華中科技大學
Huazhong University of Science and Technology

数据科学基础
FUNDATIONS OF DATA SCIENCE

Lecture 12: Curve Fitting

- Analyzing data is fundamental to any aspect of science. Often data can be noisy in nature and only the trends in the data are sought. A variety of curve fitting (曲线拟合) schemes can be generated to provide simplified descriptions of data and its behavior. The **least-squares (最小二乘) fit method** is explored along with fitting methods of polynomial fits (多项式拟合) and splines(曲线).

Least-Square Fitting Methods

One of the fundamental tools for data analysis and recognizing trends in physical systems is **curve fitting(曲线拟合)**. The concept of curve fitting is fairly simple: use a simple function to describe a trend by minimizing the error between the selected function to fit and a set of data. The mathematical aspects of this are laid out in this section.

Suppose we are given a set of *n* data points

$$(x_1, y_1),\ (x_2, y_2),\ (x_3, y_3),\ \cdots\ ,(x_n, y_n)$$

Further, assume that we would like to fit a best fit line through these points. Then we can approximate the line by the function

$$f(x) = Ax + B$$

where the constants *A* and *B* are chosen to minimize some error.

Thus the function gives an approximation to the true value which is off by some error so that

$$f(x_k) = y_k + E_k$$

where y_k is the true value and E_k is the error from the true value.

Various error measurements can be minimized when approximating with a given function *f(x)*. Three standard possibilities are given as follows

1. MaxmumError $\quad E_\infty(f) = \max_{1<k<n} |f(x_k) - y_k|.$
2. AverageError $\quad E_1(f) = \frac{1}{n}\sum_{k=1}^{n} |f(x_k) - y_k|.$
3. Root-meanSqure $\quad E_2(f) = \left(\frac{1}{n}\sum_{k=1}^{n} |f(x_k) - y_k|^2\right)^{1/2}$

In practice, the **root-mean square error** is most widely used and accepted. Thus when fitting a curve to a set of data, the root-mean square error is chosen to be minimized. This is called a **least-square fit**.

Figure 3.1 depicts three line fits for the errors E_∞, E_1 and E_2 listed above. The E_∞ error line fit is strongly influenced by the one data point which does not fit the trend. The E_1 and E_2 line fit nicely through the bulk(大量) of the data.

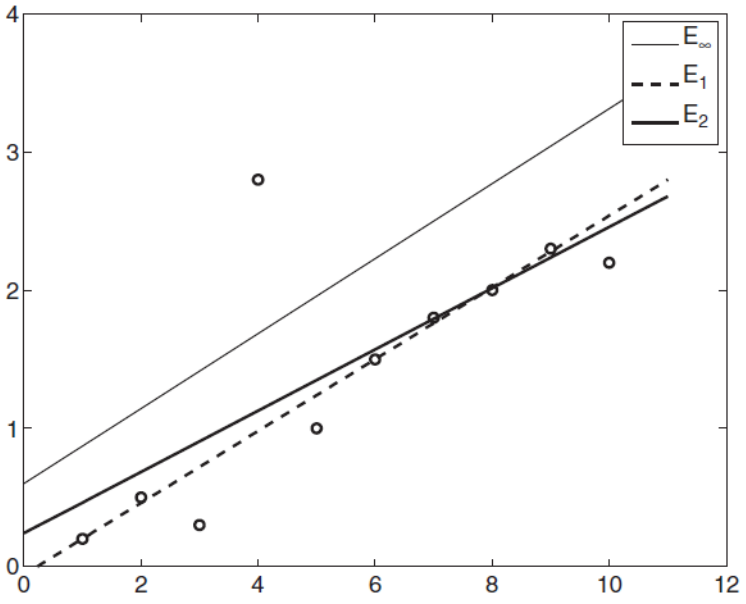


Figure 3.1: Linear fit $Ax + B$ to a set of data (open circles) for the three error definitions given by E_∞, E_1 and E_2. The weakness of the E_∞ fit results from the one stray data point at $x = 4$.

Least-squares line

To apply the least-square fit criteria, consider the data points $\{x_k, y_k\}$, where $k = 1, 2, 3, . . . , n$. To fit the curve

$$f(x) = Ax + B$$

to this data, the E_2 is found by minimizing the sum

$$E_2(f) = \sum_{k=1}^{n} |f(x_k) - y_k|^2 = \sum_{k=1}^{n} (Ax_k + B - y_k)^2$$

Minimizing this sum requires differentiation (微分). Specifically, the constants A and B are chosen so that a minimum occurs; thus we require: $\partial E_2/\partial A = 0$ and $\partial E_2/\partial B = 0$. Note that although a zero derivative can indicate either a minimum or maximum, we know this must be a minimum to the error since there is no maximum error. The minimization condition gives:

$$\frac{\partial E_2}{\partial A} = 0 : \sum_{k=1}^{n} 2(Ax_k + B - y_k)x_k = 0$$

$$\frac{\partial E_2}{\partial B} = 0 : \sum_{k=1}^{n} 2(Ax_k + B - y_k) = 0 \,.$$

Upon rearranging, the 2 × 2 system of linear equations is found for A and B:

$$\begin{pmatrix} \sum_{k=1}^{n} x_k^2 & \sum_{k=1}^{n} x_k \\ \sum_{k=1}^{n} x_k & n \end{pmatrix} \begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} \sum_{k=1}^{n} x_k y_k \\ \sum_{k=1}^{n} y_k \end{pmatrix}$$

This equation can be easily solved using the *backslash* command in MATLAB.

This method can be easily generalized to higher polynomial fits. In particular, a parabolic fit (抛物线拟合) to a set of data requires the fitting function

$$f(x) = Ax^2 + Bx + C$$

where now the three constants *A*, *B* and *C* must be found. These can be found from the *3* × *3* system which results from minimizing the error $E_2(A, B, C)$ by taking

$$\frac{\partial E_2}{\partial A} = 0$$
$$\frac{\partial E_2}{\partial B} = 0$$
$$\frac{\partial E_2}{\partial C} = 0$$

Data linearization

Although a powerful method, the minimization procedure can result in equations which are nontrivial to solve. Specifically, consider fitting data to the **exponential function (指数函数)**

$$f(x) = C\exp(Ax)$$

The error to be minimized is

$$E_2(A, C) = \sum_{k=1}^{n} (C\exp(Ax_k) - y_k)^2$$

Applying the minimizing conditions leads to

$$\frac{\partial E_2}{\partial A} = 0 : \sum_{k=1}^{n} 2(C\exp(Ax_k) - y_k)Cx_k\exp(Ax_k) = 0$$

$$\frac{\partial E_2}{\partial C} = 0 : \sum_{k=1}^{n} 2(C\exp(Ax_k) - y_k)\exp(Ax_k) = 0\,.$$

This in turn leads to the 2 × 2 system

$$C\sum_{k=1}^{n} x_k \exp(2Ax_k) - \sum_{k=1}^{n} x_k y_k \exp(Ax_k) = 0$$

$$C\sum_{k=1}^{n} \exp(Ax_k) - \sum_{k=1}^{n} y_k \exp(Ax_k) = 0\,.$$

This system of equations is nonlinear and cannot be solved in a straightforward fashion. Indeed, a solution may not even exist. Or many solution may exist.

To avoid the difficulty of solving this nonlinear system, the exponential fit (指数拟合) can be linearized by the transformation

$$Y = \ln(y)$$
$$X = x$$
$$B = \ln C$$

Then the fit function

$$f(x) = y = C\exp(Ax)$$

can be linearized by taking the natural log of both sides so that

$$\ln y = \ln(C\exp(Ax)) = \ln C + \ln(\exp(Ax)) = B + Ax \rightarrow \ Y = AX + B$$

So by fitting to the natural log of the y-data

$$(x_i, y_i) \rightarrow (x_i, \ln y_i) = (X_i, Y_i)$$

the curve fit for the exponential function becomes a linear fitting problem which is easily handled.

General fitting

Given the preceding examples, a theory can be developed for a general fitting procedure.

The key idea is to assume a form of the fitting function

$$f(x) = f(x, C_1, C_2, C_3, \cdots, C_M)$$

where the C_i are constants used to minimize the error and $M < n$. The root-mean square error is then

$$E_2(C_1, C_2, C_3, \cdots, C_m) = \sum_{k=1}^{n} (f(x_k, C_1, C_2, C_3, \cdots, C_M) - y_k)^2$$

which can be minimized by considering the $M \times M$ system generated from

$$\frac{\partial E_2}{\partial C_j} = 0 \quad j = 1, 2, \cdots, M$$

In general, this gives the nonlinear set of equations

$$\sum_{k=1}^{n} (f(x_k, C_1, C_2, C_3, \cdots, C_M) - y_k) \frac{\partial f}{\partial C_j} = 0 \quad j = 1, 2, 3, \cdots, M \,.$$

Solving this set of equations can be quite difficult. **Most attempts at solving nonlinear systems are based upon iterative schemes which require good initial guesses to converge to the solution.**

Regardless, the general fitting procedure is straightforward and allows for the construction of a best fit curve to match the data. Section 2.6 gives an example of a specific algorithm used to solve such nonlinear systems. In such a solution procedure, it is imperative that a reasonable initial guess be provided for by the user. Otherwise, the desired rapid convergence or convergence to the desired root may not be achieved.

Polynomial Fits and Splines

One of the primary reasons for generating data fits from polynomials, splines or least-square methods **is to interpolate (插补) or extrapolate (外推) data values**. In practice, when considering only a finite number of data points

$$\begin{matrix} (x_0, y_0) \\ (x_1, y_1) \\ \vdots \\ (x_n, y_n) \end{matrix}$$

the value of the curve at points other than the x_i are unknown. Interpolation uses the data points to predict values of $y(x)$ at locations where $x \neq x_i$ and $x \in [x_0, x_n]$. Extrapolation is similar, but it predicts values of $y(x)$ for $x > x_n$ or $x < x_0$, i.e. outside the range of the data points.

Interpolation and extrapolation are easy to do given a least-squares fit. Once the fitting curve is found, it can be evaluated for any value of *x*, thus giving an interpolated or extrapolated value.

Polynomial fitting is another method for getting these values. With polynomial fitting, a polynomial is chosen to go through all data points. For the *n* + *1* data points given above, an *nth* degree polynomial is chosen

$$p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

where the coefficients a_j are chosen so that the polynomial passes through each data point. Thus we have the resulting system.

$$\begin{aligned}
(x_0, y_0) : & y_0 = a_n x_0^n + a_{n-1} x_0^{n-1} + \cdots + a_1 x_0 + a_0 \\
(x_1, y_1) : & y_1 = a_n x_1^n + a_{n-1} x_1^{n-1} + \cdots + a_1 x_1 + a_0 \\
& \vdots \\
(x_n, y_n) : & y_n = a_n x_n^n + a_{n-1} x_n^{n-1} + \cdots + a_1 x_n + a_0
\end{aligned}$$

This system of equations is nothing more than a linear system $Ax = b$ which can be solved by using the *backslash* command in MATLAB. Note that **unlike the least-square fitting method, the error of this fit is identically zero since the polynomial goes through each individual data point.**

However, this does not necessarily mean that the resulting polynomial is a good fit to the data. This will be shown in what follows.

Although this polynomial fit method will generate a curve which passes through all the data, **it is an expensive computation** since we have to first set up the system and then perform an $O(n^3)$ operation to generate the coefficients a_j.

A more direct method for generating the relevant polynomial is the **Lagrange polynomials** method. Consider first the idea of constructing a line between the two points (x_0, y_0) and (x_1, y_1). The general form for a line is $y = mx + b$ which gives

$$y = y_0 + (y_1 - y_0)\frac{x - x_0}{x_1 - x_0}$$

Although valid, it is hard to continue to generalize this technique to fitting higher order polynomials through a larger number of points. **Lagrange developed a method which expresses the line through the above two points as**

$$p_1(x) = y_0 \tfrac{x - x_1}{x_0 - x_1} + y_1 \tfrac{x - x_0}{x_1 - x_0}$$

which can be easily verified to work.

In a more compact and general way, this first degree polynomial can be expressed as

$$p_1(x) = \sum_{k=0}^{1} y_k L_{1,k}(x) = y_0 L_{1,0}(x) + y_1 L_{1,1}(x)$$

where the Lagrange coefficients are given by

$$L_{1,0}(x) = \frac{x - x_1}{x_0 - x_1}$$
$$L_{1,1}(x) = \frac{x - x_0}{x_1 - x_0}$$

Note the following properties of these coffecients:

$$L_{1,0}(x_0) = 1$$
$$L_{1,0}(x_1) = 0$$
$$L_{1,1}(x_0) = 0$$
$$L_{1,1}(x_0) = 1$$

By design, the Lagrange coefficients take on the binary values of zero or unity at the given data points. The power of this method is that it can be easily generalized to consider the *n + 1* points of our original data set. In particular, **we fit an nth degree polynomial through the given data set of the form**

$$p_n(x) = \sum_{k=0}^{n} y_k L_{n,k}(x)$$

where the **Lagrange coefficient** is

$$L_{n,k}(x) = \frac{(x-x_0)(x-x_1)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_n)}{(x_k-x_0)(x_k-x_1)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)}$$

so that

$$L_{n,k}(x_j) = \begin{cases} 1 & j = k \\ 0 & j \neq k. \end{cases}$$

Thus there is no need to solve a linear system to generate the desired polynomial. This is the preferred method for generating a polynomial fit to a given set of data and is the core algorithm employed by most commercial packages such as MATLAB for polynomial fitting.

Figure 3.2 shows the **polynomial fit** that in theory has zero error. In this example, 15 data points were used and a comparison is made between the polynomial fit, least-square line fit and a spline. The polynomial fit does an excellent job in the interior of the fitting regime, but generates a phenomenon known as polynomial wiggle (多项式摆动) at the edges (边界). **This makes a polynomial fit highly suspect for any application due to its poor ability to accurately capture interpolations or extrapolations**.

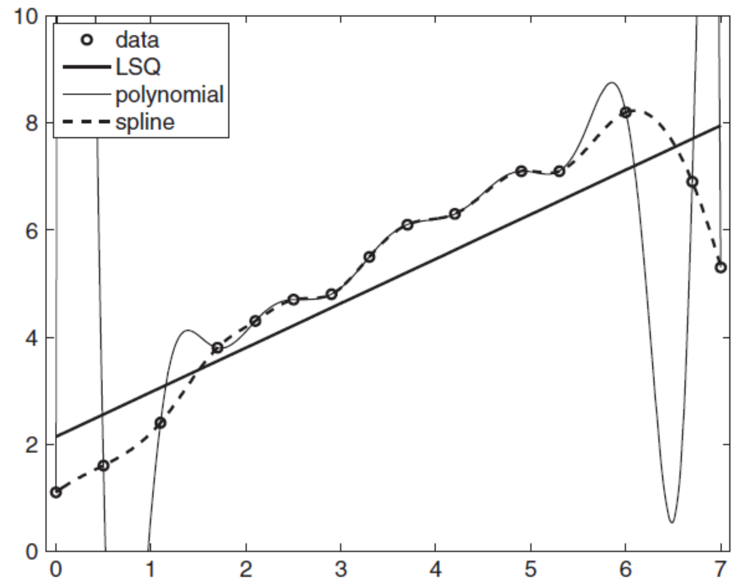


Figure 3.2: Fitting methods to a given data set (open circles) consisting of 15 points. Represented is a least-square (LSQ) line fit, a polynomial fit and a spline. The polynomial fit has the drawback of generating polynomial wiggle at its edges making it a poor candidate for interpolation or extrapolation.

Splines

Although MATLAB makes it a trivial matter to fit polynomials or least-square fits through data, a fundamental problem can arise as a result of a polynomial fit: polynomial wiggle (see Fig. 3.2). Polynomial wiggle is generated by the fact that an nth degree polynomial has, in general, $n - 1$ turning points from up to down or vice versa.

One way to overcome this is to use a **piecewise polynomial interpolation (分段多项式插值)** scheme. Essentially, this simply draws a line between neighboring data points and uses this line to give interpolated values. This technique is rather simple minded, but it does alleviate the problem generated by polynomial wiggle. However, the interpolating function is now only a piecewise funkction. Therefore, when considering interpolating values between the points (x_0, y_0) and (x_n, y_n), there will be n linear functions each valid only between two neighboring points.

The data generated by a piecewise linear fit can be rather crude and it tends to be choppy in appearance. Splines provide a better way to represent data by constructing cubic functions between points so that the first and second derivatives are continuous at the data points. This gives a smooth looking function without polynomial wiggle problems. The basic assumption of the spline method is to construct a cubic function between data points:

$$S_k(x) = S_{k,0} + S_{k,1}(x - x_k) + S_{k,2}(x - x_k)^2 + S_{k,3}(x - x_k)^3$$

where $x \in [x_k, x_{k+1}]$ and the coefficients $S_{k,j}$ are to be determined from various constraint conditions. Four constraint conditions are imposed:

$$\begin{aligned} S_k(x_k) &= y_k \\ S_k(x_{k+1}) &= S_{k+1}(x_{k+1}) \\ S'_k(x_{k+1}) &= S'_{k+1}(x_{k+1}) \\ S''_k(x_{k+1}) &= S''_{k+1}(x_{k+1}) \end{aligned}$$

This allows for a smooth fit to the data since the four constraints correspond to fitting the data, continuity of the function, continuity of the first derivative, and continuity of the second derivative, respectively.

To solve for these quantities, a large system of equations *Ax* = *b* is constructed. The number of equations and unknowns must first be calculated.

$$S_k(x_k) = y_k \rightarrow \text{Solutionfit}: n+1\,\text{equations};$$
$$S_k = S_{k+1} \rightarrow \text{Continity}; n-1\,\text{equations};$$
$$S'_k = S'_{k+1} \rightarrow \text{Smoothness}; n-1\,\text{equations};$$
$$S''_k = S''_{k+1} \rightarrow \text{Smoothness}: n-1\,\text{equations}$$

This gives a total of $4n - 2$ equations. For each of the n intervals, there are four parameters which gives a total of $4n$ unknowns. Thus two extra constraint conditions must be placed on the system to achieve a solution. There are a large variety of options for assumptions which can be made at the edges of the spline. It is usually a good idea to use the default unless the application involved requires a specific form. The spline problem is then reduced to a simple solution of an $Ax = b$ problem which can be solved with the *backslash* command.

As a final remark on splines, splines are heavily used in computer graphics and animation. The primary reason is for their relative ease in calculating, and for their smoothness properties. There is an entire spline toolbox available for MATLAB which attests (证实) to the importance of this technique for this application. Further, splines are also commonly used for smoothing data before differentiating.

Data Fitting with MATLAB

This section will discuss the practical implementation of the curve fitting schemes presented in the preceding two sections. The schemes to be explored are least-square fits, polynomial fits, line interpolation and spline interpolation. Additionally, a nonpolynomial least-square fit will be considered which results in a nonlinear system of equations. This nonlinear system requires additional insight into the problem and sophistication in its solution technique.

To begin the data fit process, we first import a relevant data set into the MATLAB environment. To do so, the *load* command is used. The file *linefit.dat* is a collection of *x* and *y* data values put into a two-column format separated by spaces. The file is the following:

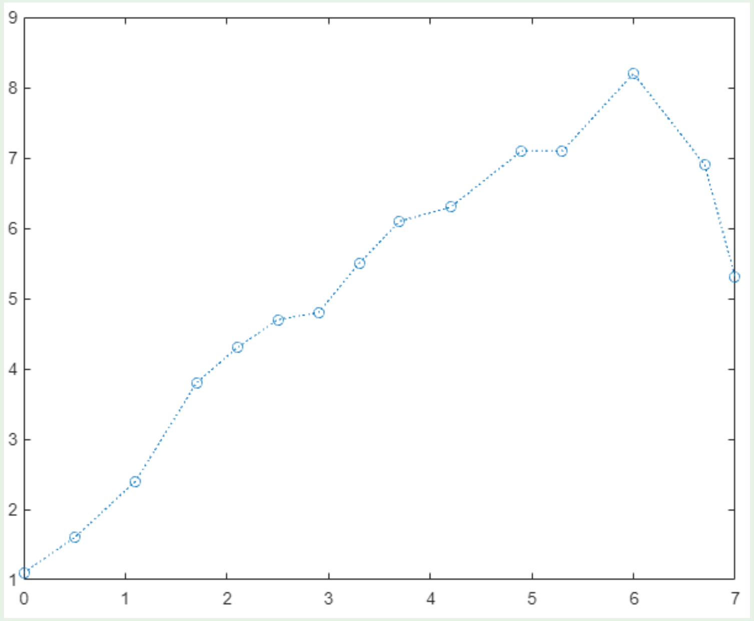
linefit.dat

```
0.0    1.1
0.5    1.6
1.1    2.4

1.7    3.8
2.1    4.3
2.5    4.7

2.9    4.8
3.3    5.5
3.7    6.1

4.2    6.3
4.9    7.1
5.3    7.1
6.0    8.2
6.7    6.9
7.0    5.3
```

This data set is just an example of what you may want to import into MATLAB. The command structure to read this data is as follows

```
load linefit.dat
x = linefit(:, 1);
y = linefit(:, 2);
figure, plot(x, y, 'o:)
```

After reading in the data, the two vectors *x* and *y* are created from the first and second column of the data, respectively. It is this set of data which will be explored with line fitting techniques. The code will also generate a plot of the data in *figure 1* of MATLAB.



Least-squares fitting

The least-squares fit technique is considered first. The *polyfit* and *polyval* commands are essential to this method. Specifically, the *polyfit* command is used to generate the $n + 1$ coefficients a_j of the *nth* degree polynomial

$$p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

used for fitting the data. The basic structure requires that the vectors *x* and *y* be submitted to *polyval* along with the desired degree of polynomial fit *n*. To fit a line (n = 1) through the data, use the command

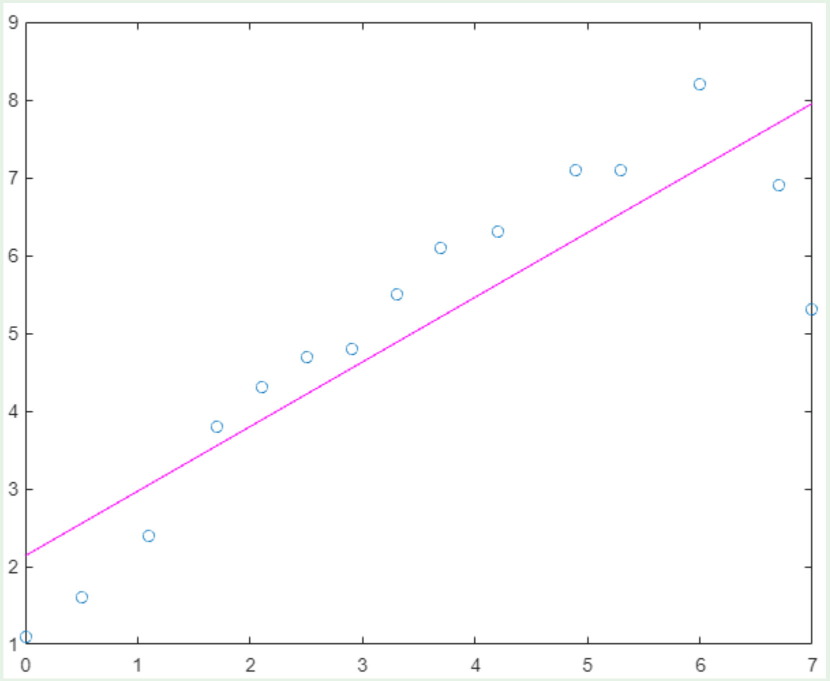
```
pcoeff = polyfit(x, y, 1);
```

The output of this function call is a vector *pcoeff* which includes the coefficients a_1 and a_0 of the line fit $p_1(x) = a_1 x + a_0$.

To evaluate and plot this line, values of x must be chosen. For this example, the line will be plotted for *x* ∈ *[0, 7]* in steps of $\Delta x = 0.1$.

```
xp = 0:0.1:7;
yp = polyval(pcoeff, xp);
figure(2), plot(x, y, 'O', xp, yp, 'm')
```

The *polyval* command uses the coefficients generated from *polyfit* to generate the *y*-values of the polynomial fit at the desired values of *x* given by *xp*. Figure 2 in MATLAB depicts both the data and the best line fit in the least-square sense. Figure 3.2 of the last section demonstrates a least-squares line fit through the data.



To fit a parabolic profile (抛物线剖面) through the data, a second degree polynomial is used. This is generated with

```
pcoeff2 = polyfit(x, y, 2)
yp2 = polyval(pcoeff2, xp);
figure(3), plot(x, y, 'O', xp, yp2, 'm')
```

Here the vector *yp2* contains the parabolic fit to the data evaluated at the *x*-values *xp*. These results are plotted in MATLAB figure 3. Figure 3.3 shows the least-square parabolic fit to the data. This can be contrasted to the linear least-square fit in Fig. 3.2. To find the least-square error, the sum of the squares of the differences between the parabolic fit and the actual data must be evaluated. Specifically, the quantity

$$E_2(f) = \left(\frac{1}{n}\sum_{k=1}^{n} |f(x_k) - y_k|^2\right)^{1/2}$$

is calculated.

For the parabolic fit considered in the last example, the polynomial fit must be evaluated at the *x*-values for the given data linefit.dat. The error is then calculated

```
yp3 = polyval(pcoeff2 ,x)
E2 = sqrt(sum((abs(yp3-y)).^2)/n)
```

This is a quick and easy calculation which allows for an evaluation of the fitting procedure. In general, the error will continue to drop as the degree of the polynomial is increased. This is because every extra degree of freedom allows for a better least-squares fit to the data. Indeed, a degree $n - 1$ polynomial has zero error since it goes through all the data points.

Interpolation

In addition to least-square fitting, interpolation techniques can be developed which go through all the given data points. The error in this case is zero, but each interpolation scheme must be evaluated for its accurate representation of the data. The first interpolation scheme is a polynomial fit to the data. Given *n + 1* points, an nth degree polynomial is chosen. The *polyfit* command is again used for the calculation

```
n=length(x)-1;
pcoeffn=polyfit (x,y,n);
ypn=polyval(pcoeffn ,xp);
figure (4),plot (x,y,'O',xp,ypn,'m')
```

The MATLAB script will produce an nth degree polynomial through the data points. But as always there is the danger with polynomial interpolation: polynomial wiggle can dominate the behavior. The strong oscillatory phenomenon at the edges is a common feature of this type of interpolation. Figure 3.2 has already illustrated the pernicious behavior of the polynomial wiggle phenomenon. Indeed, the idea of using a high-degree polynomial for a data fit becomes highly suspect upon seeing this example.

In contrast to a polynomial fit, a piecewise linear fit (逐段线性拟合) gives a simple minded connect-the-dot interpolation to the data. The *interp1* command gives the piecewise linear fit algorithm

```
yint=interp1(x,y,xp);
figure(5),plot (x,y,'O',xp,yint ,'m')
```

The linear interpolation is illustrated in *figure 5* of MATLAB. There are a few options available with the *interp1* command, including the nearest option and the *spline* option. The nearest option gives the nearest value of the data to the interpolated value while the *spline* option gives a cubic spline fit interpolation to the data.

The two options can be compared with the default linear option.

```
yint=interp1(x,y,xp);
yint2=interp1(x,y,xp,'nearest')
yint3=interp1(x,y,xp,'spline)
figure(5),plot (x,y,'O',xp,yint,'m',xp,int2,'k',xp,int3,'
    r')
```

Note that the spline option is equivalent to using the spline algorithm supplied by MATLAB. Thus a smooth fit can be achieved with either spline or interp1. Figure 3.3 demonstrates the nearest neighbor and linear interpolation fit schemes. The most common (default) scheme is the simple linear interpolation between data points.

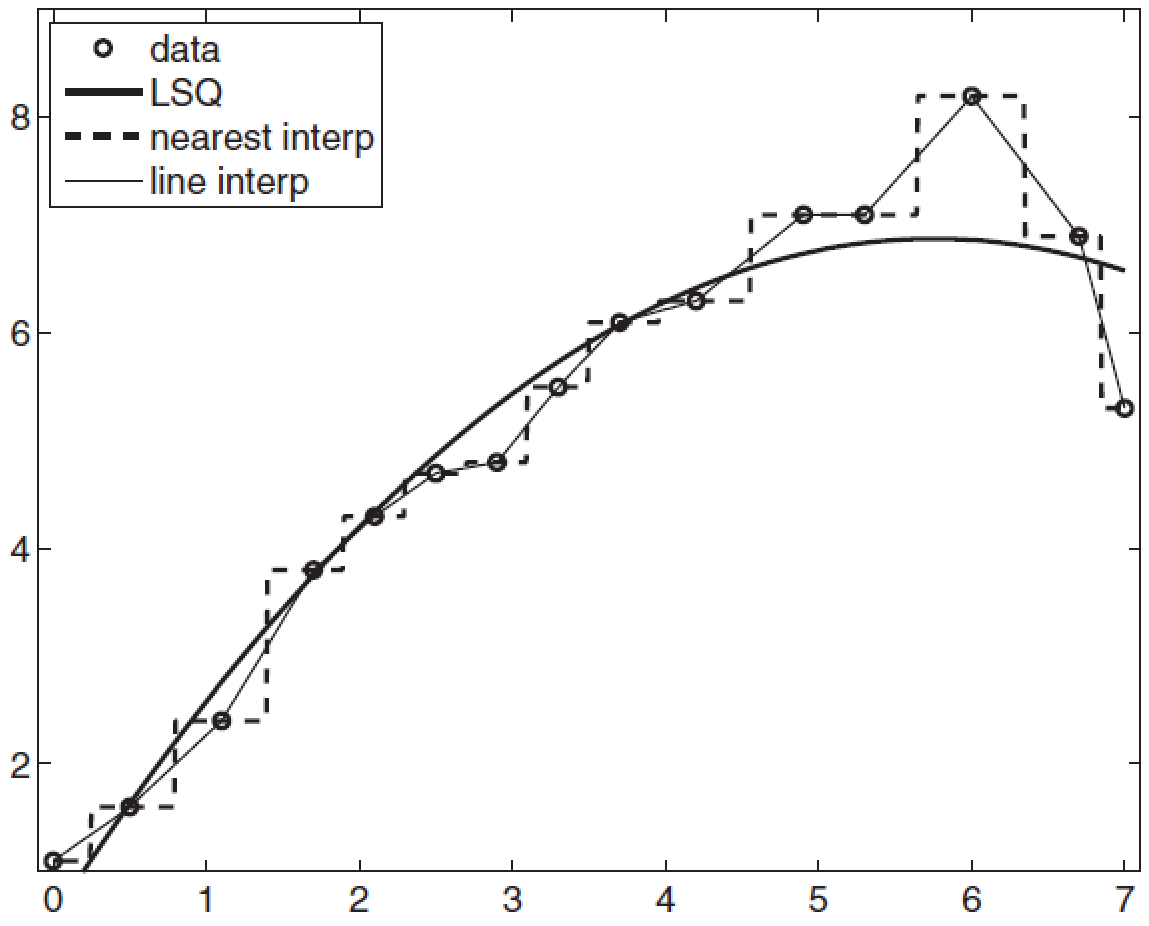


Figure 3.3: Fitting methods to a given data set (open circles) consisting of 15 points. Represented is a least-square (LSQ) quadratic fit, a nearest neighbor interpolation fit and a linear interpolation between data points.

The *spline* command is used by giving the *x* and *y* data along with a vector *xp* for which we desire to generate corresponding *y*-values.

```
yint=interp1(x,y,xp);
figure(5),plot (x,y,'O',xp,yint,'m')
```

The generated spline is depicted in *figure 6*. This is the same as that using *interp1* with the spline option. Note that the data is smooth as expected from the enforcement of continuous smooth derivatives.

Nonpolynomial least-square fitting

To consider more sophisticated least-square fitting routines, consider the following data set which looks like it could be nicely fitted with a Gaussian profile.

gaussfit.dat

```
−3.0  −0.2
−2.2  0.1
−1.7  0.05
−1.5  0.2
−1.3  0.4
−1.0  1.0
−0.7  1.2
−0.4  1.4
−0.25  1.8
−0.05  2.2
0.07  2.1
0.15  1.6
0.3  1.5
0.65  1.1
1.1  0.8
1.25  0.3
1.8  −0.1
2.5  0.2
```

To fit the data to a Gaussian, we indeed assume a Gaussian function of the form

$$f(x) = A\exp(-Bx^2)$$

Following the procedure to minimize the least-square error leads to a set of nonlinear equations for the coefficients A and B. In general, solving a nonlinear set of equations can be a difficult task. A solution is not guaranteed to exist. And in fact, there may be many solutions to the problem. Thus the nonlinear problem should be handled with care.

To generate a solution, the least-square error must be minimized. In particular, the sum

$$E_2 = \sum_{k=0}^{n} |f(x_k) - y_k|^2$$

must be minimized. The command *fminsearch* in MATLAB minimizes a function of several variables.

In this case, we minimize (3.3.4) with respect to the variables *A* and *B*. Thus the minimum of

$$E_2 = \sum_{k=0}^{n} |A\exp(-Bx_k^2) - y_k|^2$$

must be found with respect to *A* and *B*. The *fminsearch* algorithm requires a function call to the quantity to be minimized along with an initial guess for the parameters which are being used for minimization, i.e. *A* and *B*.

```
coeff=fminsearch ('gauss_fit',[1 1]);
```

This command structure uses as initial guesses *A* = *1* and *B* = *1* when calling the file *gauss_fit.m*. After minimizing, it returns the vector *coeff* which contains the appropriate values of *A* and *B* which minimize the least-square error.

The function called is then constructed as follows:

gauss_fit.m

```
function E=gauss_fit(x0)
load gaussfit.dat
x=gaussfit(:,1);
y=gaussfit(:,2);
E=sum((x0(1)* exp(-x0(2)*x.^2)-y).^2)
```

The vector x_0 accepts the initial guesses for A and B and is updated as A and B are modified through an iteration procedure to converge to the least-square values. Note that the sum is simply a statement of the quantity to be minimized, namely (3.3.5).

The results of this calculation can be illustrated with MATLAB figure 7:

```
xga=-3:0.1:3;
a=coeff(1);b=coeff(2)
yga=a*exp(-b*xga.^2);
figure(7),plot (x2,y2,'O',xga,yga,'m')
```

Note that for this case, the initial guess is extremely important. For any given problem where this technique is used, an educated guess for the values of parameters like *A* and *B* can determine if the technique will work at all. The results should also be checked carefully since there is no guarantee that a minimization near the desired fit can be found. Figure 3.4 shows the results of the least-square fit for the Gaussian example considered here. In this case, MATLAB determines that *A = 1.8733* and *B = 0.9344* minimizes the least-square error.

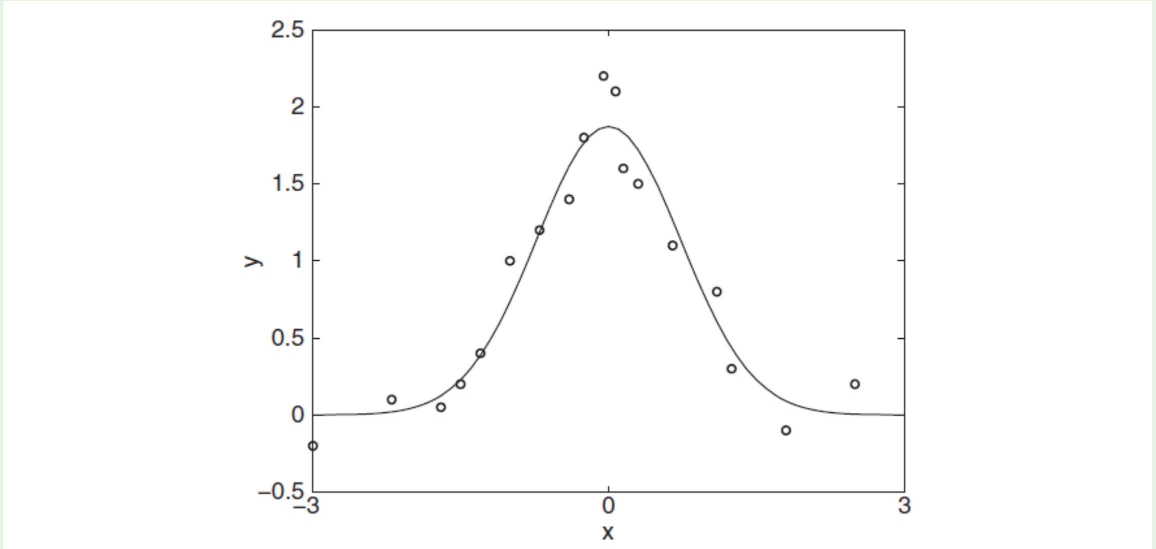


Figure 3.4: Gaussian fit $f(x) = A\exp(-Bx^2)$ to a given data set (open circles). The **fminsearch** command performs an iterative search to minimize the associated, nonlinear least-squares fit problem. It determines the values to be $A = 1.8733$ and $B = 0.9344$.