optU.m

```
function [out]=optU(pr)
% Function optU finds a local minimizer of an unconstrained function on R^n
% using user-selected a gradient based method.
% Author: Tom Asaki
% Version: December 17, 2023
%
% Call:
%
%   [out]=optU(pr)
%
% Inputs:
%
%   pr      a structure variable containing all necessary problem
%           information. The various fields and [default values] are as
%           follows.
%       .progress   A positive intgeger.  Progress will be displayed
%                   every (pr.progress) iterations.
%       .obj        function handle to the objective/gradient computation
%       .x0         vector of intial decision variable values.
%                   (or) n by p array of p initial points for NM or GA.
%       .par        variable to pass to objective function (for example
%                   containing parameters in a structure variable).
%       .method     string indicating optimization method.  Options:
%                       'GD'  (GradientDescent)
%                       'CG'  (ConjugateGradient)
%                       'BFGS' (quasi-Newton)
%                       'LBFGS' (Limited-Memory BFGS)
%                       'TR' (SR1 Trust Region and Steihaug-Toint steps)
%       .maxiter [inf]  maximum number of decision variable updates
%       .ngtol [1E-8]  stop tolerance on gradient norm
%       .dftol [1E-8]  stop tolerance on change in objective
%       .dxtol [1E-8]  stop tolerance on change in decision varable norm
%       .LS.method  string indicating the type of linesearch to perform
%                       'Armijo'       (appropriate for GD)
%                       'StrongWolfe'  (appropriate for CG and BFGS)
%       .LS.lambda [1]  line search initial step size multiplier
%       .LS.lambdamax [100] maximum line search step length
%       .LS.c1 [0.0001]  Armijo sufficient decrease parameter ( 0 < c1 < 1/2 )
%       .LS.c2 [0.9] Curvature condition parameter ( 0 < c1 < c2 < 1 )
%                 If using Conjugate Gradient method ( 0 < c1 < c2 < 1/2 )
%                 with default value [0.4]
%       .LBFGS.m [7] number of L-BFGS iterations to save
%       .CG.reset [0.1] orthogonality reset value for Conjugate Gradient
%       .TR.delta [1] initial trust region size
%       .TR.deltamax [100] maximum trust region size
%       .TR.deltatol [1E-8] stop tolerance on trust region size.
%       .TR.eta [0.01 0.25 0.75]  trust region parameters
%                  [ sufficient decrease , shrink , expand ]
%                  ( 0 <= eta1 < eta2 < eta3 < 1 )
%       .TR.maxcond [1000]  maximum condition number on approximate model
%                   hessian for trust region method.
%
```

```
53 % Outputs:
54 %
55 %   out     a structure variable containing all intial input values
56 %           and additional results of the optimization procedure.
57 %       .pr     copy of the input structure variable with added
58 %               default values and possibly some other algorithmic
59 %               necessary changes.
60 %       .x      (n by iter) array whose columns are the decision
61 %               variable iterates at each iteration
62 %       .f      (1 by iter) array whose entries are the corresponding
63 %               objective function values
64 %       .g      (n by iter) array whose columns are the gradient
65 %               vectors at each iteration.
66 %       .feval  total number of function evaluations
67 %       .geval  total number of gradient evaluations
68 %       .teval  total evaluation clock time
69 %       .msg    output message - reason for algorithm termination
70 %
71
72 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
73 %% Intialization
74
75 out=struct();
76 pr=setdefaults(pr);
77
78 % The output structure variable is initialized here
79 out.pr=pr;
80
81 % start execution timer
82 beg=datetime('now');
83
84 % Make the initial call to the objective function
85 [out.f,out.g]=ev(pr.x0,pr,2);
86 out.x=pr.x0;
87 out.feval=1;
88 out.geval=1;
89
90 % Set iteration information
91 out.msg='';         % terminate when the output message is not empty
92 iter=1;             % counter
93 n=length(pr.x0);    % dimension of decision variable space
94
95 % Initialize terminal output
96 if pr.progress
97     fprintf('\n');
98     fprintf('    date      time    iter       f          |g|        |ap|
   |df|   \n');
99     fprintf('
   ------------------------------------------------------------------------\n');
100    fprintf([char(datetime),' %5d  %+7.4e  %6.4e  \n'],iter-1,out.f(end),
   norm(out.g(:,end)))
101 end
102
103 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
104 %% Main Routines
105
```

```
106 % This section is the main iterative solver.  In each iteration,
107 % a descent direction is chosen (for linesearch) or the model is
108 % formed (for Trust Region).  Then the step is computed as a
109 % subroutine.  Then updates and termination checks are performed.
110
111 while isempty(out.msg)
112
113     switch pr.method
114
115         case 'GD' %%%%% GRADIENT DESCENT %%%%%
116
117             % use negative gradient direction
118             p=-out.g(:,iter);
119
120             % call the linesearch with direction p
121             ox=out.x(:,iter);
122             of=out.f(iter);
123             og=out.g(:,iter);
124             [xnew,flag,nf,ng]=linesearch(ox,of,og,p,pr);
125             out.feval=out.feval+nf;
126             out.geval=out.geval+ng;
127
128         case 'CG' %%%%% CONJUGATE GRADIENT %%%%%
129
130             % Compute the conjugate gradient direction.  If the iteration
131             % is the first, or if descent is stalling, then use the
132             % negative gradient direction (reset)
133             g1=out.g(:,iter);
134             if iter==1
135                 p=-g1;
136             else
137                 g0=out.g(:,iter-1);
138                 rfactor=abs(g1'*g0)/(g1'*g1);
139                 if rfactor>=pr.CG.reset
140                     p=-g1;
141                 else
142                     beta=(g1'*(g1-g0))/(g0'*g0);
143                     beta=max(beta,0);
144                     p=-g1+beta*p;
145                 end
146             end
147
148             % call the linesearch in direction p
149             ox=out.x(:,iter);
150             of=out.f(iter);
151             og=out.g(:,iter);
152             [xnew,flag,nf,ng]=linesearch(ox,of,og,p,pr);
153             out.feval=out.feval+nf;
154             out.geval=out.geval+ng;
155
156         case 'BFGS'  %%%%% QUASI-NEWTON (GFGS) %%%%%
157
158             % Compute the BFGS direction.  If iter=1 then use the negative
159             % gradient direction.  Also use the negative gradient direction
160             % if the update will yield a nearly non-positive definite H.
```

```
161             if iter==1
162                 H=max(abs(out.f(1)),sqrt(pr.dftol))*speye(n);
163                 p=-H*out.g;
164             else
165                 s=out.x(:,end)-out.x(:,end-1);
166                 y=out.g(:,end)-out.g(:,end-1);
167                 if y'*s>0.0001*norm(s)*norm(y)
168                     r=1/(y'*s);
169                     I=eye(n);
170                     H=(I-(r*s)*y')*H*(I-(r*y)*s')+(r*s)*s';
171                 else
172                     H=max(abs(out.f(end)),sqrt(pr.dftol))*speye(n);
173                 end
174                 p=-H*out.g(:,end);
175             end
176 
177             % call the linesearch in direction p, set alpha=1
178             pr.LS.alpha=1;
179             ox=out.x(:,iter);
180             of=out.f(iter);
181             og=out.g(:,iter);
182             [xnew,flag,nf,ng]=linesearch(ox,of,og,p,pr);
183             out.feval=out.feval+nf;
184             out.geval=out.geval+ng;
185 
186         case 'LBFGS' %%%%% LIMITED MEMORY BFGS %%%%%
187 
188             % compute the L-BFGS newton step.
189             % if the first iteration, this is scaled gradient descent
190             if iter==1
191                 p=-max(abs(out.f(1)),sqrt(pr.dftol))*out.g;
192                 s=[];
193                 y=[];
194             % If not the first iteration, update H using the previous
195             % m steps
196             else
197                 s(:,end+1)=out.x(:,end)-out.x(:,end-1);   %#ok
198                 y(:,end+1)=out.g(:,end)-out.g(:,end-1);   %#ok
199                 if iter>pr.LBFGS.m+1
200                     s(:,1)=[];
201                     y(:,1)=[];
202                 end
203                 p=-out.g(:,end);
204                 for i=min(pr.LBFGS.m,iter-1):-1:1
205                     alph(i)=(s(:,i)'*p)/(s(:,i)'*y(:,i));
206                     p=p-alph(i)*y(:,i);
207                 end
208                 p=(s(:,end)'*y(:,end))/(y(:,end)'*y(:,end))*p;
209                 for i=1:min(pr.LBFGS.m,iter-1)
210                     beta=(y(:,i)'*p)/(s(:,i)'*y(:,i));
211                     p=p+(alph(i)-beta)*s(:,i);
212                 end
213             end
214 
215             % Call the linesearch in direction p,  set alpha=1
```

```
            pr.LS.alpha=1;
            ox=out.x(:,iter);
            of=out.f(iter);
            og=out.g(:,iter);
            [xnew,flag,nf,ng]=linesearch(ox,of,og,p,pr);
            out.feval=out.feval+nf;
            out.geval=out.geval+ng;

        case 'TR'  %%%%% SR1 TRUST REGION %%%%%

            % Update model hessian using SR1
            if iter==1
                B=abs(out.f)*eye(n);
            else
                s=out.x(:,end)-out.x(:,end-1);
                y=out.g(:,end)-out.g(:,end-1);
                w=y-B*s;
                B=B+(w*w')/(w'*s);
            end

            % Call the Trust Region step algorithm (Steihaug-Toint)
            ox=out.x(:,end);
            of=out.f(end);
            og=out.g(:,end);
            [xnew,flag,nf,delta]=TrustRegionStep(ox,of,og,B,pr);
            out.feval=out.feval+nf;

            % Save the trust region size for the next iteration
            pr.TR.delta=delta;

    end   % end of method switch

    % update the iteration counter
    iter=iter+1;

    % update x,f,g in the output structure
    out.x(:,iter)=xnew;
    [out.f(iter),out.g(:,iter)]=ev(xnew,pr,2);
    out.geval=out.geval+1;

    % check termination criteria and set output message
    if iter>pr.maxiter
        out.msg='Maximum number of iterations reached.';
    end
    if norm(out.g(:,iter))<pr.ngtol
        out.msg='Minimum gradient norm reached.';
    end
    if (iter>1 && norm(diff(out.x(:,iter-1:iter),[],2))<pr.dxtol)
        out.msg='Minimum step size reached.';
    end
    if (iter>1 && abs(diff(out.f(iter-1:iter)))<pr.dftol)
        out.msg='Minimum change in objective reached.';
    end
    if flag % step determination "failed" for some reason
        switch pr.method
```

```
271             case 'TrustRegion'
272                 out.msg='Minimum trust region size reached.';
273             otherwise
274                 out.msg='Linesearch failed to find an acceptable iterate.';
275         end
276     end
277 
278     % Print iteration status/result to terminal
279     if pr.progress
280         if ~mod(iter-1,pr.progress) || ~isempty(out.msg)
281             ff=out.f(end);
282             gg=norm(out.g(:,end));
283             pp=norm(out.x(:,end-1)-out.x(:,end));
284             df=out.f(end-1)-out.f(end);
285             fprintf([char(datetime), ...
286                 ' %5d  %+7.4e  %6.4e  %6.4e  %6.4e  \n'], ...
287                 iter-1,ff,gg,pp,df)
288         end
289     end
290 
291 end
292 
293 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
294 %% Wrap Up
295 
296 % save execution run time
297 fin=datetime('now');
298 out.teval=seconds(duration(fin-beg));
299 
300 % closing terminal messages (key results)
301 if pr.progress
302     fprintf('\n');
303     fprintf('Objective Value : %+8.5e\n',out.f(end));
304     fprintf('      Algorithm : %s\n',pr.method);
305     fprintf('        Message : %s\n',out.msg);
306     fprintf(' Execution Time : %g seconds\n',out.teval);
307     fprintf(' Function Evals : %d\n',out.feval);
308     fprintf(' Gradient Evals : %d\n',out.geval);
309     fprintf('Effective Evals : %d\n',out.feval+n*out.geval);
310     fprintf('\n');
311 end
312 
313 return
314 
315 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
316 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
317 
318 function [pr]=setdefaults(pr)
319 
320 df.par              =   [];
321 df.method           =   'BFGS';
322 df.linesearch       =   'StrongWolfe';
323 df.maxiter          =   inf;
324 df.ngtol            =   1E-8;
325 df.dftol            =   1E-8;
```

```
df.dxtol            =   1E-8;
df.progress         =   1;
fn=fieldnames(df);
for k=1:length(fn)
    if ~isfield(pr,fn{k}) || isempty(pr.(fn{k}))
        pr.(fn{k})=df.(fn{k});
    end
end

df.LS.lambda        =   1;
df.LS.lambdamax     =   100;
df.LS.c1            =   0.0001;
df.LS.c2            =   0.9;
fn=fieldnames(df.LS);
if ~isfield(pr,'LS')
    gn=fn';gn{2,1}=[];
    pr.LS=struct(gn{:});
end
for k=1:length(fn)
    if ~isfield(pr.LS,fn{k}) || isempty(pr.LS.(fn{k}))
        pr.LS.(fn{k})=df.LS.(fn{k});
    end
end

df.LBFGS.m          =   7;
fn=fieldnames(df.LBFGS);
if ~isfield(pr,'LBFGS')
    gn=fn';gn{2,1}=[];
    pr.LBFGS=struct(gn{:});
end
for k=1:length(fn)
    if ~isfield(pr.LBFGS,fn{k}) || isempty(pr.LBFGS.(fn{k}))
        pr.LBFGS.(fn{k})=df.LBFGS.(fn{k});
    end
end

df.CG.reset         =   0.2;
fn=fieldnames(df.CG);
if ~isfield(pr,'CG')
    gn=fn';gn{2,1}=[];
    pr.CG=struct(gn{:});
end
for k=1:length(fn)
    if ~isfield(pr.CG,fn{k}) || isempty(pr.CG.(fn{k}))
        pr.CG.(fn{k})=df.CG.(fn{k});
    end
end

df.TR.delta         =   1;
df.TR.deltamax      =   100;
df.TR.deltatol      =   1E-8;
df.TR.eta           =   [0.01 0.25 0.75];
df.TR.maxcond       =   1000;
fn=fieldnames(df.TR);
if ~isfield(pr,'TR')
```

```
381     gn=fn';gn{2,1}=[];
382     pr.TR=struct(gn{:});
383 end
384 for k=1:length(fn)
385     if ~isfield(pr.TR,fn{k}) || isempty(pr.TR.(fn{k}))
386         pr.TR.(fn{k})=df.TR.(fn{k});
387     end
388 end
389 
390 if strcmp(pr.method,'CG') && pr.LS.c2>=0.5
391     pr.LS.c2=0.45;
392 end
393 
394 if pr.LS.c1>=pr.LS.c2
395     pr.LS.c1=0.0001;
396 end
397 
398 pr.LBFGS.m=min(pr.LBFGS.m,length(pr.x0));
399 
400 return
401 
402 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
403 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
404 
405 function [f,g]=ev(x,pr,m)
406 % this function governs all calls to an objective function.
407 %   x   is the point to evaluate
408 %   pr  is the problem structure variable
409 %   m   is an evaluation indicator.  If m=1 then compute f only
410 %           otherwise compute both f and g (gradient).
411 if m>1
412     if isempty(pr.par)
413         [f,g]=pr.obj(x);
414     else
415         [f,g]=pr.obj(x,pr.par);
416     end
417 else
418     if isempty(pr.par)
419         [f]=pr.obj(x);
420     else
421         [f]=pr.obj(x,pr.par);
422     end
423 end
424 return
425 
426 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
427 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
428 
429 function [xnew,flag,nf,ng]=linesearch(x,f,g,p,pr)
430 
431 % when a linesearch is started, we are considering a current iterate x
432 % and we already have the objective f and gradient g at x.  We also have
433 % a descent direction vector p.  The other variable pr is the entire
434 % problem structure variable that contains linesearch hyperparameters.
435 
```

```
436 nf=0;
437 ng=0;
438
439 switch pr.linesearch
440
441     case 'Armijo'
442         goflag=true;
443         flag=false;
444         L=pr.LS.lambda;
445         c1=pr.LS.c1;
446         dx=pr.dxtol;
447         d0=p'*g;
448         while goflag
449             xnew=x+L*p;
450             fnew=ev(xnew,pr,1);
451             nf=nf+1;
452             if fnew>f+c1*L*d0
453                 L=L/2;
454                 if norm(L*p)<dx,goflag=false;flag=true;end
455             else
456                 goflag=false;
457             end
458         end
459
460     case 'StrongWolfe'
461         goflag=true;
462         flag=false;
463         k=1;
464         L(k)=pr.LS.lambda;
465         c1=pr.LS.c1;
466         c2=pr.LS.c2;
467         d0=p'*g;
468         while goflag
469             F(k)=ev(x+L(k)*p,pr,1);                          %#ok
470             nf=nf+1;
471             if (F(k)>f+c1*L(k)*d0) || (k>1 && F(k)>=F(k-1))
472                 if k==1
473                     [lambdastar,mf,mg]=zoom(0,L(k),x,f,p,d0,f,pr);
474                 else
475                     [lambdastar,mf,mg]=zoom(L(k-1),L(k),x,f,p,d0,F(k-1),pr);
476                 end
477                 nf=nf+mf;
478                 ng=ng+mg;
479                 goflag=false;
480             end
481             if goflag
482                 [dummy,g]=ev(x+L(k)*p,pr,2);                          %#ok
483                 nf=nf+1;
484                 ng=ng+1;
485                 dk=p'*g;
486                 if abs(dk)<=-c2*d0
487                     lambdastar=L(k);
488                     goflag=false;
489                 end
490             end
```

```
            if goflag && dk>=0
                if k==1
                    [lambdastar,mf,mg]=zoom(L(k),0,x,f,p,d0,F(k),pr);
                else
                    [lambdastar,mf,mg]=zoom(L(k),L(k-1),x,f,p,d0,F(k),pr);
                end
                nf=nf+mf;
                ng=ng+mg;
                goflag=false;
            end
            if goflag
                k=k+1;
                L(k)=2*L(k-1);
                if L(k)>pr.LS.lambdamax
                    flag=true;
                    goflag=false;
                    lambdastar=0;
                end
            end
        end
        xnew=x+lambdastar*p;

    otherwise

end

return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [lambdastar,nf,ng]=zoom(L,H,x,f,p,d0,fL,pr)
nf=0;
ng=0;
goflag=true;
while goflag
    M=(L+H)/2;
    fM=ev(x+M*p,pr,1);
    nf=nf+1;
    if fM>f+pr.LS.c1*M*d0 || fM>=fL
        H=M;
    else
        [~,gM]=ev(x+M*p,pr,2);
        nf=nf+1;
        ng=ng+1;
        dk=p'*gM;
        if abs(dk)<=-pr.LS.c2*d0
            lambdastar=M;
            goflag=false;
        end
        if goflag
            if dk*(H-L)>=0
                H=L;
            end
            L=M;
```

```
        end

    end
    if M*norm(p)<2*pr.dxtol
        lambdastar=M;
        goflag=false;
    end
end
return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [xnew,flag,nf,delta]=TrustRegionStep(x,f,g,B,pr)

% when calling for a TR step we have a current iterate x and we have
% already evaluated the objective f and gradient g at x.  We also have
% a model defined by a local approximate hessian B.  The only other
% input is the entire problem structure variable pr containing
% necessary TR hyperparameters.

flag=false;
delta=pr.TR.delta;
rho=-1;
nf=0;
g0=g;
n=length(x);

% loop until an acceptable point is found
while rho<=pr.TR.eta(1)

    % Compute the Steihaug-Toint step.
    z=zeros(size(g));
    r=g;
    d=-g;
    ng=norm(g);
    e=min(0.5,sqrt(ng))*ng;
    StopCondition=false;
    inneriter=0;
    while StopCondition==false
        t=d'*(B*d);
        if t<=0
            tau=posroot(z,d,delta);
            p=z+tau*d;
            StopCondition=true;
        else
            alpha=(r'*r)/t;
            z=z+alpha*d;
            if norm(z)>=delta
                tau=posroot(z,d,delta);
                p=z+tau*d;
                StopCondition=true;
            else
                rold=r;
                r=r+alpha*(B*d);
```

```
                if norm(r)<e
                    p=z;
                    StopCondition=true;
                end
                beta=(r'*r)/(rold'*rold);
                d=-r+beta*d;
            end
        end
        inneriter=inneriter+1;
        if inneriter==n
            p=z;
            StopCondition=true;
        end
    end

    % evaluate rho
    [fnew]=ev(x+p,pr,1);
    nf=nf+1;
    modelchange=-g0'*p-(p'*(B*p))/2;
    rho=(f-fnew)/modelchange;

    % updates: shrink or grow delta, keep an improved point
    if rho<pr.TR.eta(2)
        delta=delta/4;
    elseif rho>pr.TR.eta(3) && norm(p)>0.999*delta
        delta=min(2*delta,pr.TR.deltamax);
    end
    if rho>pr.TR.eta(1)
        xnew=x+p;
    end

    % if trust region gets too small, then stop
    if delta<pr.TR.deltatol
        flag=true;
        xnew=x+p;
        rho=inf;
    end

end

% if the last iteration did not find an improved point because
% it stopped on deltatol, then do not update x.
if fnew>=f
    xnew=x;
end

return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function tau=posroot(z,d,delta)
a=z'*d;
b=d'*d;
tau=-(a/b)+sqrt((a/b)^2+(delta^2-z'*z)/b);
```

```
656 return
657
658
```