Chapter 12

Curve Fitting and Interpolation

12.1 Expanding Universe

In 1929, Edwin Hubble analyzed the observational data for a group of 24 nebulae (large clouds of gas and dust) beyond the Milky Way galaxy.[1] For these few nebulae the data were thought to be reliable enough to provide an estimate of the distance to the nebula, d, as well as the radial velocity of the nebula, v. The data are given in Table 12.1, and plotted in Fig. 12.1. Although the data are very scattered, they show a rough trend: galaxies that are farther away from us are receding from us at a faster rate.

d (Mpc)	v (km/s)	d (Mpc)	v (km/s)	d (Mpc)	v (km/s)
0.032	170	0.5	270	1.1	450
0.034	290	0.63	200	1.1	500
0.214	−130	0.8	300	1.4	500
0.263	−70	0.9	−30	1.7	960
0.275	−185	0.9	650	2.0	500
0.275	−220	0.9	150	2.0	850
0.45	200	0.9	500	2.0	800
0.5	290	1.0	920	2.0	1090

Table 12.1: Data from Hubble's 1929 study of extra–galactic nebulae. Distances d are in mega–parsecs (Mpc) and velocities v are in kilometers per second (km/s). One Mpc is equal to 3.26×10^6 light–years.

Hubble's law is the assertion that, on average, the recession velocity v is proportional to the distance d. That is, the data shown in Fig. 12.1

[1]Hubble, E. (1929), A relation between distance and radial velocity among extra-galactic nebulae, *Proc. Natl. Acad. Sci.* U.S.A. **15**, 3, pp. 168–173.

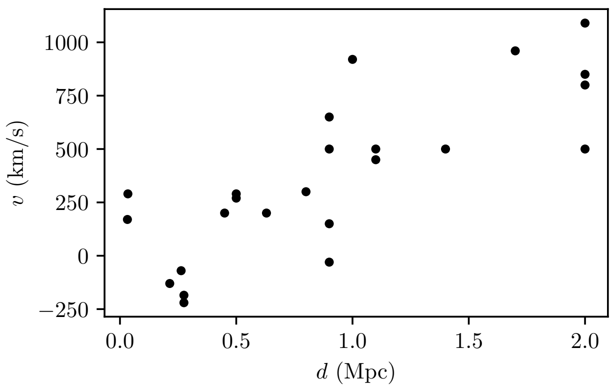

Fig. 12.1: The data from Table 12.1 suggest that the recession velocity between galaxies is greater for galaxies that are farther apart.

represent a straight line[2]

$$v = H_0 d \; . \tag{12.1}$$

The slope H_0 is called the *Hubble constant*. We need a rational way to find the slope from the scattered data.

Currently, the best measurements of the Hubble constant yield values around $H_0 = 70\,(\text{km/s})/\text{Mpc}$. In other words, galaxies that are 1 Mpc apart are (on average) receeding away from each other at 70 km/s. Galaxies that are 2 Mpc apart are receeding away from each other at about 140 km/s.

Exercise 12.1

The data from Table 12.1 is contained in the file *HubbleData.txt*. See the Appendix. Write a code that will read this data and reproduce the graph in Fig. 12.1.

[2]The data are affected by the motion of our solar system. Hubble applied a correction to the raw data to account for this motion. The corrected data is still quite scattered, but slightly more suggestive of a straight–line relationship.

12.2 Least squares method

How do we find the slope H_0 from the scattered data of Fig. 12.1? One way to determine H_0 is the *least squares method*.

Let d_i and v_i denote the data values from Table 12.1, where the index i ranges from 0 through 23. Thus, $d_0 = 0.032$, $v_0 = 170$, *etc.* With the least squares method, we look for the value of H_0 that minimizes the sum of squares of differences between $H_0 d_i$ and v_i. This is depicted in Fig. 12.2. To be precise, we define the sum

$$S = \sum_{i=0}^{23} (H_0 d_i - v_i)^2 \ , \tag{12.2}$$

and minimize S with respect to H_0.

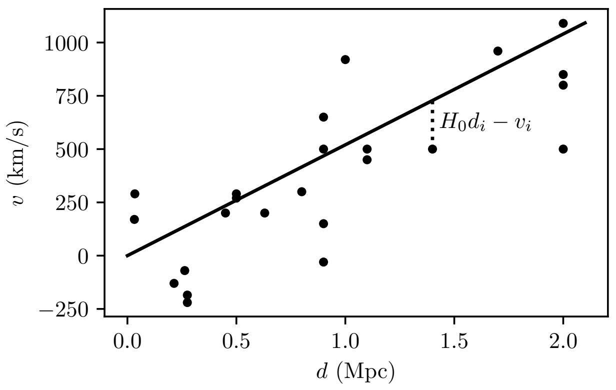

Fig. 12.2: $H_0 d_i - v_i$ is the "distance" between the ith data point and the line $v = H_0 d$. The least squares method determines the value of H_0 that minimizes the sum of the squares of these distances.

Let's adopt a simpler notation by dropping the limits on the sum and writing $S = \sum (H_0 d_i - v_i)^2$. Note that S is a quadratic function of H_0. The minimum of S is found by setting the derivative to zero:

$$\frac{dS}{dH_0} = \sum 2H_0 (d_i)^2 - \sum 2 v_i d_i = 0 \ . \tag{12.3}$$

Since H_0 is a constant, independent of i, we can bring this factor outside the summation sign and solve; the result is

$$H_0 = \frac{\sum v_i d_i}{\sum (d_i)^2} \ . \tag{12.4}$$

With this value of H_0, the line $v = H_0 d$ is a "best fit" for the data in Table 12.1.

Exercise 12.2

Extend your code from Exercise 12.1 to determine the Hubble constant H_0 using the least squares method. Plot the best fit line $v = H_0 d$ and the data points on the same graph.

Hubble's result for H_0 is rather far from the modern–day value of around 70 (km/s)/Mpc. In 1929 the observational errors were quite large. Nevertheless, Hubble's analysis is historically important because it represents the first observational evidence that the Universe is expanding.

12.3 Best fit line

Let x_i, y_i denote a set of data with $i = 0, \ldots, N-1$. We can apply the least squares method to find a function $f(x)$ that fits this data. The first step is to choose a form for $f(x)$. There are many situations in which the expected relationship between the variables x and y is linear. So let's assume a linear form for the unknown function, $f(x) = ax + b$. Hubble's law (12.1) is the special case in which the y–intercept b vanishes.

The sum of squares of differences between the function values $f(x_i) = ax_i + b$ and the data values y_i is

$$S = \sum_{i=0}^{N-1} (ax_i + b - y_i)^2 \ . \tag{12.5}$$

S is a quadratic function of the coefficients a and b, and its minimum satisfies $\partial S/\partial a = 0$ and $\partial S/\partial b = 0$. Explicitly, we have

$$\frac{\partial S}{\partial a} = 2 \sum (ax_i + b - y_i) x_i = 0 \ , \tag{12.6a}$$

$$\frac{\partial S}{\partial b} = 2 \sum (ax_i + b - y_i) = 0 \ . \tag{12.6b}$$

Limits on the summation signs have been dropped for notational simplicity. Since a and b are constants, independent of the summation index i, these results simplify to

$$a\sum x_i^2 + b\sum x_i = \sum x_i y_i \tag{12.7a}$$

$$a\sum x_i + bN = \sum y_i \ , \tag{12.7b}$$

where $\sum b = Nb$. This is a system of linear equations for the two unknowns, the slope a and the y–intercept b of the line that best fits the data.

Exercise 12.3a

Solve the two equations (12.7) for a and b in terms of $\sum x_i^2$, $\sum x_i$, $\sum x_i y_i$, and $\sum y_i$. (Hint: Invent a simplified notation for the sums, such as Sxx for $\sum x_i^2$ and Sxy for $\sum x_i y_i$. Then use SymPy to solve for a and b.)

Exercise 12.3b

Import the data file *LineData.txt* (see the Appendix). Apply the least squares method to fit the data to a straight line, using the formulas for the slope a and y–intercept b from the previous exercise. Make a plot that shows the data points and the best–fit line.

12.4 Best fit curve

Least squares analysis can be applied to any class of functions $f(x)$. For example, consider the data from the graph of Fig. 12.3. The data appear to follow a sin–curve relationship, but we don't know the amplitude, frequency or phase. We can carry out a least–squares analysis to fit this data to a function of the form

$$f(x) = a\sin(bx + c) \ . \tag{12.8}$$

The goal is to find the parameter values a, b and c that minimize

$$S = \sum_{i=1}^{N} (a\sin(bx_i + c) - y_i)^2 \ , \tag{12.9}$$

by solving the equations $\partial S/\partial a = 0$, $\partial S/\partial b = 0$ and $\partial S/\partial c = 0$. These equations are nonlinear and cannot be solved analytically; they must be solved numerically. Fortunately, the function `curve_fit()` in the `scipy.optimize` library will do the work for us. Begin by importing `scipy.optimize` as `so` and creating the function definition

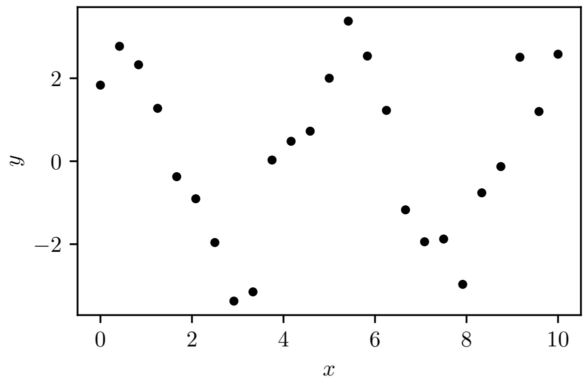

Fig. 12.3: The data suggest a sin–curve relationship, but the amplitude, frequency and phase are unknown.

```
def func(x,a,b,c):
    return a*np.sin(b*x + c)
```

The first argument is the independent variable (in this case x), and the remaining arguments are the parameters (in this case a, b and c). Let's assume the data are contained in arrays called `xdata` and `ydata`. Then the command

```
params, cov = so.curve_fit(func,xdata,ydata)
```

will return the least–squares values of a, b and c in the parameter array `params`. The `curve_fit()` function also returns an array `cov` containing the *covariance matrix*. The square root of each diagonal element of `cov` provides an estimate of the standard error in the corresponding parameter value. For example, the parameter a is contained in `params[0]` and its standard error is `sqrt(cov[0,0])`. The standard errors tell us how well the parameter values are constrained by the data, and give us a measure of how well the function fits the data.

Exercise 12.4a

Write a code that will import the data file *SineData.txt* (see the Appendix) and carry out a least–squares fit to a function of the form (12.8). Plot a graph showing the data points, as well as the best fit curve. What values did you get for the amplitude, frequency and phase?

Exercise 12.4b

Write a code to import the data file *PolyData.txt* (see the Appendix). Carry out a least–squares fit to each of the three polynomials

$$f(x) = cx + d$$
$$f(x) = bx^2 + cx + d$$
$$f(x) = ax^3 + bx^2 + cx + d$$

Have your code plot the data and each of the three best–fit curves on the same graph.

Exercise 12.4c

Use `curve_fit()` to find the best–fit line for the data from *LineData.txt*. Do your results match those from Exercise 12.3b? What is the standard error in each of the parameter values?

12.5 Data interpolation

Consider the data displayed in Table 12.2 and plotted in Fig. 12.4. What is the value of y at $x = 3.6$? To answer this question, we need to *interpolate* the data.

x	0.4	1.3	3.1	4.1	4.9	5.8	6.9
y	1.7	2.9	2.8	2.0	1.1	0.7	1.3

Table 12.2: Discrete data obtained from a set of measurments, observations, or simulations.

You might wonder how the data were obtained. Can't we simply measure or observe or compute the value of y at $x = 3.6$? It might not be practical to do so. For example, the data might come from observations of a comet passing near the sun and can't be repeated for years. Or the data

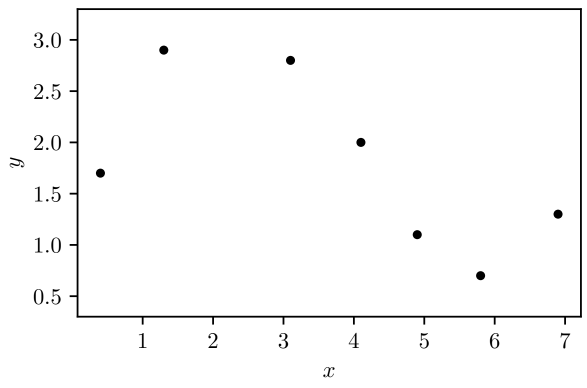

Fig. 12.4: Graph of the data listed in Table 12.2.

might come from a complicated numerical calculation. Each data point in Table 12.2 could take weeks or months to compute. We want to know $y(3.6)$ now.

A familiar example of interpolation is seen on a daily weather map showing isotherms, curves of constant temperature. (Or isobars, curves of constant pressure.) The data for these maps are collected at discrete times from weather stations at discrete locations, spread around the world. The data must be interpolated, first in time then in space, to obtain the isotherms.

Interpolation is appropriate when the known data are relatively accurate, but further data values are difficult to obtain. In the context of interpolation, the data points are often called "knots."

Return to the data of Table 12.2. We can use *linear interpolation* to find an approximate value for $y(3.6)$. With linear interpolation we simply connect the knots with straight lines. More precisely, for every adjacent pair of knots, we find a linear function that passes through the knots. For adjacent knots (x_0, y_0) and (x_1, y_1), the interpolating function is

$$y(x) = \frac{(y_1 - y_0)}{(x_1 - x_0)}(x - x_0) + y_0 \ . \qquad (12.10)$$

For the data from Table 12.2, the interpolating function between knots

$(3.1, 2.8)$ and $(4.1, 2.0)$ is

$$y(x) = \frac{(2.0 - 2.8)}{(4.1 - 3.1)}(x - 3.1) + 2.8 = 5.28 - 0.8x \tag{12.11}$$

We can now answer our question: $y(3.6) = 5.28 - (0.8)(3.6) = 2.4$.

Exercise 12.5

The data from Table 12.2 are contained in the file *InterpData.txt*. Write a code to define the function $y(x)$ that linearly interpolates this data. Plot the function along with the data points. Use your function to approximate $y(2.0)$ and $y(6.5)$. (Note: The function $y(x)$ is piecewise continuous, defined by Eq. (12.10) in the intervals between each adjacent pair of knots.)

12.6 Cubic splines

Linear interpolation is not always desirable because the resulting function is not smooth. In particular, it is generally not differentiable at the location of each knot. With linear interpolation we cannot answer questions such as: What is the (approximate) value of the derivative dy/dx at $x = 5.8$?

A better option is to span the gaps between knots using higher order polynomials. We could use quadratic polynomials, but it turns out that cubic polynomials are usually preferred. There is enough freedom in a cubic polynomial to insure that first and second derivatives are continuous at the knots. The interpolating polynomials are called *cubic splines*.

Cubic splines are constructed as follows. For simplicity, let's consider just 3 data points (x_0, y_0), (x_1, y_1) and (x_2, y_2). There are two gaps between these three data points, as shown in Fig. 12.5. We seek a cubic polynomial

$$p_1(x) = a_1 + b_1 x + c_1 x^2 + d_1 x^3 \tag{12.12}$$

to span the first gap, and a cubic polynomial

$$p_2(x) = a_2 + b_2 x + c_2 x^2 + d_2 x^3 \tag{12.13}$$

to span the second gap. We want these polynomials to meet smoothly at the middle knot (x_1, y_1).

To be precise, we need eight conditions on the polynomials (12.12) and (12.13) to determine the eight unknown coefficients a_1, b_1, c_1, d_1, a_2, b_2, c_2, and d_2. To begin, we demand that the polynomials match the data at

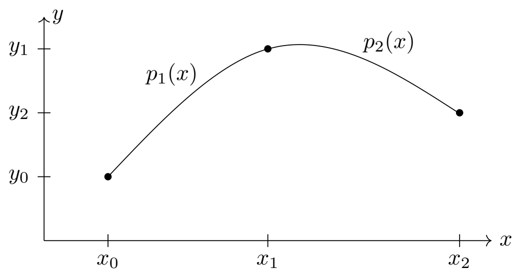

Fig. 12.5: Two cubic spline polynomials span the gaps between three knots.

their endpoints. This provides four equations: $p_1(x_0) = y_0$, $p_1(x_1) = y_1$, $p_2(x_1) = y_1$ and $p_2(x_2) = y_2$. Explicitly, these conditions yield

$$a_1 + b_1x_0 + c_1x_0^2 + d_1x_0^3 = y_0 \ , \tag{12.14a}$$
$$a_1 + b_1x_1 + c_1x_1^2 + d_1x_1^3 = y_1 \ , \tag{12.14b}$$
$$a_2 + b_2x_1 + c_2x_1^2 + d_2x_1^3 = y_1 \ , \tag{12.14c}$$
$$a_2 + b_2x_2 + c_2x_2^2 + d_2x_2^3 = y_2 \ . \tag{12.14d}$$

We also demand that the first and second derivatives of the two polynomials should match at the middle knot, (x_1, y_1). This gives $p_1'(x_1) = p_2'(x_1)$ and $p_1''(x_1) = p_2''(x_1)$, which are explicitly

$$b_1 + 2c_1x_1 + 3d_1x_1^2 = b_2 + 2c_2x_1 + 3d_2x_1^2 \ , \tag{12.14e}$$
$$2c_1 + 6d_1x_1 = 2c_2 + 6d_2x_1 \ . \tag{12.14f}$$

We need two more equations to determine the eight coefficients. The usual choice is to set the second derivatives at the endpoints to zero. That is, $p_1''(x_0) = 0$ and $p_2''(x_2) = 0$, which give

$$2c_1 + 6d_1x_0 = 0 \ , \tag{12.14g}$$
$$2c_2 + 6d_2x_2 = 0 \ . \tag{12.14h}$$

The eight equations (12.14) can be solved for the eight coefficients.

Exercise 12.6

Solve the system of equations (12.14) with the values $(x_0, y_0) = (1.3, 2.9)$, $(x_1, y_1) = (3.1, 2.8)$ and $(x_2, y_2) = (4.1, 2.0)$ from Table 12.2. (Hint: use `SymPy`.) Construct the polynomials $p_1(x)$ and $p_2(x)$

and graph the results. What is $y(3.6)$? What is the derivative dy/dx at $x = 3.1$?

12.7 More data points

The pattern for cubic splines will work with any number of knots. Let's check.

If there are $N + 1$ knots, then there are N gaps between knots. With a cubic polynomial in each gap, there are $4N$ coefficients. We need $4N$ equations to determine these coefficients. Now, each spline should match the data at their endpoints. This provides $2N$ equations. Note that there are $N-1$ *interior* knots where two splines meet. Setting the first derivative of adjacent splines equal to each other gives $N - 1$ equations. Setting the second derivatives of adjacent splines equal to one another gives another $N - 1$ equations. Finally, we set the second derivative of the first spline equal to zero at its left and, and the second derivative of the last spline equal to zero at its right end. This is a total of $4N$ equations, as desired.

With a bit of effort we could write a computer program to construct cubic splines for an arbitrary number of knots. Fortunately, this has been done for us, and implemented in the library `scipy.interpolate`. Given data arrays `x` and `y`, the command

```
fun = si.CubicSpline(x,y,bc_type='natural',extrapolate=False)
```

(with `scipy.interpolate` imported as `si`) produces a function that interpolates the data with cubic splines. This function, which we have named `fun`, is the *cubic spline interpolator* for the data set. You can evaluate `fun` like any other function. If `z` is a number, `fun(z)` gives the value of `fun` at `z`. If `z` is a NumPy array, `fun(z)` is the NumPy array obtained by evaluating `fun` at each element of `z`.

The option `bc_type='natural'` implements the "natural" boundary conditions discussed previously, namely, the vanishing of second derivatives at the first and last knots. The option `extrapolate=False` tells Python not to extrapolate the data beyond the first and last knots. If you try to evaluate `fun` at a value outside the domain, Python will return `nan` (which stands for "not a number").

Exercise 12.7a

Use `CubicSpline()` to compute the cubic spline approximation to the data given in Table 12.2 and *InterpData.txt*. Plot the cubic spline interpolator along with the data.

Because the cubic spline interpolator (which we call `fun`) is a piecewise polynomial, it is straightforward to differentiate and integrate. The first derivative is given by `fun.derivative()`. The `n`th derivative is obtained by replacing `()` with `(n)`. The indefinite integral of the interpolator is `fun.antiderivative()`. The constant of integration is chosen such that the integral equals zero at the left–end knot.

Exercise 12.7b

Create a data set by sampling the function

$$f(x) = \cos(x) + \sin(2x)$$

for x between $-\pi$ and π. That is, define the arrays

```
x = np.linspace(-np.pi,np.pi,N)
y = np.cos(x) + np.sin(2*x)
```

for some chosen value of `N`. Use `CubicSpline()` to obtain the cubic spline interpolator and compare to the original function $f(x)$ by plotting the difference. Find the derivative of the interpolator and compare to the derivative of $f(x)$ by plotting the diffference. Find the integral of the interpolator and compare to the integral of $f(x)$ by plotting the difference. (If needed, adjust the constants of integration.) How do your results depend on `N`?

12.8 Bilinear interpolation

Let $f(x, y)$ denote a function of independent variables x and y whose value is known only at discrete points in the x–y plane. We can use *bilinear interpolation* to estimate the values of f at other points in the plane.

Bilinear interpolation is just linear interpolation applied twice, once for each dimension. For simplicity let's consider a single "tile" in the x–y plane, bounded by $x_0 \le x \le x_1$ and $y_0 \le y \le y_1$. The function f is known only at the corners, (x_0, y_0), (x_0, y_1), (x_1, y_0), (x_1, y_1). Our interpolating function, which we will call $F(x, y)$, should match $f(x, y)$ at the corners of the tile.

First, interpolate in the x–direction, from x_0 to x_1, along the line $y = y_0$:

$$F(x, y_0) = \frac{f(x_1, y_0) - f(x_0, y_0)}{x_1 - x_0}(x - x_0) + f(x_0, y_0) \ . \qquad (12.15)$$

This is just the linear interpolation formula (12.10) with some changes of notation. Next, we perform a linear interpolation along the line $y = y_1$:

$$F(x, y_1) = \frac{f(x_1, y_1) - f(x_0, y_1)}{x_1 - x_0}(x - x_0) + f(x_0, y_1) \ . \qquad (12.16)$$

Now we have estimates for the function values along the top and bottom edges of the tile. The final step is to interpolate in the y–direction, from y_0 to y_1, along the line $x = \text{const}$:

$$F(x, y) = \frac{F(x, y_1) - F(x, y_0)}{y_1 - y_0}(y - y_0) + F(x, y_0) \ . \qquad (12.17)$$

Combining these results, we find

$$\begin{aligned} F(x, y) = {} & \frac{(x - x_1)(y - y_1)}{(x_1 - x_0)(y_1 - y_0)} f(x_0, y_0) - \frac{(x - x_0)(y - y_1)}{(x_1 - x_0)(y_1 - y_0)} f(x_1, y_0) \\ & - \frac{(x - x_1)(y - y_0)}{(x_1 - x_0)(y_1 - y_0)} f(x_0, y_1) + \frac{(x - x_0)(y - y_0)}{(x_1 - x_0)(y_1 - y_0)} f(x_1, y_1) \ . \end{aligned} \qquad (12.18)$$

You can verify by inspection that F matches f at the corners of the tile. We would obtain the same result for $F(x, y)$ by interpolating first in the y–direction, then in the x–direction.

Exercise 12.8a

Create a function definition `F(x,y)` for the bilinear interpolating function of Eq. (12.18). Set $x_0 = y_0 = 0$ and $x_1 = y_1 = 1$, and use the data $f(0, 0) = -3.0$, $f(0, 1) = 2.0$, $f(1, 0) = 1.0$ and $f(1, 1) = 3.0$. Compare `F(0.5,0.5)` to the average of the corner values. Plot `F(x,y)` along the two diagonals, from (x_0, y_0) to (x_1, y_1) and from (x_0, y_1) to (x_1, y_0).

Exercise 12.8b

Create a contour plot of the interpolating function from the previous exercise. One way to do this: Set

```
xarray = np.linspace(x0,x1,N)
yarray = np.linspace(y0,y1,N)
Farray = np.zeros((N,N))
```

for some chosen `N`, then fill the array elements `Farray[i,j]` with `F(xarray[i],yarray[j])`. The command

```
matplotlib.pyplot.contour(Farray,[0])
```

will create the plot, where `[0]` is a list of contour values. Add more contours to your plot.