```
function [beta, chisquare, errors, fitresult] = nlfit00_class1(X, y, model, beta0, varargin)
  % Non-linear fit code using Marquardt-Levenberg Technique.
  % Inputs: X, y, model, beta0, varargin (optional arguments)
  % Outputs: beta, chisquare, errors, fitresult

  % Validate input dimensions
  if length(X) ~= length(y)
    error('X and y must be the same length.');
  end

  % Initialize optional input variables
  a = ones(size(beta0));
  sig = sqrt(abs(y(:)));  % Use absolute value to avoid issues with negative y
  maxiter = 100;         % Default maximum number of iterations
  plotFlag = false;      % Flag for plotting fit progress
  fun_struct = struct();  % Default function structure

  % Handle optional input parameters
  for i = 1:length(varargin)
    switch i
      case 1, a = varargin{i};
      case 2, sig = varargin{i};
      case 3, fun_struct = varargin{i};
      case 4, maxiter = varargin{i};
      case 5, plotFlag = varargin{i};
    end
  end

  % Initial setup
  n = length(y);
  X = X(:); y = y(:); beta0 = beta0(:); sig = sig(:);
  sig(sig == 0) = 1e-6;   % Avoid division by zero in sig
  A = find(a(:));
  p = length(A);         % Number of parameters being fitted

  % Initialize fitting variables
  J = zeros(n, p);
  beta = beta0;
  betanew = beta0;
  betanew(A) = beta(A) * 1.01;
  iter = 0;
  betatol = 1e-4;
  rtol = 1e-4;
  sse = 1;
```

```
sseold = 1;
lambda = 0.01;

% Iterative fitting process
while ((any(abs((betanew - beta) ./ (beta + eps)) > betatol)) || ...
      ((sseold - sse) / (sse + eps) > rtol)) && ...
     (iter < maxiter)

   if iter > 0, beta = betanew; end
   iter = iter + 1;

   % Evaluate model
   yfit = feval(model, beta, X);
   r = (y - yfit) ./ sig;
   sseold = r' * r;

   % Calculate Jacobian
   for k = 1:p
      J(:, k) = nlfit_deriv(model, beta, X, yfit, A(k), numel(varargin) + 3, fun_struct) ./ sig;
   end

   % Levenberg-Marquardt adjustment
   JJ = J' * J;
   Jr = J' * r;
   stepLM = (JJ - diag(diag(JJ)) + JJ .* (eye(p) * (1 + lambda))) \ Jr;
   betaLM = beta;
   betaLM(A) = beta(A) + stepLM;

   % Evaluate new model
   yfitnew = feval(model, betaLM, X);
   rnew = (y - yfitnew) ./ sig;
   sseLM = rnew' * rnew;

   % Adjust step size
   iter1 = 0;
   while sseLM > sseold && iter1 < 12
      stepLM = stepLM / sqrt(10);
      betaLM(A) = beta(A) + stepLM;
      yfitnew = feval(model, betaLM, X);
      rnew = (y - yfitnew) ./ sig;
      sseLM = rnew' * rnew;
      iter1 = iter1 + 1;
   end
```

```
        % Update parameters
        if iter1 < 12
            lambda = lambda / 2;
            betanew = betaLM;
            sse = sseLM;
        else
            lambda = lambda * 10;
        end

        % Optional plotting
        if plotFlag
            plot(X, y, 'bo', X, yfitnew, 'r-');
            legend('Data', 'Fit');
            xlabel('X');
            ylabel('y');
            title('Nonlinear Fit Progress');
            drawnow;
        end
    end

    % Check for convergence
    if iter == maxiter
        disp('NLINFIT did NOT converge. Returning results from last iteration.');
    end

    % Prepare output values
    chisquare = sse / (n - p);
    fitresult.iterations = iter;
    fitresult.phi = sse;
    fitresult.lambda = lambda;
    fitresult.NumPts = n;
    fitresult.NumParam = p;
    fitresult.yfit = yfitnew;
    fitresult.residual = rnew;
    fitresult.sigma = sig;
    fitresult.J = J;
    errors = nlparci(beta(A), rnew, J, A); % Calculate errors in fit
end


function y = nlfit_deriv(model, beta, X, y, n, nargin_nlfit, fun_struct)
    % nlfit_deriv(beta, X, n): Computes the first derivative of y with respect to beta(n) at
    % points X. This is a subroutine of nlfit.
    %
```

```
    % This function only calculates dy/dbeta at each point. Later in the main
    % routine, they will all be summed and weighted together.

    % Initialize delta to zero and calculate a suitable step size
    delta = zeros(size(beta));
    stepSize = max(sqrt(eps)*abs(beta(n)), eps);
    delta(n) = stepSize;

    % Evaluate the model with perturbed parameters
    if nargin_nlfit > 6
        y1 = feval(model, beta + delta, X, fun_struct);
    else
        y1 = feval(model, beta + delta, X);
    end

    % Evaluate the model with original parameters
    y2 = feval(model, beta, X);

    % Calculate initial derivative
    y = (y1 - y2) / delta(n);

    % Adjust step size if the derivative is zero
    while sum(y) == 0 && delta(n) < 0.01 * abs(beta(n))
        delta(n) = delta(n) * 10;
        if nargin_nlfit > 6
            y1 = feval(model, beta + delta, X, fun_struct);
        else
            y1 = feval(model, beta + delta, X);
        end
        y = (y1 - y2) / delta(n);
    end
end

function [delta, ci] = nlparci(x, f, J, A, confLevel)
    % nlparci: Estimates the confidence intervals on parameters of nonlinear models.
    % Inputs:
    %   x - Parameter estimates from the nonlinear fit
    %   f - Residuals from the fit
    %   J - Jacobian matrix at the solution
    %   A - Array indicating which parameters are active
    %   confLevel - (Optional) Confidence level for intervals (default is 0.68 for 68%)
    %
    % Outputs:
    %   delta - Standard deviation of the parameter estimates
```

```
%   ci - Confidence intervals of the parameter estimates

% Check for necessary inputs
if nargin < 4
   error('Requires at least four inputs: x, f, J, and A.');
end
if nargin < 5
   confLevel = 0.68; % Default confidence level (68%)
end

% Reshape f to a column vector
f = f(:);
[m, n] = size(J);

% Check that the number of observations exceeds the number of parameters
if m <= n
   error('The number of observations must exceed the number of parameters.');
end

% Check for size consistency between x and J
if length(x) ~= n
   error('The length of x must equal the number of columns in J.');
end

% Calculate covariance matrix
[Q, R] = qr(J, 0);
if rcond(R) < 1e-15  % Check for singularity
   warning('Jacobian matrix is close to singular. Results may be inaccurate.');
end
Rinv = R \ eye(size(R));
diag_info = sum((Rinv .* Rinv)')';

% Calculate standard deviation of the parameter estimates
v = m - n;
rmse = sqrt(sum(f .* f) / v);
t_val = tinv(0.5 * (1 + confLevel), v);  % t value for confidence level
delta = sqrt(diag_info) .* rmse * t_val;

% Calculate confidence intervals
ci = [(x(:) - delta) (x(:) + delta)];

% Return only the active parameters if specified
if nargin >= 4 && ~isempty(A)
   delta = delta(A);
```

```
      ci = ci(A, :);
   end
end
```

You are supposed to find those :

1) Flaws

2) Unnecessary lines, or parts

3) Misplaced lines – meaning if you replace the line, the code will work better.

4) Extra things : if you can take necessary actions to make this code work better , for example introducing an option as a structure variable, in which you can add necessary line(s) to create the fitting picture of the data while fitting, that will be just fine.

1. **Introduction of Rigorous Input Validation:**
   In `nlfit00_class1.m`, there's a significant emphasis on input validation, particularly the check to ensure `X` and `y` are of equal length. This is a crucial improvement over `nlfit00_class.m`, as it directly addresses potential data inconsistencies and prevents runtime errors that could arise from mismatched input dimensions.

2. **Streamlining and Optimizing Code:**
   Unnecessary lines and parts appear to have been streamlined in `nlfit00_class1.m`. For instance, the handling of optional parameters is more organized, using a loop and a `switch` statement. This approach is more efficient and readable compared to the less structured method in the original code.

3. **Repositioning and Refining Code Blocks:**
   By repositioning certain lines and restructuring the code, `nlfit00_class1.m` achieves better coherence and efficiency. The iterative fitting process, for instance, is more logically structured. Adjustments in the order of operations and checks within the fitting loop contribute to a more predictable and stable algorithm.

4. **Introduction of New Features and Enhancements:**
   `nlfit00_class1.m` introduces additional features like the `plotFlag`, which enables visualization of the fitting process. This is a significant enhancement as it allows users to visually assess the fit quality, which is particularly useful in educational and research settings. The default initialization

of the `fun_struct` variable is another added feature that enhances the code's flexibility in handling various models.

5. **Improved Documentation and Comments:**
   The revised version includes more detailed and structured documentation, clearly describing the purpose, inputs, and outputs of the function. This makes the code more user-friendly and accessible, particularly for new users or in an educational context.

6. **Enhanced Error Handling in Sub-functions:**
   The sub-functions `nlfit_deriv` and `nlparci` in `nlfit00_class1.m` include better error handling and checks, reflecting an overall improvement in code robustness and reliability.