CHAPTER 7

Interpolation

Interpolation is a mathematical method for constructing a function from a discrete set of data points. The interpolation function, or interpolant, should exactly coincide with the given data points, and it can also be evaluated for other intermediate input values within the sampled range. There are many applications of interpolation: A typical use-case that provides an intuitive picture is the plotting of a smooth curve through a given set of data points. Another use-case is to approximate complicated functions, which, for example, could be computationally demanding to evaluate. In that case, it can be beneficial to evaluate the original function only at a limited number of points, and use interpolation to approximate the function when evaluating it for intermediary points.

Interpolation may at a first glance look a lot like least square fitting, which we saw already in both Chapter 5 (linear least square) and Chapter 6 (nonlinear least square). Indeed, there are many similarities between interpolation and curve fitting with least square methods, but there are also important conceptual differences that distinguish these two methods: In least square fitting, we are interested in approximately fitting a function to data points in manner that minimize the sum of square errors, using many data points and an overdetermined system of equations. In interpolation, on the other hand, we require a function that exactly coincides with the given data points, and only use the number of data points that equals the number of free parameters in the interpolation function. Least square fitting is therefore more suitable for fitting a large number of data points to a model function, and interpolation is a mathematical tool for creating a functional representation for a given minimal number of data points. In fact, interpolation is an important component in many mathematical methods, including some of the methods for equation solving and optimization that we used in Chapters 5 and 6.

Extrapolation is a concept that is related to interpolation. It refers to evaluating the estimated function outside of the sampled range, while interpolation only refers to evaluating the function within the range that is spanned by the given data points. Extrapolation can often be riskier than interpolation, because it involves estimating a function in a region where it has not been sampled. Here we are only concerned with interpolation. To perform interpolation in Python we use the `polynomial` module from NumPy and the `interpolate` module from SciPy.

Importing Modules

Here we will continue with the convention of importing submodules from the SciPy library explicitly. In this chapter we need the `interpolate` module from SciPy, and also the `polynomial` module from NumPy, which provides functions and classes for polynomials. We import these modules as follows:

```
In [1]: from scipy import interpolate
In [2]: from numpy import polynomial as P
```

R. Johansson, *Numerical Python*, DOI 10.1007/978-1-4842-0553-2_7

In addition, we also need the rest of the NumPy library, the linear algebra module `linalg` from SciPy, and the Matplotlib library for plotting:

```
In [3]: import numpy as np
In [4]: from scipy import linalg
In [5]: import matplotlib.pyplot as plt
```

Interpolation

Before we dive into the details of how to perform interpolation with NumPy and SciPy, we first state the interpolation problem in mathematical form. For notational brevity, here we only consider one-dimensional interpolation, which can be formulated as follows: For a given set of n data point $\{(x_i, y_i)\}_{i=1}^{n}$, find a function $f(x)$ such that $f(x_i) = y_i$, for $i \in [1,n]$. The function $f(x)$ is known as the interpolant, and it is not unique. In fact, there are an infinite number of functions that satisfy the interpolation criteria. Typically we can write the interpolant as a linear combination of some basis functions $\phi_j(x)$, such that $f(x) = \sum_{j=1}^{n} c_j \phi_j(x)$, where c_j are unknown coefficients. Substituting the given data points into this linear combination results in a linear equation system for the unknown coefficients: $\sum_{j=1}^{n} c_j \phi_j(x_i) = y_i$. This equation system can be written in explicit matrix form as

$$\begin{bmatrix} \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_n(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(x_n) & \phi_2(x_n) & \cdots & \phi_n(x_n) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix},$$

or in a more compact implicit matrix form as $\Phi(x)c = y$, where the elements of the matrix $\Phi(x)$ are $\{\Phi(x)\}_{ij} = \phi_j(x_i)$. Note that here the number of basis functions is the same as the number of data points, and $\Phi(x)$ is therefore a square matrix. Assuming that this matrix has full rank, we can solve for the unique c-vector using the standard methods discussed in Chapter 5. If the number of data points is larger than the number of basis functions, then the system is overdetermined, and in general there is no solution that satisfies the interpolation criteria. Instead, in this situation it is more suitable to consider a least square fit than an exact interpolation; see Chapter 5.

The choice of basis functions affects the properties of the resulting equation system and a suitable choice of basis depends on the properties of the data that is fitted. Common choices of basis functions for interpolation are various types of polynomials, for example, the power basis $\phi_i(x) = x^{i-1}$, or orthogonal polynomials such as Legendre polynomials $\phi_i(x) = P_{i-1}(x)$, Chebyshev polynomials $\phi_i(x) = T_{i-1}(x)$, or piecewise polynomials. Note that in general $f(x)$ is not unique, but for n data points there is a unique interpolating polynomial of order $n-1$, regardless of which polynomial basis we use. For power basis $\phi_i(x) = x^{i-1}$, the matrix $\Phi(x)$ is the Vandermonde matrix, which we already have seen applications of in least square fitting in Chapter 5. For other polynomial bases, $\Phi(x)$ are generalized Vandermonde matrices, which for each basis defines the matrix of the linear equation system that has to be solved in the interpolation problem. The structure of the $\Phi(x)$ matrix is different for different polynomial bases, and its condition number and the computational cost of solving the interpolation problem varies correspondingly. Polynomials thus play an important role in interpolation, and before we can start to solve interpolation problems we need a convenient way of working with polynomials in Python. This is the topic of the following section.

Polynomials

The NumPy library contains the submodule `polynomial` (here imported as `P`), which provides functions and classes for working with polynomials. In particular, it provides implementations of many standard orthogonal polynomials. These functions and classes are useful when working with interpolation, and we therefore review how to use this module before looking at polynomial interpolation.

■ **Note** There are two modules for polynomials in NumPy: `numpy.poly1d` and `numpy.polynomial`. There is a large overlap in functionality in these two modules, but they are not compatible with each other (specifically, the coordinate arrays have reversed order in the two representations). The `numpy.poly1d` module is older and has been superseded by `numpy.polynomial`, which is now recommended for new code. Here we only focus on `numpy.polynomial`, but it is worth being aware of `numpy.poly1d` as well.

The `np.polynomial` module contains a number of classes for representing polynomials in different polynomial bases. Standard polynomials, written in the usual power basis $\{x^i\}$ are represented with the `Polynomial` class. To create an instance of this class we can pass a coefficient array to its constructor. In the coefficient array, the ith element is the coefficient of x^i. For example, we can create a representation of the polynomial $1+2x+3x^2$ by passing the list `[1, 2, 3]` to the `Polynomial` class:

```
In [6]: p1 = P.Polynomial([1, 2, 3])
In [7]: p1
Out[7]: Polynomial([ 1.,  2.,  3.], [-1,  1], [-1,  1])
```

Alternatively, we can also initialize a polynomial by specifying its roots using the class method `P.Polynomial.fromroots`. The polynomial with roots at $x=-1$ and $x=1$, for example, can be created using:

```
In [8]: p2 = P.Polynomial.fromroots([-1, 1])
In [9]: p2
Out[9]: Polynomial([-1.,  0.,  1.], [-1.,  1.], [-1.,  1.])
```

Here, the result is the polynomial with the coefficient array `[-1, 0, 1]`, which corresponds to $-1+x^2$. The roots of a polynomial can be computed using the `roots` method. For example, the roots of the two previously created polynomials are:

```
In [10]: p1.roots()
Out[10]: array([-0.33333333-0.47140452j, -0.33333333+0.47140452j])
In [11]: p2.roots()
Out[11]: array([-1.,  1.])
```

As expected, the roots of the polynomial `p2` are $x=-1$ and $x=1$, as was requested when it was created using the `fromroots` class method.

In the examples above, the representation of a polynomial is in the form `Polynomial([-1., 0., 1.], [-1., 1.], [-1., 1.])`. The first of the lists in this representation is the coefficient array. The second and third lists are the `domain` and `window` attributes, which can be used to map the input domain to of a polynomial to another interval. Specifically, the input domain interval `[domain[0], domain[1]]` is mapped to the interval `[window[0], window[1]]` through a linear transformation (scaling and translation). The default values are `domain=[-1,1]` and `window=[-1,1]`, which corresponds to an identity transformation (no change). The `domain` and `window` arguments are particularly useful when working with polynomials that are orthogonal with respect to a scalar product that is defined on a specific interval. It is then desirable to map the domain of the input data onto this interval. This is important when interpolating with orthogonal polynomials, such as the Chebyshev or Hermite polynomials, because performing this transformation can vastly improve the condition number of the Vandermonde matrix for the interpolation problem.

The properties of a `Polynomial` instance can be directly accessed using the `coeff`, `domain`, and `window` attributes. For example, for the `p1` polynomial defined above we have:

```
In [12]: p1.coef
Out[12]: array([ 1.,  2.,  3.])
In [13]: p1.domain
Out[13]: array([-1,  1])
In [14]: p1.window
Out[14]: array([-1,  1])
```

A polynomial that is represented as a `Polynomial` instance can easily be evaluated with arbitrary values of x by calling the class instance as a function. The x variable can be specified as a scalar, a list, or an arbitrary NumPy array. For example, to evaluate the polynomial `p1` at the points $x = \{1.5, 2.5, 3.5\}$, we simply call the `p1` class instance with a list of x values as this argument:

```
In [15]: p1([1.5, 2.5, 3.5])
Out[15]: array([ 10.75,  24.75,  44.75])
```

Instances of `Polynomial` can be operated on using the standard arithmetic operators `+`, `-`, `*`, `/`, and so on. The `//` operator is used for polynomial division. To see how this works, consider the division of the polynomial $p_1(x) = (x-3)(x-2)(x-1)$ with the polynomial $p_2(x) = (x-2)$. The answer, which is obvious when written in factorized form, is $(x-3)(x-1)$. We can be compute and verify this using NumPy in the following manner: First create Polynomial instances for the `p1` and `p2`, and then use the `//` operator compute the polynomial division.

```
In [16]: p1 = P.Polynomial.fromroots([1, 2, 3])
In [17]: p1
Out[17]: Polynomial([ -6.,  11.,  -6.,   1.], [-1.,  1.], [-1.,  1.])
In [18]: p2 = P.Polynomial.fromroots([2])
In [19]: p2
Out[19]: Polynomial([-2.,  1.], [-1.,  1.], [-1.,  1.])
In [20]: p3 = p1 // p2
In [21]: p3
Out[21]: Polynomial([ 3., -4.,  1.], [-1.,  1.], [-1.,  1.])
```

The result is a new polynomial with coefficient array [3, -4, 1], and if we compute its roots we find that they are 1 and 3, so this polynomial is indeed $(x-3)(x-1)$:

```
In [22]: p3.roots()
Out[22]: array([ 1.,  3.])
```

In addition to the Polynomial class for polynomials in the standard power basis, the polynomial module also has classes for representing polynomials in Chebyshev, Legendre, Laguerre and Hermite bases, with the names Chebyshev, Legendre, Laguerre, Hermite (Physicists') and HermiteE (Probabilists'), respectively. For example, the Chebyshev polynomial with coefficient list [1, 2, 3], that is, the polynomial $1T_0(x)+2T_1(x)+3T_2(x)$, where $T_i(x)$ is the Chebyshev polynomial of order *i*, can be created using:

```
In [23]: c1 = P.Chebyshev([1, 2, 3])
In [24]: c1
Out[24]: Chebyshev([ 1.,  2.,  3.], [-1,  1], [-1,  1])
```

and its roots can be computed using the roots attribute:

```
In [25]: c1.roots()
Out[25]: array([-0.76759188,  0.43425855])
```

All the polynomial classes have the same methods, attributes, and operators as the Polynomial class discussed above, and they can all be used in the same manner. For example, to create the Chebyshev and Legendre representations of the polynomial with roots $x=-1$ and $x=1$, we can use the fromroots attribute, in a same way as we did previously with the Polynomial class:

```
In [26]: c1 = P.Chebyshev.fromroots([-1, 1])
In [27]: c1
Out[27]: Chebyshev([-0.5,  0. ,  0.5], [-1.,  1.], [-1.,  1.])
In [28]: l1 = P.Legendre.fromroots([-1, 1])
In [29]: l1
Out[29]: Legendre([-0.66666667,  0.        ,  0.66666667], [-1.,  1.], [-1.,  1.])
```

Note that the same polynomial, here with the roots at $x=-1$ and $x=1$ (which is a unique polynomial), have different coefficient arrays when represented in different bases, but when evaluated at specific values of *x*, the two gives the same results (as expected):

```
In [30]: c1([0.5, 1.5, 2.5])
Out[30]: array([-0.75,  1.25,  5.25])
In [31]: l1([0.5, 1.5, 2.5])
Out[31]: array([-0.75,  1.25,  5.25])
```

Polynomial Interpolation

The polynomial classes discussed in the previous section all provide useful functions for polynomial interpolation. For instance, recall the linear equation for the polynomial interpolation problem: $\Phi(x)c=y$, where *x* and *y* are vectors containing the x_i and y_i data points, and *c* is the unknown coefficient vector. To solve the interpolation problem we need to first evaluate the matrix $\Phi(x)$ for a given basis, and then solve the resulting linear equation system. Each of the polynomial classes in polynomial conveniently provides a function for computing the (generalized) Vandermonde matrix for the corresponding basis. For example, for polynomials in the power basis, we can use `np.polynomial.polynomial.polyvander`, and for polynomials in the Chebyshev basis we can use the corresponding `np.polynomial.chebyshev.chebvander` function, and so on. See the docstrings for `np.polynomial` and its submodules for the complete list of generalized Vandermonde matrix functions for the various polynomial bases.

Using the above-mentioned functions for generating the Vandermonde matrices, we can easily perform a polynomial interpolation in different bases. For example, consider the data points (1, 1), (2, 3), (3, 5), and (4, 4). We begin with creating NumPy array for the *x* and *y* coordinates for the data points.

```
In [32]: x = np.array([1, 2, 3, 4])
In [33]: y = np.array([1, 3, 5, 4])
```

To interpolate a polynomial through these points, we need to use a polynomial of third degree (number of data points minus one). For interpolation in the power basis, we seek the coefficients c_i such that $f(x) = \sum_{i=1}^{4} c_i x^{i-1} = c_1 x^0 + c_2 x^1 + c_3 x^2 + c_4 x^3$, and to find these coefficients we evaluate the Vandermonde matrix and solve the interpolation equation system:

```
In [34]: deg = len(x) - 1
In [35]: A = P.polynomial.polyvander(x, deg)
In [36]: c = linalg.solve(A, y)
In [37]: c
Out[37]: array([ 2. , -3.5,  3. , -0.5])
```

The sought coefficient vector is [2, -3.5, 3, -0.5], and the interpolation polynomial is thus $f(x) = 2 - 3.5x + 3x^2 - 0.5x^3$. Given the coefficient array c, we can now create a polynomial representation that can be used for interpolation:

```
In [38]: f1 = P.Polynomial(c)
In [39]: f1(2.5)
Out[39]: 4.1875
```

To perform this polynomial interpolation in another polynomial basis, all that we need to change is the name of the function that was used to generate the Vandermonde matrix `A` in the previous example. For example, to interpolate using the Chebyshev basis polynomials, we can do this:

```
In [40]: A = P.chebyshev.chebvander(x, deg)
In [41]: c = linalg.solve(A, y)
In [42]: c
Out[42]: array([ 3.5  , -3.875,  1.5  , -0.125])
```

As expected, the coefficient array has different values in this basis, and the interpolation polynomial in the Chebyshev basis is $f(x) = 3.5T_0(x) - 3.875T_1(x) + 1.5T_2(x) - 0.125T_3(x)$. However, regardless of the polynomial basis, the interpolation polynomial is unique, and evaluating the interpolant will always result in the same values:

```
In [43]: f2 = P.Chebyshev(c)
In [44]: f2(2.5)
Out[44]: 4.1875
```

We can demonstrate that the interpolation with the two bases indeed results in the same interpolation function by plotting the `f1` and `f2` together with the data points (see Figure 7-1):

```
In [45]: xx = np.linspace(x.min(), x.max(), 100)  # supersampled [x[0], x[-1]] interval
In [45]: fig, ax = plt.subplots(1, 1, figsize=(12, 4))
    ...: ax.plot(xx, f1(xx), 'b', lw=2, label='Power basis interp.')
    ...: ax.plot(xx, f2(xx), 'r--', lw=2, label='Chebyshev basis interp.')
    ...: ax.scatter(x, y, label='data points')
    ...: ax.legend(loc=4)
    ...: ax.set_xticks(x)
    ...: ax.set_ylabel(r"$y$", fontsize=18)
    ...: ax.set_xlabel(r"$x$", fontsize=18)
```

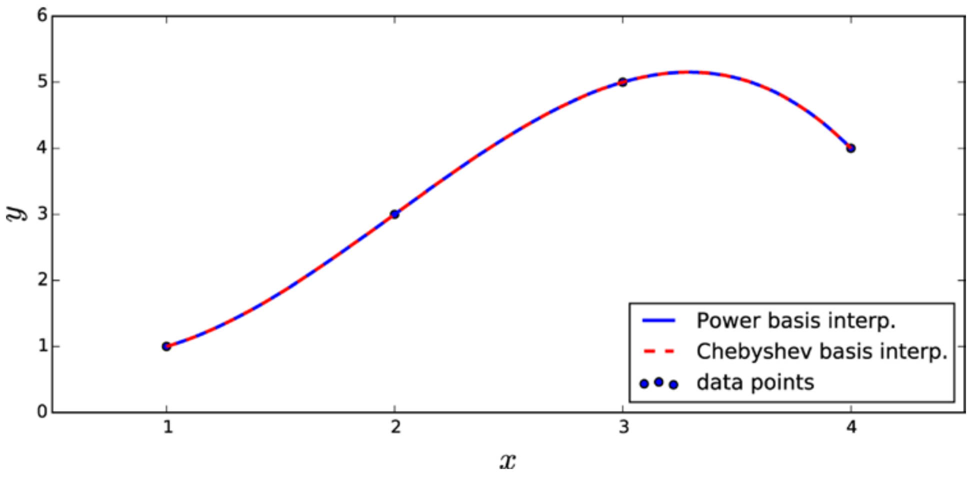

***Figure 7-1.** Polynomial interpolation of four data points, using power basis and the Chebyshev basis*

While interpolation with different polynomial bases is convenient due to the functions for the generalized Vandermonde matrices, there is an even simpler and better method available. Each polynomial class provides a class method `fit` that can be used to compute an interpolation polynomial.[1] The two interpolation functions that were computed manually in the previous example could therefore instead be computed in the following manner: Using the power basis, and its `Polynomial` class we obtain:

```
In [46]: f1b = P.Polynomial.fit(x, y, deg)
In [47]: f1b
Out[47]: Polynomial([ 4.1875,  3.1875, -1.6875, -1.6875], [ 1.,  4.], [-1.,  1.])
```

and by using the class method `fit` from the `Chebyshev` class instead, we obtain:

```
In [48]: f2b = P.Chebyshev.fit(x, y, deg)
In [49]: f2b
Out[49]: Chebyshev([ 3.34375 ,  1.921875, -0.84375 , -0.421875], [ 1.,  4.], [-1.,  1.])
```

[1]If the requested polynomial degree of the interpolant is smaller than the number of data points minus one, then a least square fit is computed rather than an exact interpolation.

Note that with this method, the `domain` attribute of the resulting instances are automatically set to the appropriate *x* values of the data points (in this example, the input range is [1, 4]), and the coefficients are adjusted accordingly. As mentioned previously, mapping the interpolation data into the range that is most suitable for a specific basis can significantly improve the numerical stability of the interpolation. For example, using the Chebyshev basis with *x* values that are scaled such that $x \in [-1, 1]$, rather than the original *x* values in the previous example, reduces the condition number from almost 4660 to about 1.85:

```
In [50]: np.linalg.cond(P.chebyshev.chebvander(x, deg))
Out[50]: 4659.7384241399586
In [51]: np.linalg.cond(P.chebyshev.chebvander((2*x-5)/3.0, deg))
Out[51]: 1.8542033440472896
```

Polynomial interpolation of a few data points is a powerful and useful mathematical tool, which is an important part of many mathematical methods. When the number of data points increase, we need to use increasingly high-order polynomials for exact interpolation, and this is problematic in several ways. To begin with, it becomes increasing demanding to both determine and evaluate the interpolant for increasing polynomial order. However, a more serious issue is that high-order polynomial interpolation can have undesirable behavior between the interpolation points. Although the interpolation is exact at the given data points, a high-order polynomial can vary wildly between the specified points. This is famously illustrated by polynomial interpolation of Runge's function $f(x) = 1/(1 + 25x^2)$ using evenly spaced sample points in the interval $[-1, 1]$. The result is an interpolant that nearly diverges between the data points near the end of the interval.

To illustrate this behavior, we create a Python function `runge` that implements Runge's function, and a function `runge_interpolate` that interpolates an `n`th order polynomial, in the power basis, to the Runge's function at evenly spaced sample points:

```
In [52]: def runge(x):
    ...:     return 1/(1 + 25 * x**2)
In [53]: def runge_interpolate(n):
    ...:     x = np.linspace(-1, 1, n)
    ...:     p = P.Polynomial.fit(x, runge(x), deg=n)
    ...:     return x, p
```

Next we plot Runge's function together with the 13th and 14th order polynomial interpolations, at supersampled *x* values in the $[-1, 1]$ interval. The resulting plot is shown in Figure 7-2.

```
In [54]: xx = np.linspace(-1, 1, 250)
In [55]: fig, ax = plt.subplots(1, 1, figsize=(8, 4))
    ...: ax.plot(xx, runge(xx), 'k', lw=2, label="Runge's function")
    ...: # 13th order interpolation of the Runge function
    ...: n = 13
    ...: x, p = runge_interpolate(n)
    ...: ax.plot(x, runge(x), 'ro')
    ...: ax.plot(xx, p(xx), 'r', label='interp. order %d' % n)
    ...: # 14th order interpolation of the Runge function
    ...: n = 14
    ...: x, p = runge_interpolate(n)
    ...: ax.plot(x, runge(x), 'go')
    ...: ax.plot(xx, p(xx), 'g', label='interp. order %d' % n)
    ...:
```

```
...: ax.legend(loc=8)
...: ax.set_xlim(-1.1, 1.1)
...: ax.set_ylim(-1, 2)
...: ax.set_xticks([-1, -0.5, 0, 0.5, 1])
...: ax.set_ylabel(r"$y$", fontsize=18)
...: ax.set_xlabel(r"$x$", fontsize=18)
```

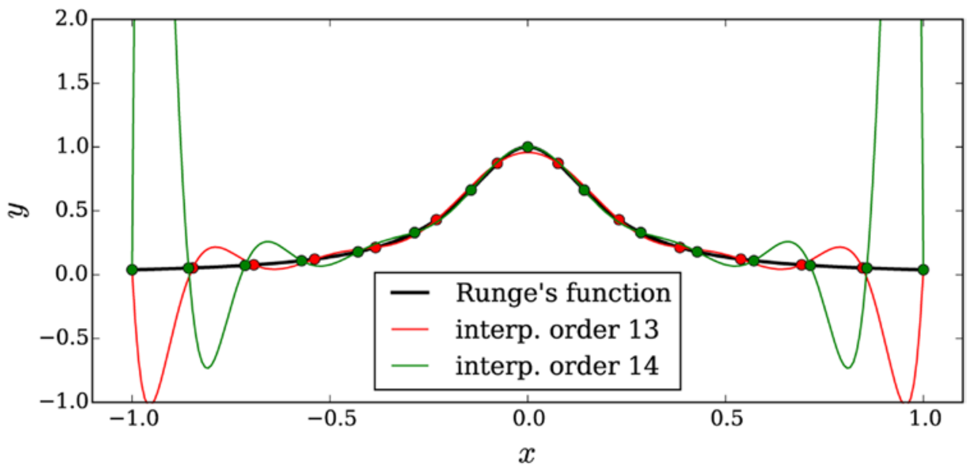

Figure 7-2. *The Runge function together with two high-order polynomial interpolations*

We note that in Figure 7-2, the interpolants exactly agree with Runge's function at the sample points, but between these points they oscillate wildly near the ends of the interval. This is an undesirable property of an interpolant, and it defeats the purpose of the interpolation. A solution to this problem is to use piecewise low-order polynomials when interpolating with large number of data points. In other words, instead of fitting all the data points to a single high-order polynomial, a different low-order polynomial is used to describe each subinterval bracketed by two consecutive data points. This is the topic of the following section.

Spline Interpolation

For a set of n data points $\{x_i, y_i\}$, there are $n-1$ subintervals $[x_i, x_{i+1}]$ in the full range of the data $[x_0, x_{n-1}]$. An interior data point that connects two such subintervals is known as a *knot* in the terminology of piecewise polynomial interpolation. To interpolate the n data points using piecewise polynomials of degree k on each of the subintervals, we must determine $(k+1)(n-1)$ unknown parameters. The values at the knots give $2(n-1)$ equations. These equations, by themselves, are only sufficient to determine a piecewise polynomial of order one, that is, a piecewise linear function. However, additional equations can be obtained by requiring also that derivatives and higher-order derivatives are continuous at the knots. This condition ensures that the resulting piecewise polynomial has a smooth appearance.

A spline is a special type of piecewise polynomial interpolant: a piecewise polynomial of degree k is a spline if it is continuously differentiable $k-1$ times. The most popular choice is the third-order spline, $k=3$, which requires $4(n-1)$ parameters. For this case, the continuity of two derivatives at the $n-2$ knots gives $2(n-2)$ additional equations, bringing the total number of equations to $2\,(n-1)+2(n-2)=4(n-1)-2$.

There are therefore two remaining undetermined parameters, which must be determined by other means. A common approach is to additionally require that the second order derivatives at the end points are zero (resulting in the *natural* spline). This gives two more equations, which closes the equation system.

The SciPy `interpolate` module provides several functions and classes for performing spline interpolation. For example, we can use the `interpolate.interp1d` function, which takes *x* and *y* arrays for the data points as first and second arguments. The optional keyword argument `kind` can be used to specify the type and order of the interpolation. In particular, we can set `kind=3` (or, equivalently, `kind='cubic'`) to compute the cubic spline. This function returns a class instance that can be called like a function, and which can be evaluated for different values of *x* using function calls. An alternative spline function is `interpolate.InterpolatedUnivariateSpline`, which also takes *x* and *y* arrays as first and second argument, but which uses the keyword argument `k` (instead of `kind`) to specify the order of the spline interpolation.

To see how to the `interpolate.interp1d` function can be used, consider again Runge's function, and we now want to interpolate this function with a third-order spline polynomial. To this end, we first create NumPy arrays for the *x* and *y* coordinates of the sample points. Next we call the `interpolate.interp1d` function with `kind=3` to obtain the third-order spline for the given data:

```
In [56]: x = np.linspace(-1, 1, 11)
In [57]: y = runge(x)
In [58]: f_i = interpolate.interp1d(x, y, kind=3)
```

To evaluate how good this spline interpolation is (here represented by the class instance `f_i`), we plot the interpolant together with the original Runge's function and the sample points. The result is shown in Figure 7-3.

```
In [59]: xx = np.linspace(-1, 1, 100)
In [60]: fig, ax = plt.subplots(figsize=(8, 4))
    ...: ax.plot(xx, runge(xx), 'k', lw=1, label="Runge's function")
    ...: ax.plot(x, y, 'ro', label='sample points')
    ...: ax.plot(xx, f_i(xx), 'r--', lw=2, label='spline order 3')
    ...: ax.legend()
    ...: ax.set_xticks([-1, -0.5, 0, 0.5, 1])
    ...: ax.set_ylabel(r"$y$", fontsize=18)
    ...: ax.set_xlabel(r"$x$", fontsize=18)
```

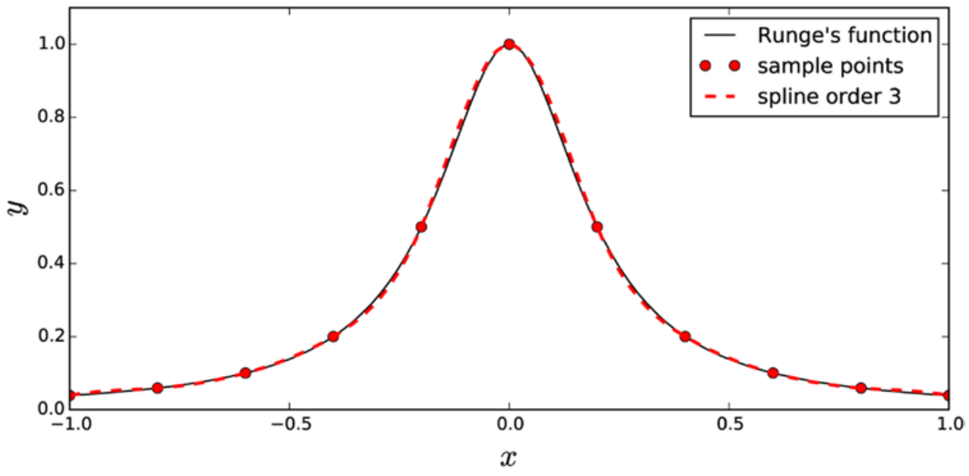

Figure 7-3. *Runge's function with a third-order Spline interpolation using 11 data points*

Here we used 11 data points and a spline of third order. We note that the interpolant agrees very well with the original function in Figure 7-3. Typically spline interpolation of order three or less does not suffer from the same type of oscillations that we observed with high-order polynomial interpolation, and normally it is sufficient to use splines of order three if we have a sufficient number of data points.

To illustrate the effect of the order of a spline interpolation, consider the problem of interpolating the data (0,3), (1, 4), (2, 3.5), (4, 2), (5, 1.5), (6, 1.25), and (7, 0.7) with splines of increasing order. We first define the x and y arrays, and then loop over the required spline orders, computing the interpolation and plotting it for each order:

```
In [61]: x= np.array([0, 1, 2, 3, 4, 5, 6, 7])
In [62]: y= np.array([3, 4, 3.5, 2, 1, 1.5, 1.25, 0.9])
In [63]: xx = np.linspace(x.min(), x.max(), 100)
In [64]: fig, ax = plt.subplots(figsize=(8, 4))
    ...: ax.scatter(x, y)
    ...:
    ...: for n in [1, 2, 3, 6]:
    ...:     f = interpolate.interp1d(x, y, kind=n)
    ...:     ax.plot(xx, f(xx), label='order %d' % n)
    ...:
    ...: ax.legend()
    ...: ax.set_ylabel(r"$y$", fontsize=18)
    ...: ax.set_xlabel(r"$x$", fontsize=18)
```

From the spline interpolation shown in Figure 7-4, it is clear that spline order two or three already provides a rather good interpolation, with relatively small errors between the original function and the interpolant function. For higher-order splines, the same problem as we saw for high-order polynomial interpolation resurfaces. In practice, it is therefore often suitable to use third-order spline interpolation.

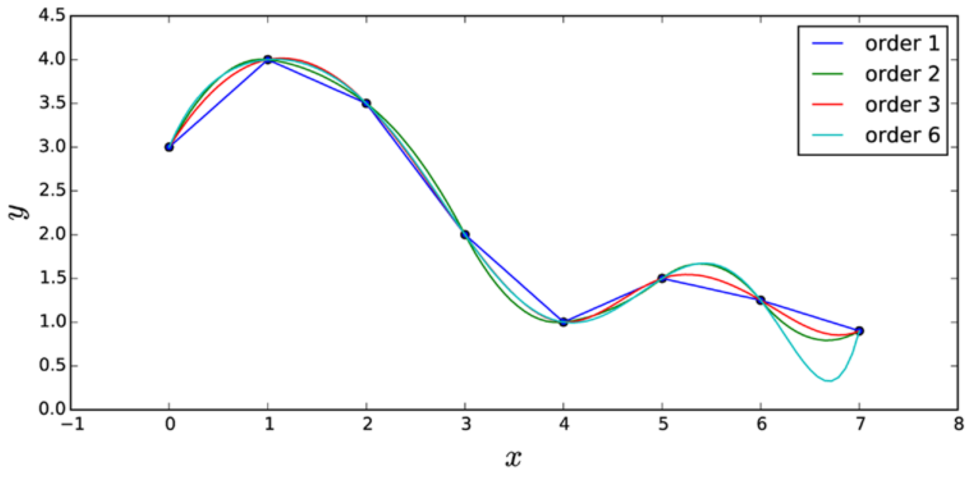

Figure 7-4. *Spline interpolations of different orders*

Multivariate Interpolation

Polynomial and spline interpolation can be straightforwardly generalized to multivariate situations. In analogy with the univariate case, we seek a function whose values are given at a set of specified points, and that can be evaluated for intermediary points within the sampled range. SciPy provides several functions and classes for multivariate interpolation, and in the following two examples we explore two of the most useful functions for bivariate interpolation: the `interpolate.interp2d` and `interpolate.griddata` functions, respectively. See the docstring for the `interpolate` module and its reference manual for further information on other interpolation options.

We begin by looking at `interpolate.interp2d`, which is a straightforward generalization of the `interp1d` function that we previously used. This function takes the *x* and *y* coordinates of the available data points as separate one-dimensional arrays, followed by a two-dimensional array of values for each combination of *x* and *y* coordinates. This presumes that the data points are given on a regular and uniform grid of *x* and *y* coordinates.

To illustrate how the `interp2d` function can be used, we simulate noisy measurements by adding random noise to a known function, which in the following example is taken to be $f(x,y)=\exp\left(-(x+1/2)^2-2(y+1/2)^2\right)-\exp\left(-(x-1/2)^2-2(y-1/2)^2\right)$. To form an interpolation problem, we sample this function at 10 points in the interval $[-2,2]$, along the *x* and *y* coordinates, and then add a small normal-distributed noise to the exact values. We first create NumPy arrays for the *x* and *y* coordinates of the sample points, and define a Python function for *f*(*x*, *y*):

```
In [65]: x = y = np.linspace(-2, 2, 10)
In [66]: def f(x, y):
    ...:     return np.exp(-(x + .5)**2 - 2*(y + .5)**2) - np.exp(-(x - .5)**2 - 2*(y - .5)**2)
```

Next we evaluate the function at the sample points and add the random noise to simulate uncertain measurements:

```
In [67]: X, Y = np.meshgrid(x, y)
In [68]: # simulate noisy data at fixed grid points X, Y
    ...: Z = f(X, Y) + 0.05 * np.random.randn(*X.shape)
```

At this point, we have a matrix of data points `Z` with noisy data, which is associated with exactly known and regularly spaced coordinates *x* and *y*. To obtain an interpolation function that can be evaluated for intermediary *x* and *y* values, within the sampled range, we can now use the `interp2d` function:

```
In [69]: f_i = interpolate.interp2d(x, y, Z, kind='cubic')
```

Note that here `x` and `y` are one-dimensional arrays (of length 10), and `Z` is a two-dimensional array of shape (10, 10). The `interp2d` function returns a class instance, here `f_i`, that behaves as a function that we can evaluate at arbitrary *x* and *y* coordinates (within the sampled range). A supersampling of the original data, using the interpolation function, can therefore be obtained in the following way:

```
In [70]: xx = yy = np.linspace(x.min(), x.max(), 100)
In [71]: ZZi = f_i(xx, yy)
In [72]: XX, YY = np.meshgrid(xx, yy)
```

Here, XX and YY are coordinate matrices for the supersampled points, and the corresponding interpolated values are ZZi. These can, for example, be used to plot a smoothed function describing the sparse and noisy data. The following code plots contours of both the original function and the interpolated data. See Figure 7-5 for the resulting contour plot.

```
In [73]: fig, axes = plt.subplots(1, 2, figsize=(12, 5))
    ...: # for reference, first plot the contours of the exact function
    ...: c = axes[0].contourf(XX, YY, f(XX, YY), 15, cmap=plt.cm.RdBu)
    ...: axes[0].set_xlabel(r"$x$", fontsize=20)
    ...: axes[0].set_ylabel(r"$y$", fontsize=20)
    ...: axes[0].set_title("exact / high sampling")
    ...: cb = fig.colorbar(c, ax=axes[0])
    ...: cb.set_label(r"$z$", fontsize=20)
    ...: # next, plot the contours of the supersampled interpolation of the noisy data
    ...: c = axes[1].contourf(XX, YY, ZZi, 15, cmap=plt.cm.RdBu)
    ...: axes[1].set_ylim(-2.1, 2.1)
    ...: axes[1].set_xlim(-2.1, 2.1)
    ...: axes[1].set_xlabel(r"$x$", fontsize=20)
    ...: axes[1].set_ylabel(r"$y$", fontsize=20)
    ...: axes[1].scatter(X, Y, marker='x', color='k')
    ...: axes[1].set_title("interpolation of noisy data / low sampling")
    ...: cb = fig.colorbar(c, ax=axes[1])
    ...: cb.set_label(r"$z$", fontsize=20)
```

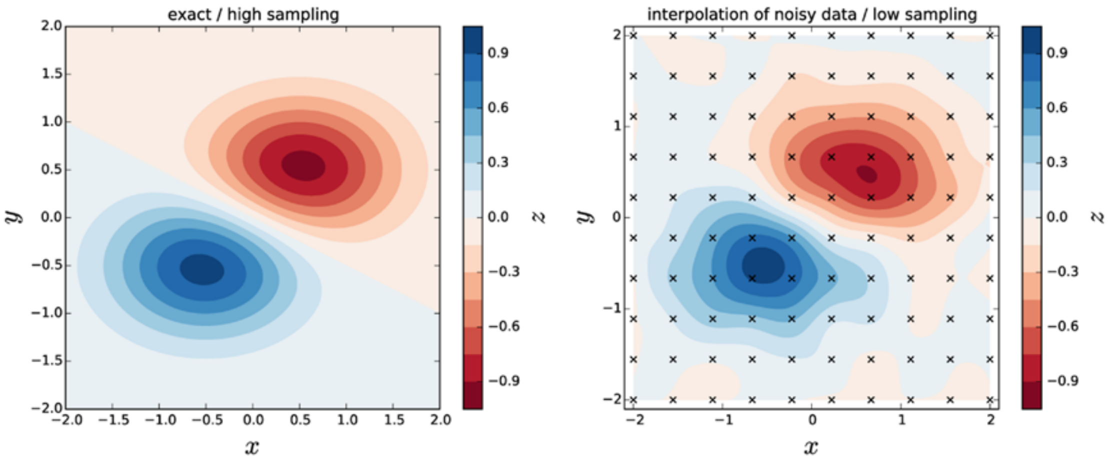

***Figure 7-5.** Contours of the exact function (left) and a bivariate cubic spline interpolation (right) of noisy samples form the function on a regular grid (marked with crosses)*

With relatively sparsely spaced data points, we can thus construct an approximation of the underlying function by using the `interpolate.interp2d` to compute the bivariate cubic spline interpolation. This gives a smoothed approximation for the underplaying function, which is frequently useful when dealing with data obtained from measurements or computations that are costly, in time or other resources. For higher-dimensional problems, there is a function `interpolate.interpnd`, which is a generalization to *n*-dimensional problems.

Another typical situation that requires multivariate interpolation occurs when sampled data is given on an irregular coordinate grid. This situation frequently arises (for example in experiments or other data collection processes) when the exact values at which the observations are collected cannot be directly controlled. To be able to easily plot and analyze such data with existing tools, it may be desirable to interpolate it onto a regular coordinate grid. In SciPy we can use the `interpolate.griddata` for exactly this task. This function takes as first argument a tuple of one-dimensional coordinate vectors (`xdata, ydata`) for the data values `zdata`, which are passed to the function in matrix form as the third argument. The fourth argument is a tuple (`X, Y`) of coordinate vectors or coordinate matrices for the new points at which the interpolant is to be evaluated. Optionally, we can also set the interpolation method using the `method` keyword argument (`'nearest'`, `'linear'`, or `'cubic'`):

```
Zi = interpolate.griddata((xdata, ydata), zdata, (X, Y), method='cubic')
```

To demonstrate how to use the `interpolate.griddata` function for interpolating data at unstructured coordinate points, we take the function $f(x,y) = \exp\left(-x^2 - y^2\right)\cos 4x \sin 6y$ and randomly select sampling points in the interval $\left[-1,1\right]$ along the x and y coordinates. The resulting $\{x_i, y_i, z_i\}$ data is then interpolated and evaluated on a supersampled regular grid spanning the $x, y \in \left[-1,1\right]$ region. To this end, we first define a Python function for $f(x, y)$ and then generate the randomly sampled data:

```
In [75]: def f(x, y):
    ...:     return np.exp(-x**2 - y**2) * np.cos(4*x) * np.sin(6*y)
In [76]: N = 500
In [77]: xdata = np.random.uniform(-1, 1, N)
In [78]: ydata = np.random.uniform(-1, 1, N)
In [79]: zdata = f(xdata, ydata)
```

To visualize the function and the density of the sampling points, we plot a scatter plot for the sampling locations overlaid on a contour graph of $f(x, y)$. The result is shown in Figure 7-6.

```
In [80]: x = y = np.linspace(-1, 1, 100)
In [81]: X, Y = np.meshgrid(x, y)
In [82]: Z = f(X, Y)
In [83]: fig, ax = plt.subplots(figsize=(8, 6))
    ...: c = ax.contourf(X, Y, Z, 15, cmap=plt.cm.RdBu);
    ...: ax.scatter(xdata, ydata, marker='.')
    ...: ax.set_ylim(-1,1)
    ...: ax.set_xlim(-1,1)
    ...: ax.set_xlabel(r"$x$", fontsize=20)
    ...: ax.set_ylabel(r"$y$", fontsize=20)
    ...: cb = fig.colorbar(c, ax=ax)
    ...: cb.set_label(r"$z$", fontsize=20)
```

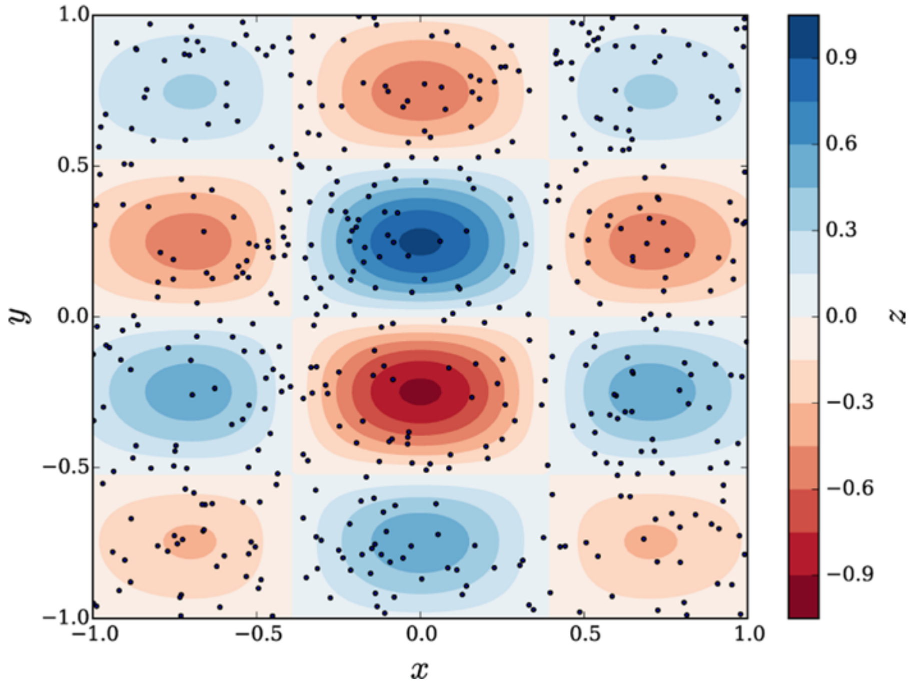

Figure 7-6. *Exact contour plot of a randomly sampled function. The 500 sample points are marked with black dots*

From the contour graph and scatter plots in Figure 7-6, it appears that the randomly chosen sample points cover the coordinate region of interest fairly well, and it is plausible that we should be able to reconstruct the function *f*(*x*, *y*) relatively accurately by interpolating the data. Here we would like to interpolate the data on the finely spaced (supersampled) regular grid described by the `X` and `Y` coordinates arrays. To compare different interpolation methods, and the effect of increasing number of sample points, we define the function `z_interpolate` that interpolates the given data points with the nearest data point, a linear interpolation, and a cubic spline interpolation:

```
In [84]: def z_interpolate(xdata, ydata, zdata):
    ...:     Zi_0 = interpolate.griddata((xdata, ydata), zdata, (X, Y), method='nearest')
    ...:     Zi_1 = interpolate.griddata((xdata, ydata), zdata, (X, Y), method='linear')
    ...:     Zi_3 = interpolate.griddata((xdata, ydata), zdata, (X, Y), method='cubic')
    ...:     return Zi_0, Zi_1, Zi_3
```

Finally we plot contour graph of the interpolated data for the three different interpolation methods applied to three subsets of the total number of sample points that use 50, 150, and all 500 points, respectively. The result is shown in Figure 7-7.

```
In [85]: fig, axes = plt.subplots(3, 3, figsize=(12, 12), sharex=True, sharey=True)
    ...:
    ...: n_vec = [50, 150, 500]
    ...: for idx, n in enumerate(n_vec):
    ...:     Zi_0, Zi_1, Zi_3 = z_interpolate(xdata[:n], ydata[:n], zdata[:n])
    ...:     axes[idx, 0].contourf(X, Y, Zi_0, 15, cmap=plt.cm.RdBu)
    ...:     axes[idx, 0].set_ylabel("%d data points\ny" % n, fontsize=16)
    ...:     axes[idx, 0].set_title("nearest", fontsize=16)
    ...:     axes[idx, 1].contourf(X, Y, Zi_1, 15, cmap=plt.cm.RdBu)
    ...:     axes[idx, 1].set_title("linaer", fontsize=16)
    ...:     axes[idx, 2].contourf(X, Y, Zi_3, 15, cmap=plt.cm.RdBu)
    ...:     axes[idx, 2].set_title("cubic", fontsize=16)
    ...: for m in range(len(n_vec)):
    ...:     axes[idx, m].set_xlabel("x", fontsize=16)
```

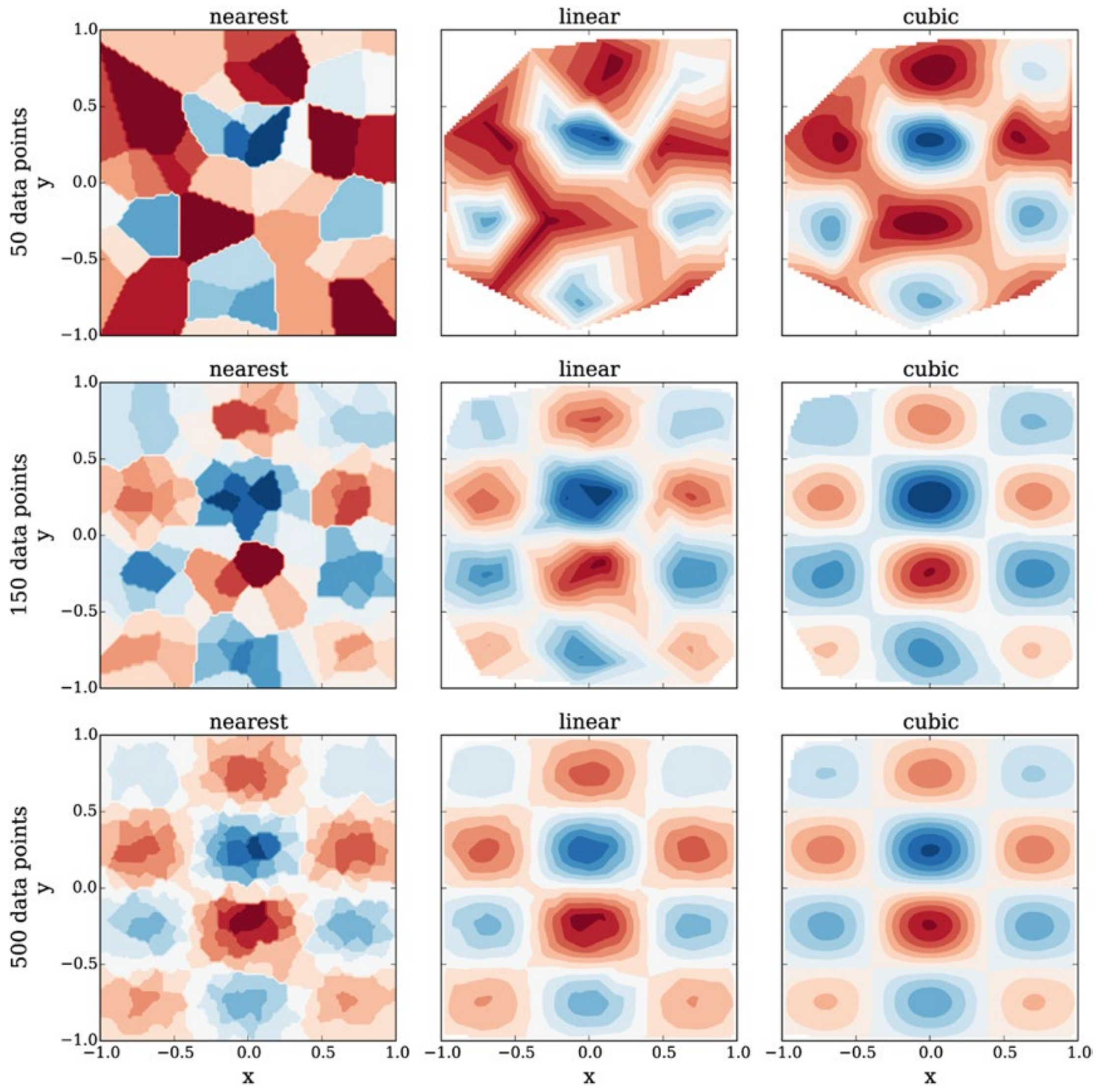

***Figure 7-7.** Bivariate interpolation of randomly sampled values, for increasing interpolation order (left to right) and increasing number of sample points (top to bottom)*

Figure 7-7 shows that it is possible reconstruct a function fairly well from interpolation of unstructured samples, as long as the region of interest is well covered. In this example, and quite generally for other situations as well, it is clear that the cubic spline interpolation is vastly superior to the nearest point and linear interpolation, and although it is more computationally demanding to compute the spline interpolation it is typically worthwhile.

Summary

Interpolation is a fundamental mathematical tool that has significant applications throughout scientific and technical computing. In particular, interpolation is a crucial part in many mathematical methods and algorithms. It is also a practical tool in itself, which is useful when plotting or analyzing data that is obtained from experiments, observations, or resource-demanding computations. The combination of the NumPy and SciPy libraries provides good coverage of numerical interpolation methods, in one or more independent variables. For most practical interpolation problems that involve a large number of data points, cubic spline interpolation is the most useful technique, although polynomial interpolation of low degree is most commonly used as a tool in other numerical methods (such as root finding, optimization, numerical integration). In this chapter we have explored how to use NumPy's `polynomial` and SciPy's `interpolate` modules to perform interpolation on given datasets with one and two independent variables. Mastering these techniques is an important skill of a computational scientist, and I strongly encourage further exploring the content in `scipy.interpolate` that was not covered here by studying the docstrings for this module and its many functions and classes.

Further Reading

Interpolation is covered in most texts on numerical methods. For a more thorough theoretical introduction to the subject, see, for example, the books by Hamming and Stoer.

References

Hamming, R. (1987). *Numerical Methods for Scientists and Engineers*. New York: Dover Publications.

Stoer, J., & Burlirsch, R. (1992). *Introduction to Numerical Analysis*. New York: Springer.