Function minimization in N dimensions

(1) Downhill simplex, a.k.a. amoeba (*NR* §10.4) OCTAVE & MATLAB: fminsearch( )
- O(N^2) storage ⇒ UNSUITABLE FOR VERY LARGE PROBLEMS!
- robust, simple (NO DERIVATIVES USED; CAN TOLERATE DISCONTINUITIES)
- INTUITIVE. MODEST CONVERGENCE RATE.

(2) Conjugate gradient (*NR* §10.6)
- Uses Brent's method (*NR* §10.3) in 1-D
- O(N) storage ==> large problems
- fast/efficient
- requires derivatives (==> smoothness assumed)

(3) Simulated annealing (*NR* §10.9)
- most robust and simplest of algorithms!
- often finds *global* minima, even of rough functions
- O(N) storage ==> large problems
- slow as molasses in January
- DOESN'T REQUIRE THAT $(\vec{x})$ BE A CONTINUOUS VARIABLE!

C. C. Kankelborg, 1999-11-18 (rev. 2009-Mar-24)

Multidimensional Minimization

I. INTRODUCTION

This lecture is devoted to the task of *minimization* in N dimensions, which may be stated as follows:

For a scalar function $f(\mathbf{x})$, *where* $\mathbf{x} = [x_1, x_2, ..., x_N]$, *find* $\mathbf{x}$ *that minimizes* $f(\mathbf{x})$.

Many challenging problems in physics, engineering, operations research, economics, *etc.* require carefully choosing the values of a set of parameters ($\mathbf{x}$) to achieve some optimal result. If what is meant by "optimal" can be quantified — reduced to a mere number via some function $f(\mathbf{x})$ — then this N-dimensional *optimization* problem may be reduced to minimization as defined above. Applications of this technique range from experimental tests of scientific theories to practical design problems to questions of international trade policy.

Minimization is difficult. In 2 or more dimensions, it is not possible to "bracket" a minimum. Convergence cannot be guaranteed. It is impossible to guarantee that a minimum, once found, is the global minimum of the function.

The design of minimization algorithms is part deduction and part intuition. The programs used for minimizing tend to be finicky about which problems they will solve. In this handout I'll describe the *downhill simplex* method ("amoeba") which is the basis of MATLAB's `fminsearch`. My own amoeba implementation is also included with an example.

II. THE AMOEBA METHOD

Nelder and Meade in 1965 invented one of the simplest and most robust minimization methods. It is called the downhill *simplex*, or simply the *amoeba*. Not all problems are susceptible to the amoeba, but surprisingly many are. A simplex, or amoeba, is like a little animal with N+1 feet crawling around in a hot world, looking for a cool, comfortable spot. The feet move around in response to the local "temperature", $f(\mathbf{x})$.

Whether N is large or small, the amoeba utilizes the value of f at only three of its feet at any one time: the hottest (P), the second hottest (Q), and

the coolest (R). The basic moves used by the amoeba are summarized in figure 1, which is simplified by assuming a two-dimensional space ($N = 2$, so the amoeba has 3 feet). The main loop of the program always begins by "reflecting" point P through the opposite face of the amoeba, as in [*a*]. The remaining moves are chosen (or not) depending on how f(P) has changed as compared to f(Q) and f(R). Figure 2 is a flowchart summarizing the tactics that lead from one move to the next. The choice of which points are called P, Q, and R is revised after each cycle through the main loop. *[Note: the flowchart includes a convergence criterion, a decision box that points to END when satisfied. The idea here is that if the highest and lowest points are close enough to one another, then the amoeba has located the minimum to reasonable precision. I don't use a convergence criterion for the examples. It is often sufficient to just run through the main loop a fixed number of times and examine the results visually.]*

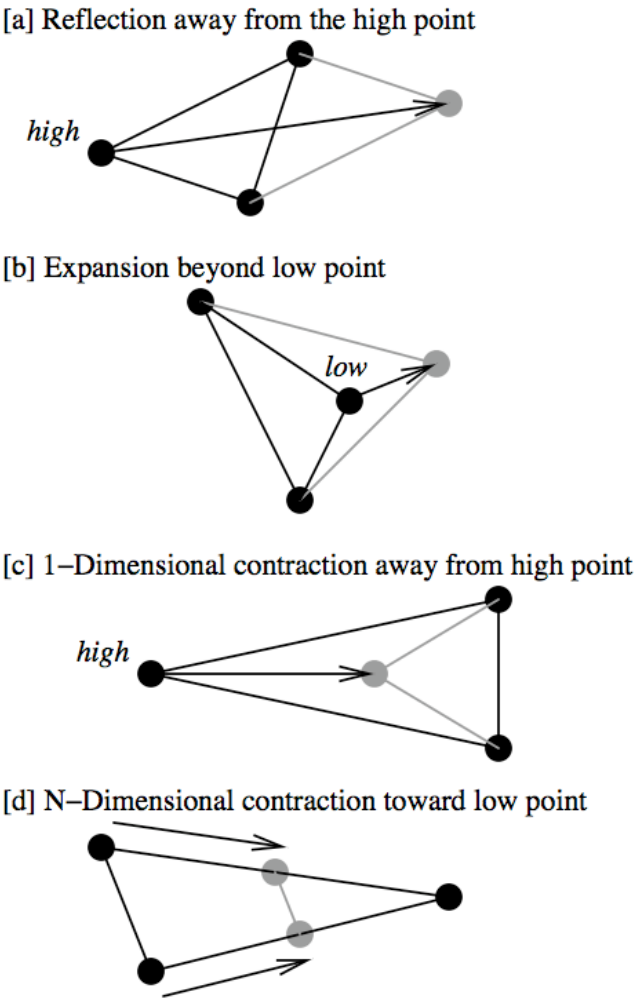

Fig. 1. The basic moves of the amoeba.

For a two-dimensional problem, the amoeba's "feet" consist of three points forming a triangle. The three feet are then just P, Q, and R. Figure 3 shows two example sequences of moves in two dimensions. These are best followed while looking at the flowchart. Following is a short description of each sequence moves, identified by the letters [a], [b], [c], or [d] of Figure 1.

Sequence A. A straightforward example. A1:[a],[b]. The [b] move is risky, but it's a good way of capitalizing on a reflection that turned out very well. A2: [a]. A3:[a],[c]. Note how the last move pulls back from an excursion that almost left the page. The amoeba doesn't have eyes, only feet to feel with, and so it can be fooled for a little while. But overall, it tends to make progress toward a minimum.

Sequence B. This is a situation contrived to "fool" the amoeba, but only for a little while. B1: [a],[c]. This first panel is like A3, a move that turned out to be a bad idea. But after pulling point P closer to the center, the value of f gets even higher! You can see how I had to draw distorted contour lines to make this possible. Real minimization problems are full of such cruel tricks. B2:[d]. In the d-type move, the amoeba moves *N* of its feet at once (instead of just one, as in a, b, c-type moves); the amoeba shrinks around the coolest foot as a last resort, because every other stratagem has gone bad.

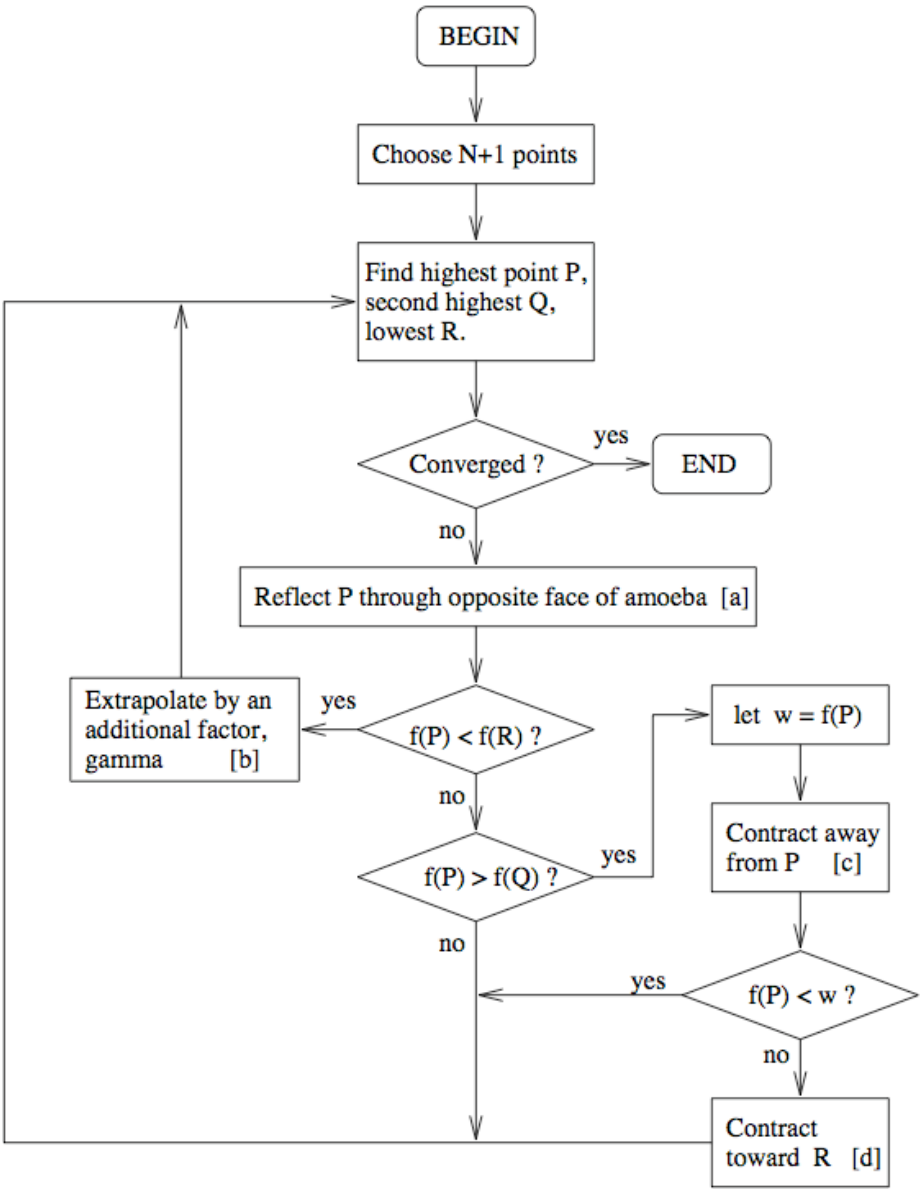

Fig. 2. Amoeba flowchart.

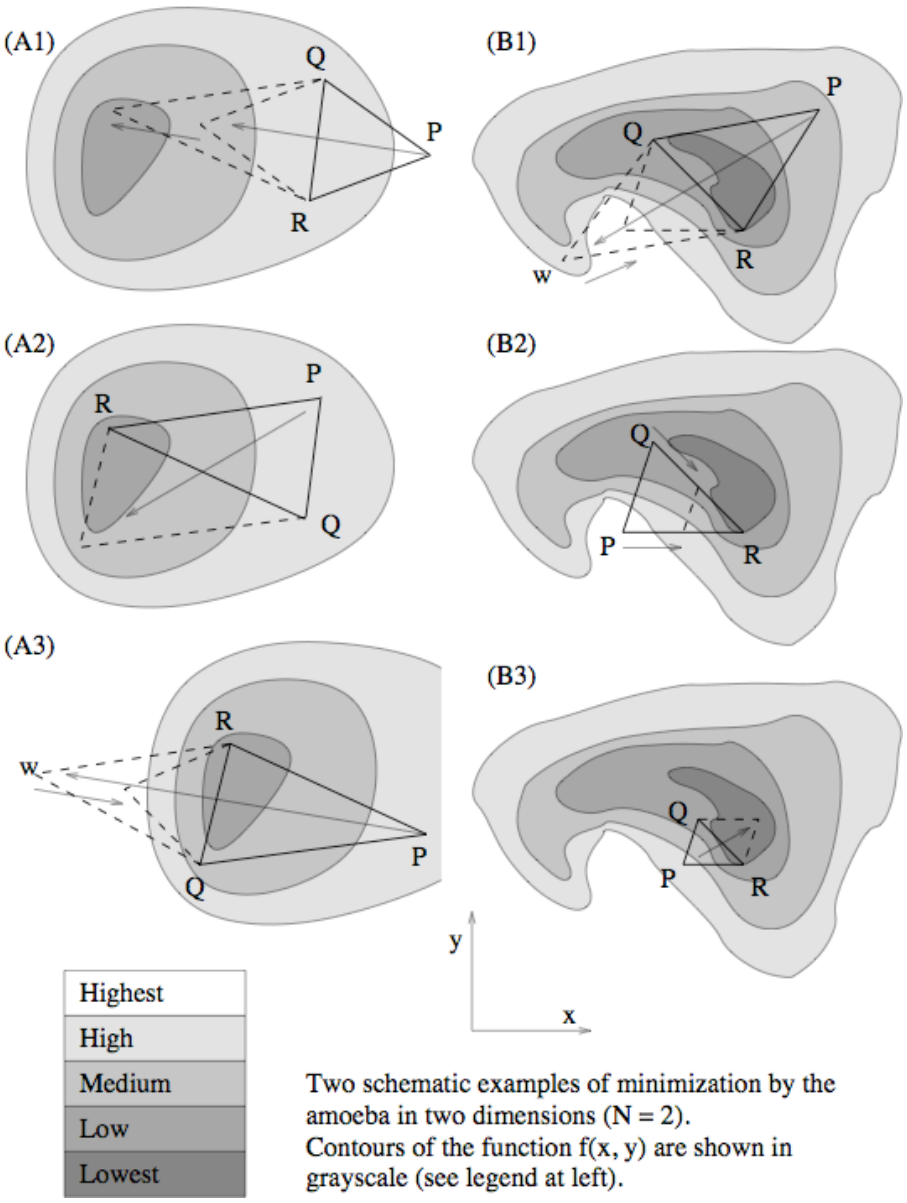

Fig. 3. Sketched examples showing the behavior of the amoeba.

III. SAMPLE PROGRAMS

I will put the .m files on the web in case you want to play with them:

http://solar.physics.montana.edu/kankel/phys567/examples/octave/minimization/amoeba

3.1 Amoeba code

Here is `amoeba.m`. The idea is simple: you give it an initial guess for **x**, and the name of *funk*, a program that returns *f*(**x**). You also tell the amoeba how many *iterations* to go through the main loop. The result is what amoeba

thinks is the value of **x** that minimizes *funk*. Note that the source code is divided into four modules: the main program (amoeba), a sorter to choose P, Q, and R (pickPQR), an implementation of moves [a, b, c] (reflect_foot), and an implementation of move [d] (ndcontract).

```
function bestfit=amoeba(funk,xguess,iterations)
% A simple implementation of the downhill simplex method of
% Nelder & Meade (1963), a.k.a. "amoeba". The function funk
% should be a scalar, real-valued function of a vector argument
% of the same size as xguess.

%Standard scale values for the reflect function
reflect = 1;
expand = -0.5;
contract  = 0.25;

[M,N] = size(xguess); %N will be the number of dimensions.
if M~=1
   error('xguess is not a row vector!')
end

%Define the N+1 'feet' of the amoeba
basis = eye(N);
feet = [xguess;ones(N,1)*xguess+basis];

%Evaluate funk at each of the 'feet'
f=zeros(1,N+1);
for i=1:N+1
   f(i)=feval(funk,feet(i,:));
end

for persistence=1:iterations %The main loop
   [P,Q,R] = pickPQR(f); %Identify highest, second highest, lowest feet
   [feet,f]=reflect_foot(funk,feet,f,P,reflect); %Reflect
   if f(P) < f(R)
      [feet,f]=reflect_foot(funk,feet,f,P,expand); %Expand
   elseif f(P) > f(Q)
      w = f(P); %Keep track of the current worst value of f
      [feet,f]=reflect_foot(funk,feet,f,P,contract); %1-dim Contract
      if f(P) < w
         [feet,f]=ndcontract(funk,feet,f,R); %N-dim contract
      end
   end
end
```

```
[P,Q,R] = pickPQR(f); %Identify highest, second highest, lowest feet
bestfit = feet(R,:); %Use lowest foot as best fit.
%end amoeba
%---------------------------------------------------------------------

function [P,Q,R]=pickPQR(f)
% Identify indices of highest (P), second highest (Q),
% and lowest (R) feet.

[foo,Nfeet]=size(f);
if foo ~=1
   error('f has wrong dimensions!')
end

P=1; Q=2; R=1; % Initial guess, certainly wrong
if f(Q) > f(P) % Correct the P/Q order for first 2 feet
   P=2; Q=1;
end
for i=1:Nfeet  % Loop thru feet, finding P,Q,R
   if f(i) > f(P)
      Q=P; P=i;
   elseif (f(i) > f(Q)) & (i ~= P)
      Q=i;
   end
   if f(i) < f(R)
      R=i;
   end
end
%end pickPQR
%---------------------------------------------------------------------

function [feet,f]=reflect_foot(funk,feet,f,j,scale)
% Reflect the jth foot through the centroid of the other
% feet of the amoeba. The displacement may be scaled by
% using scale, whose default value of 1 results in a
% reflection that preserves the volume of the amoeba.
% A scale of 0.5 should never be used, as this would result
% in a degenerate simplex. Typical scale values:
%           1 ..... reflect through amoeba's opposite face
%    -0.5 .... stretch the foot outward, doubling amoeba size
%     0.25 ... shrink inward, halving amoeba size
% The following variables get updated:
%     feet(j,:) -- location of the jth foot
%     f(j) -- value of funk at the jth foot
if nargin ~= 5
   scale=1; %default
```

```
end
if scale == 0.5
   error('Oops, you squashed the amoeba!')
end

[Nfeet,N]=size(feet); %Get amoeba dimensions
if Nfeet ~= N+1
   error('Not an amoeba: wrong number of feet!')
end

% Calculate displacement vector
cent = ( sum(feet,1) - feet(j,:) )/N; %centroid of the feet, except the
jth foot.
disp = 2*(cent - feet(j,:));

% Move the foot, and update f
feet(j,:) = feet(j,:) + (scale*disp); %scaled displacement
f(j) = feval(funk,feet(j,:)); %evaluate funk

%end reflection
%------------------------------------------------------------------------

function [feet,f]=ndcontract(funk,feet,f,j)
% Contract all feet, except jth, toward the jth foot.
% The following variables get updated:
%     feet -- location of each foot
%     f -- value of funk at each foot

[Nfeet,N]=size(feet); %Get amoeba dimensions
if Nfeet ~= N+1
   error('Not an amoeba: wrong number of feet!')
end

for i=1:Nfeet
   if i ~= j
      feet(i,:) = ( feet(i,:) + feet(j,:) )/2;
      f(i) = feval(funk,feet(i,:));
   end
end
%end ndcontract
```

3.2 Application example: least squares fitting

Fitting a line (or a polynomimal) to data can be done most efficiently by the traditional linear least squares technique, but you could use the strategy

below to fit *any* sort of complicated, nonlinear multi-parameter model. It would be easy to modify this code for alternate fit criteria, such as *minimum absolute deviation*, which is more robust against outliers than least squares. The first module is just a main program that picks a random slope and a y-intercept, creates some pseudo-random noisy data that fluctuates about the mean of that line, and then tries to find a best fit line by looking only at the noisy data. The program compares the initial line, the noisy data, and the best-fit line by plotting all three on a single graph. Note that the number of data points (20) is much larger than the number of parameters used for the minimization (N=2, just a slope and a y-intercept).

```
function no_result=least_squares()
% least_squares.m
% Least Squares Curve Fitting Demonstration
hold off

slope=4*(rand(1)-0.5)
yintercept=2*(rand(1)-0.5)

%Make some data, complete with 'noise'
global datax datay %share these with the badness function (below)
datax = 0:0.05:1.0;
datay = (slope*datax)+yintercept;
plot(datax,datay,'r--'); %Plot the theory used to generate the data
hold; %save current display for overplotting
[M,N] = size(datax);
error = 0.5*(rand(M,N)-0.5).*(max(datay)-min(datay)); %create noise
datay = datay + error; %add noise to data
plot(datax,datay,'ro'); %Overlay more realistic noised data

%Find best fit slope and yintercept from noised data
result = amoeba('badness',[0,0],20);
slope=result(1)
yintercept=result(2)
result = amoeba('badness',result,20);
slope=result(1)
yintercept=result(2)

%Generate best fit curve, plot for comparison
fit_y = (slope*datax)+yintercept;
plot(datax,fit_y,'b-'); %Plot the theory
hold off
%end of main program
```

Below is the function f(x), which I call "badness" — a measure of how bad the fit is. This is where *least-squares* comes in: the badness is the sum of the squares of the differences between the real data points and a trial fit. The trial fit is derived from `slopint`, a 2-vector containing a guess for slope and y-intercept. The best fit therefore minimizes `badness`.

```
function sum_squares=badness(slopint)
global datax datay
[M,N] = size(datax);
theory_y = slopint(1)*datax + slopint(2);
sum_squares = sum((theory_y - datay).^2);
%end badness
```