```
% simple_minimizer.m
%
% This MATLAB m-file contains a function that implements
% a particularly simple form of a quasi-Newton minimization
% routine employing finite difference estimates of the
% Hessian, a Levenberg-Marquardt type of modification to
% ensure that each search direction is a descent direction,
% and a secant method for the line search.
%
% K. Beers.
% MIT ChE
% 12/6/2001


function [x,iflag,Traj] = ...
   simple_minimizer(fun_name,x_guess,Opt);

iflag = 0;

num_dim = length(x_guess);

% Initialize the estimate of the solution.
x = x_guess;

% First, we calculate the gradient for the initial guess.
[F,g] = feval(fun_name,x);

% Next, we approximate the Hessian using finite differences.
Hess = approx_Hessian_FD(x_guess,g,fun_name);

% Next, we set the initial diagonal element used in the
% Levenberg-Marquardt method.
tau_LM = 1e-3;

% Set the maximum number of iterations for the
% minimization routine.
max_iter = 100;
if(isfield(Opt,'max_iter'))
   max_iter = Opt.mat_iter;
end

% Set the maximum number of iniital guesses used to
% begin the line searches.
max_line_guess = 10;
if(isfield(Opt,'max_line_guess'))
   max_line_guess = Opt.max_line_guess;
end

% Set the maximum number of secant method iterations
```

```
% per line search guess.
max_secant = 5;
if(isfield(Opt,'max_secant'))
   max_secant = Opt.max_secant;
end

% Set the convergence tolerance for ending the secant
% method line search.
atol_line = 1e-4;
if(isfield(Opt,'atol_line'))
   atol_line = Opt.atol_line;
end

% Set the convergence tolerance for ending the
% minimizer iterations.
atol = 1e-10;
if(isfield(Opt,'atol'))
   atol = opt.atol
end

% Set the integer flag telling how often to print out
% the simulation results to the trajectory data structure.
iprint_traj = 0;
if(isfield(Opt,'iprint_traj'))
   iprint_traj = Opt.iprint_traj;
end


% Begin the iterations of the minimization routine.
count_traj = 0;

for iter = 0:max_iter

   % save the last value of the cost function for later
   % use in the descent criterion
   F_last_iter = F;

   % Calculate the search direction, ensuring through the
   % Levenberg-Marquardt method that it is a descent direction.
   ifound_descent = 0;
   while(~ifound_descent)
      p = (Hess + tau_LM*eye(num_dim))\(-g);
      direct_deriv = dot(g,p);
      if(direct_deriv < 0)
         ifound_descent = 1;
         tau_LM = 0.1*tau_LM;
      else
         tau_LM = 10*tau_LM;
      end
   end
```

```
% Now, since we know that this is a search direction, there must
% exist some small step length that will reduce the function.
% We try first the full step length sugessted by the calculation
% above.  If the secant method with this initial guess does not
% yield a lower value of the cost function, then we try again with
% a smaller step size.
for line_guess = 0:max_line_guess

   % Use as a first guess of alpha the full step from the
   % calculation above.  Then, if the secant method does
   % not find a point lowering the cost function, try
   % again wih half of the last initial step size.
   alpha = 2^-line_guess;
   [F,g] = feval(fun_name,x+alpha*p);

   % To start the secant method, take the "-1" iteration
   % to be a point just offset from the initial guess of
   % the step.  This is used only to approximate the
   % derivative.
   alpha_old = alpha - sqrt(eps);
   [F_old,g_old] = feval(fun_name,x+alpha_old*p);

   % Begin secant method iterations
   for isecant = 1:max_secant

      % Calculate the update to alpha
      delta_alpha = -(dot(g,p)*(alpha-alpha_old)) / ...
         (dot(g,p) - dot(g_old,p));
      % Update the value of alpha
      alpha_old = alpha;
      alpha = alpha + delta_alpha;

      % Calculate new gradient.
      g_old = g;
      F_old = F;
      [F,g] = feval(fun_name,x+alpha*p);

      % Check for convergence if the magnitude of the
      % directional derivative drops below the convergence
      % criterion.
      if(abs(dot(g,p)) <= atol_line)
         break;
      end

   end

   % Check to make sure that identified point is satisfies
   % the descent criterion.
   if(F < F_last_iter)
      x = x + alpha*p;
```

```
        % Update estimate of Hessian.
        Hess = approx_Hessian_FD(x,g,fun_name);

        % If desired, print out the current results to the
        % trajectory data structure.
        if(mod(iter,iprint_traj)==0)
           count_traj = count_traj + 1;
           Traj.iter(count_traj) = iter;
           Traj.x(count_traj,:) = x';
           Traj.F(count_traj) = F;
           Traj.g(count_traj,:) = g';
        end

        % stop the line search process
        break;

     end

  end


  % Finally, we check for convergence to see whether the
  % magnitude of the gradient has reached a sufficiently
  % small number.
  if(dot(g,g) <= atol*atol)
     iflag = 1;
     break;
  end

end


return;


% ===========================================
% approx_Hessian_FD.m

function [Hess,iflag] = approx_Hessian_FD(x,g,fun_name);

iflag = 0;

% extract the number of state variables
Nvar = length(x);

% Allocate space for the Hessian in memory using full
% matrix format.
Hess = zeros(Nvar,Nvar);

% We set the offset used in the finite difference formula.
epsilon = sqrt(eps);
```

```
% Begin iterations over each state variable to estimate corresponding
% elements of the Hessian by finite differences.
for k = 1:Nvar

   % Get offset state vector.
   x_off = x;
   x_off(k) = x_off(k) + epsilon;

   % Calculate function vector for offset state vector.
   [F_off,g_off] = feval(fun_name,x_off);

   % Calculate the Hessian elements in column ivar.
   Hess(:,k) = (g_off - g)/epsilon;

end

% We now ensure that the approximate Hessian
% is symmetric.
Hess = (Hess' + Hess)/2;

iflag = 1;

return;
```