CHAPTER TEN

Interpolation

Synopsis

"Interpolation is the cornerstone on which other methods are based."
— Joel Ferziger

"Numerical computations with clever algorithms can yield exact result."

In this chapter, we show how to interpolate among the data points.

LAGRANGE INTERPOLATION

IN EXPERIMENTS AND simulations, we record values of a function at finite number of points. For further processing we construct a smooth function that passes through these points. A process of construction of such a function is called *interpolation*, which is illustrated below using several example.

1. For weather predictions, weather data (e.g., temperature, wind velocity) is recorded at discrete locations on the Earth. We employ interpolation to decipher the weather data at intermediate points on the Earth.
2. In television or computer games, GPUs (graphical processor units) interpolate the variables at various pixels of an image to reconstruct the full image.

In addition, interpolation forms a basis for many numerical algorithms: integration, differentiation, differential equation solver, etc. This observation will become evident when we discuss these methods in future.

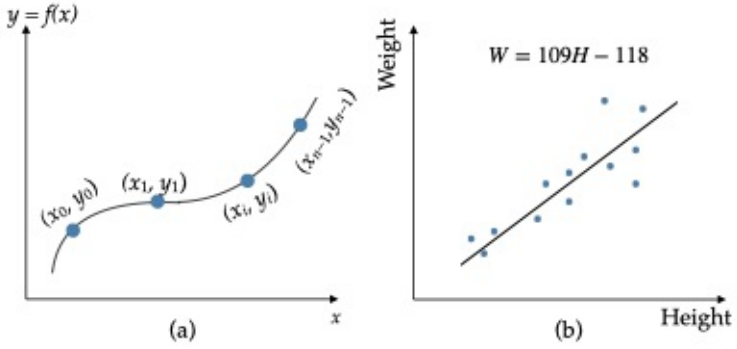

Figure 54: (a) An interpolation function $f(x)$ (black curve) passes through the data points, which are represented as blue dots. (b) The best-fit curve (black curve) passes through a set of data points (blue dots).

In Figure 54(a), we exhibit an extrapolated curve that passes through the data

points (x_i, y_i), which are called *knot*s. For a smooth interpolation, the uncertainties in the values of the data points need to minimal. However, when the data points have significant spread due to random errors, we employ *regression analysis* (to be discussed in Section Regression Analysis) to find a *best-fit curve* through the data points. See Figure 54(b) for an illustration.

There are many interpolation schemes—Lagrange interpolation, Hermite interpolation, divided difference, splines, etc. We start with one-dimensional (1D) Lagrange interpolation.

Lagrange Interpolation in 1D

Figure 54(a) Illustrates 1D interpolation through a set of data points. Lagrange constructed a $(n-1)$th order *polynomial* that passes through n data points. In the following discussion, we will discuss Lagrange's interpolation procedure.

It is easy to construct a linear interpolation function through two points, (x_0,y_0) and (x_1,y_1), of the function $f(x)$. Note that $f(x_0) = y_0$ and $f(x_1) = y_1$. In Figure 55(a), the dashed line is the linear interpolating curve.

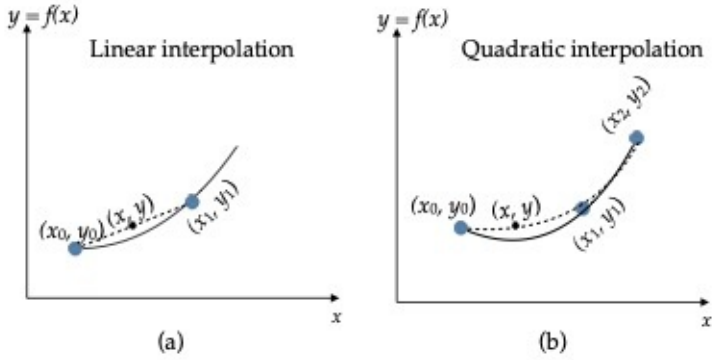

Figure 55: (a) Linear interpolation through (x_0,y_0) and (x_1,y_1). (b) Quadratic interpolation through (x_0,y_0), (x_1,y_1), and (x_2,y_2). The dashed lines are the interpolating curves, while the solid curve are actual $f(x)$.

It is straightforward to derive the following linear interpolating function $P_2(x)$ passing through the two points:

[redacted]

,

where $L_0(x)$ and $L_1(x)$ are linear functions of x, and they have the following properties: $L_0(x_0) = 1$, $L_0(x_1) = 0$, $L_1(x_0) = 0$, and $L_1(x_1) = 1$. Note that in general,

$P_2(x) \neq f(x)$. However, $P_2(x) = f(x)$ when $f(x)$ is a linear function of x.

Now we generalise the above discussion to n data points: (x_0,y_0), (x_1,y_1), …, (x_{n-1},y_{n-1}). Lagrange constructed the following interpolation function passing through these points:

$$P_n(x) = \sum_{j=0}^{n-1} L_j(x) y_j ,$$

where

$$L_j(x) = \prod_{i,i\neq j} \frac{(x - x_i)}{(x_j - x_i)} .$$

Note that $L_j(x_k) = \delta_{jk}$, and that the interpolation function $P_n(x)$ is a $(n-1)$th degree polynomial. For $n = 2$, $P_2(x)$ is the formula for the linear interpolation described above, while for $n = 3$, $P_3(x)$ is quadratic interpolating function of $f(x)$:

$$P_3(x) = \frac{x - x_1}{x_0 - x_1}\frac{x - x_2}{x_0 - x_2} y_0 + \frac{x - x_0}{x_1 - x_0}\frac{x - x_2}{x_1 - x_2} y_1 + \frac{x - x_0}{x_2 - x_0}\frac{x - x_1}{x_2 - x_1} y_2$$

$$= L_0(x) y_0 + L_1(x) y_1 + L_2(x) y_2 .$$

Note that $P_3(x) = f(x)$ if $f(x)$ is a polynomial of degree 2 or lower, and $P_3(x) \neq f(x)$ otherwise. See Figure 55(b) for an illustration.

Like any numerical computation, it is important to compute the error, $f(x) - P(x)$, for the Lagrange interpolation. Using reasonably complex arguments, it has been shown that the error for $P_n(x)$ is

$$E_n(x) = f(x) - P_n(x) = \frac{f^{(n)}(\zeta)}{n!}\prod(x - x_i), \quad \ldots(18)$$

where ζ is an intermediate point, and $f^{(n)}(\zeta)$ is the nth oder derivative of $f(x)$ at $x = \zeta$. The proof of the above statement is somewhat complex, hence it is detailed in Appendix A: Error in Lagrange Interopolation. If f(x) is a polynomial of degree (n−1), then $f^{(n)}(\zeta) = 0$, and hence f(x) = P_n(x). Thus, the Lagrange interpolation function P_n(x) is same f(x) for this case.

The derivation of $E_n(x)$ tells about the existence ζ, but its determination is quite complex. Hence, an accurate error estimation is quite difficult. In practical, we estimate bounds on $E_n(x)$ depending on the bounds of ζ.

The following Python function *Lagrange_interpolate*() evaluates $P_n(x)$ at a given x given dataset $\{(x_i,y_i)\}$. The arrays *xarray* and *yarray* contain $\{x_i\}$ and $\{y_i\}$ respectively.

```
# P(x) = Pi_i (x-x_i)/(x_j-x_i) y_i
# returns value at x
def Lagrange_interpolate(xarray,yarray,x):
    n = len(xarray)
    ans = 0
    for j in range(n):
        numr = 1; denr = 1;
        for i in range(n):
            if (j != i):
                numr *= (x-xarray[i])
                denr *= (xarray[j]-xarray[i])
        ans += (numr/denr)*yarray[j]
    return ans

Usage:

xarray = np.array([3,4])
yarray = np.array([1/3.0,1/4.0])
P2 = Lagrange_interpolate(xarray,yarray,x)
```

Example 1: Given $f(x) = 1/x$, we construct Lagrange interpolating polynomials using 2, 3, and 4 points between x = 2 to 5. Using the function

Lagrange_interpolate() and the two points (2,1/2) and (5,1/5) we construct a linear $P_2(x)$ illustrated in Figure 56(a). After this we employ two sets of points, {(2,1/2), (4,1/4), (5,1/5)} and {(2,1/2), (3,1/3), (5,1/5)}, to construct quadratic interpolating functions, which are illustrated in Figure 56(b,c) respectively. In Figure 56(b,c), the errors are significant between (2,4) and (3,5) respectively, which is due to the location of the intermediate points. At the end, we use 4 points {(2,1/2), (3,1/3), (4,1/4), (5,1/5)} to derive $P_4(x)$ illustrated in Figure 56(d).

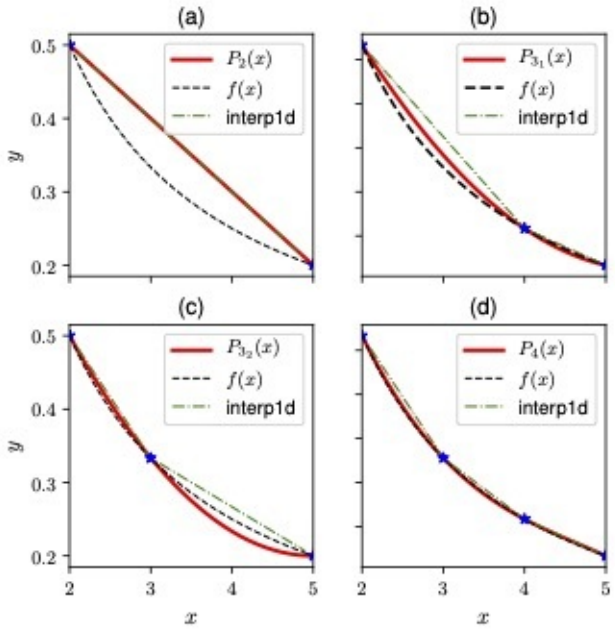

Figure 56: (a) Linear interpolation $P_2(x)$ using 2 points. (b,c) Quadratic interpolation $P_3(x)$ using three points. (d) Interpolation using 4 points ($P_4(x)$). The blue stars represent the data points, the actual function $f(x)$ is shown using black dashed lines, the Lagrange interpolation polynomials using red solid line, and interpolation function using Python's *interp1d* using green dashed-chained line.

Example 2: Error analysis of the data of Example 1: In Figure 57, we plot the errors $E_n(x) = f(x) - P_n(x)$ for n = 2, 3, 4, and 5. Note that $E(x) = 0$ at the knots. As expected, error decreases with the increase of n.

We estimate the upper bound on $E_2(x) = f(x) - P_2(x)$ as follows. Since

$E_2(x) = (½)\, f^{(2)}(\zeta)\, (x-2)(x-5)$.

$E_2(x)$ is maximum somewhere in the middle of the interval. The product $(x-2)(x-5)$ takes maximum value of 2.25 at $x = 3.5$. In addition, $\max(|f^{(2)}(\zeta)|) = 2/\zeta^3 = 1/4$ at $\zeta = 2$. Therefore,

$|E_2(x)| < (1/8)\text{x}2.25 = 0.28125.$

Thus, the error $|E_2(x)|$ is bounded within 0.2815. This estimate is consistent with the numerical values of $E_2(x)$, which is shown in Figure 57 as a red curve. We can perform similar error estimation for other polynomials as well.

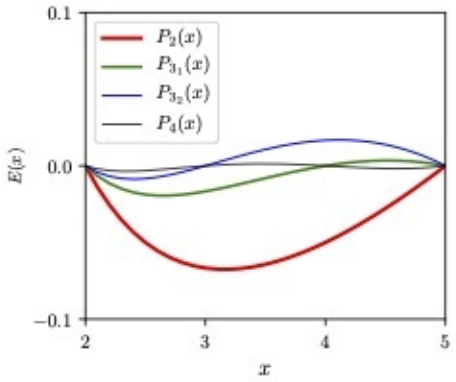

Figure 57: The plots of errors $E_n(x) = f(x) - P_n(x)$ for various Lagrange interpolation polynomials of Example 1.

In addition, we also compute error at a specific point, say at $x = 3.5$, which is the middle point in [2,5]. For the polynomials $P_2(x)$ to $P_4(x)$, the errors $E_n(3.5) = 1/3.5 - P_n(3.5)$ are −0.06429, −0.00804, 0.01071, and 0.00134 respectively. Here too, the error decreases with increase of n.

Lagrange Interpolation Using *Scipy*'s *Interp1d*

Python's *Scipy* module has an interpolating function named *interp1d*. This function makes a linear interpolation between consecutive points of the data. The following Python code illustrates how to use *interp1d* function for the same data as Example 1.

```
from scipy import interpolate

xarray = np.array([2,5])
yarray = np.array([1/2.0,1/5.0])
f = interpolate.interp1d(xarray,yarray)
# f contains the interpolation function.

xinter = np.arange(2.1,5,.1)
P2_scipy = f(xinter)
# P2_scipy is an array containing interpolated values for xinter
array.
```

In the above code sengment, *f* contains the interpolation function. We use *f* to generate interpolated values at *x* coordinates of *xinter* array. The interpolated values are stored in *P2_scipy* array. A point to note that the range of *xinter* must be within extreme abscissa values of the data.

In Figure 56, we exhibit the results of *interp1d*. Note that *interp1d* produces piece-wise linear functions.

Lagrange Interpolation in 2D

One-dimensional Lagrange interpolation descried earlier can be easily generalised to two dimensions (2D). Two-dimensional interpolation is very useful for image processing and other applications.

Assume a Cartesian 2D mesh with points as (x_i, y_i). Consider an arbitrary point (x,y) where we wish to interpolate the function f(x,y). We denote the interpolating function as P(x,y). See Figure 58 for an illustration:

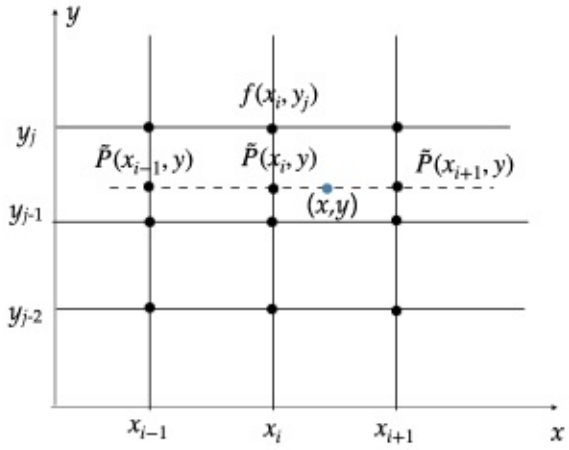

Figure 58: Interpolation of $f(x,y)$ in a 2D mesh.

We derive the 2D interpolating function in the following three steps:

First step: We estimate $f(x,y)$ using interpolation along the x axis. That is,

$$P(x,y) = \sum_i \prod_{i'.i' \neq i} \frac{(x - x_{i'})}{(x_i - x_{i'})} \tilde{P}(x_i, y)$$

Second step: We construct ~P(x,y) using interpolation along y:

$$\tilde{P}(x_i, y) = \sum_i \prod_{i' \, i' \neq i} \frac{(y - y_{j'})}{(y_j - y_{j'})} f(x_i, y_j) .$$

Third step: We substitute the above expression in $P(x,y)$ that yields the desired interpolation function:

.

$$\begin{aligned} P(x, y) &= \sum_i \prod_{i', i' \neq i} \frac{(x - x_{i'})}{(x_i - x_{i'})} \tilde{P}(x_i, y) \\ &= \sum_i \sum_j \prod_{i', i' \neq i} \frac{(x - x_{i'})}{(x_i - x_{i'})} \prod_{j', j' \neq j} \frac{(y - y_{j'})}{(y_j - y_{j'})} f(x_i, y_j) . \end{aligned}$$

Hence,

$$P(x, y) = \sum_i \sum_j L_{i,j}(x, y) y_{i,j} ,$$

where $y_{i,j}$ is shorthand for $f(x_i,y_j)$, and

$$L_{i,j} = \prod_{i', i' \neq i} \prod_{j', j' \neq j} \frac{(x - x_{i'})}{(x_i - x_{i'})} \frac{(y - y_{j'})}{(y_j - y_{j'})} .$$

Similarly, interpolation polynomials for 3D functions is

$$P(x, y, z) = \sum_i \sum_j L_{i,j,k}(x, y, z) y_{i,j,k} ,$$

where

$$L_{i,j,k} = \prod_{i',i' \neq i} \prod_{j',j' \neq j} \prod_{k',k' \neq k} \frac{(x - x_{i'})}{(x_i - x_{i'})} \frac{(y - y_{j'})}{(y_j - y_{j'})} \frac{(y - y_{k'})}{(y_k - y_{k'})}$$

The accuracy of Lagrange interpolation increases with more points. However, large number of data points need high-order interpolating polynomials that have spurious oscillations. Also, the computational cost for the construction of high-order polynomials may be exorbitant. Thus, we need to use optimum number of points for Lagrange interpolation. For a large number of data points, piece-wise interpolation may be employed.

There are several other polynomial-based interpolation schemes, such as *Hermite interpolation* and *Netwon's divided difference*. We will not discuss them here due to lack of space and time. Rather, we move to a different class of interpolation scheme called *splines*, which is topic of the next section.

****

Conceptual Questions

1. List scientific applications where interpolation is used.

Exercises

1. For $f(x) = \sin(x)$ in the domain $x = [0,\pi/2]$, construct Lagrange interpolating polynomials using 2, 3, and 4 points. Plot the interpolating functions, as well as the errors for them. Repeat the exercise when the domain is $x = [0,\pi]$.
2. For $f(x) = \exp(x)$ in the domain $x = [0,2]$, construct Lagrange interpolating polynomials using 2, 3, and 4 points. Analyse the errors associated with them.
3. Consider a 2D function $f(x,y) = \exp(x+y)$ in the domain $x = [0,1]$ and $y = [0,1]$. Construct Lagrange interpolating polynomials using 2, 3, and 4 points along each directions. Make contour plots for the interpolating polynomials along with the original function.

SPLINES

AS WE DISCUSSED in the previous section, Lagrange interpolation is not suitable for large number of data points. A way out is to employ piece-wise interpolation. However, such interpolating functions are not smooth at the nodes (x_i's). That is, the first derivatives at the nodes do not match from both sides. For example, the piece-wise linear function generated by *interp1d* has kinks at the nodes (see Figure 56). Splines, to be discussed in this section, help us smoothen the piece-wise interpolating functions.

In this section, we will describe *cubic splines* that are derived using the beam equation. The equation for the beam equation is

$$EI\frac{d^4}{dx^4}f(x) = F(x),$$

where E is the Young's modulus of the material, I is the second moment of beam's cross-section, and $F(x)$ is the applied force. In the spline framework, the force is assumed to be active at the nodes, denoted by (x_i, y_i) with i = 0:(n-1). In Figure 59, the curve represents a beam.

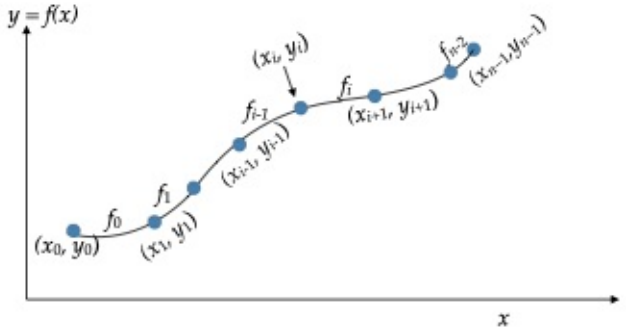

Figure 59: Illustration of a spline with nodes as (x_i, y_i) and the piece-wise functions as $f_i(x)$.

The equation for the beam on which the forces active at x_i is

$$EI\frac{d^4}{dx^4}f(x) = F_i\delta(x - x_i)\,. \qquad \ldots (19)$$

Note that F_i acts at x_i. An integration of the above equation yields $f'''(x)$ (triple derivative of $f(x)$) with discontinuities around the nodes. The second derivative exhibits a kink, but $f'(x)$ and $f(x)$ are smooth functions.

In Figure 60, we exhibit $f(x)$ and its derivatives when the RHS of Eq. (19) is $\delta(x-1)$ (forcing of unit amplitude applied at $x = 1$). For such a force, $f'''(x)$ exhibits a jump, $f''(x)$ is piece-wise linear with a kink, but $f'(x)$ and $f(x)$ are smooth functions.

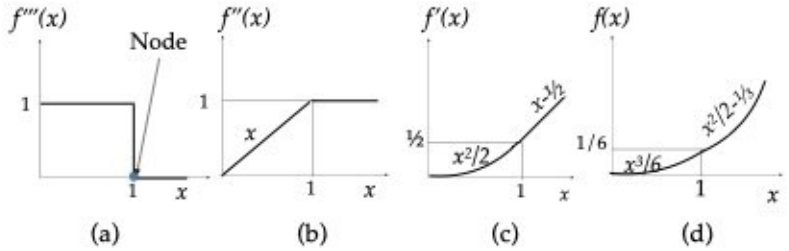

Figure 60: Function $f(x)$ and its derivatives for a spline with forcing function $\delta(x-1)$.

The construction of a spline requires finding a smooth $f(x)$ passing through the nodes (x_i, y_i). As shown in Figure 60, $f(x)$ and its derivatives satisfy the following properties:

1. The second derivative, $f''(x)$, is piecewise linear in each interval (x_i, x_{i+1}).
2. The function $f(x)$ and its first derivative are continuous and smooth everywhere, including at the nodes.

3. The function $f(x)$ passes through (x_i, y_i).

Following property (1), the linear interpolation of $f''(x)$ in the interval (x_i, x_{i+1}) yields

$$f_i''(x) = f''(x_i)\frac{x_{i+1} - x}{x_{i+1} - x_i} + f''(x_{i+1})\frac{x - x_i}{x_{i+1} - x_i}$$

Double integration of the above equation yields $f(x)$ in the interval (x_i, x_{i+1}) as

$$f_i(x) = f''(x_i)\frac{(x_{i+1} - x)^3}{6h_i} + f''(x_{i+1})\frac{(x - x_i)^3}{6h_i} + \left[\frac{y_i}{h_i} - \frac{h_i}{6}f''(x_i)\right](x_{i+1} - x) + \left[\frac{y_{i+1}}{h_i} - \frac{h_i}{6}f''(x_{i+1})\right](x - x_i)$$

where $h_i = x_{i+1} - x_i$. The two constants of integration have been determined using the conditions: $f_i(x_i) = y_i$ and $f_{i+1}(x_{i+1}) = y_{i+1}$. Similarly, for the interval (x_{i-1}, x_i), the function is

$$f_{i-1}(x) = f''(x_{i-1})\frac{(x_i - x)^3}{6h_{i-1}} + f''(x_i)\frac{(x - x_{i-1})^3}{6h_{i-1}} + \left[\frac{y_{i-1}}{h_{i-1}} - \frac{h_{i-1}}{6}f''(x_{i-1})\right](x_i - x) + \left[\frac{y_i}{h_{i+1}} - \frac{h_{i-1}}{6}f''(x_i)\right](x - x_{i-1})$$

Now we impose an additional condition that the first derivative at the nodes is continuous at both sides. Applying this condition at $x = x_i$, i.e., $f'_i(x_i) = f'_{i-1}(x_i)$, we obtain

$$-f''(x_i)\frac{h_i}{2} - \left[\frac{y_i}{h_i} - \frac{h_i}{6}f''(x_i)\right] + \left[\frac{y_{i+1}}{h_i} - \frac{h_i}{6}f''(x_{i+1})\right]$$
$$= -f''(x_i)\frac{h_{i-1}}{2} - \left[\frac{y_{i-1}}{h_{i-1}} - \frac{h_{i-1}}{6}f''(x_{i-1})\right] + \left[\frac{y_i}{h_{i-1}} - \frac{h_{i-1}}{6}f''(x_i)\right]$$

which yields

$$\frac{h_{i-1}}{6}f''(x_{i-1}) + \frac{1}{3}(h_i + h_{i-1})f''(x_i) + \frac{h_i}{6}f''(x_{i+1})$$
$$= \frac{y_{i+1}}{h_i} - y_i\left(\frac{1}{h_i} + \frac{1}{h_{i-1}}\right) + \frac{y_{i-1}}{h_{i-1}}$$

We obtain $(n-2)$ linear equations for nodes at $i = 1\ldots(n-2)$. However we have n unknowns [$f''(x_i)$ for $i = 0\ldots(n-1)$]. To solve this problem, we use one of the following boundary conditions:

1. *Parabolic run-out*: We assume that $f''(x)$ are constant on both end intervals, i.e., $f''(x_0) = f''(x_1)$ and $f''(x_{n-1}) = f''(x_{n-2})$. Hence $f(x)$ is quadratic in these intervals. With these two additional constraints, we have n equations to determine the n $f''(x_i)$'s.
2. *Free end*: We assume that $f'' = 0$ at both the ends, or $f''(x_0) = f''(x_{n-1}) = 0$.
3. *Cantilever end*: A intermediate condition between the cases 1 and 2, i.e., $f''(x_0) = \lambda f''(x_0)$ and $f''(x_{n-1}) = \lambda f''(x_{n-2})$ with $0 \le \lambda \le 1$.
4. *Periodic spline*: We assume that the data is periodic with y_0 identified with y_{n-1}, I.e., $y_0 = y_{n-1}$. With this, we have $(n-1)$ matching conditions at $(n-1)$ points to determine $(n-1)$ $f''(x_i)$.

With one of the above four boundary conditions, we have equal number of unknowns and linear equations. These equations can be solved using matrix methods that will be discussed in Section Solution of Algebraic Equations. In the

following example, we take a simple case with four points that can be solved analytically.

Example 1: We work out the splines for the data of Example 1 discussed in Section Lagrange Interpolation. We consider the same four points {(2,1/2), (3,1/3), (4,1/4) (5,1/5)} as before. Let us use the free-end boundary condition (f_0''= f_3''=0) for this example. Note that h_i = 1. The equations for the splines at the intermediate nodes are

$$\frac{2}{3}f_1'' + \frac{1}{6}f_2'' = y_2 - 2y_1 + y_0 = \frac{1}{4} - \frac{2}{3} + \frac{1}{2} = \frac{1}{12}$$
$$\frac{1}{6}f_1'' + \frac{2}{3}f_2'' = y_3 - 2y_2 + y_1 = \frac{1}{5} - \frac{2}{4} + \frac{1}{3} = \frac{1}{30}$$

The solution of the above equations are f_1'' = 3/25 and f_2'' = 1/50. Using these values and $f''_0 = f''_3 = 0$, we construct the piece-wise functions and plot them together. The plot is shown in Figure 61 as a black dashed curve. The code for computation of $f(x)$ is given below.

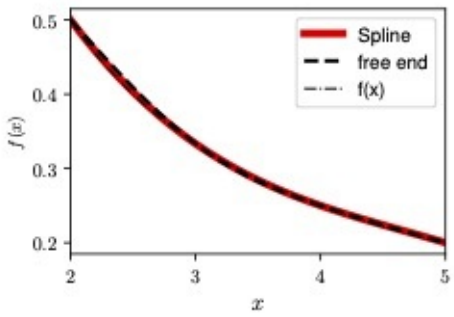

Figure 61: For the data {(2,1/2) (3,1/3), (4,1/4) (5,1/5)}, the plots of splines computed using free-end boundary condition (black dashed curve) and Scipy's *CubicSpline* (solid red curve). These curves are close to each other, and to the actual function 1/x. See Figure 62 for errors.

The following Python code computes the splines using free-end boundary

condition (as described above) and using Scipy's *interpolate.CubicSpline* function. The function generated using *interpolate.CubicSpline* function is shown as red solid curve in Figure 61.

```
### x-y data
xd = np.array([2,3,4,5])
yd = np.array([1/2.0,1/3.0,1/4.0,1/5.0])
### double derivatives
fdd = np.array([0,3/25,1/50,0])

def f(i,x):
    first_term = (fdd[i]*(xd[i+1]-x)**3 + fdd[i+1]*(x-xd[i])**3)/6
    second_term = (yd[i]-fdd[i]/6)*(xd[i+1]-x) + (yd[i+1]-
fdd[i+1]/6)*(x-xd[i])
    return first_term + second_term

# range of x
x0r = np.arange(2,3.1,0.1)
x1r = np.arange(3,4.1,0.1)
x2r = np.arange(4,5.1,0.1)
y0r = f(0,x0r);
y1r = f(1,x1r);
y2r = f(2,x2r);

xtot = np.concatenate((x0r, x1r, x2r))
ytot = np.concatenate((y0r, y1r, y2r))

# Using splines
cs=interpolate.CubicSpline(xd,yd)
x=np.arange(2,5.1,0.1)
y = cs(x)
```

Even though the splines generated by free-end boundary condition and Scipy functions are quite close to the actual function, $1/x$, there are small errors ($E(x) = f(x) - P(x)$), which are plotted in Figure 62. The black-dashed and solid-red curves represent the errors for the two cases. The accuracy of Lagrange polynomial and splines computed using *interpolate.CubicSpline* are comparable. However, the errors of splines computed using free-end boundary condition is quite significant. This is because $f''(x_0) = 1/4$, not zero, as is assumed in free-end boundary condition. The error at the other end, $x = 5$, is less damaging because $f''(x_3) = 2/25 \approx 0$.

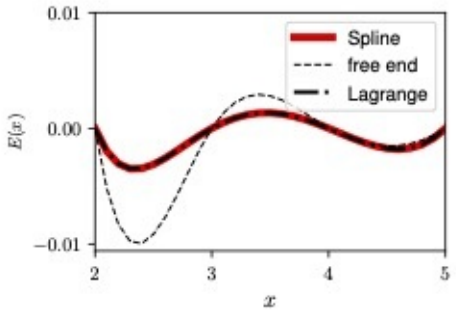

Figure 62: The plots of error $E(x) = f(x) - P(x)$ for the splines calculated using free-end boundary condition (black dashed curve), using *CubicSpline* (solid red curve), and using Lagrange polynomial with 4 points (solid black curve).

B-Splines is another prominent spline, but it is not covered in this book. For B-splines, you can use Scipy's function *interpolate.splrep*() to compute the spline parameters (*tck*), and then compute the curve by supplying *x* array to *interpolate.splev(x, tck)*.

In summary, splines provide *smooth* interpolating functions for a large number of points. However, these computations require solution of a matrix equation. Fortunately, the matrix is tridiagonal that can be easily solved using methods that will be described in Section Solution of Algebraic Equations.

In this chapter, we have assumed that the sampling points (x_i, y_i) have zero noise. In the presence of random noise, regression analysis are used to determine the underlying function $f(x)$. We will discuss regression in Section Regression Analysis.

****

Conceptual Questions

1. In what ways spline interpolation is better than Lagrange interpolation?

Exercises

1. Rework Example 1 with parabolic run-out boundary condition.
2. Consider $f(x) = \sin(x)$ for the domain $x = [0,2\pi]$. Construct periodic splines using appropriate number of points. After this, rework the splines with parabolic run-out boundary condition.
3. For $f(x) = \exp(x)$ in the domain $x = [0,2]$, construct splines with free-end boundary condition. Assume appropriate number of points.