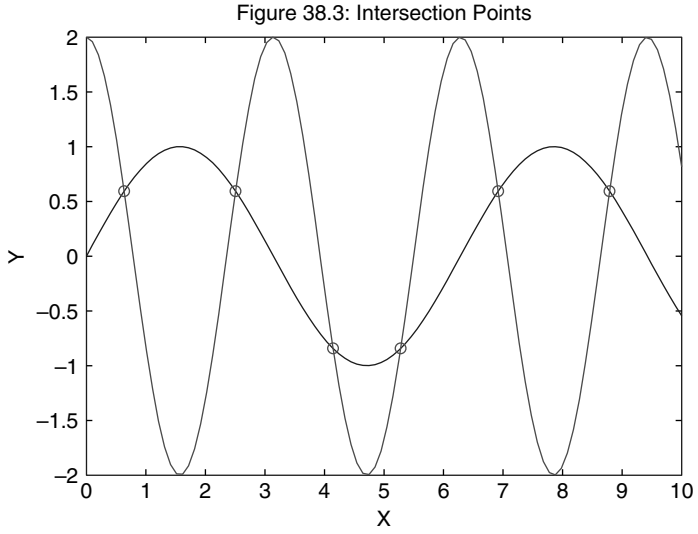
```
>> xo = mminvinterp(x,y-z,0); % find zero crossings of difference

>> yo = interp1(x,y,xo);      % find corresponding y values

>> plot(x,y,x,z,xo,yo,'o')    % regenerate plot showing intersection points
>> xlabel X
>> ylabel Y
>> title 'Figure 38.3: Intersection Points'
```

Figure 38.3: Intersection Points



38.9 POLYNOMIAL CURVE FITTING

As discussed in Chapter 20, polynomial curve fitting is performed by using the function `polyfit`. Because polynomial curve fitting is such a basic numerical analysis topic, it is worth exploring more fully. Consider a general polynomial, written as

$$y = p_1 x^n + p_2 x^{n-1} + \cdots + p_n x + p_{n+1}$$

Here, there are $n + 1$ coefficients for an nth order polynomial. For convenience, the coefficients have subscripts that are numbered in increasing order as the power of x decreases. Written in this way, the polynomial can be written as the matrix product

$$y = [x^n \quad x^{n-1} \quad \cdots \quad x \quad 1] \cdot \begin{bmatrix} p_1 \\ p_2 \\ \dots \\ p_n \\ p_{n+1} \end{bmatrix}$$

In this format, the polynomial coefficients are grouped into a single column vector.

Common polynomial curve fitting uses this form to compute the polynomial coefficients, given a set of data points $\{x_i, y_i\}$ for $i = 1, 2, \dots N$. Substituting each of these data points into the preceding relationship and grouping the results leads to the matrix equation

$$\begin{bmatrix} x_1^n & x_1^{n-1} & \cdots & x_1 & 1 \\ x_2^n & x_2^{n-1} & \cdots & x_2 & 1 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ x_N^n & x_N^{n-1} & \cdots & x_N & 1 \end{bmatrix} \cdot \begin{bmatrix} p_1 \\ p_2 \\ \cdots \\ p_n \\ p_{n+1} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$$

As written, the matrix on the left is a Vandermonde matrix, as discussed earlier in Section 38.4. If $N = n + 1$, the Vandermonde matrix is square. Furthermore, if the data points are distinct, the matrix has full rank, and the unique polynomial vector $p = [p_1\ p_2 \cdots p_n\ p_{n+1}]'$ is found in MATLAB by using the backslash or left-division operator as `p = V\y`, where `V` is the Vandermonde matrix and `y` is the right-hand side vector.

On the other hand, when $N \geq n + 1$ and the data points are distinct, the Vandermonde matrix has more rows than columns, and no exact solution exists. In this case, `p = V\y` in MATLAB computes the polynomial coefficient vector that minimizes the least squared error in the set of equations. For example, the following code segment duplicates the example shown in Figure 20.2, without the use of `polyfit`:

```
% polyfit1.m
% find polynomial coefficients without polyfit

x = [0 .1 .2 .3 .4 .5 .6 .7 .8 .9 1]';  % column vector data
y = [-.447 1.978 3.28 6.16 7.08 7.34 7.66 9.56 9.48 9.30 11.2]';

n = 2; % desired polynomial order

pm = polyfit(x,y,n) % MATLAB polyfit result
```

```
% create Vandermonde matrix using code from vander3.m
m = length(x);   % number of elements in x
V = ones(m,n+1); % preallocate memory for result

for i=n:-1:1     % build V column by column
   V(:,i) = x.*V(:,i+1);
end

p = V\y; % find least squares solution

p = p'   % convert to row vector to match MATLAB's convention for polynomials
```

Running this code shows that both polynomial vectors are equal:

```
>> polyfit1
pm =
      -9.8108       20.129    -0.031671
p =
      -9.8108       20.129    -0.031671
```

From this basic understanding, it is possible to consider potential numerical problems. In particular, because the Vandermonde matrix contains elements ranging from 1 to x^n, it can suffer accuracy problems if the x data points differ a great deal from the number 1. For example, if the desired polynomial order is increased to 4, and the x data is scaled by 10^4, the Vandermonde matrix becomes

```
>> x = 1e4*[0 .1 .2 .3 .4 .5 .6 .7 .8 .9 1]'; % scale x data by 1e4

>> n = 4;                                     % change order to 4
>> m = length(x);   % number of elements in x
>> V = ones(m,n+1); % preallocate memory for result
>> for i=n:-1:1     % build V column by column
   V(:,i) = x.*V(:,i+1);
end
```

```
>> V
V =
            0            0            0            0            1
       1e+012       1e+009       1e+006         1000            1
     1.6e+013       8e+009       4e+006         2000            1
     8.1e+013     2.7e+010       9e+006         3000            1
    2.56e+014     6.4e+010     1.6e+007         4000            1
    6.25e+014    1.25e+011     2.5e+007         5000            1
   1.296e+015    2.16e+011     3.6e+007         6000            1
   2.401e+015    3.43e+011     4.9e+007         7000            1
   4.096e+015    5.12e+011     6.4e+007         8000            1
   6.561e+015    7.29e+011     8.1e+007         9000            1
       1e+016       1e+012       1e+008        10000            1
```

Now the values in V vary in value from 1 to 10^{16}. This disparity causes trouble for the function polyfit:

```
>> p = polyfit(x,y,n)
  Warning: Rank deficient, rank = 4,  tol =   3.1633e+001.
  (Type "warning off MATLAB:rankDeficientMatrix" to suppress this warning.)
p =
  3.0675e-015 -4.8147e-011  9.5904e-008    0.0018927            0
```

Since we have reached the limits of double-precision mathematics, to eliminate this problem we must somehow scale the data so that the Vandermonde matrix does not exhibit this disparity in values. Of the numerous ways to scale the data, use of the mean and standard deviation often leads to good results. That is, instead of fitting a polynomial to the data in x, the fit is done with respect to a new independent variable z, given by

$$z = \frac{x - x_m}{s}$$

where x_m and s are the mean and standard deviation, respectively, of the x data. Subtracting by the mean shifts the data to the origin, and dividing by the standard deviation reduces the spread in the data values. Applying this to the previous data that were scaled by 10^4 gives

```
>> xm = mean(x)
>> s = std(x)
>> z = (x - xm)/s;
```

```
>> m = length(z);   % number of elements in x
>> V = ones(m,n+1); % preallocate memory for result
>> for i=n:-1:1     % build V column by column
   V(:,i) = z.*V(:,i+1);
end
>>V
V =
       5.1653      -3.4263       2.2727      -1.5076            1
       2.1157      -1.7542       1.4545       -1.206            1
      0.66942     -0.74007      0.81818     -0.90453            1
      0.13223     -0.21928      0.36364     -0.60302            1
    0.0082645     -0.02741     0.090909     -0.30151            1
            0            0            0            0            1
    0.0082645      0.02741     0.090909      0.30151            1
      0.13223      0.21928      0.36364      0.60302            1
      0.66942      0.74007      0.81818      0.90453            1
       2.1157       1.7542       1.4545        1.206            1
       5.1653       3.4263       2.2727       1.5076            1
```

Now the Vandermonde matrix is numerically well conditioned, and the polynomial curve fit proceeds without difficulty. For this data, the resulting polynomial is

```
>> p = (V\y)'
p =
      0.26689      0.58649      -1.6858       2.4732       7.7391
```

Comparing these polynomial coefficients to those computed before scaling shows that these are much better scaled as well.

While the previous result works well, it changes the original problem to

$$y = p_1 z^n + p_2 z^{n-1} + \cdots + p_n z + p_{n+1}$$

That is, the polynomial is now a function of the variable z. Because of this, evaluation of the original polynomial requires a two-step process. First, using the values of x_m and s used to find z, x data points for polynomial evaluation must be converted to their z data equivalents by using $z = (x - x_m)/s$. Then, the polynomial can be evaluated, as in the following code:

```
>> xi = 1e4*[.25 .35 .45]; % sample data for polynomial evaluation
>> zi = (xi-xm)/s;         % convert to z data


>> yi = polyval(p,zi)      % evaluate using polyval
yi =
        4.752       6.2326        7.326
```

Rather than performing this step manually when data scaling is used, the MATLAB functions `polyfit` and `polyval` implement this data scaling through the use of an additional variable. For example, the previous results are duplicated by the following code:

```
>> [p,Es,mu] = polyfit(x,y,n);
>> p
p =
      0.26689      0.58649      -1.6858       2.4732       7.7391
>> yi = polyval(p,xi,Es,mu)
yi =
        4.752       6.2326        7.326
```

The optional `polyfit` output variable `mu` contains the mean and the standard deviation of the data used to compute the polynomial. Providing this variable to `polyfit` directs it to perform the conversion to `z` before evaluating the polynomial. As shown here, the polynomial `p` and the interpolated points `yi` are equal to those just computed by using the Vandermonde matrix.

The structure variable `Es` (which has not been discussed here) is used to compute error estimates in the solution. (See the documentation for further information regarding this variable.)

Beyond a consideration of data scaling, it is sometimes important to perform weighted polynomial curve fitting. That is, in some situations, there may be more confidence in some of the data points than in others. When this is true, the curve fits procedure should take this confidence into account and return a polynomial that reflects the weight given to each data point.

While `polyfit` does not provide this capability, it is easy to implement in a number of ways. Perhaps the simplest way is to weigh the data before the Vandermonde matrix is formed. For example, consider the case of a third-order polynomial being fit to five data points. Assuming that the confidence in the third data point is α times that of all other data points, the matrix equation to be solved becomes

$$\begin{bmatrix} x_1^3 & x_1^2 & x_1 & 1 \\ x_2^3 & x_2^2 & x_2 & 1 \\ \alpha \cdot x_3^3 & \alpha \cdot x_3^2 & \alpha \cdot x_3 & \alpha \\ x_4^3 & x_4^2 & x_4 & 1 \\ x_5^3 & x_5^2 & x_5 & 1 \end{bmatrix} \cdot \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \alpha \cdot y_3 \\ y_4 \\ y_5 \end{bmatrix}$$

By multiplying the third row of the matrix equation by α, the error in this equation is weighted by α. The process of minimizing the least squared error forces the error at this data point to decrease relative to the others. As α increases, the error at the data point decreases.

In general, it is possible to give a weight to each data point, not just to one of them, as previously shown. Implementing this approach in MATLAB is illustrated in the following code segment:

```
% polyfit2.m
% find weighted polynomial coefficients

x = [0 .1 .2 .3 .4 .5 .6 .7 .8 .9 1]';
y = [-.447 1.978 3.28 6.16 7.08 7.34 7.66 9.56 9.48 9.30 11.2]';

n = 3;

pm = polyfit(x,y,n) % MATLAB polyfit result

% create Vandermonde matrix using code from vander3.m

m = length(x);   % number of elements in x

V = ones(m,n+1); % preallocate memory for result

for i=n:-1:1     % build V column by column
   V(:,i) = x.*V(:,i+1);
end

w = ones(size(x)); % default weights of one
w(4) = 2;          % weigh 4th point by 2
w(7) = 10;         % weigh 7th point by 10
```

```
V = V.*repmat(w,1,n+1); % multiply rows by weights
y = y.*w;               % multiply y by weights

p = (V\y)' % find polynomial
```

Running this code shows that the unweighted and weighted polynomials are different:

```
>> polyfit2
pm =
       16.076      -33.924       29.325      -0.6104
p =
       28.576      -50.761       34.104     -0.67441
```

An alternative to this approach uses the MATLAB function `lscov`, which specifically computes weighted least squares solutions. Repeating the previous example by using this function is straightforward, as shown in the following code segment and its associated output:

```
% polyfit3
% find weighted polynomial coefficients using lscov

x = [0 .1 .2 .3 .4 .5 .6 .7 .8 .9 1]';
y = [-.447 1.978 3.28 6.16 7.08 7.34 7.66 9.56 9.48 9.30 11.2]';

n = 3;

pm = polyfit(x,y,n) % MATLAB polyfit result

% create Vandermonde matrix using code from vander3.m

m = length(x);   % number of elements in x
V = ones(m,n+1); % preallocate memory for result

for i=n:-1:1     % build V column by column
   V(:,i) = x.*V(:,i+1);
end
```

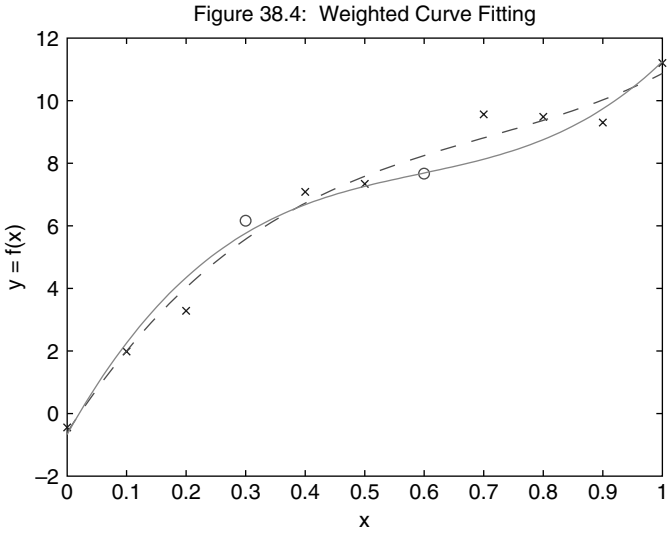
```
w = ones(size(x)); % default weights of one
w(4) = 2^2;        % here weights are the square of those used in polyfit2
w(7) = 10^2;

p = lscov(V,y,w)'
```

```
>> polyfit3
pm =
       16.076      -33.924       29.325      -0.6104
p =
       28.576      -50.761       34.104     -0.67441
```

To see how these polynomials differ, they can be plotted over the range of the data:

```
% mm3804.m
% find weighted polynomial coefficients using lscov

x = [0 .1 .2 .3 .4 .5 .6 .7 .8 .9 1]';
y = [-.447 1.978 3.28 6.16 7.08 7.34 7.66 9.56 9.48 9.30 11.2]';
n = 3;
pm = polyfit(x,y,n); % MATLAB polyfit result

% create Vandermonde matrix using code from vander3.m

m = length(x);   % number of elements in x
V = ones(m,n+1); % preallocate memory for result

for i=n:-1:1     % build V column by column
   V(:,i) = x.*V(:,i+1);
end

w = ones(size(x)); % default weights of one
w(4) = 2^2;
w(7) = 10^2;
```

```
p = lscov(V,y,w)';

xi = linspace(0,1,100);
ym = polyval(pm,xi);
yw = polyval(p,xi);

pt = false(size(x)); % logical array pointing to weighted points
pt([4 7]) = true;

plot(x(~pt),y(~pt),'x',x(pt),y(pt),'o',xi,ym,'--',xi,yw)
xlabel('x'), ylabel('y=f(x)')
title('Figure 38.4: Weighted Curve Fitting')
```



The weighted points are marked with circles in the figure, while the unweighted points are marked with an `x`. Clearly, the weighted polynomial is closer to the fourth and seventh data points than the original `polyfit` results. In addition, the seventh data point is closer than the fourth because its weight is much higher.

38.10 NONLINEAR CURVE FITTING

The polynomial curve fitting discussed in the previous section is popular in part because the problem can be written in matrix form and solved by using linear least-squares techniques. Perhaps most important, or most convenient, is that no initial

guess is required. The optimum solution is found without searching an n-dimensional solution space. In the more general case, where the unknown parameters do not appear linearly (i.e., nonlinear curve fitting), the solution space must be searched, starting with an initial guess. For example, consider fitting data to the function

$$f(t) = a + b\mathrm{e}^{\alpha t} + ce^{\beta t}$$

If α and β are known constants and a, b, and c are the unknowns to be found, this function can be written as

$$f(t) = [1 \quad \mathrm{e}^{\alpha t} \quad \mathrm{e}^{\beta t}] \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

Substituting a given a set of data points, $\{t_i, y_i = f(t_i)\}$ for $i = 1, 2, \ldots N$, into this expression and gathering the results in matrix format results in the equation

$$\begin{bmatrix} 1 & \mathrm{e}^{\alpha t_1} & \mathrm{e}^{\beta t_1} \\ 1 & \mathrm{e}^{\alpha t_2} & \mathrm{e}^{\beta t_2} \\ \vdots & \vdots & \vdots \\ 1 & \mathrm{e}^{\alpha t_N} & \mathrm{e}^{\beta t_N} \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$$

Written in this way, a, b, and c appear linearly, just as the polynomial coefficients did in the previous section. As a result, the backslash operation in MATLAB produces the least-squares solution for the unknown variables. That is, if E is the N-by-3 matrix on the left, p is the vector $[a\, b\, c]'$, and y is the right-hand side vector, then `p = E\y` gives the least-squares solution for p.

When α and β are not known, the linear least-squares solution does not apply, because α and β cannot be separated as a, b, and c are in the preceding equation. In this case, it is necessary to use a minimization algorithm such as `fminsearch` in MATLAB. There are two ways to set up this problem for solution with `fminsearch`. The first is to write $f(t)$ as

$$f(t) = x_3 + x_4\, \mathrm{e}^{x_1 t} + x_5\, e^{x_2 t}$$

where the unknown variables have been combined into a vector $x = [x_1\, x_2\, x_3\, x_4\, x_5]'$. Given this form and a set of data points $\{t_i, y_i\}$ for $i = 1, 2, \ldots N$, the least-squares solution minimizes the norm of error between the data points y_i and the function evaluated at t_i, $f(t_i)$. That is, if y is a vector containing the y_i data points and if f is a vector containing the previous function evaluated at the time points t_i, then the least-squares solution is given by

$$\min_x \|y - f\|$$

To use fminsearch, this norm must be computed in a function as follows:

```
function enorm=fitfun1(x,tdata,ydata)
%ENORM Norm of fit to example nonlinear function
% f(t) = x(3)+x(4)*exp(x(1)*t)+x(5)*exp(x(2)*t)
%
% ENORM(X,Tdata,Ydata) returns norm(Ydata-f(Tdata))

f = x(3)+x(4)*exp(x(1)*tdata)+x(5)*exp(x(2)*tdata);

enorm = norm(f-ydata);
```

The following code segment tests this approach by creating some data, making an initial guess, and calling fminsearch to find a solution:

```
% testfitfun1
% script file to test nonlinear least squares problem

% create test data
x1 = -2; % alpha
x2 = -5; % beta
x3 = 10;
x4 = -4;
x5 = -6;

tdata = linspace(0,4,30)';
ydata = x3+x4*exp(x1*tdata)+x5*exp(x2*tdata);

% create an initial guess

x0 = zeros(5,1); % not a good one, but a common first guess

% call fminsearch

fitfun = @fitfun1; % create handle

options = []; % take default options
```

```
x = fminsearch(fitfun,x0,options,tdata,ydata)

% compute error norm at returned solution

enorm = fitfun(x,tdata,ydata)
```

Running this code produces the following output:

```
>> testfitfun1
Exiting: Maximum number of function evaluations has been exceeded
         - increase MaxFunEvals option.
         Current function value: 3.234558
x =
      0.17368
      -1.5111
       13.972
      -2.2059
      -9.4466


enorm =
       3.2346
```

For this initial guess, the algorithm has not converged. Increasing the number of function evaluations and algorithm iterations permitted to 2000 leads to

```
>> options = optimset('MaxFunEvals',2e3,'MaxIter',2e3);


x = fminsearch(fitfun,x0,options,tdata,ydata)
x =
     -0.78361
      -3.7185
       10.061
     -0.80533
      -9.1907
```

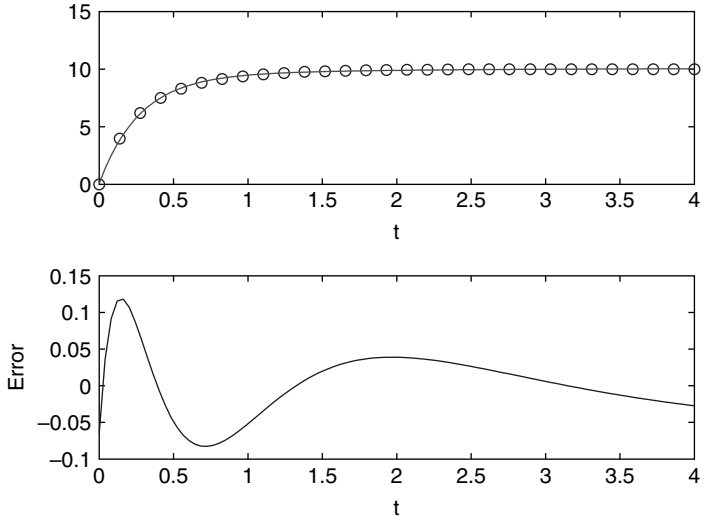
```
enorm =
  0.23727
```

The algorithm now converges, but not to the values $x = [-2\ -5\ 10\ -4\ -6]'$ used to create the data. A plot of the data, the actual function, and the fitted solution provides further information, computed and displayed as follows:

```
% testfitfun2
% script file to test nonlinear least squares problem

% create test data
x1 = -2; % alpha
x2 = -5; % beta
x3 = 10;
x4 = -4;
x5 = -6;
tdata = linspace(0,4,30)';
ydata = x3+x4*exp(x1*tdata)+x5*exp(x2*tdata);

% create an initial guess

x0 = zeros(5,1); % not a good one, but a common first guess

% call fminsearch

fitfun = @fitfun1; % create handle

options = optimset('MaxFunEvals',2e3,'MaxIter',2e3);

x = fminsearch(fitfun,x0,options,tdata,ydata);

ti = linspace(0,4); % evaluation points
actual = x3+x4*exp(x1*ti)+x5*exp(x2*ti); % actual function

fitted = x(3)+x(4)*exp(x(1)*ti)+x(5)*exp(x(2)*ti); % fitted solution
subplot(2,1,1)
plot(tdata,ydata,'o',ti,actual,ti,fitted)
xlabel t
title 'Figure 38.5: Nonlinear Curve Fit'
```

```
subplot(2,1,2)
plot(ti,actual-fitted)
xlabel t
ylabel Error
```

Figure 38.5: Nonlinear Curve Fit



The fitted solution in the upper plot looks good visually. However, the lower error plot shows that there is significant error. If the overall goal of this problem is to minimize the error between the actual coefficients represented by the data and the results returned by `fminsearch`, then this algorithm essentially failed, even though the fitted solution plot looks good visually.

At this point, you must decide whether this solution is satisfactory. Certainly, you can try setting tighter convergence criteria, as in the following code:

```
>> options=optimset('TolX',1e-10,'TolFun',1e-10,'MaxFunEvals',2e3,'MaxIter',2e3);

>> x = fminsearch(fitfun,x0,options,tdata,ydata)
x =
     -0.78359
      -3.7185
       10.061
```

```
     -0.80533
      -9.1907
>> enorm = fitfun(x,tdata,ydata)
enorm =
      0.23727
```

In this case, the solution has not changed. You can also try better initial guesses, such as the following:

```
>> x0 = [-1 -4 8 -3 -7]'; % much closer initial guess

>> options = []; % default options

>> x = fminsearch(fitfun,x0,options,tdata,ydata)
x =
           -2
           -5
           10
           -4
           -6
>> enorm = fitfun(x,tdata,ydata)
enorm =
  3.2652e-006
```

In this case, `fminsearch` returns coefficients that very closely match those used to create the data. This example illustrates an ambiguity inherent in most nonlinear optimization algorithms. That is, other than having the error norm be zero, there is no set way of knowing when a solution is the best that can be expected.

When a nonlinear curve fitting problem contains a mixture of linear and nonlinear terms, as is true in this example, it is possible to use linear least squares to compute the linear terms and use a function such as `fminsearch` to compute the nonlinear terms. This leads to better results in almost all cases.

To understand how this is done, consider the problem formulation stated earlier, where α and β were known constants:

$$\begin{bmatrix} 1 & e^{\alpha t_1} & e^{\beta t_1} \\ 1 & e^{\alpha t_2} & e^{\beta t_2} \\ \vdots & \vdots & \vdots \\ 1 & e^{\alpha t_N} & e^{\beta t_N} \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$$

In this case, α and β become the variables manipulated by `fminsearch`. Within the function that evaluates the error norm, this linear least-squares problem is solved by using the current estimates for α and β. That is, if $x_1 = \alpha$, $x_2 = \beta$, and $p = [a\ b\ c]'$, then `fitfun1` is rewritten as follows:

```
function [enorm,p]=fitfun2(x,tdata,ydata)
%ENORM Norm of fit to example nonlinear function
% f(t) = p(1)+p(2)*exp(x(1)*t)+p(3)*exp(x(2)*t)
%
% ENORM(X,Tdata,Ydata) returns norm(Ydata-f(Tdata))
%
% [e,p]=ENORM(...) returns the linear least squares
%                  parameter vector p

% solve linear least squares problem given x input supplied by fminsearch

E = [ones(size(tdata)) exp(x(1)*tdata) exp(x(2)*tdata)];
p = E\ydata; % least squares solution for p=[a b c]'

% use p vector to compute error norm
f = p(1)+p(2)*exp(x(1)*tdata)+p(3)*exp(x(2)*tdata);

enorm = norm(f-ydata);
```

Now there are only two parameters for `fminsearch` to manipulate. The following code segment tests this alternative approach:

```
% testfitfun3
% script file to test nonlinear least squares problem

% create test data
x1 = -2; % alpha
x2 = -5; % beta
p1 = 10;
p2 = -4;
p3 = -6;
```

```
tdata = linspace(0,4,30)';
ydata = p1+p2*exp(x1*tdata)+p3*exp(x2*tdata);

% create an initial guess

x0 = zeros(2,1); % only two parameters to guess now!

% call fminsearch

fitfun = @fitfun2; % create handle to new function

options = []; % take default options

x = fminsearch(fitfun,x0,options,tdata,ydata)

% find p and compute error norm at returned solution

[enorm,p] = fitfun(x,tdata,ydata)
```

Running this code produces the following output:

```
>> testfitfun3
x =
           -2
           -5
enorm =
  7.1427e-006
p =
           10
           -4
           -6
```

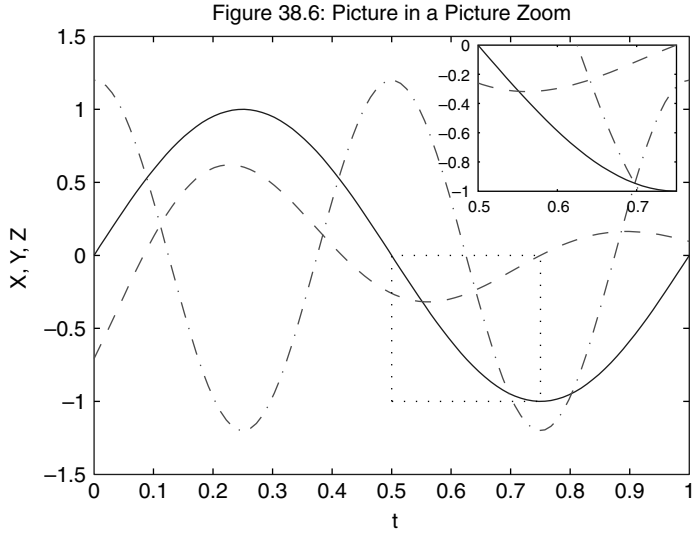
Without any difficulty, `fminsearch` finds a minimum very close to the actual values. The incorporation of an internal linear least-squares algorithm within a nonlinear problem reduces dimension of the parameter space to be searched. This almost always improves the speed and convergence of the nonlinear algorithm. In the preceding example, the problem was reduced from five dimensions to two.

Finally, note how the parameter `p` was included as a second output argument to `fitfun2`. The minimization algorithm `fminsearch` ignores this argument, so it does not influence the search process. However, after the algorithm terminates, calling the `fitfun2` again with two output arguments returns the values of the linear parameters at the solution point. Global variables should not be used to obtain `p`, since there is no way of knowing if `fminsearch` evaluates `fitfun2` at its solution on its last iteration. Calling `fitfun2` after `fminsearch` finishes execution guarantees that the parameter vector `p` is evaluated at the solution returned by `fminsearch`.

38.11 PICTURE-IN-A-PICTURE ZOOM

The final example in this chapter demonstrates use of the Handle Graphics features in MATLAB. As illustrated in the next figure, this example implements a picture-in-a-picture zoom. That is, the user calls the function, then goes to the current axes and, by using the mouse, drags a selection rectangle. Once created, a dotted outline of the selection rectangle remains, and a new, but smaller, axis is created showing the graphical contents inside of the drawn selection rectangle. This function allows the user to zoom into a portion of an axis without hiding the original plot. Although not shown, the smaller axis can be selected, dragged, and resized.

Figure 38.6: Picture in a Picture Zoom



There are a number of steps involved in the creation of this function. Rather than implement the function as one large function, it is convenient to create supporting functions first that could be subfunctions of the primary functions or simply