NUMERICAL RECIPES

Webnote No. 15, Rev. 1

Code for Minimization with Simulated Annealing

You should compare the following routine, `Amebsa`, with its counterpart `Amoeba` in §10.5. Note that the argument `iter` is used in a somewhat different manner. Just like `Amoeba`, this routine also has three different user interfaces. The simplest requires you to specify the starting simplex as in equation (10.5.1):

```
VecDoub point = ...; Doub del = ...;
Amebsa<Doub(VecDoub_I &)> amebsa(point,del,funk,ftol);
Int iter = ...; Doub temperature = ...;
Bool converged=amebsa.anneal(iter,temptr);
```

In practice you loop over calls to `anneal` with an annealing schedule that resets `temperature` and possibly `iter` on each call. You test `converged` to see whether the convergence criterion has been met. The final minimum point is available in `amebsa.pb` and the value at the minimum in `amebsa.yb`.

The second interface allows you to specify the starting simplex with a vector of increments Δ_i:

```
VecDoub dels = ...;
Amebsa<Doub(VecDoub_I &)> amebsa(point,dels,funk,ftol);
```

The most general interface lets you specify the simplex as an $(N+1) \times N$ matrix whose rows are the coordinates of each vertex:

```
MatDoub p = ...;
Amebsa<Doub(VecDoub_I &)> amebsa(p,funk,ftol);
```

Here is the routine:

amebsa.h

```
template <class T>
struct Amebsa {
```

Multidimensional minimization by simulated annealing combined with the downhill simplex method of Nelder and Mead.

```
    T &funk;
    const Doub ftol;
    Ranq1 ran;
    Doub yb;                    Function value at the minimum.
    Int ndim;
    VecDoub pb;                 Minimum point.
    Int mpts;
    VecDoub y;                  Function values at the vertices of the simplex.
    MatDoub p;                  Current simplex.
    Doub tt;                    Communicates annealing temperature to amotsa.
    Amebsa(VecDoub_I &point, const Doub del, T &funkk, const Doub ftoll) :
        funk(funkk), ftol(ftoll), ran(1234),
        yb(numeric_limits<Doub>::max()), ndim(point.size()), pb(ndim),
        mpts(ndim+1), y(mpts), p(mpts,ndim)
```

Constructor specifying the initial simplex as in equation (10.5.1) by a point[0..ndim-1] and a constant displacement del along each coordinate direction. Other arguments are ftoll, the fractional convergence tolerance to be achieved in the function value for an early return, and funkk, the function or functor to be minimized. The constructor also initializes the random number generator and sets yb to a very large value.

```
{
    for (Int i=0;i<mpts;i++) {
        for (Int j=0;j<ndim;j++)
            p[i][j]=point[j];
        if (i != 0) p[i][i-1] += del;
    }
    inity();
}
Amebsa(VecDoub_I &point, VecDoub_I &dels, T &funkk, const Doub ftoll) :
    funk(funkk), ftol(ftoll), ran(1234),
    yb(numeric_limits<Doub>::max()), ndim(point.size()), pb(ndim),
    mpts(ndim+1), y(mpts), p(mpts,ndim)
```

Alternative constructor that takes different displacements dels[0..ndim-1] in different directions for the initial simplex.

```
{
    for (Int i=0;i<mpts;i++) {
        for (Int j=0;j<ndim;j++)
            p[i][j]=point[j];
        if (i != 0) p[i][i-1] += dels[i-1];
    }
    inity();
}
Amebsa(MatDoub_I &pp, T &funkk, const Doub ftoll) : funk(funkk),
    ftol(ftoll), ran(1234), yb(numeric_limits<Doub>::max()),
    ndim(pp.ncols()), pb(ndim), mpts(pp.nrows()), y(mpts), p(pp)
```

Most general constructor: the initial simplex is specified by the matrix pp[0..ndim][0..ndim-1]. Its ndim+1 rows are ndim-dimensional vectors that are the vertices of the starting simplex.

```
{ inity(); }

void inity() {
```

Initialize y to the values of funk at the vertices of p.

```
    VecDoub x(ndim);
    for (Int i=0;i<mpts;i++) {
        for (Int j=0;j<ndim;j++)
            x[j]=p[i][j];
        y[i]=funk(x);
    }
}
Bool anneal(Int &iter, const Doub temperature)
```

Performs the minimization. The routine makes iter function evaluations at an annealing temperature temperature, then returns. You should then decrease temperature according to your annealing schedule, reset iter, and call the routine again (leaving its internal state unaltered between calls). If converged is returned as true, then the required tolerance ftol on the function value at the minimum has been achieved. In this case, iter is returned with a positive value reflecting the number of unused iterations on this pass. At the end of the annealing, yb and pb[0..ndim-1] will contain the best function value and point ever encountered (even if it is no longer a point in the simplex).

```
{
    VecDoub psum(ndim);
    tt = -temperature;
    get_psum(p,psum);
    for (;;) {
        Int ilo=0;
        Int ihi=1;
        Doub ylo=y[0]+tt*log(ran.doub());
        Doub ynhi=ylo;
        Doub yhi=y[1]+tt*log(ran.doub());
        if (ylo > yhi) {
            ihi=0;
```

Determine which point is the highest (worst), next-highest, and lowest (best).

Whenever we "look at" a vertex, it gets a random thermal fluctuation.

```
            ilo=1;
            ynhi=yhi;
            yhi=ylo;
            ylo=ynhi;
        }
        for (Int i=3;i<=mpts;i++) {          Loop over the points in the simplex.
            Doub yt=y[i-1]+tt*log(ran.doub());    More thermal fluctuations.
            if (yt <= ylo) {
                ilo=i-1;
                ylo=yt;
            }
            if (yt > yhi) {
                ynhi=yhi;
                ihi=i-1;
                yhi=yt;
            } else if (yt > ynhi) {
                ynhi=yt;
            }
        }
        Doub rtol=2.0*abs(yhi-ylo)/(abs(yhi)+abs(ylo));
```

Compute the fractional range from highest to lowest and return if satisfactory.

```
        if (rtol < ftol || iter < 0) {       If returning, put best point and value in
            SWAP(y[0],y[ilo]);                   slot 0.
            for (Int n=0;n<ndim;n++)
                SWAP(p[0][n],p[ilo][n]);
            if (rtol < ftol)
                return true;
            else
                return false;
        }
        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.

```
        Doub ytry=amotsa(p,y,psum,ihi,yhi,-1.0);
        if (ytry <= ylo) {
```

Gives a result better than the best point, so try an additional extrapolation by a factor of 2.

```
            ytry=amotsa(p,y,psum,ihi,yhi,2.0);
        } else if (ytry >= ynhi) {
```

The reflected point is worse than the second-highest, so look for an intermediate lower point, i.e., do a one-dimensional contraction.

```
            Doub ysave=yhi;
            ytry=amotsa(p,y,psum,ihi,yhi,0.5);
            if (ytry >= ysave) {             Can't seem to get rid of that high point.
                for (Int i=0;i<mpts;i++) {       Better contract around the lowest
                    if (i != ilo) {              (best) point.
                        for (Int j=0;j<ndim;j++) {
                            psum[j]=0.5*(p[i][j]+p[ilo][j]);
                            p[i][j]=psum[j];
                        }
                        y[i]=funk(psum);
                    }
                }
                iter -= ndim;
                get_psum(p,psum);            Recompute psum.
            }
        } else ++iter;                       Correct the evaluation count.
    }
}
inline void get_psum(MatDoub_I &p, VecDoub_O &psum)
```

Utility function.

```
{
    for (Int n=0;n<ndim;n++) {
        Doub sum=0.0;
```

```
            for (Int m=0;m<mpts;m++) sum += p[m][n];
            psum[n]=sum;
        }
    }
    Doub amotsa(MatDoub_IO &p, VecDoub_O &y, VecDoub_IO &psum,
        const Int ihi, Doub &yhi, const Doub fac)
```

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.

```
    {
        VecDoub ptry(ndim);
        Doub fac1=(1.0-fac)/ndim;
        Doub fac2=fac1-fac;
        for (Int j=0;j<ndim;j++)
            ptry[j]=psum[j]*fac1-p[ihi][j]*fac2;
        Doub ytry=funk(ptry);
        if (ytry <= yb) {                    Save the best-ever.
            for (Int j=0;j<ndim;j++) pb[j]=ptry[j];
            yb=ytry;
        }
        Doub yflu=ytry-tt*log(ran.doub());   We added a thermal fluctuation to all
        if (yflu < yhi) {                        the current vertices, but we subtract
            y[ihi]=ytry;                         it here, so as to give the simplex a
            yhi=yflu;                            thermal Brownian motion: It likes
            for (Int j=0;j<ndim;j++) {           to accept any suggested change.
                psum[j] += ptry[j]-p[ihi][j];
                p[ihi][j]=ptry[j];
            }
        }
        return yflu;
    }
};
```