function [X, fX, i] = fmincg(f, X, options, P1, P2, P3, P4, P5)

% Minimize a continuous differentialble multivariate function. Starting point

% is given by "X" (D by 1), and the function named in the string "f", must

% return a function value and a vector of partial derivatives. The Polack-

% Ribiere flavour of conjugate gradients is used to compute search directions,

% and a line search using quadratic and cubic polynomial approximations and the

% Wolfe-Powell stopping criteria is used together with the slope ratio method

% for guessing initial step sizes. Additionally a bunch of checks are made to

% make sure that exploration is taking place and that extrapolation will not

% be unboundedly large. The "length" gives the length of the run: if it is

% positive, it gives the maximum number of line searches, if negative its

% absolute gives the maximum allowed number of function evaluations. You can

% (optionally) give "length" a second component, which will indicate the

% reduction in function value to be expected in the first line-search (defaults

% to 1.0). The function returns when either its length is up, or if no further

% progress can be made (ie, we are at a minimum, or so close that due to

% numerical problems, we cannot get any closer). If the function terminates

% within a few iterations, it could be an indication that the function value

% and derivatives are not consistent (ie, there may be a bug in the

% implementation of your "f" function). The function returns the found

% solution "X", a vector of function values "fX" indicating the progress made

% and "i" the number of iterations (line searches or function evaluations,

% depending on the sign of "length") used.

%

% Usage: [X, fX, i] = fmincg(f, X, options, P1, P2, P3, P4, P5)

%

% See also: checkgrad

%

% Copyright (C) 2001 and 2002 by Carl Edward Rasmussen. Date 2002-02-13

%

%

% (C) Copyright 1999, 2000 & 2001, Carl Edward Rasmussen

%

% Permission is granted for anyone to copy, use, or modify these

% programs and accompanying documents for purposes of research or

% education, provided this copyright notice is retained, and note is

% made of any changes that have been made.

%

% These programs and documents are distributed without any warranty,

% express or implied. As the programs were written for research

% purposes only, they have not been tested to the degree that would be

% advisable in any important application. All use of these programs is

% entirely at the user's own risk.

%

% [ml-class] Changes Made:

% 1) Function name and argument specifications

% 2) Output display

%

% Read options

if exist('options', 'var') && ~isempty(options) && isfield(options, 'MaxIter')

length = options.MaxIter;

else

length = 100;

end

RHO = 0.01; % a bunch of constants for line searches

SIG = 0.5; % RHO and SIG are the constants in the Wolfe-Powell conditions

INT = 0.1; % don't reevaluate within 0.1 of the limit of the current bracket

EXT = 3.0; % extrapolate maximum 3 times the current bracket

MAX = 20; % max 20 function evaluations per line search

RATIO = 100; % maximum allowed slope ratio

argstr = ['feval(f, X']; % compose string used to call function

for i = 1:(nargin - 3)

argstr = [argstr, ',P', int2str(i)];

end

argstr = [argstr, ')'];

if max(size(length)) == 2, red=length(2); length=length(1); else red=1; end

S=['Iteration '];

i = 0; % zero the run length counter

ls\_failed = 0; % no previous line search has failed

fX = [];

[f1 df1] = eval(argstr); % get function value and gradient

i = i + (length<0); % count epochs?!

s = -df1; % search direction is steepest

d1 = -s'\*s; % this is the slope

z1 = red/(1-d1); % initial step is red/(|s|+1)

while i < abs(length) % while not finished

i = i + (length>0); % count iterations?!

X0 = X; f0 = f1; df0 = df1; % make a copy of current values

X = X + z1\*s; % begin line search

[f2 df2] = eval(argstr);

i = i + (length<0); % count epochs?!

d2 = df2'\*s;

f3 = f1; d3 = d1; z3 = -z1; % initialize point 3 equal to point 1

if length>0, M = MAX; else M = min(MAX, -length-i); end

success = 0; limit = -1; % initialize quanteties

while 1

while ((f2 > f1+z1\*RHO\*d1) || (d2 > -SIG\*d1)) && (M > 0)

limit = z1; % tighten the bracket

if f2 > f1

z2 = z3 - (0.5\*d3\*z3\*z3)/(d3\*z3+f2-f3); % quadratic fit

else

A = 6\*(f2-f3)/z3+3\*(d2+d3); % cubic fit

B = 3\*(f3-f2)-z3\*(d3+2\*d2);

z2 = (sqrt(B\*B-A\*d2\*z3\*z3)-B)/A; % numerical error possible - ok!

end

if isnan(z2) || isinf(z2)

z2 = z3/2; % if we had a numerical problem then bisect

end

z2 = max(min(z2, INT\*z3),(1-INT)\*z3); % don't accept too close to limits

z1 = z1 + z2; % update the step

X = X + z2\*s;

[f2 df2] = eval(argstr);

M = M - 1; i = i + (length<0); % count epochs?!

d2 = df2'\*s;

z3 = z3-z2; % z3 is now relative to the location of z2

end

if f2 > f1+z1\*RHO\*d1 || d2 > -SIG\*d1

break; % this is a failure

elseif d2 > SIG\*d1

success = 1; break; % success

elseif M == 0

break; % failure

end

A = 6\*(f2-f3)/z3+3\*(d2+d3); % make cubic extrapolation

B = 3\*(f3-f2)-z3\*(d3+2\*d2);

z2 = -d2\*z3\*z3/(B+sqrt(B\*B-A\*d2\*z3\*z3)); % num. error possible - ok!

if ~isreal(z2) || isnan(z2) || isinf(z2) || z2 < 0 % num prob or wrong sign?

if limit < -0.5 % if we have no upper limit

z2 = z1 \* (EXT-1); % the extrapolate the maximum amount

else

z2 = (limit-z1)/2; % otherwise bisect

end

elseif (limit > -0.5) && (z2+z1 > limit) % extraplation beyond max?

z2 = (limit-z1)/2; % bisect

elseif (limit < -0.5) && (z2+z1 > z1\*EXT) % extrapolation beyond limit

z2 = z1\*(EXT-1.0); % set to extrapolation limit

elseif z2 < -z3\*INT

z2 = -z3\*INT;

elseif (limit > -0.5) && (z2 < (limit-z1)\*(1.0-INT)) % too close to limit?

z2 = (limit-z1)\*(1.0-INT);

end

f3 = f2; d3 = d2; z3 = -z2; % set point 3 equal to point 2

z1 = z1 + z2; X = X + z2\*s; % update current estimates

[f2 df2] = eval(argstr);

M = M - 1; i = i + (length<0); % count epochs?!

d2 = df2'\*s;

end % end of line search

if success % if line search succeeded

f1 = f2; fX = [fX' f1]';

fprintf('%s %4i | Cost: %4.6e\r', S, i, f1);

s = (df2'\*df2-df1'\*df2)/(df1'\*df1)\*s - df2; % Polack-Ribiere direction

tmp = df1; df1 = df2; df2 = tmp; % swap derivatives

d2 = df1'\*s;

if d2 > 0 % new slope must be negative

s = -df1; % otherwise use steepest direction

d2 = -s'\*s;

end

z1 = z1 \* min(RATIO, d1/(d2-realmin)); % slope ratio but max RATIO

d1 = d2;

ls\_failed = 0; % this line search did not fail

else

X = X0; f1 = f0; df1 = df0; % restore point from before failed line search

if ls\_failed || i > abs(length) % line search failed twice in a row

break; % or we ran out of time, so we give up

end

tmp = df1; df1 = df2; df2 = tmp; % swap derivatives

s = -df1; % try steepest

d1 = -s'\*s;

z1 = 1/(1-d1);

ls\_failed = 1; % this line search failed

end

if exist('OCTAVE\_VERSION')

fflush(stdout);

end

end

fprintf('\n');