% simple_nonlinear_LS.m

```
%
% This MATLAB function uses only standard MATLAB
% routines to implement the nonlinear fitting
% of parameters using the Marquardt method.
% The errors for each response measurement are
% assumed to be independently distributed
% according to a normal distribution with a
% mean of zero and a common variance.
%
% Once the parameter vector minimizing the
% residual sum of squarred errors is calculated,
% approximate confidence intervalues are generated
% using the linearized design matrix around the
% minimum.  The normal equations are solved using
% singular value decomposition.
%
%
% INPUT :
% =======
% y - the vector of observed responses
%
% calc_yhat - the name of a function that takes as input
%   a given estimate of the model parameters and returns
%   as output the vector of model predictions
%
% theta_guess - the initial guess of the vector of parameters
%
% alpha_CI - determines the width of the confidence intervals;
%   a value of 0.05 returns a 95% confidence interval.
%
% Options - a data structure that provides information to
%   override the default options of the solver routine.
%   Fields are (with default values in parentheses) :
%       .atol - convergence tolerance of gradient norm (1e-8)
%       .max_iter - max # of Gauss-Newton iterations (100)
%       .max_line - max # of weak line steps (10)
%       .tau_LM - the initial gues of the positive
%             value added to the diagonal elements of the
%             approximate Hessian in the Levenberg-Marquardt
%             method (1e-3)
%       .search_angle_max - the maximum angle, in degrees, allowed
%             between the search direction and the direction of
%             steepest descent (80)
%       .verbose - if 0, don't print to screen results.  If 1,
%             print final results.  If >1, print results of
%             each iteration (0)
%
% Param_fix - a data structure of fixed parameters that may be
```

```
%   passed to the function calc_yhat
%
%
% OUTPUT :
% ========
% theta - the parameter found as a local minimum
%   of the residual sum of squared errors
%
% theta_CI - a vector containing the half-widths
%   for the confidence intervals for each parameter
%
% yhat - the vector of model predictions using the
%   fitted values of the model parameters
%
% yhat_CI - a vector containing the half-widths for
%   the confidence intervals for each model
%   prediction.
%
% Stats - a data structure containing various measures
%   of fit of the data
%   .RSS - the residual sum of squared errors
%   .sample_var - the sample variance
%   .sample_std - the sample standard deviation
%   .R2_corr - the R^2 correlation coefficient that
%       reports the quality of the fit
%
% iflag - an integer that takes a value of 1 upon
%   successful completion of the routine
%
% K. Beers
% MIT ChE
% 12/8/2001

function [theta,theta_CI,yhat,yhat_CI,Stats,iflag] = ...
   simple_nonlinear_LS(...
   y,calc_yhat,theta_guess,alpha_CI,Options,Param_fix);

iflag = 0;


% We extract the number of the measurements and
% the number of parameters to be fitted.
num_exp = length(y);
num_param = length(theta_guess);

% From the initial guess we now calculate the initial
% model predicions.  From this, we then calculate the
% residual error vector.
theta = theta_guess;
[yhat,iflag2] = feval(calc_yhat,theta,Param_fix);
if(iflag2 <= 0)
```

```
   iflag = -1;
   error(['calc_yhat returned error : ', ...
         int2str(iflag2)]);
end
residual = y - yhat;
RSS = dot(residual,residual);

% We now use finite differences to estimate the linearized
% design matrix X.
[X,iflag2] = approx_X_FD(theta,yhat,calc_yhat,Param_fix);
if(iflag2 <= 0)
   iflag = -2;
   error(['approx_X_FD returned error : ', ...
         int2str(iflag2)]);
end

% We now set the parameters that direct the optimization
% algorithm.
if(~exist('Options'))
   Options.null = 0;
end

atol = 1e-8;
if(isfield(Options,'atol'))
   atol = Options.atol;
end

max_iter = 100;
if(isfield(Options,'max_iter'))
   max_iter = Options.max_iter;
end

max_line = 10;
if(isfield(Options,'max_line'))
   max_line = Options.max_line;
end

tau_LM = 1e-3;
if(isfield(Options,'tau_LM'))
   tau_LM = Options.tau_LM;
end

search_angle_max = 80;
if(isfield(Options,'angle_max'))
   search_angle_max = Options.search_angle_max;
end
search_cos_min = cos(search_angle_max/180*pi);

verbose = 0;
if(isfield(Options,'verbose'))
   verbose = Options.verbose;
```

```
   if(verbose < 0)
      verbose = 0;
   end
end

if(verbose)
   disp(' ');
   disp(['Initial RSS = ', num2str(RSS)]);
end


% We now begin the iterations of the Gauss-Newton method,
% using the Levenberg-Marquardt method to modify the
% approximate Hessian if the search direction is too far
% away from the direction of steepest descent.
if(verbose)
   disp(' ');
   disp('Iteration    RSS    |grad.|    tau_LM');
end

coverged = 0;
for iter=1:max_iter

   % First, we store the old value of the
   % residual sum of squares to use in the
   % descent criterion.
   RSS_old = RSS;

   % We now calculate the search direction using the
   % Levenberg-Marquardt method of adding a diagonal
   % matrix (here we use Levenberg's choice of I) to
   % bias the search direction towards the steepest
   % descent direction when the approximate Hessian
   % behaves poorly.  Since we are adding a postive
   % diagonal matrix, problems with near-singularity
   % are somewhat reduced.  We use a QR decomposition
   % to solve.

   % We use a while loop to keep calculating search
   % directions until we find one that is less than
   % the maximum angle away from the direction of
   % steepest descent.
   accept_search = 0;
   d = X'*residual;
   while(~accept_search)

      [Q,R] =qr(X'*X + tau_LM*eye(num_param));
      p = R\(Q'*X'*residual);

      try
         search_cos = dot(p,d)/sqrt(dot(p,p))/ ...
```

```
            sqrt(dot(d,d));
         if(search_cos >= search_cos_min)
            accept_search = 1;
            tau_LM = 0.1*tau_LM;
         else
            tau_LM = 10*tau_LM;
         end
      catch
         accept_search = 1;
      end
   end

   % Now that we have selected the search direction, we begin
   % a weak line search to force acceptance of the descent
   % criterion.
   ifound_descent = 0;
   for iter_line = 0:max_line

      % We try the following fractional step length.
      alpha = 2^-iter_line;

      % We now calculate the model predictions for this
      % new estimate of the parameter vector.
      yhat = feval(calc_yhat,theta+alpha*p,Param_fix);
      RSS = dot(y-yhat,y-yhat);

      % If this fractional step satisfies the descent
      % criterion, accept it.  Otherwise, try a smaller
      % fractional step.
      if(RSS < RSS_old)
         theta = theta + alpha*p;
         X = approx_X_FD(theta, ...
            yhat,calc_yhat,Param_fix);
         residual = y - yhat;
         ifound_descent = 1;
         break;
      end

   end

   % If we have not been able to decrease the cost function,
   % increase tau_LM and try again.
   if(~ifound_descent)
      tau_LM = 10*tau_LM;
   end

   % We print the results of the iteration to the screen
   % if requested.
   if(verbose > 1)
      disp([int2str(iter), '    ', ...
            num2str(RSS), '    ', ...
```

```
            num2str(sqrt(dot(d,d))), '    ', ...
            num2str(tau_LM)]);
   end

   % If we have updated the estimate, we check the magnitude
   % of the gradient for convergence.  d, the direction of
   % steepest descent, is merely the negative of the gradient,
   % and so we use this vector.
   d = X'*residual;
   if(dot(d,d) <= atol*atol)
      converged = 1;
      break;
   end

end

iflag = converged;

if(verbose)
   disp(' ');
   disp('Final parameter vector estimate = ');
   disp(theta);
   disp(['converged = ', int2str(converged)]);
end


% Now that we have identified a local minimum for fitting
% the model parameters, we calculate approximate confidence
% intervalues using the final linearized design matrix X.

% From the value of alpha, we calculate the t-value for
% the confidence intervals.
num_DOF = num_exp - num_param;
t_value = calc_t_value(num_DOF,alpha_CI/2);
if(verbose)
   disp(' ');
   disp(['t value for CI = ', num2str(t_value)]);
end

% Next, we estimate the sample variance and sample
% standard deviation, and the correlation coefficent.
Stats.RSS = RSS;
Stats.sample_var = RSS/num_DOF;
Stats.sample_std = sqrt(Stats.sample_var);
Stats.R2_corr = 1 - RSS/dot(y,y);
if(verbose)
   disp(' ');
   disp('Quality of fit parameters : ');
   disp(['RSS = ', num2str(Stats.RSS)]);
   disp(['sample_var = ', num2str(Stats.sample_var)]);
   disp(['sample_std = ', num2str(Stats.sample_std)]);
```

```
   disp(['R^2 corr. coeff. = ', num2str(Stats.R2_corr)]);
end

% We then make a plot of the residuals.
% make a plot of the residuals
if(verbose > 1)
   figure;
   plot(y,residual,'o');
   hold on;
   plot([min(y) max(y)],[0 0],'-.');
   xlabel('y'); ylabel('residual');
   title(['Residual errors, s = ', ...
         num2str(Stats.sample_std)]);
end

% We now use this to develop confidence intervals for
% the model predictions.

% Calculate the inverse of the matrix X'*X.  If this
% matrix is singular, we do not calculate the confidence
% intervals, but instead return values of zero.
do_CI = 0;
try
   XT_X_inv = inv(X'*X);
   do_CI = 1;
catch
   disp('Singular X^T * X matrix, no CI calculated');
   do_CI = 0;
end

% We now calculate the confidence interval half-widths for
% the model predictions.
yhat_CI = zeros(num_exp,1);
if(do_CI)
   for k=1:num_exp
      xk = X(k,:)';
      yhat_CI(k) = t_value*Stats.sample_std * ...
         sqrt(xk'*XT_X_inv*xk);
   end
end

% Next, we calculate half-widths for the confidence intervals
% for each model parameter.
theta_CI = zeros(num_param,1);
if(do_CI)
   for k=1:num_param
      uk = zeros(num_param,1);
      uk(k) = 1;
      theta_CI(k) = t_value*Stats.sample_std * ...
         sqrt(uk'*XT_X_inv*uk);
   end
```

```
end
if(verbose)
   disp('Confidence interval half-widths : ');
   disp(theta_CI);
end


return;


% ====================================
% ====================================
% approx_X_FD.m
% This MATLAB function uses finite differences to approximate the
% design matrix used to form the normal equations.

function [X,iflag] = approx_X_FD(theta,yhat,calc_yhat,Param_fix);

iflag = 0;

% extract the number of measurements and the number
% of parameters
num_exp = length(yhat);
num_param = length(theta);

% Allocate space for the linearized design matrix in
% full matrix format
X = zeros(num_exp,num_param);

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


% We now offset the value of each parameter slightly to
% calculate the corresponding rows of the linearized
% design matrix.
for k = 1:num_param

   % Get offset parameter vector.
   theta_off = theta;
   theta_off(k) = theta_off(k) + epsilon;

   % Calculate yhat vector for offset parameter vector.
   yhat_off = feval(calc_yhat,theta_off,Param_fix);

   % Calculate the elements of the linearized design matrix
   % in the column corresponding to this parameter.
   X(:,k) = (yhat_off - yhat)/epsilon;

end
```

```
iflag = 1;

return;



% ================================================
% ================================================
% calc_t_value
% This MATLAB function calculates the t-value for a confidence
% interval.  This is the value of t, for which the probability
% of finding a t greater than this value is equal to alpha_u.
% We use first the symbolic toolkit to calculate the
% hypergeometric function needed to evalue the area above
% the given value of t.
%
% K. Beers
% MIT ChE
% 12/9/2001

function [t,iflag] = ...
   calc_t_value(nu,alpha_up);

iflag = 0;

% This is the pre-multiplier of the t-distribution
% that depends solely on the number of degrees of
% freedom.
C = 1/(sqrt(nu)*beta(0.5,nu/2));
a = -(nu+1)/2;


% We make an initial guess of t based on the normal
% distribution limit for many degees of freedom and
% a 95% confidence interval.
t = 1.96;

% We now calculate the area under the curve above
% this value of t.
f = calc_t_value_f(t,C,a,nu,alpha_up);

% We then initialize the secant method by taking
% as the -1 iteration an offset value from t.
t_old = t - sqrt(eps);
f_old = calc_t_value_f(t_old,C,a,nu,alpha_up);

% We now begin the secant method iterations.
max_iter = 100;
atol = 1e-8;
for iter=1:max_iter
   delta_t = - f*(t - t_old)/(f - f_old);
```

```
   t_old = t;
   t = t + delta_t;
   f_old = f;
   f = calc_t_value_f(t,C,a,nu,alpha_up);
   if(abs(f) <= atol)
      iflag = 1;
      break;
   end
end

return;

% ================================================
function f = calc_t_value_f(t,C,a,nu,alpha_up);

t_hi = 10;
if(t > t_hi)
   t_hi = t + 2;
end

% Now, use the symbolic toolkit to calculate the
% area under the t-distribution above the given
% value.
Pint_hi = vpa(C*t_hi*hypergeom([1/2, -a],[3/2],-t_hi^2/nu));
Pint = vpa(C*t*hypergeom([1/2, -a], [3/2], -t^2/nu));
Pint_area = double(Pint_hi - Pint);
f = Pint_area - alpha_up;

return;
```