveling salesman program, using the Metropolis algorithm, illustrates the main aspects of the simulated annealing technique for combinatorial problems.

```

SUBROUTINE anneal(x,y,iorder,ncity)
INTEGER ncity,iorder(ncity)
REAL x(ncity),y(ncity)
C  USES irbit1,metrop,ran3,revcst,revers,trncst,trnspt
    This algorithm finds the shortest round-trip path to ncity cities whose coordinates are in
    the arrays x(1:ncity),y(1:ncity). The array iorder(1:ncity) specifies the order
    in which the cities are visited. On input, the elements of iorder may be set to any per-
    mutation of the numbers 1 to ncity. This routine will return the best alternative path
    it can find.
INTEGER i,i1,i2,idec,idum,iseed,j,k,nlimit,nm,nover,nsucc,n(6),
*   irbit1
REAL de,path,t,tfactr,ran3,alen,x1,x2,y1,y2
LOGICAL ans
alen(x1,x2,y1,y2)=sqrt((x2-x1)**2+(y2-y1)**2)
nover=100*ncity      Maximum number of paths tried at any temperature.
nlimit=10*ncity     Maximum number of successful path changes before continuing.
tfactr=0.9          Annealing schedule: t is reduced by this factor on each step.
path=0.0
t=0.5
do 11 i=1,ncity-1   Calculate initial path length.
    i1=iorder(i)
    i2=iorder(i+1)
    path=path+alen(x(i1),x(i2),y(i1),y(i2))
enddo 11
i1=iorder(ncity)   Close the loop by tying path ends together.
i2=iorder(1)
path=path+alen(x(i1),x(i2),y(i1),y(i2))
idum=-1
iseed=111
do 13 j=1,100      Try up to 100 temperature steps.
    nsucc=0
    do 12 k=1,nover
1      n(1)=1+int(ncity*ran3(idum))      Choose beginning of segment ..
        n(2)=1+int((ncity-1)*ran3(idum)) ..and end of segment.
        if (n(2).ge.n(1)) n(2)=n(2)+1

```

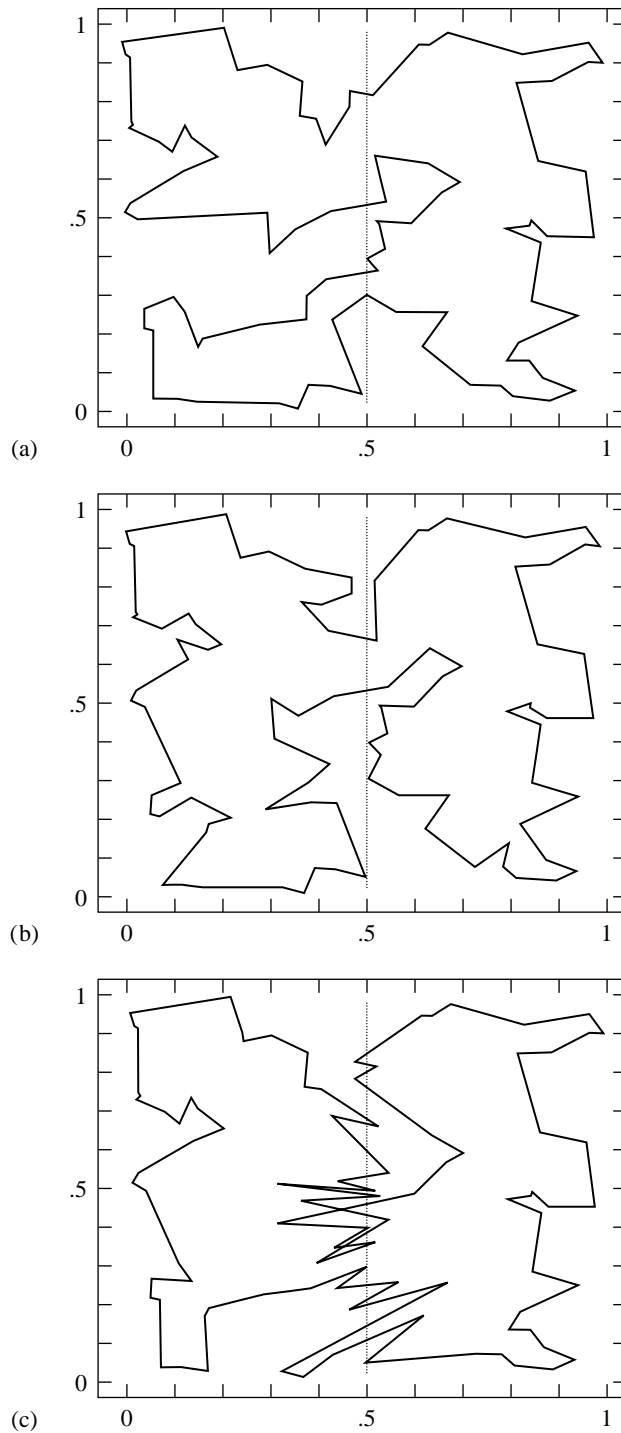


Figure 10.9.1. Traveling salesman problem solved by simulated annealing. The (nearly) shortest path among 100 randomly positioned cities is shown in (a). The dotted line is a river, but there is no penalty in crossing. In (b) the river-crossing penalty is made large, and the solution restricts itself to the minimum number of crossings, two. In (c) the penalty has been made negative: the salesman is actually a smuggler who crosses the river on the flimsiest excuse!

```

nn=1+mod((n(1)-n(2)+ncity-1),ncity)    nn is the number of cities not on the
if (nn.lt.3) goto 1                    segment.
idec=irbit1(iseed)                    Decide whether to do a segment reversal or transport.
if (idec.eq.0) then                    Do a transport.
  n(3)=n(2)+int(abs(nn-2)*ran3(idum))+1
  n(3)=1+mod(n(3)-1,ncity)            Transport to a location not on the path.
  call trncst(x,y,iorder,ncity,n,de)  Calculate cost.
  call metrop(de,t,ans)                Consult the oracle.
  if (ans) then
    nsucc=nsucc+1
    path=path+de
    call trnspt(iorder,ncity,n)        Carry out the transport.
  endif
else
  call revcst(x,y,iorder,ncity,n,de)  Do a path reversal.
  call metrop(de,t,ans)                Calculate cost.
  if (ans) then                        Consult the oracle.
    nsucc=nsucc+1
    path=path+de
    call revers(iorder,ncity,n)        Carry out the reversal.
  endif
endif
if (nsucc.ge.nlimit) goto 2            Finish early if we have enough
enddo 12                                successful changes.
2 write(*,*)
write(*,*) 'T =',t,' Path Length =',path
write(*,*) 'Successful Moves: ',nsucc
t=t*tfactr                             Annealing schedule.
if (nsucc.eq.0) return                 If no success, we are done.
enddo 13
return
END

```

```

SUBROUTINE revcst(x,y,iorder,ncity,n,de)
INTEGER ncity,iorder(ncity),n(6)
REAL de,x(ncity),y(ncity)

```

This subroutine returns the value of the cost function for a proposed path reversal. *ncity* is the number of cities, and arrays *x(1:ncity)*, *y(1:ncity)* give the coordinates of these cities. *iorder(1:ncity)* holds the present itinerary. The first two values *n(1)* and *n(2)* of array *n* give the starting and ending cities along the path segment which is to be reversed. On output, *de* is the cost of making the reversal. The actual reversal is not performed by this routine.

```

INTEGER ii,j
REAL alen,xx(4),yy(4),x1,x2,y1,y2
alen(x1,x2,y1,y2)=sqrt((x2-x1)**2+(y2-y1)**2)
n(3)=1+mod((n(1)+ncity-2),ncity)      Find the city before n(1) ..
n(4)=1+mod(n(2),ncity)                 .. and the city after n(2).
do 11 j=1,4
  ii=iorder(n(j))                       Find coordinates for the four cities involved.
  xx(j)=x(ii)
  yy(j)=y(ii)
enddo 11
de=-alen(xx(1),xx(3),yy(1),yy(3))      Calculate cost of disconnecting the segment
*   -alen(xx(2),xx(4),yy(2),yy(4))      at both ends and reconnecting in the op-
*   +alen(xx(1),xx(4),yy(1),yy(4))      posite order.
*   +alen(xx(2),xx(3),yy(2),yy(3))
return
END

```

```
SUBROUTINE revers(iorder,ncity,n)
INTEGER ncity,iorder(ncity),n(6)
```

This routine performs a path segment reversal. *iorder(1:ncity)* is an input array giving the present itinerary. The vector *n* has as its first four elements the first and last cities *n(1),n(2)* of the path segment to be reversed, and the two cities *n(3)* and *n(4)* that immediately precede and follow this segment. *n(3)* and *n(4)* are found by subroutine *revcst*. On output, *iorder(1:ncity)* contains the segment from *n(1)* to *n(2)* in reversed order.

```
INTEGER itmp,j,k,l,nn
nn=(1+mod(n(2)-n(1)+ncity,ncity))/2
do ii j=1,nn
  k=1+mod((n(1)+j-2),ncity)
  l=1+mod((n(2)-j+ncity),ncity)
  itmp=iorder(k)
  iorder(k)=iorder(l)
  iorder(l)=itmp
enddo ii
return
END
```

This many cities must be swapped to effect the reversal.

Start at the ends of the segment and swap pairs of cities, moving toward the center.

```
SUBROUTINE trncst(x,y,iorder,ncity,n,de)
INTEGER ncity,iorder(ncity),n(6)
REAL de,x(ncity),y(ncity)
```

This subroutine returns the value of the cost function for a proposed path segment transport. *ncity* is the number of cities, and arrays *x(1:ncity)* and *y(1:ncity)* give the city coordinates. *iorder* is an array giving the present itinerary. The first three elements of array *n* give the starting and ending cities of the path to be transported, and the point among the remaining cities after which it is to be inserted. On output, *de* is the cost of the change. The actual transport is not performed by this routine.

```
INTEGER ii,j
REAL xx(6),yy(6),alen,x1,x2,y1,y2
alen(x1,x2,y1,y2)=sqrt((x2-x1)**2+(y2-y1)**2)
n(4)=1+mod(n(3),ncity)
n(5)=1+mod((n(1)+ncity-2),ncity)
n(6)=1+mod(n(2),ncity)
do ii j=1,6
  ii=iorder(n(j))
  xx(j)=x(ii)
  yy(j)=y(ii)
enddo ii
de=-alen(xx(2),xx(6),yy(2),yy(6))
* -alen(xx(1),xx(5),yy(1),yy(5))
* -alen(xx(3),xx(4),yy(3),yy(4))
* +alen(xx(1),xx(3),yy(1),yy(3))
* +alen(xx(2),xx(4),yy(2),yy(4))
* +alen(xx(5),xx(6),yy(5),yy(6))
return
END
```

Find the city following *n(3)*..
..and the one preceding *n(1)*..
..and the one following *n(2)*.

Determine coordinates for the six cities involved.

Calculate the cost of disconnecting the path segment from *n(1)* to *n(2)*, opening a space between *n(3)* and *n(4)*, connecting the segment in the space, and connecting *n(5)* to *n(6)*.

```
SUBROUTINE trnspt(iorder,ncity,n)
INTEGER ncity,iorder(ncity),n(6),MXCITY
PARAMETER (MXCITY=1000)
```

Maximum number of cities anticipated.

This routine does the actual path transport, once *metrop* has approved. *iorder* is an input array of length *ncity* giving the present itinerary. The array *n* has as its six elements the beginning *n(1)* and end *n(2)* of the path to be transported, the adjacent cities *n(3)* and *n(4)* between which the path is to be placed, and the cities *n(5)* and *n(6)* that precede and follow the path. *n(4)*, *n(5)*, and *n(6)* are calculated by subroutine *trncst*. On output, *iorder* is modified to reflect the movement of the path segment.

```
INTEGER j,jj,m1,m2,m3,nn,jorder(MXCITY)
m1=1+mod((n(2)-n(1)+ncity),ncity)
```

Find number of cities from *n(1)* to *n(2)*

```

m2=1+mod((n(5)-n(4)+ncity),ncity)      ...and the number from n(4) to n(5)
m3=1+mod((n(3)-n(6)+ncity),ncity)      ...and the number from n(6) to n(3).
nn=1
do 11 j=1,m1
  jj=1+mod((j+n(1)-2),ncity)           Copy the chosen segment.
  jorder(nn)=iorder(jj)
  nn=nn+1
enddo 11
do 12 j=1,m2
  jj=1+mod((j+n(4)-2),ncity)           Then copy the segment from n(4) to n(5).
  jorder(nn)=iorder(jj)
  nn=nn+1
enddo 12
do 13 j=1,m3
  jj=1+mod((j+n(6)-2),ncity)           Finally, the segment from n(6) to n(3).
  jorder(nn)=iorder(jj)
  nn=nn+1
enddo 13
do 14 j=1,ncity
  iorder(j)=jorder(j)                  Copy jorder back into iorder.
enddo 14
return
END

```

```

SUBROUTINE metrop(de,t,ans)

```

```

REAL de,t

```

```

LOGICAL ans

```

```

C USES ran3

```

Metropolis algorithm. `ans` is a logical variable that issues a verdict on whether to accept a reconfiguration that leads to a change `de` in the objective function `e`. If `de < 0`, `ans = .true.`, while if `de > 0`, `ans` is only `.true.` with probability $\exp(-de/t)$, where `t` is a temperature determined by the annealing schedule.

```

INTEGER jdum

```

```

REAL ran3

```

```

SAVE jdum

```

```

DATA jdum /1/

```

```

ans=(de.lt.0.0).or.(ran3(jdum).lt.exp(-de/t))

```

```

return

```

```

END

```

Continuous Minimization by Simulated Annealing

The basic ideas of simulated annealing are also applicable to optimization problems with continuous N -dimensional control spaces, e.g., finding the (ideally, global) minimum of some function $f(\mathbf{x})$, in the presence of many local minima, where \mathbf{x} is an N -dimensional vector. The four elements required by the Metropolis procedure are now as follows: The value of f is the objective function. The system state is the point \mathbf{x} . The control parameter T is, as before, something like a temperature, with an annealing schedule by which it is gradually reduced. And there must be a generator of random changes in the configuration, that is, a procedure for taking a random step from \mathbf{x} to $\mathbf{x} + \Delta\mathbf{x}$.

World Wide Web sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
 Copyright (C) 1988-1992 by Cambridge University Press. Programs Copyright (C) 1988-1992 by Numerical Recipes Software.
 Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to trade@cup.cam.ac.uk (outside North America).

The last of these elements is the most problematical. The literature to date [7-10] describes several different schemes for choosing $\Delta\mathbf{x}$, none of which, in our view, inspire complete confidence. The problem is one of efficiency: A generator of random changes is inefficient if, *when local downhill moves exist*, it nevertheless almost always proposes an uphill move. A good generator, we think, should not become inefficient in narrow valleys; nor should it become more and more inefficient as convergence to a minimum is approached. Except possibly for [7], all of the schemes that we have seen are inefficient in one or both of these situations.

Our own way of doing simulated annealing minimization on continuous control spaces is to use a modification of the downhill simplex method (§10.4). This amounts to replacing the single point \mathbf{x} as a description of the system state by a simplex of $N + 1$ points. The “moves” are the same as described in §10.4, namely reflections, expansions, and contractions of the simplex. The implementation of the Metropolis procedure is slightly subtle: We *add* a positive, logarithmically distributed random variable, proportional to the temperature T , to the stored function value associated with every vertex of the simplex, and we *subtract* a similar random variable from the function value of every new point that is tried as a replacement point. Like the ordinary Metropolis procedure, this method always accepts a true downhill step, but sometimes accepts an uphill one. In the limit $T \rightarrow 0$, this algorithm reduces exactly to the downhill simplex method and converges to a local minimum.

At a finite value of T , the simplex expands to a scale that approximates the size of the region that can be reached at this temperature, and then executes a stochastic, tumbling Brownian motion within that region, sampling new, approximately random, points as it does so. The efficiency with which a region is explored is independent of its narrowness (for an ellipsoidal valley, the ratio of its principal axes) and orientation. If the temperature is reduced sufficiently slowly, it becomes highly likely that the simplex will shrink into that region containing the lowest relative minimum encountered.

As in all applications of simulated annealing, there can be quite a lot of problem-dependent subtlety in the phrase “sufficiently slowly”; success or failure is quite often determined by the choice of annealing schedule. Here are some possibilities worth trying:

- Reduce T to $(1 - \epsilon)T$ after every m moves, where ϵ/m is determined by experiment.
- Budget a total of K moves, and reduce T after every m moves to a value $T = T_0(1 - k/K)^\alpha$, where k is the cumulative number of moves thus far, and α is a constant, say 1, 2, or 4. The optimal value for α depends on the statistical distribution of relative minima of various depths. Larger values of α spend more iterations at lower temperature.
- After every m moves, set T to β times $f_1 - f_b$, where β is an experimentally determined constant of order 1, f_1 is the smallest function value currently represented in the simplex, and f_b is the best function ever encountered. However, never reduce T by more than some fraction γ at a time.

Another strategic question is whether to do an occasional *restart*, where a vertex of the simplex is discarded in favor of the “best-ever” point. (You must be sure that the best-ever point is not currently in the simplex when you do this!) We have found problems for which restarts — every time the temperature has decreased by a factor of 3, say — are highly beneficial; we have found other problems for which restarts

have no positive, or a somewhat negative, effect.

You should compare the following routine, `amebsa`, with its counterpart `amoeba` in §10.4. Note that the argument `iter` is used in a somewhat different manner.

```
SUBROUTINE ameba(p,y,mp,np,ndim,pb,yb,ftol,funk,iter,temptr)
INTEGER iter,mp,ndim,np,NMAX
REAL ftol,temptr,yb,p(mp,np),pb(np),y(mp),funk
PARAMETER (NMAX=200)
EXTERNAL funk
```

C *USES amotsa, funk, ran1*

Multidimensional minimization of the function `funk(x)` where `x(1:ndim)` is a vector in `ndim` dimensions, by simulated annealing combined with the downhill simplex method of Nelder and Mead. The input matrix `p(1..ndim+1,1..ndim)` has `ndim+1` rows, each an `ndim`-dimensional vector which is a vertex of the starting simplex. Also input is the vector `y(1:ndim+1)`, whose components must be pre-initialized to the values of `funk` evaluated at the `ndim+1` vertices (rows) of `p`; `ftol`, the fractional convergence tolerance to be achieved in the function value for an early return; `iter`, and `temptr`. The routine makes `iter` function evaluations at an annealing temperature `temptr`, then returns. You should then decrease `temptr` according to your annealing schedule, reset `iter`, and call the routine again (leaving other arguments unaltered between calls). If `iter` is returned with a positive value, then early convergence and return occurred. If you initialize `yb` to a very large value on the first call, then `yb` and `pb(1:ndim)` will subsequently return the best function value and point ever encountered (even if it is no longer a point in the simplex).

```
INTEGER i,idum,ihi,ilo,inhi,j,m,n
```

```
REAL rtol,sum,swap,tt,yhi,ylo,ynhi,ysave,yt,ytry,psum(NMAX),
* amotsa,ran1
```

```
COMMON /amebsa/ tt,idum
```

```
tt=-temptr
```

```
1 do 12 n=1,ndim Enter here when starting or after overall contraction.
  sum=0. Recompute psum.
```

```
  do 11 m=1,ndim+1
    sum=sum+p(m,n)
  enddo 11
  psum(n)=sum
```

```
enddo 12
```

```
2 ilo=1 Enter here after changing a single point. Determine which
  inhi=1 point is the highest (worst), next-highest, and lowest
  ihi=2 (best).
```

```
  ylo=y(1)+tt*log(ran1(idum)) Whenever we "look at" a vertex, it gets a random thermal
  ynhi=ylo fluctuation.
```

```
  yhi=y(2)+tt*log(ran1(idum))
  if (ylo.gt.yhi) then
    ihi=1
    inhi=2
    ilo=2
    ynhi=yhi
    yhi=ylo
    ylo=ynhi
```

```
endif
```

```
do 13 i=3,ndim+1 Loop over the points in the simplex.
  yt=y(i)+tt*log(ran1(idum)) More thermal fluctuations.
```

```
  if (yt.le.ylo) then
    ilo=i
    ylo=yt
```

```
endif
```

```
  if (yt.gt.yhi) then
    inhi=ihl
    ynhi=yhi
    ihi=i
    yhi=yt
```

```
  else if (yt.gt.ynhi) then
    inhi=i
    ynhi=yt
```

```

endif
enddo 13
rtol=2.*abs(yhi-ylo)/(abs(yhi)+abs(ylo))
  Compute the fractional range from highest to lowest and return if satisfactory.
if (rtol.lt.ftol.or.iter.lt.0) then    If returning, put best point and value in slot 1.
  swap=y(1)
  y(1)=y(ilo)
  y(ilo)=swap
  do 14 n=1,ndim
    swap=p(1,n)
    p(1,n)=p(ilo,n)
    p(ilo,n)=swap
  enddo 14
  return
endif
iter=iter-2
  Begin a new iteration. First extrapolate by a factor -1 through the face of the simplex across
  from the high point, i.e., reflect the simplex from the high point.
ytry=amotsa(p,y,psum,mp,np,ndim,pb,yb,funk,ihl,yhi,-1.0)
if (ytry.le.ylo) then
  Gives a result better than the best point, so try an additional extrapolation by a factor 2.
  ytry=amotsa(p,y,psum,mp,np,ndim,pb,yb,funk,ihl,yhi,2.0)
else if (ytry.ge.ynhi) then
  The reflected point is worse than the second-highest, so look for an intermediate lower point,
  i.e., do a one-dimensional contraction.
  ysave=yhi
  ytry=amotsa(p,y,psum,mp,np,ndim,pb,yb,funk,ihl,yhi,0.5)
  if (ytry.ge.ysave) then    Can't seem to get rid of that high point. Better contract
    do 16 i=1,ndim+1        around the lowest (best) point.
      if(i.ne.ilo)then
        do 15 j=1,ndim
          psum(j)=0.5*(p(i,j)+p(ilo,j))
          p(i,j)=psum(j)
        enddo 15
        y(i)=funk(psum)
      endif
    enddo 16
    iter=iter-ndim
    goto 1
  endif
else
  iter=iter+1                Correct the evaluation count.
endif
goto 2
END

```

```

FUNCTION amotsa(p,y,psum,mp,np,ndim,pb,yb,funk,ihl,yhi,fac)
INTEGER ihl,mp,ndim,np,NMAX
REAL amotsa,fac,yb,yhi,p(mp,np),pb(np),psum(np),y(mp),funk
PARAMETER (NMAX=200)
EXTERNAL funk

```

C *USES funk,ran1*

```

  Extrapolates by a factor fac through the face of the simplex across from the high point,
  tries it, and replaces the high point if the new point is better.
INTEGER idum,j
REAL fac1,fac2,tt,yflu,ytry,ptry(NMAX),ran1
COMMON /ambsa/ tt,idum
fac1=(1.-fac)/ndim
fac2=fac1-fac
do 11 j=1,ndim
  ptry(j)=psum(j)*fac1-p(ihl,j)*fac2
enddo 11

```

```

ytry=funk(ptry)
if (ytry.le.yb) then          Save the best-ever.
  do 12 j=1,ndim
    pb(j)=ptry(j)
  enddo 12
  yb=ytry
endif
yflu=ytry-tt*log(ran1(idum))  We added a thermal fluctuation to all the current vertices,
if (yflu.lt.yhi) then        but we subtract it here, so as to give the simplex
  y(ihi)=ytry                a thermal Brownian motion: It likes to accept any
  yhi=yflu                   suggested change.
  do 13 j=1,ndim
    psum(j)=psum(j)-p(ihi,j)+ptry(j)
    p(ihi,j)=ptry(j)
  enddo 13
endif
amotsa=yflu
return
END

```

There is not yet enough practical experience with the method of simulated annealing to say definitively what its future place among optimization methods will be. The method has several extremely attractive features, rather unique when compared with other optimization techniques.

First, it is not “greedy,” in the sense that it is not easily fooled by the quick payoff achieved by falling into unfavorable local minima. Provided that sufficiently general reconfigurations are given, it wanders freely among local minima of depth less than about T . As T is lowered, the number of such minima qualifying for frequent visits is gradually reduced.

Second, configuration decisions tend to proceed in a logical order. Changes that cause the greatest energy differences are sifted over when the control parameter T is large. These decisions become more permanent as T is lowered, and attention then shifts more to smaller refinements in the solution. For example, in the traveling salesman problem with the Mississippi River twist, if λ is large, a decision to cross the Mississippi only twice is made at high T , while the specific routes on each side of the river are determined only at later stages.

The analogies to thermodynamics may be pursued to a greater extent than we have done here. Quantities analogous to specific heat and entropy may be defined, and these can be useful in monitoring the progress of the algorithm towards an acceptable solution. Information on this subject is found in [1].

CITED REFERENCES AND FURTHER READING:

- Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P. 1983, *Science*, vol. 220, pp. 671–680. [1]
 Kirkpatrick, S. 1984, *Journal of Statistical Physics*, vol. 34, pp. 975–986. [2]
 Vecchi, M.P. and Kirkpatrick, S. 1983, *IEEE Transactions on Computer Aided Design*, vol. CAD-2, pp. 215–222. [3]
 Otten, R.H.J.M., and van Ginneken, L.P.P.P. 1989, *The Annealing Algorithm* (Boston: Kluwer) [contains many references to the literature]. [4]
 Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller A., and Teller, E. 1953, *Journal of Chemical Physics*, vol. 21, pp. 1087–1092. [5]
 Lin, S. 1965, *Bell System Technical Journal*, vol. 44, pp. 2245–2269. [6]
 Vanderbilt, D., and Louie, S.G. 1984, *Journal of Computational Physics*, vol. 56, pp. 259–271. [7]
 Bohachevsky, I.O., Johnson, M.E., and Stein, M.L. 1986, *Technometrics*, vol. 28, pp. 209–217. [8]

- Corana, A., Marchesi, M., Martini, C., and Ridella, S. 1987, *ACM Transactions on Mathematical Software*, vol. 13, pp. 262–280. [9]
- Bélisle, C.J.P., Romeijn, H.E., and Smith, R.L. 1990, Technical Report 90–25, Department of Industrial and Operations Engineering, University of Michigan, submitted to *Mathematical Programming*. [10]
- Christofides, N., Mingozi, A., Toth, P., and Sandi, C. (eds.) 1979, *Combinatorial Optimization* (London and New York: Wiley-Interscience) [not simulated annealing, but other topics and algorithms].