All code generated with Matlab® Software

```
% reduced_Newton.m
%
% This MATLAB® m-file uses a reduced-Newton algorithm with a
% weak line search to solve a set of non-linear algebraic
% equations.
%
% The input parameters are :
%
% x0 = a column vector of the initial guess of the unknowns
%
% calc_f = the name of a MATLAB® function that calculates
%     the function vector
%
% calc_Jac = the name of a function that calculates the Jacobian
%
% Options = a data structure containing optional flags
% .max_iter = max # of Newton's method iterations
% .max_iter_LS = max # of weak line search iterations
% .rtol = relative tolerance
% .atol = absolute tolerance
% .step_tol = abs. tolerance below which we switch to full Newton's method
% .verbose = return a trajectory matrices containing the history
%           of the Newton's method iterations
% .use_range = if non-zero, limit the maximum magitude of the full Newton
%            step so that the change in each component is not greater than
%            that in the vector .range
% .range = a vector of the ranges for each of the unknowns.  Each component
%          of the Newton step
%
% Param = a data structure containing parameters that are to be passed to
%         the calc_f and calc_Jac functions
%
% The output parameters are :
%
% x = the final estimate of the solution
%
% iflag = an integer flag that is 1 for convergence,
%      0 for no convergence, and negative for an error
%
% iter_conv = number of iterations required for convergence
%
% x_traj = a matrix where row # j is the solution estimate at iteration j-1
% f_traj = a matrix where row # j is the function vector at iteration j-1

function [x,iflag,iter_conv,x_traj,f_traj] = ...
   reduced_Newton(x0,calc_f,calc_Jac,Options,Param);


% First, signal no convergence.
iflag = 0;
```

```
% Set number of iterations required for convergence.
iter_conv = 0;

% Extract number of state variables.
Nvar = length(x0);

% Initialize solution estimate.
x = x0;

% Calculate initial function vector.
f = feval(calc_f,x,Param);
if(length(f) ~= Nvar)
   iflag = -1;
   error('reduced_Newton: calc_f returns vector of improper length');
end
% ensure f is a column vector
if(size(f,1)~=Nvar)
   f = f';
end

% Obtain initial norm of the function vector for later
% convergence tests.

f0_norm_inf = max(abs(f));

f_norm_2sq = dot(f,f);


% Record initial state and function vectors in trajectory.
count_traj = 1;
x_traj(count_traj,:) = x';
f_traj(count_traj,:) = f';


% Set the flag telling us to perform weak line searches.
i_do_LS = 1;


% Begin Newton's method iterations

for iter = 1:Options.max_iter

   % calculate the Jacobian
   Jac = feval(calc_Jac,x,Param);

   % Solve the set of linear equations for the full line step
   try
      p = Jac\(-f);
   catch
      iflag = -2;
      error('reduced_Newton: full Newton step calculation error');
```

```
end


% Now, reduce the magnitude of the Newton step if the user has
% specified a maximum change allowable for each component.
if(Options.use_range)

   % Calculate the unit vector lying in the Newton line search
   % direction.
   p_length = norm(p,2);
   p_unit = p/p_length;

   % Calculate the maximum step in this direction allowable under
   % the condition that each state variable must not change by
   % a magnitude greater than the specified range for that variable.
   step_allow = max(abs(Options.range));
   for ivar=1:Nvar
      try
         step_ivar = abs(Options.range(ivar)/p_unit(ivar));
         if(step_ivar < step_allow)
            step_allow = step_ivar;
         end
      end
   end
   step_allow = min(step_allow,p_length);
   p = p_unit*step_allow;
end


% Begin the weak line search
if(i_do_LS)   % perform a weak line search

   for iter_LS = 0:Options.max_iter_LS
      iconv_LS = 0;

      % Calculate fractional step length
      lambda = 2^(-iter_LS);

      % Calculate new solution estimate
      x_new = x + lambda*p;

      % Calculate function at the new solution estimate
      f_new = feval(calc_f,x_new,Param);

      % Check descent criterion
      f_new_norm_2sq = dot(f_new,f_new);
      if(f_new_norm_2sq <= f_norm_2sq)
         x = x_new;
         f = f_new;
         f_norm_2sq = f_new_norm_2sq;
         iconv_LS = 1;
```

```
            break;
        end

    end

    % If we did not satisfy descent condition, update
    % with final result.
    if(~iconv_LS)
        x = x_new;
        f = f_new;
        f_norm_2sq = f_new_norm_2sq;
    end


else  % use full Newton step instead

    % Calculate new solution estimate
    x = x + p;

    % Calculate function at the new solution estimate
    f = feval(calc_f,x,Param);

end


% if in verbose mode, record state and function vectors
if(Options.verbose)
    count_traj = count_traj + 1;
    x_traj(count_traj,:) = x';
    f_traj(count_traj,:) = f';
end

% check for convergence to the solution
f_norm_inf = max(abs(f));
i_conv_rel = 0;
if(f_norm_inf <= Options.rtol*f0_norm_inf)
    i_conv_rel = 1;
end
i_conv_abs = 0;
if(f_norm_inf <= Options.atol)
    i_conv_abs = 1;
end
if((i_conv_rel==1)&(i_conv_abs==1))
    iter_conv = iter;
    iflag = 1;
    break;
end


% Check to see whether need to perform a line search
% at the next step.
```

```
   if(f_norm_inf <= Options.step_tol)
      i_do_LS = 0;
   else
      i_do_LS = 1;
   end

end

return;
```