MATLAB Least Squares for Nonlinear Regression.

Generating a best fit by solving the normal equations is widely used and certainly adequate for many curve-fitting applications in engineering and science. It must be mentioned, however, that the *normal equations*

$$(Z^T Z)a = Z^T y$$

can be ill-conditioned and hence sensitive to roundoff errors.

Two more advanced methods, *QR factorization* and *singular value decomposition*, are more robust in this regard.

QR decomposition. This decomposition, also known as a QR factorization or QU factorization, is a decomposition of a matrix A into a product $A = QR$ of an orthogonal matrix Q (its columns are orthogonal unit vectors meaning $Q^{\mathrm{T}} = \mathrm{Q}^{-1}$) and an upper triangular matrix R (also called right triangular matrix). If A is invertible, then the factorization is unique if we require the diagonal elements of R to be positive.

More generally, for overdetermined systems ($m > n$), we can factor an $m \times n$ matrix A as the product of an $m \times m$ unitary matrix Q and an $m \times n$ upper triangular matrix R. As the bottom $(m - n)$ rows of an $m \times n$ upper triangular matrix consists entirely of zeroes, it is often useful to partition R, or both R and Q:

$$A = QR = Q\begin{pmatrix} R_1 \\ \mathbf{0} \end{pmatrix} = (Q_1 \quad Q_2)\begin{pmatrix} R_1 \\ \mathbf{0} \end{pmatrix} = Q_1 R_1$$

where R_1 is an $n \times n$ upper triangular matrix, $\mathbf{0}$ is an $(m - n) \times n$ zero matrix, $Q1$ is $m \times n$, $Q2$ is $m \times (m - n)$, and $Q1$ and $Q2$ both have orthogonal columns.

There are several approaches to compute QR factorization. The most common ones are: *Gram–Schmidt process, Givens rotations,* and *Householder reflections*.

Note. It is worth mentioning that QR factorization is automatically used in two simple ways within MATLAB. First, for cases where you want to fit a polynomial, the built-in `polyfit` function automatically uses QR factorization to obtain its results. Second, the general linear least-squares problem can be directly solved with the backslash operator.

Singular value decomposition (SVD) is a factorization of a real or complex matrix. It generalizes the eigendecomposition of a square normal matrix with an orthonormal eigenbasis to any $m \times n$ matrix.
Specifically, the singular value decomposition of an $m \times n$ real matrix M is a factorization of the form

$$M = U\Sigma V^T,$$

where U is an $m \times m$ real orthogonal matrix, Σ is an $m \times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal, V is an $n \times n$ real orthogonal matrix.

Nonlinear Regression Using MATLAB.
As with linear least squares, nonlinear regression is based on determining the values of the parameters that minimize the sum of the squares of the residuals. However, for the nonlinear case, the solution must proceed in an iterative fashion.
Assume that we would like to find the best fit to data based on the following nonlinear model:

$$y = g(x, a_1, a_2, \dots, a_m) + e$$

where $a_1, a_2, \dots, a_m$ are the parameters that must be determined so that the sum of squares of residuals (errors) function

$$f(a_1, a_2, \dots, a_m) = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - g(x_i, a_1, a_2, \dots, a_m))^2$$

is minimized. An optimization routine can then be used to determine the values of $a_1, a_2, \dots, a_m$ that minimize the function.

MATLAB's `fminsearch` function can be used for this purpose. It has the general syntax

```
>> [a, fval] = fminsearch(fun,x0,options,p1,p2,...)
```

where `a` = a vector of the values of the parameters that minimize the function fun, `fval` = the value of the function at the minimum, `x0` = a vector of the initial guesses for the parameters, `options` = a structure containing values of the optimization parameters as created with the `optimset` function, and `p1, p2, etc.` = additional arguments that are passed to the objective function.

Example. Assume that we want to fit a function of the form

$$g(a,b,x) = ax^b$$

For the following data set:

$$x = [10\ 20\ 30\ 40\ 50\ 60\ 70\ 80]$$

$$y = [25\ 70\ 380\ 550\ 610\ 1220\ 830\ 1450]$$

Try the following:

```
    function f = fSSR(a,xm,ym)
    yp = a(1)*xm.^a(2);
    f = sum((ym - yp).^2);



    >> x = [10 20 30 40 50 60 70 80];

    >> y = [25 70 380 550 610 1220 830 1450];

    >> fminsearch(@fSSR, [1, 1], [], x, y)
```

Newton's interpolating polynomial: It is straightforward to develop an M-file to implement Newton interpolation.

```
    function yint = Newtint1(x,y,xx)
    % Newtint: Newton interpolating polynomial
    % yint = Newtint(x,y,xx): Uses an (n ? 1)?order Newton
    % interpolating polynomial based on n data points (x, y)
    % to determine a value of the dependent variable (yint)
    % at a given value of the independent variable, xx.
    % input:
    % x = independent variable
    % y = dependent variable
    % xx = value of independent variable at which
    % interpolation is calculated
    % output:
```

```
% yint = interpolated value of dependent variable
% compute the finite divided differences in the form of a
% difference table
n = length(x);
if length(y)~=n,
    error('x and y must be same length');
end
b = zeros(n,n);
% assign dependent variables to the first column of b.
b(:,1) = y(:); % the (:) ensures that y is a column vector.
for j = 2:n
    for i = 1:n-j+1
        b(i,j) = (b(i+1,j-1)-b(i,j-1))/(x(i+j-1)-x(i));
    end
end
% use the finite divided differences to interpolate
xt = 1;
yint = b(1,1);
for j = 1:n-1
    xt = xt*(xx-x(j));
    yint = yint+b(1,j+1)*xt;
end
```

```
>> x = [1 4 6 5]';

>> y = log(x);

>> Newtint(x,y,2)
```